A Dynamic Service-Oriented Implementation for Java EE Servers

Mikael Desertot[1,2], Didier Donsez[1] and Philippe Lalanda[1]

[1]*LSR-IMAG, 220 rue de la Chimie*
Domaine Universitaire, BP 53
F-38041 Grenoble, Cedex 9, France
firstname.surname@imag.fr

[2]*Bull SAS*
1, rue de Provence - BP 208
F-38432 Echirolles Cedex - France
mikael.desertot@imag.fr

Abstract

This paper proposes to enhance the dynamism and the flexibility of Java Enterprise Edition (EE) servers by introducing a Service-Oriented Architecture (SOA) inside. The purpose is to ease the deployment and offer dynamic server configuration and reconfiguration. Such an approach limits consumed resources and is capable of context adaptation. After defining the properties that must be verified for the service platform, we propose to use OSGi technology as the basis for the architecture. We have experimented with integrating OSGi into Java EE servers. Moreover, this architecture has been chosen for the next generation of JOnAS ObjectWeb's open source Java EE implementation.

1. Introduction

Nowadays, since the introduction of component models in mid-nineties, most components developed in the industry are those deployable on dedicated application servers. The market is divided between the two main leaders, Java Enterprise Edition (formerly J2EE) in the Java world and .NET in the Microsoft world. .NET offers a single industry-level implementation, provided by Microsoft, and is still an emerging technology.

On the other hand, Java EE is currently the most used industrial application server. It is specified by Sun. There are many implementations available, being commercial (IBM, Sun, HP, Oracle…) or open-source (ObjectWeb's JOnAS, JBoss' Application Server and the recent Apache's Geronimo). It offers an environment to deploy and execute applications and ensures that the specified technical services are available. The usage of those services is hidden to the developer by containers that manage application logic components and their interaction with technical services.

Server specifications (with some additional Java Specification Requests) aim to simplify application development and deployment. But currently, only the Java Business Integration (JBI) specification (JSR 208) tackles Java EE architecture (as well as J2SE). It standardizes a way of assembling and binding the components making up an application. As of now, none of the existing open source servers are JBI compliant, it is up to each implementer to provide the functionalities he needs for the assembly of services. This is why we can find disparate server capabilities depending on the choices that have been made by the servers' providers. The most useful, and also the trickiest, implementation-dependent capability is the introduction of dynamism for services [1], which is currently limited in Java EE implementations.

The objective of our work is to tackle the Java EE services layer limitations. In this paper we argue that we can benefit from using a Service-Oriented Architecture (SOA) as a basis for Java EE services dynamism. First of all, it brings the flexible architecture that Java EE servers are lacking. Indeed, a particularity of service architectures is that service implementations can be registered or unregistered at runtime. This is due to the loosely-coupled connections that exist between each component when building applications. Second, it can also help interaction with third-party applications also offering services because of the abstraction level offered by service contracts.

This paper proposes an architecture for a dynamic service-oriented backbone for Java EE. This work is realized in collaboration with the JOnAS project leader, Bull SAS. Section 2 focuses on Java EE, more precisely on its services layer, its limits and existing solutions to compensate for them. Section 3 explains our approach. Section 4 gives a description of the new architecture and the choices that have been made concerning the services platforms. Section 5 illustrates an implementation of a dynamic Java EE server. In section 6 we describe a typical use case for such a server. Finally, in section 7 we run through experiments that are currently performed around Java EE.

2. Java EE Application Servers

This section presents the general Java EE architecture with its advantages and limits. It describes existing solutions for introducing dynamism into the Java EE services layer. We rely on these principles to introduce our approach for enhancing Java EE architectures.

2.1. Java EE

Applications deployed on Java EE application servers are assemblies of components dealing with user presentation or business logic. Those two layers (presentation and logic) are composed of JSPs and Servlets for the first one and of different kinds of components (i.e., EJB for Enterprise Java Beans) for the second one (figure 1). An EJB can be:

- a Session Bean implementing application logic.
- an Entity Bean to simplify access to persistent data contained in databases.
- a Message Driven Bean that reifies a message queue polling (like JMS queues or topics).

