Dependability as a Cloud Service - A Modular Approach

Jan S. Rellermeyer[1]
[1]*IBM Research,*
Austin Research Lab,
Austin, TX 78758
rellermeyer@us.ibm.com

Saurabh Bagchi[1,2]
[2]*Purdue University,*
School of Electrical and Computer Engineering,
West Lafayette, IN 47907
sbagchi@purdue.edu, sbagchi@us.ibm.com

***Abstract*—Failures of services on cloud platforms are only to be expected. To deal with such failures, one is naturally inclined to use the traditional measure of replication. However, replication of services on distributed cloud platforms poses several challenges that are not well met by today's Java middleware systems. These challenges are the need to isolate state in the application components so that easy migration and recovery are possible and the requirement for client transparency when dealing with different replicated service instances. For example, Java Enterprise Edition (JEE) makes it difficult to have transparent replication of services due to the above two reasons plus the fine-grained nature of interactions between its components (the Enterprise Java Beans). In this paper, we show parts of the design of OSGi, a specification defining a dynamic component system in Java, that make it suitable for the above task. We then propose two extensions to OSGi which will allow exposing and exporting application component state and transparent invocation of service instances. These two together can enable easy replication and recovery from failures in cloud environments. We show through experiments that our prototype can migrate a failed service quickly enough to a new machine so that a client experiences only a moderate increase in service invocation time during system recovery.**

I. INTRODUCTION

A decision to move away from privately owned computer resources and towards cloud computing is affected by a variety of factors, dependability being one of them. Dependability is usually defined through its attributes of availability, reliability, safety, integrity, and maintainability [1]. In a complex system, the overall dependability of the system depends on the corresponding attributes of its components. What changes in cloud computing is that one important component—the platform—is no longer under one's own control. Depending on the type of the cloud service, the platform may mean one of several things; at least it includes the physical machine but may also include higher layers like a complete runtime system. This arguably outsources part of the maintenance problem while introducing new challenges. Most importantly, the underlying physical resources (i.e., the machines and the network) are shared between multiple customers and applications and cannot be actively managed. This is often perceived to be a threat to the overall dependability [2], [3], [4].

Typical ways of mitigating these threats in a cloud computing environment revolve around introducing redundancy into the system, especially for increased availability and reliability. Since adverse incidents can always render a single instance of a functional unit unusable, the hope is that by creating multiple copies of the component a sufficient number of copies always survive to deliver the service without degradation. This idea is neither new nor specific to cloud computing and has been successfully applied many many times in earlier distributed systems such as clusters or grids [5], [6]. In cloud computing, however, the effectiveness of redundancy benefits from two particular factors. First, the elasticity of the resource fabric makes redundancy more agile and more affordable. This means that it is possible to acquire additional resources for increasing the degree of redundancy in response to perceived threats much more easily than it can be done in traditional hosted environments. Second, the resource fabric is typically globally distributed, thereby allowing intelligent placement to guard against high-impact outages that would be much harder to circumvent in a hosted environment. By spanning multiple geographic zones, a system can survive even an outage of an entire data center that could be caused by a natural disaster or a far-reaching power outage.

Unfortunately, not every component of a complex system can easily profit from redundancy. Components that have state in them cannot be easily split up for purposes of replicating the parts to different replication degrees and executed on different machines. There have been prior attempts to address this by structuring web services such that most of the components are stateless and the few stateful components expose their state in an orderly manner. Thereby, it can be handled through special, and more expensive, mechanisms such as database replication. However, we discern that this kind of structuring of applications is becoming harder to do. Web services are becoming contextualized by various factors including the user, the client environment, the results of prior requests, and interactions with coordinating and collaborating web services. For example, with the advent of social networks, the services are contextualized by the user profiles and for a rich interactive user experience, the

state (in this case, the profile) is driving the majority of the functionality. *Therefore, our position is that existing mechanisms for structuring applications for purposes of reliability will no longer work with emerging web services, that are being hosted on the cloud.*

The effectiveness of elasticity depends to a high degree on the coupling of components. Systems with fine granular components tend to suffer from a high degree of entanglement, hence limiting the flexibility of selectively replicating components since all tightly-coupled components usually have to reside on the same host. Consider for example, Enterprise Java Beans running on a JEE middleware are typically structured to have final granularity and with frequent invocations between them, both of which highlight the challenges of replication for cloud environments.

