TESTEJB - A Measurement Framework for EJBs

Marcus Meyerhöfer and Christoph P. Neumann

Friedrich-Alexander University of Erlangen and Nuremberg
{Marcus.Meyerhoefer,sichneum}@immd6.informatik.uni-erlangen.de

Abstract. Specification of Quality of Service (QoS) for components can only be done in relation to the QoS the components themselves are given by imported components. Developers as well as users need support in order to derive valid data for specification respectively for checking whether a selected component complies with its specification. In this paper we introduce the architecture of a measurement framework for EJBs giving such support and discuss in detail the measurement of the well understood property of response time.

1 Introduction

The concept of component-based software development has gained much attention in the last years as mature component models, especially on the server side have evolved.

A key factor for success in a component market is the specification, but beside the functional interfaces additional levels should be addressed [1], defining on each level what the “user” of the component can expect, given that the requirements of the component are met. In the COMQUAD project[1] we are focusing on the specification of non-functional properties (NFPs) [2] of software components, using an extension of the language CQML, called CQML$^+$ [3] that enables the specification of NFPs, their relationships and additional resource properties.

In order to offer support for the determination of specific properties for specification of components as well as for runtime monitoring, we aimed at a solution that enables to measure selected NFPs and at the same by its non-intrusive approach provides means to monitor components. In this paper we introduce the architecture and concepts of a measurement environment for components following the Enterprise Java Beans (EJB) specification. Although the all-embracing concept should be capable of measuring many different NFPs, we focus in this paper on the measurement of response time (RT) as the importance of RT for any kind of application is commonly understood.

This work is also a preparation for deriving call dependencies between interacting EJBs, 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. But this will be topic of future work.

[1] “COMponents with QUantitative properties and ADaptivity” is a project funded by the German Research Council (DFG FOR 428). It started October 1, 2001, at Dresden University of Technology and Friedrich-Alexander University of Erlangen and Nuremberg.

2 Related work

The *COMPAS* framework [4] aims at a solution aiding developers of distributed, component-oriented software in understanding performance related issues of their applications. Therefore, 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.

EJB Monitor [5,6], the monitoring module of COMPAS, is developed as an own project. The monitoring approach applied is based on the proxy pattern; for each EJB in an application a proxy bean is generated, stealing the JNDI name and therefore appearing as the bean to be monitored; thereby it is possible to receive, monitor and forward the original invocations. Although transparent to the client and the other components, this "heavyweight" approach introduces an additional indirection between each caller and callee: an additional marshalling/unmarshalling between the proxy-bean and the original bean takes place for remote interfaces. Furthermore the amount of EJBs is doubled if each component should be monitored.

Commercial environments like *PerformaSure* from Quest Software or *OptimizeIt ServerTrace* from Borland offer software to monitor J2EE applications and pinpoint bottlenecks. They cover a broader spectrum of J2EE integrating tools to monitor database connections, network traffic or virtual machine parameters. For EJB-components e.g. OptimizeIt ServerTrace supports method-level drilldown of performance; however, these tools are primarily targeted to support pre-deployment teams and to reveal performance bottlenecks if occurred in productive use and therefore limit themselves to performance properties.

3 Background and Environment

The component model of our environment are the J2EE 1.3 platform specifications and the EJB 2.0 specifications. The common EJB component implements an interface that allows the remote method invocation and the control of its life-cycle. The EJB component is not viable without a special runtime environment: any EJB component has to be deployed to an EJB container. The container runs on a JVM and provides middleware services to the EJBs. These services are for example transactional support, persistence, security mechanisms and the registration to a central component registry. Our run-time environment consists of the application server JBoss running on a Sun JVM 1.4.2. JBoss is open-source software and the release version 3.2.1 fully supports the EJB component model. JBoss is based on the Java Management eXtensions (JMX). JMX allows the implementation of a modular J2EE architecture with JMX as microkernel and the modules represented as JMX' MBeans. The most important part of JBoss in our context and related to JMX is the EJB container module.

