mKernel: A manageable kernel for EJB-based systems

Jens Bruhn
Distributed and Mobile Systems Group
Feldkirchenstr. 21
96052 Bamberg, Germany
jens.bruhn@wiai.uni-bamberg.de

Guido Wirtz
Distributed and Mobile Systems Group
Feldkirchenstr. 21
96052 Bamberg, Germany
guido.wirtz@wiai.uni-bamberg.de

ABSTRACT

Due to the ever increasing complexity of todays Enterprise Applications (EA), component technology has become the major means to keep the development of such applications under control. Although container technology provides tools to deploy component-based EAs, high demands regarding, e.g., availability, security and fault-tolerance combined with constantly changing user demands, varying loads and rapid change of business processes, introduce the need for adjusting systems in regular intervals without halting, restructuring and re-deploying the system as a whole. Consequently, the administration of EAs is a very complex task which has to be performed during runtime of the managed system. Hence, techniques from the area of Autonomic Computing (AC) that allow for controlling and changing a running system without the need to go back to development can become highly useful. This paper presents the design rationale and overall architecture of a manageable kernel that equips the broadly accepted Enterprise Java Beans 3.0 (EJB) component standard for enterprise applications with additional facilities in order to make EJB components AC manageable at runtime. The system was realized EJB 3.0-compliant and provides container infrastructure enhancements as well as a tool needed to adapt standard EJB components.

Keywords

Enterprise Applications, EJB, Management, mKernel

1. INTRODUCTION

*Enterprise Application*s (EA) represent a family of very complex software systems used for supporting the business of companies. Their complexity emerges from the different application areas within which they are used in combination with a high degree of interrelation among those areas, e.g. ordering and warehousing. Additionally, the environment of an EA constantly changes due to strategic and operative aspects. An EA might on the one hand be extended to integrate solutions for demands arising from new business areas of the operating company. On the other hand, certain parts of an application might be deactivated in case they should not be provided anymore. If workload increases, decreases, or shifts, administrators of an EA might face the need to reorganize the system in response. Certain parts of an EA might be provided to external users, e.g. to suppliers or customers. Especially those parts must be isolated and protected to a very high degree regarding security aspects. Types of threats will probably change over time leading to constant administrative demand. The manifold areas of possible adjustments, in combination with the inherent complexity of systems render the task of administration very complex. Additionally, the very high demands on availability of EAs leads to the need to perform adjustments of a system during runtime. A complete system shutdown in combination with an offline re-configuration would be very costly and probably lead to a loss of reputation, even when the downtime is very short. Consequently, this option seems to be unacceptable, especially if it has to be carried out in more or less regular intervals.

The concept of *Component Orientation* (CO) [22] represents one approach for establishing a software system in a modular way. The modules – called *Component*s – a system is built from, encapsulate functionality and expose it through well defined *Interface*s to their environment. In return they can make use of other components through their provided interfaces. An interface required by a component is called *Receptacle*. Consequently, a component-based system can be seen as a collection of loosely-coupled modules which collaborate among each other through their interfaces. Therefore, different functional aspects of a system can be treated in isolation and a modular design of software systems is promoted. Normally, a component is developed with respect to a certain *Component Standard*. A standard defines different obligations regarding the implementation of components. In return, a component implementation can rely on concepts provided by the standard. Besides other aspects, e.g. the target programming language or application area, component standards differ with respect to the number and type of included features. While some standards might only standardize formats for message exchange, others might also include specifications for services and facilities, or deployment formats for components. An implementation of a component standard is called *Component Platform*. A platform must implement at least all recommended parts of the underlying standard and can realize the optional ones. Moreover, it is possible to enhance the underlying standard with additional features. Relying on

AUTONOMICS 2007, 28-30 October 2007, Rome, Italy

DOI 10.4108/ICST.AUTONOMICS2007.2222

optional or non-standard features for the development of components imposes the risk for the implementation of not being usable with all standard-compliant platforms. Typically, a platform includes a runtime environment for components, called *Container*, which provides the services and facilities of the corresponding standard. One broadly accepted standard for component-oriented EA systems based on the *Java* programming language is the *Enterprise Java Beans*-standard (EJB), currently available in version 3.0 [8, 9]. This standard provides a reasonable component-model and a rich set of services and facilities, e.g. for persistence management and transaction control. Additionally, the isolated treatment of different aspects – e.g. application logic and security – allows separation of concerns for development, execution, and administration.
While the concept of CO facilitates the development and initial configuration of complex systems, the vision of *Autonomic Computing* (AC) [13, 16, 18] was developed to find a solution for handling the constantly increasing complexity of todays and future computer systems during runtime. Its main idea relies on the assignment of low-level administrative tasks upon the managed system itself. A promising approach for addressing the complexity of EAs during development and runtime could consequently be to bring together the concepts of CO with the vision of AC. Therefore, a component platform must be enhanced with facilities and services allowing managing entities to gather information about the structure and behavior of the underlying component-based EA. Additionally, the manipulation of the component system must also be supported. The enhancements should be highly generic to ensure that they can be used within lots of contexts with respect to self-management properties [21].

Within this paper we discuss a system of component-based enhancements on top of the EJB 3.0 standard, called *mKernel*. It can be used to establish a sound and extensible foundation for AC in the context of EAs. The system is realized through a set of components which can be deployed into an EJB-compliant container. The main contribution of *mKernel* lies on its platform specific approach of providing a generic manageable layer regarding applications developed for the considered platform. It enables the provision of facilities for monitoring and re-configuration on a unified foundation. Compared to existing platforms for AC like [12, 14], *mKernel* is not intended to be used for managing different types of resources. Because of being platform specific, it is possible to rely on guidelines of a standard which enable the realization of very fine-grained and rich opportunities for analysis and control in a homogeneous fashion. Unified sensors and effectors prevent from the need to address characteristics of different types of managed resources, thus reducing complexity for the management layer. Manageability is integrated into managed resources automatically. Therefore, autonomic management needs not be considered for the core application logic and development is facilitated. Moreover, *mKernel* is intended to control the participating components of the managed layer on a low level for providing a very high degree of re-configuration freedom. Domain specific approaches for the Java Enterprise Edition like [3, 4] address the management of J2EE-based systems on deployment level. In contrast, *mKernel* focuses on the detailed management of components during runtime. Moreover, our realization does not require any adjustment of the underlying container implementation.

