The LEAF Platform: Incremental Enhancements for the J2EE

Philipp H. Oser[1], Christian Gasser[1], Daniel Gorostidi[1], Rachid Guerraoui[2]

[1]ELCA, Av. de la Harpe 22-24, 1000 Lausanne, Switzerland
{pos, cga, dgo}@elca.ch
[2]Swiss Federal Institute of Technology (EPFL), 1015 Lausanne, Switzerland
Rachid.Guerraoui@epfl.ch

Abstract

LEAF, the Lean and Extensible Architectural Framework, is an enhancement wrapper for J2EE implementations. Basically, LEAF fixes some identified J2EE issues and extends, as well as simplifies, the use of the J2EE by providing several incremental improvements. These improvements are seamlessly integrated, include an additional component type, allow the same interfaces for local and remote service implementations, offer better J2EE implementation compatibility and ORB interceptors, and encompass several new technical services.
This paper explains the need for LEAF through a diagnosis of the J2EE, presents the fundamental concepts underlying LEAF, overviews its implementation, reports on field experiences from using it in a number of commercial projects, and points out some interesting tradeoffs in using the J2EE with and without LEAF.

Keywords

J2EE, Enterprise Computing Platform, Distributed Components, IT Architecture, EJB, Service Location Transparency

1. Introduction

Development platforms are a big help in building enterprise information systems. They encapsulate and abstract away many technical details related to data storage and communication, allowing the developer to focus more on business logic. The Java 2 Enterprise Edition (J2EE) [1] is such a platform. It is oriented towards enterprise computing and has gained a lot of industry momentum due to its technical qualities and its operating system independence.
As an IT services company, we are naturally interested in a sound enterprise platform that can provide a competitive advantage through increased productivity, simplified reuse of components across projects, and increased quality of the resulting software. Because our customers each have their own vendor choices and sourcing strategies, we are particularly attached to the platform vendor independence promises of the J2EE. Unfortunately such promises are in general relative, and it is usually impossible to rebuild an existing J2EE application for a different J2EE implementation without changes. The recent EJB specifications [2] lead to improved J2EE implementation compatibility, but a migration between different J2EE implementations is still not automatic. In addition, as J2EE implementation vendors try to differentiate and extend the J2EE, these incompatibility issues are unlikely to be resolved. While the initial reasons for these incompatibilities were a lack of standardization and a quickly evolving specification, they have now become mostly inherent to the various extensions of the J2EE implementations. Besides these J2EE implementation incompatibilities, we identified other limitations: the lack of support for daemons, singleton services and batch jobs, the limited service location transparency of services, and lack of flexibility of J2EE services.
After giving a comprehensive diagnosis of the J2EE, this paper presents LEAF[1], the Lean and Extensible Architectural Framework, our thin enterprise platform based on the J2EE. LEAF wraps a J2EE implementation and seamlessly adds several incremental improvements. It adds a thin abstraction layer on top of the J2EE ORB, a new component type with its runtime environment, and several useful technical services that either enhance existing J2EE services or provide complementary functionality. LEAF has been fully implemented (70'000 lines of code) and put to use in several practical settings. Our experiences were positive, and we argue that leanly wrapping an emerging enterprise platform is valuable, as our experiences unanimously indicate.

The paper is structured as follows. We present some problematic issues underlying the J2EE in Section 2. In Section 3, we show how LEAF resolves these issues and further extends the J2EE. In Section 4, we overview our

[1] LEAF is only our internal name, no trademark.

implementation of LEAF. Section 5 explains how LEAF was successfully used in several commercial projects within and outside our company. Section 6 discusses related work. Section 7 summarizes the paper and looks at the future of LEAF's evolution.

2. J2EE limitations

The Java 2 Enterprise Edition (J2EE) platform provides a simplified approach to developing highly scalable and high-availability Internet- or intranet- based applications. It extends the Java 2 Standard Edition (J2SE) with many enterprise-related APIs, the Web and the EJB component model, and runtime containers to host Web- and EJB components. The Enterprise JavaBeans (EJB) component model, a significant part of the J2EE, is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture can be made scalable, transactionally safe and multi-user secure.

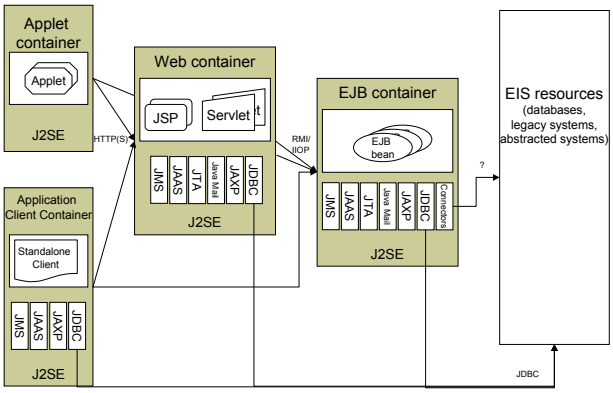

Figure 1: J2EE Architectural Diagram

J2EE is indeed a *very powerful platform* (which is why we have based LEAF entirely on it!). However, there are limitations and, hence, room for incremental enhancements.

We have identified the following limitations in the J2EE: (1) the limited location transparency of services, (2) the lack of support for daemons, singleton services and batch jobs, (3) missing flexibility of some J2EE services, and (4) J2EE incompatibility issues. We will discuss these limitations in the following section. Note that we refer to the most recent J2EE platform version (1.3) unless mentioned otherwise.