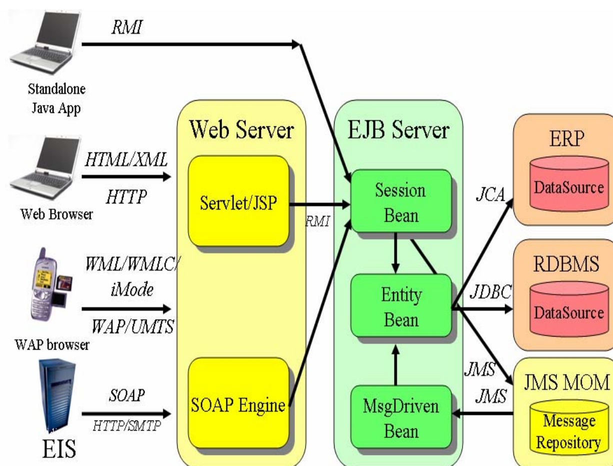

Figure 1. The Java EE environment

Application deployment, undeployment and update can be performed during the server runtime. This capability is addressed by a Java Specification Request (JSR 88) and is an absolute must in every Java EE implementation.
At the lowest levels of the server, the Java EE platform manages all the technical services specified by Java EE (figure 2). Those technical services are orthogonal aspects with regard to the application logic. These services are required in most enterprise applications. They are offered to release the developer (or assembler) from these concerns. If applications have dependencies on some services, services can also have dependencies to other services. For instance the Servlet (or JSP) engine has a dependency on the Security service for providing secure web pages. Among the services offered we can mention transactional, persistency or mail services.

Those services are very specific and complex. They can only be implemented by domain experts. It means that providing a Java EE server requires the integration of components dealing with very different concepts. Such components are most of the time large and complicated projects on their own, and their implementation is independent from Java EE servers.

To access services, application component implementations are placed into containers. Those containers are used as the glue between components and technical services. But, contrarily to the application level, nothing is specified concerning the service level behaviour. In fact, the specification assumes that the services are always available as soon as the server is started.

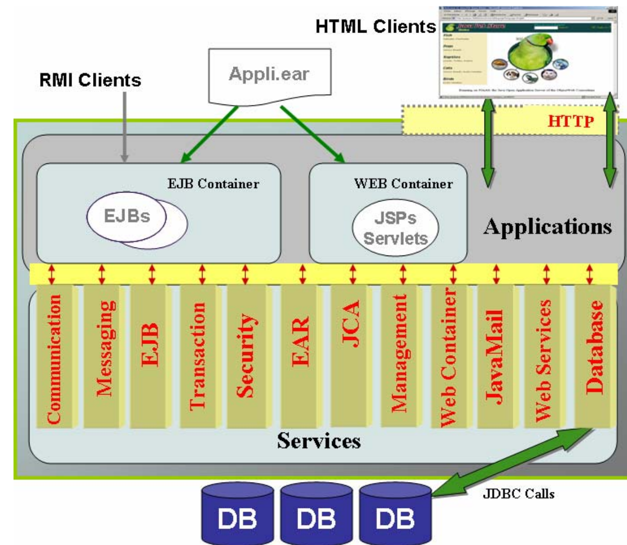

Figure 2. Java EE classical architecture

This drives each server implementer to offer his own solution for introducing new service properties, e.g., dynamism. This also allows each server implementer to distinguish his implementation from other ones. It leads to many different implementations, often proprietary and incompatible.

2.2. Current Solutions

In this section we present the choices that have been made in different Java EE implementations for providing a dynamic layer for non-functional services. We studied in particular the three open-source implementations available today, that are JBoss Application Server (AS) [2] [3], Geronimo [4] and JOnAS (http://jonas.objectweb.org).

JBoss AS considers Java EE containers meta-level components (MBeans) whereas EJB are base-level components. This container relies on the JMX (Java Management eXtensions) dynamic class loading capabilities for offering services dynamism; a JMX server is currently always deployed with a Java EE (or a third party JMX server is accessible).

