Friedrich-Alexander-Universität Erlangen-Nürnberg
Institut für Informatik
Lehrstuhl für Datenbanksysteme

Entwurf und Realisierung eines Komponenten-Teststands zur Durchführung von Messungen an Enterprise JavaBeans

Studienarbeit im Fach Informatik
vorgelegt von
Christoph Peter Neumann
geb. am 04.05.1980 in Emmendingen

Revidierte Fassung
Entkoppelte Publikationen: [MN04]

Prüfer:	Univ.-Prof. Dr.-Ing. habil. Klaus Meyer-Wegener
Betreuer:	Dipl.-Inf. Marcus Meyerhöfer
Beginn der Arbeit:	01.07.2003
Abgabe der Arbeit:	29.02.2004

Zusammenfassung

Diese Studienarbeit beschreibt den Entwurf und die Implementierung eines Teststands für Komponenten, die der Enterprise JavaBean Spezifikation entsprechen, und deren Leistungsmessung. Die Messungen dienen der Gewinnung von Daten, die es später erlauben sollen eine Spezifikation der Dienstgüte, die von der Komponente erbracht werden kann, zu erstellen oder zu verifizieren.

Die Ergebnisse einer Literaturrecherche hinsichtlich generellen Verfahrensweisen zum Messen und existierenden Implementierungen von Messtechniken werden dargelegt, und dahingehend untersucht, ob sie auf Komponenten wie Enterprise JavaBeans anwendbar sind. Um die verschiedenen Techniken zu vergleichen wird ein Klassifikationssystem eingeführt, anhand dessen die Techniken diskutiert werden.

Es wird beschrieben werden wie das „Interceptor“ Entwurfsmuster angewandt werden kann, um Messungen der Antwortzeit von Komponenten im frei verfügbaren Applikationsserver JBoss durchzuführen. Außerdem wird ein Algorithmus dargelegt, der es erlaubt Aufrufabhängigkeiten zwischen interagierenden Komponenten abzuleiten, weil es in einem solchen Verband von Komponenten nicht nur auf die Leistung der einzelnen Komponenten ankommt, sondern auch auf die Aufrufhäufigkeiten und Abhängigkeiten zwischen den Komponenten.

Um die Belastung des Systems durch eine Komponente hinsichtlich Verbrauch von CPU-Zeit und Hauptspeicher zu ermitteln sind Interceptoren nicht geeignet. Zu diesem Zweck werden die Techniken JVMPI und Bytecode Instrumentierung untersucht, wobei schließlich ein JVMPI-basierter Ansatz konzipiert wird, der auf bereits publizierten Arbeiten aufsetzt und diese erweitern wird.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 29. Februar 2004

(Unterschrift)

Design and Realization of a "Component Test Stand" for Measurements on Enterprise JavaBeans

Abstract

This thesis introduces the concept of a performance measurement environment for components following the Enterprise JavaBeans (EJB) specification, aiming at the derivation of data that will allow to create and to verify a quality of service specification for a component.

The results of a literature research will be given for measurement techniques, approaches and implementations that are applicable in the context of EJBs, and a classification system will be presented to compare and discuss the different approaches.

It will be shown how the interceptor pattern can be applied for response time measurements in the open-source application server JBoss. An algorithm is presented to derive call dependencies between several interacting EJBs, because in such a component assembly not only the performance of an individual component is actually what matters, but also the invocation frequentness and dependencies.

For the measurement of resource consumption due to components the interceptor approach is not feasible. Therefore JVMPI and bytecode instrumentation will be evaluated. Finally, the design of a JVMPI based approach is described, that adopts and enhances the work of other publications.

Contents

List of Figures

1. Introduction

The concept of component based software development has gained much attention in the last years as mature component models like *Enterprise JavaBeans* (EJB) [Sun01a] or *CORBA Component Model* (CCM) [Obj02a] have evolved. These systems ease the development of large-scale, distributed applications by offering services like transparent access to remote components, transactions or security, thereby enabling the developer to concentrate on the core business logic. In this thesis a component corresponds to the EJB model and its definitions of a software component.

The motivation of our work has its origin in the COMQUAD[1] project: The vision is that components carry a *quality of service* (QoS) specification that describes under which preconditions a certain level of QoS can be delivered by an individual component. For example, a video codec component can guarantee an output of 25 fps as long as the input-stream of the supplying components fulfills several bandwidth and jitter conditions. Or another component, part of a stock information system, guarantees the delivery of stock quotations change notifications with a delay under 50 ms under the precondition that its supplier have appropriate response times. Ideally, there exist several variations of the QoS level, so that a flexible adaption to available resources can be done. Naturally, there will be many competing services, thus a *contract* has to be negotiated that binds the participating components to their QoS promises.

One part of the COMQUAD project develops a real-time capable component model for a new kind of component container, that will be based on a real-time environment like DROPS ("Dresden Real-time OPerating System") [DRO]. Another part of the

[1] "COMponents with QUantitative properties and ADaptivity" is a project funded by the German Research Council. It started October 1, 2001, at Dresden University of Technology and Friedrich-Alexander University of Erlangen and Nuremberg.

project focuses on the specification of *non-functional properties* (NFP) [MM04] of software components, including performance and QoS properties, as well as security properties. Also, the derivation and the specification scheme of dependencies between the component's NFPs and the NFPs of the used services/components that are needed for providing its service will be developed. As specification language an extension of CQML [Aag01] is used, called CQML+ [RZ03a, ZM03].

This thesis started when Comquad still focused on JBoss as J2EE application server, before the new component model and container were initiated. Thus, this thesis is not yet concerned with the emerging component model, and not with the specifications of NFPs either. Rather, the EJB component model and the JBoss server-implementation were given as premises. The testing environment has to provide the design for a facility that tests the following run-time performance properties of EJBs: response time, CPU and memory usage as well as call dependencies between the EJBs. The facility aims at extensibility and the future inclusion of more properties.

The environment will provide EJB developers with assistance in specifying the QoS characteristics of their components. For EJBs in use it will allow the verification of the EJBs run-time compliance to its specification, and could provide monitoring information to prevent denial-of-service (DoS) attacks or support for a load-balancer.

To the Comquad project we contribute the overview of component-related measurement approaches and implicitly provide requirements for the new container for integrated measurement and accounting support.

Generally speaking, there exist only few publications that regard performance and resource consumption measurement according to EJBs. And there are even less that aim at interdependencies between EJBs, as well as interdependencies due to the parameters of EJB method invocations.

The remainder of this thesis is organized as follows: Section 2 describes the runtime environment of the open-source application server JBoss as the technical foundation of our framework. Further, it gives introductions to Java profiling and time measurement. Section 3 presents a basic classification system that will be applied to the various measurement approaches. Section 4 lists related work, trying to give an inspiring overview over currently available publications that can be related to performance measurement of EJBs. Section 5 presents the designed solutions, concerning the measurement of response time, determination of call dependencies as well as CPU and memory profiling. Section 6 concludes with an outlook on further work and the summary of the thesis.

We use some special formatting in the text: Filenames and commands are written `sans-serif` as well as Java, XML and SQL code. At first occurrence of a fixed term – either introduced by the specifications, the vendor, a publication or by ourselves – the term is *emphasized* in order to indicate its atomicity. The terms section, figure, table and listing are abbreviated if they do not begin a sentence.

2. Preliminaries

2.1. Background and Environment

The following section will provide a basic knowledge about EJBs and their run-time environment as well as an overview over the open-source application server JBoss and its interceptor concept. For a comprehensive and freely downloadable book about EJBs refer to "Mastering Enterprise JavaBeans" written by Roman et al. [RAJ02]. In general, we require that the common EJB model is known to the reader. We provide only a brief introduction to EJBs and we will rather go into some detail with the JBoss specifics.

Two further sections will give an introduction to Java profiling techniques as well as a discussion of process- and thread-related time measurement.

2.1.1. Component Model

The J2EE 1.3 platform specifications [Sun01b] and the EJB 2.0 specifications [Sun01a] serve as component model of our environment. The common EJB component implements an interface that provides remote method invocation and control of its life-cycle. The run-time object of the component is called *EJB Instance* and is not visible to the client, but accessible via two delegating classes: the *Home Object* for life-cycle control, and the *EJB Object* for invocations of business methods.

The EJB component is not viable without a special runtime environment: any EJB has to be *deployed* to an EJB container, that means to install the bean so that it can be remotely invoked – the exact mechanism of deployment depends on the specific server implementation. The container runs on a *Java Virtual Machine* (JVM) and provides middleware services to the EJBs. For example, these services are transactional support, persistence, caching, logging, security mechanisms and registration to a central component registry.

The registry is the first contact point for client applications and is accessible via the *Java Naming and Directory Interface* (JNDI). Each EJB has at least one XML-based document for metadata information called *deployment descriptor*[1]. For the actual J2EE server, to which the EJB will be deployed to, a *vendor-specific deployment descriptor*[2] has to exist.

The JNDI name is defined inside the vendor-specific deployment descriptor. The administrator of the container can adjust the JNDI name by editing the deployment descriptor. He will also have to announce the correct JNDI name to all clients, users or developers, that need access to the component. The client application uses the JNDI name to receive a stub as a remote reference to the Home Object of the EJB. The life-cycle API of the Home Object is used to gain access to one or more EJB Instances. As already mentioned above, the EJB Instance is always represented to the client application by its remote EJB Object. The basic application flow between client and EJB(-container) is shown in Fig. 2.1.

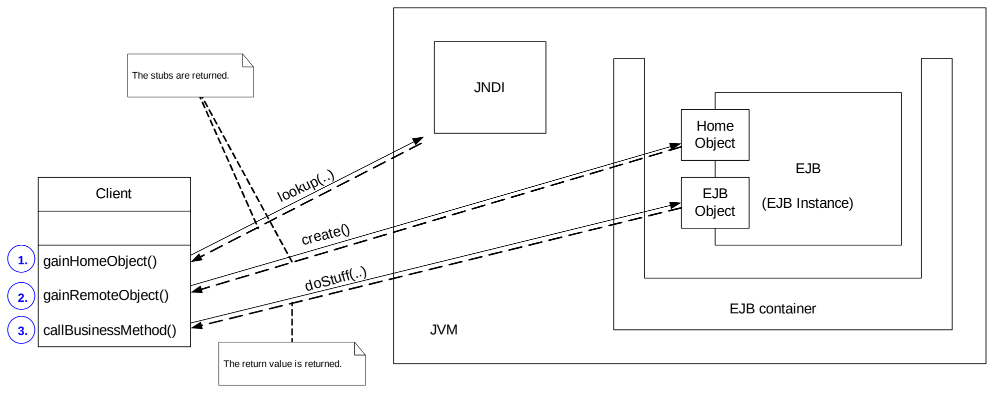


Figure 2.1.: Basic J2EE scenario

[1]The (server-independent) deployment descriptor is stored as META-INF/ejb-jar.xml inside the EJB's jar-file. For a description of the structure and the semantics refer to Roman et al. [RAJ02, 545–568].

[2]The vendor-specific deployment descriptors are also stored in the META-INF directory, with a filename and structure specific to the application server. For example META-INF/jboss.xml for the JBoss server.

The EJB specification distinguishes several types of EJBs, e.g. session beans, entity beans and message-driven beans [Sun01a, 47ff.]. It is possible to provide an EJB component conforming only or additionally to a “local interface” rather than to a remote one. These local EJB Objects can only be invoked inside the EJB container’s JVM [Sun01a, 52ff.], for example by other EJBs. This is an efficiency issue because the cost-intensive marshalling[3] can be omitted for local invocations. The container does not decide dynamically whether to use local or remote interface – this decision has to be taken by the developer.

The communication between client and EJB/server is based on the *Remote Method Invocation* (RMI), that describes the API and mechanism of a Java-based remote procedure call. But there exist two RMI variants:

(Native) RMI was developed before the J2EE specification and uses the *Java Remote Method Protocol* (JRMP). The “native” prefix was added when the other variant (see below) was developed, and denotes that it is the primordial version in Java. The native RMI implementation is available as java.rmi.* in J2SE.

In contrast to native RMI, the *RMI-IIOP* is, first of all, an API that allows RMI-style programming independent of the underlying protocol. This API is available as javax.rmi.* in J2SE. The IIOP suffix refers to the *Internet Inter-ORB Protocol* [Obj02a] of CORBA [Obj02b], in conjunction with its stub/skeleton approach. But the RMI-IIOP API does only guarantee the possibility to support IIOP, and a general compliance to CORBA. It does not mean that the IIOP protocol is actually used: The J2EE specification requires to use the RMI-IIOP API when accessing EJB components [Sun01b, 8], however a J2EE server may also use other protocols like JRMP as default of the underlying protocol of the RMI-IIOP API.

We will see in Sect. 2.1.2 that our run-time environment does not use primitive stubs and skeletons, but more complex and intelligent *dynamic proxies*. One major task of the Home and EJB Objects, which are generated by the server according to the interface specification, is the marshalling and unmarshalling of the invocation and its parameters according to RMI-IIOP.

[3]The exact necessity and semantics of marshalling within the scope of J2EE is described by Roman et al. [RAJ02, 491–505], for example in respect of call-by-reference vs. call-by-value (pass-by-copy) semantics.

2.1.2. Run-time Software Environment

Our run-time environment consists of the application server JBoss [JBo] running on a Sun JVM 1.4.2 [JVM]. JBoss is open-source software and the release version 3.2.1 fully supports the EJB component model. The most important part of JBoss in our context is the EJB container module. For a better understanding of our solution, it is necessary to give an introduction to the *Java Management eXtensions* (JMX) [Sun02] and to the JBoss interceptor concept [SJ03, 192f.].

JMX introduces *Manageable Beans* (MBeans) which represent an abstraction of the management side of any resource. Note that "beans" has the general meaning of component, and abstracts from special component models like EJBs, JavaBeans and MBeans. An MBean exports a management interface for an arbitrary application. The exported interface consists of getter and setter methods for attributes or arbitrary public functions. This interface can be used to automatically generate a management front-end for the MBeans, managing the underlying application. For example, a simple HTML-based front-end is created by JBoss for all deployed MBeans.

Every MBean has a string-based and therefore human-readable identifier. Any run-time object that knows this identifier can access the MBean's exposed interface by the JMX bus. The invocation through the JMX bus is also string-based, so it is not very fast, but it is possible to have an MBean expose a JMX-accessible method that returns its own direct reference, allowing full access to all `public` methods. Notice that JMX was not capable of remote invocations until on October 23rd 2003 Sun released the new JMX Remote API Specification [Sun03d]. JBoss has not yet adopted to this.

JBoss is based on JMX, which allows the implementation of a modular J2EE architecture with JMX as microkernel and the MBeans representing modules. JBoss uses this approach to have a minimal core independent from the modules that implement the numerous J2EE specifications, in order to ensure extensibility. The core EJB container and JNDI from Fig. 2.1 are such modules connected by JMX, as is shown by official architecture picture of JBoss in Fig. 2.2.

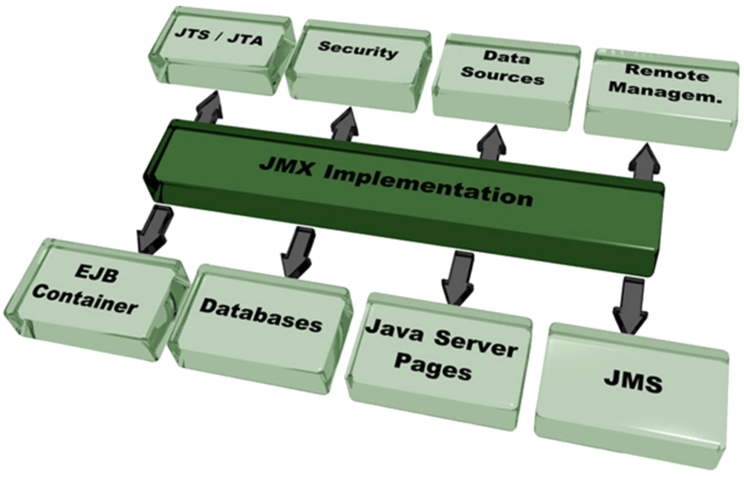


Figure 2.2.: Official JMX based architecture of JBoss [JBo01]

In contrast to the CORBA stubs and skeletons JBoss uses more complex *dynamic proxies*. The basic functionality of stubs and skeletons is the marshalling and unmarshalling of the invocation information, the parameters and the result object according to the IIOP serialization. There exist different stubs and skeletons for every remote object, whereby Home Object and EJB Object for one EJB Instance would represent different remote objects.

A dynamic proxy is an object that uses Java reflection to implement numerous interfaces at run-time. That way it implements either the Home Interface or the EJB Interface and allows the integration of interceptors at client-side (interceptors are discussed in the next section). For the purpose of this thesis it is sufficient to think of dynamic proxies as stubs with integrated interceptors, but a thorough description of the interaction and implementation of dynamic proxies is given by Jenny Liu [Liu02], as part of an analysis of the whole JBoss architecture.

JBoss uses JRMP as protocol of the RMI-IIOP API by default (see Sect. 2.1.1 for an explanation of this rather confusing statement). It is possible to configure an EJB by its vendor-specific deployment descriptor so that JBoss generates the dynamic proxies, for Home and EJB Object, with using IIOP as protocol. JBoss uses the free and open-source Java implementation JacORB [Jac] as its IIOP engine.

2.1.3. Interceptors

Primarily, interceptor is a design pattern [SSRB00, 109ff.], describing a callback mechanism for middleware remote invocations. It was developed by MiddleSoft for their MiddleORB CORBA framework. In syntax of the design pattern, the component container defines the *interceptor callback interface* with its hook methods, and the implemented *concrete interceptors* act as callback handlers before and after a method invocation. Several concrete interceptors can be installed. This allows the non-intrusive integration of services into the component container: An installed interceptor is involved in all the invocations to the component and thereby allows the insertion of any additional functionality into the invocation path of a component invocation – either at client-, server-, or both sides of the invocation. The multiple interceptors, that are installed for a component, are executed consecutively to their registration during each request.

OMG has already adopted and standardized CORBA interceptors as *Portable Interceptors* [Obj01] and the J2SE 1.4 is compatible to them [Sun03f]. J2EE 1.3 has not standardized an own interceptor mechanism yet, and JBoss' interceptor interface is not compatible to the CORBA API, but we assume that the J2EE community will adopt and standardize this great feature soon.

JBoss interceptors [SJ03, 192f.] can be configured pro EJB: Every EJB deployed to JBoss has according to its type several standard interceptors such as security or logging. The installation is done by adding a JBoss specific XML line to the EJB's vendor-specific deployment descriptor META-INF/jboss.xml. Listing 2.1 shows the relevant excerpt for a stateless session bean. For JBoss interceptors the interceptor callback interface is different for client- and server-side. For server-side interceptors the <container−interceptors> tag contains the list of concrete interceptor classes, which will intercept the EJB's invocations. For client-side interceptors the <client−interceptor> tag contains two lists: One for the invocations of the Home Object and a separate one for the invocations of the EJB Object.

The reason why there is only one list for the server-side JBoss interceptors and two lists for the client-side is the following: By server-side the interface consists of two methods invoke() and invokeHome() that have to be implemented. Thus, only one interceptor is installed for both kinds by inserting a single line in jboss.xml. At client-side there is only the invoke() method. Therefore two interceptors have to be implemented to cover both Home and EJB Object type, and the interceptors are installed separately in jboss.xml. Of course, the same client-side interceptor can be installed for both purposes if the needed functionality is the same. But it still has to be installed twice in this separated fashion.

```
<!-- SERVER-SIDE -->
...
  <container-interceptors>
    <!-- server-side interceptors handle both Home and EJB Object -->
    <interceptor>org.jboss.ejb.plugins.
                 ProxyFactoryFinderInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.
                 LogInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.
                 SecurityInterceptor</interceptor>
   ...
    <interceptor>org.jboss.resource.connectionmanager.
                 CachedConnectionInterceptor</interceptor>
  </container-interceptors>
...


<!-- CLIENT-SIDE -->
...
  <client-interceptors>
  <home> <!-- interceptors for Home Objects -->
    <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
    <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
    <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
  </home>
  <bean> <!-- interceptors for EJB Objects -->
    <interceptor>org.jboss.proxy.ejb.
                 StatelessSessionInterceptor</interceptor>
    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
    <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
    <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
  </bean>
  </client-interceptors>
...
```

Listing 2.1: Interceptor definition excerpt from the jboss.xml deployment descriptor

The JBoss interceptors are not called one after the other by a central unit (as are the CORBA Portable Interceptors) which would result in stern-like callbacks, instead a chain[4] is formed: The Dynamic Proxy calls the first interceptor, which has to call the next interceptor and so on. The reference to the next interceptor is available by the inherited getNext() method. The class that concludes the chain – for example InvokerProxy at client-side – also implements the interceptor interface, so interceptors can adhere always to the same scheme and need no differentiation of their position in the chain.

The scheme for client-side interceptors is given in Lst. 2.2. As it is apparent by the Java source code there are not two individual callbacks before and after the invocation. Instead the interceptor is part of an invocation-stack: Calling the getNext().invoke() method blocks the interceptor until the invocation is returning from the EJB Instance. Interceptors that do not call the next one, or ones not returning with the received result value, are destructive to the invocation. The administrator of the application server has to check for the conformity and correctness of all applied interceptors.

There is a specialty about invocations at the local interface[5], because only the server-side interceptors are instantiated and invoked for them, by JBoss. However, if an EJB invokes other ones by the remote interface all defined interceptors will be used. It is not exceptional that an EJB invokes another EJB by the remote interface, even if they are inside the same server, for example if the remote interface is the only provided one.

There is a limitation if only server-side interceptors are used: The JNDI name of the invoked EJB is only available from the Invocation objects of client-side interceptors. Also, any information about the client application, like IP as well as process or thread ID, is not available from the JBoss interface. Therefore, such information about the client is only available if a client-side interceptor is used to extract it.

[4]Similar to the *Chain of Responsibility* pattern [GHJV94, 223ff.], but still in the specific intention of the interceptor pattern.

[5]For a better understanding of the difference between remote and local interfaces refer to Roman et al. [RAJ02, 401f.].

```
import org.jboss.proxy.Interceptor;

public class ClientInterceptor extends Interceptor {

        /* invoke() methods are called by the preceeding
         * interceptor -- or initially by the DynamicProxy */
        public Object invoke(Invocation mi) throws Throwable {

                // INWARD ACTION
                ...

                // DELEGATE the invocation to the next interceptor
                Object result = getNext().invoke(mi);

                // OUTWARD ACTION
                ...

                // Must RETURN with the result of the EJB Instance
                return result;
}       }
```

Listing 2.2: Common behavior of a JBoss interceptor

Finally, we want to give some introduction to the *Transaction ID* (TA-ID) provided by the application server. It is unique to all invocations of the same transactional context. Every EJB defines its own kind of participation by its deployment descriptor. These kinds are specified by J2EE: Required, RequiresNew, Supports, Mandatory, NotSupported and Never [Sun01a, 357–359]. For example the semantics of RequiresNew is: The EJB container starts always a new transaction for the bean and the bean does not join in on any transaction running for the caller.

Because the EJBs themselves decide about their participation, the TA-ID may be different for each invocation between some interacting EJBs. For example EJB1 (Never) invokes EJB2 (Required), which invokes EJB3 (Required) and EJB4 (RequiresNew). The EJB container does not create a TA for the invocation of EJB1. For the invocation of EJB2 a transaction will be created. The invocation of EJB3 will participate in the TA of EJB2, but not EJB4 for which a new TA is created. In this scenario there exist two different TA-IDs for every invocation of EJB1 by a client application. Some visualization of the TA-ID is provided in the appendix A.2.1 as Fig. A.7.

2.1.4. Hardware Platform

The run-time hardware environment of the JBoss server used for this thesis was based on a standard PC system with an AMD K7 1400 MHz processor and 1GB SDRAM. The SuSE 8.2 Linux distribution was used as operating system (OS), with a kernel version 2.4.20-4GB-athlon. The server also hosted the Eclipse [Ecl] development environment, running the JBoss application server simultaneously. The graphical user interface was KDE3. All measurements were taken either on the server or on clients with equivalent hardware equipment, interconnected by an Ethernet-based 100 MBit LAN.

2.2. Java Profiling Techniques

This section is a short introduction and is provided at this point mainly for a general understanding of some of the Related Work, in Sect. 4.1. The *Java Virtual Machine Profiler Interface* (JVMPI) is more deeply discussed in Sect. 5.4.3 as well as *bytecode instrumentation* for resource accounting in Sect. 5.4.2.

First of all, the term *profiler* has the general meaning of an application that generates a profile about an arbitrary kind of aspect of some other application.

In our context a profiler is related to performance characteristics, and in the context of Java common available profilers are based on JVMPI. The profiler must be installed to the JVM to be able to provide information about a specific execution of a Java application. The information includes normally the type and number of instantiated objects as well as the names and numbers of the invoked methods and their approximated and averaged run-time. Some profilers even create call-graphs and the proportion of the individual method-call's run-time in relation to the entire run-time of the superior call.

Most profiler implementations target dedicated development environments and profiling of stand-alone applications. They can hardly be used to gather the necessary information to create performance characteristics of EJBs, or to monitor components in a productive environment. But this is discussed in Sect. 5.4 later.

2.2.1. Java Virtual Machine Profiling Interface

The JVMPI defines a callback interface for obtaining profiling data from the JVM. It is based on the *Java Native Interface* (JNI) [Sun03e], which ensures binary compatibility of native method libraries across JVM implementations on a given platform.

The *profiler agent* comes as a .so or .dll file, implemented in C or C++. Its name is given to the java command as parameter. This way, it is loaded by the JVM at start-up time. The agent uses JVMPI as a two-way function call mechanism, because on the one hand the agent is called back by the JVM or invoked by a provided JNI native interface, and on the other hand the agent holds a reference to the JVM and can invoke Java methods.

The profiler agent uses the JVMPI to register for certain types of events, and may request more information from the JVM in response to a particular event. The easiest example is the *Hprof* [HPr] profiler agent that is distributed with the Sun JDK.

The *profiler front-end* either stores or renders the profiling information. It communicates with the profiler agent by an arbitrary, developer-defined wire protocol. The JVMPI architecture is shown in Fig. 2.3.

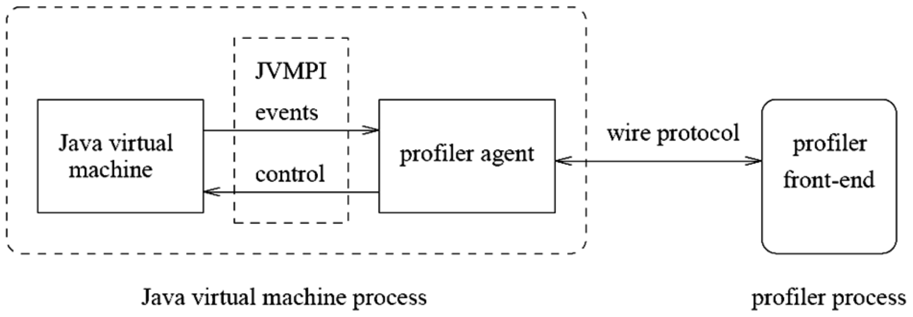


Figure 2.3.: JVMPI architecture [LV99]

The JVMPI is designed to be completely independent of the underlying JVM implementation and profiler vendors need not rely on custom instrumentation in the

JVM. It covers CPU, memory, thread and garbage collector events. A mechanism for run-time deactivation can be provided by the developer, using a boolean value to order the agent to return immediately after any callback. The order can be given either by the profiler front-end or by a JNI call to the profiler agent from inside the JVM. The exact semantics of a deactivation is defined by the profiler agent and a callback always occurs.

2.2.2. Bytecode Instrumentation

Java bytecode is the OS-independent code that can directly be interpreted and executed by a JVM.

Bytecode modification is some general kind of manipulation applied to bytecode. There exist some Java libraries, for example the *bytecode engineering library* (BCEL) [BCE], that support developers in modifying bytecode. Modifications are possible in a static way, applying them to class files before they are executed, or the modifications can be applied dynamically, either enhancing the JVM's class-loader or using a JVMPI agent to apply them during class-loader events (see Sect. 5.4.3).

Bytecode instrumentation is a special kind of bytecode modification, and the term will always refer to the principle of the explanations below. Bytecode instrumentation exclusively instruments the bytecode itself to measure resource consumption. There exist approaches which use bytecode modifications just to insert callbacks to complex monitoring objects that instrument OS dependent system calls for thread resource consumption. Such approach does not apply bytecode instrumentation in our terminology.

This following introduction is only concerned with CPU time [Vid01], but instrumentations for memory and network are possible likewise.

At first, the bytecode instructions of a class are split up into *blocks of execution* (BoE) by static analysis. A BoE is atomic, which means that it contains no conditional operators. Thereby, a BoE is executed either not at all or completely, so each BoE has a specific amount of bytecode instructions. A counter is added to each class. Also, an instruction is inserted into each BoE that increments the counter by the block-specific number of bytecode-instructions. The counter will be accessed during run-time, to gain the consumption performed by the executed BoEs of an Java object instance, at any time, as long as the object exists. The bytecode manipulations take place after compilation and before running the application, and this first primitive approach does not account for the different types of bytecode instructions.

The advantage of bytecode analysis in contrast to source code analysis is the applicability on already compiled and deployable EJBs. The overhead is low, and the instrumentation can be dynamically applied by the JVM's class-loader or the J2EE server's mechanism for EJB construction. The limitations of bytecode instrumentation will be discussed in Sect. 5.4.2, and are for example poor precision of CPU time and allowing only a maximum estimation of memory usage [VB01, 9].

2.3. Time Measurement

Most computer systems represent time internally as the elapsed seconds or microseconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970. The accuracy of this counter can only be guaranteed if an appropriate clocking hardware is available. Every IBM PC compatible computer features a battery-backed *Real Time Clock* (RTC) chip that contains an oscillator, which drives a date/time counter.

2.3.1. Process Global vs. Process Local

On POSIX-compliant systems, the gettimeofday(2) system call allows time measurement with an accuracy and quantization error of one microsecond. The exact resolution depends on the quality of the RTC, but tests on our system showed that the assumed resolution of gettimeofday(2) is reasonable. If the run-time is taken by calling gettimeofday(2) before and after an execution, the difference may not necessarily be equal to the CPU usage, because the OS process-scheduler will probably have scheduled other processes during the measured run-time and their CPU usage is included as well as the overhead for I/O operations. Therefore the usage of gettimeofday(2), or any equivalent, is called *process global* time measurement.

By means of the OS process-scheduler and its transparency to the processes, the CPU consumption of an individual process, with no interference from any other process, can only be accounted by the OS itself. The OS has to ensure that any scheduling is timestamped and the pairwise differences are accumulated to a counter specific to the intermediary process, representing the *process local* CPU time consumption. For example, on POSIX-compliant systems the `getrusage(2)` system call allows to query the OS for process local information.

2.3.2. Java Threads – Thread Global vs. Thread Local

The same that applies for processes at OS layer does apply for Java threads[6] at JVM layer. The *thread global* measurement is based[7] on process local measurement, but it is not aware of thread-scheduling. The *thread local* measurement can be achieved by using process local measurement before and after the scheduling of the threads, accounting each thread individually. Unfortunately, there is no thread local information available from current JVMs.

The only method for time measurement is `System.currentTimeMillis()`. It has an unit and nominal accuracy of $1ms$, but its accuracy can only be assumed to be "in units of tens of milliseconds" [Sun03a]. If a higher accuracy is needed, a JNI-based access to OS dependent system calls, like `gettimeofday(2)`, has to be implemented.

Thread local measurement does either need the modification of the JVM, or the installation of a JVMPI profiler agent. In theory, a JVM that implements the JVMPI specification provides a C-method named `GetCurrentThreadCpuTime()` that should provide thread local times. Examination of the Sun reference implementation for Linux shows that `GetCurrentThreadCpuTime()` relies on the process global `gettimeofday(2)` system call, and not even conforms to the specified unit of nanoseconds, but uses microseconds. Even for Solaris and Windows the `GetCurrentThreadCpuTime()` relies completely on OS systems calls and implements no own accounting. This is possible for these OS, because the JVM is run in just one process and its implementation

[6]Besides Java threads, the OS itself can also realize threads (*OS threads*). There exist different types and implementations of OS threads. A JVM does not have to map its Java threads to OS threads, so these two types of threads have to be distinguished carefully.

[7]Thread global measurement is identical to process local measurement if the JVM is run in just one process. But this is not the case for the Sun reference implementation for Linux.

maps the Java threads to OS threads, so the OS-provided thread local CPU-time information is reliable for the Java threads.

Thus, the GetCurrentThreadCpuTime() method is in general not reliable to report correct thread local time information. But using process local time measurement in combination with thread-related events allows the profiler agent to realize thread local time measurement on its own, by accounting the differences of process local time measurement for each thread individually before and after the thread is scheduled.

2.3.3. Performance Counter

The resolution of the RTC-based measurement does not go beyond microseconds. To achieve more exact time information, it is necessary to use another source of clocking signals. A modern CPU with a frequency of 2 GHz has an CPU cycle impulse generated every 0.5 nanoseconds. CPUs like Intel's PentiumPro and Pentium4 have several new registers called *performance counters* that can be configured and used for the counting of various events. This way, a 64bit register exists that increments on each CPU cycle. This register can be accessed for high-resolution time measurement as accurate as the CPU's frequency.

There already exist several libraries that abstract from the various processors and OS: PAPI [PAP] and PCL [PCL], both supporting many different processors and OS, and Rabbit [Rab], that concentrates on Intel or AMD processors and Linux. PAPI and Rabbit use the C language. PCL has language bindings for C, C++, Fortran, and Java. Specializing on timing routines is the PTR working group [PTR] that aims at multi-processor and workstation platforms, but their work is in an early stage and no implementation is available at the time of writing this thesis.

2.3.4. Measurement Intrusion Effect

Any kind of measurement not only introduces overhead but also changes and influences the result in a way called "Heisenberg-like effect"[8]. Every time we take a

[8]Werner Heisenberg, German Physics Scientist (∗1901-12-05, † 1976-02-01), stated in his work, that it is impossible in Quantum mechanics to do an observation without influencing the experiment itself. In the year 1927, he postulated his famous "Unschärferelation".

timestamp or query the OS for process local information we have to be aware that the corresponding call itself takes time for execution. The given result value is not only the time for the wrapped execution part but does include time for the measurement call, visualized in Fig. 2.4.