2.1. Potential remote use of a service implies a particular interface

Services in the J2EE that may be used remotely (i.e., EJB beans and RMI services) need particular remote interfaces, which must extend `java.rmi.Remote`. All methods that can be invoked remotely need to throw a *checked* `java.rmi.RemoteException`. A rationale for this separation of local and remote interfaces is discussed in [5]. We believe this separation is often too strong. It makes it inconvenient to use the same interface for a service that may either be used locally or remotely. It also prevents the design of the logical component with its service interfaces before mapping the service implementations to their execution environment. Sometimes, even different mappings of service implementations are useful when applications are deployed differently, e.g., for performance improvements or security reasons. As Java requires catching checked RemoteExceptions where they occur unless they are rethrown, it furthermore requires the handling of communication-related problems[1] local to the method calls where they occur. However, a sensible treatment of this exception is rarely possible locally because communication problems are usually fatal for the application. The Business Delegate pattern [8], therefore, proposes that any network- or infrastructure-related exceptions be translated into business exceptions to shield clients from knowledge of the implementation specifics of business services.

2.2. Missing component types

The J2EE lacks three types of components that we tend to use often in our enterprise applications: (1) long-running batch jobs, (2) services that act as global singletons, and (3) daemons. These components do not fit into the EJB component model because EJB components may not run for a "long time"[2], they may never act as a singleton, and they have to respect several limitations, e.g., they must not use threading, listen on a server socket, use static variables changing during runtime, or use native code. An example of a daemon is an SMS gateway that allows forwarding SMS messages to a telecom provider and invokes a service whenever an SMS message arrives. User session management requires the global singleton characteristics.

Some EJB experts discourage the use of EJB when "you can't deal with the limitations of EJB" [2]. However, we are convinced of the advantages of the EJB model and, therefore, would like to use EJB even though some of our components do not fit in the model because it is beneficial

[1] RemoteException is thrown in case of problems related to communication (problems with connection establishment, marshalling, or the remote server).

[2] EJB components are designed for short requests only. Long calculations result in transaction timeouts and are interrupted by the EJB container.

to have a platform supporting EJB and non-EJB in a homogeneous way. There are other ways to implement such components outside the EJB container, but they require more manual infrastructure work, and they usually do not provide container support for the components.
With most of these alternative solutions, components developed for one project are not easily reusable in other projects because the infrastructure tends to be interwoven with the component's implementation, and different assumptions about the infrastructures make the mixing of components from different projects difficult.

2.3. Inappropriate J2EE services

We believe that the *combined* naming and configuration service of the J2EE, the Java Naming and Directory Interface (JNDI), should be split in two separate services, a naming service and a configuration service, because they are fundamentally different and because they are used differently. Splitting them would cleanly separate the two concerns [6]. Both services allow basically retrieving entries under a given name, but the similarity ends there. The organization of the entries in the naming service is usually simpler (usually not hierarchical), and in the case of replicated services the stubs in the naming service might have to change after failures, unless the stubs themselves are intelligent. The entries of the naming service are usually not modified by the programmer while configuration entries can be adapted during runtime, and, finally, the semantic of an element returned by the naming service is usually a newly created stub, while a configuration service will most likely return the same value object each time it is called. The J2EE, therefore, splits the acquisition of a stub into two phases (Listing 1): retrieving a home interface and invoking the create method on that interface (for EJB beans).

```
// Set up the JNDI context, here for JBoss:
Hashtable prop = new Hashtable();
prop.setProperty("java.naming.factory.initial",
       "org.jnp.interfaces.NamingContextFactory");
prop.setProperty("java.naming.provider.url",
                 "localhost:1099");
try {
  InitialContext ctxt = new InitialContext(prop);
  // Get a reference to the bean's home interface
  Object ref=ctxt.lookup("interest/Interest");
  InterestHome home = (InterestHome)
  PortableRemoteObject.narrow(ref,
InterestHome.class);
  // Create an Interest object from the home
  // interface
  Interest interest = home.create();
  // use the bean
}
catch (CreateException ce) { … }
catch (RemoteException re) { … }
```

Listing 1: What the J2EE requires to retrieve a stub for an EJB bean.

A common design is to wrap this two-step process when using JNDI as a naming service. [8] calls this pattern Service Locator, "to abstract all JNDI usage, ... EJB home object lookup, and EJB object re-creation …".

It is an unnecessary source of complexity if an avoidable layer of patterns is recommended on top of a platform to make it more usable.

In the security framework of the EJB model, we identified three gaps in the standardization: before EJB 2.0, client authentication was not standardized and, even in EJB 2.0, there is no standard way for the server to verify a client's authentication credentials. The attachment of the EJB authorization to the underlying security infrastructure is also not standardized.
Only the EJB 2.0 specification defined JAAS (Java Authentication and Authorization Service, [3]) as the client authentication mechanism. JAAS delegates the actual authentication to Pluggable Authentication Modules (PAMs) and, therefore, supports any authentication infrastructure. Unfortunately, this only happens in the client, from where the identity of the authenticated principal is then transferred to the server. If an attacker can tamper with the code in the client, he/she can pretend to be another user, because there is no way of verifying client authentication credentials on the server (i.e., the attachment of the authentication infrastructure in the server is not standardized). Moving the PAMs to the EJB container and performing the JAAS authentication there fails also because the PAMs potentially call back to receive user authentication information, which would require threading support in the EJB container. A final limitation of the EJB security framework is the lack of standardization for the attachment to the authorization infrastructure in case the default static authorizations are not sufficient.

