Using OSGi as a Cloud Platform
The Cloud - Challenges

- Cloud Computing is the economies of scale
  - E.g., system of engagements

- Traditional software stacks usually do not scale well
  - Often designed for vertical scalability.
  - If they were not designed to scale horizontally, they don’t.

- Monolithic software cannot be expected to be elastic.
Cloud Computing is outsourcing
- Computation by the hour
- Focus on the software, no longer control the platform

Problem: Dependability
- Amazon EC2: 22 minutes of connection issues on 03/15 in the US-EAST-1 zone due to router configuration problems
  [Link](http://news.ycombinator.com/item?id=3707590)
- Microsoft Azure: ½ day partial service outage on 02/29 due to leap day bug
- Zoho: 3h45 full outage on 02/20 through a power failure at an Equinix data center. Full recovery took another 6 hours.
  [Link](http://www.zoho.com/general/blog/our-friday-outage-and-actions-we-are-taking.html)

You need to design for failure?!?
- Availability is a software issue, not a platform issue
The Problem

- Both vertical scalability and redundancy (design for failure) require a distributed systems approach.

- Distribution adds complexity and its own failure models.  
  Example: Kerberos, Distributed File Systems

- Modular systems are easier to design  
  - Focus on functional partitioning and composition, not communication

- Well-designed modular systems can be turned into distributed systems  
  - Let’s bring OSGi to the cloud  
  - Let’s see what is already in OSGi
Modularity

- **Declaratively Self-Contained**
  - A module is self-contained with regard to its own content and its declared dependencies

- **Encapsulated**
  - A module exposes its content solely through well-defined interfaces

- **Decomposed**
  - Modules are created by segregating a larger problem into smaller sub-problems so that a module ideally only deals with a single, not further separable concern and the content of the module is highly cohesive.

- **Composable**
  - Modules are created for reuse in different applications and can be composed into new applications. Declared dependencies and declared interfaces put constraints on the validity of compositions. Ideally, the degree of coupling between modules is low so that composition is facilitated and not prohibited.

- **Substitutable**
  - Two modules (or sets of modules) providing the same interfaces can be exchanged for one another.

- **Localized Behavior**
  - Modules are designed to behave locally, i.e., the effect of the code is restricted to the content of the module or its declared dependencies. A module should not make any assumptions about its dependents other than the ones expressed through the declared dependencies.
Modularity in OSGi

- Package dependencies = tight coupling
- Services = loose coupling

How far can we get in terms of dependability with plain loose coupling?

=> now components can fail independently.
Parrot

- Talk to the parrot
- Parrot repeats what it has heard

- Problem: Parrots are inherently unreliable
- Solution: Design for microreboots

Failure model: single component fails
State and Identity

- ConfigAdmin gives each service an identity
  - Can be set through the PID

- Each individual service can have state

- Managing the state through the ConfigAdmin by attaching it to the identity
  - By value for a small amount of state
  - By reference for a large state
And now for something completely different.

Remote Service Admin: Mechanism
– Expose this local service
– Import this remote service

Topology Manager: Magic
– Which service from where, …
Parrot - Version 2.0

- Person listens to Parrot (poll)
- Parrot autonomously fetches the most popular terms from Twitter

Failure model: entire node fails
Dependability as a Service

- Manage the available resources in the cloud deployment.
  - As a sensor and as an actor

- Manage the identity of reliable services in the cloud deployment.

- Act when a reliable service experiences a failure.

- Prototype based on Remote Services for OSGi (was: R-OSGi), Zookeeper, optionally a key-value store
  - Remote Services for OSGi can re-bind service proxies

- Using it on my Eucalyptus cloud as well as external providers

- Interesting implications: Hot Spares
OSGi as a Cloud Platform

- Instance creation is likely to remain vendor-specific
- REST API provides access to the running instance
- Can be easily integrated into languages other than Java.

- Examples:
  - GET  http://my_host/framework/bundles/representations
  - POST http://my_host/framework/bundles
  - GET  http://my_host/framework/bundle/5/state
  - GET  http://my_host/framework/services/representations/(objectClass=org.osgi.*)

- This is work in progress (RFC 182)
OSGi as a Cloud Platform
Architecture

HTTP, REST

Framework Instance

REST

Framework Instance

WebSocket

Framework Instance

http://mycloud.com:8080/instances/20132
HTTP Service++

- State of the art
- Pluggable web applications
- Elastic web portal
Authentication and User Management

- UserAdmin is flexible enough to represent different concepts
  - User Identity through OpenID
  - Authorization through OAuth

- Most cloud-implementations would be read-only though.
Other possible services

- Key-Value Store
- BlockStorage
- Message Queue
- ...

Problem: common abstraction?

What is the consistency model of my key-value-store?
- equivalent to /dev/null?
- ACID?
- Can I read my own writes?
Conclusions

- Modularity as in OSGi solves a major issue with architecting elastic applications for the cloud.
- It helps to structure applications in a more flexible way.
- It enables platform support for making software more dependable.

- Existing standards like ConfigAdmin, RemoteServiceAdmin, UserAdmin can be used with no or little modifications
- Other standards like HttpService are not yet fit for the cloud and need to be reworked.
- The REST API is an important addition for bootstrapping cloud deployments.

What would you like to see in the cloud?