iZENEsoft Cloud Platform

iZENEsoft R&D Team

January 14, 2011

Contents

1 Introduction 2

2 Application Isolation in PaaS 2

3 SOA and Cloud 4

4 Design Issues 4

5 OSGI Based Cloud 5

1 Introduction

It's hard to clearly differentiate our cloud platform between Saas (Software as a Service) and PaaS (Platform as a service), because either of them are possible: We might be able to provide public application services to different kinds of customers, such as `Recommender as a Service`, or deploy different kinds of applications on single platform. As a result, this report will firstly start from illustrating key design principles, and then presenting our own architecture.

2 Application Isolation in PaaS

Providing a Platform as a Service offering means running multiple applications from different customers inside the same infrastructure and on top of the same platform (also described as multi-tenancy). The infrastructure could be a number of ordinary servers or virtual machines typically provided by IaaS providers. In both cases its extremely important to guarantee application isolation. This means one application cannot interfere with another application running inside the same platform. To achieve this multiple requirements have to be met:

- Application isolation:
  - CPU quotas: Multiple applications running on top of a physical machine have to share the CPU in a fair way. Particularly it must not be possible that one application starves other applications while processing an CPU intensive task. Also SLAs could define a fixed amount of CPU capacity for one application.
  - Memory limits and isolation: Each application must be guaranteed a minimum amount of memory and must not use more memory than permitted. Also one application may not access (read or manipulate) the memory of another application.
  - Network limits and isolation: Each application must be guaranteed a minimum amount of the available network bandwidth and my only communicate using specific protocols and destination addresses. Furthermore one application may not access the network traffic of another application.
  - Often applications running inside an PaaS infrastructure cannot access the disk. So the following requirements may be untenable:
    - Disk quotas: Maximum hard drive space an application can use.
    - I/O rate limiting: Limits the I/O rate with which an application is accessing the hard drive.
    - File system isolation: Files managed by one application may not accessed by another application.
- Data isolation: Typical applications need some type of database. One application must not access data stored by another application. Particularly in PaaS environments this is important as a number of providers use a shared storage solutions for all applications. Data isolation has to be guaranteed by the used storage solution, which is not topic of this post.
- Virtualization Technology
  There basically two approaches to realize isolation. The first one is to use the isolation provided by virtualization. Virtualization technology is heavily used by IaaS providers. In order to be efficient multiple VMs of different customers are running on top of one physical machine, typically using a bare metal virtualization approach. The hypervisor is now in charge of managing the physical resources (CPU, Memory, Dis, Network, . . .), allocating them to the VMs and isolating the VMs from each other. It should be impossible that one VM can access or manipulate another VM running on the same machine.

PaaS providers can use the same technique as IaaS providers. In this case each application instance is executed in its own VM. Therefore multiple applications could run on top of a physical machine encapsulated in VMs. The hypervisor is now in charge of isolating the VMs and therefore applications. This approach has the advantage that its pretty easy for the PaaS provider to realize application isolation as they are using proven technology. However, this approach has a number of downsides. Each application instance requires its own operating system to be booted which increases the resource footprint of each application instance. Also the start process takes longer, as it includes the VM setup, initialization and operating system start process or alternatively the continuation of a VM snapshot.

Open source virtualization based IaaS solution contains:

1. Eucalyptus [12]
2. Opennebula [20]
3. Ganeti [1]
4. libvirt [9]
5. AbiCloud [**?**]

- Operating System and In-Process Security Mechanisms
  The second approach uses existing security mechanisms based on the operating system or implemented inside the application process. For example SE-Linux [8], approaches like Linux VServer [10], OpenVZ [18] and Linux Containers [11] or other approaches described as operating system level virtualization could be used to provide application sandboxing. The huge advantage of operating system level isolation is, that it does not add any overhead to starting an application instance and running it. As a downside these approaches are heavily dependent to the used operating system.