2.4. J2EE evolution and implementation incompatibilities

There are two natural limitations of a complex enterprise platform such as the J2EE, where the same specification is implemented by different software vendors: (1) the platform will be in an emerging state for some time, and (2) platform vendors try to differentiate themselves through proprietary extensions.
These limitations cause a problem for those writing enterprise applications: at some point, one needs to commit to both a platform version and a platform implementation. From then on, platform evolutions usually require adaptations to the application, which often depends on particular implementation features. To make things worse, adoption of a new platform version is

often an all-or-nothing proposition: there is little room for incremental adaptations and gradual deployment. Platform dependencies also make cross-project reuse and know-how transfer more difficult, which can be a major challenge for IT services companies serving many customers or large organizations with extensive heterogeneous IT infrastructure.
Platform changes that lead to incompatibilities are documented in [16] and the EJB 2.0 specification has to lists for example two different deployment descriptor definitions for the EJB versions 1.1 and 2.0. Functionality that completes the J2EE platform is, for example, the message-driven bean support (introduced in EJB 2.0), timers for EJB (upcoming EJB 2.1 [11]), management of J2EE (only specified in JSR-77 [14]), or standard deployment for J2EE applications (specified in JSR-88 [12]).

J2EE implementations vary due to missing standardization of important features (e.g., user login (defined only in EJB 2.0), deployment descriptors (only standardized in EJB 1.1, extended in EJB 2.0), or the management of the J2EE) and due to J2EE implementation vendor differentiation. While the situation with respect to the missing standardization is improving, vendors increasingly differentiate their products.
For example, [16] discusses many differences between successive EJB versions, in addition, the specification of JNDI names in deployment descriptors and the build and deployment processes are not standardized. Moreover, most J2EE implementations extend their containers with proprietary security services or management support.

3. The LEAF enhancements

LEAF enhances J2EE implementations. It resolves the issues raised in the previous section, thereby increasing the flexibility of the J2EE, and enriching it in several other respects.
LEAF augments the J2EE with basically 3 elements: (1) a thin abstraction layer on top of the J2EE ORB, (2) a new component type and its runtime environment, and (3) technical services that either enhance existing J2EE services or provide complementary functionalities. Figure 2 gives an overview of LEAF.

3.1. The LEAF layer

We wrap the EJB ORB with a thin layer, written as an additional, LEAF-generated stub that delegates calls to the underlying EJB stub (Figure 3). This layer provides the following functionality:

- it passes control to the LEAF invokers,
- it adds implicit caller context to remote invocations, and
- it implements service location transparency.

LEAF invokers are pieces of code that can modify the behavior of method invocations [26]. Whenever a method is invoked, the LEAF ORB first passes the control to an invoker. The invoker then either invokes the method, performs pre- or post-processing for the invocation, or aborts the invocation altogether. Invokers make the ORB more flexible and help keep it lean by relieving the ORB of all but its core functionality. All other functionality is provided (if required) by invokers.
We use invokers for security, logging, runtime exception logging, performance and usage statistics, audits, load balancing, and fault-tolerance. To combine the functionality of multiple invokers, they are *chained.* There is a global default invoker chain, which can be adapted for each LEAF service. Invokers can be installed both in the stub (on the side of the caller) and in the skeleton (on the side of the implemented service). The overhead of each invoker is small, as it is only a local method call, and we often only set them up for façade beans.

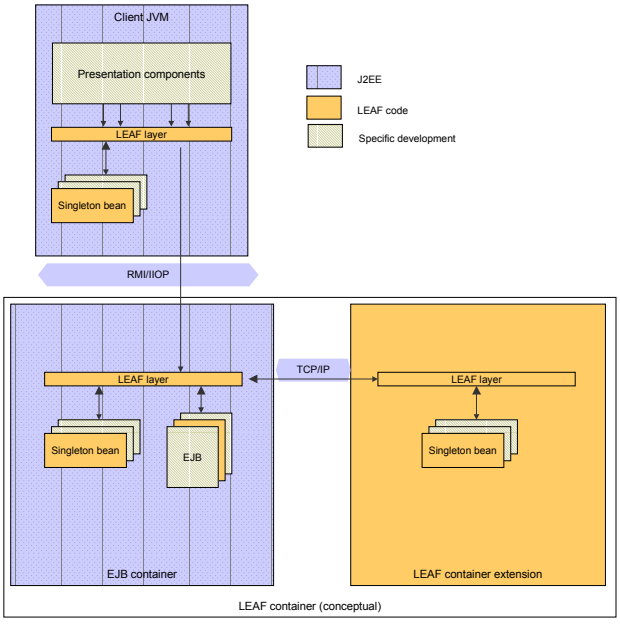

Figure 2: LEAF overview: A LEAF client with the LEAF extended container. LEAF occupies each JVM in the form of the LEAF layer. The additional LEAF services are not shown here because they are provided like normal EJBs and singleton beans in the same way an application would write its own beans.

Many RPC infrastructures allow passing *implicit context* with invocations, i.e., data that is not part of the explicit formal parameters of methods but passed nevertheless

with invocations. The advantage of such implicitly passed context is that this information becomes optional: e.g., a principal is only passed once the user is authenticated, or a transactional context is only passed when a transaction is active. We use implicitly passed context to exchange data between invokers. For example, the performance statistics invoker adds a unique identifier to an invocation to track graphs of invocations. LEAF allows adding information to the implicitly passed context of the ORB, extending the EJB model's basic implicit context passing. The implementation of the implicit context passing can either be made by using, e.g., the underlying IIOP facilities for context passing or by adding a *context* argument (hidden in the LEAF layer) to each method signature. In the current version, we pass the implicit con[illegible]onal method parameter.