The remainder of this paper is structured as follows: Section 2 discusses the different requirements of AC addressed by the presented system. To get a grip on the advantages and shortcomings of the underlying component-standard, section 3 briefly introduces those parts of the EJB-standard that are essential for the presented system. In section 4 the architecture of the constituting components for autonomic management as well as the enhanced facilities needed are discussed. Necessary extensions to EJB components are treated in section 5. Afterwards, *mKernel* is evaluated against the requirements stated in section 2, and the results of first performance evaluations of the implementation are discussed in section 6. The paper concludes with a summary and a discussion of future work in section 7.

2. REQUIREMENTS

With his paper [16], Paul Horn established the foundation for the vision of AC. As part of this paper, eight recommended characteristics are stated for AC-systems, i.e. *Anticipatory*, *Self-Healing*, *Self-Protection*, *Self-Optimization*, *Self-Configuration*, *Self-Awareness*, *Context-Awareness*, and *Openness*. While there is no terminological consensus regarding these AC characteristics in literature yet, this core collection has broadly been adopted [19]. *Anticipatory* addresses the ability of an AC-system to anticipate the goals of its users. The four following characteristics are merely *Objectives* to reach the superior goal of *Self-Management* [18, 21]. They cover reactions of the system to certain situations, like e.g. failures (*Self-Healing*), attacks (*Self-Protection*), or varying resource needs (*Self-Optimization*). In this context *Self-Configuration* can be seen as generic, supporting the others via facilities for adjusting the system accordingly. *Self-Awareness* and *Context-Awareness* both address the information demand of an AC-system to fulfill its duties. While the first one considers the introspection into the internals of the system, the second one addresses the need to gather information about its environment. Finally, *Openness* can be seen as a generally desirable property for AC-systems, i.e. an AC-system should be based upon open standards in contrast to being proprietary.
While the aforementioned characteristics deal with the question of what an AC-system should be capable of, the so-called *Control Loop* represents a concept for realizing these characteristics [10, 18]. It consists of the four stages as shown in figure 1. The first stage (*monitor*) includes all steps of information discovery addressing *Self-Awareness* and *Context-Awareness*. Secondly (*analyze*), the gathered information is analyzed regarding aspects of aggregation and detection of situations making modifications of the system necessary. In case the need for re-configuration is identified, the third stage (*plan*) assembles a collection of operations to transfer the system from the current into the desired state which should better fulfill the *Objectives* of the autonomic system. Finally (*execute*), these operations are performed. During execution of a control loop-cycle, internal *Knowledge* is used which e.g. covers information about symptoms of malicious behavior and options for re-configuration. It has to be pointed out that the control loop is not a one-way process. It is e.g. also conceivable that – during planning – additional information is needed which must be obtained from the managed system or its environment. This may lead to a selective execution of the *monitor*- and maybe the *analyze*-stage. Consequently, an autonomic system concep-

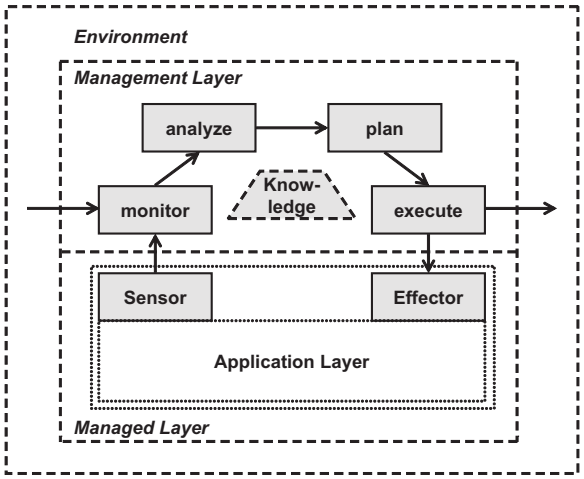

Figure 1: The control loop-concept

tually consists of two main layers. The original services a system provides to its users are allocated inside a *Managed Layer* which is supervised and manipulated by the *Management Layer*. Interaction with the managed layer takes place at the first and the last stage of the control loop. During monitoring, information is obtained via so-called *Sensor*s while during execution *Effector*s are applied for reconfiguration. Figure 1 also covers the interaction with the environment of the considered system during these stages of the control loop. This environment might also contain other autonomic systems. There are also architectures conceivable that make coordination among planning entities of different *Management Layer*s necessary, like e.g. [6]. These are not covered explicitly in figure 1.

To provide a foundation for autonomic management of component-based EA systems, a manageable layer must be established on top of an existing component standard. The remainder of this section discusses three different classes of requirements for such a manageable layer: *Manageability Requirements* ensure sufficient functionality of the layer itself. *Platform Requirements* prevent the system from becoming proprietary. *Development Requirements* address the desired property of an AC-platform to hide manageability aspects from developers of the original business logic by means of preventing them from the need to program in a manner that is explicitly aware of manageability issues. Whereas the first class guarantees the overall functionality of the system, the latter ones should ease its wide-spread use for different containers as well as for existing components.

Manageability Requirements (MR) subsume all requirements directly related to the provided functionality for the management layer. Because they are essential for the establishment, they are indispensable.

MR-1: The manageable kernel must provide facilities or services to allow the management layer to gather information about the structure of managed EAs. This covers the constituting components and connections among them. The requirement is subsumed under the term ***structural inspection***.

MR-2: Support for ***behavioral inspection*** must also be part of a manageable kernel. Interactions among components inside a container and across its boundaries must be observable in a fine-grained fashion. This does not only cover the occurrence of method calls itself but also information about call chains, potentially spanning multiple participating components.