It offers life-cycle activities on services that include creation, start, stop and destruction as well as a way of deploying those services. In order to do this, a service descriptor is added in the deployment unit. When deploying a service, this description is used to also install

and activate (start) all the required services. The base internal class loader architecture (called UnifiedClassLoader) is flat, each one being at the same level as the others. Once loaded, the services are bound together with the help of dynamic proxies.

Geronimo models complex systems as components capable of keeping their state, having relationships and reacting to events. Even though it distinguishes between two kinds of components, containers and applications, each component is a GBean that could be in one of three different states: stored, loaded or running. Geronimo proposes a loosely-coupled architecture where service bindings are managed by the framework. The latter uses both inversion of control (IoC) and byte code injection to resolve dependencies. When a GBean is deployed, dependency resolutions are taken into account by the framework by injecting in the bean the needed code for bindings. This injection of one or a set of dependencies is done by getter/setter methods or directly inside the component constructor.

JOnAS does not offer for now any dynamic services capabilities. Moreover, as soon as the server is started it is impossible to install or start new services or even reconfigure deployed services without restarting the whole server.

The first two solutions, JBoss and Geronimo, are already providing a certain degree of dynamic capabilities but still have some limitations. JBoss does not offer truly modular service loading. All the classes of a service are not isolated from the other service classes. It is not possible to guarantee that they will not interfere with something already running and make the system incoherent. JBoss and Geronimo do not offer a real service registry which is a key characteristic of the Service-Oriented Architecture to perform service trading enabling the use of any service implementation that fits the requirements. And they both do not follow any standard. This can impose a penalty because it complicates the reuse of third-party services. Finally even if it is not really the purpose of the paper, none of them address the service deployment issues, assuming that the administrator manages the deployment of service implementations.

3. A Service-Oriented Approach for Java EE Server Architectures

This section describes our service-oriented approach for Java EE servers, our motivations, the way we handle the services and the requirements we introduced.

3.1. Motivations

Considering current Java EE servers capabilities, our goal is to offer a way of using the specified services by relying on a Service-Oriented Architecture for bindings. Java EE architecture can benefit from the interesting features brought by the SOA layer. What we can expect from it can be useful for resource consumption, (re)configuration, simplified server construction and update:

- Resource consumption: an application may not require all of a servers' services. In that case, those which are not used can be stopped to release memory resources. If the server is dedicated to some particular application, non-needed services may not be directly loaded at server activation. SOA can also bring benefit to the environments' dynamic capabilities. A service can be started on demand if a newly deployed application requires it. This property can also be useful if a server provider offers different licensing (and different costs) for its server, depending of the offered services. A client can then upgrade his licensing, and the additional services he needs can be deployed at runtime.
- Configuration and reconfiguration: it is again the services' inherent dynamism that offers a suitable response. It is also possible to perform configuration or reconfiguration at runtime of the deployed services. In the worst case, if a service must be restarted for taking the new configuration into account, the capability of stopping and restarting it at runtime is available; and this without interfering with the rest of the collocated services.
- Server building: it is simplified by the abstraction level that services provide. A service is only defined by a contract and potentially some additional properties. For using it, only this contract has to be known. As previously said, an application server is an assembly of different components coming from different implementers. To simplify this assembly, the service abstraction appears to be a good first step. Indeed, integrating a service within this architecture is much simpler than hard coding the integration directly in the server code. Moreover it eases server maintenance and engineering. But to benefit from this capacity, standard interfaces must be defined for all Java EE services, which is currently not the case.
- Service update: it is the last enhancement brought by SOA. Still with the same idea that the different application servers' service components are managed by different specifications and most of the time (in open source projects) provided by third-party projects, they have very different life cycles compared to the core of the server. Versions are produced more frequently than the server ones and it may be interesting, for debug or performance purposes, to migrate rapidly toward the new version. Because of the contract abstraction, we are able to update only one service and this at runtime. There is no need of providing a whole complete server version and all configurations that have been done on the running server don't have to be executed again.