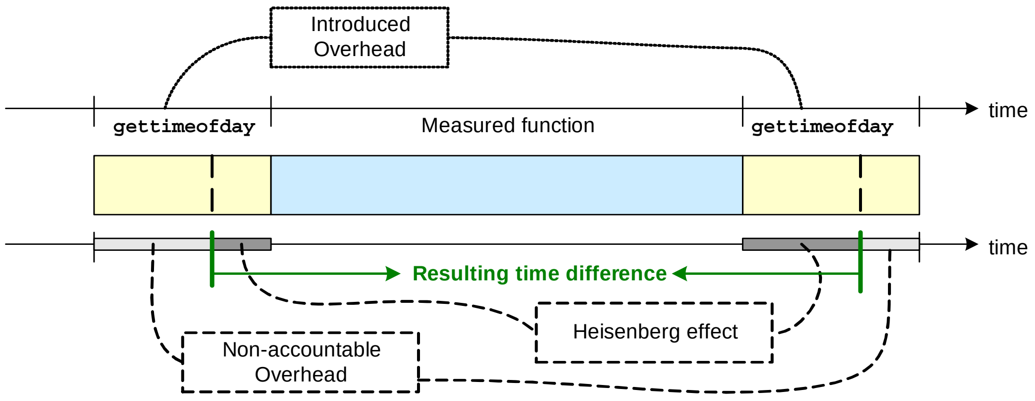


Figure 2.4.: Time measurement intrusion effect

Because of the constant time needed to call gettimeofday(2), the intrusion effect is comparatively the smaller the longer the measured execution block is. If the average duration and rate of timer calls is known their measurement intrusion effect can be quantified.

In our environment the call of gettimeofday(2) takes about $0.254\mu s$ as the mean average of 10000 tests[9]. This implies that it takes less time than its resolution. An interesting in-depth look at quantization, resolution and operation of the Linux time measurement functions is given by Millsap and Holt [MH03, 141ff.] in relation to the Oracle kernel timings. There is also denoted that some time ago the gettimeofday(2) system call had a much greater delay than its resolution, but was improved especially for the Oracle DBS to its current performance.

2.4. Summary

In this chapter, a brief introduction to the EJB component model was provided as well as the JBoss architecture and its interceptor mechanism. The description of our

[9]The methodology and source code for this statistics is given in the appendix A.3.

hardware platform is needed in later chapters to qualify our empiric measurement values.

A short introduction into JVMPI and bytecode instrumentation will provide enough understanding of Java profiling techniques until we discuss their usage for CPU and memory measurement in detail.

Also, an introduction in time measurement was provided, focusing on POSIX-compatible OS system calls. Only the OS can provide process local resource information, and in the same way thread local information can only be provided at JVM layer. This is not available by current JVM implementations, which provide insufficient functionality for time measurement. Finally, we have discussed performance counters for high accuracy timestamping, and the limitations of any time measurement due to the Heisenberg-like measurement intrusion effect.

3. Basic Classification System

For the architectural stack that consists of client, EJB, EJB container, JVM and OS there exist several measurement approaches and frameworks, but the semantics and vocabulary of their descriptions differ in some major aspects. Especially, the term “intrusion” is often used sloppy or even improperly. Thus, a common classification system is helpful to understand, compare and discuss the different approaches.

3.1. Instrumentation

Instrumentation is the usage, adaption or rewriting of an application, or parts of an application, in order to gain a required functionality or service. By the principle of abstraction there are several layers and locations among the layers where such instrumentation can take place.

We differentiate between three *basic layers*: Application, Execution Environment and Operating System. Each basic layer can be subclassified into the (exact) *layer of instrumentation*, where performance characteristics can be gathered and the accounting information can be produced:

1. Basic layer: application
   Layer of instrumentation: e.g. EJB or client-software
2. Basic layer: execution environment
   Layer of instrumentation: e.g. JVM or EJB container
3. Basic layer: operating system
   Layer of instrumentation: e.g. kernel, kernel-modules, drivers or applications accessing and listening to hardware-related interfaces

The *types of instrumentation* are various, for example changing the source code, modification of bytecode or insertion of an interceptor.

3.2. Intrusion

A discussion of the intrusion has always to be related to some context. We distinguish mainly between the behavior and the system resources. Both contexts will be subclassified soon.

The *types of intrusion* are the *intrusive* type and the *non-intrusive* type. For example, using a separated and passively listening hardware monitor for LAN activities, that transfers its measurement data by a separated LAN, is exemplary for the the non-intrusive type in all respects and contexts. On the other hand the instrumentation of the client source code for the measurement of response time, taking timestamps with `System.currentTimeMillis()` and sending the values over the network, is an example for the intrusive-type in nearly all respects and contexts.

3.2.1. Basic Contexts

Behavior and system resources are completely different contexts where intrusion is relevant and will be discussed separately:

The most interesting subset of **behavior** is the *source code*, because at the location of instrumentation the downright behavior is always changed, but the source code does not need to be affected for this necessarily. For example bytecode modification, or binary code modification, is non-intrusive to the source code.

Another important view, in the context of behavior, is the *participants view.* In contrast to the source code view it is independent from the type of instrumentation and relates to the communicating partners, e.g. the client and the EJB. If the instrumentation (for measurement) is transparent to the participants, it is non-intrusive according to this view.

One can think of more role-dependent views, for example the *administrator view* which is concerned whether the administrator of the EJB container has to consider the instrumentation by any chance or not as well as the *deployer view* likewise.

System resources are basically CPU, memory, network and mass storage IO. Most solutions use either network or mass storage to transfer their data. All software solutions have to be intrusive in regard to CPU and memory. Additional hardware is always needed for non-intrusive approaches.

3.2.2. Views at Intrusion

From the hierarchy that results from both contexts and their sub-classification there are especially six views[1] which are most interesting for this thesis. The intrusiveness of any measurement framework will be discussed according to them:

- Basic Context: Behavior
  1. Participants view (considers transparency for the users/layers above)
  2. Source code view (considered only at the location of instrumentation)
- Basic Context: System Resources
  3. CPU view
  4. Memory view
  5. Network view
  6. IO view

3.2.3. Short Classification Scheme

A short classification table, that will be provided for the approaches, includes the exact layer of instrumentation and the six views at intrusion. It adheres this scheme:

Layer of Instrumentation	Participants	Source code	CPU	Memory	Network	IO
...	...	...	...	...	...	...

Intrusive and non-intrusive will be abbreviated as I and NI. If a certain type of intrusion depends on a kind of implementation which is not known a dash ('–') will be used (this omission regards mostly the usage of network or mass storage, because most approaches discuss only the principle of measurement, not its communication with the analyzers).

[1]The term "views at intrusion" could be substituted, according to the instrumentation nomenclature, by "(exact) *context of intrusion*". However, we prefer to use the term "view" for this purpose, because the classification of instrumentation and intrusion is orthogonal to each other.

3.3. Time-driven vs. Event-driven

The **time-driven** measurement is sometimes called *statistical profiling* or *periodically sampling*. It is based upon a timer with a fixed interval. The measurement is performed after each time-out interval. The problem of time-driven approaches is that their accuracy always depends on the length of the sampling interval. But its advantage is that the length can be changed and therefore the overhead can be reduced by elongating the interval. So in the majority of cases the time-driven technique is the one with the least overhead.

As an example for CPU measurement in Java, a pseudo-algorithm [LV99] is given:

```
while ( true ) {
  sleep for a short interval;
  suspend all threads;
  record the stack traces of all threads that have run in the last
     interval;
  attribute a cost unit to these stack traces;
  resume all threads;
}
```

The main task consists of the decision whether a thread has run or not. Solutions for this task are exemplarily discussed for JVMPI in Sect. 5.4.3.

Event-driven measurement means, that in case of an event, the execution environment or OS delegates the control of execution flow to an event handler. The event-handler must take care of the accounting. With the absence of intervals, the time-differences have to be taken explicitly and the accuracy is of most possible exactness. Every scheduling operation of the OS or execution environment has to generate an event to allow the profiler the correct tracking of currently running processes or threads, and thereby to assure the correct assignment of resource consumption. The interval of scheduling is naturally shorter than any timer interval, which has to be requested by the above layer. Thus, the number of callbacks to the event-handler is higher than the number of calls for the time-driven approaches, and the overhead has to be considered carefully. As major example, refer to JVMPI in Sect. 5.4.3, which allows both the time-driven and event-driven approach.

3.4. Direct Support vs. Profiling Interface

The “Direct Support vs. Profiling Interface” discussion is generally possible at each of the three basic layers (OS, execution environment, application layer[2]). But the main focus in the context of EJBs lies obviously on the execution environment layer, concerning JVM and EJB container.

For **Direct Support**, the execution environment has to offer the accounting of system resources as engageable service. The accounted information can be requested from the environment by interested applications like an analysis front-end.

As **Profiling Interface** there exist two slightly different types available: The first kind of Profiling Interface uses callbacks, which take place after events. Profilers register as event-handlers to the execution environment, which in turn calls them at specific events. The profiler has to gather most information on its own, like ID of the programmatic object, amount of memory or the timestamps, by enquiring the execution environment. The second kind also allows the registration of profilers, but the execution environment does some accounting on its own and is able to deliver that gathered information at callback-time to the profiler.

3.5. Variations of Profilers

Independent from the technique that is applied by a profiler[3] there are some aspects that allow further classification of the approach. We assume that most readers have already used profilers to improve their own source code, so that the reader associates specific attributes to profilers. Therefore, this section is mainly provided to broaden the view at profiler implementations.

3.5.1. Granularity

The granularity at which performance characteristics can be made available depends on the profiling technique and the existing API. With the several hardware and

[2]For example, a special type of EJB could be instrumented for its own accounting. So, the architecture designers would also have to address this differentiation at application layer.

[3]The term profiler has been previously defined in Sect. 2.2.

operating systems in mind, there exist different abstractions between run-time application and source code elements. Concentrating on the Java environment, the types of granularity are:

1. Application / EJB Container
2. EJB
3. Process
4. Thread
5. Method
6. Line of source code
7. Expression (of progr. language)
8. Bytecode instruction
9. Machine-code instruction

Naturally, the profiler's overhead increases along with the fineness. Some profilers support more than one granularity.

3.5.2. Usage Site

In the majority of cases, the purpose of the gathered information can be related to several states of the application's life-cycle. Thus, a distinction between profilers is possible:

1. Provide information to programmers
2. Provide information to the compiler
3. Feedback to administrators
4. Feedback to the runtime system

Points 2 and 4 aim at an automated consumption, analysis and reaction. The Comquad project inter alia researches solutions for 1, 3 and 4. Classic non-component profilers are already available for 1.

3.5.3. Online vs. Offline Analysis

The question whether a framework allows access to performance values during run-time allows further distinction:

Offline means the analysis of collected data after termination of the process. The post-processing of the data has the advantage that enclosing knowledge is available and overhead at run-time can be reduced.

Online analysis takes place at run-time, for example by a graphical front-end as interaction with the administrator. Also monitoring, load-balancing or automated reaction by triggers can be based on online performance characteristics.

The view-point for "termination" depends on the current layer: At execution environment layer, it is either the JVM and/or the EJB container that terminates – and there are several Java-Profilers available that need the JVM to terminate before analysis can be applied. Within the scope of J2EE the JVM includes the EJB container, thus, the application of such a technique implies many restrictions, like the non-practicability in productive environments.

But termination could also be seen from application layer, referring to the whole EJB or even a single invocation. Largely, the focus does not lie on long running EJB-calls, and if not stated otherwise, even online-solutions are at least offline in relation to the EJB calls, because they allow no analysis unless all information of one specific business-call is guaranteed to be available.

3.5.4. Dynamic Activation

In order to use the performance monitoring of EJBs in a productive system, the profiler should provide a mechanism for activation and deactivation; thereby, profilers can be distinguished.

Note, that the possibility to dynamically activate the profiler results in the *partial profiling problem*: When profiling is enabled after the thread (or EJB) was started, or after a number of classes have been loaded and a number of instances of these classes have been created, the profiler may encounter events that contain unknown IDs. Therefore, the profiler needs an interface to the execution environment, that allows to request information about such unknown IDs.

3.6. Summary

This chapter has introduced several criteria a classification of profilers can be based on. We particularized the terms instrumentation and intrusion which are orthogonal to each other. There have been opposed the time-driven and even-driven measurement, as well as the direct approach and the profiling interface approach.

Related to profilers there are some classifying aspects which are independent from the applied technique, namely the granularity at which performance characteristics can be made, the usage site of the profiler, whether it allows analysis at run-time and whether it can be deactivated to cut off run-time overhead.

For the classification by the layer of instrumentation and the type of intrusion, according to the six views at intrusion, a short classification scheme has been defined. It will be applied to all performance measurement approaches in the following chapters. The other classification aspects will be applied if they are appropriate.

4. Related Work

Before presenting our own approach to the topic, we will present published approaches and technologies for various means of performance measurements. Figure 4.1 gives an overview. They are filed by their layer of instrumentation, and the targeted characteristics are indicated by the boxes. The dashed boxes relate to characteristics, which are not primarily concerned by the technique, but for which some information can be provided. Especially, the resource monitoring applications that access OS interfaces are drawn as dashed, because the available information can hardly be related to individual components.

Each approach is described in the following sections, where additional classification is applied: There is given the type of intrusion for each approach in regard of the six views at intrusion from Sect. 3.2.2. The source code view concerns only the layer of instrumentation, so do not confuse it with the EJBs. Clients' and EJBs' source codes are implicitly represented as participants. The intrusion views for the system resources always relate only to the server-system.

4.1. Software Solutions

Commercial environments like *PerformaSure* [PSu] from Quest Software or the upcoming *OptimizeIt ServerTrace* [OIt] from Borland offer software for monitoring J2EE applications. They cover a broader spectrum of J2EE integrating tools to monitor database (DBS) connections, network traffic or virtual machine parameters.

Regarding EJB-components in particular, OptimizeIt ServerTrace will support method level drilldown of CPU and memory consumption as do classic profilers. However, these tools are primarily targeted to support pre-deployment teams and to reveal performance bottlenecks in a special development environment after their

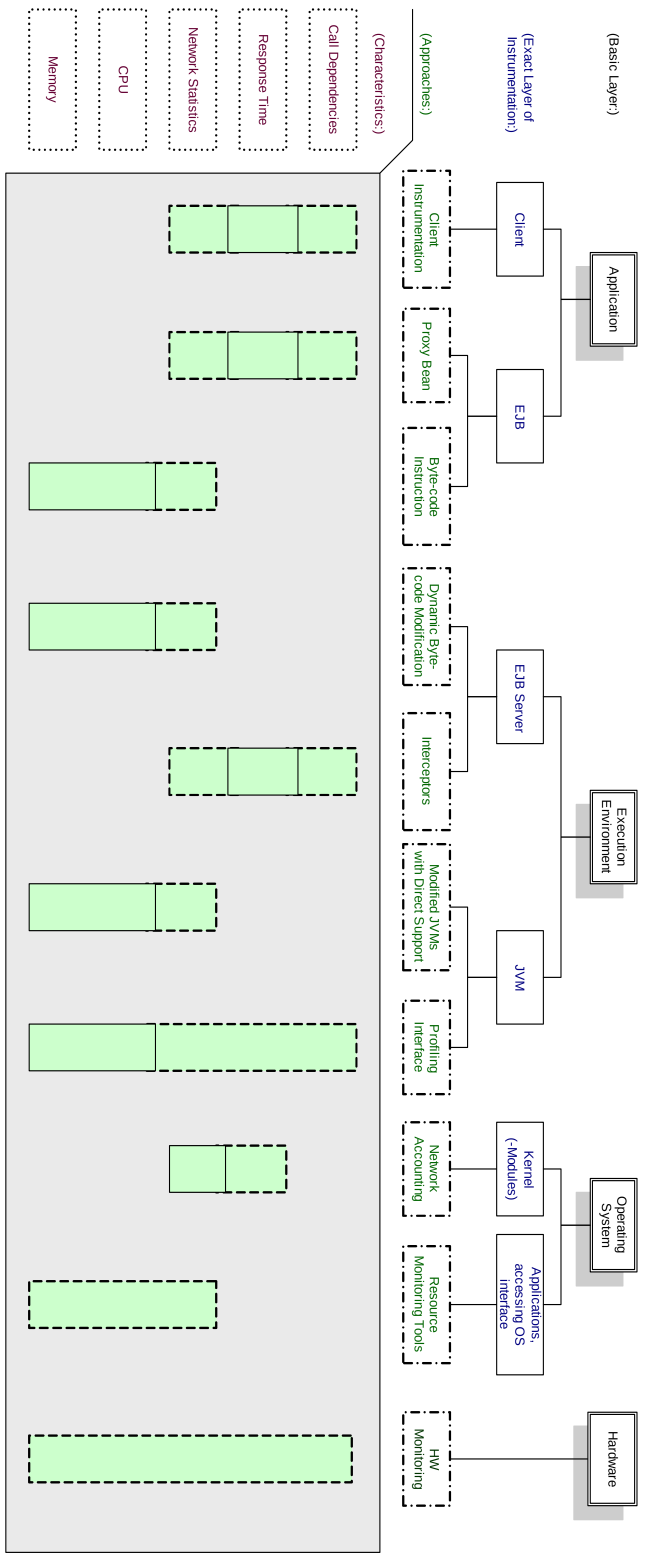


Figure 4.1.: Overview over the Related Work

discovery in productive use. The availability of performance statistics and characteristics about the whole EJB and its sorts of business methods or its interdependencies to other EJBs is not provided. Nor are the interdependencies to the method's parameters considered as it is the goal of our part of the COMQUAD project. But these tools and vendors do seem to possess the required technology for such kinds of measurements and statistics.

With the exception of the proxy-bean approach (see below), we have found no other work that addresses transparent measurement of the response time of EJBs as well as we have found work to the deduction of call dependencies between EJBs only by Aguilera et al. [AMW+03] in the generally related contexts of CORBA and free-form communication (see Sect. 5.3.5).

The following three sections will present software-based techniques concerning response time and resource consumption sorted by the basic layers.

4.1.1. Basic Layer: Application

Client instrumentation

The most simple approach to measure response time is to instrument the clients[1] of the EJBs. Related to the measurement of response times, there exist frameworks designed for load testing functional behavior to be used as accompanying tools.

Examples are the open-source project *The Grinder* [Gri] and commercial products like Empirix' *Bean-test* [Bea]. Usually they focus on the task of generating load to test software, according to their behavior and stability under many concurrent requests. They can be used as request generating clients to the components which are to be measured.

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
Client-application	I	I	I	I	–	–

[1]As plainly as client instrumentation for response time is the instrumentation of the EJB to measure its own (process-global) run-time. For example with with `System.currentTimeMillis()`, as do some EJB developers to log their EJB's performance.

Proxy-bean

Another approach to apply response time measurement and monitoring is based on the proxy pattern: a proxy component is generated for each real component. The proxy bean steals the JNDI name and therefore appears as the bean to be monitored. This way, it is possible to receive, monitor and delegate the original invocations. This feature is implemented by the Compas framework [MM02], which aims at a solution to aid developers of distributed, component-oriented software in better understanding performance related issues of their applications. It consists of a monitoring module to obtain information about a running application, a modeling module to create and annotate models of the application with performance parameters gathered, and an environment for the prediction of expected performance under different load conditions.

The *EJB Monitor* [MM01a, MM01b], the monitoring module of Compas, is developed as an own project. It is a tool for runtime-monitoring of components following the EJB standard, and is usable with J2EE compatible application servers (currently, just WebSphere Application Server 5.x is supported). Although transparent to the client and the other components, this heavyweight approach adds an additional indirection between each caller and callee: an extra marshalling/unmarshalling between the proxy-bean and the original bean takes place for remote interfaces, because they are deployed separately. Furthermore, the amount of EJBs is doubled if each component should be monitored, and the vendor-specific deployment descriptor of the monitored bean has to be changed, because the JNDI name has to be altered.

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
EJB	NI	I	I	I	I	–

Bytecode instrumentation

J-RAF2 [HK03] uses the bytecode instrumentation mechanism, described in Sect. 2.2.2, for resource management. It is being developed as part of the *JSEAL-2* project [Bin01] that aims at mobile agents. The objectives are prevention against malevolent code and DoS attacks based on CPU, memory or communication abuse as well as the fair distribution of resources [BHV01]. Because JSEAL-2 emphasizes on portability, the J-RAF2 framework for resource accounting was designed as a pure Java implementation, and therefore uses bytecode instrumentation as accounting technique, utilizing the BCEL [BCE] bytecode engineering library. The bytecode instrumentation is applied statically after compilation, which will force a redeployment, if used with EJBs.

The limitations of bytecode instrumentation, concerning the precision of CPU time and memory usage, is only a minor problem in preserving a system from DoS attacks and achieving the other defined objectives – but it is a major problem if precise characteristics of EJBs are desired.

Concerning J2EE, it seems possible to apply the bytecode instrumentation on EJBs before deployment, but in [BHV01] it is mentioned that J-RAF2 is restricted to the J-SEAL2 platform, because the agents and applications have to conform to a specific programming model. Thus, the adaptability of the existing implementation needs further evaluation.

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
EJB	NI	NI	I	I	–	–

4.1.2. Basic Layer: Execution Environment

Dynamic bytecode modification

JRes [CE98] is a library for resource control of individual threads. Although designed as library, JRes is specifically coded for the Microsoft JVM [CE98, 13]. The accounting includes CPU, memory, and network resource consumption, using native code of the underlying OS, e.g. by providing a complete new set of java.net.* classes for network accounting. JRes does not apply the bytecode instrumentation as we have described in Sect. 2.2.2, but only inserts bytecode that results in callbacks to the Microsoft system calls for OS native thread resource consumption information, assuming or requiring that Java threads are mapped one-to-one to OS threads, which is not demanded by the Java specification.

In the original JRes there is no notion of an application or EJB, for example as a group of threads, but Czajkowski et al. present an enhanced interface [CHS+03], that introduces the concept of "isolates" as abstraction of applications.

Paradyn-J [NM99] is another example for bytecode modification at run-time for performance measurement. Paradyn-J instruments the JVM's runtime compiling routines. The focus lies on profiling dynamically compiled Java executions, additionally to interpreted bytecode executions. Paradyn-J aims at the identification of bytecode that profits most by dynamical compilation, and at performance tuning of applications. In this context, Tia Newhall provides broad information about dynamic modification of bytecode in her dissertation [New99].

By extending the JVM's class-loader to perform bytecode modifications to classes, JRes inspires the instrumentation of the J2EE creation mechanism for EJBs (we call this *EJB-loader*), and therefore the modification of EJBs without their source code being available. These dynamic bytecode modifications could even equal the principle of bytecode instrumentation from Sect. 2.2.2 and 5.4.2.

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
EJB server	NI	I	I	I	–	–

Interceptors

We have found no approach that instruments interceptors to measure response time and to derive call dependencies, as we will later propose as part of our solution. But there are projects that use JBoss interceptors for various purpose, for example client-side caching of components [PS03].

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
EJB server	NI	NI	I	I	–	–

Direct accounting support in the JVM

KaffeOS [BHL00] is based on a modified Kaffe JVM and includes a non-Java *base OS* layer. Because the base OS is not visible to the application layer, we file the KaffeOS not at OS layer but as a JVM. It supports the process abstraction as known from common OS to isolate Java applications or mobile agents, as if they were run on their own JVM. Therefore, untrustworthy programs can be executed within the same underlying address space, because direct object sharing between process spaces is prevented – although shared heaps are provided for efficient sharing between trusted processes. KaffeOS can account the usage of CPU, memory and garbage collector time for each process, with the purpose to extend the protection model by detection and thwarting of DoS attacks. It focuses on CPU time and heap memory and is not designed for application-specific resources. The modified JVM of KaffeOS is inherently non-portable unless the base OS is ported. Also, it is not possible to benefit from optimizations that are found in compilers and standard JVMs.

As long as the resource management functionality is proprietary and based on modified JVMs we classify these solutions as intrusive according to the source code view at the layer of instrumentation.

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
JVM	NI	I	I	I	–	–

Profiling interface

To provide a comprehensive overview in this section, we have to mention the JVMPI and the various implementations of profiling agents and front-ends. But again, we would like to refer to Sect. 5.4.3 for the in-depth overview and discussion of JVMPI. Among others there will be discussed the frameworks TMon [RR00], JPMT [HQGM02] and FORM [SM00, SMS01].

Some interesting comparison of JVMPI profilers according to the comprehensiveness of the extracted call graphs, also in comparison to approaches that modify the JVM or the bytecode, is provided by Xie and Notkin [XN02]. Their research and the call graphs do not recognize components or EJBs, but are just related to Java objects and common application profiling.

The classification of any JVMPI approach is provided as follows:

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
JVM	NI	NI	I	I	–	–

4.1.3. Basic Layer: Operating System

Network accounting

Network accounting at OS layer can be achieved by instrumenting the kernel or kernel modules, for example the network card drivers or the TCP/IP-stack implementation. But it is also possible at other layers: At JVM layer the java.net.* classes can be instrumented [CE98], and at the EJB container layer Mos and Murphy discuss the same idea and state that it is error-prone [MM01a].

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
OS (or JVM or EJB server)	NI	I	I	I	–	–

Resource monitoring tools

There exist various tools, that allow the monitoring of system resources, either for processes or for the whole OS. We will give some examples of commonly known Linux tools, without intending to be comprehensive.

The current memory usage information is provided by free. The top command provides CPU utilization, process statistics and memory utilization. The *sysstat* suite provides the tools iostat (statistics for several disk drives: read and write rates per second, average wait, service, and CPU utilization), mpstat (CPU statistics) and sar (collecting information over a longer period plus reporting).

Network related tools are tcpdump (packet sniffer) and iptraf (network statistics: IP, TCP, UDP, ICMP and non-IP packet counts, IP checksum errors, interface activity, packet size counts).

An interesting tool for memory debugging of applications is the *Valgrind* profiler. Valgrind works directly with program binaries. It is independent of the programming language and it is independent whether the application is compiled, just-in-time compiled, or interpreted.

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
OS	NI	NI	I	I	–	–

4.2. Hardware Solutions

The measurement approach that is non-intrusive to CPU requires the physical separation of the place of execution, which is possible on a multiprocessor system. The same can be achieved for memory usage, by taking advantage of separated memory spaces.

As long as messages of the accounting framework are sent over the network network behavior is altered, for example the agent's communication to its front-end. The accumulated quantity can be measured adequately, but throughput, time of arrival and jitter are not reliable any more. The non-intrusive approach is the separated physical LAN for the accounting communication. Because some basic information about the network characteristics is a requirement to most measurement frameworks, and because the separate LAN is cost-intensive, the intrusive measurement can be considered as reliable enough on an appropriately loose scale in most cases.

Hardware Monitor

One example for the employment of hardware monitors and separated LANs is described by Wybranietz and Haban [WH88]. They describe a transparent[2] real-time monitoring hardware, that displays measurement information about distributed systems.

Using a network of MC68000 microcomputers, the paper does not recognize components, but it provides an overview of problems related to hardware monitoring and supplies a list of further bibliography.

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
Hardware	NI	NI	NI	NI	NI	–

[2]This special approach of Wybranietz and Haban requires a special compiler that integrates event generation into the binary code.

4.3. Summary

This chapter has given an overview over related work and measurement approaches that could be adopted in the context of J2EE. At all three basic layers there is information available that can be used to characterize the QoS of an EJB component.

The absence of direct support of performance measurement by the standard JVMs and EJB containers indicates that an integration of measurement techniques is still underdeveloped. The mere broadness of the approaches shows that the exchange between the several layers is necessary for a comprehensive solution of performance measurement.

In the following chapter, we will use some of the techniques to finally obtain the design of the desired measurement facility.

5. Proposed Solutions

In this chapter our solutions are described, regarding response time measurement and call dependencies as well as CPU and memory consumption. For the response time (Sect. 5.2) and the call dependencies (Sect. 5.3) there is a complete implementation available that accompanies this thesis. An installation instruction is provided in the appendix. For the measurement facility of CPU and memory consumption (Sect. 5.4) there exists no implementation; this thesis provides rather basic research, but more detailed research and implementation will be done by two further student research projects.

5.1. The "Perfect" Solution

A comprehensive and accurate solution which can be accessed easily can only be realized architecture-layer-spreading:

1. *Adaption of the OS:*
    - Offer a simple and efficient access mechanism to performance counters, for highest accuracy and independence within a hardware architecture family.
2. *Adaption of the JVM:*
    - Utilize performance counters under guarantee of OS independence.
    - Provide direct support for *thread local* resource accounting of Java threads, as discussed in Sect. 2.3.2.
    - Embed a resource manager that administers the information about workload and causality. Thus, the resource management is centralized and the development costs for analyzers is reduced.

3. *Adaption of the EJB container:*
   - Abstract the data of the JVM resource manager on the level of EJBs for further minimization of developing costs for analyzers.
   - Provide a standardized callback interface for EJB invocations, resembling the JBoss interceptors.
   - Allow to identify invocations that are in causal relationship. For example by an ID common to all correlated invocations (resembling the TA-IDs for transactions).

We do not provide such all-embracing implementation, but to achieve a usable implementation the paradigm of our proposed solutions is the orientation towards standards.

5.2. Response Time Measurement

One of the most basic NFPs of a component is the response time (RT) of an invocation at one of its exposed methods. We assume that there exist EJBs from many different vendors that are not available as source code. So the EJBs act as black-box and are only available as bytecode, inside the deployable jar (or ear) archive.

At first we take a look at the several layers in any J2EE implementation at which RT measurements can be taken. Afterwards we introduce our chosen approach, explaining the concept and its various applications, and discuss the overhead of our method.

5.2.1. Layers for Instrumentation

There are several levels in the J2EE environment at which RT measurements are possible. The most simple approach would be to instrument the *clients* of the EJBs to measure RT. This is done by the load test frameworks given in Sect. 4.1.1. The focus of the thesis lies on a scenario where existing clients are invoking productive beans, and with client applications not available as source code but all the same with the assignment to measure as many characteristics as possible in such a productive system. Therefore, instrumentation of the client's source code can't be considered, instead the mechanism has to be transparent to the client. But, any software approach for client-side measurement of RT requires the adaption of the client's environment, for example at least by delivering a jar-library. However, one transparent solution that actually instruments the client would be the application of bytecode modifications on the client program. If the client application is not instrumented, one side-effect is the measured RT obviously not being perfectly identical to the RT that is perceived by the method of the client application.

The bottom level of possible instrumentation is the *operating system*. The instrumentation of the kernel or kernel-modules, see Sect. 4.1.3, allows the entire monitoring of network traffic, so the J2EE traffic could be filtered and thereby the RT could be deduced. This idea is applicable to the JVM's java.net.* classes [CE98] and to the EJB server by filtering the RMI calls [MM01a], an denoted in Sect. 4.1.3. These approaches are neither portable across OSs, EJB containers and JVMs respectively, nor easy to implement and maintain.

The layer above the OS is the *JVM* where the measurement of time-information seems possible by using standard interfaces like the *Java Virtual Machine Debugger Interface* (JVMDI) [Sun96] or JVMPI. But filtering would be necessary as well as complex efforts to collate the received events to the EJBs.

The next level is the *J2EE server* or the *EJB container*. A transparent location to introduce the sensors for RT seems to be the framework that automatically generates the Home and EJB Object, thereby instrumenting the stubs and skeletons. Again, it would be necessary to maintain the code for coming versions of the container and it lacks the portability to other container implementations.

The highest level, apart from the already rejected client application, is the *EJB* itself. At this level the client-side RT can obviously not be measured, but a server-side RT could be provided.

The non-intrusive[1] instrumentation of the EJB container using a callback mechanism like JBoss interceptors seems most promising, because both client- and server-side are considered and a transparent solution can be gained.

[1]According to the source code oriented view.

5.2.2. Response Time Interceptor

In a client-server setting there are several possible ways of defining RT [RZ03b]. However, the *client-side response time* ($T_{\text{resp,client}}$), including communication overhead, and the *server-side response time* ($T_{\text{resp,server}}$), approximating the process global runtime of the EJB Instance method, are the most important ones. The injection of interceptors on both sides of the invocation allows the transparent measurement of $T_{\text{resp,client}}$ and $T_{\text{resp,server}}$ by timestamping the delegation. We call them *response time interceptors* (RTI), and the scenario is visualized in Fig. 5.1.

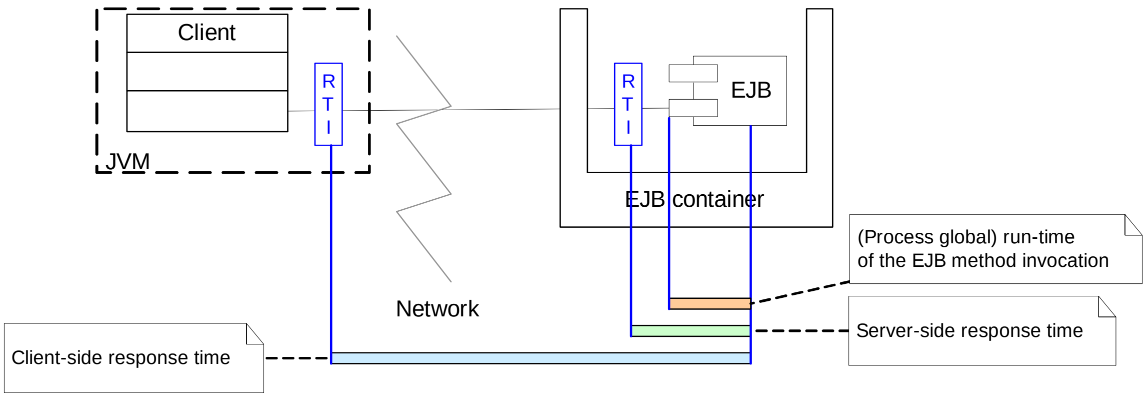

Client
R T I
JVM
Network
R T I
EJB
EJB container
(Process global) run-time of the EJB method invocation
Server-side response time
Client-side response time

Figure 5.1.: Two types of response time

The placement of the RTIs, according to the standard interceptors, is crucial, because the sequence, the interceptors are executed in, is the textual order in the META-INF/jboss.xml-file (compare List. 2.1 in Sect. 2.1.3). The top one is invoked first and the bottom one last before the invocation is delegated to the EJB itself. Therefore the RTIs have to be inserted as the first one ($T_{\text{resp,client}}$) and the last one ($T_{\text{resp,server}}$) in the interceptor chain. The possibility to place the RTIs arbitrarily makes additional semantics of the measured times possible.