MR-3: While structural inspection mainly address static aspects of the managed layer, behavioral inspection deals with dynamic aspects regarding the occurrence of different situations. To address the particular specifics of the different kinds of inspection, a manageable kernel should provide ***pull- and push-oriented information provision***. While the former type involves information acquisition via the usage of different sensor-interfaces, the latter type relies on information supply through invocation of callback methods or capturing of events. A pull-oriented approach is desirable for obtaining static information, e.g. the set and structure of deployed components. For the occurrence of situations, e.g. the invocation of a method, push-oriented information of the management layer is preferable, both for timeliness- and performance reasons. Otherwise, the management layer has to poll for the occurrence of relevant situations in regular intervals. This would imply the risk of missing them.

MR-4: A managed layer must support a management layer with a rich set of opportunities for ***structural re-configuration*** of EAs during runtime. This covers the possibility to re-organize the internal architecture of an application via re-connecting its parts. Re-configuration should be supported on component- and instance-level, meaning that it should be possible to apply manipulation-instructions generally or for a concrete connection.

MR-5: In addition to the previous requirement which deals with the establishment of new connections, ***behavioral re-routing*** addresses the need to manipulate already existing connections. For this purpose it must be possible to re-route the invocation of a certain method to a new target. In case certain parts of the system should be isolated or protected, a manageable kernel must additionally provide the possibility to ***prevent the execution of incoming or outgoing method calls***.

In summary, the previous requirements together address the need to support *structural* and *behavioral reflection* [20].

MR-6: The information used in the context of the previous requirements has to be based on a ***sound information model***. It should be possible to identify related parts for information items and to put them into a context like, e.g. identifying the source of a method-call. Additionally, a relation must be established between the information obtained from sensors and the information needed at effectors.

MR-7: ***Extensibility*** is needed for the collection of provided sensors and effectors to cope with future needs. While a manageable kernel should provide a degree of manageability as high as possible, one must expect that the included facilities and services are not sufficient for all considerable future application areas. It should be possible to integrate extensions during runtime to prevent the need for a restart of a productive system. Furthermore, potential extensions should get by without any adjustments of certain components of the affected applications which would result in the need to un-deploy, adjust, and redeploy them. In particular, a solution which implies a partial or complete reboot of EAs would be unacceptable.

Platform Requirements (PR) include two aspects for broad usability of a provided kernel.

PR-1: For integration of autonomic management facili-

ties, ***there should be no need for adjusting the implementation of the underlying platform and its corresponding container***. Otherwise – in case a concrete implementation is manipulated – each new release has to be adjusted accordingly. Moreover, the solution would be limited to a concrete container implementation for the addressed component-standard.

PR-2: ***No use of specific platform- or container-provided enhancements*** is permitted. This should lead to the usability of the kernel inside many environments. Similar to the previous requirement, the use of container-specific APIs or services as well as relying on optional parts of a standard provided by some containers would lead to a commitment to a concrete implementation which is not desirable.

In summary, these requirements postulate that the integration of manageability should solely rely on the underlying component-standard. Basically, a violation of one of these requirements points out shortcomings in the implementation of the kernel or indicates that the underlying standard does not specify all aspects needed for the establishment of a manageable kernel.

Development Requirements (DR) address the development stage of the lifecycle of components and the influence of the manageable kernel onto their execution. Their fulfillment should support its acceptance.

DR-1: The ***insertion of sensors and effectors into components should be transparent for developers***, i.e., developers are not responsible for ensuring manageability of their applications, e.g. via the usage of a recommended API. Generally, the integration of capabilities for autonomic management into containers as well as into components should not hinder the tasks of developers.

DR-2: ***No limitations regarding the use of services and facilities provided by the standard*** should be imposed on developers. They should be enabled to develop components as if there is no autonomic management performed.

DR-3: For the management of components there should exist ***no additional information needs***. Consequently, a developer should not be enforced to write additional artifacts besides those recommended by the underlying component standard. Note that this requirement only refers to the basic aspects covered by a kernel. It does not imply that it should be generally avoided to include additional information about entities being target of autonomic management. This might be reasonable for concrete application areas of AC, but the fulfillment of the generic manageability requirements should get by without them.

DR-4: The ***preparation of components should be automated to a very high degree***. It should be possible to provide a standard-compliant, deployment-ready component for which the integration of enhancements should be performed automatically.

DR-5: For the integration of a component into a container ***no complicated deployment process should be needed***. Instead of that, the deployment should be realizable as intended by the provider of the original target container.

3. ENTERPRISE JAVA BEANS 3.0

Enterprise Java Beans represent a standard for distributed, component-oriented EAs implemented with the object-oriented programming language *Java*. The synonym *Write Once, Run Anywhere* ([8], P. 27) stands for two main goals of the development of the standard, namely interoperability and re-usability. It means that a component should be deployable into each container following the EJB-standard without the need to manipulate its source code anymore. Version 3.0 of the standard is available since May 2006. In [9] aspects of persistence management are covered which are of minor relevance for the kernel presented here. Therefore, [8] was considered as foundation for this paper which includes all aspects relevant for development, deployment and runtime of components. The standard was specified under the leadership of `Sun Microsystems` and is supported by well-known companies, e.g. `IBM Corporation` and `Oracle Corporation`. In the following, different aspects of the EJB 3.0 standard are discussed as far as they are relevant for the implementation of *mKernel*.

Building Blocks of Components: EJB-based components consist of a collection of so-called *Enterprise Beans* or *Beans* for short. Within the standard there are three different types of beans considered. Namely these are *Message Driven Beans*, and *Stateless-* and *Stateful Session Beans*. Message driven beans can be used via sending asynchronous messages and provide no additional interfaces. Session beans provide interfaces of which the standard considers different types. The main difference between the two types of session beans lies within the provision of a client-specific state. Instances of stateful session beans are exclusively used by a single client and retain their state across multiple invocations. Consequently, this state is specific for a single client. Moreover, a client can rely on interacting with the same instance in case it uses the same reference for multiple invocations. Stateless session beans in contrast are usable by the container for handling method invocations originating from different clients. Furthermore, it is not guaranteed that a client, performing more than one method invocation on the same reference, is always interacting with the same session bean instance. An instance might keep its state during its lifetime. This client-neutral state might be the source of performance benefits, e.g. in case an open database connection is kept for reuse. One important property of session beans is, that they are by definition non-reentrant. Therefore, it is not possible that more than one method call is active on a session bean instance at any given time. Moreover, bean instances are not allowed to perform any kind of thread handling like e.g. starting new threads. A component is provided in terms of a so-called *Bean-module*. Besides the constituting beans, such a module covers additional artifacts like, e.g. a *Deployment Descriptor* (DD).