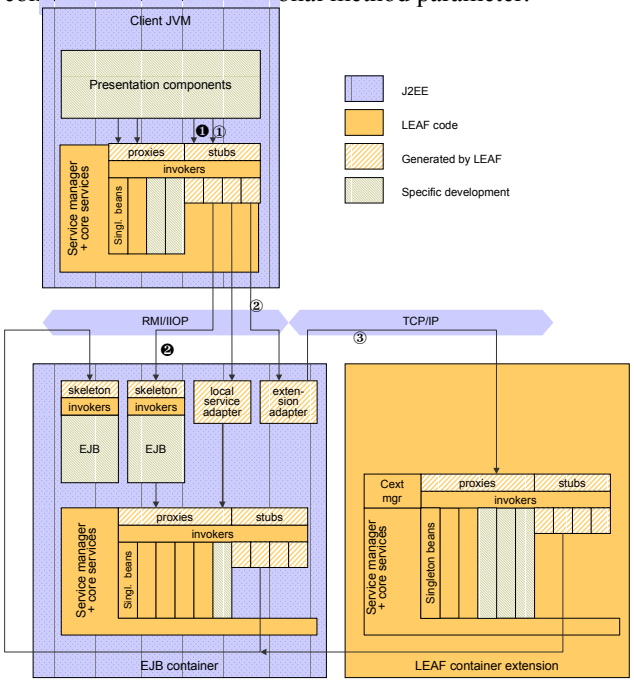

Figure 3: The internal structure of LEAF. The picture shows three JVMs: a client, an EJB container and the container extension. In each JVM there is a service manager that activates and supervises the services. Examples: ❶ and ❷ show an invocation of a bean running in the EJB container; ①,② and ③ show an invocation of a singleton bean running in the container extension.

In LEAF, there are no remote interfaces as in the J2EE. We avoid them through three means:

- We replace the checked `RemoteException` in remote interfaces with an unchecked `RemoteRTException` that is only thrown by service implementations running remotely. It does not need to be caught in the block it occurs.
- We provide guidelines on service interface design for services that are planned to be used both locally and remotely. For example, we discourage modifying service parameters in the service implementation.
- We provide metadata about services in order to know at runtime where a service runs.

Hence, we can move the implementation of a service more flexibly between containers, and map service implementations lately to their execution environment. We do not need to treat the `RemoteExceptions` immediately where they occur, and we can have the same interfaces for services that may be implemented locally or remotely. Note that we do not claim that service interfaces and their uses never change according to whether they are run locally or remotely; but we prefer an infrastructure that does not force them to look different.

Figure 3 shows the internal structure of the LEAF layer: invocations first go through the stubs (proxies for local services), then through the invoker chain, and are then forwarded via the default invocation mechanism to their destination. In the EJB container, invocations arrive in the LEAF skeleton, and pass the invoker chain again before they arrive at their EJB bean. An example illustrates a client call to an EJB bean: the client invokes the LEAF stub ❶. The stub then forwards the call to the client-side invoker chain. The last invoker on the invoker chain forwards the call to the stub of the EJB container, which itself forwards the call (usually via RMI/IIOP) to the EJB container ❷. There, the LEAF skeleton intercepts the call, and forwards it to the server-side invoker chain, whose last element then finally invokes the EJB bean. Invocations on singleton beans running in the container extension start similar up to the EJB container (①②). In the EJB container, the extension adapter is used to forward the call to the container extension ③, where a local invoker chain is consulted and the invocation is forwarded to the implementation of the singleton bean.

The LEAF layer does not compromise the interoperability between pure EJB beans and beans that profit from the LEAF layer. Calling an EJB from LEAF is not a problem, as all it requires is pure Java code. Calling a LEAF bean from EJB is also trivial: all the EJB bean needs to do is to get the stub to work on the LEAF bean from the LEAF naming service.

Sometimes people unfamiliar with LEAF think at first that the LEAF layer makes applications non-J2EE compliant. However, the LEAF layer is only a thin and *syntactic* wrapper. The *semantics*, i.e., the component model and semantics of EJB interactions, are *unchanged* by the LEAF layer. The LEAF layer can even be applied to *existing EJB applications*, allowing developing an application without the LEAF layer first and then adding the layer only when needed!

3.2. Singleton beans and their runtime environment

LEAF adds one new component type to the J2EE: the (non-EJB) *singleton bean*. A singleton bean exists either as a *global* singleton within a LEAF application or as a *local* singleton in one or several JVMs. It is similar to EJB beans: it can be deployed independently, it can expose methods that are invokable remotely, and it indicates other parts it depends upon. Its container is responsible for its activation and re-launch after a crash (of either the service itself or of the JVM it runs in).
Singleton beans support the three missing component types identified in the previous section: Daemons and global singleton services may be directly implemented as LEAF singleton beans, and a particular singleton bean, the batch service, can host batch jobs.

There are two runtime environments for singleton beans: they can either run in the JVM of their user, provided that the environment does not forbid any of its functionality (e.g., the limitations of the EJB container), or they can be hosted in a separate container, called the *LEAF container extension*. Conceptually, the EJB container and the LEAF container extension together form an *extended container*, in which singleton beans run alongside traditional beans (Figure 2).
Both runtime environments for singleton beans provide independent deployment, activation, and access to configuration. In addition, the container extension provides fault-tolerance, remote access, and support for load balancing and partitioning of singleton beans.