3.2. Services

In this paper, we argue that we would benefit from treating with a homogeneous life cycle both the services

and application level. As said previously, the application layer already provides dynamic capabilities. We want to offer in the same manner the deployment and activation of both the technical services and applications. If the enhancements listed below could be reached, the management of the server would become easier. To achieve it, our goal is to propose a new dynamic architecture for the Java EE services backbone.

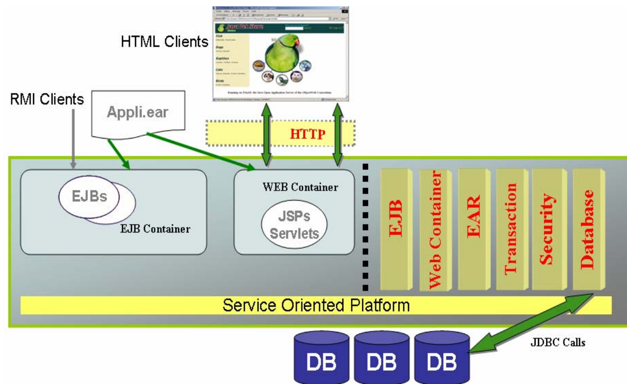

Figure 3. Java EE environment proposition

In figure 3, services are at the same level as applications and benefit from the same capabilities. Both the application layer and the services layer rely on a Service-Oriented Platform that will manage the binding between containers and services. An important point in the architecture is that we do not want to reduce the application layer's capabilities but only increase the service layer's.

3.3. Requirements

The Service Platform we propose has to meet a number of requirements to respond to the needs of the Java EE domain:

- Language: it must accept the Java language for services implementation. The application server we target is Java based.
- Service announcement: we want to ensure that the application we are deploying will run properly, without any missing service dependencies. The environment must offer a way of announcing the arrival or departure of services.
- Service registry: the service registry is centralized, for performance purposes. Binding a set of distributed services to construct a single server instance may be costly and not reliable due to network overhead. Even in cluster cases, the whole server is currently replicated on each node instead of some parts distributed over different nodes.
- Management: it must be manageable. As we introduce new capabilities, we want the administrator to be able to pilot service deployment, deal with security or deal with service life cycle easily.
- Persistency: it must provide a persistency mechanism for service configurations. A deployed service may be stopped. But it can always be asked to restart, recovering the state it has before stopping. This property is also interesting when replicating services on different nodes of a cluster, keeping the sessions alive.

4. Our Java EE Server Architecture

By introducing a Service-Oriented Platform into Java EE servers, we would like to benefit from their dynamic and flexible capabilities. But different service environments exist, each one of them with particular advantages and drawbacks. A first step of our work is to compare them and to choose the most appropriate one; that would be the one that meets the requirements listed above. We can then propose our new Java EE architecture.

4.1. Existing Service Platforms

There are many different service platforms and environments that are differentiated by particular properties. The most important ones are the way the provided services are accessible (invocation), either locally or remotely, the way a service departure is announced and the kind of directory they use for registering services. The most well known and used service environment relates to Web Services [5]. JINI [6] initiated by Sun is now losing speed in the face of UPnP (http://upnp.org). A summary of most important platforms' characteristics are summarized in the figure 4.

	Invocation	Departure Announce	Directory Type	Language
JINI	Remote (RMI)	Lease	Distributed (ad-hoc)	Java
CORBA CosTrading	Remote (IIOP)	No	Distributed	All
UPnP V1 DPWS	Remote (HTTP SOAP)	Message Bye	Distributed (ad-hoc)	All
Web Services	Remote (HTTP SOAP)	No	Centralized (Replicated)	All
SLP / DNSSD	/	Message Bye	Distributed	All
OSGi	Local (Reference)	Event	Centralized	Java

Figure 4. Services platform/environment

Among those the closest from the requirements we have listed above is the OSGi platform. The OSGi Alliance (http://www.orgi.org) is an independent, non-profit corporation working to define and promote open specifications originally intended for the delivery of managed services to networked environments, such as homes, cars or servers. These specifications include the

definition of the OSGi Service Platform, which consists of two pieces: the OSGi framework and a set of standard service definitions. The OSGi framework is a Java-based deployment and execution environment for components.