Component Composition: For gaining access to session beans and their interfaces, the container must – according to the standard – provide two alternative facilities which can be used in combination. On the one hand, an implementation of the *Java Naming and Directory Interface* (JNDI) [2] must be provided by each container which enables the lookup of bean references during runtime by submitting their name. It has to be pointed out that the entries of this naming facility can not be manipulated directly. On the other hand, in the context of the so-called *Dependency Injection*, dependencies of enterprise beans can be declared during development. This, amongst others, also covers the specification of receptacles. During execution, these are bound to a concrete session bean instance.

Interaction Control: To each enterprise bean an arbitrary number of *Interceptor*s can be attached which includes

a specification of methods the interceptor is interested in. In case a method should be invoked upon a bean instance, the invocation is firstly directed to the matching interceptors, if any. These interceptors gather full control over the control flow and the submitted parameters. They might e.g. analyze or change parameters, or prevent the invocation from reaching its original target. The return value can also be subject of inspection and manipulation.

Component Specification: During implementation of a certain bean, developers can integrate different *Metadata-Annotations* into the source code for configuring the corresponding bean. It is e.g. possible to specify interfaces and receptacles as well as interceptors to attach. Through an XML-based DD, included in the component, it is possible to configure the beans of the corresponding component. The options for configuration cover all aspects of the metadata-annotations and open up additional opportunities on component-level, e.g. to attach a certain interceptor to all beans of a module. In case certain annotations refer to the same aspects of a bean as parts of the DD, the content of the DD is privileged. Hence, it is possible to adjust the configuration of a component and its constituting beans respectively without the need to manipulate their source code.

Component Lifecycle: After the development of the constituting beans is completed, they are assembled into a module. As preparation for its integration into the target container, a subsequent configuration can be performed upon the component. There is no procedure designated within the standard to adjust the configuration of a component during runtime. Consequently, the deployment is the latest time for the specification of configuration aspects discussed above concerning a certain module. A re-configuration must be performed outside the container and applied via redeployment of the affected module.

Lifecycle of Enterprise Bean Instances: For the instantiation of beans the EJB-standard specifies a special proceeding. A detailed discussion of the corresponding states, methods, and the specifics for the different bean types is omitted here for brevity. It has to be pointed out, that the injection of references for dependencies and the authorization to use them is performed in two separate steps consecutively. Only if the injection phase has finished completely, the bean instance is allowed to invoke operations upon the provided references. The beginning of the usage phase can optionally be identified by receiving the invocation of a so called `PostConstruct`-method. No method invocations regarding its application logic will be forwarded to the bean instance before the usage phase has started. All other state-transitions during the lifecycle of a bean instance are observable through similar method invocations, too. Moreover, all of these invocations are firstly directed to interested interceptors attached to the instance, if any.

Within the EJB-standard there are no subordinate aspects included which allow the supervision of the current state of a container regarding instances of beans, established connections among them, or ongoing method invocations. Moreover, it is not intended to re-configure components after their deployment.

4. ARCHITECTURE

This section presents the architecture of *mKernel* and the different components which have to be integrated into a container for making it ready for autonomic administration according to the requirements discussed in section 2. *mKernel* was built and tested on top of the `Glassfish Application Server` [1] which provides, amongst others, facilities for the *Java Platform Enterprise Edition*, an EJB container, and has proven itself of being compliant to the EJB standard to a very high degree. Therefore, it was considered a good foundation for the development. The facilities presented here, are realized as standard-compliant EJB modules and make use of the infrastructure provided by the container.

Naming: The implementation of JNDI, which is mandatory for each EJB-compliant container, is used to lookup references to session bean instances via submitting their name. Because *mKernel* is developed for usage in a multi-container environment, the lookup of session bean instances residing in a remote container is also be supported. Regarding this aspect, the EJB-standard has a shortcoming in that there is no specified standard-compliant way of how to dynamically access a session bean instance residing in a remote container through its *Remote Business Interface*. In fact, the `Glassfish Application Server` does not even provide a container-specific opportunity for dynamic connection establishment for this type of interface across container boundaries. Because remote business interfaces are the preferable choice for using the application logic of session beans, a solution had to be found. As foundation for naming, the *dyName* system was applied which we developed as independent solution for dynamic naming in EJB-based systems. *dyName* provides, amongst others, the required functionality. It is solely based on facilities specified by the EJB-standard, requires no enhancements of the applied container, and is encapsulated inside *mKernel*. Consequently, the application of *dyName* does not violate any of the requirements stated in section 2. For a detailed discussion of *dyName*, refer to [5]. The *Naming* facility is neither a sensor nor an effector but used as infrastructure for connection establishment.

Connector: While the *Naming* facility enables the actual establishment of connections, the *Connector* facility supports the specification of targets for connections during runtime. A request is submitted to the *Connector* including the originator of the request and a target mapped name as used for connection establishment in the context of the EJB-standard. As response the *Connector* delivers information processible by the *Naming* facility. Managing entities can define connection targets on different granularity levels, namely on container-, module-, bean- and instance level. Moreover, the *Connector* also provides the opportunity to re-route existing connections. In combination, this allows a fine grained steering of interactions taking place among managed beans. The *Connector* does not support any kind of state transfer from the original to the new target of a connection which would especially be critical in case a connection to a stateful session bean instance should be switched. We assume this of being application specific and not being solvable in a generic fashion. Consequently, the *Connector* facility provides an effector with a rich variety of options for controlling interconnections inside the managed layer. Therefore, the *Connector* is an effector allowing to re-configure the architecture of a component system during runtime.