As mentioned in Sect. 2.1.3, the interceptors of client- and server-side do not share a common interface so they are functionally different. But, we converge the behavior to a common logical one: an RTI_i has to take four timestamps (TS), as shown in Fig. 5.2, wherein $i \in [1; n]$ denotes the logical position of the RTI instance in the RTI chain. Only the n RTIs are iterated by i, not the other interceptors. The RTI_{i+1}

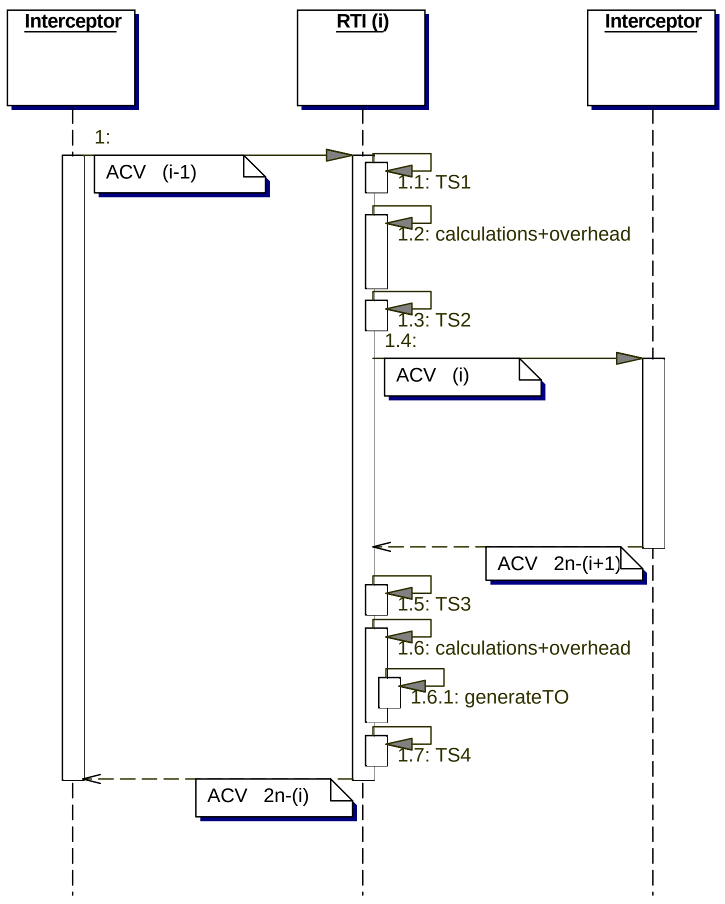


Figure 5.2.: General behavior of RTIs

is subsequent to RTI$_i$ on the inward way, but preceding to RTI$_i$ on the outward way. Therefore, we have to clarify our terminology: the usage of "successor" or "subsequent" will always refer to RTI$_{i+1}$ as well as "predecessor" refers always to RTI$_{i-i}$ from now on – independent of the matter whether we discuss the inward or outward way.

Among the four TS the TS$_2$ and TS$_3$ are for calculating the RT and the remaining TS$_1$ and TS$_4$ are for accounting the inserted overhead. The overhead between TS$_1$ and TS$_2$ mainly consists of initialization and gathering of minimal context information. Between TS$_3$ and TS$_4$ more has to be done: collecting additional context information, calculation of RTs, construction and transmission of a transfer object (TO) to an asynchronous storage mechanism (see Sect. 5.2.5).

Because the callback to the RTIs produces overhead there are two views at both client- and server-side RTs. The *uncleansed RT* ignores this fact and is plainly calculated as $(\mathrm{TS}_3 - \mathrm{TS}_2)$. The *cleansed RT* considers the overheads: Each interceptor calculates and transmits the *accumulated correction value* (ACV) on the inward ($\mathrm{ACV}_{(i)}$) and outward ($\mathrm{ACV}_{2n-(i)}$) way. The TS of an individual RTI is represented as $\mathrm{TS}_x^{(i)}$ – with $x \in \{1, 2, 3, 4\}$ and i like above.

$$\begin{array}{rrcl}
[\quad \mathit{Init}: & \mathrm{ACV}_{(1)} & := & \mathrm{TS}_2^{(1)} - \mathrm{TS}_1^{(1)} \quad] \\
\mathit{Inwards}: & \mathrm{ACV}_{(i)} & := & \mathrm{ACV}_{(i-1)} + (\mathrm{TS}_2^{(i)} - \mathrm{TS}_1^{(i)}) \\
\mathit{Outwards}: & \mathrm{ACV}_{2n-(i)} & := & \mathrm{ACV}_{2n-(i+1)} + (\mathrm{TS}_4^{(i)} - \mathrm{TS}_3^{(i)})
\end{array}$$

On the outward way the RTI_i has to account the overhead of the RTIs that have already been passed completely – these are the RTIs with an index greater than i, expressed as $\{\mathrm{RTI}_x : x > i\}$. The received correction value $\mathrm{ACV}_{2n-(i+1)}$ from RTI_{i+1} equals the overhead for all $\{\mathrm{RTI}_x : x > i\}$ plus the overhead produced of the $\{\mathrm{RTI}_y : y < i\}$ on the inward way, and the own $\mathrm{TS}_2 - \mathrm{TS}_1$. Fortunately the overhead of the $\{\mathrm{RTI}_y : y < i\}$ plus $\mathrm{TS}_2 - \mathrm{TS}_1$ has already been calculated as $\mathrm{ACV}_{(i)}$. So RTI_i can calculate the overhead of the $\{\mathrm{RTI}_x : x > i\}$ as $\mathrm{ACV}_{2n-(i+1)} - \mathrm{ACV}_{(i)}$. The cleansed RT can now be calculated:

$$(\mathrm{TS}_3 - \mathrm{TS}_2) - (\mathrm{ACV}_{2n-(i+1)} - \mathrm{ACV}_{(i)})$$

The TS are taken by a JNI module that uses OS system calls like gettimeofday(2), due to the lack of precision of System.currentTimeMillis(). A wrapping Java class tries to load the module and falls back to System.currentTimeMillis() if it was not installed to the client's JVM. In either case the unit is μs.

This kind of timestamping provides process global time measurement. In order to converge the RT to the CPU consumption, the server-side RTIs could (additionally) use process local measurement, for example using getrusage(2) by (another) JNI module. But, this process local measurement still does not equal the CPU consumption, which has to be taken as thread local run-time. Extraction of CPU consumption due to an EJB invocation can not be provided at the layer of the EJB container, but only inside the JVM as discussed in Sect. 5.4. Therefore, we did not implement the process local measurement of server-side RT either, because it has insufficient significance.

Global synchronization of clocks is not needed because the RT is a difference of local times. As soon as there arises the desire to derive causal relations by temporal order, synchronization of clocks is necessary, for example if no server-side interceptor is registered for EJBs that are invoked by remote client applications (see Sect. 5.3). Examples for current techniques for clock synchronization are the *Global Positioning System* (GPS), the *Global Time Base* (GTB) and the *Network Time Protocol* (NTP). If logical time is sufficient, for example for pure temporal order between several EJB containers, Lamport clock synchronization can be used. This can be realized by an additional interceptor type that inserts scalar logical time into the Invocation objects of all[2] JBoss EJB invocations.

5.2.3. Enhanced Usage

The ordinary use-case consists of one RTI on both client and server side, measuring client- and server-side RT. As shown by List. 2.1 in Sect. 2.1.3, the interceptor chain always consists of interceptors for the provided middleware services, like transactional support or security. The overhead of their execution can be measured by wrapping the interceptor of interest between two RTIs, and storing not only the RT but also the four TS. The overhead of a chosen non-RTI interceptor that is wrapped between $\mathrm{RTI}_{(i)}$ and $\mathrm{RTI}_{(i+1)}$ can be calculated as follows:

$$(\mathrm{TS}_1^{(i+1)} - \mathrm{TS}_2^{(i)}) + (\mathrm{TS}_3^{(i)} - \mathrm{TS}_4^{(i+1)})$$

Even the network delay can be approximated by placing RTIs as last one at client-side and first one at server-side, taking the differences separately: $\mathrm{TS}_1^{(i+1)} - \mathrm{TS}_2^{(i)}$ on

[2]To register interceptors globally for the default configurations of interceptor chains, the $JBOSS/server/default/conf/standardjboss.xml file can be adopted.

the inward way and $TS_3^{(i)} - TS_4^{(i+1)}$ on the outward way. But, the synchronization of clocks is required for this use case.

The interceptor-based measuring approach not only allows to identify hot-spot EJBs and performance bottlenecks, but even to track down the constituents of the RT experienced by a client. Using the interceptor mechanism pro EJB (see Sect. 2.1.3) gives the developer the means to monitor just some selected components, e.g. long-time running ones, which then might be split up in subtasks.

A configuration like this can even be changed at run-time, using the hot-deployment feature of JBoss, facilitating the transparent attachment of the RTIs to any new instances of an already deployed component. The RTI configuration of instantiated EJBs can not be changed with hindsight at run-time. But without the hot-deployment feature the existing EJB would have to be undeployed, waiting for the termination of all instances (or terminating them) and any client-attempts to create new instances would have to be denied, until the bean is again deployed.

5.2.4. Measurement Data

The information gathered by the RTIs from TS_1 to TS_2 and from TS_3 to TS_4 is wrapped by a TO as denoted in Sect. 5.2.2. There is some information like the invoked EJB and method that is the same for all RTIs between a single invocation. To reduce the inserted overhead the information common to all RTIs is only gathered and stored once. Thereby, the resulting DBS structure is normalized. The RTI_n which is nearest to the EJB gathers this metadata of the invocation. The metadata TOs of the several invocations are stored in the DBS to relation raw_CALLMETADATA shown in Fig. 5.3. It consists of the following attributes:

1. CALLID : known to all participating RTIs of one unique EJB invocation
2. JNDI_NAME : the JNDI name of the invoked EJB
3. EJBINSTANCECLASS : the Java class of the EJB Instance
4. EJBOBJECTCLASS : the Java class of the invoked Home or EJB Object
5. METHOD : the name of the invoked EJB method
6. INSTANCEID : the ID of the instance – for stateful session or entity beans
7. ARGUMENTS : the Client's arguments to the invocation

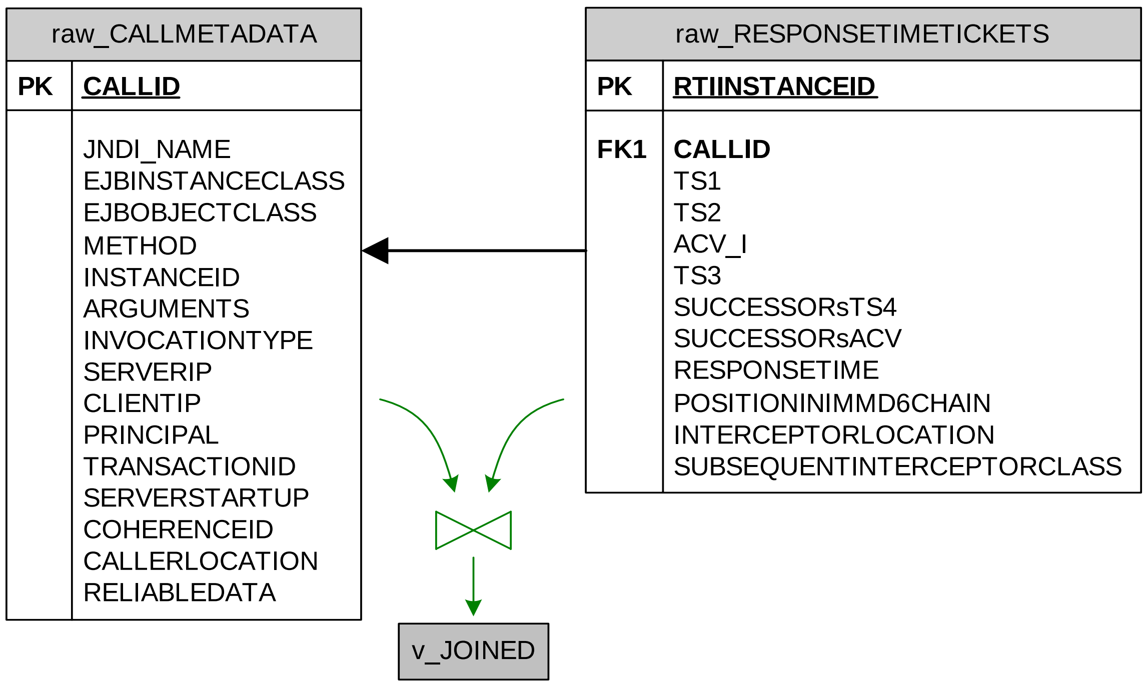


Figure 5.3.: Relational schema of RTI information

8. INVOCATIONTYPE : element of {Home, Remote, LocalHome, Local}
9. SERVERIP and CLIENTIP : the IP address of the container and of the client

The Arguments attribute keeps the serialized arguments of the invocation. Originally, it is of type Object[], because it is the result of the JBoss org.jboss.invocation.Invocation.getArguments() method.

There are just more attributes available in raw_CALLMETADATA with additional information which has not already been discussed:

- PRINCIPAL, TRANSACTIONID, SERVERSTARTUP, COHERENCEID, CALLERLOCATION and RELIABLEDATA

The Principal refers to the *Java Authentication and Authorization Service* (JAAS). Each principal represents an identity for a user.

The TransactionID equals the common TA-ID, as discussed in Sect. 2.1.3. For JBoss the TA-ID is unique only for one instantiation of the application server. In order to generate a *local unique Transaction ID* the SERVERSTARTUP attribute has to be used additionally, storing the start-up time of the server (in *ms* by System.currentTimeMillis(), instead of a μs value for a better convertibility into a Date object and because the server start-up time needs no high accuracy). Finally, a *global unique Transaction ID* can be created, by also including the IP of the server.

The CoherenceID is related to the call dependencies of several interacting EJBs, which is discussed in Sect. 5.3.

The CallerLocation contains either "Client", identifying the invocations that originate from remote[3] client applications, or "EJB", if another EJB is responsible for the specific invocation. The value is extracted by the RTI_1 nearest to the client, which must be a client-side RTI. So, EJBs invoked by external clients must have at least one client-side RTI installed, because we use the non-presence of the JBoss server instance in the JVM of the client-side RTIs of remote client applications to identify remote Client invocations. If another EJB acts as client, a JBoss server instance can be found in any RTIs' JVM that intercepts such invocation. This presence of JBoss is tested by the JVM's system Properties map, which includes JBoss specific values if it is the JBoss server's JVM. We have not found another transparent way to gain the information whether a client is an EJB or a real client application, but it has the advantage that the values are correct even in a system of several distributed JBoss servers.

The ReliableData is a boolean attribute. The storage mechanism, as presented in the following section, allows to extract and store a boolean tag that gives information whether the intercepted invocation occurred during a period where overhead-intensive flushing was performed. Depending on this tag, analysis can reduce or neglect the time values of these TOs. Similar to this boolean attribute would be the number of occurred garbage collector runs during the methods run-time. Unfortunately this information is not available from the JVM.

There exist some limitations for invocations if no client-side RTI is registered: The ClientIP and CallerLocation are null. The JNDIname is null for invocations by remote client applications, but can be provided for invocations between EJBs inside the JBoss-JVM. Exceptional are invocations at the local interface, for which are in

[3]The term "remote" corresponds to the JVM and not to the IP. Even a client application at the same computer as the server is considered as remote to the server.

principle no client-side[4] RTIs applied, but more information is implicitly available: The ClientIP is always provided and equals the ServerIP. Also, the CallerLocation can be provided as “EJB”, but the JNDIname is never available.

These limitations are mostly inherent to the JBoss interfaces, some are due to logical reasons. We strongly advice to install always one server-side RTI. Client-side RTIs are required if call dependencies are of interest, because the CallerLocation is needed. In order to reduce overhead at server-side one could say that for this purpose the client-side RTIs are required only for EJBs that are invoked by remote clients. But, the overhead discussion in Sect. 5.2.7 will show that for client-side RTIs inside the JBoss-JVM (for EJBs that are invoked by other EJBs, via the remote interface) the extra overhead is trivial. Thus, it is safest to register both client- and server-side RTIs for all considered EJBs.

Installing no server-side RTIs is problematical: Invocations at local interfaces are never intercepted by an RTI, and the ServerStartUp, EJBinstanceClass and InstanceID are null. The ServerIP and TransactionID are null for invocations by remote client applications. There could even be problems with the CallID: The generator is seeded time-based and accessed by the RTI$_n$ after the EJB has returned; it passes it backwards to all participating RTI instances and must not be generated by the DBS with hindsight. If server-side RTIs are always present, all CallIDs are generated inside the JBoss-JVM and they are unique. If some remote invocations are not intercepted at server-side, the remote JVM has to generate the CallID and the uniqueness of it is not guaranteed any more. It is not sufficient to composite the CallID with the ClientIP, but also process ID and thread ID would have to be taken into account. The implementation of our CallID generator assumes the existence of server-side RTIs and does not cope with the complexity of such a situation. This assumption also allows to use values of type Long, instead of string-based or composite keys, for efficient joining with the RTI-instance specific data, we will discuss now.

After the RTI$_n$ has stored the metadata TO to the DataCollector, itself and all other RTIs provide the time values in additional TOs. Also, some information is gathered

[4]Again, we want to emphasize that client-side RTIs will also be instantiated inside the EJB container if EJBs invoke other EJBs by their remote interface. This is not uncommon especially if no local interface is provided for an EJB.

that allows to reconstruct the RTI's individual position in relation to the non-RTI interceptors. These TOs are stored in the DBS to relation raw_RTI shown in Fig. 5.3. It consists of the following attributes:

1. RTIINSTANCEID : unique to every RTI instance
2. CALLID : as foreign key, subsuming all RTI instances of one EJB invocation
3. TS2 and TS3 : the TS_2 and TS_3 that represent implicitly the uncleansed RT
4. TS1 and SUCCESSORsTS4 : TS_1 and successor's TS_4 ($= TS_4^{(i+1)}$)
5. ACV_I and SUCCESSORsACV : the $ACV_{(i)}$ and the the $ACV_{2n-(i+1)}$
6. RESPONSETIME : the cleansed RT
7. POSITIONINIMMD6CHAIN : the Index of the RTI in the RTI chain as $i \in [1; n]$
8. INTERCEPTORLOCATION : element of {clientside,serverside}
9. SUBSEQUENTINTERCEPTORCLASS : the Java class of the RTI_i's subsequent interceptor; for some reconstruction of the RTI's position and the partial reconstruction of the interceptor chain

Note that $TS_4^{(i)}$ is only available as successor's-TS_4 of RTI_{i-1} because the TS_4 is taken after the completed storage. This means that $TS_4^{(1)}$ is never available, because there is no other RTI to whom the TS_4 could be passed and then be stored. This unaccounted gap in time is further discussed in Sect. 5.2.7.

Both relations of metadata and time information can be joined naturally, presenting all available information for each RTI. Such a view is available as v_JOINED, according to Fig. 5.3.

5.2.5. Storage Mechanism

As buffered, threaded and asynchronous storage mechanism for the TOs a JMX DataCollector-MBean is used. The buffer reduces overhead at call-time by excluding the cost-intensive persistency mechanism from the return path. The TOs are only transferred to the MBean, and not flushed until the buffer's (manageable) limit is exceeded, which is tested periodically by a separate TimerTask – the interval is again manageable. An alternative to the TimerTask would be to test the exceeding of the limit by the method which allows to give TOs to the MBean. This method would then have to create the flushing thread. But this method does block the interceptor, so the overhead for thread creation (non negligible as denoted in Sect. 5.2.7) would be part of the return path. Therefore, the TimerTask approach was chosen.

The client-invocation is protected from failure of the underlying DBS, because an error can only occur during the flushing to the DBS, ergo in another thread when the client-invocation has finished. The DataCollector implements a publisher that allows subscribers to register. Currently a basic subscriber stores the TOs of interest to the DBS. Adoption of a generic and transparent storage mechanism like Java Data Objects (JDO) [Sun03b] for more possible interceptor TOs can be done by implementing additional subscribers. The comprehensive architecture of our solution is shown in Fig. 5.4.

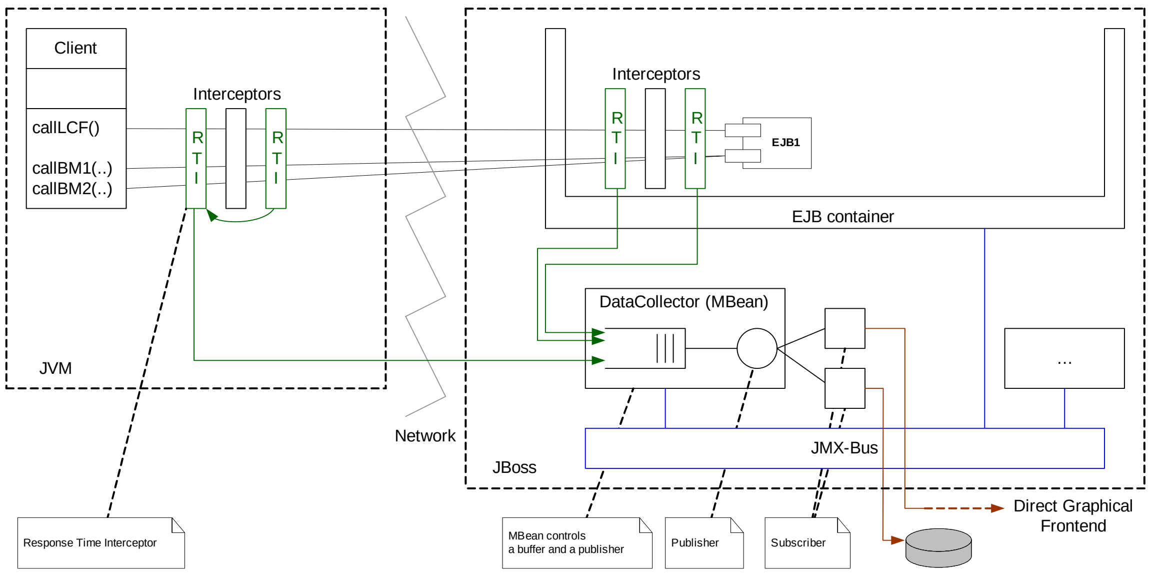


Figure 5.4.: Architecture view at the measurement framework

As mentioned above, the DataCollector allows to extract whether the intercepted invocation occurred during a period of overhead-intensive flushing. The DataCollector keeps a structure that stores information about flushing times. Its size is manageable. If the size exceeds the maximum, “old” flushing-time information will be deleted. The classification of old values is done by a (manageable) relative time boundary and is therefore critical for the informational value of these tags.

Especially in case of a mix of long-time running and short-time running invocations that are intercepted the administrator has to choose that parameter at least as long as a common long-time running invocation approximately takes, and the buffer size has to be set big enough so that the TOs of all short-time running invocations can be buffered and cost-intensive flushing during long-time running invocations is avoided.

The DataCollector is available in a direct and cost-efficient way only at server-side. The client-side RTIs have to transfer the TOs to the server's MBean by some remote call. Because of the static nature of interceptors this has to be done before the invocation is allowed to return to the client. The remote access to the MBean is done by a Proxy-EJB, but as soon as the new JMX Remote API 1.0 [Sun03d] is integrated into JBoss an adoption of the implementation is possible with minimal effort. Another alternative is the transmission based on the *Java Messaging Service* (JMS), but this will be evaluated in further work.

As an optimization of the overhead at client-side, only the last client-side RTI_1 on the return path collects all client-side TOs and transfers all of them using just one remote call to the DataCollector.

Because the measurement of the RT is transparent to the client, a client application that is interested in its RT has to query the DBS. Therefore it needs an unambiguous tuple that represents one of its invocations. Such an interested client has to timestamp directly before (or after) the invocation that roughly equals the TS_2 (or TS_3). Adding its IP address and the used parameters is enough to uniquely query the DBS for the tuples of all RTIs that have participated in this invocation. This approach assumes that either a client-side RTI is present or the clocks of client and server are synchronized. For parallel clients (e.g. multi-threaded ones) from the same IP, the client-side RTIs would have to additionally store the process ID and thread ID to allow these clients an unique request, but this is not done by our implementation, yet. But most clients are not expected to be interested in their RT values after all, because analysis is done by the administrators or developers, and takes place by the stored data in the DBS.

5.2.6. Classification

Because the interceptors are conceptional part of the EJB container, we classify our solution having the EJB container as layer of instrumentation. The interceptors are transparent to client and EJB, and therefore non-intrusive according to the participants view. The source code of the EJB container is not changed, so the source code oriented view shows the non-intrusive type. The interceptor approach uses CPU and memory of the server system and the network behavior is changed, because the RTIs increase the size of the transferred data by passing ACV and CallID information for example.

Also, the deployer has to adapt the jboss.xml deployment descriptor – referring to the deployer view that is not included in the short classification scheme (Sect. 3.2.3) provided below.

Layer of Instr.	Participants	Source code	CPU	Memory	Network	IO
EJB container	NI	NI	I	I	I	NI

5.2.7. Overhead

A software approach to measurement has always some overhead being never accountable exactly. In addition to the efforts to minimize the overhead and to maximize the accuracy the following influences have to be considered[5]:

On server-side the JNI module should always be deployed, otherwise the accuracy is reduced to System.currentTimeMillis(). Especially on some client systems it could happen that the JNI module is not deployed or correctly installed, thus the accuracy of the data from remote client-side RTIs has always to be considered carefully during analysis.

Due to the RTIs the client perceives a greater RT compared with a situation without the RTIs. The accountable overhead due to the RTIs can be computed for each RTI and invocation individually by the stored data as $(TS_2 - TS_1) + (TS_4 - TS_3)$.

[5]For the hardware configuration refer to Sect. 2.1.4. (Abstract: a standard PC system with an AMD K7 1400 MHz processor and 1GB SDRAM, running the SuSE 8.2 Linux distribution.)

In the remainder of this section we will provide empiric data for the overhead, using absolute values. We ignored values of the few invocations at objects whose class type is used for the first time in the JVM, because they are drastically higher than all others – we assume that this is due to the JVM's ClassLoader that has to load the class for the first time, presumably from the storage device. The given values are not based on enough statistical basis and analysis to be taken for granted, but it will provide an overview and estimation of the overhead. For example, we experimented with several numbers of registered RTIs per invocation. We carefully examined and interpreted the resulting data and outline them below.

If there are installed many RTIs, an RTI that is not the first or last one introduces usually overhead about $60\mu s$. Most overhead is caused by the first and last one and we discuss them separately below. From the 60 μs, a portion of $30\mu s$ is introduced on the inward way as well as $30\mu s$ on the outward way, according to the control-flow of an invocation in relation to the interceptor.

The gathering and storage of the metadata TO by the last RTI takes about $400\mu s$ additionally. In the common scenario with one client- and one server-side RTI registered, the (last) RTI at server-side causes overhead about $460\mu s$. If the invocation was caused by a real client application, the first RTI at server-side ($\mathrm{RTI}_y : y \neq 1$) will produce an additional overhead of $550\mu s$ on the inward way, because the values that are passed from the remote client-side RTIs need deserialization. So, if a single RTI is registered at server-side, it will cause approximately $1010\mu s$ during an invocation by a real client, and $460\mu s$ if the caller is another EJB.

Before we can give the values for client-side RTIs, we have to discuss another matter: The TS_4 of RTI_1 being the last returning one cannot be stored. The storage takes place between $\mathrm{TS}_3^{(1)}$ and $\mathrm{TS}_4^{(1)}$, and there is no other RTI to whom the $\mathrm{TS}_4^{(1)}$ or $\mathrm{ACV}_{2n-(1)}$ could be passed. So, there exists an unaccountable overhead $\mathrm{O}_{\mathrm{unacc}}^{(1)} := \mathrm{TS}_4^{(1)} - \mathrm{TS}_3^{(1)}$ for each invocation.

If the RTI_1 is at the side of the client application, $\mathrm{O}_{\mathrm{unacc}}^{(1)}$ includes the transfer of the client-side TOs to the DataCollectorProxy by a remote call, and its overhead is not negligible. Measurements have shown that its duration is in units of $10ms$. For this reason, and the reason that the overhead intensive storage increases the client-perceived RT, the remote storage to the DataCollectorProxy is executed by a separate thread. Thereby the $O^{(1)}_{\text{unacc}}$ is reduced. The creation of a thread takes about $300\mu s$ ($289.325\mu s$ as the average of 10000 tests[6]). Therefore the first client-side RTI will only produce (unaccountable) overhead of $330\mu s$ on the outward way. Because interceptors are instantiated anew for each invocation, new invocations by the client application are independent of an already running storage-thread, because the new RTI will create an own thread. Temporarily timestamping the relevant code-segments and run-time output of the differences confirms that this calculation ($300\mu s$ for the thread creation plus the common $30\mu s$) is correct.

An additional aspect to consider for analysis is that the TS_3 of the RTI_1 represents nearly the same time the client would have received the result if the RTI would not be present.

For a client-side RTI_1 in the JBoss-JVM, as appear for invocations between EJBs, the $O^{(1)}_{\text{unacc}}$ is approximately $30\mu s$, because the MBean can be accessed directly and no thread is needed. This reduction of unaccounted overhead for server-local invocations is of special interest for RT-calculations in invocation chains as will be discussed in Sect. 5.3. But, this consideration has an exception according invocations at local interfaces if only one server-side RTI is applied (which is the default): $O^{(1)}_{\text{unacc}}$ equals the $430\mu s$ of $\text{RTI}_n = \text{RTI}_1$ at the outward way. If a reduction of unaccounted overhead is essential for a special analysis it is possible to reduce $O^{(1)}_{\text{unacc}}$ even for this case by registering two server-side RTIs, so that the outward overhead of RTI_n is not part of the $O^{(1)}_{\text{unacc}}$ anymore. The additional server-side RTI introduces more overhead, but only the marginal $60\mu s$. The particular kind of analysis has to decide which kind of overhead is more important for its purpose.

In order to give an overview over the approximated overhead, we provide two tables, that summarize the above values for our run-time environment: Table 5.1 (client-side RTIs) and 5.2 (server-side RTIs) show an approximation of the overhead that is introduced for the various kinds of position and invocations. Table 5.3 accumulates the values for the three possible scenarios. The values distinguish between “in” and “out”, which is the overhead on the inward and outward way of the invocation.

[6]The methodology and source code for this statistics is given in the appendix A.3.

OVERHEAD **Client-side RTIs**	First RTI_1 Remote Client	EJB as Client	Any other RTI
Local, LocalHome	—	—	—
Remote, Home	in:30μs, out:**330**μs	in:30μs, out:30μs	in:30μs, out:30μs

Table 5.1.: Approximation of the empiric overhead for client-side RTIs

OVERHEAD **Server-side RTIs**	Remote Client ($\Rightarrow$ first $RTI_y \neq RTI_1$)	Last RTI_n	Any other RTI
Local, LocalHome	—	in:30μs, out:**430**μs	in:30μs, out:30μs
Remote, Home	RTI_y: in:+**550**μs (additional)	in:30μs, out:**430**μs	in:30μs, out:30μs

Table 5.2.: Approximation of the empiric overhead for server-side RTIs (If the first server-side RTI is not the RTI_1 there exist client-side RTIs)

SUMMED OVERHEAD	No. of RTIs	Accounted Overhead		$O^{(1)}_{\text{unacc}}$	Overhead
Real client invokes EJB	2	$(30 + 580 + 430 =)$	1040μs	330μs	$\approx 1.5ms$
EJB invokes EJB (Remote)	2	$(30 + 30 + 430 =)$	490μs	30μs	$\approx 0.5ms$
EJB invokes EJB (Local)	1	$(30 =)$	30μs	430μs	$\approx 0.5ms$

Table 5.3.: Approximation of the empiric overhead per invocation; with the standard configuration applied: one client-side and one server-side RTI

The JNI mechanism itself introduces additional non-accountable overhead. Tests showed that the JNI call to the Linux' gettimeofday(2) lasts about $0.970\mu s$ as the average of 10000 tests[7] – the pure gettimeofday(2) call took $0.254\mu s$ as discussed in Sect. 2.3.4.

Calculating the ACVs, and putting them into the environment maps passed forward or backward, must inevitably be done after the TS_2 and TS_4 are taken. Temporarily timestamping the relevant code-segments showed that each RTI adds about $5\mu s$ of not accountable overhead. If necessary, this knowledge can be used during analysis for a more accurate interpretation of the data.

For an initial client, the gap between its receiving of the result from the DynamicProxy and the previous return of RTI_1, as discussed in Sect. 5.2.1, was measured to be shorter than $1ms$.

[7]The methodology and source code for this statistics is given in the appendix A.3.

5.2.8. Configuration at Run-time

At server-side it is possible to control the behavior of the RTIs by the implemented InterceptorController-MBean. The initial configuration is either read from the $JBOSS/server/default/conf/interceptorconfig.properties file, or done by default values. At the moment, the MBean is the interface to the static InterceptorConfig object that influences the behavior by some boolean values. In the default configuration the parameters of the invocation are not stored, because we use the framework to test video-related invocations where the video-frames as parameter generate too much data if they are stored, and we have not yet the intention to analyze these invocations in relation to the frames.

The jboss.xml deployment descriptor allows to activate interceptors at EJB-level granularity (although the interception itself is done per method invocation). The ability to select the interception at method-level is surely preferable to the EJB-level: An MBean like the InterceptorController is able to provide an interface allowing the transfer of a string-based filter for method-names into the JBoss-JVM. These strings can, for example, be applied as regular expression to a run-time decision mechanism, that decides whether the RTIs of the invocation collect and calculate the data or not. Thereby, allowing the (de-)activation of the interception at method-level granularity. A similar filtration mechanism could be applied to decide whether the values of the parameters have to be stored, for example by the types of the parameters.

Client-side interceptors in a remote JVM can obviously not be configured by a JBoss-MBean. At the moment, the management by a properties-file is possible – it has to be available at the client's classpath. In a large environment with complex and long-running clients the remote control of client interceptors has to be possible. For example JMS or an self-implemented listener-thread can provide the necessary service, but we have not implemented this.

Another feature that does not exist at the moment would be the support of dynamic adding/removal of interceptors at run-time by an API per EJB, without the necessity to change the deployment descriptor and to redeploy the bean. JBoss does not support this, yet.

5.2.9. Reusage of the Environment

Until now we have introduced only interceptors of type "RTI" into the call chain. But these are only one type of interceptors that could be instrumented in the context of the measurement of NFPs. To get a complete overview of an EJB's behavior we can think of interceptors introducing Jitter or simulating time-outs, or interceptors that implement fault injection by manipulating the parameters and the result object. All such interceptors can be thought of as a logical chain inside the entire interceptor chain and we call it *logical interceptor chain* (LIC) for NFP measurement.

Because a mechanism to abstract from the overhead introduced by our RTIs is necessary anyway, it seemed obvious to capsulate the timestamping and the communication of the overhead up and down the chain into a reusable Java object: the GeneralIMMD6handler that belongs to the de.uni_erlangen.inf6.test_stand.interceptor.util package. It is possible for the developers of other NFP related interceptors to take part in the LIC, gaining automatic timestamping and abstraction of their overhead concerning the calculated RT.

The functionality and the API is provided in the appendix in Sect. A.1.5.1, together with templates for the implementation of other LIC-interceptors in Sect. A.1.5.2.