  Some isolation capabilities are already provided by programming languages and their runtime environment. Especially the Java Virtual Machine includes a mature set of security features. Theses have been developed under the assumption that untrusted and potentially malicious code is executed within a Java Applet which therefore has to be isolated. This features also could be used for server side applications in order to provide process isolation. Particularly the Java VM and its features are interesting as a number of programming languages like Ruby, Python, PHP and JavaScript could be executed on top of the VM and therefore reuse existing security features. Other interesting projects are Google NativeClient [2] to isolate C and C++ applications and the Google V8-Engine to isolate JavaScript applications. The downside of this approach is, that the PaaS provider has to implement and in some cases even extend the security mechanisms on its own. Also the mechanisms distinguish between different languages which causes increased implementation and maintenance efforts. Furthermore this approaches often cannot provide isolation on CPU quotas and networking. An advantage however is, that these approaches do not depend on the used operating system.

In summary there are two approaches to provide application isolation in PaaS environments. The virtualization technology which provides strong isolation capabilities at the price of resource usage and a longer application start time. Techniques running under the name operating system level virtualization could be used to isolate processes. This approaches heavily depend on the used operating system but do not increase the resource usage of one application. The last described approach is to use existing security mechanisms in programming languages and their runtime environments. Compared to the operating system level virtualization mechanisms they are independent to the operating system but require the PaaS provider to implement security mechanisms on its own. Furthermore they often cannot provide features like CPU quotas or Network isolation.

3 SOA and Cloud

SOA [7] refers to Service-oriented architecture. The relationship between cloud computing and SOA is that cloud computing provides IT resources you can leverage on demand, including resources that host data, services, and processes. Thus, you have the ability to extend your SOA outside of the enterprise firewall to cloud computing providers, seeking the benefits already described. We describe this process as "SOA using cloud computing". As a result, SOA is the aim for enterprises instead of the approach itself. So we could not easily introduce a mature SOA implementation such as Tuscany[15] easily, because we serve for SOA instead of design based on SOA, SOA is more like a terminology for integrating heterogeneous enterprise flow. However, SOA metamodel provides a good way to see how SOA leverages a process/orchestration layer to change major business processes without driving changes to all systems. This is a loosely coupled architecture 1. The principles of service oriented design in SOA are the same as ours cloud platform.

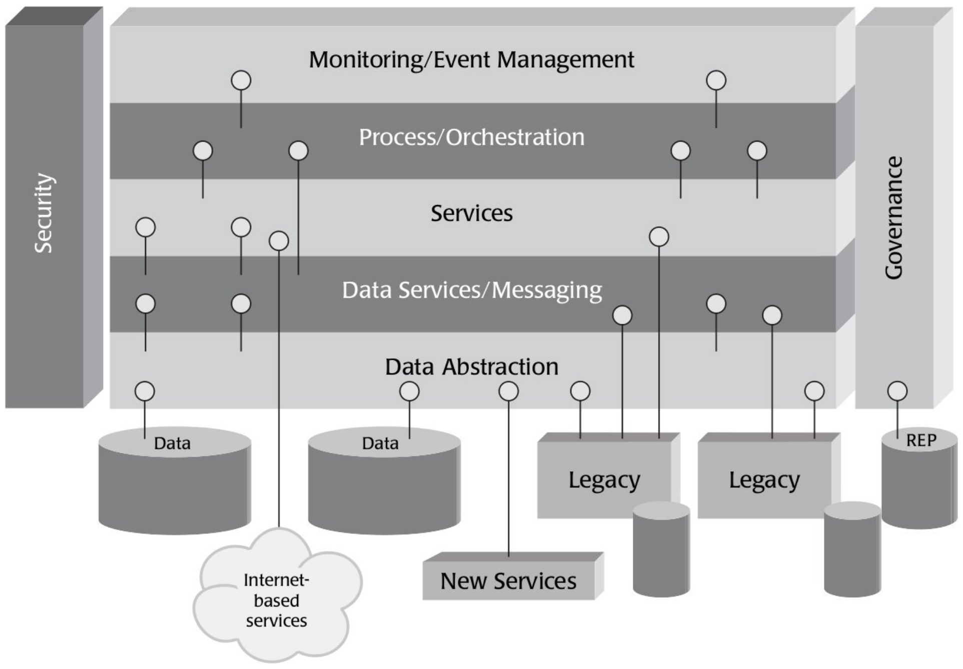

Figure 1: SOA

4 Design Issues

What are the design issues for our cloud platform then?