We do not use an existing ORB as base for the container extension. The reason for this decision was that we found no lightweight fault-tolerant ORB available that could satisfy our flexibility requirements, e.g., for implicit context passing or invokers, and for other implementations of the container extension (e.g., to be able to put the container extension as BEA Weblogic startup class [22] within the JVM of the EJB container).
We chose to implement the container extension as a federated set of JVMs that supervise each other. On each machine, there is one *node* (a JVM), with only a minimal installation of LEAF, so as to be as reliable as possible. The node supervises the nodes on other machines. The singleton beans run in *sites* (separate JVMs), which are launched and then supervised by the *node*s. The implementation of the inter-JVM communication, the fault-detection, the node/site topology, and the fault-tolerance have been realized as layered LEAF services. In case a node detects a site failure, it coordinates with the other nodes to launch a new instance of the site with the singleton beans that were running at the site. In case a node (i.e., we assume its machine) fails, the remaining nodes use a consensus algorithm to decide where the failed sites will be recreated. It is the responsibility of each singleton bean to keep its important state persistent and to restore it after failovers.

Not surprisingly, singleton beans are not automatically as scalable as regular EJB beans (as the latter make the assumption that they do not run as singletons). We use different strategies to deal with this problem: our guidelines propose avoiding unnecessary singleton beans or structuring them so that they avoid becoming bottlenecks and that some singleton beans (e.g., the batch service) use the container extension's infrastructure to distribute load themselves. Singleton beans can be assigned to run on a particular machine, and we are adding support for *partitioned* singleton beans: a partitioned singleton bean can run on multiple machines, and each one serves a subset of all requests in function of the request's arguments.

3.3. Technical LEAF services

We discuss here how LEAF services are implemented, and we present a subset of the technical LEAF services (see [6] for a more complete list of services): the naming, the configuration and the security services, which we use instead of their normal J2EE counterparts (see Section 2.3).
LEAF services are implemented in different ways: as singleton beans (running either JVM-local or in the container extension) or as EJB beans. It is also possible to access services that run in other service infrastructures (e.g., SOAP, CORBA [19] or COM services). The use of a service is independent from the service's implementation: the configuration specifies where a service is located, and then the service stub may be retrieved from the LEAF naming service.
The LEAF naming service wraps the JNDI and, possibly, other naming services: it is the factory for LEAF service stubs. It uses configuration information to determine how to construct the naming service's access context, where a service is implemented, and how service stubs are retrieved.
The LEAF configuration service *replaces* the (weak) configuration service role of the JNDI. It provides a functionality that is similar to that of the recent Java 1.4 preference package, including support for storing configuration information in files, storing it in databases, or retrieving it from the command line, and a powerful mechanism for determining default values. It also provides support for configuration information that can vary according to different contexts, e.g., to different

users, locales, or the architecture of the underlying machine.

We split the LEAF security service into two parts, both based on the JAAS model: the authentication service and the authorization service. The authentication is delegated to LEAF-PAM plugins; but contrary to the J2EE *within* the EJB container, avoiding the security problem when one can tamper with the client's code. This solution also works for pre-EJB 2.0 J2EE implementations, where the J2EE client authentication was not standardized. The LEAF PAM's interface departs from the JAAS PAM's (the LoginModule) so that the PAM does not have to issue callbacks to the authentication service. Instead, the authentication service repetitively invokes the PAM until the authentication has failed or the user is authenticated. The PAM returns an array of JAAS callbacks to be populated by the client, or returns either a security token or an exception to indicate that the authentication procedure has succeeded or failed. After a successful login, the security token is stored, together with the JAAS login subject, in order to verify the access rights at each subsequent invocation and to run the server-side code under the identity of the authenticated principal. Storing all valid tokens in the server unfortunately makes the container stateful; however, implementing the system without sharing this secret between the client and the container is impractical for performance reasons.

3.4. Resolving the compatibility issues

An important goal for us was to be able to leverage leading-edge features and technologies while protecting our applications from the instability of emerging implementations and specifications. Several LEAF features help solve such incompatibility issues. Basically, we must either abstract the functionality of the J2EE or provide missing parts of the J2EE where most J2EE implementations provide their proprietary extensions. LEAF is clearly proprietary as well, but it is lean and accessible for projects using it. By providing our own abstractions for functionality like security or management, we avoid LEAF components becoming dependent on particular J2EE extensions.

On several occasions, we also provided our own abstractions for emerging J2EE features. Our preliminary support for message-driven beans (MDBs) can serve here as an example (MDBs were only introduced in the latest EJB 2.0 specification, which is part of the J2EE 1.3). We let MDBs run as singleton beans in the container extension with only a trivial wrapper to make them look like MDBs (at first without load balancing). We can then easily move them to the EJB container's "MDB compartment" when it becomes available or at any time thereafter. Other features we adopted in this way are the EJB timers (planned for EJB 2.1 [11]), JAAS authentication integration (part of EJB 2.0), or standard application deployment for J2EE [12].

3.5. Modularity and infrastructure support for component reuse

We wanted LEAF to remain as *lean* as its name suggests. Extensibility is not enough: it must also be trivial to remove features as they become unnecessary (e.g., because they make it into the standard J2EE feature set). As the many ideas and demands for extensions [6] threatened its leanness, we built the LEAF core to be extensible and introduced a module concept into the framework's build system. The extensibility facilities of the LEAF core are based on enabling developers to replace any LEAF service implementation through configuration and on invokers (as already discussed in Section 3.1). A LEAF module is a collection of code with default configuration information that explicitly states its dependencies on other modules. Only two modules are mandatory, the `leaftools` (low-level coding support) and the `leafcore` modules. Other modules provide additional functionality, such as the container extension, the batch service, the security features, and the SOAP and CORBA interoperability.

To activate a module, one only needs to place it in the Java execution CLASSPATH, and its default configuration will be merged with the configuration of the rest of LEAF. Because each of the new services and invokers present in the module are listed in its configuration, they are automatically available for use by an application. As within the EJB container one should not tamper with the Java execution CLASSPATH, we merge the required modules into the EJB .jar file of the application.