Usage of the GeneralIMMD6handler ensures that the last LIC-interceptor creates a metadata TO, conforming to raw_CALLMETADATA. The four TS are taken and the ACVs are calculated and passed along the inward and outward way. Access is provided to information like TSs and ACVs as well as the transparent and location-independent access to the DataCollector-MBean, either directly at server-side or by the Proxy at client-side.

The initialization of TS_1 is implicitly done by the constructor of GeneralIMMD6handler. But because there is an unaccounted time gap of some microseconds between the call to the constructor and the timestamping as first command inside it, we have made it possible to take the TS_1 externally, passing it to the constructor. The programmer has to ensure, that the provided TS_1 was created by NativeTimeProvider.currentTimeMicros(), and for example not by System.currentTimeMillis(). We measured this time gap (on our environment) to be about $3.891\mu s$ for 10000 tests[8].

[8]The methodology and source code for this statistics is given in the appendix A.3.

We also tried to insert lots of arbitrary other code in the loop, in order to eliminate caching-effects of the JVM, but it had insignificant influence. Regarding the principle of abstraction the implicit approach is to be favored, but for a better coverage of the overhead the alternative constructor is provided. Both approaches are presented by the templates in appendix A.1.5.2; the implicit approach in Lst. A.4, and taking the TS_1 externally in Lst. A.5.

The most important thing to know about the LIC is the design of the *back-passing mechanism* – the way an RTI_b can pass information to an $\mathrm{RTI}_a : a < b$ on the outward way. On the inward way it is easy to pass custom information from RTI_a to RTI_b by the Invocation object, that is passed along the chain as parameter of the invoke() method. JBoss provides no equivalent on the outward way. The result object of the EJB Instance itself is returned by each interceptor directly, as shown in Sect. 2.1.3.

This situation allows to replace the result object of the EJB Instance temporarily by some kind of Map that contains the result object plus additional information objects. This replacement has to be undone before the invocation is returned to the client application. In the LIC the RTI_n nearest to the EJB receives the result object and wraps it in a BackpassedInformationMap that extends a HashMap. The RTI_1 is the last member of the LIC before the result is returned to the client, so it discards the Map, unwraps the result object and returns it.

This back-passing mechanism is inherently incompatible to any other logical interceptor chain from other developers that tries to pass information on the outward way in a similar fashion. But common interceptors like the JBoss standard interceptors have a static behavior, that means they are unpaired and need not to pass information to any other interceptors. So we don't see this as major problem at the moment. The mechanism is also incompatible to interceptors that analyze the result object if this interceptor is positioned between RTI_1 and RTI_n. But again no JBoss standard interceptor behaves in such a way, and if there are affected custom interceptors they just have to be positioned compatibly. For all other interceptors this mechanism is without concern.

The "clean" solution would need the JBoss objects that conclude the interceptor chain at client- and server-side to implement such a wrapping. Thereby every interceptor could pass information on the outward way in a standardized fashion, as is possible by the Invocation object on the inward way. The necessity of such a modification of JBoss is already acknowledged by the JBoss developers, and will possibly be available soon.

5.2.10. Summary

This section has described how the interceptor pattern can be applied for response time measurements in the JBoss application server. Following a discussion of several possible layers of instrumentation, the application of our concept has been explained in detail for response time measurement.

Basically, it is possible to not only measure client- or server-side response times, but also to determine delays introduced by standard interceptors in the call chain wrapping them with RTIs.

The data structure, the attributes' semantics and the storage framework has been described. Among these descriptions we have mentioned several limitations due to the JBoss interface. The classification of the RTI framework was provided according to the scheme from Sect. 3.2.3.

Because a software solution always introduces some overhead, we have shown how to calculate cleansed response times using correction values (ACVs) and we have discussed the overhead due to the RTIs.

Finally, the reusage of the implementation for more types of interceptors has been explained, taking into account the mechanism for back-passing information between interceptors.

5.3. Call Dependency

In the preceding section we focused on an individual invocation. Now we assume the existence of several components interacting to fulfill a particular service. In an assembly of interacting components, not only the performance of an individual component is actually what matters, but also the invocation frequentness and dependencies.

First we have to define our vocabulary and notation, then we introduce an algorithm that extracts the call dependencies by the temporal order of the invocations. Finally, we describe our implementation and discuss some of its limitations.

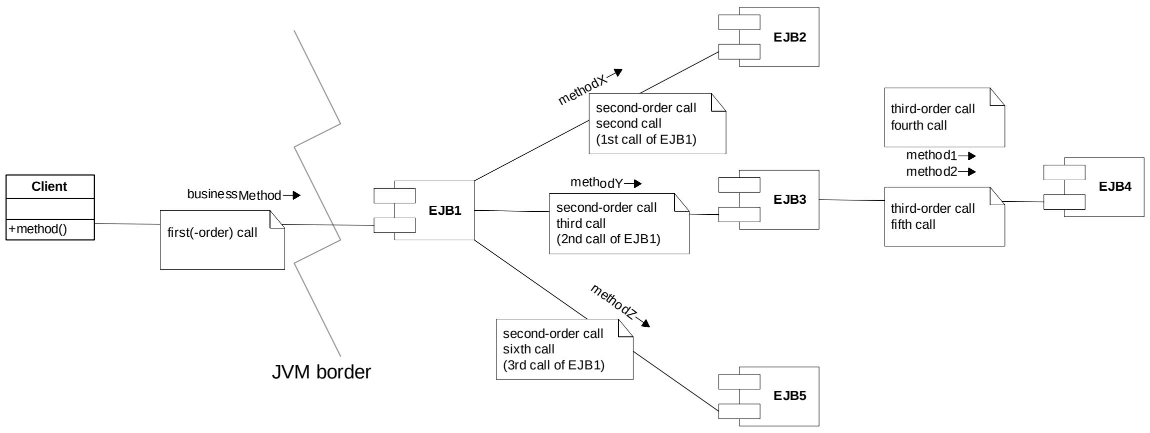
5.3.1. Definitions

Our run-time scenarios always start when a *client application* from the outside of the container JVM calls an EJB. Concerning the information gathering we do not specialize on calls to the EJB Object but also consider calls to the Home Object.

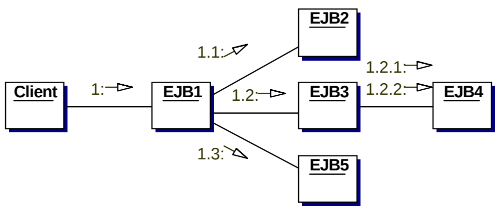
The bean that is initially invoked is called *first EJB* and the invocation itself is the *initial invocation* or *first invocation*. For example, a simple client that creates an EJB using the Home Interface, calls one business function at the Remote Object, removes the EJB and terminates, has made three initial invocations in our terminology. The first EJB will possibly invoke *second-order EJBs* and the invocations are named *second-order invocation*. All subsequent EJBs that are invoked by the second-order EJBs are named likewise.

All invocations and components that are directly related to one specific initial invocation form a *coherency scope*. All invocations and EJBs of a coherency scope are part of an invocation tree that could be represented as an UML collaboration diagram. Figure 5.5 shows a simple example of an invocation tree with our nomenclature, where the temporal order of the invocations is from left to right and from top to bottom – the calls to the Home Objects are omitted due to simplification.

We number the components and the invocations successively in the temporal order of their appearance. Note that any intermediate component that is invoked and invokes other components is in the role of both a server and a client in turns. So



(a) A simple example of an invocation tree and our nomenclature



(b) The corresponding UML collaboration diagram representation

```
Client.method(..)
    +-> EJB1.businessMethod(..)
        +-> EJB2.methodX(..)
        +-> EJB3.methodY(..)
            +-> EJB4.method1(..)
            +-> EJB4.method2(..)
        +-> EJB5.methodZ(..)
```

(c) The corresponding invocation tree in ASCII representation

Figure 5.5.: Invocations and components of a coherency scope (the calls to the Home Objects are omitted due to simplification)

only the term "first client" or "real client" refers to the remote[9] client application. The general interpretation of "client" depends on the context.

For every invocation we distinguish between client-side and server-side, because invocations by the remote interface can be intercepted on both sides. Invocations by the local interfaces are intercepted by JBoss only at server-side. Again it has to be made clear that the distinction of remote invocation and local invocation does not depend on the location of client(-EJB) and server(-EJB) or on their sharing of the same JVM, but only on the interface that is exposed by the EJB. If there is no local interface defined or used, EJBs inside the same server will also invoke each other by the remote interface.

[9]The client application is always remote to the server's JVM, but not necessarily remote in relation to the IP address.

The challenge lies in identifying all invocations and EJBs that belong to a specific coherency scope as well as their temporal order and causal relationship.

In formulas we use the following naming convention (the initial client is not part of the sets, because the information is always related to the invoked EJB and method):

- Entities $E_e : e \in [1; \sharp\mathrm{E}]$ are the individual invocations (method instances) of the EJBs inside a single coherency scope. E_1 is the initial invocation and $\sharp\mathrm{E}$ is the number of invocations inside the coherency scope. Two types of entities are distinguished:
  - Leaf nodes $E_l : l \in \mathrm{L} \subset \mathbb{N}$
  - Middle nodes $E_m : m \in \mathrm{M} \subset \mathbb{N}$

    $[\ \sharp\mathrm{E} = \sharp\mathrm{L} + \sharp\mathrm{M}\ ;\ \mathrm{L} \cap \mathrm{M} = \emptyset\ ;\ \mathrm{L} \cup \mathrm{M} = \{1, 2, \ldots, \sharp\mathrm{E}\}\ ]$
- Relative navigation – corresponding to a specific E_e:
  - Child $E_e.Ch_k : k \in [1; \sharp Ch_e]$ represents the kth invocation, that is instantiated by an E_e and $\sharp Ch_e$ is the number of children of E_e
  - Father $E_e.Fa$ represents the caller of E_e

    $[\ E_e.Ch_k$ is an entity $E_x : x > e$, and $E_e.Fa$ is an entity $E_y : y < e\ ]$
- Interceptor $E_e.I_i : i \in [1; n]$ represents the RTI$_i$ of the invocation E_e

Interceptors finally carry all the gathered information, discussed in Sect. 5.2, for example $E_e.I_i(\mathrm{RT})$ as the cleansed RT of the interceptor at position i inside the interceptor chain of invocation e of some EJB.

5.3.2. Temporal-order Algorithm

The following algorithm concludes the call dependencies in the temporal order of the invocations. For the the initial explanation of the algorithm we limit the possible scenarios to the following conditions: All EJBs in the system are instrumented to be intercepted by at least one RTI, and no more than one initial client invocation takes place at the same time, so that no other EJBs are invoked during its run-time which would not be related to the coherency scope of the client invocation of interest. We will discuss these limitations after the algorithm in Sect. 5.3.5.

In an environment simplified like this, all EJBs that belong to the coherency scope must have been called between the TS$_2$ and TS$_3$ of the initial invocation. Figure 5.6

shows an exemplary scenario, in which all invocations are numbered by their temporal occurrence. Parallel invocations can not occur because of the EJB specification's prohibition of thread-usage. Therefore, we can order these invocations temporally, as shown at the right side of the figure.

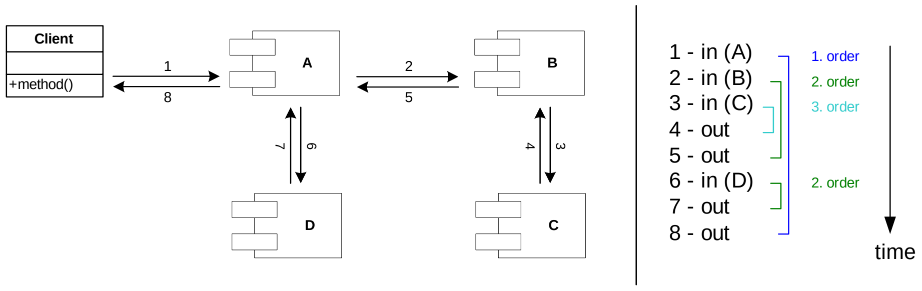


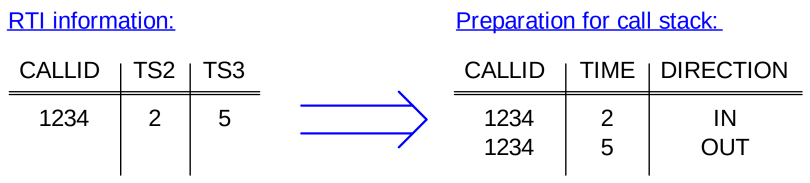
Figure 5.6.: Invocations and the corresponding call stack

If the "in" events are interpreted as "push" and the "out" events interpreted as "pop" a stack-view is created. In this stack the caller of an EJB is always available as the top-most element. Therefore, the call dependencies and the invocation tree can be constructed during a stack-like parsing of the temporal ordered in and out events.

The in and out events are available as the TS_2 and TS_3 of any RTI that intercepts the invocation. If there are several RTIs for one invocation, always the first can be used. Synchronization of the clocks is not needed, because all RTIs are in the same JVM, with one exception: The client-side RTIs of the remote client invocation. But if at least one server-side RTI is registered for the remotely invoked EJB, the RTI_n of the initial client invocation is guaranteed to share the clock with all the second-order invocation RTIs.

5.3.3. Implementation

The temporal arrangement of the RTIs' information can be done inside the database system. But, each RTI_1 tuple has to be available twice: Once with the TS_2 as the time information and once again with TS_3 as the same relational attribute for easy sorting. Therefore the information whether it was TS_2 or TS_3 has to be stored in an extra attribute, as shown in Fig. 5.7.



CALLID	TS2	TS3
1234	2	5

CALLID	TIME	DIRECTION
1234	2	IN
1234	5	OUT

Figure 5.7.: Example transformation of TS_2 and TS_3 into separated tuples

We use two helper relations h_In and h_Out in order to transform one RTI tuple in two "direction-tagged" time-tuples. Both helper relations have the same attribute Direction and only one (!) tuple that contains "in", or "out" respectively. The Cartesian product of h_In with v_Joined projects TS_2 into the Time attribute; the Cartesian product of v_Joined with h_Out does the same with TS_3. (The relation v_Joined was introduced in Sect. 5.2.4 and contains all information from an RTI.) All RTIs except the first one are filtered out by the selection of PositionInIMMD6chain = 1. Now, we have two relations: One with TS_2 as Time value showing "in" as direction, and the other with TS_3 as Time value and "out" as direction. The union of both relations can be ordered by the time attribute and then equals the temporally ordered invocations.

This concept is visualized in Fig. 5.8. The actual SQL commands are available in the appendix A.2.1 as Lst. A.9. RTI_n of the client invocation has to be a server-side RTI if synchronization of clocks shall be avoided and must be used instead of RTI_1. The figure does not include the usage of RTI_n for the one remote client invocation per coherency scope, but the listing in the appendix does. Instead of RTI_1, we could always use RTI_n, but because the number n of RTIs may be different for the several EJBs, it is more efficient to use the constant PositionInIMMD6chain = 1 as selection as far as possible.

Having prepared the invocations by temporal order, the coherency scope can be identified and manifested. The CoherenceID attribute of the RTIs' tuples are initialized at run-time with "-1". They carry this attribute as a preparation for a possible future run-time detection of the coherency scope. All initial client invocations carry the information CallerLocation = 'CLIENT', thus the client invocation of interest can be selected. Using its TS_2 and TS_3 values and the temporal order view all CallIDs of the

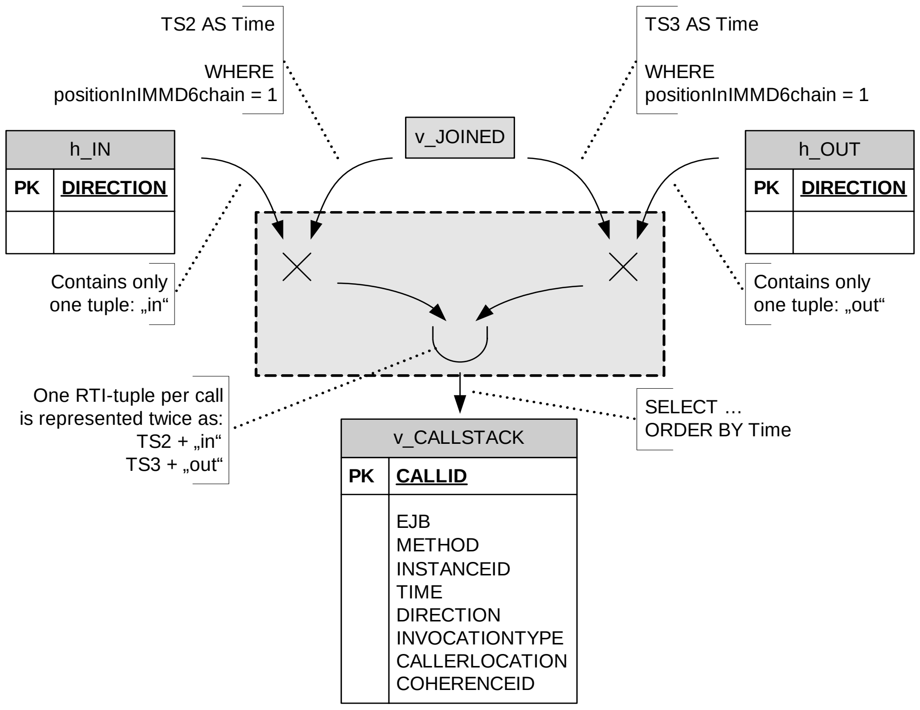

TS2 AS Time
WHERE positionInIMMD6chain = 1
TS3 AS Time
WHERE positionInIMMD6chain = 1
v_JOINED
h_IN
PK DIRECTION
h_OUT
PK DIRECTION
Contains only one tuple: „in“
Contains only one tuple: „out“
One RTI-tuple per call is represented twice as: TS2 + „in“ TS3 + „out“
SELECT ... ORDER BY Time
v_CALLSTACK
PK CALLID
EJB
METHOD
INSTANCEID
TIME
DIRECTION
INVOCATIONTYPE
CALLERLOCATION
COHERENCEID

Figure 5.8.: Visualization of the SQL operations that are needed for the temporal order of “in” and “out” events on basis of the RTI-provided data

coherency scope can be selected, and an artificial CoherenceID can be generated for them.

Being able to select all tuples from the temporal order view that are part of the coherency scope of interest, the call dependencies can be revealed by the construction of the invocation tree. The tuples are fetched one-by-one and all information, except the Direction, form the abstract representation of a node inside the invocation tree.

An exemplary cutout of Java for the parsing algorithm is given in Lst. 5.1. The first tuple to be fetched is the one that holds information on the call of the client application to the first EJB. Every node contains the given information as well as a reference to its caller's node and an array of nodes of the EJB methods it invokes. The primary key for a node (= entity E_e) is not only the EJB Instance, but also the invoked method. Thus, several calls to the same EJB are arranged correctly. Several instances of the same type of EJB can be distinguished by the InstanceID, that is provided by the application server as discussed in Sect. 5.2.4.

```
/* info: contains the RTI's information
 * direction: either "in" or "out"         */
public void parse(InvocationInformation info, String direction) {

    if (direction.equalsIgnoreCase("IN")) {
        if (stack.empty() == true)
            _Root = InvocationTree.makeNode(info);
            // _Root is the root-node of the invocation tree
            stack.push(_Root);
            return;
        }
        TreeNode topNode = (TreeNode) stack.peek();
        // top is on the one hand the top-element of the stack
        // and on the other hand part of the tree that is
        // spanned by _Root

        TreeNode newNode = InvocationTree.makeNode(info, topNode);
        topNode.addChild(newNode);
        // every TreeNode knows its Father and has an array
        // of children

        stack.push(newNode);
    }
    if (direction.equalsIgnoreCase("OUT"))
        stack.pop();
}
```

Listing 5.1: Algorithm for parsing the temporal ordered tuples and generation of an invocation tree

The root of the tree is initiated with the data for the initial client invocation. All succeeding “in”-tuples are formed into a node by storing a reference to the top-node of the stack (the current node has not yet been pushed), because the top-node equals the caller of the current invocation. The top-node is informed about the existence of his new child by addChild(..), and it will add the reference of the child to its array of child-invocations. After this, the current node is pushed onto the stack. The “out”-tuples are not formed into nodes. They carry no other information than to assure the stack’s correct height, by removing the top-node with a pop-operation.

The Java class that fetches the tuples from the DBS is the Analyzer in the de.uni_erlangen.inf6.test_stand.jdbc.analyzer package, at the moment it contains most functionality to access the DBS, to create the helper tables and view, to generate the CoherenceIDs and to select and fetch the tuples of a coherency scope. The tree is available as InvocationTree in the de.uni_erlangen.inf6.test_stand.adjacency package. The parsing algorithm is part of the InvocationTree and it is feeded by the Analyzer. The Analyzer realizes a filter mechanism that instruments the DBS and that is based on method-names or invocation types (Home, Remote, LocalHome, Local).

The InvocationTree.toString() method allows a simple print-out of the invocation tree, shown in Lst. 5.2. There is the possibility to search in the tree for the first occurrence of a node/invocation by searchSubNode(), and the tree can be cloned.

```
----------- TREE ----------
Coherence ID ---: 1076177330677
EJB ------------: example.A_Session
Method ---------: businessFunc
CallID of Client: 1076177160075
Filtered Methods: create, remove
Filtered Types -:
---------------------------
example.A_Session.businessFunc(..)
    +-> example.B_Session.f1(..)
        +-> example.C_Session.f2(..)
    +-> example.D_Session.f3(..)
-------------END-----------
```

Listing 5.2: Simple ASCII representation of the invocation tree

It has to be pointed out that there is never available any special information about the initial client application. The information as provided by the RTIs are related to the invoked component, because there is no programmatic reference to the client. The only information about the initial client application is its IP and the fact, that it is actually a client application and not another EJB, by the fact that no JBoss is present in its JVM (see Sect. 5.2.4). Thus, the information of the root-node corresponds to the first EJB.

5.3.4. Adjacency Matrix

Relating to the COMQUAD project, the *assembly matrix* [MM04, 12f.] describes the frequentness of invocations between components. For an individual coherency scope this matrix equals an adjacency matrix of the edges that are implicitly represented by the invocation tree.

A generic implementation is provided by the AdjacencyMatrix in the de.uni_erlangen.inf6.test_stand.adjacency package. The matrix is always square and allows to register edges successively. The individual entries in the matrix are incremented for each registered edge and the matrix grows its dimensions dynamically if an edge with a new entity is registered. The accumulated number of callers or callees for an entity can be calculated, and the matrix can be pivoted, that means the axis of callers and callees are exchanged, as is the semantics of the several query methods.

The InvocationMatrix that extends the AdjacencyMatrix provides the functionality to store metadata information like CoherenceID and CallID of the initial client invocation, as same as the applied filters during creation. The granularity of the entities that span the dimensions can be configured to be at component-level or at method-level.

There exist two ways to create the InvocationMatrix. Either using the InvocationTree.generateInvocationMatrix() method of an existing InvocationTree, or directly from the DBS with an slightly changed parsing-algorithm of the ordered tuples by the Analyzer.generateInvocationMatrix() method. The Analyzer can store the InvocationMatrix instances to the DBS, so that a new generation can be saved. Again a simple textual print-out is provided by the InvocationMatrix.toString() method (shown in Lst. 5.3).

5.3.5. Discussion

The constraint that all EJBs that could participate in an invocation chain have to be intercepted by at least one RTI can be fulfilled. Most EJBs that interact together are normally packed in a single jar-file with one deployment descriptor. Other EJBs that are not included in the jar-file have to be referenced by the deployment descriptor using the <ejb−ref> tag. Thus, all candidate EJBs can be found out. An automation of this process is not implemented yet, but there will be available a Comquad tool that allows to easily register RTIs for the individual EJBs of a selected jar-file by means of a graphical interface, soon.

The other constraint that no more than one initial client invocation takes place at the same time is very restrictive and prevents the usage of the proposed algorithm

```
---------- MATRIX ---------
Coherence ID ---: 1076177330677
EJB ------------: example.A_Session
Method ---------: businessFunc
CallID of Client: 1076177160075
Filtered Methods: create, remove
Filtered Types -:
---------------------------
Index 1 corresponds to Entity: example.A_Session.businessFunc(..)
Index 2 corresponds to Entity: example.B_Session.f1(..)
Index 3 corresponds to Entity: example.C_Session.f2(..)
Index 4 corresponds to Entity: example.D_Session.f3(..)
---------------------------
Caller  1:   0   1   0   1 |   2
Caller  2:   0   0   1   0 |   1
Caller  3:   0   0   0   0 |   0
Caller  4:   0   0   0   0 |   0
---------------------------
             0   1   1   1
------------END------------
```

Listing 5.3: Simple ASCII representation of the adjacency matrix

for invocation-information from productive systems. One possible solution would be that the application server provides the CoherenceID at run-time. Because invocations from one single coherency scope can always be ordered temporally, and the constraint only exists to be able to demarcate the several coherency scopes by the events of real client invocations. If the invocations can be selected by their run-time generated CoherenceID, the algorithm can be applied without any limitation. Unfortunately the JBoss server does not provide any CoherenceID.

An information provided by JBoss that is similar to the coherency scope is the transactional scope represented by the TA-ID, which was introduced in Sect. 2.1.3 and which is stored as TransactionID attribute. There exist naturally several TA-IDs inside the same coherency scope, and the coherency scope envelops all the participating transactions, with no transaction being part of more than one coherency scope. Some visualization of the difference between the transactional scope and the coherency scope is provided in the appendix A.2.1 as Fig. A.7.

It is possible to force the TA-ID to equal the CoherenceID by means of setting the transactional behavior of all EJBs to "Required", having the following semantics: If a transaction is already running for the caller, e.g. some other EJB, the bean joins in on that transaction – if no transaction is running for the caller, e.g. for client applications, the EJB container starts a transaction for that invocation.

Changing the transactional behavior of a bean does with some probability have influence on its performance characteristics and should definitely not be a long-term solution. But to recognize the possibility of such instrumentation could prove useful, as long as the application container does not provide the CoherenceID. Also, the existence of the Transaction ID gives evidence that the desired run-time generation of CoherenceIDs should be possible to a facility similar to the transactional service provider, either implementing such new facility for JBoss or enhancing the existing transactional one.

The alternative to run-time generation of the IDs is a more complex and statistical analysis of the temporal order, if there take place several client invocations at the same time. Aguilera et al. have developed an algorithm to detect causal relations for “RPC-style communication” [AMW$^+$03, 4] and another for “free-form message-based communication”[AMW$^+$03, 6]. They are based on statistical methods and probability and can not guarantee the correctness and integrity of their results.

To improve the effectivity of such statistic analysis, a two-phase evaluation can be applied. In the first phase the EJBs are invoked in a dedicated run-time environment where the compliance to the heavy constraint can be guaranteed, the coherency scopes can be separated and our simple temporal-order algorithm is used to extract the call dependencies. Testing the call dependencies in relation to a various broad of parameters allows to acquire knowledge about the interdependencies of the beans. During the second phase where the EJBs are deployed and invoked in a productive system, this knowledge can be used to support statistical analysis of call dependencies. For the generation of appropriate client invocations during the first phase a test-driver is needed. Empirix Bean-test [Bea] is a commercial and expensive tool that can be used, but the COMQUAD project is developing an own test-driver for EJBs at the moment, for several purposes.

Not discussed yet is the correlation of several client invocations and their accompanying coherency scopes to one and the same client application. This is not possible with the currently available information, because there is no programmatic access to the client by the client-side interceptors. One consideration is to implement a JNI library that requests the OS for the process ID, which would be requested by the client-side RTIs. This would have the advantage that even multi-threaded clients are recognized as a whole application, and we could even identify clients as multithreaded ones, because their coherency scopes overlap one another temporally (presumed that the CoherenceID can till then be run-time generated).

5.3.6. Calculations

Inside a coherency scope the RT provided by the RTI$_n$ does not equal the run-time T$_{\text{run}}$ of the isolated EJB method, see Fig. 5.9. It does include the invocations to other beans and the overhead of the RTIs that intercept these second-order invocations.

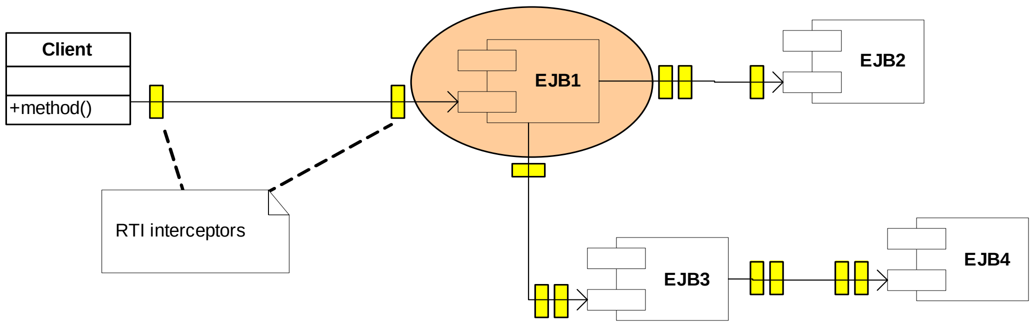


Figure 5.9.: An isolated node in the middle of its coherency scope in relation to the RTIs

The RTI$_1$ of the second-order invocations provide the information to approximate T$_{\text{run}}$ of the bean's method. The interval between TS$_1$ and TS$_4$ of all the second-order invocations' RTI$_1$ approximate the run-times that do not belong to the method itself. Because the $\text{TS}_4^{(1)}$ is not available in the DBS, by means discussed in Sect. 5.2.4, it has to be substituted by $\text{TS}_3^{(1)}$. For all second-order EJBs the time-gap $(\text{TS}_4^{(1)} - \text{TS}_3^{(1)})$ is roughly $30\mu s$, as discussed in Sect. 5.2.7. The formula is not recursive, because every node has knowledge about the number of its children and it has direct access to their data.

$$\text{T}_{\text{run}} \text{ of any node } E_e \quad := \quad E_e.I_n(\text{RT}) - \sum_{k=1}^{\sharp Ch_e} E_e.Ch_k.I_1(\text{TS}_3 - \text{TS}_1)$$

The cleansed response time $E_e.I_i(\mathrm{RT})$ is only cleansed according to the overhead of the few RTIs that intercept that invocation. Inside the coherency scope, the cleansed RT of a middle node still does include the overhead of the RTIs of its second-order invocations. Therefore a *globally cleansed response time* ($\mathrm{RT}_{\mathrm{GloClean}}$) for an invocation E_e has to be calculated.

This is possible using the stored ACVs. For an invocation that is intercepted by n RTIs, there are stored $2 * (n - 1)$ ACV values. They are numbered from $\mathrm{ACV}_{(1)}$ to $\mathrm{ACV}_{2n-(2)}$, with the later being passed from RTI_2 to RTI_1. There exists no $\mathrm{ACV}_{2n-(1)}$ by the same means of the non-existence of $\mathrm{TS}_4^{(1)}$. But only the $\mathrm{ACV}_{2n-(1)}$ would represent the whole overhead. The time-gap between the desired $\mathrm{ACV}_{2n-(1)}$ and the available $\mathrm{ACV}_{2n-(2)}$ is nearly the same as $\mathrm{TS}_4^{(1)} - \mathrm{TS}_3^{(1)}$. Independent from any implementation the $\mathrm{ACV}_{2n-(2)}$ has inherently to be stored by RTI_1, so we write it as $I_1(\mathrm{ACV}_{2n-(2)})$ and not as part of I_2.

The ACVs of all the invocations that are in direct causal relation to an invocation E_e inside the coherency scope – E_e's children and sub-children – can be accumulated to an $\mathrm{ACV}_{\mathrm{globalized},E_e}$ in a recursive fashion:

$$\begin{aligned}
\mathrm{ACV}_{\mathrm{globalized},E_l} \text{ of leaf node } E_l \quad &:= \quad E_l.I_1(\mathrm{ACV}_{2n-(2)}) \\
\mathrm{ACV}_{\mathrm{globalized},E_m} \text{ of middle node } E_m \quad &:= \quad E_m.I_1(\mathrm{ACV}_{2n-(2)}) + \sum_{k=1}^{\sharp Ch_m} \mathrm{ACV}_{\mathrm{globalized},Ch_k}
\end{aligned}$$

The formula for the middle nodes would also be representative for the leaf nodes, because the number of children is just 0, and the sum drops out. Even so, we keep both types separated to emphasize the termination of the algorithm.

The globally cleansed $\mathrm{RT}_{\mathrm{GloClean},E_e}$ for an invocation E_e is easily calculated if $\mathrm{ACV}_{\mathrm{globalized},E_e}$ can be computed:

$$\mathrm{RT}_{\mathrm{GloClean},E_e} \text{ of any node } E_e \quad := \quad E_e.I_1(\mathrm{TS}_3 - \mathrm{TS}_1) - \mathrm{ACV}_{\mathrm{globalized},E_e}$$

The formula for $\mathrm{RT}_{\mathrm{GloClean},E_e}$ uses $\mathrm{TS}_3 - \mathrm{TS}_1$ as the uncleansed response time, instead of $\mathrm{TS}_4 - \mathrm{TS}_1$, because the TS_4 is never available for the RTI_1. It is not allowed to subtract the $\mathrm{ACV}_{\mathrm{globalized},E_e}$ from the cleansed response time $I_1(\mathrm{RT})$ instead of the uncleansed response time $I_1(\mathrm{TS}_3 - \mathrm{TS}_1)$, because this would subtract the $E_e.I_1(\mathrm{ACV}_{2n-(2)})$ two times.

If a constant number of RTIs is registered to all EJBs, an average overhead ($\mathrm{O}_{\mathrm{avg}}$) of all RTIs per invocation can be calculated once. For example as mean average $\mathrm{O}_{\mathrm{avg}} := \sum_{x=1}^{X} E_x.I_1(\mathrm{ACV}_{(2n-(2))})/X$, in which X is a statistical representative number of available RTI tuples. Thereby, $\mathrm{O}_{\mathrm{avg}}$ allows to faster approximate the $\mathrm{RT}_{\mathrm{GloClean}}$ for the initial invocation E_1 of a coherency scope:

$$\begin{aligned}\mathrm{RT}_{\mathrm{GloClean}} \text{ of } E_1 \;&:=\; E_1.I_1(\mathrm{TS}_3 - \mathrm{TS}_1) - \mathrm{O}_{\mathrm{avg}} * \sharp\mathrm{E} \\ &\approx E_1.I_1(\mathrm{RT}) - \mathrm{O}_{\mathrm{avg}} * (\sharp\mathrm{E} - 1)\end{aligned}$$

Both approaches can be used alternatively. The formula is worth mentioning, because the $\mathrm{O}_{\mathrm{avg}}$ can efficiently be gained by the AVERAGE() function of the DBS, using the ACV data of the RTIs with index 1 in the raw_RTI relation, for example even without regard of coherency scopes using all available RTI tuples.