The OSGi framework supports uninterrupted deployment of components within deployment units called bundles. The framework also provides a service registry that allows the components delivered through the bundles to interact following a service-oriented approach. The continuous deployment activities supported by the framework include bundle installation, activation, deactivation, update and un-installation of bundles. The framework ensures that deployment dependencies at the bundle level are satisfied before allowing the bundle to be activated. Bundle activation results in the creation of the component instance deployed inside the bundle.

Component instances can publish or discover services provided by other component instances at run time. In OSGi, a service is published from a service interface, a reference toward the component implementing the service and a set of properties. Those properties, defined as keys and values, allow clients to differentiate among two equivalent service offerings (i.e., two services with the same interface). Moreover, the registry allows constraint searches to be made using LDAP filters based on the properties. Because service publication or departure can occur at anytime, the service registry supports a notification mechanism that allows service clients to be aware of a particular service arrival or departure. In OSGi, application assembly is done at execution time as a result of the interaction between components and the service registry.

The OSGi service platform permits Java service implementations and provides event notification for announcing service state changes. It has a centralized service registry. It is easily manageable as it already specifies security policies and piloting services is possible locally or remotely. But unfortunately, nothing exists in the platform specification concerning service state persistence. Additionally, this service environment is very interesting because of its small memory footprint, since it was designed for embedded platforms. This property ensures we are not going to impose significant overhead on the server. And the last key point of this architecture is the packaging and deployment capabilities it provides. This can considerably simplify the administrator's task since this work is supported by the framework.

4.2. Proposed Architecture

Our proposed architecture based on the OSGi service platform can be divided into three key points: the packaging, the deployment and the services' binding. These three points are discussed in this section.

The packaging is an important point of our architecture because as we want to dynamically deploy the services we must provide a way to modularize them. When deployed, a module must not interfere with already deployed modules. And it must be possible to uninstall them. The bundles provided and standardized by OSGi offer these properties. Concretely they are Java archive (JAR) files with additional meta-data included. Standardizing packaging in Java holds center stage and is being addressed by JSR 277 concerning Java Modules Systems, but for now it is an emerging work.

Our modularized services have to be deployed on the framework when needed. This task is also delegated to the OSGi platform. It permits installation of local or remote bundles by managing a cache. It can also manage module updates and their code dependencies. It manages the stop and restart of concerned services if needed when different versions of classes are loaded.

```
<component name="WebContainer">
   <implementation
     class="org.objectweb.jonas.service.impl.WebContainerImpl"/>
   <service>
      <provide
        interface="org.objectweb.jonas.service.WebContainer"/>
   </service>
   <reference name="security"
      interface="org.objectweb.jonas.service.Security"
      cardinality="0..1" policy="dynamic"
      bind="setSecurity" unbind="unsetSecurity"/>
</component>
```

Figure 5: Services' ADL

Java EE services are expressed by the dependencies they have with other services [7]. For instance the HTTP service (Servlet + JSP engine) may require a security service if deployed application require some security. The services are defined by contracts and some properties that allow selection among the whole set of services by trading with the registry. Each module containing services embeds a descriptor that specifies the services it offers and the ones it requires. This description follows the OSGi's Declarative Service Specification. An example of description is given in figure 5.

This descriptor allows the framework to manage the binding between the services and the services' life cycles. If a required service is not present then a service will not start. As soon as the dependencies are resolved the service starts. The framework also manages stopping and rebinding services in case a service is updated or unavailable.

We do not explicitly express the dependencies we have between the containers and the services. Indeed, depending on the application, the container may require all of them.

This new Java EE architecture consists of a set of bundles (figure 6) for the technical services and one for the core of the server (this bundle will mainly manage application deployment). It is still possible with this architecture to deploy applications at runtime, but now the services also have the same capabilities.