Interceptor is a design pattern of a callback mechanism for middleware remote invocations. The component container defines the *interceptor callback interface* with its hook methods and the implemented *concrete interceptors* act as callback handlers during a method invocation. This allows the non-intrusive integration of services to the component container. OMG has already adopted and standardized interceptors as *Portable Interceptors* and the J2SE 1.4 is compatible to them.

JBoss' interceptors can be configured on a per-EJB basis: Every EJB deployed to JBoss has – according to its type – several standard interceptors such as security or logging. An installed interceptor is involved with all the J2EE invocations to the bean and thereby allows the insertion of any additional functionality into the invocation path of a component invocation – either at the client-, server-, or both sides of the invocation. But note that client- and server-side interceptors inherit from separate interfaces. The multiple interceptors that are installed for an EJB are executed consecutively during each request. The installation is done by adding a JBoss-specific XML line to the vendor-specific deployment descriptor (for the JBoss container this is META-INF/jboss.xml).

4 Proposed Solutions

One of the most basic NFPs of a component is the RT of an invocation at one of its exposed functions. We assume that there exist EJBs from many different vendors that are not available as source-code. So first of all the EJB acts as "black-box" and is only available as byte-code. At first we identify several layers in any J2EE implementation at which measurements can be taken. Afterwards, we introduce our chosen approach explaining the concept and its various applications and discuss the overhead of our method. The comprising architecture of our solution is presented in Fig. 1, but a thorough understanding will only be achieved after Sect. 4.3 has been read.

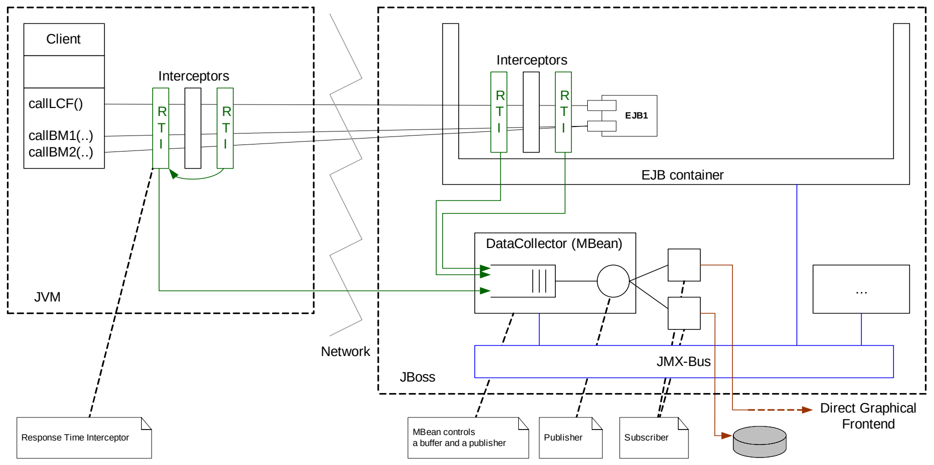

Fig. 1. Architecture view at the measurement framework.

4.1 Layers for Instrumentation

There are several levels in the J2EE environment at which measurements are possible: The most simple approach would be to instrument the clients of the EJBs to measure RT. We focus on a scenario where existing clients are calling productive beans and with client applications not available as source code. Therefore instrumentation of clients can't be considered and the mechanism has to be transparent to the client's source-code – even though an adaption of the client's environment is inevitable.

The bottom level of possible instrumentation is the *Operating System* (OS). The instrumentation of the kernel(-modules) allows the entire monitoring of network traffic so the J2EE traffic could be filtered. A similar idea where RMI calls are filtered by modification of the EJB server code is given in [6], but this approach is neither portable across operating systems (OS) – respectively EJB containers – nor easy to implement and to maintain for coming versions of any OS.

Especially in J2EE the layer above the OS is the *Java Virtual Machine* where the measurement of time-information seems possible by using standard interfaces like the Java Virtual Machine Debugger Interface or the Java Virtual Machine Profiler Interface. But again a filtration would be necessary. Another alternative at this layer is the instrumentation of the `java.net.*` classes [7] having the same implications as the OS instrumentation.