The $\sharp$E, representing the total number of invocations inside the coherency scope, can easily be gained using COUNT() on a selection of tuples of a coherency scope, for example selecting by the CoherencyID on raw_CallMetadata.

The same is possible for any invocation E_e using instead of $\sharp$E the recursive number of all E_e's children and sub-children. We do not want to express this mathematically, but on the DBS it is easy: Half of the COUNT()-result of the selection of tuples in v_CallStack, using $E_e.I_1(\mathrm{TS}_2)$ as minimum boundary of the TIME attribute and $E_e.I_1(\mathrm{TS}_3)$ as maximum boundary. Halving the COUNT()-result is necessary, because v_CallStack contains always two tuples for each invocation (one "in" and one "out" tuple).

5.3.7. Summary

Using the information stored by the RTIs not only the response time can be determined but also call dependencies between the different EJBs of a coherency scope. The theory and the implementation of an algorithm for the temporal-order of invocations have been explained, as well as the creation of an invocation tree and an adjacency matrix for the invocations of a coherency scope.

It has been pointed out that for an unrestrained application of the algorithm it is necessary to generate the CoherenceID at run-time, and that this is not possible at the moment, but could be provided by a facility similar to the one that provides the TA-IDs as part of the application server.

Finally, a formula was given that allows the approximated calculation of the run-time of an isolated EJB inside a coherency scope. Also, a formula was presented that allows the accumulation of the overhead due to the RTIs of all invocations inside the coherency scope, making it possible to calculate a globally cleansed response time.

5.4. CPU and Memory Consumption

This chapter provides basic research and evaluation of profiling techniques for the measurement of CPU and memory consumption due to an EJB.

The discussions will give advice for the design of an implementation, but this thesis is merely a preparation for two further student research projects that will provide more detailed research and an implementation.

The following sections assume that the introductory explanations to JVMPI and bytecode instrumentation in Sect. 2.2 have been read.

5.4.1. Definitions

There can be distinguished several types of resource consumption due to EJB components. The first one is the *object local EJB* (OLE) resource consumption. It covers only the consumption by the byte code of the concrete EJB Instance object, and it does not cover the time for method calls or memory for instantiated objects that are not defined by the class itself, nor inherited methods' resource consumption. Even classes in the same package are not considered either, nor are the Java standard classes considered. For example, bytecode instrumentation covers OLE consumption measurement, but even JVMPI is able to provide it.

The *archive comprising EJB* (ACE) resource consumption contains the OLE consumption for all the classes that are contained in the EJB's jar archive, but no other classes. It still does not consider used Java standard classes. Applying bytecode instrumentation to all classes of the EJB's archive covers ACE measurement.

The *thread local EJB* (TLE) resource consumption equals the thread local measurement, introduced in Sect. 2.3.2. It does contain any resource consumption by objects that are in direct causal relationship to the EJB invocation, as long as the objects belong to the same thread. The Java standard classes are now included, but the consumption due to invocations of other EJBs, that are carried out in other threads, are not considered.

The *thread spanning EJB* (TSE) resource consumption is aware of EJB assemblies and includes the consumption of all invocations to other EJBs that are in causal relationship to the current EJB. Beware that TSE consumption is only aware of threads in the local JVM.

Finally, the *process spanning EJB* (PSE) resource consumption additionally includes resource consumption due to the current EJB invocation that appears in other processes – like the DBS, remote EJB containers or extern applications that are plugged to the EJB server, for example by the *J2EE Connector Architecture*.

5.4.2. Bytecode Instrumentation

Bytecode instrumentation[10] uses static analysis to fragment the bytecode into atomic blocks (BoE), for example using the occurrences of `goto`, `jsr` and `ret` instructions as well as conditional `if` operators or `athrown` instructions for exceptional code. More detailed information is provided by Rory Vidal [Vid01], whose work stands in relation to J-RAF2 [HK03].

For each BoE the specific amount of bytecode instructions can be counted. A counter is added to each class and in each BoE an instruction is added that increments the object counter by the block-specific number of bytecode instructions, as described in Sect. 2.2.2.

To account for the different types of bytecode instructions separate counters for the different types would be possible, but this introduces more overhead. Instead the different run-time of the instructions could be approximated during static bytecode analysis, accumulated for each BoE and used for incrementing the single field at run-time. But this is still quite an approximation of time consumption, because the run-time influence of cache misses or page table walk operations are ignored.

Memory allocation is measured by having the size of each created object added to an additional field variable, instrumenting the new() operations and calculating the size from the number of object fields.

[10] We want to mention again the semantic difference between our usage of the terms “bytecode instrumentation” and “bytecode modification” which has already been emphasized in Sect. 2.2.2. The later one has the more general meaning, whereas the bytecode instrumentation is a specific kind of bytecode modification, that instruments the bytecode itself for resource consumption measurement.

Memory deallocation can be estimated by adding and instrumenting the finalizers. Unfortunately there is neither any guarantee of the point in time the JVM calls finalizers, nor that the JVM calls the finalizers at all. Therefore, memory consumption calculation is only exact according to a maximum bound [VB01, 9]. The actual memory consumption is only estimated.

It seems possible to adopt this technique for EJBs, but information passing must be guaranteed, because analyzers have to read out the field variables before the EJB is destroyed, but they have no knowledge about the appropriate point in time. Either an additional bytecode modification of `ejbRemove()` pushes the performance information to an analyzer, or the EJB container is modified to callback analyzers in the event of EJB removal.

Unfortunately, the J-RAF2 [HK03] implementation of bytecode instrumentation is not publicly available. Further student research projects will also evaluate *Prof-Builder* [CLZ98] that provides bytecode instrumentation similar to Vidal and J-RAF2. It is based on the *Bytecode Instrumenting Tool* (BIT) [BIT] and it is designed as extendable framework, for example allowing the definition of custom metrics for CPU and memory consumption.

The comprehensiveness of the information by bytecode instrumentation according to EJBs is limited, because only the EJB classes of the `jar` archive will be modified for accounting. Therefore, only OLE and ACE resource consumption is accountable. Of course, it is possible to modify the bytecode of the complete JVM and EJB container to gain TLE and TSE comprehension, but this would introduce massive overhead. While only the classes of the `jar` archive are instrumented, the bytecode instrumentation introduces little overhead, because it is inherently selective in contrast to other profiling techniques like JVMPI, where filtration mechanisms for classes are necessary.

5.4.3. Java Virtual Machine Profiling Interface

This section will mainly discuss the event-driven usage of a JVMPI profiler agent. Nevertheless, we want to remark a technique for time-driven measurement to detect whether a certain thread has run during the past interval: Liang and Viswanathan

[LV99] showed that by hashing the JVM's register set of the thread, any change can easily be found out and implies that the thread has been run.

5.4.3.1. Overview over the API

This section gives a brief overview over the API of the JVMPI. The specification and documentation is provided by Sun [Sun98].

The profiler agent must implement an JNI function named JVM_OnLoad(), shown in Lst. 5.4. The JVM will call this function during its initialization, which allows the profiler to store the reference to the JVMPI implementation of the JVM (jvm−>GetEnv()) and to install the callback handler (jvmpi−>NotifyEvent = ...). Registration for events (jvmpi−>EnableEvent()) can be done either during JVM_OnLoad() or at any time later.

```
#include <jvmpi.h>
JavaVM *jvm = NULL;
static JVMPI_Interface *jvmpi = NULL;

extern "C" {
        JNIEXPORT jint JNICALL JVM_OnLoad ( JavaVM *vm, ... ) {
                jvm = vm;
                jvm->GetEnv((void **)&jvmpi, JVMPI_VERSION_1);
                jvmpi->NotifyEvent = notifyEvent;
                ...
                jvmpi->EnableEvent(JVMPI_EVENT_METHOD_ENTRY, NULL);
                ...
        }
}

void notifyEvent(JVMPI_Event *event){
        ...
}
```

Listing 5.4: Basic entry function of a profiler agent

The installed callback handler, in our example notifyEvent(), will be called by the JVM for the various events. Listing 5.5 shows an example for such an event handler, and thereby lists several possible events, for example thread handling, class loading, method invocations, object (de-)allocations and garbage collector events. The complete listing of events can be found in the specification [Sun98].

```
void notifyEvent(JVMPI_Event *event) {
        JNIEnv *env;
        jvm->GetEnv((void **)&env, JNI_VERSION_1_2);

        if ((event->envent_type ! JVMPI_REQUESTED_EVENT) == 0) ...;

        // (This does not list all possible events!)
        switch(event->event_type) {
        // JVM initialization and shut-down:
        case JVMPI_EVENT_JVM_INIT_DONE: ...; break;
        case JVMPI_EVENT_JVM_SHUT_DOWN: ...; break;
        // Threads:
        case JVMPI_EVENT_THREAD_START: ...; break;
        case JVMPI_EVENT_THREAD_END:   ...; break;
        // Class loading:
        case JVMPI_EVENT_CLASS_LOAD:      ...; break;
        case JVMPI_EVENT_CLASS_LOAD_HOOK: ...; break;
        // Method invocations:
        case JVMPI_EVENT_METHOD_ENTRY:  ...; break;
        case JVMPI_EVENT_METHOD_ENTRY2: ...; break;
        case JVMPI_EVENT_METHOD_EXIT:   ...; break;
        // Objects:
        case JVMPI_EVENT_OBJECT_ALLOC: ...; break;
        case JVMPI_EVENT_OBJECT_DUMP:  ...; break;
        case JVMPI_EVENT_OBJECT_FREE:  ...; break;
        case JVMPI_EVENT_OBJECT_MOVE:  ...; break;
        // Bytecode instructions:
        case JVMPI_EVENT_INSTRUCTION_START: ...; break;
        // Garbage collector:
        case JVMPI_EVENT_GC_START:  ...; break;
        case JVMPI_EVENT_GC_FINISH: ...; break;
        }
}
```

Listing 5.5: Exemplary behavior of the installed callback handler

The objects, methods and threads are only identified by IDs when they are passed to the callback handler. Therefore, the agent has to gather all information about the IDs using its reference to the JVM to request more information.

The JVMPI does include several methods, but we only want to mention the methods to control the garbage collector (DisableGC() and EnableGC()) and the method that theoretically allows OS independent thread local time measurement: GetCurrentThreadCpuTime(). As we have already discussed in Sect. 2.3.2, the examination of Sun's Linux implementation of this method showed that it relies on the process global gettimeofday(2) system call, and not even conforms to the specified unit of nanoseconds, but uses microseconds. Therefore, this method is not reliable to report correct thread local time information. But using process local time measurement in combination with thread-related events allows the profiler agent to realize thread local time measurement on its own.

To gather the resource consumption of particular EJBs a two-phase filtering has to be applied by the agent: The first filtering is based on the event types. Because all events that are registered will lead to a callback regardless of the identity of the causing object, a second filtering has to be applied that filters the relevant classes, threads, objects and methods. Therefor a complex registry has to be implemented.

The second filtering can be realized either by the profiler agent or by the profiler front-end. If only the front-end applies the second filtering the agent has to request the JVM for the information about every ID, because it has to send the information to its front-end that is outside the JVM process. In the context of J2EE this could result in gigabytes of information per minute, as we have seen by basic test-implementations and as is denoted by other projects which we will discuss later.

Therefore, we recommend to realize at least a coarse filtering in the profiler agent itself. This can be done by filtering method names of EJB Instances during JVMPI_EVENT_METHOD_ENTRY event handling. Alternatively, bytecode modifications during class-loader event handling (JVMPI_EVENT_CLASS_LOAD_HOOK) can be used to insert callbacks to the profiler agent at the beginning (ENTRY) and at the end (EXIT) of EJB methods. The bytecode modification approach saves overhead, because filtration of every JVMPI method entry event is quit expensive, but run-time changing the filtration needs the removal of the bytecode modifications if the EJB is not part of the inclusive filter any more, and the dynamic removal is difficult. We will further discuss both approaches in Sect. 5.4.5.

5.4.3.2. Discussion

To begin with, JVMPI does not need source code to be available and it is transparent to the EJB server and the application layer. Filtering of classes and methods is possible, therefore EJB related resource consumption can be filtered. And the influence of the garbage collector can be considered by its accompanying events.

The types of resource consumption that can be provided by JVMPI profilers are OLE, ACE, TLE and TSE. Even spanning several distributed EJB containers is possible using a central remote profiler front-end that collects data from all JVMs of the EJB containers, converging to PSE resource consumption. The front-end can only achieve comprehensive PSE if also the external non-Java applications, e.g. the DBS, provide appropriate profiling interfaces.

The JVMPI is designed to be independent of the underlying JVM implementation, so the same profiler can work with different JVM implementations for the same OS. A run-time deactivation can be provided, using a boolean value to order notifyEvent() to return immediately after any callback. Even the partial profiling problem can be solved for reactivation [LV99, 11f.].

Several open-source JVMPI profiler implementations exist that can be reused, which will be discussed in the next section.

But there are several shortcomings and drawbacks of JVMPI in contrast to other accounting techniques. On the one hand "a general-purpose interface might not be able to expose all kinds of interesting events that occur in implementations of different vendors" [LV99, 4]. On the other hand Liang and Viswanathan acknowledge that the callback "is somewhat slower than directly instrumenting the virtual machine to gather specific profiling information" [LV99, 4], because using a callback mechanism to realize a profiling interface makes it necessary to have a function call to the handler. But, they put this into perspective by arguing that "most events occur in situations where this additional cost is tolerable".

The most unpleasant drawback of JVMPI for profiling EJBs is the complexity of the required implementation, because all information has to be gathered, filtered and collated by the profiler on its own. The JVMPI implementation has to be multithread aware, realizing mutual exclusion for its data structures.

5.4.3.3. Evaluated Profilers

In order to reuse an existing implementation we have evaluated several JVMPI profilers according to their applicability within the scope of J2EE. If not mentioned otherwise all profilers are related to classic Java application profiling and do not recognize EJBs.

The first profiler was the basic HProf [HPr] profiler agent that is freely included in the Sun JDK. It allows statistical profiling using either a file-based output or the sending of its data over a socket. There is no class-based filtration possible inside the agent and only limited influence on its behavior using its start-up parameter. Thus, we searched for more complex implementations. For the sake of completeness we want to mention an available profiler front-end for HProf: *PerfAnal* [PAn] from Sun, which is based on HProf's file-based output and, therefore, requires the termination of the JVM.

Jinsight 2.1 [Jin] was developed by IBM and provides both agent and graphical front-end. Note that versions prior to 2.1 are not based on JVMPI, but are modified JVMs of IBM's JDK 1.1.8. Jinsight is now integrated into IBM's *WebSphere Studio Application Developer*, but the 2.1 version is still downloadable independently. Neither the Java sources are available nor the C source code of the profiler agent.

The *Performance Inspector* tools [PIn] are also provided by IBM. There are tools that work with C/C++ and Java applications, but for Java an IBM JVM 1.4 is required. The profiler agent for Java applications is called *JProf*. It allows socket communication of the profiling data as well as dynamic activation. A specialty about the Performance Inspector tools are their usage of performance counters. The JProf profiling agent is only binary available under the LGPL license, and it allows no class-based filtration.

The *Extensible Java Profiler* (EJP) [EJP] is an open source profiler, including an agent and a graphical front-end. The implementation of the profiler agent is rather simple. It creates a binary, optionally compressed file that is used by the front-end to create a call tree. Several views at the data and the tree can be realized with *filters*. The EJP distribution of the front-end provides basic filters and is designed as plug-in architecture allowing the extension of EJP with custom filters. It seems possible to implement filters that accumulate the profiling information to gain EJB related information. But the termination of the JVM is required, and no class-based filtration is possible inside the agent. Starting JBoss itself led to a compressed file of the size about 1GB and a noticeable slowdown. Although the agent allows to be deactivated at start-up time and to be activated later the data created for the JBoss start-up gives an impression to what extend of data a non-filtered JVMPI profiling will lead.

We also evaluated several commercial JVMPI-based profilers: *JProbe* [JPb] from Quest Software is a comprehensive (and expensive) profiler suite for Java. It supports the installation of its agent into J2EE servers and provides filtration on basis of packages, classes and method names. It allows the graphical browsing of call graphs and the definition of triggers according to exceedance of memory thresholds. But it is mainly a classic Java application profiler that still provides the information at object and method level and not at the level of EJBs. JProbe's filtration mechanism allows settings so that profiling overhead is little enough to be applicable even in productive environments. Unfortunately, there is no API that allows the creation of an extension that could provide the desired EJB related information.

JProfiler [JPf] from Borland allows to be integrated into several Java development environments, for example JBuilder and Eclipse. It provides similar functionality as JProbe, but for example the filtration support of JProbe is superior to JProfiler.

PerformaSure [PSu] from Quest Software and OptimizeIt Server Trace [OIt] from Borland are promising and were already mentioned in Sect. 4.1. PerformaSure is only available for the two J2EE servers WebLogic from BEA and WebSphere from IBM, and an evaluation version was not available to us. For OptimizeIt Server Trace there is a preview demonstration for Windows downloadable. Several attempts to contact Borland representatives in order to obtain a full evaluation or a release date had no success, therefore we assume the software to be in an early stage. However, the preview demonstration shows that these tools recognize EJBs, instead of pure Java Objects, but aim primarily at the revelation of performance bottlenecks by EJB programmers in a dedicated development environment before the EJB is released, and not at the gathering of performance statistics and characteristics about the whole EJB. We could not find any hints that these tools consider the interdependencies of

EJBs, nor that the interdependencies to the method's parameters are considered and set into relation with the performance information, which will both be necessary for QoS specification.

Aside the profiler agent implementations from the various companies above there are several JVMPI-based academic frameworks that were mentioned in Sect. 4.1.2.

Reiss and Renieris [RR00] describe a three-layer architecture to generate Java trace data based on JVMPI. The profiler agent is called *TMon* and creates output streams for each thread. The profiler front-end is called *TMerge*. TMon communicates to TMerge either file-based or by a shared memory buffer. TMerge merges the several streams providing consistency and uniqueness of the IDs. It stores the IDs and their information to a DBS and generates a comprehensive trace file. Analysis and visualization is provided by *TFilter*, which depends on the data from TMerge. Because there is no filtration inside the first layer, the TMon agent, they have generated more than 1.6 GB data for a 25 seconds run of an unnamed server application [RR00, 4]. Because their intention is not to filter, a later publication [RR01] discusses several methods to compress the data by different encoding techniques, like run-length encoding, grammar-based encoding and finite state automata encoding. The implementation was not available to us and the authors do not consider components and EJBs.

The *Java Performance Monitoring Toolkit* (JPMT) [HQGM02] supports event filtering at run-time. It allows event-driven measurement as well as time-driven measurement. JPMT uses memory-mapped (binary) files to store the data. The OS interface for memory-mapped files is different for the several OS and the description of Harkema et al. implies the implementation to be available for Linux. A specialty about JPMT is its instrumentation of performance counters, using the *PerfCtr* [PCt] Linux kernel patch by Mikael Pettersson for accessing the PAPI [PAP] performance counter library. The authors do not consider components and EJBs, but nevertheless, they describe the modification of bytecode in order to insert callbacks to the profiler agent into classes of interest during JVMPI_EVENT_CLASS_LOAD_HOOK event handling. Adaption of the JPMT approach would allow efficient filtering for EJB method entries, allowing to initiate full event profiling for the accordant thread at EJB method entry to realize TLE resource consumption. Unfortunately, contacting the authors unfolded that they have not yet decided whether, or under which license, to hand out the implementation.

The *Form* framework [FOR] uses JVMPI, explicitly considers EJBs and is designed as distributed system. It is based on a three-layer architecture. The profiler agent generates XML code that encapsulates information about the event. The agent transfers the XML event messages via socket communication to one or several remote profiler front-ends, called *Form controller*. The agent does not filter the events, therefore the data is voluminous. The Form controllers are mediators between the profiler agents and the actual event consumers of the profiling data, the *Form Views*.

A Form View application registers at one or several Form controllers for events. Event registration is based on regular expressions that are applied to the full classified Java names of objects and methods, and the regular expression filters are called *second order filters*. The Form controller merges all second order filters of its registered Views to a *first order filter* that maps the events to the interested Views. The normalization of time events from the several JVMs' profiler agents is done by the Form Views using a distributed time algorithm [SMS01, 5]. Form Views are also allowed to register directly to a profiler agent if unfiltered event information is desired.

The Form framework is accompanied by the *Sequence Diagram* (SD) tool [SMS01, 6–8] which is an implementation of a Form View that provides the creation of UML sequence diagrams for monitored executions. SD handles event consumption from multiple Form profiler agents, thus, it considers executions across multiple JVMs.

The authors have published the source code of the Form framework [FOR]. The profiler agent library is available for Linux, Solaris and Win32, and the architecture and the data format of the messages is described in detail by Souder and Mancoridis [SM00]. The COMQUAD project will evaluate the adaptability of the Form framework for QoS specification in further student research thesis. For example, the overhead due to the profiler agents and their XML generation is not apparent from the publications, and coarse filtering inside the profiler agents is not provided.

Another JVMPI profiler that is available as source code is part of the *Eclipse Profiler Plugin* [EPP]. Among the open source profilers it offers the most complex implementation, besides Form, allowing filtration inside the agent. It also includes a graphical profiler front-end, realized as an Eclipse plug-in. CPU profiling is realized by bytecode modification during class-loader event handling, adding callbacks to the profiler agent at the beginning and the end of the interested methods. The agent does not support dynamic removal of its bytecode modifications, therefore the characteristics of the run-time filtration have to be static, and they are statically defined by the agent's start-up parameters. The front-end implements a second filtration that can be dynamically changed. The agent also supports time-driven measurement (sampling) which introduces less overhead than the event-driven approach but is not as accurate. The Eclipse Profiler Plugin's agent can be applied to the JBoss JVM, but it does not recognize EJBs, but allows classic profiling on basis of threads, objects and methods, for usage in dedicated development environments. However, the Eclipse Profiler Plugin agent can be used as basis for a custom implementation of a profiler agent.

5.4.4. Alternative Layers for Instrumentation

We have seen that bytecode instrumentation can provide OLE and ACE resource consumption types, whereas JVMPI profilers can provide nearly all types. However, higher layers are able to relate the resource consumption information with additional knowledge. For example, the JVMPI profiler agent does not have to provide EJB related information, but only thread local resource consumption measurement, if the EJB container itself keeps track of the EJB invocations, and just uses the JVMPI agent to gain the resource information before and after the EJB method invocations. This would dramatically reduce the required complexity of the JVMPI profiler agent implementation. Also, bytecode instrumentation information can be correlated for the EJBs of a coherency scope using global knowledge of the EJB container. Therefore, the question arises which layer has to be aware of EJBs, that means which layer shall gather the CPU and memory consumption for a specific EJB instance, independent of the technique that provides the resource consumption information. There are several layers at which this would be possible.

The highest layer is the EJB itself at *application layer*. Similar to the plain client instrumentation for RT measurement is instrumenting the EJB for its own CPU and memory consumption measurement. For example the EJB developer could access initial thread local information in the ejbCreate() method, storing it as a member. The ejbPassivate(), ejbActivate() and ejbRemove() methods could be used to further track the consumptions. The ejbRemove() could finally store the resource values to a DBS, or the EJB implements a (direct support) interface allowing analyzers to query the current status. Such approach shifts the responsibility for the resource values to the EJB's developer and no guarantee for their correctness can be given.

Transparent to the EJB would be the instrumentation of the *EJB container*. It could take the same actions as above before invoking methods like ejbCreate(), ejbPassivate(), ejbActivate() and ejbRemove(), but it would have to expose the information by an interface to make it available to analyzers.

Instead of using access to thread local information, the EJB container can also dynamically apply bytecode instrumentation to the EJB classes during EJB deployment. Special actions for ejbCreate(), ejbPassivate() and ejbActivate() would not be necessary any more, it would be sufficient to access the fields that are provided by the bytecode instrumentation.

Finally, the bottom layer for gathering EJB related resource consumption information is the *JVM*, and the Form framework has already demonstrated how JVMPI can be instrumented to gather EJB related performance information. Instrumentation of the JVM layer makes the implementation independent of the EJB container, and even independent of the component model. Therefore, we propose a JVMPI based solution as the conceptual design of our solution, which is described in the next section.

5.4.5. Conceptual Design

As it is demonstrated by Form and SD (Sect. 5.4.3.3), it is possible to instrument the profiler front-end to be aware of EJBs and to gather TSE resource consumption even across several JVMs. However, being especially interested in monitoring the resource consumption of EJBs in a productive environment, it is most important to reduce the run-time overhead for the server JVMs. Therefore, it is necessary to reject profiling of threads that have no EJB methods of interest in their callstack as fast as possible. Thus, filtration is necessary inside the profiler agent.

The agent has only to gather resource consumption information about threads once an EJB method becomes part of the thread's callstack. After the entry of an EJB method is noticed the profiler agent gathers the resource consumption of all thread local methods and objects as long as the EJB method does not return, because the resource consumption of all of these methods and objects are associated to the EJB method. The fact that an EJB method is part of the thread's callstack can easily be tracked with a boolean tag pro thread that is set at EJB method entry and unset when the method returns. As it was mentioned in Sect. 5.4.3.1 there are two different methods to track the entry and exit of an EJB method.

The first approach is based on the JVMPI_EVENT_METHOD_$*$ events. For example, the full qualified class name of the EJB Instances (or Home and EJB Objects) could be given to the profiler agent by JNI, socket or shared memory communication, possibly supporting regular expressions and wildcards. The entry of EJB methods can be tracked by filtering the JVMPI_EVENT_METHOD_ENTRY events as well as the leaving can be tracked by filtering the JVMPI_EVENT_METHOD_EXIT events. The exact characteristics of the filtering mechanism, for example supporting regular expression, needs further evaluation in relation to efficiency aspects, because filtration must be applied to most method entry events, so its overhead has to be well considered.

The alternative to event based filtering of EJB methods is the bytecode modification during JVMPI_EVENT_CLASS_LOAD_HOOK event handling. The class-loading event handler filters EJB classes and adds bytecode to EJB methods so that at the beginning and end of the methods a call back to the profiler agent occurs. An introduction to the insertion of such callbacks is given by Harkema et al. [HQGM02], and the source code of such an implementation is available by the Eclipse Profiler Plugin [EPP]. The callbacks can be used for setting and unsetting boolean tags pro thread, identifying threads that have EJB methods as part of their current callstack.

The bytecode modification saves overhead, because filtration of every JVMPI method entry event is quit expensive. Allowing changes of the filter at run-time is difficult, because bytecode modifications seem to be possible only during class-loading. Nevertheless, filter changes are at least possible if the involved EJBs are redeployed, because the server has to reload the jar archive and the EJB classes. Obviously, the according filter changes have to be made before the redeployment.

Having implemented such coarse filtration by tagging threads with EJB methods as part of the thread's callstack, and generating comprehensive thread local profiling information in between EJB method entry and leaving, the architecture of the Form framework seems promising according to the implementation and design of the front-ends. Although, further evaluation has to analyze whether the XML generation inside the profiler agent is efficient enough for its usage in full-fledged and productive EJB containers or whether a custom implementation using binary formats is required.

5.4.6. Summary

This section has introduced five different types of resource consumption due to EJBs. Two Java profiling approaches for resource consumption, bytecode instrumentation and JVMPI, have been discussed in more detail. Their instrumentation for EJB-related measurement have been outlined, and for JVMPI the API was described as well as several implementations of JVMPI-based profilers.

We have described how higher layers could instrument basic thread local resource consumption information to enrich it with global knowledge, thus reducing the required complexity of the profiler implementation.

Finally the pure JVMPI-based approach has been chosen as recommended architecture, because it is transparent to the EJB container and independent from a vendor's implementation. We recommend to adopt the Form framework, but it has to be extended by a coarse filtering mechanism inside the profiler agent in order to reduce the introduced overhead for applicability to productive environments.

6. Conclusion

6.1. Summary

This thesis introduced the concepts of a measurement environment for components following the EJB specification and the implementation of selected concepts, aiming at the derivation of data that will allow to create and to verify a quality of service specification for a component.

Therefor a literature research has been accomplished for measurement techniques, approaches and implementations that are applicable in the context of J2EE and EJB components. We have seen that an integration of measurement techniques into the JVM and the EJB container is still underdeveloped, as well as the access to performance counter needs standardization. A classification system was worked out to compare and discuss the different approaches.

It has been shown how the interceptor pattern can be applied for response time measurements in the open-source application server JBoss. Following a discussion of several possible layers of instrumentation, the application of our concept has been explained in detail for response time measurement. The documentation of our implementation was provided according to data structure, information passing mechanism and reusage for more types of interceptors. It has also been described how response time interceptors can be used to measure the run-time of other interceptors, by wrapping the interceptor of interest between two response time interceptors.

The theory and the implementation of an algorithm for the temporal ordering of invocations have been explained, using the information stored by the RTIs so that call dependencies between several EJBs can be derived. Because in an assembly of interacting components, not only the performance of an individual component is actually what matters, but also the invocation frequentness and dependencies.

For the measurement of resource consumption due to components the interceptor approach is not feasible. Therefore JVMPI and bytecode instrumentation have been evaluated. After the description of several available JVMPI based profilers for classic profiling, possible layers for collecting EJB-related resource information have been discussed. Finally, we described the design of a JVMPI based framework adopting, reusing and enhancing the work of other publications.

6.2. Further Work

In relation to the measurement of CPU and memory consumption there exists a new progression in the Java community: the Java Platform Profiling Architecture (JPPA). It could succeed JVMPI but standardization is still in progress as Java Specification Request JSR-163 [Sun03c]. Keeping compatibility to JVMPI, JPPA seems to better support bytecode modifications, for example used for insertion of callbacks to the profiler agent into methods which makes overhead-critical filtration of method events needless.

The evaluation of JBoss interceptors demonstrated their flexibility, and their applicability for measurement of jitter or throughput as well as fault injection will be studied further. We will also look into extending the JBoss facility that provides the TA-IDs in order to provide the CoherenceID at run-time.

To allow complete interpretation of the gained performance statistics a comprehensive knowledge about the run-time environment has to be collected and stored in relation to the measurement values. A characterization scheme for machine characteristics has to be developed that is independent from OS and hardware platforms. An example for a source of such machine characteristics is the Linux /proc filesystem. The filing and classification of run-time conditions and utilized virtual machines is another challenge, abstracting JVMs of different vendors as well as products like VMware [VMw].

To guarantee consistency and integrity of the information to analyzers, an integrated and thorough relational schema has to be developed, that includes all the data from the interface specifications, QoS specification, machine characterizations, virtual machines, run-time conditions and performance measurement sensors. These tasks are currently addressed by other student research projects.

For the validation of the QoS specification by the measurement framework, using a broad range of parameters conditions, a test driver for test-series and a load generator is necessary. Finally, after the performance information is available, many kinds of analysis for easy interpretation are needed as are visualizations of the data as well as a console for the management of the framework.

Appendix A. Documentation

This appendix chapter gives more resources concerning the implementation of our performance measurement facility. The RTI framework is treated first, followed by the framework for the analysis of call dependencies.

This thesis is accompanied by several resources. The variable $CD references to the root directory of these resources. Contact to the author is possible by eMail, using sichneum@stud.informatik.uni-erlangen.de or c.p.neumann@web.de.

A.1. Response Time Interceptor Framework

For the RTIs the installation process is described concerning the server, the clients and the necessary adoptions of the EJB's deployment descriptor. Templates are provided for the reusage of the timestamping functionality in other interceptors that want to participate in our LIC. Additionally, an overview over the implemented classes and their API is provided as UML diagrams.

Before trying to install the framework on your JBoss, please make sure that the JVM is at least at least a Sun JVM 1.4.2.

A.1.1. Installation at Server-side

The referenced files are available in the $CD/deploy-server/ directory of the accompanying resources of this thesis. The server should not be running during the installation steps.

At first, install the libNativeTimeProvider.so to the $JRE/lib/i386/ directory. Then, install the immd6_Interceptor-interceptors.jar and the common immd6_util.jar to the $JBOSS/server/⟨config⟩/lib/ directory. Finally, deploy the main framework archive immd6_Interceptor.ear to the $JBOSS/server/⟨config⟩/deploy/ directory.

After this, read carefully the next section about the DBS settings and make your adoptions. Finally, the JBoss server can be started.

A.1.2. Defining the Database for the Storage of the Transfer Objects

The RTI framework includes a basic storage mechanism for the TOs as described in Sect. 5.2.5, using CMP entity beans. For CMP-EJBs there has to be defined a correct datasource, that is installed and available to the EJB container. It is specified in one of the CMP-EJB's vendor-specific deployment descriptors, for JBoss this is the META-INF/jbosscmp-jdbc.xml file.

If the source code of our implementation is available to you, it is easiest to adopt our Ant properties file and to regenerate the framework. If the sources are not available to you, but the binary ear archive, the standard procedure has to be applied: Unpacking the archives and editing the deployment descriptors. Both ways are explained in more detail below.

The standard configuration for the DBS is the usage of datasource java:/OracleDS with the mapping Oracle9i, as we use at our chair. The $CD/deploy-sever/JBoss(Hypersonic)_deploy/immd6_Interceptor.ear is configured to use the java:/DefaultDS datasource and the Hypersonic SQL mapping and can be used instead of the standard CD/deploy-sever/JBoss_deploy/immd6_Interceptor.ear file.

For Oracle9i you will need to copy the JDBC driver (ojdbc14.jar, for example available in the $CD/undocumented/DBS.oracle_driver/ directory) to the $JBOSS/server/⟨config⟩/lib/ directory.

(This concerns mainly local Comquad team members, who are installing an own JBoss instance:) A JBoss datasource definition file for our Oracle instance is in the $CD/deploy-server/JBoss_deploy/ directory as file oracle-ds.xml, which has also to be copied to the $JBOSS/server/⟨config⟩/deploy/ directory. You have to edit it, providing correct <user−name> and <password> settings.

A.1.2.1. Source-Code is Available

If the source code of the RTI framework is available to you, the easiest way to change the targeted DBS datasource that is used to store the TOs is editing the release.properties (or debug.properties) file in the $ECLIPSE/workspace/Interceptor/build/ directory, setting the correct jboss.datasource and jboss.datasourcemapping variables. Running Ant's codegen (or compile, deploy or staging) target will generate the correct META-INF/jbosscmp-jdbc.xml descriptor file inside the immd6_Interceptor-ejb.jar, which is part of the immd6_Interceptor.ear archive.

A.1.2.2. Source-Code is not Available

If the source code of the RTI framework is not available to you, but only the immd6_Interceptor.ear archive, the DBS settings have to be edited manually in the META-INF/jbosscmp-jdbc.xml inside the immd6_Interceptor-jmx.sar, which is part of the immd6_Interceptor.ear archive. Edit the values of the <datasource> and <datasource−mapping> tags, which are between the <defaults> ... </defaults> tags at the beginning of the document.