In [7], a list of conflicting assumptions that hinder interoperability of components is presented. The LEAF framework enforces architectural patterns and decisions about the infrastructure that therefore also simplify inter-project component reuse significantly further than the J2EE alone. For example, the singleton bean infrastructure imposes the singleton bean's activation, RPC mechanism, the supervision and management, the placement of service implementations, and the use of the configuration information on the developer. This permits singleton beans from different projects to cohabit and interoperate among themselves more easily.

4. Implementation

The first implementation of LEAF used in production consisted of 40'000 lines of source code (including tests and demos, but without comments). However, the extensible core of LEAF requires only 15'000 lines of code, which indicates the leanness of the framework. Due to the fact that there are many extension ideas and requests, LEAF is still undergoing active development, even though it has already been used successfully. The current full release consists of 70'000 lines of source code. Once the extensible core was finished, most other extensions were independent of each other, which permitted us to efficiently split up the work on further extensions.

5. Experience report

Our experience with LEAF has been extremely positive. All costs and overhead of using and developing the LEAF framework and components have been largely offset by its benefits, and LEAF has been eagerly adopted by our architects and developers.
In this section, we first report on field experiences of using LEAF in commercial projects, present the simplicity LEAF brings to a project, and then compare using the J2EE with LEAF and without.

5.1. LEAF benefits in concrete commercial projects

In this section, we will describe the adoption of LEAF in two concrete commercial projects: an e-business server and an enterprise application integration (EAI) platform.
The former project integrates a range of new J2EE-based e-commerce services and host-based back-end functionality, such as product information, customer information, and routing and pricing algorithms. These services are made available via a number of different channels: a Swing GUI channel for the internal administration and towards the Internet via email, SMS, and a content management system (CMS) channels. The system requires high availability (24x7) and supports up to 60'000 requests per hour. The application was completed within 8 months and represents an effort of more than 15 person-years.
The goal of the second project was to design and implement a standardized interface layer between new business applications written for the J2EE and legacy data and transaction on an OS/390 host and DB/2 databases. The integration layer also had to offer a set of technical services (90% of which were provided by LEAF in the first iteration), as well as data access services based on the DAO (Data Access Object, [8]) pattern.
Both applications used the LEAF core and many additional modules, including the container extension, the batch service, and the security module. The code of LEAF is considered part of the application.
In these two projects, LEAF proved invaluable, as illustrated by the following examples:

- The EJB container selected by one of the customers did not yet fully support the EJB 1.1 specification, was resource-intensive, and imposed time-consuming build-deploy-run cycles (up to 1.5 hours in our case). LEAF allowed us to develop the application with the lightweight JBoss J2EE implementation (where a development cycle was only 20 minutes), and deploy it, unchanged, on the integration and production platforms. Each developer could run its own (free) copy of JBoss on his machine, which had the additional benefit of reducing development license costs. Even taking additional non-regression and integration testing into account, the time and productivity savings were considerable, since most developers could focus on business code, while technical problems were handled by a small, specialized team.
- Due to a limitation of the CMS system used in the e-commerce application, two clicks on the submit button of a form would initiate two EJB requests. A simple invoker fixed this problem for all critical method calls.
- Because the CMS system did not support server affinity for user sessions, we set up a second invoker that would load user state before each request by using a token ID stored in the attributes of the Web request which was then passed via implicit context passing (the state was stored in a read-cached singleton bean).
- A notification invoker was used to log unexpected `RuntimeExceptions` that occurred in the EJB container. Another invoker passed more explicit exception information (stack trace and original exception, potentially as string when the exception class was absent on the client side) back to the client after a `RuntimeException`.
- A release-switching invoker enabled atomic switching between different versions of an application.
- To tune the performance of the application, we used LEAF's location transparency to move service implementations between the EJB container and the container extension in order to avoid unnecessary inter-process communication.

5.2. Gains in programmer efficiency

LEAF simplified the work of architects and developers in several respects: (1) by providing crucial additional architectural abstractions, (2) through direct support for recommended architectural and design patterns and best practices, and (3) by handling a number of the technical details related to J2EE for the developer.
One crucial architectural abstraction LEAF adds is the *singleton bean* abstraction. It is so valuable for us that we use it in practically every J2EE enterprise application. The abstraction is backed by infrastructure support for running and supervisioning singleton beans. When designing an application, this abstraction significantly simplifies the architect's job. Other useful abstractions are batch jobs or invokers. Best practices are often collected in the form of patterns. LEAF eliminates the need for many patterns for EJB development, such as the Service Locator, Service Activator, and the Business Delegate Pattern [8]. LEAF hides many elements of the J2EE, including, Home Interfaces, Deployment Descriptors, the complicated method of locating an EJB stub (see Listing 1), and details like the JNDI API distinction between the `javax.naming.Name` interface and the name as a `String`. The different service types of LEAF can all be located and used in the same way, which is a particularly valuable benefit when client code is also prepared for remote service types (i.e., catches RemoteRTException).

5.3. Evaluating LEAF for general J2EE applications

In the following section, we will discuss the tradeoffs of using the J2EE both with and without LEAF. Licensing and training make up most of the costs of using LEAF. Because we have deliberately adopted a tool-neutral approach, LEAF is not tightly integrated in the current IDEs. While this decision may somewhat reduce comfort and speed in the coding phase, it turns out to be a distinct advantage in integration and production phases, where the deployment process is not bound to a particular development tool. It also facilitates reuse across projects and environments and allows each developer to use his/her preferred IDE, an additional advantage in multi-cultural teams. The performance overhead of the LEAF layer has been negligible in the projects we have carried out to date.