In this paper, we discuss how a dependability service can be built on the OSGi framework. OSGi is a set of specifications [7] that defines a dynamic component system for Java. These specifications enable a development model where applications are *dynamically* composed of many different reusable components. The OSGi specifications enable components to hide their implementations from other components while communicating through services, which are objects that are specifically shared between components. We show why a module system for Java, like OSGi, makes our goal simpler to achieve and then show how the dependability service can be architected using features already provided by OSGi. We propose an extension to the OSGi standard that acts as a dependability service, which we call CLOUDDEP. The service provides the following dependability properties:

1) Re-deployment: Migrating the application to a new physical machine or a different data center when the underlying platform fails.
2) Clustering: Eliminating single points of failure in the system by replicating functional units.
3) Elasticity and Load-balancing: Distributing the load to a flexible pool of resources to avoid overloading single instances.

II. SOFTWARE MODULARITY AND DEPENDABILITY

Software modules are self-contained functional units of encapsulation. They contain data and functions, objects, or other entities of the programming languages used to implement them. Ideally, the correctness of a modular system depends only on the correctness of its modules and the correctness of a single module does not depend on other modules [8]. The requirement of being self-contained, however does not automatically imply full isolation. Many implementations of module technology allow modules to be *declaratively self-contained*, i.e., they not only rely on their own content but can additionally declare dependencies to other modules and have access to their accessible content. In such setups, the effects of errors occurring in a single module are still restricted to cause faults in this module or any module that has a declared dependency but not in unrelated modules.

This property has been successfully exploited to increase the fault-tolerance of systems. For instance, in the Actor model [9] the modular units are fully isolated and can only communicate through explicit message passing.

Micro Kernel Operating Systems [10] restrict themselves to implement only the minimally necessary functionality in the kernel and outsource the remaining responsibilities into separate user-space modules. Thereby, faults in one module cannot affect other modules or the privileged kernel.

Microreboots [11] describe a methodology for recovering a system by rebooting only a subset of affected components, in the concrete example Enterprise Java Beans (EJBs). The design approach is to keep application state carefully parceled away in separate state stores which are only accessed through well-defined interfaces. Thus, the application components can be recovered by rebooting them (since the rebooting is fast in the absence of state, it is called microrebooting), while the state store can persist across reboots. This approach shares some similarities with our proposed approach. However, in order to be micro-rebootable, the components need to be well-isolated and stateless except for externalized application state kept in specialized state stores. Such compartmentalization is non-trivial to do in the context of JEE because of the reasons mentioned earlier (tight coupling and fine granular decomposition).

III. OSGI

A. OSGi Background

OSGi is a set of standards for modularity on the Java virtual machine released and maintained by the OSGi Alliance. Originally, it was developed by several major vendors to solve an availability problem on embedded home gateway machines [12]. The intention was that multiple software components should be able to co-exist on the same hardware while each of the software packages needs to run continuously and over long periods of time, even in scenarios that require periodic maintenance updates to the software. To enable dynamic updates without stoppage, it aimed at overcoming the traditional entanglement of small-granular objects and formed larger modules that can be managed individually and composed to form applications. OSGi defines a unit of modularity as the *Bundle*, which is a deployable Java JAR file enriched with additional meta-data. Most importantly, this meta-data describes which packages are exported, i.e., accessible to other bundles, and which packages it needs to import from other bundles. Recently, OSGi has successfully entered the domain of enterprise software and most major Java application servers (e.g., IBM WebSphere, JBoss, Oracle WebLogic, Spring Dynamic Module Platform) are by now internally based on the OSGi technology.

The OSGi runtime environment, the *Framework*, provides the user with mechanisms for installing and controlling the life-cycle of bundles at runtime. Applications are typically composed out of many different bundles. These bundles can interact in a tightly-coupled way by depending on packages provided by other bundles. *Tight coupling* in this context means that the dependencies are required to resolve the bundle and failure to provide the dependencies renders the bundle inoperable. Another way of interaction is through loosely-coupled services. *Loose coupling* means that service consumers only rely on the service interfaces under which the service has been registered with the OSGi framework but not on any knowledge of implementation details. Bundles can consequently query the framework for available services and retrieve the service instance, which is an ordinary Java object in OSGi. The result of a query is either a reference, a set of references from which the client can pick the best matching service, or the information that no such service is available. Since the consumers only rely on the service interfaces but not on any concrete implementation, bundles can potentially operate even in the absence of certain services. OSGi applications are expected to actively deal with dynamism in the setup. For instance, a service can certainly fulfill its purpose in the absence of a log service but if it has logging support, e.g., for debugging, it should enable it whenever a log service becomes available and disable it when the service becomes unavailable.

The modularity in OSGi solves a major issue with creating elastic applications for the cloud. Cloud resources are inherently dynamic in their nature and undergo frequent changes either due to explicit management operations (adding and removing resources) or due to their volatility (sharing effects, failures). Therefore, we believe that monolithic software incapable of dynamic adaptation cannot effectively run in such an environment. In fact, the compositional approach of modular software—traditionally applied to a single runtime system—is key to building scalable and dependable systems across a varying set of machines in the cloud [13], [14]. OSGi particularly facilitates composition and localized behavior by separating tightly coupled and loosely-coupled interaction between modules into two different concepts: package dependencies and services, respectively. Thereby, when services are used as a design element, the resulting systems are less entangled and more flexible.