Deployment Information: The *Deployment Information* facility is part of each module deployed in an autonomously manageable container. It provides information about included enterprise beans. The information covers all

relevant aspects of enterprise beans including amongst others interfaces, receptacles, and simple environment entries. Consequently, this facility delivers information about deployed components on type level. This also includes unique identifiers for each particular interface and concrete bean implementation. Consequently, it is possible to, e.g., identify all deployed implementations of a given interface to find candidates for a certain receptacle. In case of the identification of an error inside one bean implementation, the affected components can be easily found. Because of the allocation of the deployment information inside the corresponding module, it implies a built-in up-to-dateness. On removal of a component, its part of the information about the overall structure is implicitly removed, too. With the *Deployment Information* facility, a sensor is provided which allows – in combination with the information covered within the *Connector* – structural inspection of the managed layer.

Events: This facility represents a broker for event producers and -consumers. The event types provided for bean instances correspond to the lifecycle of beans as specified in the EJB-standard. Namely, these are the construction and destruction of all bean types, and additionally the passivation and activation of stateful session beans. Moreover, events for business calls on session bean instances and for message reception at message driven bean instances can be captured. The occurrence of exceptions in all of these contexts is also supported via corresponding events. The information provided for each event includes aspects of the context of its occurrence, like e.g. identifiers for the corresponding bean instances as well as for establishing a relation to the information of the *Deployment Information* facility. Additionally, for business calls, it is possible to deduce call chains spanning multiple bean instances which also covers the identification of the invoked methods. The non-reentrancy property of enterprise beans, in combination with the prohibition of starting new threads, leads to the opportunity to clearly identify dependencies among methods observed in the managed system. Local sequence numbers as well as information about the time and duration of an invocation are also included, which in combination allow an ordering of captured method invocations and an analysis of the fractions of the overall processing time of each call with respect to the different sub-calls, if any. Event consumers can register at the *Events* service through the provision of identifiers for producers of events in combination with a set of event types they are interested in. Again – as discussed in the context of the *Connector* – a fine-grained specification of producers is possible on container-, module-, bean-, and instance level. As a result a consumer receives a lease which it can renew on expiration if it is interested in obtaining the corresponding events further on. Producers are instructed to throw events via two complementary ways. On each registration of a consumer and on lease expiration, the affected modules are identified and instructed to start or stop producing the corresponding events. In case a module is deployed, there might already exist matching registrations. On first invocation of a bean instance of the new module, the submission of matching registrations is requested from the *Events* service for initialization. Afterwards, the module will be considered during each subsequent registration of a consumer as well as on lease expiration. Events are distributed via an approach consisting of two steps. Firstly, events are stored locally which allows a fast continuation of the method call under consideration. Secondly, a stateless session bean, which is integrated into each managed module, checks in regular intervals if there are any events to distribute. If so, these are published via *Java Message Service* (JMS) [15] applying a corresponding *topic* which is bound at a well known name inside the namespace of a container. This allows the asynchronous distribution of events to multiple consumers. Consequently, the *Events* service itself does not need to keep track of the different consumers. For each producer-type-pair it is only necessary to store the latest lease-timeout. In summary, the *Events* facility provides a push-oriented sensor for making the runtime behavior of a managed system observable.

Interceptors: The facilities discussed up to now provide sound effectors and sensors for the autonomic management of an EJB-based container. The *Interceptors* facility represents a generic opportunity to integrate additional aspects not already considered within *mKernel*. It allows the integration and removal of interceptors into a running component system. Again, the targets of interception can be controlled in a fine-grained way as discussed for the *Events* facility. Via interceptors it is possible to intercept any method-invocation upon enterprise bean instances covering calls on interfaces as well as calls regarding state transitions during their lifecycle. In this context, it is possible to be informed about the occurrence of a specific method invocation, to gain insight into the parameters of the call, to manipulate those parameters, or even to prevent the call from being forwarded to its original target. The return from a method invocation can also be intercepted which includes the opportunity to analyze and manipulate the return value. *mKernel* provides the opportunity to specify which of the different places of possible interception are of interest for a certain interceptor. It is e.g. possible to only intercept business method before they are reaching their original target, or to only intercept exceptions. Furthermore, context information for a method invocation is provided also, e.g. an identification of the caller, if known. The implementation was inspired by the *Interceptors* facility already designated as part of the EJB standard, but it provides a much higher degree of flexibility w.r.t. the built-in feasibility to re-configure the set of attached interceptors during runtime which is not provided by the EJB standard. *mKernel*-interceptors are realized through session beans, which means that the provider of an interceptor has to implement a certain interface and deploy the implementation in form of a module. Afterwards, the new interceptor must be registered at the *Interceptors* service. Interceptors are intended to be used temporary, e.g. during re-configuration or as reaction to failures. It is also possible to attach interceptors to beans permanently, but it has to be kept in mind that each invocation of an *mKernel*-base interceptor implies the invocation of an additional session bean which leads to a certain overhead. With the *Interceptors* facility a combined sensor and effector is provided which allows a fine grained intervention into interactions on *Managed Layer*.

Two of the facilities presented, namely *Naming* and *Interceptors*, can be seen as extensions of the facilities already considered in the EJB standard. For those, serious limitations were identified which had to be overcome for making them usable in the context of runtime re-configuration. Consequently, they were taken as foundation upon which the *mKernel*-specific facilities were realized. *Deployment Infor-*

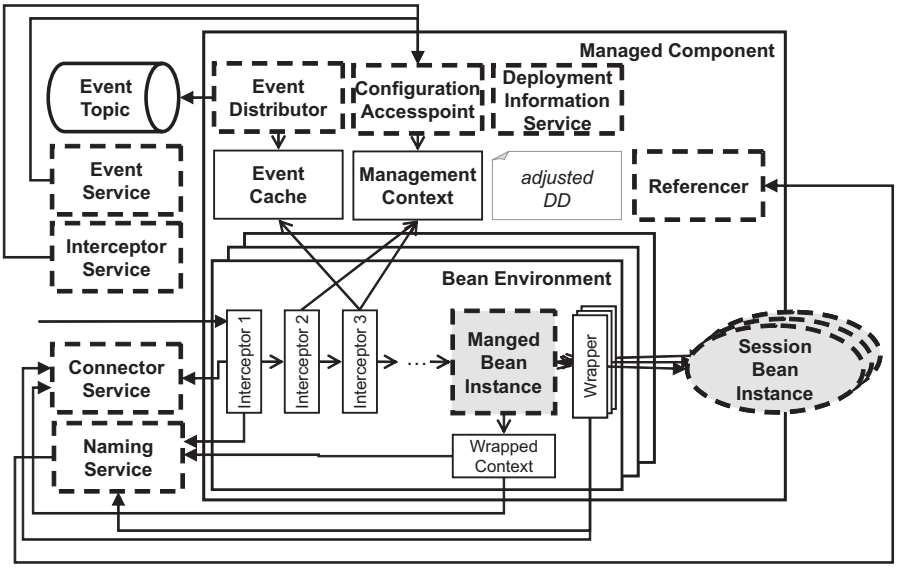

Figure 2: Elements of a managed component

mation was inspired by [17]. Because of the specific view, *mKernel* takes on a managed system and because of limitations identified, the standard was neither usable directly nor as foundation. The other two facilities do not have any corresponding counterpart in the EJB standard itself or in related standards.