- Service oriented design
  Similar with SOA, we also have many internal services, which should also be loop coupled and reused.
- Application or service isolation
- Resource oriented architecture
  Different from general SOA, each of our internal services are based on intensive data. We use the terminology `Collection` to partition data—Different services are binded to certain

collection, while data from different collections are relative independent. This leads to a very intuitive structure to map resources over collections, leading to a Resource Oriented Architecture (or, ROA[6]) , which is a specific set of guidelines of an implementation of the REST architecture. `RESTful` interfaces are essentially cloud friendly as well.

- Data intensive computing Services binded to each collection are either computing intensive or storage intesive.

- Multiple languages support and easy migration overhead
  We have Java based bridges as well as lots of C++ based indexing and mining libraries. Multiple languages should be supported with least migration overhead. UIMA AS [16] as well as UIMA Cpp [17] are good toolkits to build up cloud mining services because of their service oriented design, however they are not suitable for us because:

  1. UIMA defines a strict type system to describe collection data, which can not cater for existing mining algorithms easily.
  2. Each C++ written services should utilize UIMA's own C++ libraries to interact UIMA core framework in Java, including string, thread, socket,...,etc, which make code chaotic and high migration guide.

5 OSGI Based Cloud

After studying results from Java enterprise community, we presented an OSGI based architecture to resolve design issues mentioned above. OSGI [5] is a module system and service platform for the Java programming language that implements a complete and dynamic component model. Applications or components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot; management of Java packages/-classes is specified in great detail. Application life cycle management (start, stop, install, etc.) is done via APIs which allow for remote downloading of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly. The OSGI framework is conceptually divided into these following areas, 2:

- Bundles
  Any framework that implements the OSGi standard provides an environment for the modularization of applications into smaller bundles. Each bundle is a tightly-coupled, dynamically loadable collection of classes, jars, and configuration files that explicitly declare their external dependencies.

- Services
  The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects (POJO).

- Services Registry
  The API for management services (ServiceRegistration, ServiceTracker and ServiceReference).

- Life-Cycle
  The API for life cycle management for (install, start, stop, update, and uninstall) bundles.

- Execution Environment
  Defines what methods and classes are available in a specific platform.

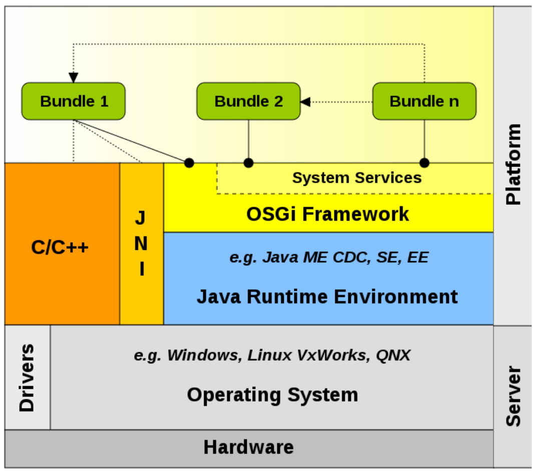

Figure 2: OSGI Structure

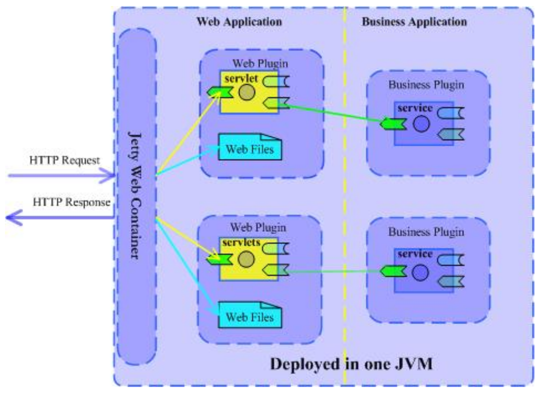

Figure 3: An example of OSGI Based Java EE Architecture

Figure 3 provides an example on how to design an OSGI based J2EE application—all services are encapsulated into plugins(bundles) and deployed dynamically within any application container.