A.1.2.3. Defining the Database Relations

The relations that are used to store the TOs are defined by the deployment descriptors of the two CMP-EJBs. The EJBs are named PersistentCallMetadata (default relation: raw_CallMetadata) and PersistentResponsetimeTO (default relation: raw_RTI). The descriptor is stored in the META-INF/ejb-jar.xml inside the immd6_Interceptor-ejb.jar, inside the immd6_Interceptor.ear archive.

A.1.3. Installation at Client-side

Copy the immd6_Interceptor-client.jar and the common immd6_util.jar to the client and add them to its classpath. The needed JBoss libraries are the common $JBOSS/

client/jbossall-client.jar as well as the $JBOSS/server/default/lib/jboss.jar, $JBOSS/lib/jboss-system.jar and $JBOSS/lib/jboss-jmx.jar. All of these files are available in the $CD/deploy-client/ directory.

The later three files are normally server-side libraries. Due to the fact that the RTI implementation is generic to client- and server-side we need these libraries present to satisfy the JVM's ClassLoader. Because we cache at server-side, for example, the reference to the MBeanServer by a field variable, even at client-side where this field will always be null the ClassLoader has need for the class' type information.

For a μs support at client-side also the libNativeTimeProvider.so has to be installed to the client-side $JRE/lib/i386/ directory.

A.1.4. Activate Measurement for an EJB

The activation of the measurement framework for an EJB is nothing more than the registration of the response time interceptors to the EJB's vendor-specific deployment descriptor.

If the source code of the considered EJB is available to you, it is easiest to adopt Ant for correct generation of the descriptors. If the sources are not available to you, but the binary jar archive, the standard procedure has to be applied: Unpacking the archives and editing the deployment descriptors. Both ways are explained in more detail below.

A.1.4.1. Source-Code is not Available

If the source code of the considered EJB is not available to you, but only the jar archive, the interceptor settings have to be edited manually in the META-INF/jboss.xml file. The basic structure of this file is shown in Lst. A.1. According to Sect. 2.1.3, and its Lst. 2.1, the relevant sections are <container−configurations> for server-side interceptors, and <invoker−proxy−bindings> for client-side interceptors.

The following describes the standard way to install a custom interceptor chain configuration to any EJB: For the type of the considered bean the standard configurations can be found in $JBOSS/server/default/conf/standardjboss.xml. Copy the appropriate configuration to the EJB's jboss.xml. Change the <configuration−name>

```
<jboss>
  <enterprise-beans>
  <!-- Listing of EJBs, for example their JNDI names -->
  </enterprise-beans>

  <resource-managers>
  <!-- ... (will probably be empty!) -->
  </resource-managers>

  <container-configurations>
  <!-- That is the important section -->
  <!-- for SERVER-SIDE interceptors  -->
  <!-- NEEDS TO BE EDITED -->
  </container-configurations>

  <invoker-proxy-bindings>
  <!-- That is the important section -->
  <!-- for CLIENT-SIDE interceptors  -->
  <!-- NEEDS TO BE EDITED -->
  </invoker-proxy-bindings>

</jboss>
```

Listing A.1: Torso of the jboss.xml file

(server-side interceptor chain) and the <invoker–proxy–binding><name> (client-side interceptor chain). Because the server-side configuration references the client-side configuration, and the EJB references only the server-side configuration, the <invoker –proxy–binding–name> tag inside the server-side settings has to equal the <invoker– proxy–binding><name> of the client-side configuration.

Now you have an interceptor chain configuration with a custom name, and you are ready to add <interceptor> tags, for example the RTI ones similar to Lst. 2.1 in Sect. 2.1.3.

In the section below, Lst A.3 gives example for a whole configuration for stateless session beans. Looking it over you should be able to identify the corresponding sections in the JBoss standardjboss.xml file.

A.1.4.2. Comquad Tool

The Comquad project will soon provide a graphical tool, that will support the automation of the above steps. It will simplify the adoption of any EJB's binary jar distribution according to our measurement framework.

A.1.4.3. Source-Code is Available

If the source code of the considered EJB is available and the compilation process uses XDoclet/Ant (e.g. Lomboz-projects), the easiest approach is the usage of a jboss-container.xml in the Ant project's merge directory. If such file is put into the merge directory, it will automatically become part of the jboss.xml deployment descriptor. For Lomboz, the merge directory is the $ECLIPSE/⟨project⟩/⟨module⟩ /META-INF/ directory.

We have prepared a jboss-container.xml for CMP 2.0 entity beans and stateful or stateless session beans, running on a JBoss 3.2.1. It is available in the $CD/ helper-files/ directory. Therefor the equivalent sections from $JBOSS/server/default/ conf/standardjboss.xml were taken and the <interceptor> tags for our RTIs were added.

Listing A.3 shows a reduced version of the jboss-container.xml file where only the stateless session bean configuration is included. But thereby, it provides the information to identify the corresponding sections in the JBoss standardjboss.xml file.

After copying the $CD/helper-files/jboss-container.xml to the Ant project's merge directory an XDoclet tag has to be added to the EJB's class source. To enable an interceptor configuration for the EJB XDoclet has to add the <configuration−name> ...</configuration−name> to the EJB-entries that are part of the <enterprise−beans> listing in jboss.xml. This is automated by adding the @jboss: container−configuration name="..." entry to the javadoc comment of the EJB class, shown in Lst. A.2

```
/**
 * @ejb.bean name="Example"
 *       jndi-name="myArbitraryNamespace/ExampleBean"
 *       type="Stateless"
 *   view-type="remote"
 *
 * @jboss:container-configuration name="IMMD6 Test-Stand Stateless
    SessionBean"
 * @ejb.util generate="physical"
**/
public abstract class ExampleBean implements javax.ejb.SessionBean {
  ...
```

Listing A.2: XDoclet example for an EJB class

```
<!--             -->
<!-- SERVER-SIDE -->
<!--             -->
<container-configurations>
  <!-- STATELESS SESSION BEANS -->
    <container-configuration>
        <container-name>IMMD6 Test-Stand Stateless SessionBean</container-name>
        <call-logging>false</call-logging>
        <container-invoker>org.jboss.proxy.ejb.ProxyFactory</container-invoker>
        <invoker-proxy-binding-name>immd6-test_stand-stateless-rmi-invoker</
            invoker-proxy-binding-name>
        <container-interceptors>
           <interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</
               interceptor>
           <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
           <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
           <!-- CMT -->
           <interceptor transaction = "Container">org.jboss.ejb.plugins.
               TxInterceptorCMT</interceptor>
           <interceptor transaction = "Container" metricsEnabled = "true">org.
               jboss.ejb.plugins.MetricsInterceptor</interceptor>
           <interceptor transaction = "Container">org.jboss.ejb.plugins.
               StatelessSessionInstanceInterceptor</interceptor>
           <!-- BMT -->
           <interceptor transaction = "Bean">org.jboss.ejb.plugins.
               StatelessSessionInstanceInterceptor</interceptor>
           <interceptor transaction = "Bean">org.jboss.ejb.plugins.
               TxInterceptorBMT</interceptor>
           <interceptor transaction = "Bean" metricsEnabled = "true">org.jboss.ejb
               .plugins.MetricsInterceptor</interceptor>
           <interceptor>org.jboss.resource.connectionmanager.
               CachedConnectionInterceptor</interceptor>
           <interceptor>de.uni_erlangen.inf6.test_stand.interceptor.
               ResponsetimeServerInterceptor</interceptor>
        </container-interceptors>

      <instance-pool>org.jboss.ejb.plugins.StatelessSessionInstancePool</instance-
          pool>
      <instance-cache/>
      <persistence-manager/>
```

```
        <transaction-manager>org.jboss.tm.TxManager</transaction-manager>
        <container-pool-conf>
        <MaximumSize>100</MaximumSize>
        </container-pool-conf>
    </container-configuration>
</container-configurations>

<!--             -->
<!-- CLIENT-SIDE -->
<!--             -->
<invoker-proxy-bindings>
   <!-- STATELESS SESSION BEANS -->
      <invoker-proxy-binding>
         <name>immd6-test_stand-stateless-rmi-invoker</name>
         <invoker-mbean>jboss:service=invoker,type=jrmp</invoker-mbean>
         <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
         <proxy-factory-config>
            <client-interceptors>
               <home>
                  <interceptor>de.uni_erlangen.inf6.test_stand.interceptor.
                      ResponsetimeClientInterceptor</interceptor>
                  <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
                  <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
                  <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
                  <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
               </home>
               <bean>
                  <interceptor>de.uni_erlangen.inf6.test_stand.interceptor.
                      ResponsetimeClientInterceptor</interceptor>
                  <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</
                      interceptor>
                  <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
                  <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
                  <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
               </bean>
            </client-interceptors>
         </proxy-factory-config>
      </invoker-proxy-binding>
</invoker-proxy-bindings>
```

Listing A.3: Interceptor definition file for ant-based generation of the JBoss-specific deployment descriptor jboss.xml

A.1.5. Reusage of Timestamping Functionality in other Interceptors

This section gives an overview of the GeneralIMMD6handler, MetadataIMMD6handler and IMMD6generalTO classes that are part of the de.uni_erlangen.inf6.test_stand.interceptor.util package.

A.1.5.1. API

The GeneralIMMD6handler provides access to the RTI-specific information according to raw_RTI: $ACV_{(i-1)}$, TS_1, TS_2, $ACV_{(i)}$, Position in the RTI chain, successor's-ACV ($ACV_{2n-(i+1)}$), successor's-TS_4, TS_3, the interceptor location, the class of the subsequent interceptor and the CallID.

The call-specific metadata-TO is created and stored by the GeneralIMMD6handler for the RTI_n after the invocation has returned. The creation and the management of the call-specific information is done by the MetadataIMMD6handler. But the metadata is not available to the RTIs, because it is transparently generated only for the last RTI_n. If the information about the caller location, the JNDI name, the EJB Instance class, the transaction ID, client IP, method name, arguments, principal or instance ID is needed, the behavior of the MetadataIMMD6handler can be simulated, but is reliable only on the outward way! The server IP and server start-up time is only available at server-side, and the reliability from the server's DataCollector-MBean can not be extracted exactly, because for all other RTIs than RTI_n the TS_2 and TS_3 include more time than the representative $TS_2^{(n)}$ and $TS_3^{(n)}$.

Further, it can be queried whether the interceptor is the first or last one, whether it is on client- or server-side, and whether it is in the JVM of the JBoss application server.

Static methods provide cached access to the MBeanServer and the DataCollector or -Proxy. Transparent and location independent storage to the DataCollector-MBean is provided by storeAnyTO().

Because the interceptor-class itself provides the method to delegate the call to the next interceptor in line, a reference to it has to be passed by the constructor.

These references are available as get_serverInterceptor() or get_clientInterceptor(). Also the Invocation object is available by get_mi().

The interceptor has to use callNextInterceptor() instead of the standard getNext().invoke() mechanism in order to guarantee the correct timestamping and ACV-calculation.

Because the passing of information on the backward way has to be realized by the returned object, it is not correct to return with the object given by callNextInterceptor(), instead the finishing() method of GeneralIMMD6handler has to be used. Its exact behavior depends on the RTI's position in the LIC, in general the ACV is passed correctly and the wrapping of the EJB return value is (un-)done.

The result value of the EJB Instance is given by callNextInterceptor() and can additionally by accessed by getEJBsResult(). Passing custom information on the outward way can be done by putBackpassedValue() and getBackpassedValue().

As long as the LIC-participants TO extends IMMD6generalTO the prepareTransferObject() can be used to fill CallID, interceptor's position in the LIC, interceptor location (client-/server-side) and the class of the subsequent interceptor.

At server-side the interceptor has to notice the GeneralIMMD6handler after construction, whether it is a call to the Home Object or not – with set_HomeObjectInvocation().

A.1.5.2. Templates

If the interceptor that wants to participate in the LIC is provided at client- and server-side (like the RTI) the template in Lst. A.4 should be imitated. If only one side is concerned it is not necessary to implement the real handler in an extra class as it is done by ResponsetimeGeneralizedInterceptor in the provided template.

The exact handling of any such (generalized) interceptor has to imitate the template in Lst. A.6.

```
import org.jboss.proxy.Interceptor;
import org.jboss.ejb.plugins.AbstractInterceptor;
import de.uni_erlangen.inf6.test_stand.interceptor.util.
   NativeTimeProvider;
public class ClientInterceptor extends Interceptor {
        public Object invoke(Invocation mi) throws Throwable {
                GeneralIMMD6handler gh =
                        new GeneralIMMD6handler(mi, false, this);
                ResponsetimeGeneralizedInterceptor bothsides =
                        new ResponsetimeGeneralizedInterceptor();
                return bothsides.unDistinguishedInvoke(gh);
}       }
public class ServerInterceptor extends AbstractInterceptor {
        public Object invoke(Invocation mi) throws Exception {
                GeneralIMMD6handler gh =
                        new GeneralIMMD6handler(mi, true, this);
                gh.set_bHomeCall(false);
                return unDistinguishedInvoke(gh);
        }
        public Object invokeHome(Invocation mi) throws Exception {
                GeneralIMMD6handler gh =
                        new GeneralIMMD6handler(mi, true, this);
                gh.set_bHomeCall(true);
                return unDistinguishedInvoke(gh);
        }

        protected Object unDistinguishedInvoke(GeneralIMMD6handler gh)
                        throws Exception {
                ResponsetimeGeneralizedInterceptor bothsides =
                        new ResponsetimeGeneralizedInterceptor();
                return bothsides.unDistinguishedInvoke(gh);
}       }
```

Listing A.4: Templates for server- and client-side interceptors

```
public Object invoke(Invocation mi) throws Exception {
        long TS1 = NativeTimeProvider.currentTimeMicros();
        GeneralIMMD6handler gh =
                new GeneralIMMD6handler(mi, true/false, this, TS1);
        // ...
        return /*...*/unDistinguishedInvoke(gh);
}
```

Listing A.5: Interceptor taking the TS_1 on its own

```
public class TemplateForSomeGeneralizedInterceptor {
        Invocation _mi = null;

        /* Converges all invoke*()-calls. */
        public Object unDistinguishedInvoke(GeneralIMMD6handler gh)
                                throws Exception {

                /* DEACTIVATED by InterceptorConfig(-Controller) */
                if (InterceptorConfig.deactivateIMMD6Interceptors)
                        return gh.callNextInterceptor();

                _mi = gh.get_mi();
                /* DO Interceptor-specific STUFF */
                // ...

                // CALL NEXT INTERCEPTOR
                gh.callNextInterceptor();

                // OUTBOUND ACTION

                /* DO Interceptor-specific STUFF, for examlpe: */
                SomeSpecificTO = new SomeSpecificTO();
                // ...

                /* if the TO extends IMMD6generalTO then the CallID,
                 * PositionInIMMD6chain, InterceptorLocation and
                 * SubsequentInterceptorClass can be filled
                 * by the GeneralIMMD6handler */
                gh.prepareTransferObject( (IMMD6generalTO)
                   SomeSpecificTO );
                // Storage is also provided by the GeneralIMMD6handler
                gh.storeAnyTO(_ResponseTimeTO);

                /* FINISHING does:
                 *      - pass ACV correctly (-> final Timestamping)
                 *      - undo the wrapping of the EJB-ReturnValue if
                 *    this is the last interceptor of the framework */
                return gh.finishing();
}       }
```

Listing A.6: Template for the class that realizes generalized interceptor behavior, participating in our LIC

A.1.6. UML of Implemented API

The following figures show the API of the RTI framework implementation that accompanies this thesis. The complete documentation is available in $CD/Javadoc_API-docu and was generated by javadoc. The source code is available as Eclipse project in the $CD/Java-sources/Interceptor/ directory. This section provides UML diagrams for a general overview over the RTI framework API. The diagrams were generated by ESS-Model [ESS], an UML reversing tool for Java, and revised manually.

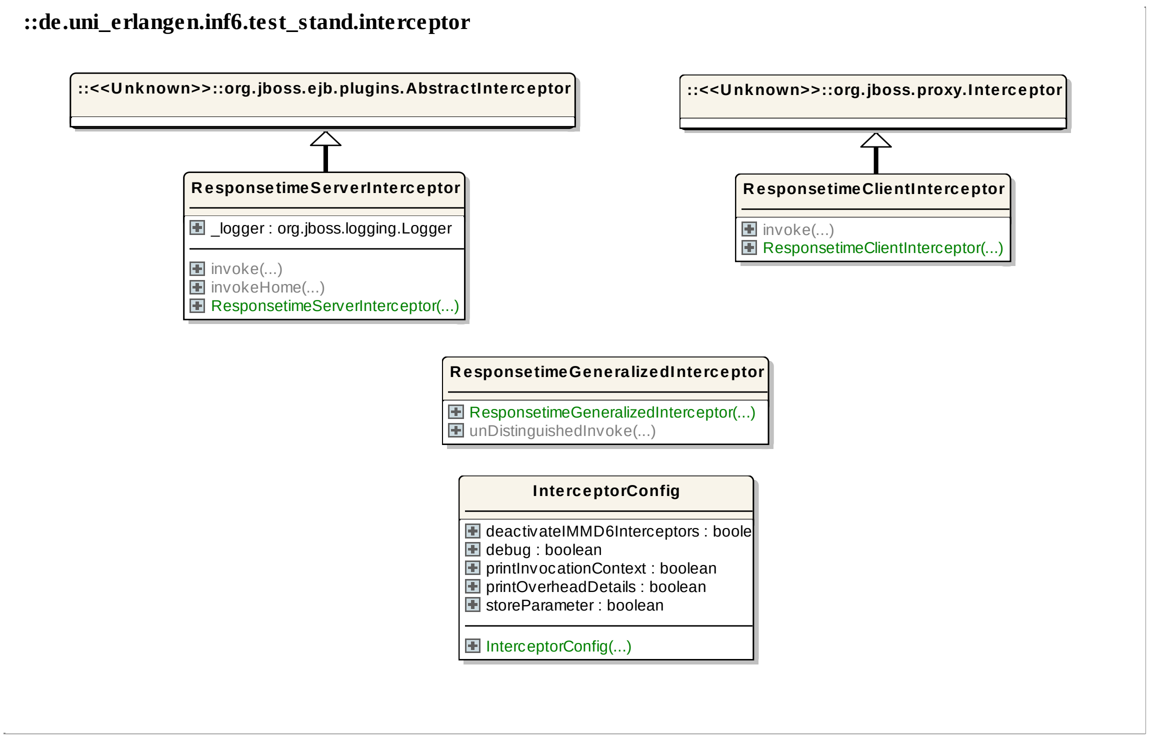

::de.uni_erlangen.inf6.test_stand.interceptor
::<<Unknown>>::org.jboss.ejb.plugins.AbstractInterceptor
::<<Unknown>>::org.jboss.proxy.Interceptor
ResponsetimeServerInterceptor
_logger : org.jboss.logging.Logger
invoke(...)
invokeHome(...)
ResponsetimeServerInterceptor(...)
ResponsetimeClientInterceptor
invoke(...)
ResponsetimeClientInterceptor(...)
ResponsetimeGeneralizedInterceptor
ResponsetimeGeneralizedInterceptor(...)
unDistinguishedInvoke(...)
InterceptorConfig
deactivateIMMD6Interceptors : boole
debug : boolean
printInvocationContext : boolean
printOverheadDetails : boolean
storeParameter : boolean
InterceptorConfig(...)

Figure A.1.: Package: de.uni_erlangen.inf6.test_stand.interceptor

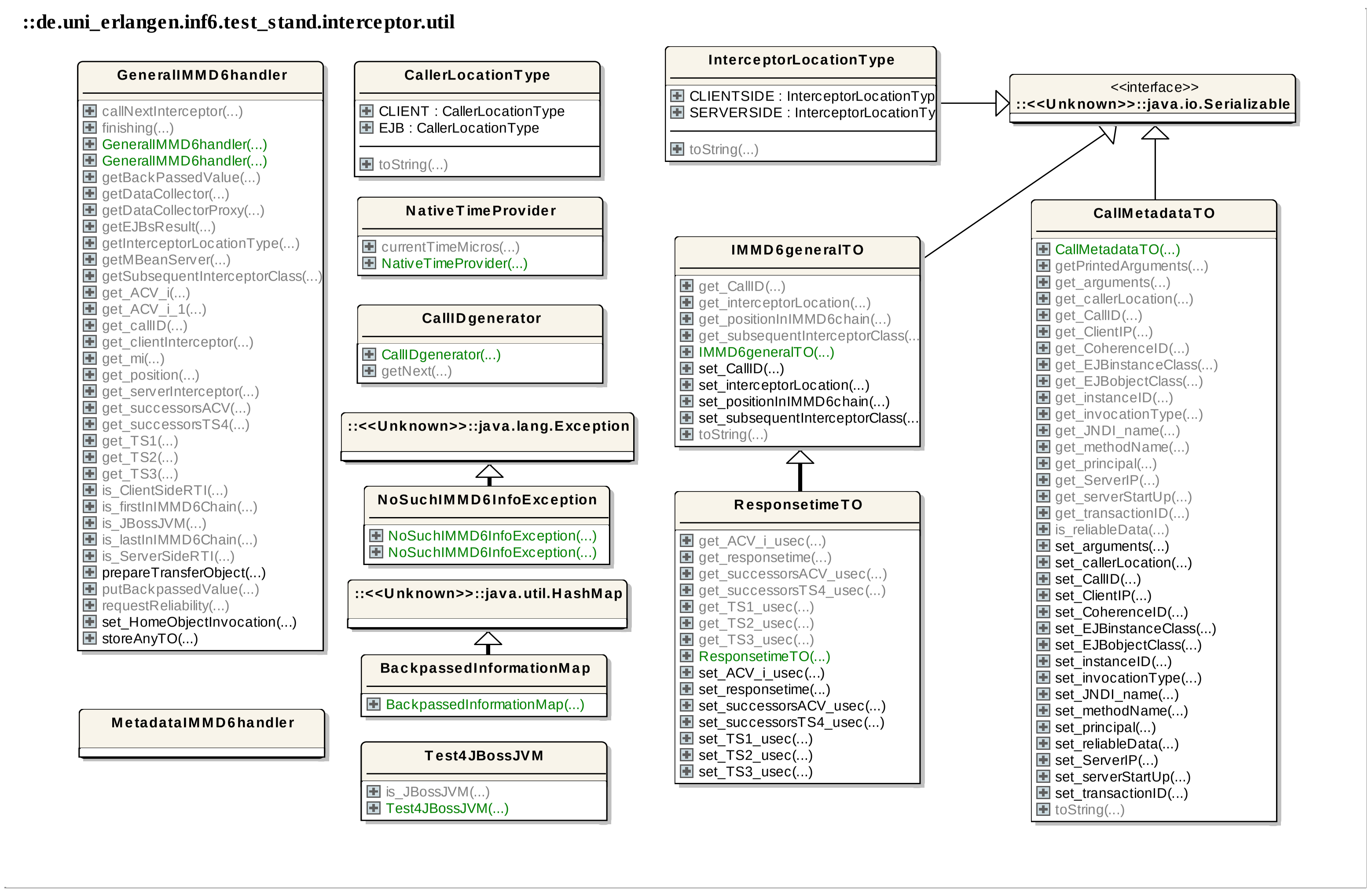


Figure A.2.: Package: de.uni_erlangen.inf6.test_stand.interceptor.util

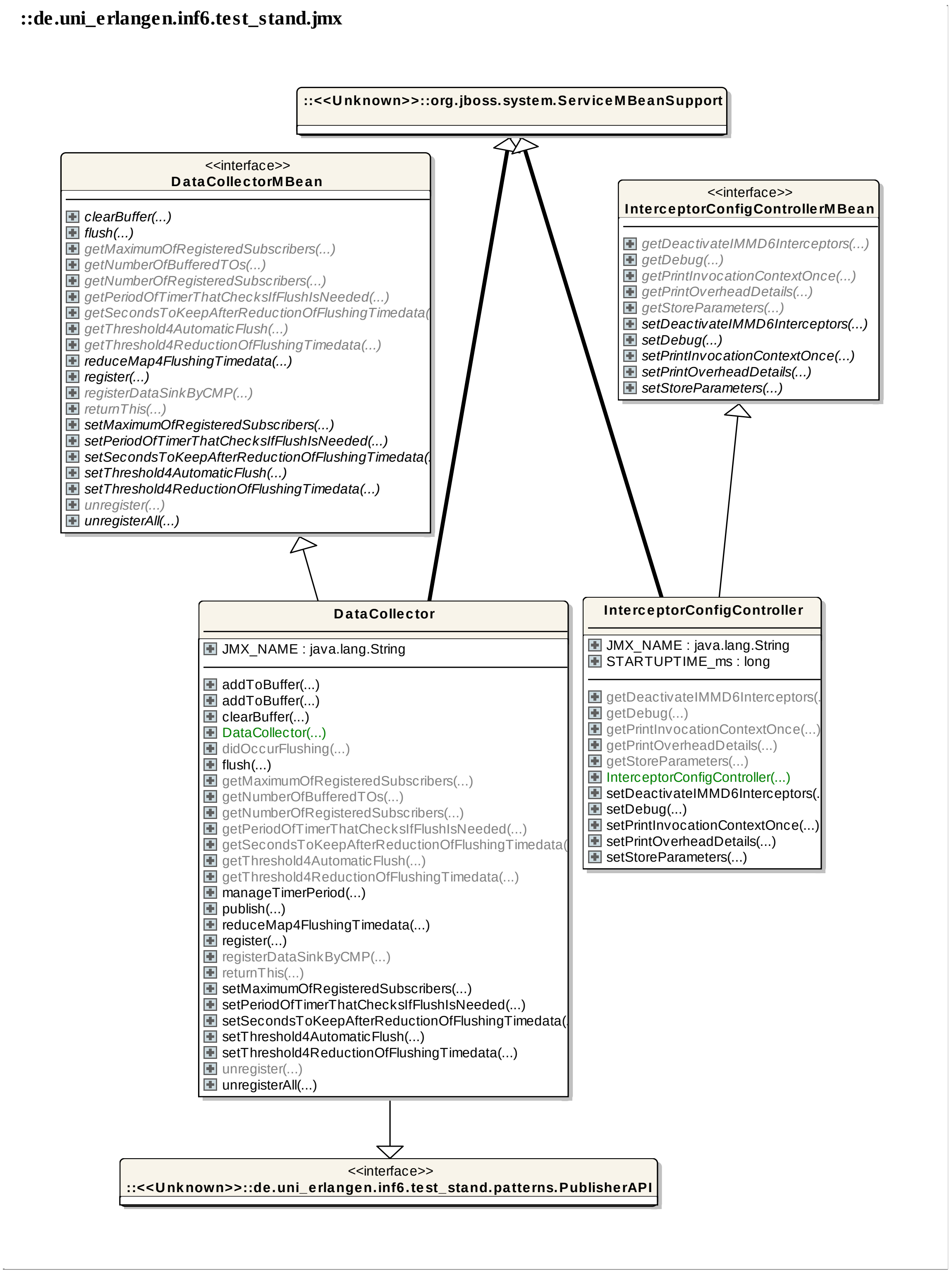

::de.uni_erlangen.inf6.test_stand.jmx
::<<Unknown>>::org.jboss.system.ServiceMBeanSupport
<<interface>>
DataCollectorMBean
clearBuffer(...)
flush(...)
getMaximumOfRegisteredSubscribers(...)
getNumberOfBufferedTOs(...)
getNumberOfRegisteredSubscribers(...)
getPeriodOfTimerThatChecksIfFlushIsNeeded(...)
getSecondsToKeepAfterReductionOfFlushingTimedata(
getThreshold4AutomaticFlush(...)
getThreshold4ReductionOfFlushingTimedata(...)
reduceMap4FlushingTimedata(...)
register(...)
registerDataSinkByCMP(...)
returnThis(...)
setMaximumOfRegisteredSubscribers(...)
setPeriodOfTimerThatChecksIfFlushIsNeeded(...)
setSecondsToKeepAfterReductionOfFlushingTimedata(
setThreshold4AutomaticFlush(...)
setThreshold4ReductionOfFlushingTimedata(...)
unregister(...)
unregisterAll(...)
<<interface>>
InterceptorConfigControllerMBean
getDeactivateIMMD6Interceptors(...)
getDebug(...)
getPrintInvocationContextOnce(...)
getPrintOverheadDetails(...)
getStoreParameters(...)
setDeactivateIMMD6Interceptors(...)
setDebug(...)
setPrintInvocationContextOnce(...)
setPrintOverheadDetails(...)
setStoreParameters(...)
DataCollector
JMX_NAME : java.lang.String
addToBuffer(...)
addToBuffer(...)
clearBuffer(...)
DataCollector(...)
didOccurFlushing(...)
flush(...)
getMaximumOfRegisteredSubscribers(...)
getNumberOfBufferedTOs(...)
getNumberOfRegisteredSubscribers(...)
getPeriodOfTimerThatChecksIfFlushIsNeeded(...)
getSecondsToKeepAfterReductionOfFlushingTimedata(
getThreshold4AutomaticFlush(...)
getThreshold4ReductionOfFlushingTimedata(...)
manageTimerPeriod(...)
publish(...)
reduceMap4FlushingTimedata(...)
register(...)
registerDataSinkByCMP(...)
returnThis(...)
setMaximumOfRegisteredSubscribers(...)
setPeriodOfTimerThatChecksIfFlushIsNeeded(...)
setSecondsToKeepAfterReductionOfFlushingTimedata(
setThreshold4AutomaticFlush(...)
setThreshold4ReductionOfFlushingTimedata(...)
unregister(...)
unregisterAll(...)
InterceptorConfigController
JMX_NAME : java.lang.String
STARTUPTIME_ms : long
getDeactivateIMMD6Interceptors(
getDebug(...)
getPrintInvocationContextOnce(...)
getPrintOverheadDetails(...)
getStoreParameters(...)
InterceptorConfigController(...)
setDeactivateIMMD6Interceptors(
setDebug(...)
setPrintInvocationContextOnce(...)
setPrintOverheadDetails(...)
setStoreParameters(...)
<<interface>>
::<<Unknown>>::de.uni_erlangen.inf6.test_stand.patterns.PublisherAPI

Figure A.3.: Package: de.uni_erlangen.inf6.test_stand.jmx

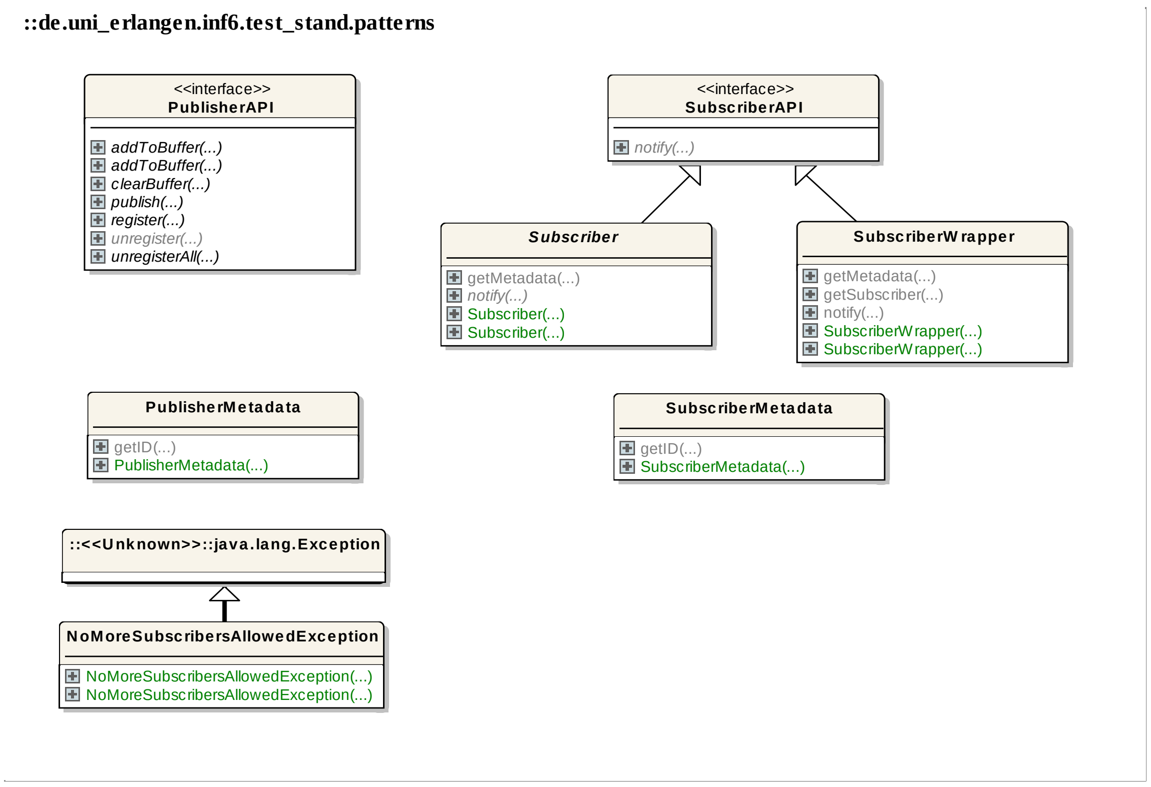


Figure A.4.: Package: de.uni_erlangen.inf6.test_stand.patterns

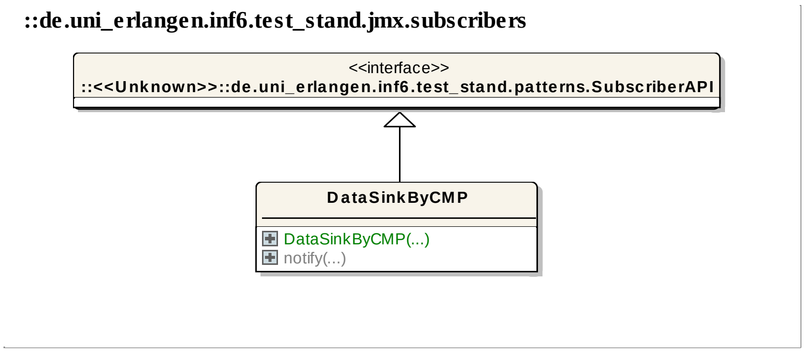

::de.uni_erlangen.inf6.test_stand.jmx.subscribers
<<interface>>
::<<Unknown>>::de.uni_erlangen.inf6.test_stand.patterns.SubscriberAPI
DataSinkByCMP
DataSinkByCMP(...)
notify(...)