5. MANAGEABLE COMPONENTS

For making an EJB-module manageable, it has to be preprocessed by a tool being part of *mKernel*. This tool accepts a standard-compliant *Java Archive* (JAR) containing an EJB-module ready for deployment without any further configuration being necessary. The content of this module is manipulated and extended for making it autonomously manageable. Analysis, manipulations and extensions on bean level are performed via application of the *Java Programming Assistant* (Javassist) [7] which contains, amongst others, a very convenient API for analyzing and manipulating Java bytecode. The DD of the module is processed with the aid of the *Java Architecture for XML Binding 2.0* (JAXB) [23]. For the generation of a new DD, covering manageability aspects, JAXB is also used. The steps performed upon a module by the tool are discussed in the following. For each enhancement performed, its particular contribution for *mKernel* is explained. Figure 2 presents an overview over the results of preprocessing a component. Additionally, it covers relations between integrated parts and the services discussed in section 4.

The first step during module preprocessing includes the extraction of the submitted JAR. This is performed to keep the submitted module in its original state for being usable without *mKernel*.Secondly, all class-files of the application are analyzed on bytecode-level – especially the metadata-annotations of the included beans – to collect information about the component regarding e.g. provided beans and declared dependencies. With this information a representation is generated that contains all relevant information. Afterwards, the DD of the module is parsed and a representation is generated also. Via merging of the two representations according to the demands of the EJB-standard a comprehensive image of the inspected module is gained.

All interfaces and receptacles are extended by means of enhancing all provided methods with an additional parameter used by *mKernel* for forwarding context information along call chains. The original methods are provided further on to allow an external usage of the beans without even recognizing that they are managed by *mKernel*. The affected session beans are extended accordingly. Internally, the new method bodies solely consist of an invocation of the corresponding original methods. Therefore, the submission of context information is not even noticed by bean instances.

For all receptacles *Wrapper*s are generated. These are used as replacement for connections to session beans during their creation. As shown in figure 2, all interaction among bean instances is performed through these wrappers, allowing the interception of the control flow. On invocation, wrappers contact the *Connector* which instructs them how to proceed. In case the connection should be switched, the *Naming* service is contacted for obtaining a new interaction endpoint.

To prevent the establishment of connections that bypass *mKernel*, the usage of the container-provided naming facility is replaced with an alternative implementation without any effect on the container itself. This is done through integration of a *Wrapped Context* in combination with manipulating each part of the bytecode trying to open a connection to the naming facility of the container. This is redirected to the *Connector* and the *Naming* service of *mKernel*.

As part of the `class`-files the component is enhanced with, a *Management Context* is integrated. This class acts as configuration cache for the corresponding module and its constituting beans. Amongst others, it holds configuration information regarding aspects of the different services of *mKernel*, e.g. information about interceptors to re-route invocations to and directives for event types to throw as discussed in section 4. Configuration of a module is performed by the corresponding service via addressing the so-called *Configuration Accesspoint* which is realized as stateless session bean and integrated into each managed module. The context itself only caches this information and does not store it permanently. All information covered can be requested from the corresponding services at any given time.

Next, a so-called *Referencer* is configured and integrated into the module in process. It is an additional stateless session bean which provides references to session beans of the module. It is used by the *Naming* service of *mKernel* for being able to perform connection establishment.
Afterwards, the partition of the *Deployment Information* service covering information about the component is configured appropriately. It is integrated as stateless session bean into the module.
To prohibit the direct injection of references during preparation of a bean instance, all dependency declarations are removed from the bytecode as well as from the DD of the module. Instead of those, an additional interceptor (*Interceptor 1* in figure 2) is integrated as first interceptor of the interceptor chain attached to any given bean. This interceptor imitates the dependency injection of the container by contacting the *Connector* and the *Naming* service, and injecting wrappers for all removed dependencies, if any. This is performed on intercepting a method call indicating that the usage phase of the lifecycle has started. Regarding the lifecycle of a bean instance in combination with the proceeding prescribed for the container, the target instance still believes that it is in the dependency injection phase and consequently does not make use of references designated for dependency injection. After finishing the establishment of connections, the interceptor forwards the lifecycle call to the target instance. Through this proceeding, there is no difference recognizable for the implementation of the affected beans.
To provide the *mKernel*-based interceptors with sufficient control over incoming method calls, another interceptor (*Interceptor 2*) is attached to each bean. This interceptor is inserted as second one in the interceptor chain. As discussed above, the first one also belongs to *mKernel*, being only responsible for dependency injection. Consequently, no interceptor attached by a developer or deployer can gain access to the original method parameters or, in case the invocation should not be forwarded, even realizes its occurrence. Established connections to interceptors are kept for future use. Consequently, it is easily possible to apply instance-specific interceptors via stateful session beans even for managed stateless session beans.
For tracking of events, a third interceptor is attached to each pre-processed bean (*Interceptor 3*). It requests directives to follow from the management context and stores the relevant events inside the *Event Cache*. For distribution of events, the *Event Distributor* is integrated into each managed module as stateless session bean. It checks the event cache in regular intervals for events to distribute and, if any, sends them as messages through a JMS-based *Event Topic*. At this topic interested event consumers can register for being informed about events. This is not covered in figure 2. The three dots following *Interceptor 3* in figure 2 stand for the interceptors attached by developers or deployers. The call is transmitted to them afterwards.
During all of the processing steps, the image of the module is adjusted accordingly. It is afterwards translated into a corresponding DD and integrated into the target module.
After completion of the previous steps, all preparations of the target module are finished. Finally, the resulting module is packed into a JAR which is 100 % compliant to the EJB-standard. The integration of enhancements is transparent to the application logic of the beans inside and outside the module, i.e. their runtime behavior is not affected.