The next level is the *J2EE server* respectively the *EJB container*. One alternative is to enhance the source-code of the EJB container. The best location to introduce the sensors for RT seems to be the framework that automatically generates the EJB Home and the EJB Object. But this would result in the duty to maintain the code for coming versions of the container and lacks the portability to other container implementations.

The highest level before the already rejected application level are the *EJBs* themselves. At this level the client-side RT can obviously not be measured but a server-side RT could be provided as described for the "EJB Monitor" in Sect. 2.

The non-intrusive instrumentation of the EJB container by using a callback mechanism like interceptors seems most promising because both client- and server-side are considered.

4.2 Response Time Interceptor

In a client-server setting there are several definitions of RT possible [8]; however, the *client-side response time* $\mathrm{T}_{\mathrm{resp,client}}$ including communication overhead and the *server-side response time* $\mathrm{T}_{\mathrm{resp,server}}$ are the most important ones. The injection of interceptors on both sides of the invocation allows for the transparent measurement of $\mathrm{T}_{\mathrm{resp,client}}$ and $\mathrm{T}_{\mathrm{resp,server}}$ by timestamping the delegation.

The placement of the corresponding response time interceptors (RTI) is crucial, because the sequence of execution of the interceptors is directly related to the textual order in the manifest/jboss.xml-file. The top one is invoked first and the bottom one invoked at last before the call is delegated to the EJB itself. Therefore the RTIs have to be inserted as the last one ($\mathrm{T}_{\mathrm{resp,server}}$) respectively the first one ($\mathrm{T}_{\mathrm{resp,client}}$) in the interceptor chain. The possibility to place the RTIs arbitrarily enables additional semantics of the measured times.

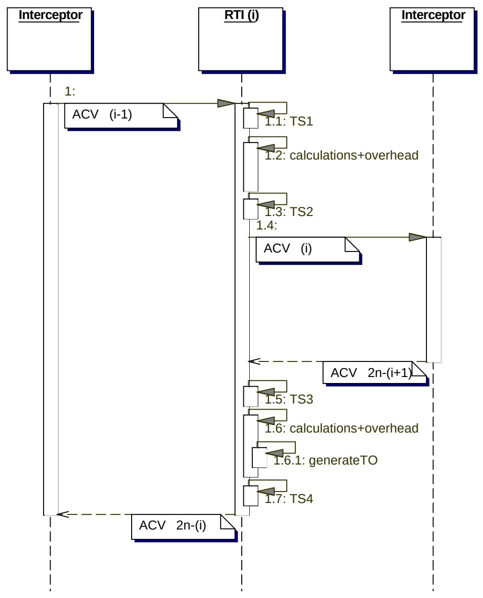

Fig. 2. The general behavior of RTIs.

As mentioned in Sect. 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: a response time interceptor RTI$_i$ has to take four timestamps (TS) as shown in Fig. 2, whereas $i \in [1; n]$ equals the logical position of the RTI instance in the RTI chain – only the n RTIs are iterated, not the other interceptors. Two are needed for calculating the RT and the remaining ones in order to account for 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: collection of additional context information, calculation of RTs, construction and transmission of a transfer object (TO) to an asynchronous storage mechanism (see Sect. 4.3).

Because the callback to the RTIs produces overhead, there are two views at both RTs. The *uncleansed RT* ignores this fact and is calculated as $(\text{TS}_3 - \text{TS}_2)$. The *cleansed RT* considers the overheads – each interceptor calculates and transmits the accumulated correction value (ACV) , shown exemplarily for RTI$_i$:
On the inward way $\text{ACV}_{(i)} := \text{ACV}_{(i-1)} + (\text{TS}_2 - \text{TS}_1)$, and on the outward way $\text{ACV}_{2n-(i)} := \text{ACV}_{2n-(i+1)} + (\text{TS}_4 - \text{TS}_3)$.

In order to account for the overhead of the $\text{RTI}_{(x)} : x > i$ the received correction value $\text{ACV}_{2n-(i+1)}$ has to be used, but as it contains the whole overhead and not just the one starting at TS$_2$ of RTI$_{(i)}$, that part represented as ACV$_{(i)}$ has to be subtracted. Therefore the cleansed RT is calculated as $(\text{TS}_3 - \text{TS}_2) - (\text{ACV}_{2n-(i+1)} - \text{ACV}_{(i)})$.