OSGI based architecture has been applied to SOA successfully, additionally, most of the application server has already adopted such a structure. Given OSGI, the design works are summarized as:

1. How to define bundles
2. How to define services within bundles
3. How to define services' interaction accrossing bundles

We can see it more clearly in figure 4.

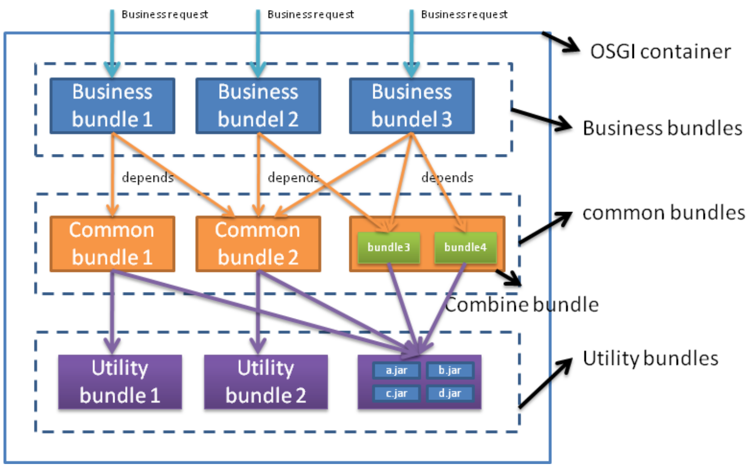

Figure 4: OSGI Based Design

There's a simpler C++ version of OSGI—SOF [14] 5, which will be a very good toolkit for us to provide cloud friendly architecture.

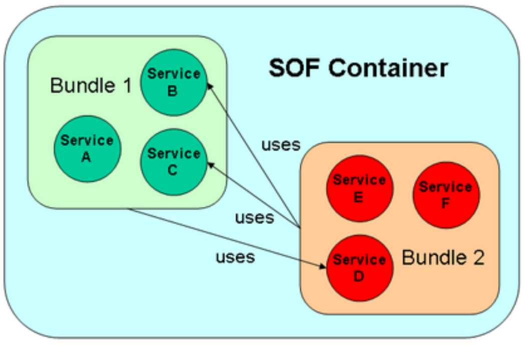

Figure 5: SOF

From the description above, we present our OSGI based architecture with RESTful interfaces, seen from 6:

- Reflecting Collection through RESTful API. Collection is one of our most important Resource.

- Each collection has its cooresponding bundle. As a result, each collection can be created/stopped/loaded dynamically. In that case, we can provide collection management totally dynamically through RESTful API instead of current static approach through configuration XML.
- Each collection bundle depends on a series of bundles, cooresponding to indexing, mining,...,etc.
- There exist dependencies among services across bundles. We provide bundle-level component reusage.
- Each bundle has its own configuration or schema, while all bundles run within same process space, so we can save memory usage because some resources are shared across bundles. Eg: We can encapsulate LA as a service within Index bundle, and TG bundle can access LA interfaces without loading LA resource files again. This is one of the major contribution of C++ based OSGI model.
- Each bundle could be a single `.so` file. SOF provides mechanism on loading dynamic libraries manually, although cooresponding implementation has not been finished,yet — it's only effective under Windows, for Linux, we could use `dlopen` to finish the implementation.
- We provide application isolation based on resource. When binding same bundles to each collection, we provide SaaS cloud, while binding different bundles to each collection, we provide PaaS cloud.
- The tasks for R&D from now on are mainly providing a series of bundles. To satisfy different requirements, we just assemble them with only writing few codes.