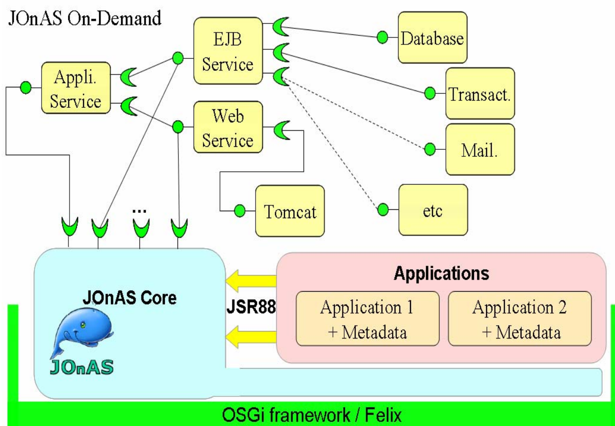

Figure 6: Java EE Architecture

5. Experimentation and Feedback

We implemented our approach, using one of the open source offerings, JOnAS, in which we integrated the OSGi service platform. Moreover, this work has been realized in collaboration with the JOnAS project leaders, Bull SAS. This prototype is called JOnAS On Demand (JOD) [8]. As previously stated, this implementation does not specifically offer any dynamic properties to its service layer. Any capability we introduce is obtained through the use of OSGi [9].

Integrating the OSGi platform inside the server means that the services (Java EE services) have to become OSGi services. The service interface we have chosen was the one already used inside JOnAS for accessing the services (figure 7). It permits in this first step not to be intrusive on the existing implementation.

The rest of the server core has also to be delivered inside a bundle. Packaging the services and the core inside bundles was the first steps of our work. As OSGi employs explicit package dependencies between the bundles, we also provide the right meta-data inside each bundle. When deployed and activated, according to the descriptor we present above, the bundles register the services in the OSGi registry and bind to the required services.

Once done, we are able to offer dynamic service deployment, installation and activation. We are also able to stop and uninstall them. With OSGi class loading properties, all the classes are destroyed when un-deploying a bundle and all the resources are freed. OSGi also proposes a mechanism for updating services. It stops the services depending on the one we are updating. It then performs the re-binding when the new service is available. So OSGi meets all the requirements we had except the persistency of the services' state. As we have all the deployment capabilities for packaging and providing, having service replication will help stopping and restarting service. This missing point will be the subject of future work. OSGi keeps one's promise but also brings additional new interesting features for Java EE.

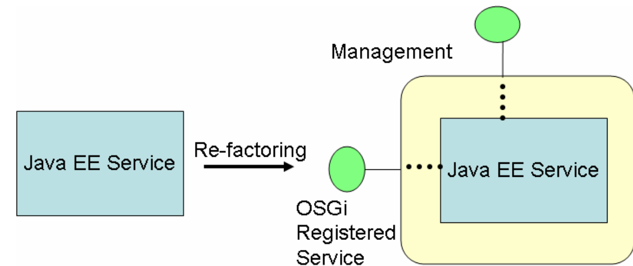

Figure 7: Re-factoring Java EE Service

Concerning applications, we do not want to break the dynamic deployment capabilities offered by the JSR 88. What we do for that point is automating the generation of OSGi bundles meta-data when an application is deployed. In this way, applications can also benefit from the explicit OSGi class path definition. It can import some classes provided by services or the core bundle. This capability solves a major class loading problem of JOnAS. In its current version it implements a hierarchical class loader tree. In this context, a class loader looking for a class first asks its parent for the class. Applications, as ending leafs, had to use the server library version and were not able to embed a more recent one. This issue is tackled by OSGi because we can ignore the import of server packages when generating the bundles metadata.

Another enhancement is OSGi's capability of managing different versions of the same class loaded at the same time in the JVM. This attribute permits the server to provide different service versions. For instance we are able to deploy both an application that requires Servlet in version 2.3 and another one requiring Servlet in version 2.4. This was not possible in the previous architecture.

Since the OSGi gateway downloads remote bundles, manages a cache and intends to host services provided from different hosts, security is a major issue in this context. For this problem we completely rely on the OSGi platform security properties. To prevent malicious code from executing, we can only authorize the execution of services that are contained in signed bundles. Moreover, we can specify security properties to prevent a service from accessing another that it is not authorised to call.