The TSs are taken by a JNI module that uses OS system calls like `gettimeofday` due to the lack of precision of `System.currentTimeMillis()`, which is the fall-back of the wrapping Java class if the module was not deployed to the client. The unit in contrast to the accuracy is in either case μs. Global synchronization of clocks is not needed because the RT is a difference of local times. As soon as there is the interest to derive causal relations by temporal order the synchronization of clocks could be useful.

The ordinary use-case consists of one RTI on both client- and server- side measuring client- and server-side RT, whereby client-side RTIs are only required for call dependency derivation, because they allow to identify invocations by real client applications.

The interceptor chain consists always of interceptors correlated to the provided middleware services like transactional support or security. The overhead of their executions can be measured by wrapping the interceptor of interest between two RTIs and storing not only the RT but also the TSs. The TSs of an individual RTI is now represented as $\mathrm{TS}_x^{(i)}$ where $x \in \{1, 2, 3, 4\}$ and i as above. 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: $(\mathrm{TS}_1^{(i+1)} - \mathrm{TS}_2^{(i)}) + (\mathrm{TS}_3^{(i)} - \mathrm{TS}_4^{(i+1)})$.

The interceptor-based measuring approach therefore allows not only to identify hot-spot EJBs and performance bottlenecks, but even to track down the constituents of the RT experienced by a client by wrapping the interceptors implementing middleware services. By using the interceptor mechanism on a per-EJB basis (see Sect. 3), a developer is enabled to monitor just some selected components, e.g. long-running ones which then might be split up in subtasks. Such configuration can even be changed at runtime using the hot-deployment feature of JBoss, facilitating the transparent “attachment” of the measurement mechanism to an already deployed component.

4.3 Storing the Measurement Data

The information gathered by the RTIs during TS_1 to TS_2 and TS_3 to TS_4 is wrapped as TO. A JMX' DataCollector-MBean (DCMB) is used as buffered and therefore asynchronous storage mechanism for the TOs. The buffer reduces overhead during call-time by taking the cost-intensive persistency mechanism out of the return path, because the TOs are only transferred to the MBean but not flushed until the buffer's (manageable) capacity exceeds. Apparently the client-invocation is also made fail-safe against the underlying database (DBS), because an error can only occur during the flushing to the DBS, ergo when the client-call is all over and done. The DCMB 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 is planned. The comprehensive architecture of our solution has been shown in Fig. 1.

The DCMB 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 call 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 is integrated into JBoss an adoption of our implementation is planned. As an optimization of client-side's overhead only the last client-side RTI on the return path collects all client-side TOs and transfers them by only one remote call to the DCMB.

Because the measurement of the RT is transparent to the client, a client that is interested in its RT has to query the DBS using an unambiguousness tuple that represents one of its invocations (e.g. invocation-near TS, IP, parameters). But most assumed clients are not interested in their RT values after all because analysis is done by the administrators or developers and takes place only by the stored data in the DBS.

4.4 Overhead

A software approach to measurement has always some overhead that is never accountable exactly. Besides the efforts to minimize the overhead and to maximize the accuracy the following influences have to be considered [2]: On server-side the JNI module should always be deployed, otherwise the accuracy is reduced to `System.currentTimeMillis()`. Although the unit of `currentTimeMillis()` is ms, its accuracy can only be assumed "in units of tens of milliseconds" [9].

The accountable overhead introduced by the RTIs that results in a greater perceived response time by the client can be gained for each RTI and call individually by the stored data as $(\mathrm{TS}_2 - \mathrm{TS}_1) + (\mathrm{TS}_4 - \mathrm{TS}_3)$. Measurements have shown that the insertion of one RTI on the server-side – this one will have to create both TOs for MetaData and RT – introduces overhead around 1.0 ms (the RTI on client-side introduces more overhead as discussed below). More RTIs increase this number only by 0.1 ms each, because most overhead is related to the first and last one.