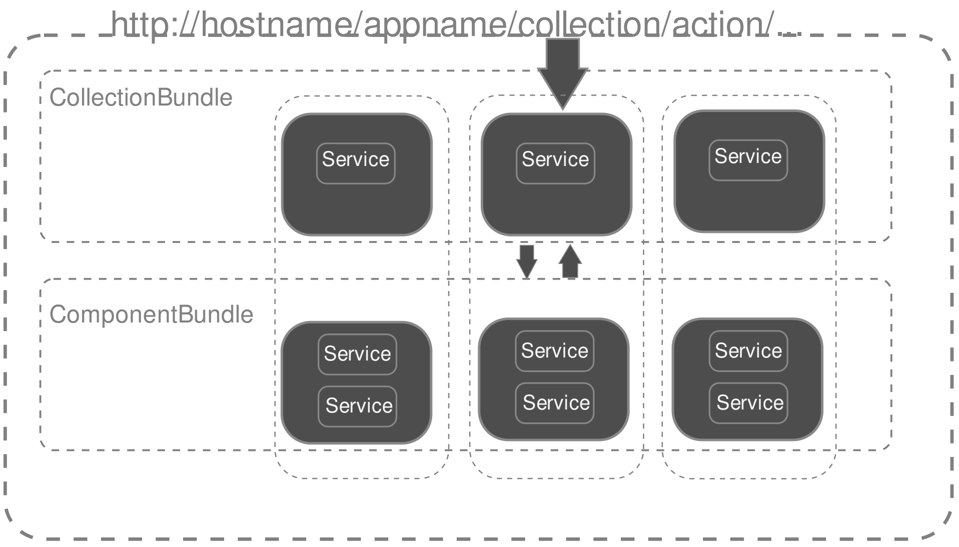

Figure 6: OSGI Based Cloud

Additional works are requires since SOF is just a simple version of OSGI, the most important three points are:

- Enrich the event mechanism of SOF. Currently, SOF only provides two kinds of events:
    1. Register—which is used when bundle is loaded

2. Unregister—which is used when bundle is stopped

We need at least two additional events:

1. Method Start
2. Method Stop

Because we need them to specify the `Pub-Sub` relationship between bundles.

- Implementing an `IoC` container [4].
  `Inversion of Control`, or `Dependency Injection` [3], are terminologies from Java and other ducktyping languages. We hope to implement it in C++ because we hope the dependency among bundles could be totally configured through XML, instead of hard coding. Eg: Customer A wants Index bundle only while customer B wants both Index bundle as TG bundle. There exist dependency between these two bundles, given `Dependency Injection`, we can keep Index bundle totally unchanged.

- How to bind distributed computing framework to OSGI.
  Distributed computing framework is not necessary at currently, because we suppose each collection can run on single node now. This assumption might not be true when we provide public cloud services in future, in such cases, mining data from huge data is inevitable, eg: frequent matrix computing over huge users and commodity items within recommender system. As a result, it's reasonable to bind a distributed framework to SOF framework. Since we provide C++ level component reusage, it's better to use C++ framework instead of popular Hadoop, such as Coord [19] and Sector [13]— the former one is the framework used by Naver for distributed data mining.

References

[1] http://code.google.com/p/ganeti/.

[2] http://code.google.com/p/nativeclient/.

[3] http://en.wikipedia.org/wiki/dependency_inversion_principle.

[4] http://en.wikipedia.org/wiki/inversion_of_control.

[5] http://en.wikipedia.org/wiki/osgi.

[6] http://en.wikipedia.org/wiki/resource-oriented_architecture.

[7] http://en.wikipedia.org/wiki/service-oriented_architecture.

[8] http://fedoraproject.org/wiki/selinux.

[9] http://libvirt.org.

[10] http://linux-vserver.org/welcome_to_linux-vserver.org.

[11] http://lxc.sourceforge.net/.

[12] http://open.eucalyptus.com/.

[13] http://sector.sourceforge.net.

[14] http://sof.tiddlyspot.com/.

[15] http://tuscany.apache.org/.

[16] http://uima.apache.org/doc-uimaas-what.html.

[17] http://uima.apache.org/doc-uimacpp-huh.html.

[18] http://wiki.openvz.org/main_pagel.

[19] http://www.coordguru.com/.

[20] http://www.opennebula.org/.