Another convenient feature of the SOA in OSGi is that the platform manages event notification when services arrive or leave. We benefit from this by studying the contract interfaces of the registering services. We assume that the ones ending with MBean indicate a service that is manageable with JMX. We take advantage of this to automate the MBean Object registration in the active JMX server.

Finally, due to the fact that OSGi is a widely used standard, we can benefit from existing services, those specified in the specification or others available in open source. Their integration is simplified and can again

separate some concerns from the core implementation. For instance the log service used in JOnAS is a library provided by Apache. We can replace this dependency by the use of the standardized OSGi Log service. This service could be implemented thanks to the Apache implementation or by another. This will not impact the server implementation.

6. Use Case

We experimented with our re-architectured Java EE server in a real industrial case, to tackle an issue occurring in Bull SAS. We addressed the Edge Computing domain [10]. It is a new computing paradigm designed to share computing and storage resources over the Internet between several organizations. The resources are allocated on-demand when load peaks and flash crowds occur on the company's IT infrastructure. Resources are scattered over the Internet Service Providers/Content Delivery Network backbones and are generally high density of low-cost blade servers. Those servers are preferably close to the end users to improve response time to the end user and to alleviate the ISP/CDN backbone (figure 8). The benefits of this paradigm for a company are on-demand performance scalability and quality of service (QoS) such as response time, jig for isochronous data (audios, videos, gamer actions …) since resources are preferably close to the end users.

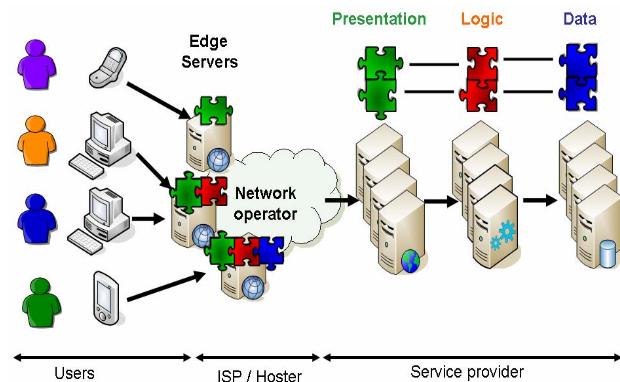

Figure 8: Edge Computing Architecture

The available power can be used for service/application execution. Moreover, in addition to the resource distribution, we can take advantage of the proximity of the server to the clients to host services to decrease response time.

In this particular and very dynamic environment, the usable Edge resources are not owned by the application provider. It is important to use them as little as possible (because this has a cost), only to guarantee the quality of service. Deployment and retreat (service un-installation and eventual state repatriation) of component services are key points of the environment. Thanks to the properties we provide with our dynamic service-oriented server we are able to deploy when needed the Java EE applications and the required non-functional services on Edge servers. Moreover we can retract then when the peak load has ended. Due to the management complexity of the environment, we rely on an autonomic manager for decision making but this is the subject of other work [11] [12].

This dynamic deployment and activation of services can also be adapted to fit a cluster environment. In that case, cluster nodes appear or disappear dynamically (for instance during a peak load). We can then benefit from deploying, adapting and configuring each service on each node during runtime. We do not tackle this problem yet since we must have state replication capabilities in our deployed services. This is a necessary condition for being capable to treat similar requests on different nodes. As this problem is also a limitation for component service retreat (see section 5), this is a perspective of our work.

7. Related Work

We have already described the way dynamism is tackled inside open source implementations. This section describes another approach that has been chosen to make Java EE servers, and more precisely JOnAS, more adaptable.

In this work [13] the idea is to homogenize the component approach, not only for application, but to the whole server. The component model used, Fractal (fractal.objectweb.org), is manageable and adaptable. But contrarily to our approach, this implementation is adaptable meaning that it is possible, at runtime, to change the binding between the components. But it is not dynamic as only the already deployed components can be re-bound. We consider that the server must be capable of incremental construction. This property is very interesting in dynamic environments, like clusters or edge computing. We also believe that the component approach is for now mainly suitable in cases of single operated applications. As we explained, the Java EE server is the assembly of different very complex components provided by different projects. Integrating those different bricks corresponds better to a service than a component approach. Indeed, trading a service instead of namely binding it offers better flexibility.