Figure A.5.: Package: de.uni_erlangen.inf6.test_stand.jmx.subscribers

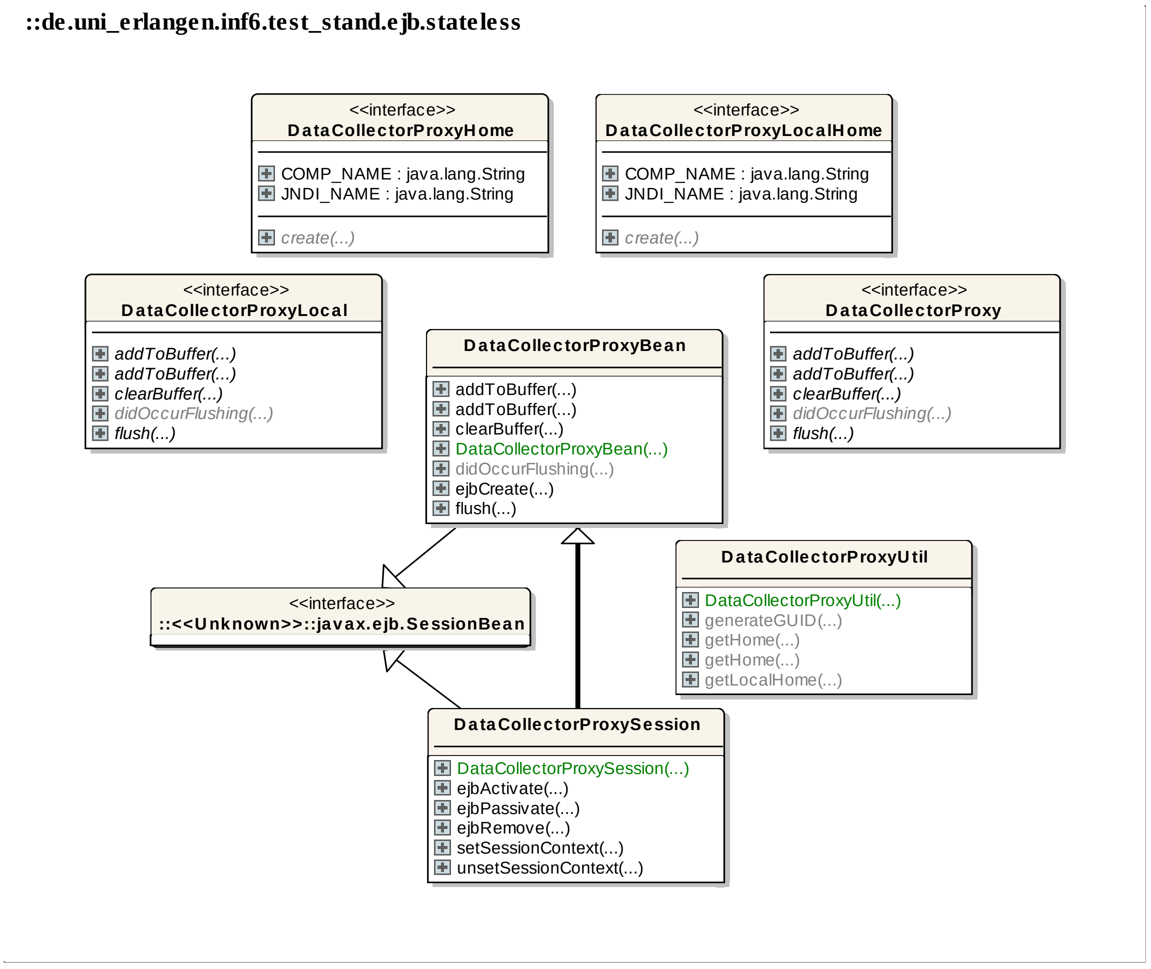


Figure A.6.: Package: de.uni_erlangen.inf6.test_stand.ejb.stateless

A.1.7. Implementing an MBean

The Java source code in Lst. A.7 can be used as template for the implementation of custom MBeans. The exemplary MBean exports a method returnThis() that returns a direct object reference that can be used inside the same JVM. Running XDoclet will generate the TheTemplateMBean interface. The templates are also available in the $CD/Java-sources/Interceptor/src/server/de/uni_erlangen/inf6/test_stand/jmx/ and /jmx/util/ directories. Listing A.8 provides the template for a class that wants to access the MBean.

```
package de.uni_erlangen.inf6.test_stand.jmx;
import org.jboss.system.ServiceMBeanSupport;
/**
 * @jmx.mbean
 *   name="immd6:service=TheTemplate"
 *   extends="org.jboss.system.ServiceMBean"
 * @jboss.service servicefile="jboss"
 */
public class TheTemplate
    extends ServiceMBeanSupport
    implements de.uni_erlangen.inf6.test_stand.jmx.TheTemplateMBean {

    public static final String JMX_NAME = "immd6:service=TheTemplate";

    /** @jmx.managed-operation */
    public void manageStuff() {
        // ...
    }

    /**
     * Returns a direct object reference to the MBean.
     * A direct call is more performant in comparison with a JMX-call,
     * and allows to call non-managed public functions.
     * (The refernce will only be valid in the local JVM!)
     *
     * @return TheTemplate as a this-reference
     *
     * @jmx.managed-operation
     */
    public TheTemplate returnThis() {
        return this;
    }
}
```

Listing A.7: Template for the implementation of an MBean using XDoclet

```
package de.uni_erlangen.inf6.test_stand.jmx.util;

import de.uni_erlangen.inf6.test_stand.jmx.TheTemplate;

import java.util.ArrayList;

import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;
import javax.management.ObjectName;

/** (Server-side only!) */
public class Template4MBeanAccess {

    /**
     * The (local-VM) facility by which MBeans can be accessed.
     */
    static protected MBeanServer _MBeanServer = null;

    /**
     * The MBean as a direct ojbect-reference.
     */
    static protected TheTemplate _theTemplate = null;

    /**
     * Obtains and caches a reference to the MBean server.
     *
     * @return MBeanServer : the facility (on local-JVM) by which
         MBeans can be accessed
     */
    static public MBeanServer getMBeanServer() {
        if (_MBeanServer == null) {
            ArrayList mbeanServers = MBeanServerFactory.findMBeanServer
                (null);
            _MBeanServer = (MBeanServer) mbeanServers.get(0);
        }
        return _MBeanServer;
    }

    /**
     * Obtains and caches a direct object-reference to the
     * <code>TheTemplate</code> (MBean).
     *
     * @return TheTemplate : some MBean
     */
    static public TheTemplate getTheTemplate() {
        if (_MBeanServer == null)
            getMBeanServer();
        if (_theTemplate == null)
        try {
            // for retrieval an ObjectName is needed:
            ObjectName objName = new ObjectName(TheTemplate.JMX_NAME);
            // get the Object-reference (usable only in the same JVM)
            _theTemplate = (DataCollector) _MBeanServer.invoke(
                    objName, "returnThis", null, null);
        } catch (Exception e) {
            e.printStackTrace();
        }

        return _theTemplate;
    }
}
```

Listing A.8: Template for a class that accesses the MBean

A.2. Call Dependency Framework

At first, some resources are provided, which were referenced in the main text. Then, an overview over the implemented classes is provided as UML diagrams.

A.2.1. Additionally Referenced Sources

The following two pages provide on the one hand the SQL to generate the temporal-order view at the RTI data, that equals the callstack representation of the events, in Lst. A.9. On the other hand a visualization of the CallID, Transaction ID, CoherenceID, test-series and sensor instances is provided in Fig. A.7. Both were referenced in the main text, but they are placed in the appendix, because of their size and because they are not essential for the understanding.

```
CREATE TABLE h_IN  (direction VARCHAR(2))
CREATE TABLE h_OUT (direction VARCHAR(3))
INSERT INTO  h_IN  (direction) VALUES ('in')
INSERT INTO  h_OUT (direction) VALUES ('out')

CREATE VIEW v_NumberOfRTIsPerInvocation AS
   (SELECT callID, count(RTIinstanceID) AS NumberOfRTIs
    FROM raw_RTI
    GROUP BY callID)

CREATE VIEW v_callStack AS
    SELECT *
    FROM (
        SELECT callID, EJBinstanceClass AS EJB, method, InstanceID,
            TS2  AS time, ACV_I AS ACV, direction,
            invocationtype, callerLocation, CoherenceID
        FROM v_joined j, h_IN
        WHERE ((callerLocation = 'EJB' AND positionInIMMD6chain = 1)
            OR  (callerLocation = 'CLIENT' AND
                  positionInIMMD6chain in
                       (SELECT NumberOfRTIs AS lastPosition
                       FROM v_NumberOfRTIsPerInvocation k
                       WHERE k.callID = j.callID)
              ))
        UNION ALL
        SELECT callID, EJBinstanceClass AS EJB, method, InstanceID,
            TS3 AS time, SuccessorsACV AS ACV, direction,
            invocationtype, callerLocation, CoherenceID
        FROM v_joined l, h_OUT
        WHERE ((callerLocation = 'EJB' AND positionInIMMD6chain = 1)
            OR  (callerLocation = 'CLIENT' AND
                  positionInIMMD6chain in
                       (SELECT NumberOfRTIs AS lastPosition
                        FROM v_NumberOfRTIsPerInvocation m
                        WHERE m.callID = l.callID)
              ))
        )
    ORDER BY time ASC
```

Listing A.9: SQL operations for temporal order of “in” and “out” events on basis of the RTI-provided data

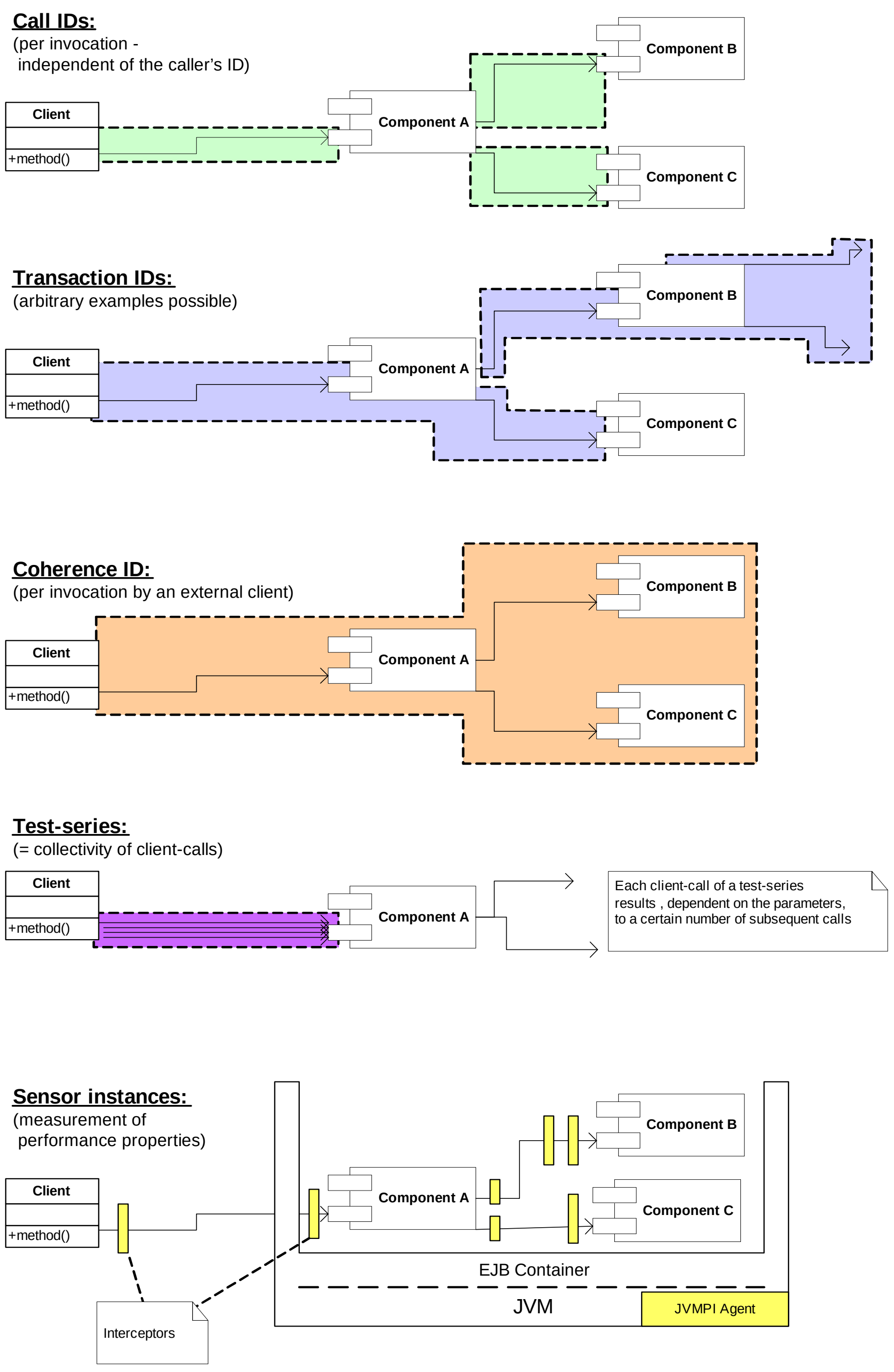

Call IDs:
(per invocation - independent of the caller's ID)
Client
+method()
Component A
Component B
Component C
Transaction IDs:
(arbitrary examples possible)
Coherence ID:
(per invocation by an external client)
Test-series:
(= collectivity of client-calls)
Each client-call of a test-series results , dependent on the parameters, to a certain number of subsequent calls
Sensor instances:
(measurement of performance properties)
Interceptors
EJB Container
JVM
JVMPI Agent

Figure A.7.: Comparison of CallID, Transaction ID and CoherenceID as well as test-series and sensor instances

A.2.2. UML of Implemented API

The following figures show the API of the call dependency framework implementation that accompanies this thesis. The complete documentation is available in $CD/Javadoc_API-docu and was generated by javadoc. The source code is available as Eclipse project in the $CD/Java-sources/IMMD6_Util/ directory. This section provides UML diagrams for a general overview over the call dependency framework. The diagrams were generated by ESS-Model [ESS], an UML reversing tool for Java, and revised manually.

There is also provided the source code for an exemplary class that instantiates the Analyzer and requests the generation of InvocationTree and InvocationMatrix for a filtered set of coherency scopes (in Lst. A.10). A class that is implemented like the template needs the $CD/analyzer/immd6_util.jar in its classpath.

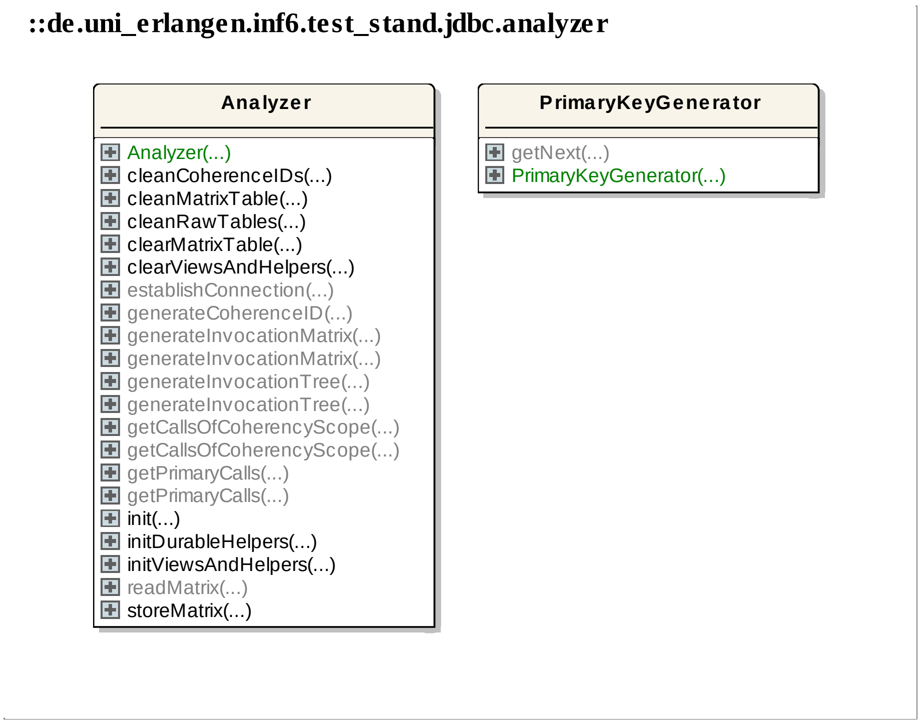


Figure A.8.: Package: de.uni_erlangen.inf6.test_stand.jdbc-analyzer

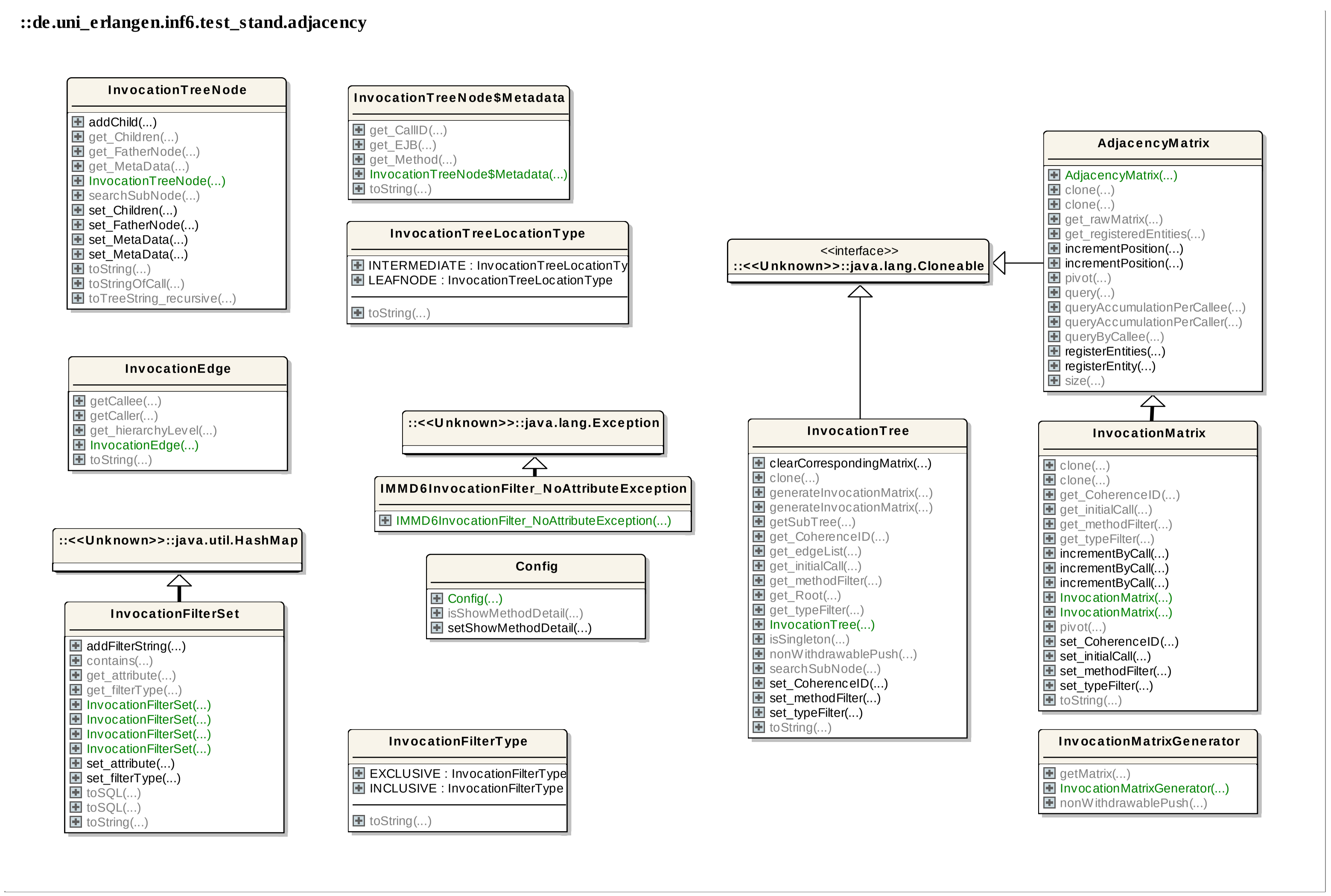


Figure A.9.: Package: de.uni_erlangen.inf6.test_stand.adjacency

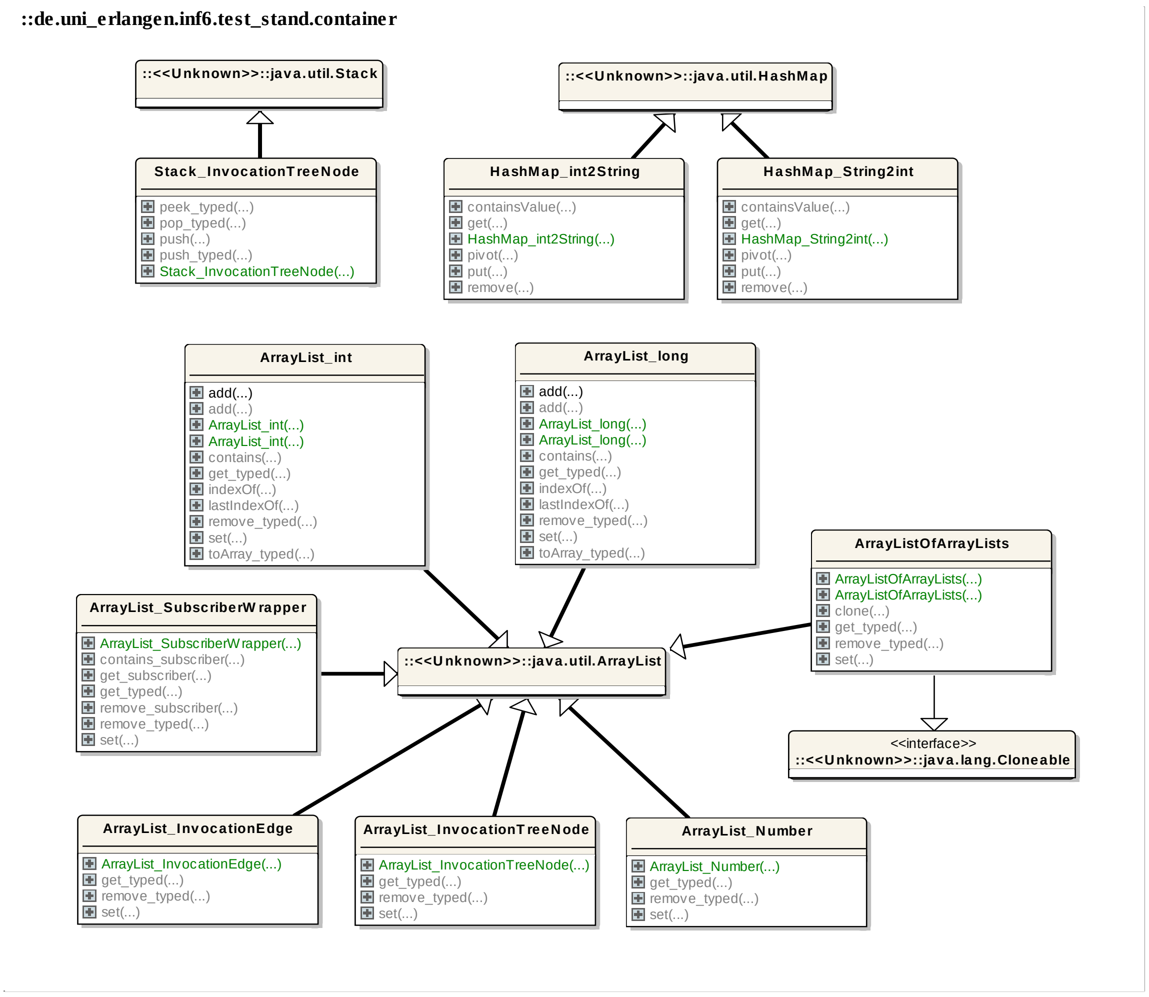


Figure A.10.: Package: de.uni_erlangen.inf6.test_stand.container

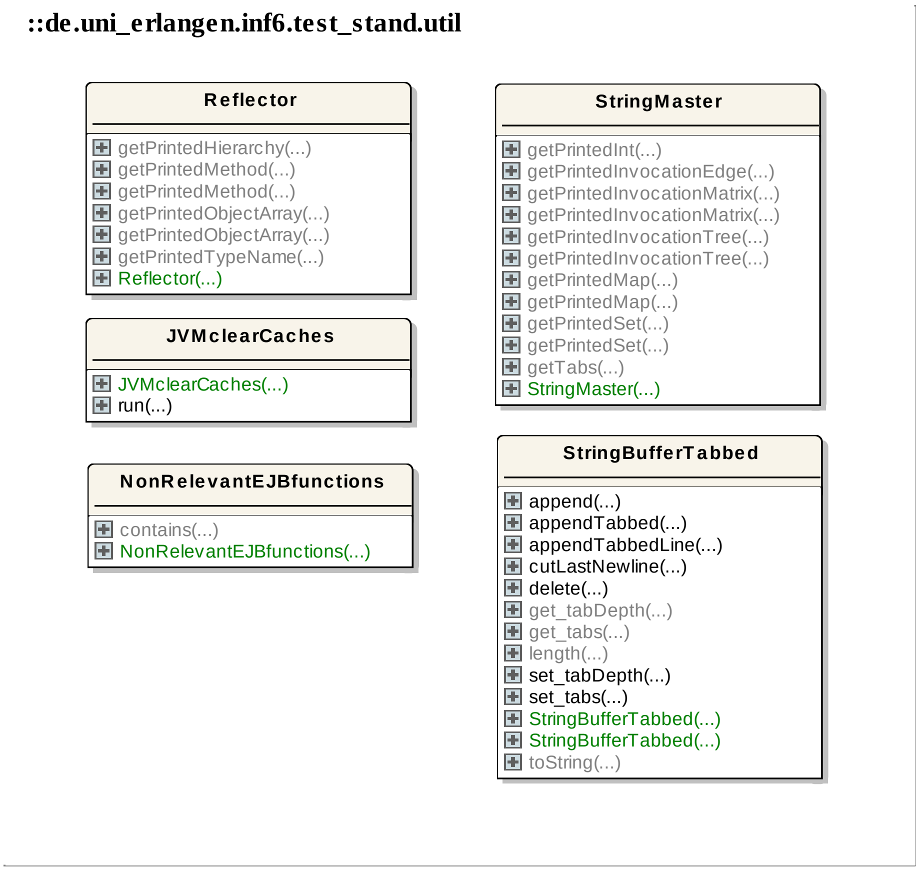


Figure A.11.: Package: de.uni_erlangen.inf6.test_stand.util

```
import de.uni_erlangen.inf6.test_stand.adjacency.InvocationFilterSet;
import de.uni_erlangen.inf6.test_stand.adjacency.InvocationFilterType;
import de.uni_erlangen.inf6.test_stand.adjacency.InvocationMatrix;
import de.uni_erlangen.inf6.test_stand.adjacency.InvocationTree;
import de.uni_erlangen.inf6.test_stand.jdbc.analyzer.Analyzer;

public class Template4PrintTreeAndMatrix {

public static void main(String[] args) {

    Analyzer analyzer = new Analyzer();

    // Filters (method name)
    InvocationFilterSet clientMethodFilter =
        new InvocationFilterSet(InvocationFilterType.INCLUSIVE);
    clientMethodFilter.addFilterString("myEJBsMethodName");
    // Filter (invocation type)
    InvocationFilterSet clientTypeFilter = null;

    // Get list of appropriate CallIDs of remote client invocations
    long[] client_calls =
        analyzer.getPrimaryCalls(clientMethodFilter, clientTypeFilter);

    // generate CoherenceID and build tree and matrix
    for (int i = 0; i < client_calls.length; ++i) {
        // get (or generate) the according CoherenceID
        long coherenceID = analyzer.generateCoherenceID(client_calls[i]);

        // [ filter inside the single coherency scope: ]
        // Filter (method name)
        InvocationFilterSet methodFilter =
            new InvocationFilterSet(InvocationFilterType.EXCLUSIVE);
        methodFilter.addFilterString("IdoNotWantToSeeThisMethod");
        // Filter (invocation type)
        InvocationFilterSet typeFilter =
            new InvocationFilterSet(InvocationFilterType.EXCLUSIVE);
        clientTypeFilter.addFilterString("Home");
        clientTypeFilter.addFilterString("LocalHome");

        InvocationTree tree =
            analyzer.generateInvocationTree(
                coherenceID,
                methodFilter,
                typeFilter);
        System.out.println(tree.toString());
        InvocationMatrix matrix =
            analyzer.generateInvocationMatrix(
                coherenceID,
                methodFilter,
                typeFilter);
        System.out.println(matrix.toString());

        if (false) {
            analyzer.storeMatrix(matrix);
            InvocationMatrix readMatrix = analyzer.readMatrix(coherenceID)
                ;
        }
    }
    System.out.println("[Analyzer:] END");
}
}
```

Listing A.10: Template for the usage of the Analyzer

A.3. Statistics

The statistics for the run-time of gettimeofday(2) in Sect. 2.3.4 has been created with the source code in Lst. A.11. It resulted in an average of $0.254\mu s$.

Both statistics in Sect. 5.2.7, for thread creation and JNI, and the statistics in Sect. 5.2.9, for the time gap between the call to the constructor of the GeneralIMMD6handler and the timestamping as first command inside it, have been created with the same methodology, shown in Lst. A.12. The StreamedStatistics class is part of the de.uni_erlangen.inf6.test_stand.statistics package. The API of the package is given in Fig. A.12.

The specific time taking for the overhead estimation of the JNI access to gettimeofday(2) is given in Lst. A.13. It resulted in an average of $0.970\mu s$.

Estimating the overhead for thread creation can be done with the t0 and t1 of Lst. A.14. It resulted in an average of $289.325\mu s$.

The time gap between the call to the constructor of the GeneralIMMD6handler and the timestamping as first command inside it can be measured without instantiating EJBs, using the source code in Lst. A.15. It resulted in an average of $3.891\mu s$

```
#include <iostream>
#include <sys/time.h>
using std::cout;
using std::endl;
int main(int argc, char* argv[]){
  struct timeval t0;
  struct timeval t1;

  int size = 10000;
  long sum = 0;
  for (int i=0; i<size; ++i){
    gettimeofday(&t0, NULL);
    gettimeofday(&t1, NULL);
    long lag = (t1.tv_usec - t0.tv_usec);
    sum += lag;
  }
  cout << "Mean average: " << ((float) sum / (float) size) << endl;
  return 0;
}
```

Listing A.11: Statistics for gettimeofday(2)

```
import de.uni_erlangen.inf6.test_stand.interceptor.util.
    NativeTimeProvider;
import de.uni_erlangen.inf6.test_stand.statistics.AbstractStatistics;
import de.uni_erlangen.inf6.test_stand.statistics.StreamedStatistics;
...
public static void main() {
        StreamedStatistics stats =
                new StreamedStatistics(AbstractStatistics.
                    IgnoreFirstValue.TRUE);
                for (int i = 0; i <= 10000; ++i) {
                long t0 = // take T_0
                // (possibly some execution ...)
                long t1 = // take T_1
                stats.next(new Long(t1 - t0));
        }
        System.out.println("[Statistics:] " + stats);
}
```

Listing A.12: General Java scheme for overhead statistics

```
long t0 = NativeTimeProvider.currentTimeMicros();
long t1 = NativeTimeProvider.currentTimeMicros();
```

Listing A.13: JNI overhead accessing gettimeofday(2)

```
long t0 = NativeTimeProvider.currentTimeMicros();
Thread t = new Thread(new Runnable() {
        public void run() {
                int dummy = 1; // dummy
        }
});
t.start();
long t1 = NativeTimeProvider.currentTimeMicros();
```

Listing A.14: Thread creation overhead

```
/* additional "imports" are needed:
 * import org.jboss.invocation.Invocation;
 * import de.uni_erlangen.inf6.test_stand.interceptor.
     ResponsetimeServerInterceptor;
 * import de.uni_erlangen.inf6.test_stand.interceptor.util.
     GeneralIMMD6handler;
 */
Invocation mi = new Invocation();
ResponsetimeServerInterceptor RTI =
    new ResponsetimeServerInterceptor();

long t0 = NativeTimeProvider.currentTimeMicros();
GeneralIMMD6handler gh = new GeneralIMMD6handler(mi, true, RTI);
long t1 = gh.get_TS1();
```

Listing A.15: Time gap concerning TS1 of GeneralIMMD6handler

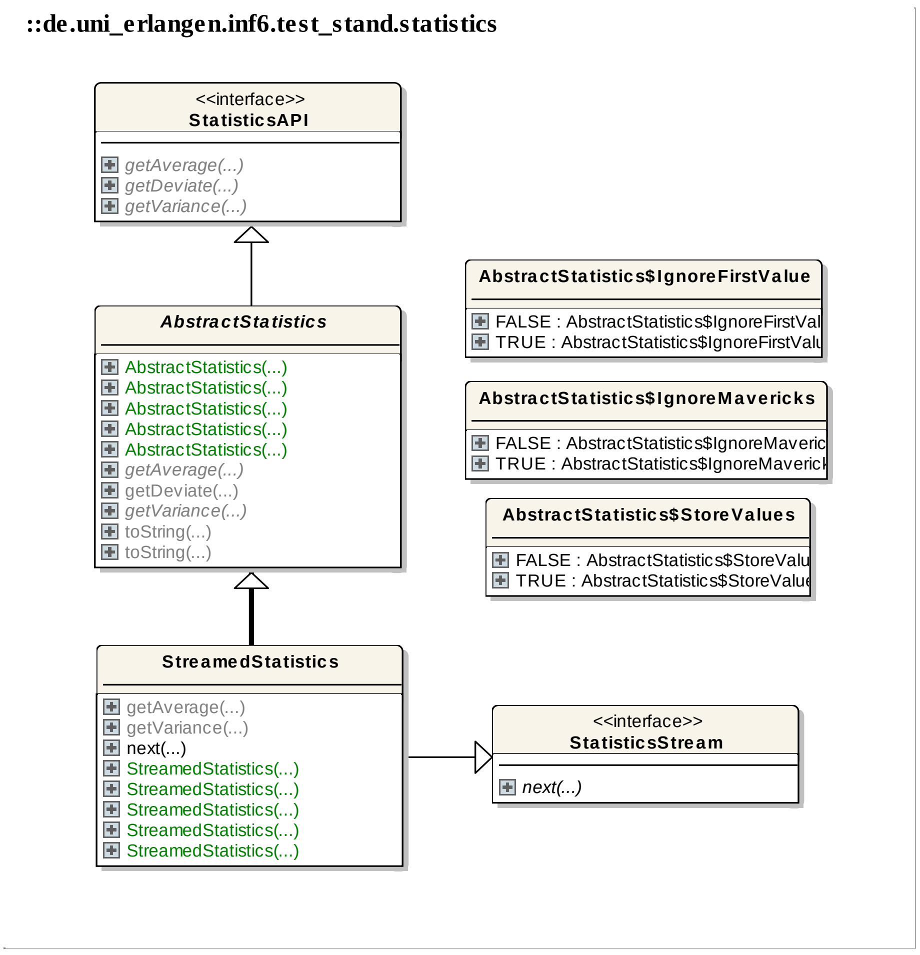


Figure A.12.: Package: de.uni_erlangen.inf6.test_stand.statistics

Appendix B.