6. EVALUATION

This section addresses two aspects. Firstly, *mKernel* is evaluated against the requirements stated in section 2. Secondly, the results of a performance analysis are discussed.

With *mKernel* the functionality required by a managed layer in the context of AC is provided. An evaluation against the requirements stated in section 2 is shown in table 1. The rating in the last column includes the possible values **-**, ***0*** and *+*. Here, **-** indicates that the corresponding requirement is not fulfilled by *mKernel*. A ***0*** is given in case the requirement is addressed and supported, but there exists a demand on improvement. If the requirement is fulfilled completely a *+* is inserted.

Table 1: Functional evaluation of mKernel

Manageability Requirements (MR)		
ID	**Remarks**	**Rating**
MR-1	*Deployment Information* Service and *Connector*	+
MR-2	*Events* Service	+
MR-3	pull-oriented: *Deployment Information* Service push-oriented: *Events* Service	+ +
MR-4	Combination of Interceptors, Wrappers, *Connector* and *Naming* Service	+
MR-5	Combination of Interceptors, Wrappers, *Connector* and *Naming* Service	0
MR-6	Built-in assignment of identifiers and tracking during interaction	0
MR-7	*Interceptors* Service	+
Platform Requirements (PR)		
ID	**Remarks**	**Rating**
PR-1	–	+
PR-2	shortcomings of standard	0
Development Requirements (DR)		
ID	**Remarks**	**Rating**
DR-1	automated via tool	+
DR-2	–	+
DR-3	EJB-compliant module is sufficient	+
DR-4	automated via tool	+
DR-5	only execution of tool needed	+

As shown in table 1, *mKernel* fulfills most of the requirements completely. Regarding MR-5, it has to be pointed out that the re-routing of existing connections is covered by the implementation, but no support for state-transfer among connection-targets is included. This was assumed to be application specific and, hence, should be handled outside of *mKernel*. Identifiers are assigned to modules and bean instances. During interaction context information regarding the control flow is also provided. This allows the tracing of interactions inside a managed container. Moreover, the *Deployment Information* service delivers rich information about deployed modules of a container. However, the underlying information model covers some shortcomings and should be subject to revision. There is no facility provided by *mKernel* regarding tracking and provision of historical data, i.e., there is no logging-facility. Altogether, this leads to a rating of ***0*** for MR-6. It has to be pointed out,

that this aspect does not comprise any unsolved technical aspects. It is solely addressable via processing of data already provided by *mKernel*. The EJB-standard does not specify the deployment process of components itself including the preparation of data source for persistence, but leaving this open for vendor-specific solutions. Additionally, the design of the naming schema for the JNDI-implementation inside a container is not addressed by the standard. It is not even specified how a name can be assigned to a bean in a standard-compliant way that has to be followed by all container implementations. Consequently, it was not possible to develop a management layer solely relying on the EJB-standard. For *mKernel* the demands of the `Glassfish`-container were preserved. In case a migration to different container-implementations should be performed, these aspects must be addressed. Therefore, PR-2 was rated ***0***.

For evaluating the performance impact of applying *mKernel* in an EJB-container, there were two sample scenarios used. Each of those was analyzed for stateful and stateless session beans. The first one should show the overhead for using the different facilities of *mKernel* without any side effects regarding application logic inside the module. Therefore, a simple session bean, providing a single method without any parameters was taken. The method delivers no return value and no application logic is present. The second scenario addresses the interaction with entities, because this is supposed to be the case for most of the EJB-based applications. In this scenario a session bean is used for accessing a datasource. Therefore, an interface is used, providing methods for creating, reading and deleting a single entity consisting of a `String`-value. An entity manager is obtained through dependency injection, and the entities are managed via *container managed persistency*. Together, the scenarios should grant a first insight of how performance is influenced by *mKernel* for session beans. Each of the scenarios was analyzed through four settings to get an insight into the impact of the different facilities. As reference measurement, the module itself was deployed and executed without extensions and without any part of *mKernel* being installed in the container (*base*). The second setting (*silent*) takes an installation of *mKernel* without any of its facilities being equipped with directives, meaning that no consumers for events and no interceptors are registered. The two remaining settings address the *Event*- and the *Interceptor* service. Within the first one (*event*), the *Event* service is instructed to throw events for all business calls. For the second one, an interceptor is registered which internally does not have any application logic, because only the overhead for re-direction is of interest. It intercepts any incoming method call before it is processed by the target bean instance.

As hardware foundation for the evaluation runs, PCs were used, equipped with a 2,4 gigahertz Pentium IV processor with Hyper-Threading, 1 gigabyte random access memory, and a 100 Megabit (MBit) ethernet card. As operating system Windows XP was installed. The runs were performed in a 100-MBit local area network where all hosts were connected to the same switch. As application server `GlassFish v2 build 45` was chosen. Of this build the standard installation without any adjustments was taken as foundation. The client side of the scenarios was connected to the application server through the `appclient` being part of the build. For each scenario a single client application performed a certain number of method invocations upon the target bean. After each single run, for the application server a new installation was prepared.

It has to be pointed out that the evaluation performed should by no means be interpreted as benchmarks for the underlying configuration. Especially the application server itself was not subject of any performance analysis. For measuring the overhead caused by the application of *mKernel*, the percental overhead ($overh_s$) of each run with *mKernel* was calculated via setting the arithmetic mean ($\overline{x}_s$) for each scenario s in relation to the arithmetic mean of the run without *mKernel* ($\overline{x}_{base}$):

$$overh_s = \frac{\overline{x}_s}{\overline{x}_{base}} * 100\%$$

For the first scenario, the client of the stateless session beans performed 50000 subsequent method invocations. For the stateful session bean case, a connection to a bean instance was established. Afterwards, five subsequent invocations were performed before connecting to a new instance. This was repeated 10000 times, again leading to 50000 observed invocations. For the second scenario, each client established a connection to a session bean for the stateless and stateful case. Afterwards, each type of method – create, read and delete – was invoked once. This was repeated 10000 times, leading to 30000 observations for each setting. Table 2 covers the results of the evaluation. Here, the '.' in the percentage values is used as decimal place.