Of LEAF's many benefits, we have already discussed the improved J2EE implementation compatibility and programmer efficiency gain of LEAF in Sections 3.4 and 9. The availability of reusable technical components saves development time. We showed in Section 5.1 how the flexibility of LEAF helped to easily resolve many issues in real-world projects. We believe that the benefits of invokers can be generalized to the benefit when one has access to the source code of a component: very often access to the source code enables problem resolutions or extending functionality. It is also one reason for the increasing popularity of open source software in commercial contexts [27].
The flexible exchange of a service implementation between sites and the role of LEAF as a repository for reusable components are also useful. Through the enforcement of best practices, reuse of proven infrastructure components, and the flexibility gained due to low-level instrumentation capabilities, LEAF also improves the quality and stability of applications while reducing project risks.

Clearly, the benefits of LEAF greatly outweigh its liabilities. Some of the listed LEAF benefits are easy to quantify in terms of savings potential. For example, when components that would otherwise need to be developed already exist in LEAF, or when more flexible deployment scenarios are required than possible in the J2EE (requiring different J2EE implementations or service implementations in different execution environments). The effects of other "soft" or more long-term benefits are difficult to quantify, be they improved flexibility or simplicity, better overall quality, risk reduction, inter-project component reuse, or the benefit of LEAF as a collection of expertise.

6. Related work

Currently there are over 20 certified and uncertified J2EE implementations available [13]. As most J2EE implementations extend the basic feature set, they are somehow all related to LEAF. Extensively comparing these different J2EE extensions is beyond the scope of this paper. Therefore we will limit our discussion to some example extensions.
In the following section, we first discuss alternative solutions to each of the three primary LEAF features (the LEAF layer, the additional component type, and the extended services) and then we discuss more general related work.

6.1. Primary LEAF features

LEAF layer. Aspect/J [24] could be used to provide some functionality of the LEAF layer, such as the interceptor functionality or the replacement of the RemoteException via a "Softened Exception" (both are only possible at compile time). However, Aspect/J cannot

generate a LEAF stub wrapping around the original EJB stub. It can only modify the code of the generated stubs. Alternatively, one could *weave* the whole EJB container with this aspect, but changing the container in such a way is usually impossible due to licensing agreements. Aspect/J currently provides no support for the "aspect" of context passing between remote JVMs.
LEAF uses the service abstraction layer pattern of [18]. Composition filters introduced in [9] are similar to our invoker chains. JBoss [21] has server-side interceptor support similar to the invokers (actually, the whole EJB container is built as a collection of interceptors), but this solution only works with the JBoss J2EE implementation.

Additional component type. It is possible to use an external ORB to host additional component types. The disadvantages of this solution were already discussed in Section 2.2.
BEA's Weblogic startup classes allow Java classes to run outside of the EJB container, but within the JVM of the container. A drawback of this kind of solution is that these startup classes lack the quality of service support we provide; however, it may be interesting to adapt our container extension to run as a Weblogic startup class.
The emerging version of JBoss (3.0) includes a light fault-tolerant ORB based on RMI that could be used to host singleton beans in a manner similar to how our container extension does.

LEAF services. As mentioned in the previous section, common J2EE patterns [8] provide functionality similar to many of our enhanced J2EE services, such as our Service Locator and Business Delegate. The disadvantage of using them is that they need to be coded again for each situation, while LEAF's abstractions are available automatically.
JBoss provides an EJB-JAAS integration [26]. The BEA Weblogic [22] server supports proprietary extensions for the attachment of an authorization infrastructure to the EJB container, which solves parts of the issues LEAF has with the J2EE, but is still not complete and proprietary.

6.2. General J2EE enhancements

In this section we discuss (1) J2EE Integrated Development Environments (IDEs), (2) J2EE implementations (examples only), (3) patterns, and (4) other J2EE extension frameworks. We exclude other enterprise platforms, because comparing them with LEAF or the J2EE goes beyond the scope of this paper. J2EE IDEs often simplify the J2EE similar to how LEAF does by hiding certain details (e.g., home interfaces, deployment descriptors, and proprietary build and deployment processes). They can therefore provide some J2EE implementation isolation, but their features are less rich when compared to LEAF. Most J2EE implementations have interesting extensions that often complement the J2EE. For example, BEA's Weblogic Server, IBM's WebSphere [23], and the open-source implementation JBoss all extend the J2EE in non-standard ways with improved management support, a more flexible security infrastructure, and Web-services support. However, given our requirement not to depend on J2EE implementation vendors, we could not choose this proprietary route. J2EE design patterns are a viable way to solve design problems, and LEAF proposes and uses many patterns. However, we believe that it is burdensome to request each developer to re-implement patterns simply to compensate for the unnecessary platform complexity.
Giving a complete overview of other J2EE extension frameworks is difficult because of lack of information from vendors. There are different types: lightweight frameworks, collections of business-generic components, and frameworks that change the EJB component model.
[25] presents a small Java framework to wrap around some of the complexities of the J2EE. Similarly, [16] shows how to isolate an application from the differences of EJB versions. AbaXX [20] is an example of a collection of business-generic EJB components. Extension frameworks such as JWelder [17] extend the EJB component model.