Because the TS_4 of the first RTI (RTI_1 being the last returning one) can't be stored its overhead is not accountable. Mostly the RTI_1 will be at client-side transferring the client-side TOs by a remote call to the DCMB so its overhead has to be respected. Measurements have shown that its duration is in units of 10 ms. Therefore, we use a separate thread for this storage, allowing the RTI to return after 400 μs (about 289.325 μs for thread creation, as the average of 10000 tests). Anyway, for analysis this is a minor issue because the TS_3 of the RTI_1 represents nearly the same time the client would receive the result if the RTIs would not be present.

The JNI mechanism introduces non-accountable overhead. Tests showed that the JNI call to the Linux' `gettimeofday(2)` lasts about 0.970 μs as the average of 10000 tests. The pure `gettimeofday(2)` call by a C program takes about 0.254 μs for the same number of tests.

Calculation of the ACVs and putting them into the forward- or backward-passed environment maps in order to transmit them to the next participant in the logical RTI chain must inevitably take place after the TS_2 and TS_4 are done. Temporarily timestamping the relevant code-segments and run-time printing of the differences showed that each RTI adds about 5 μs of never accountable overhead. If necessary, this knowledge can be used during analysis for a more accurate interpretation of the data. Finally the gap between RTI_1's return and the client's receiving as discussed in Sect. 4.1 was measured to be below 1 ms.

[2] All measurements were taken on a standard PC system with an AMD K7 1400 MHz processor and 1 GB SDRAM running Linux SuSE 8.2, Kernel version 2.4.20-4GB-athlon.

5 Conclusion

In this paper we have introduced the concepts and the implementation of a measurement environment for components following the EJB specification. Based on the interceptor pattern, it has been shown how this is applied for measurements in the open-source application server JBoss. Following a discussion of several possible layers of instrumentation, the application of our concept is explained in detail for response time measurement: basically, it is possible to not only measure client- or server-side response times, but also determine delays introduced by standard interceptors in the call chain wrapping them with RTIs. Because a software solution always introduces some overhead, we have shown how to calculate cleansed response times using correction values (ACVs) and have discussed the costs of JNI and the transmissions between interceptors which are not included in the ACVs.

We believe that the usage of interceptors allows for measurement of many different properties of components besides the response time, e.g. jitter or throughput. By now using the information stored by the RTIs not only the response time can be determined, but also call-dependencies between the different EJBs of an application. In subsequent work we plan to analyze those dependencies more deeply and to extend our framework to some selected additional properties.

References

1. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. IEEE Computer **32** (1999) 38–45
2. Meyerhöfer, M., Meyer-Wegener, K.: Estimating non-functional properties of component-based software based on resource consumption. In: SC 2004: Workshop on Software Composition Affiliated with ETAPS 2004, Barcelona, Spain. (2004)
3. Röttger, S., Zschaler, S.: CQML$^+$: Enhancements to CQML. In Bruel, J.M., ed.: Proc. 1st Intl. Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France, Cépaduès-Éditions (2003) 43–56
4. Mos, A., Murphy, J.: A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In: Proceedings of the third international workshop on software and performance. (2002) 235–236
5. Mos, A., Murphy, J.: Performance monitoring of Java component-oriented distributed applications. In: 9th International Conference on Software, Telecommunications and Computer Networks. SoftCOM, Croatia-Italy, IEEE (2001) .
6. Mos, A., Murphy, J.: New methods for performance monitoring of J2EE application servers. In: 9th International Conference on Software, Telecommunications and Computer Networks. ICT, Bucharest, Romania, IEEE (2001) 423–427 .
7. Czajkowski, G., Eicken, T.v.: JRes: a resource accounting interface for Java. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. (1998) 21–35
8. Röttger, S., Zschaler, S.: Model-driven development for non-functional properties: Refinement through model transformation. TU Dresden, submitted for publication (2003)
9. Sun Microsystems: J2SE 1.4.1 API specification: System.currentTimeMillis() (2003) http://java.sun.com/j2se/1.4.1/docs/api/java/lang/System.html.