Table 2: Performance evaluation of mKernel

No application logic		
s	$overh_s(Stateless)$	$overh_s(Stateful)$
silent	4.334 %	1.581 %
event	12.276 %	10.801 %
intercept	75.487 %	116.546 %
Database access		
s	$overh_s(Stateless)$	$overh_s(Stateful)$
silent	0.942 %	0.408 %
event	4.364 %	2.678 %
intercept	30.05 %	49.945 %

Summarizing, one can state that the overhead for the application of *mKernel* lies within acceptable boundaries. The results derived for the application of the *Interceptor* service can be explained by the fact that each invocation on an instance of a session bean is re-directed to a separate session bean instance, i.e. the interceptor. Additionally, for each instance of a session bean, a reference to each interceptor must be obtained on first invocation. This might have led to the results for the *intercept*-settings in combination with stateful session beans. It has to be pointed out that the different number of subsequent invocations for the two scenarios also had an important influence on the results. For the first scenario, each fifth invocation led to a connection establishment for stateful session beans while this was the case for each third invocation for the second scenario.

7. CONCLUSION AND FUTURE WORK

In summary, the implementation of *mKernel* showed that the EJB-standard provides a sound foundation for extensions making it manageable autonomously. However, because of the shortcomings found in the standard, it is not

feasible to provide an implementation solely relying on it. Instead, it is necessary to deal with container-specific solutions. All required aspects for manageability are realizable to a high degree. Additionally, the results of the performance evaluation showed that the overhead of integrating manageability into an EJB-container is justifiable. In this context, *mKernel* provides a domain-specific manageable layer for EA on top of a broadly accepted standard.

However, the evaluation revealed different areas of possible extensions for *mKernel*: At the time of writing, a comprehensive API is under development. It should replace the need to interact with the facilities directly for management purpose. Currently, the manageability spans those components deployed into the administrated container. It would be desirable to expand manageability with options for manipulating the collection of deployed components itself. At the moment, an additional facility for *mKernel* based on [11] is realized. Additionally, it is considered in how far application specific configuration of components can be supported during runtime. As first idea, the manipulation of *simple environment entries* is planned similar to the way dependencies are treated by *mKernel* already. Finally, it is planned to develop broader sample applications to evaluate and show the potential of *mKernel* for different application areas.

8. REFERENCES

[1] The Glassfish Application Server. glassfish.dev.java.net.

[2] Java Naming and Directory Interface (JNDI). http://java.sun.com/products/jndi/.

[3] T. Abdellatif and A. Danes. A simple approach to autonomic J2EE servers. In *IEEE International Conference on Self-Organization and Autonomic Systems in Computing and Communications (SOAS'2006)*, Erfurt, Germany, 2006.

[4] S. Bouchenak, F. Boyer, S. Krakowiak, D. Hagimont, A. Mos, S. Jean-Bernard, N. de Palma, and V. Quema. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 13–24, Washington, DC, USA, 2005. IEEE Computer Society.

[5] J. Bruhn and G. Wirtz. DyName: Enhanced Naming for EJB. In *Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2007)*, volume 1, pages 17–23, 2007.

[6] D. M. Chess, G. Pacifici, M. Spreitzer, M. Steinder, A. Tantawi, and I. Whalley. Experience with Collaborating Managers: Node Group Manager and Provisioning Manager. In *Proceedings of the Second International Conference on Autonomic Computing 2005 (ICAC 2005)*, pages 39–50. IEEE, 2005.

[7] S. Chiba. Javassist. http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[8] L. DeMichiel and M. Keith. JSR 220: Enterprise JavaBeans, Version 3: EJB Core Contracts and Requirements. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, 2006.

[9] L. DeMichiel and M. Keith. JSR 220: Enterprise JavaBeans, Version 3: Java Persistence API. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, 2006.

[10] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung. Self-Managing Systems: A Control Theory Foundation. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 441–448, Washington, DC, USA, 2005. IEEE Computer Society.

[11] J. Dochez. JSR 88: Java Enterprise Edition 5 Deployment API Specification, Version 1.2. http://jcp.org/aboutJava/communityprocess/mrel/jsr088/index.html, 2006.

[12] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: An Autonomic Computing Environment. In *Proceedings of IEEE International Conference on Performance, Computing, and Communications (IPCC)*, pages 61–68, 2003.

[13] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. In *IBM Systems Journal*, volume 42, pages 5–18. IBM, 2003.

[14] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.

[15] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. JSR 914: Java Message Service (JMS) API. http://jcp.org/aboutJava/communityprocess/final/jsr914/index.html, 2002.

[16] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, 2001. IBM Corporation.

[17] H. Hrasna. JSR 77: Java 2 Platform, Enterprise Ed. Management Specification. http://jcp.org/aboutJava/communityprocess/mrel/jsr077/index.html, 2006.

[18] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer Magazine*, 36(1):41–50, 2003.

[19] P. Lin, A. MacArthur, and J. Leaney. Defining Autonomic Computing: A software Engineering Perspective. In *ASWEC '05: Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)*, pages 88–97, Washington, DC, USA, 2005. IEEE Computer Society.

[20] M. S. Sadjadi, P. K. Mckinley, B. H. C. Cheng, and K. R. E. Stirewalt. TRAP/J: Transparent Generation of Adaptable Java Programs. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, pages 1243–1261, 2004.

[21] R. Sterritt and D. Bustard. Towards an Autonomic Computing Environment. In *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, page 699, Washington, DC, USA, 2003. IEEE Computer Society.

[22] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[23] S. Vajjhala and J. Fialli. JSR 222: Java Architecture for XML Binding (JAXB) 2.0. http://jcp.org/aboutJava/communityprocess/final/jsr222/index.html, 2006.