7. Conclusion

This paper gives a practitioner's opinion of the J2EE and proposes some improvements. We then discussed our experiences, which have been very positive compel us to continue using LEAF. In short, we believe that we can draw two conclusions from our experience with LEAF:

- The additional component type, support for invokers and implicit context passing, and J2EE implementation compatibility improvements have proven very useful. We think that the J2EE specification would greatly benefit from these features. It is also interesting to note that Microsoft's emerging .NET platform [24] already contains some of these features, such as components running as singletons with different semantics, invokers for remote service invocations, and service location transparency.
- We believe that LEAF has proven worthwhile the approach of leanly wrapping an emerging platform like J2EE. The interceptor and implicit context passing simplified the resolution of many problems and ownership of the source code of the wrapper provides much flexibility that is useful when solving problems. The isolation from changes in the J2EE specification and J2EE implementations allowed the

creation of reusable J2EE components and simplified inter-project component and know-how reuse.

LEAF will have to evolve in the future. Some of its features will become part of the J2EE, and we will be able to discontinue them in LEAF. We will also add business-generic functionality (such as simple workflow support or support for user management) that is often required in applications. We are currently extending the scope of the platform to support mobile devices. Finally, we are already beginning to bring the principles and concepts of LEAF onto the .NET platform.

8. Acknowledgements

We would like to thank the LEAF core developers Alain Borlet-Hote, Eric Castan, Olivier Cathala, Yves Martin and Paul Ersin Sevinç as well as the other LEAF contributors: Thomas Andrieu, Daniel Balmer, Laurent Bovet, Andrea Branca, Daniel Girardeau-Montaut, Felix Jäger, Stéphane Kay, Sylvain Laurent, Viet Anh Nguyen, Vincent Niederhauser, Jean-Baptiste Ranson, Silvan Saxer, Christoph Schwitter, Martin Sijka, and Bernhard Seybold. In addition we are grateful to the LEAF pilot projects, in particular to Vincent Messerli, the LEAF forerunner projects, in particular, Hans Burger and Bernhard Rytz, and the paper reviewers.

9. References

[1] B. Shannon, *Java™ 2 Platform Enterprise Edition Specification*, v1.3 and v1.2, Sun Microsystems, http://java.sun.com/j2ee.

[2] *Enterprise JavaBeans specifications*, Versions 1.0, 1.1, and 2.0, http://java.sun.com/products/ejb/docs.html.

[3] JAAS web site, http://java.sun.com/products/jaas/.

[4] E. Roman, et al., *Mastering EJB*, 2nd Edition, John Wiley & Sons, 2002.

[5] J. Waldo, S. C. Kendall, A. Wollrath and G. Wyant, "A Note on Distributed Computing", Sun Microsystem Research, http://research.sun.com/techrep/1994/abstract-29.html.

[6] F. Buschmann, et al., *Patterns of Sofware Architecture*, Wiley, 1996.

[7] D. Garlan, et al., "Architectural Mismatch or Why it's hard to build systems out of existing parts", *Proc. 7th International Conference on Software Engineering*, Seattle WA, April 1995.

[8] D. Allur, et al., *core J2EE patterns*, Java 2 Platform, Enterprise Edition Series, Sun Microsystem Press (Prentice Hall), 2001.

[9] M. Aksit et al., "Abstracting Object Interactions Using Composition Filters", *Proc. of the ECOOP'93 Workshop on Object-Based Distributed Programming*, Springer-Verlag, 1994.

[10] Wiki Web, "What is wrong with EJB?", http://c2.com/cgi/wiki?WhatsWrongWithEjb.

[11] Enterprise JavaBeans v2.1, JSR-153, http://jcp.org/jsr/detail/153.jsp.

[12] J2EE Application Deployment, JSR-88, http://jcp.org/jsr/detail/88.jsp.

[13] EJB server directory, http://www.mgm-edv.de/ejbsig/ejbservers.html.

[14] J2EE Management, JSR-77, http://jcp.org/jsr/detail/77.jsp.

[15] H. Sheil, "Frameworks save the day", Javaworld September 2000, http://www.javaworld.com/javaworld/jw-09-2000/jw-0929-ejbframe.html.

[16] R. Monson-Haefel, "Create forward-compatible beans in EJB", Parts I and II, Javaworld 12/99 and 1/00, http://www.javaworld.com/javaworld/jw-01-2000/jw-01-ssj-ejb2.html.

[17] Th. Neumann, U. Schreier, M. Fabini, "Auf dem Weg von EJB zu Fachkomponenten", OBJEKTspektrum, 1/2001.

[18] O. Vogel, "Designing a Three-Tier Architecture Pattern Language", *EuroPLoP 2001 Design Fest*, http://www.cs.wustl.edu/~mk1/ThreeTierPatterns/submissions/OliverVogel.pdf.

[19] CORBA web site, http://www.omg.org.

[20] AbaXX web site, http://www.abaxx.com/.

[21] JBoss web site, http://www.jboss.org.

[22] Weblogic web site, http://www.bea.com/products/weblogic/server/index.shtml.

[23] Websphere web site, http://www-3.ibm.com/software/info1/websphere/index.jsp?tab=highlights.

[24] Aspect/J web site, www.aspectj.org.

[25] J. P. Choi, "Aspect-Oriented Programming with Enterprise JavaBeans", *Proceedings of the 4th International Enterprise Distributed Object Computing Conference*, Makuhari, Japan, 2000.

[26] L. Taylor, "Customized EJB security in Jboss", http://www.javaworld.com/javaworld/jw-02-2002/jw-0215-ejbsecurity.html.

[27] T. Yager, "Open source takes hold", InfoWorld, 2001, http://www.infoworld.com/articles/tc/xml/01/08/27/010827tcintro.xml.

[28] .NET platform, http://www.microsoft.com/net/.

[29] C. Gasser, LEAF Datasheet, http://www.elca.ch .