Experiences

In this appendix chapter we want to pass several experiences to the reader that we have made with JBoss and our development environment.

At first we describe our development environment. Then we make a short description of our efforts to port the Börsenticker application of the COMQUAD project to JBoss – this will be not relevant to readers that are not part of the COMQUAD project. After this, we give a step-by-step introduction to the Lomboz plug-in, explain the activation of profiling and debugging for JBoss and reference the JBoss specific XDoclet documentation and the Ant framework that was used for the development of MBeans. Finally, a short guide to JAAS security activation for JBoss is provided.

B.1. System Environment

This section denotes the development environment in more detail. It does provide reference to the basic applications, required or helpful environment variables and settings for the classpath variable for EJBs or clients.

B.1.1. Basic Applications

- JDK/JRE: Sun J2SE 1.4.2 – free reference implementation
  http://java.sun.com/j2se/downloads.html

- J2EE server and EJB container: JBoss 3.2.1 – open source
  http://www.jboss.org/
- Java-IDE: Eclipse 2.1 – open source
  http://download.eclipse.org/downloads/index.php
- Eclipse plug-ins: see Sect. B.3

B.1.2. Environment Variables

The variables in Lst. B.1 were preliminary to some shell-scripts. Additional auxiliary variables, that will be referenced in further sections, are provided by Lst. B.2.

```
JAVA_ROOT    (=/usr/lib/SunJava2-1.4.2)
JAVA_HOME    (=$JAVA_ROOT)
JAVA_BINDIR  (=$JAVA_ROOT/bin)
JDK_HOME     (=$JAVA_ROOT)
JRE_HOME     (=$JAVA_ROOT/jre)
JBOSS_DIST   (=/opt/jboss)
```

Listing B.1: Variables that were preliminary to some shell-scripts

```
ECLIPSE  (=/opt/eclipse)
JBOSS    (=$JBOSS_DIST)
DEPLOY   (=$JBOSS_DIST/server/default/deploy)
```

Listing B.2: Additional auxiliary variables

B.1.3. Classpath Variable

This section describes our experience with the jar files that are needed in the classpath for developing EJBs, and developing or invoking client applications.

B.1.3.1. EJBs

For a simple SessionBean or CMP-EntityBean only $JBOSS/server/default/lib/jboss.jar has to be in the classpath. For example for JBoss-logging it is also recommended to add $JBOSS/client/jbossall-client.jar.

B.1.3.2. Clients

We recommend to use $JBOSS/client/jbossall-client.jar for clients. Lomboz does not include this specific file, but several individual files from the $JBOSS/client/ directory, which equal as a whole the jbossall-client.jar coverage. Naturally, the client needs the interface classes of the EJB additionally available in its classpath.

The $JBOSS/server/default/lib/jboss-transaction.jar seems to be needed if debugging or profiling is activated.

B.2. Börsenticker

The Börsenticker is a very early test-application of the COMQUAD project. It was developed for the Sun reference implementation of the J2EE server, and we tried to port it to the JBoss environment.

The following sections provides an overview over the contents of the Boersenticker.ear file, that was available to us, and the modifications we have enforced.

B.2.1. Content of the EAR Archive

Boersenticker.ear consists of:

- META-INF (directory)
- app-client-ic.jar (BoersentickerClient)
  - Notice Sect. B.2.3
  - Start it with: `java -jar app-client-ic.jar` ⇒ Graphical User Interface
- app-client-ic8.jar (myTimer)
  - Start it with: `java -jar app-client-ic8.jar`
- ejb-jar-ic.jar (Subscription Manager)
  - SubscriptedStockBean
  - SubscriptionManagerBean
- ejb-jar-ic9.jar (StockQuotationManager)
  - Subscription2Bean
  - StockBean

  - StockQuotationManagerBean
- ejb-jar-ic97.jar (StockQuotationDistributor)
  - SubscriptionBean
  - StockQuotationDistributorBean

B.2.2. Porting to JBoss

All ejb-jar-*.jar files contain among other things a META-INF directory. There has to be created a jboss.xml file. The <ejb−name> as defined in ejb-jar.xml must be used inside this jboss.xml as "foreign key".

We provide the completely enriched ear file in accompany to this thesis.

B.2.3. Börsenticker-Client

Because the InitialContext-properties are not hard-coded there must be created (and included to the classpath) a jndi.properties file in the classpath of the client – similar to:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=localhost
java.naming.factory.url.pkgs=org.jboss.naming
```

B.2.4. Conclusion

Finally, the ear file seemed to deploy correctly, but we did not succeed to port the Börsenticker, because the client application terminates with an "java.rmi.MarshalException: error marshalling arguments; nested exception is: java.io.NotSerializableException: boersenticker_client.ClientDisplayImpl".

We did not investigate this problem further, but contacted the initial developer of the Börsenticker. We hope that our descriptions help other Comquad members to complete this porting more easily. Therefore, we provide the completely enriched Börsenticker files for JBoss in the $CD/Comquad-related/Boersenticker/ directory.

B.3. Eclipse Plug-ins

This section names several Eclipse plug-ins that we used to develop our framework and test-EJBs. They support XML, remote control of the JBoss instance, auto-completion for XDoclet javadoc-tags and classic profiling support.

B.3.1. Used Plug-ins

- jboss-IDE – Start/Stop/Debugging integration of JBoss and XDoclet support like auto-completion
  http://eclipse-plugins.2y.net/eclipse/plugin_details.jsp?id=255
- XMLbuddy – XML support
  http://eclipse-plugins.2y.net/eclipse/plugin_details.jsp?id=131
- Lomboz – J2EE support (automatic creation of XDoclet information that leads to automatic creation of interface classes and deployment descriptors)
  http://www.objectlearn.com/index.html
- JUnit plug-in – JUnit support
  http://eclipse-plugins.2y.net/eclipse/plugin_details.jsp?id=18
- Eclipse Profiler Plugin – graphical profiler support in Eclipse
  http://eclipsecolorer.sourceforge.net/index_profiler.html

B.3.2. Lomboz

For developers that are new to EJBs we recommend the usage of the Lomboz plug-in for Eclipse. We provide a step-by-step documentation after copying the Lomboz files to the plugin directory of Eclipse.

B.3.2.1. Preferences

In “Preferences ≫ Lomboz ≫ Server Definitions” there has to be selected “Server types: JBOSS 3.0”. The “Application Server Directory” and “Classpath Variable” has to be adopted, for example to /opt/jboss-3.2.1.

BUG: The path strings should contain NO secluding “/”, otherwise several problems occurred in the used version.

Furthermore, there should be done the necessary changes to the classpath properties, in the Lomboz preferences, according to Sect. B.1.3. But they will take effect only for projects that are created from now on. Already existent projects with wrong classpaths have to be adopted individually. The available version of Lomboz adds the classpaths for EJBs only if during the creation of the Lomboz project an EJB-module was created. If this was not done, the classpaths have to be added to the projects properties by hand.

B.3.2.2. EJB Creation

Use the “File ≫ New ≫ Lomboz EJB Creation Wizard” to create an EJB, for example a stateless session bean. EJBs must have assigned a package. Otherwise the XDoclet auto-creation will create a “package ;”-line, which does not compile.

The EJB file is generated below the scr directory. All logic has to be implemented in this file. Later there will be generated files in the ejbscr directory, but do never change them, because they are *generated*, and all changes will soon be lost. Any desired change can be performed by the files in the scr directory, possibly with the appropriate usage of XDoclet tags.

B.3.2.3. Subsequent Steps

Before “Generate EJB Classes” is used the first time the ejbGenerate.xml in the META-INF directory of the Lomboz module should be edited: all not needed application server tags can be commented and the <jboss>-tag attributes could need adoption:

```
<jboss ...
        datasource (="java:/DefaultDS")
        datasourcemapping (="Hypersonic SQL")
        preferredrelationmapping (="relation-table") />
```

B.3.2.4. Generate EJB Classes

Created EJBs have to be added to an Lomboz module, which is done by “Lomboz J2EE... ≫ Add EJB to module...” from the context menu of the EJB’s java class.

After that, the “Lomboz J2EE... ≫ Generate EJB Classes” can be used to run the XDoclet task for the added EJB-files.

As long as the added java files have the necessary XDoclet tags and comply to XDoclet’s conditions for EJB prototypical files, there are created java files for the EJB Instance, the Home Object and the EJB Object (= Remote Object) in the project’s ejbsrc directory. Furthermore the EJB descriptor ejb-jar.xml and the application dependent descriptors, like jboss.xml and jbosscmp-jdbc.xml, are created in the META-INF directory of the module.

B.3.2.5. Configure EJBs for Interceptors

The EJB-Module specific <container−configurations> should be stored in META-INF/jboss-container.xml and will so be automatically merged by XDoclet. In order to specify which configuration is in effect for a concrete EJB use the XDoclet tag @jboss: container−configuration name=... in the EJB-class’ javadoc-comment. See Sect. A.1.4.3 for more details.

B.3.3. Clients

EJB and clients should be created in separate Eclipse projects. The client project has to be linked with the EJB project with “(Project) Properties ≫ Java Build Path ≫ Projects”. Using the “Lomboz EJB Test Client Wizard” to create the concrete client’s java class will automatically add the classpath entries for clients from the Lomboz’ settings (see Sect. B.1.3.2 and B.3.2.1).

NOTICE: If the client is created manually (which is very easy with the XDoclet-generated *Util-class) the jbossall-client.jar has to be added to the project’s libraries before Eclipse is able to auto-complete methods like remove() from the EJBObject, which is inherited by most Beans, because importing another project will only import its source codes and not the libraries (in the default settings), so for resolving the inheritance hierarchy the appropriate library has to be present again.

If the XDoclet-generated *Util-class is used for the development of client applications, it is necessary to add the @ejb. util generate="physical" XDoclet tag to the EJB’s class. This makes the *Util-class to lookup the JNDI with a name relative to "java:comp/env". Because, for JBoss the absolute JNDI path name is only allowed at server-side, for example by other EJBs or JSPs, and not for remote client applications.

Before using the "Lomboz EJB Test Client Wizard", the interface classes of the EJB should exist – therefore an "Generate EJB Classes" should have run in the EJB project.

Providing the correct InitialContext for clients accessing EJBs is easiest with a jndi.properties file. It should reside in the directory where the client is executed. For Eclipse it is best to place it in the /src directory of the project (or the projects root directory, if sources and class-files are not in separate directories – whether they are separated or not is controlled by "Window ≫ Preferences" and then "Java ≫ "New Project").

B.3.4. Importing External Lomboz-Projects

In most cases the path- and application server-setting will be wrong. At first, correct the project's path-entries for jar-libraries. Then look at build.properties and tagets.xml in the project's ⟨module⟩/META-INF/ directory and correct the properties. It will also be necessary to edit/(un-)comment the ejbGenerate.xml's XDoclet tasks, if a complete different application server is to be used.

B.3.5. Deploying

The deploy-button of Lomboz will neither execute a "Generate EJB Classes" nor a "Rebuild Project". It only packs the needed classes into a ⟨module-name⟩.jar and copies it to the JBoss deploy directory.

A "Generate EJB Classes" is only necessary if the interface of the EJB has changed since it was invoked last, that means if methods were added or dropped, or it was never invoked for the project before. A "Project ≫ Rebuild Project" should always be invoked before deploying.

B.3.6. Profiling

The Eclipse Profiler Plugin is a classic JVMPI profiler, that can be used to profile the JBoss server, because it allows to add Java package filters with the start-up command. This is necessary, because otherwise the start-up takes a lot of time and the proximate performance is drastically reduced.

B.3.6.1. JBoss Adoption

In order to use the Eclipse Profiler Plugin for JBoss, the JBoss start-up properties have to be adopted. The files jakarta-regexp.jar, bcel.jar, profiler_trace.jar and commons-lang.jar (all from the $ECLIPSE/plugins/ru.nlmk.eclipse.plugins.profiler/ directory) have to be added to the "java −Xbootclasspath/a:\ldots" command. This is can be done within the $JBOSS_HOME/bin/run.conf, as shown in Lst. B.3.

```
PROFILER_HOME=$ECLIPSE/plugins/ru.nlmk.eclipse.plugins.profiler
PROFILER_JARS="$PROFILER_HOME/jakarta-regexp.jar:
               $PROFILER_HOME/bcel.jar:
               $PROFILER_HOME/profiler_trace.jar:
               $PROFILER_HOME/commons-lang.jar"
PROFILER_PACKAGE_FILTER="__A__org.jboss.Main:__M__sun.:
                         __M__com.sun.:__M__java.:__M__javax."
JAVA_OPTS="$JAVA_OPTS -XrunProfilerDLL:1
           -Xbootclasspath/a:$PROFILER_JARS
           -D__PROFILER_PACKAGE_FILTER=$PROFILER_PACKAGE_FILTER
           -D__PROFILER_USE_PACKAGE_FILTER=1
           -D__PROFILER_TIMING_METHOD=1"
```

Listing B.3: Activation of profiling support to the JBoss JVM; with the profiler agent of the Eclipse Profiler Plugin

B.3.6.2. Eclipse Settings

In order to connect the Eclipse plug-in to the profiler agent a "Remote Profiler"-Configuration ("Run ≫ Run ...") must be created:
Project: <Eclipse project with the classes to be profiled in principle> (this has not restricting impacts)
Host: <IP of the JBoss server> (e.g. localhost)

B.3.7. Debugging

The JBoss JVM can be debugged like any other other application, if JVMDI is activated. Generally the debugging support of the JVM does not introduce very much overhead.

B.3.7.1. JBoss Adaptation

To activate debugging support of the JBoss JVM, the $JBOSS_HOME/bin/run.conf has to be adopted according to Lst. B.4.

```
JAVA_OPTS="$JAVA_OPTS -Xdebug
           -Xrunjdwp:transport=dt_socket,
                     address=<arbitrary, e.g. 8000>,
                     server=y,suspend=n"
```

Listing B.4: Activation of JVMDI for the JBoss JVM

B.3.7.2. jboss-IDE Plug-in

In order to connect the plug-in to the JVMDI of the JBoss JVM use "Run ≫ Debug..." to set up a "JBoss Remote" configuration:
Connection Type: "Standard (Socket Attach)"
Host: <IP of the JBoss server> (e.g. localhost)
Port: <port as defined in run.conf with "address=..."> (e.g. 8000)

B.4. XDoclet

Here, we want just to explicitly reference to the documentation of some JBoss specific javadoc-tags that are available for XDoclet: http://xdoclet.sourceforge.net/tags/jboss-tags.html.

B.5. Ant Framework

The development of the RTI framework was supported by the *NSDev JBoss Framework* [Nsd]. This is an Ant-based framework for the development of JBoss components. It was modified just slightly, and we want to thank the author.

In contrast to the Lomboz plug-in for Eclipse, it supports not only EJBs and JSPs, but also MBeans, according to the auto-generation of interface classes, deployment descriptors and jar/sar/war files as well as the enfolding ear file.

The Eclipse project for the RTI framework in the $CD/Java-sources/Interceptor/ directory is based on NSDev. Inside the project, the build/ directory keeps the Ant build-files as well as the basic properties files.

Running the codegen Ant target will generate the deployment descriptors in the tmp/classes/ directory, and it will generated appropriate Java source files (for EJBs and MBeans) in the tmp/codegen/ directory. Running the compile Ant target (depends on codegen) will compile the class files to the tmp/classes/ directory. Running the staging Ant target (depends on compile) will create jar, war and sar archives as well as the comprising ear archive. Finally, running the deploy Ant target (depends on staging) will copy the ear archive to the JBoss deploy directory.

Using the current Ant build files requires for the auto-generation for EJBs and MBeans some special package/directory structure: EJBs must have an /ejb/ directory in their package-name/path, as well as MBeans must have a /jmx/ directory. Because **/ejb/**/*Bean.java and **/jmx/**/*.java will grep all EJB and MBean classes, whereas these filesets are defined in build/main/build.xml.

B.6. Useful Linux Scripts

B.6.1. Grep the Contents of jar Archives

For complex Java implementations like J2EE servers that provide many jar archives in their /lib/ directories, it is a common problem to find the one special jar archive that has to be added to the classpath of an own implementation. Of course it is possible just to add all provided jar archives, but if they have to be added to an Eclipse project's classpath this is unpleasing and not necessary.

In the $CD/undocumented/java.tools/ directory there are several scripts that make it easy to find out which jar archive contains a necessary Java class file.

In the above directory, the makejavacp.sh script uses the commands below to add all jar archives in a given directory $1 to the classpath. If the script is used in a shell or in other scripts this has to be as "**source** makejavacp.sh <dir>".

```
LOCAL_CLASSPATH=`echo $1/*.jar | tr ' ' ':'`
export CLASSPATH=$LOCAL_CLASSPATH:$CLASSPATH
```

To find the jar archive that contains a special Java class file, the jargrep.pl can be used. Note that it uses fastjar, a C implementation of the jar command, so it has to be available inside the $PATH variable. The fastjar utility is provided by fastjar-0.93.tar.

Finally the jargrepjboss321.sh script uses makejavacp.sh and jargrep.pl to grep for Java classes inside the JBoss jar archives. The given expression $1 can just be the part of a class name.

```
export CLASSPATH=
source makejavacp.sh $JBOSS/lib
source makejavacp.sh $JBOSS/client
source makejavacp.sh $JBOSS/server/all/lib
jargrep.pl $1 -classpath $CLASSPATH
```

The result of a jargrepjboss321.sh invocation prints the found Java class file (and its package directory) in addition with the jar archive in which the Java class has been found. So it is possible to add only the appropriate jar archive to the classpath of an own application.

B.6.2. Forcing the Termination of JBoss

Sometimes executing $JBOSS/bin/shutdown –S does not completely terminate the JBoss instance. Using Linux Sun JVM, JBoss is executed as several Java processes and a plain killall is only possible for processes named java, which would also kill for example a running instance of Eclipse.

Therefore, a more complex method is required under Linux to terminate all Java JBoss processes. A solution is given below, and it is available as $CD/helper-files/

jboss_kill.sh file. It is also adoptable to kill any other kind of Java processes that belong together by adjusting the parameter of the grep command.

```
ps −ax | grep jboss | sed −e 's/^[          ]*//' | cut −d" " −f1 |
    xargs kill −9
```

B.7. Authentication by JAAS

JBoss uses the *Java Authentication and Authorization Service* (JAAS) to authenticate access to EJBs, if constraints are defined in the deployment descriptor of a considered EJB. We want to provide a short guide to the activation and configuration of JAAS for JBoss.

B.7.1. Setting up JBoss

The file that specifies application-specific security-modules is $JBOSS/server/default/conf/auth.conf. This means we can use LDAP-, JDBC- or file-based authentication individually for each JAAS-Domain. For example:

```
ResponseTime {
        org.jboss.security.auth.spi.UsersRolesLoginModule required
        ;
};
```

The special application-entry “other {...}; ” matches as the default authentication-method. But beware that the auth.conf-file only declares which sources for authentication-information should be accessed, not whether authentication is applied. This information is expressed by the constraint-entries inside the EJB deployment descriptors.

A simple file-based implementation is already give with org.jboss.security.auth.spi.UsersRolesLoginModule which was used in the above auth.conf-example. The user/password-relation is taken from $JBOSS/server/default/conf/users.properties. For example: christoph=extremeCrypticPassword

Each user can be related to several, comma-separated roles by $JBOSS/server/default/conf/roles.properties. For example: christoph=admin,user,guest

B.7.2. Securing the JMX-Console

To secure the JMX-Console the file WEB-INF/web.xml inside the $JBOSS/server/default/deploy/jmx-console.war-archive has to be edited: There exists an commented section: <security−constraint> ... </security−constraint> which has to be uncommented. Some lines below we find the role that will be allowed to access the JMX-Console (the default is JBossAdmin). Using this default, we have to add in the $JBOSS/server/default/conf/roles.properties something like userID=JBossAdmin. Trying to access the JBoss generated http://localhost:8080/jmx-console/ will now give the HTTP-Authentication-Dialog by the browser.

Bibliography

[Aag01] Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems.* PhD thesis, University of Oslo, 2001.

[AMW$^+$03] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM symposium on Operating systems principles*, pages 74–89. ACM Press, 2003. URL http://www.cs.rochester.edu/sosp2003/papers/p146-mogul.pdf, downloaded at 2004-02-07.

[BCE] BCEL - Byte Code Engineering Library. Apache Software Foundation - The Jakarta Project. URL http://jakarta.apache.org/bcel/, last visited at 2004-01-21.

[Bea] Bean-test. Empirix. URL http://www.scl.com/products/empirix/datasheets/beantest.html, last visited at 2003-11-25.

[BHL00] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego, CA, October 2000. USENIX. URL http://www.cs.utah.edu/flux/papers/kaffeos-osdi00-base.html, downloaded at 2004-02-07.

[BHV01] Walter Binder, Jane G. Hulaas, and Alex Villazon. Portable resource control in Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 139–155. ACM Press, 2001.

[Bin01] Walter Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, pages 35–42, San Diego, CA, USA, January 2001.

[BIT] BIT – bytecode instrumenting tool. University of Colorado. URL http://www.cs.colorado.edu/~hanlee/BIT/, downloaded at 2004-02-21.

[Bru03] Jean-Michel Bruel, editor. *Proc. 1st Intl. Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France.* Cépaduès-Éditions, June 2003.

[CE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: a resource accounting interface for Java. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, 1998.

[CHS$^+$03] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciaran Bryce. A resource management interface for the Java platform. Technical Report TR-2003-124, Sun Microsystems, May 2003. URL http://research.sun.com/techrep/2003/abstract-124.html, last visited at 2003-12-18.

[CLZ98] Brian F. Cooper, Han B. Lee, and Benjamin G. Zorn. Profbuilder – a package for rapidly building Java execution profilers. Technical report, University of Colorado, April 1998. http://www.cs.colorado.edu/~hanlee/techreport.ps, downloaded at 2004-02-15.

[DRO] DROPS - the Dresden real-time operating system project. Dresden University of Technology. URL http://os.inf.tu-dresden.de/drops/, last visited at 2004-02-19.

[Ecl] Eclipse: Extensible Java IDE. URL http://download.eclipse.org/downloads/index.php, downloaded at 2003-06-10.

[EJP] Extensible Java Profiler. SourceForge. URL http://ejp.sourceforge.net, downloaded at 2004-02-21.

[EPP] Eclipse Profiler Plugin. SourceForge. URL http://eclipsecolorer.sourceforge.net/index_profiler.html, downloaded at 2004-02-21.

[ESS] ESS-Model: UML reversing tool for Java. SourceForge. URL http://sourceforge.net/projects/essmodel/, downloaded at 2004-01-21.

[FOR] FORM – middleware for reverse engineering of dynamic systems. Department of Computer Science, Drexel University, PA, USA. URL http://www.cs.drexel.edu/~tsouder/form/index.html, downloaded at 2004-02-21.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Gri] The Grinder. SourceForge. URL http://grinder.sourceforge.net, last visited at 2003-11-28.

[HK03] Jarle G. Hulaas and Dimitri Kalas. Monitoring of resource consumption in Java-based application servers, 2003. URL http://www.jraf2.org/pubs/HPOVUA2003-Hulaas.pdf, downloaded at 2003-12-18.

[HPr] HProf. Sun. URL http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html, last visited at 2004-02-21.

[HQGM02] M. Harkema, D. Quartel, B. M. M. Gijsen, and R. D. van der Mei. Performance monitoring of Java applications. In *Proceedings of the Third International Workshop on Software and Performance*, pages 114–127. ACM Press, 2002.

[Jac] JacORB. Software Engineering and Systems Software Group at Freie Universität Berlin and Xtradyne Technologies AG. URL http://www.jacorb.org, last visited at 2004-01-21.

[JBo] JBoss Group. *JBoss application server website.* URL http://www.jboss.org, last visited at 2004-01-21.

[JBo01] JBoss Group. *JBoss: Overview*, 2001. URL http://www.jboss.org/overview/, last visited at 2003-11-14.

[Jin] Jinsight. IBM alphaWorks. URL http://www.alphaworks.ibm.com/tech/jinsight/, downloaded at 2004-02-21.

[JPb] JProbe. Quest Software. URL http://www.quest.com/jprobe/, last visited at 2004-02-21.

[JPf] JProfiler. ej-technologies. URL http://www.ej-technologies.com/products/jprofiler/overview.html, last visited at 2004-02-21.

[JVM] Java 2 platform, standard edition (J2SE). Sun Microsystems. URL http://java.sun.com/j2se/1.4.2/download.html, last visited at 2003-11-18.

[Liu02] Jenny Liu. Research project: An analysis of jboss architecture. University of Sydney, 2002. URL http://www.cs.usyd.edu.au/~jennyliu/jboss.html, downloaded at 2003-12-18.

[LV99] Sheng Liang and Deepa Viswanathan. Comprehensive profiling support in the Java virtual machine. In *5th USENIX Conference on Object-Oriented Technologies and Systems*, pages 229–240, Palo Alto, CA, USA, 1999. Sun Microsystems. URL http://citeseer.nj.nec.com/liang99comprehensive.html, last visited at 2003-12-18.

[MH03] Cary Millsap and Jeff Holt. *Optimizing Oracle Performance.* O'Reilly, September 2003.

[MM01a] Adrian Mos and John Murphy. New methods for performance monitoring of J2EE application servers. In *9th International Conference on Software, Telecommunications and Computer Networks*, ICT, pages 423–427, Bucharest, Romania, 2001. IEEE. URL http://citeseer.nj.nec.com/465203.html, last visited at 2003-11-20.

[MM01b] Adrian Mos and John Murphy. Performance monitoring of Java component-oriented distributed applications. In *9th International Conference on Software, Telecommunications and Computer Networks*, SoftCOM, Croatia-Italy, 2001. IEEE. URL http://citeseer.nj.nec.com/mos01performance.html, last visited at 2003-11-14.

[MM02] Adrian Mos and John Murphy. A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *Proceedings of the third international workshop on software and performance*, pages 235–236, 2002.

[MM04] Marcus Meyerhöfer and Klaus Meyer-Wegener. Estimating non-functional properties of component-based software based on resource consumption. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2004.

[MN04] Marcus Meyerhöfer and Christoph Neumann. TestEJB - a measurement framework for EJBs. In Ivica Crnkovic, editor, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 294–301. Springer, May 2004.

[New99] Tia Newhall. *Performance Measurement Of Interpreted, Just-In-Time Compiled, Dynamically Compiled Executions.* PhD thesis, University of Wisconsin, 1999. URL http://www.cs.swarthmore.edu/~newhall/perf.html, downloaded at 2003-12-18.

[NM99] Tia Newhall and Barton P. Miller. Performance measurement of dynamically compiled Java executions. In *Java Grande*, pages 42–50, 1999. URL http://www.cs.swarthmore.edu/~newhall/perf.html, downloaded at 2003-12-18.

[Nsd] No Such Device presents its JBoss Ant Framework. URL http://www.nsdev.org/jboss/stories/jboss-framework.html, downloaded at 2004-01-21.

[Obj01] Object Management Group. *Portable Interceptors spec*, March 2001. URL http://www.omg.org/cgi-bin/apps/doc?ptc/01-03-04.pdf, last visited at 2003-11-20.

[Obj02a] Object Management Group. *Common Object Request Broker Architecture: Core Specification*, December 2002. URL http://www.omg.org/technology/documents/formal/corba_iiop.htm, last visited at 2003-11-14.

[Obj02b] Object Management Group. *CORBA Basics*, 2002. URL http://www.omg.org/gettingstarted/corbafaq.htm, last visited at 2003-11-14.

[OIt] OptimizeIt Servertrace. Borland. URL http://www.borland.com/opt_servertrace/pdf/ost2_techview.pdf, last visited at 2003-11-21.

[PAn] PerfAnal. Sun. URL http://java.sun.com/developer/technicalArticles/Programming/perfanal/index.html, downloaded at 2004-02-21.

[PAP] PAPI: Performance Application Programming Interface. URL http://icl.cs.utk.edu/papi/, last visited at 2004-01-21.

[PCL] PCL: The Performance Counter Library. University of Applied Sciences Bonn-Rhein-Sieg and the Research Centre Juelich, Germany. URL http://www.fz-juelich.de/zam/PCL/, last visited at 2004-01-21.

[PCt] PerfCtr – Linux x86 performance-monitoring counters driver. Uppsala University, Sweden. URL http://user.it.uu.se/~mikpe/linux/perfctr/, last visited at 2004-02-21.

[PIn] Performance Inspector. IBM developerWorks. URL http://www-124.ibm.com/developerworks/oss/pi/, downloaded at 2004-02-21.

[PS03] Christoph Pohl and Alexander Schill. Client-side component caching: A flexible mechanism for optimized component attribute caching. In *Proc. DAIS'03*, 2003.

[PSu] PerformaSure. Quest Software. URL http://www.quest.com/performasure/index.asp, last visited at 2003-11-21.

[PTR] PTR: Portable Timing Routines. URL http://web.engr.oregonstate.edu/~pancake/ptools/ptr/, last visited at 2004-01-21.

[Rab] Rabbit - a performance counters library for Intel/AMD processors and Linux. Scalable Computing Laboratory Ames Laboratory, Iowa State University. URL http://www.scl.ameslab.gov/Projects/Rabbit/, last visited at 2004-01-21.

[RAJ02] Ed Roman, Scott Ambler, and Tyler Jewell. *Mastering Enterprise JavaBeans*. John Wiley & Sons, New York, USA, 2nd edition, 2002. URL http://www.theserverside.com/books/masteringEJB/, downloaded at 2003-06-12 (search the net for "MasteringEJB2.pdf").

[RR00] Steven P. Reiss and Manos Renieris. Generating Java trace data. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 71–77. ACM Press, 2000. URL http://citeseer.nj.nec.com/reiss00generating.html, last visited at 2003-12-18.

[RR01] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230, Toronto, Ontario, Canada, 2001. IEEE. URL http://citeseer.nj.nec.com/reiss01encoding.html, last visited at 2003-12-18.

[RZ03a] Simone Röttger and Steffen Zschaler. CQML$^+$: Enhancements to CQML. In Bruel [Bru03], pages 43–56.

[RZ03b] Simone Röttger and Steffen Zschaler. Model-driven development for non-functional properties: Refinement through model transformation. Dresden University of Technology, August 2003.

[SJ03] Scott Stark and The JBoss Group. *JBoss Administration and Development.* JBoss Group, Atlanta, GA, USA, second edition, April 2003.

[SM00] Timothy S. Souder and Spiros Mancoridis. FORM beta – A dynamic system analysis tool, 2000. URL http://www.cs.drexel.edu/~tsouder/form/download/form.pdf, downloaded at 2003-12-18.

[SMS01] Timothy S. Souder, Spiros Mancoridis, and Maher Salah. FORM: A framework for creating views of program executions. In *ICSM*, pages 612–621, 2001. URL http://citeseer.nj.nec.com/souder01form.html, last visited at 2003-12-18.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York, USA, 2000. Reprinted with corrections April 2001.

[Sun96] Sun Microsystems. *Java Platform Debugger Architecture*, 1996. URL http://java.sun.com/j2se/1.4.2/docs/guide/jpda/architecture.html, last visited at 2003-11-19.

[Sun98] Sun Microsystems. *Java Virtual Machine Profiler Interface*, 1998. URL http://java.sun.com/j2se/1.4.1/docs/guide/jvmpi/jvmpi.html, downloaded at 2003-11-19.

[Sun01a] Sun Microsystems. *EJB 2.0 Specifications*, August 2001. URL http://java.sun.com/products/ejb/docs.html, downloaded at 2003-11-14.

[Sun01b] Sun Microsystems. *J2EE 1.3 Platform Specifications*, July 2001. URL http://java.sun.com/j2ee/download.html, last visited at 2003-11-14.

[Sun02] Sun Microsystems. *JMX 1.2 Specifications*, September 2002. URL http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html, last visited at 2003-11-14.

[Sun03a] Sun Microsystems. *Java 2 Platform, Standard Edition (J2SE), 1.4.1 API Specification: System.currentTimeMillis()*, 2003. URL http://java.sun.com/j2se/1.4.1/docs/api/java/lang/System.html, last visited at 2003-11-28.

[Sun03b] Sun Microsystems. *Java Data Objects 1.0.1 Specification*, May 2003. URL http://jcp.org/aboutJava/communityprocess/final/jsr012/index2.html, last visited at 2003-11-25.

[Sun03c] Sun Microsystems. *Java Platform Profiling Architecture*, 2003. URL http://www.jcp.org/en/jsr/detail?id=163, last visited at 2003-12-10.

[Sun03d] Sun Microsystems. *JMX Remote 1.0 Specifications*, October 2003. URL http://jcp.org/en/jsr/detail?id=160, last visited at 2003-11-14.

[Sun03e] Sun Microsystems. *JNI 1.1 Specifications*, 2003. URL http://java.sun.com/j2se/1.4.2/docs/guide/jni/, last visited at 2004-02-19.

[Sun03f] Sun Microsystems. *Official Specifications for CORBA support in J2SE 1.4*, 2003. URL http://java.sun.com/j2se/1.4.2/docs/api/org/omg/CORBA/doc-files/compliance.html, last visited at 2003-11-20.

[VB01] Alex Villazn and Walter Binder. Portable resource reification in Java-based mobile agent systems, 2001. URL http://citeseer.nj.nec.com/589257.html, last visited at 2003-12-18.

[Vid01] Rory Gavino Vidal Burgoa. Implementation of CPU resource accounting for Java, July 2001. URL http://anaisoft.unige.ch/public-documents/deliverables/Diploma-Vidal.pdf, downloaded at 2003-12-18.

[VMw] VMware - virtual machines on high-performance Intel servers. VMware, EMC Corporation. URL http://www.vmware.com, last visited at 2004-01-21.

[WH88] D. Wybranietz and D. Haban. Monitoring and performance measuring distributed systems during operation. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 197–206. ACM Press, 1988.

[XN02] Tao Xie and David Notkin. An empirical study of Java dynamic call graph extractors, December 2002. URL http://citeseer.nj.nec.com/466909.html, downloaded at 2003-07-10.

[ZM03] Steffen Zschaler and Marcus Meyerhöfer. Explicit modelling of QoS-dependencies. In Bruel [Bru03], pages 57–66.