8. Conclusion and Perspectives

This paper has presented the introduction of a service-oriented architecture into a Java EE application server. It illustrates the benefits it brings to the server by the way it is made more dynamic and flexible. We have studied the different service platforms in this work and chosen the most suitable one for our implementation.

The implementation we have done to validate the approach offers many improvements to the current JOnAS implementation. This is the reason why this architecture will be introduced in the next version (5th) of the JOnAS application server (announced at the

ObjectWeb Conference [14]). We believe that the service-oriented architecture will become an absolute must in the next generation Java EE servers and may be the basis of JBI.

Perspectives for this work are twofold. The first one concerns service replication. The second one is the full integration of the service-oriented architecture into the application layer.

As we saw, replicating the services and being able to make their state persistent can become the next feature that will differentiate Java EE servers. Some implementations already propose it. In clusters or in edge environments, where nodes can be added dynamically, the capability to replicate services to offer load balancing is crucial.

Concerning applications, we saw that we treat them by adding the meta-data they are missing at deployment. This allows us to modify the Java EE packaging. But if we consider that Java EE is just a service-oriented environment that provides some specified services, why not think of forgetting the containers in which the components reside? A Java EE application is only a component providing a service and requiring some others. This vision will be the basis of our future work. To that end, we should also go deeply into the definition of architecture when talking about services.

9. References

[1] R. Allen and R. Douence and D. Garlan, "Specifying Dynamism in Software Architectures", *Proceedings of Foundations of Component-Based Systems Workshop*, 1997

[2] M. Fleury and F. Reverbel, "The JBoss Extensible Server", *ACM/IFIP/USENIX International Middleware Conference*, Middleware, 2003

[3] M. Fleury, "Professional Open Source and the Future of JBoss", *30th Internatinal CMG conference,* Las Vegas, December 2004

[4] J. Genender, B. Snyder and S. Li, "Professional Apache Geronimo", *Wiley book,* ISBN0-471-78543-1

[5] K.P. Eckert, "The Fundamentals of Web Services", *The Industrial Information Technology Handbook*, 2005

[6] K. Arnold, "The Jini Architecture: Dynamic Services in a Flexible Network", *DAC,* 1999

[7] H. Cervantes and R.S. Hall, "Automating Service Dependency Management in a service Oriented Component Model", *ICSE CBSE Workshop,* Portland, May 2003

[8] M. Desertot and D. Donsez, "Infusion of OSGi Technology into a J2EE Application Server", *OSGi World Congress, Presentation,* Paris, October 2005

[9] R.S. Hall and H. Cervantes, "An OSGi implementation and experience report", *IEEE Consumer Communication & Networking Conference*, CCNC, Las Vegas, January 2004

[10] A. Weihl,, P. Jay and E. William, "Edge computing: Extending Enterprise Applications to the Edge of the Internet" *In Proc. of the 13th International World Wide Web Conference,* May 2004

[11] M. Desertot, C. Escoffier and D. Donsez, "Autonomic Management of J2EE Edge Servers", *3rd International Workshop on Middleware for Grid Computing, MGC'05*, Grenoble, November 2005

[12] M. Desertot, C. Escoffier, P. Lalanda and D. Donsez, "Autonomic Management of Edge Servers", *In proceedings of the International Workshop on Self-Organizing Systems, New Trends in Network Architectures and Services, IWSOS'06,* 18-20 September 2006, Passau, Germany

[13] T. Abdellatif, "Enhancing the Management of J2EE Application Server Using A Component Based Architecture", *31th IEEE/Euromicro Conference, CBSE,* Porto, Septembre 2005

[14] M. Desertot and T. Abdellatif, "JonAS 5, The ObjectWeb's Next Generation Application server", *5th Annual ObjectWeb Conference, Presentation,* Paris, February 2006