OSGi was originally designed for managing software on a single Java virtual machine. In prior work, we have shown that the OSGi model can be used as an application model for building distributed systems by turning the services into potential boundaries of distribution [15] and adding simple support for creating redundancy by clustering stateless services on multiple machines [16]. However, little attention has so far been paid to the problem of managing stateful services in distributed deployments like clusters or clouds.

B. How OSGi Configuration Admin Enables Restarts

A traditional OSGi framework (running on a single JVM) already supports restarts of the runtime systems as well as a form of partial restart for bundle updates. This means that whenever a bundle has been updated and the operator of the framework requests a refresh of the system, only those bundles immediately affected by the update (this is the bundle itself and all bundles relying on the updated bundle through tightly-coupled package dependencies) are restarted whereas the remainder of the system is unaffected. However, only the compositional state of the application is preserved across restarts. This is the information about which bundles have been installed when the framework was last running and in which life-cycle state each individual bundle has been in (installed, resolved, starting, active, stopping, uninstalled). Every bundle then gets started again but is itself responsible for restoring any internal state, which includes services that it has bound to.

The only platform support in traditional OSGi for externalizing bundle state is the Configuration Admin service, an optional service described in the OSGi Compendium Specifications [17]. The configuration admin is a service in the system capable of managing and persisting configuration data in a central location. Services that want either to be externally configurable or want to persist their configuration data across restarts can register as a *Managed Service*. This expresses its intent to receive configuration updates and gives the system an interface through which it can push updates to the service. Every managed service is expected to identify itself though a persistent identifier (PID) and announce this PID with its initial registration. The config admin uses the PID of a service as a primary key for associating configuration data to services. Configuration events and changes are signaled to the managed service asynchronously, regardless of whether they are used by the service itself (as illustrated in Figure 1) or an external bundle.

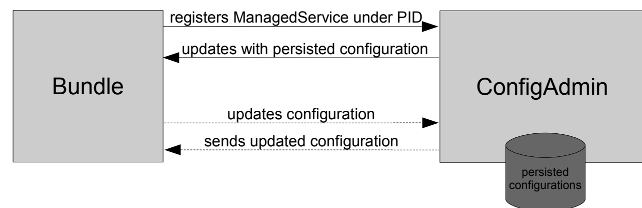

Figure 1. Interaction between Bundle and Configuration Admin

In the context of cloud computing, the most interesting and relevant property of the config admin is the concept of a service identity. When migrating a service to a new machine, it has to be assured that the newly created service instance can fully take over the role of the previous instance. Therefore, stateful services implicitly build up their own identity based on their internal state and their

history. Migration and failover can be considered a "micro-reboot" of the service on a different machine. If all state could be stored as configuration data then a distributed version of the configuration admin would turn managed services into migratable services just as the ConfigAdmin allows stateful services to become restartable. In practice, the storage capacity of the configuration admin and its unclear consistency model are prohibitive to this approach. For instance, a configuration update always fully overwrites any previous configuration so that when two writers are updating disjoint parts of the configuration concurrently the last writer wins and the update from the earlier writer is lost.

However, just as systems like Zookeeper [18] are used in cloud deployments to manage configuration, a cloud-scale version of an OSGi ConfigAdmin can be used to dynamically bind service instances to a richer instance of storage to externalize the service state, e.g., a database system or a key-value store. The challenge of building such a distributed ConfigAdmin for cloud deployments is to coordinate it with the distribution of services. It is not feasible to have a single scope for configurations across all nodes since this would prohibit multiple redundant copies of the same service. When a service needs to be migrated to a different machine, however, the configuration has to be migrated with the service.

C. How OSGi Remote Service Admin Enables Non-local Execution

The Remote Service Admin aims at integrating distribution into the OSGi platform and has been part of the OSGi Enterprise Specifications [19]. This service encapsulates the mechanism for importing remote services into the scope of a local OSGi framework and exporting local OSGi services through distribution providers. When a remote service is imported, a local proxy for the remote service is created and associated with the distribution provider. Some distribution providers like our R-OSGi [15] system are able to operate transparently, i.e., neither client nor service have to be rewritten to operate in a remote setup.

The API of the remote service admin is entirely imperative. Given a known URI or other identifier, the admin provisions the local proxy and handles binding to the remote service reachable through this address, as long as the service is reachable. Finding a service in the network and re-binding in the event of a failure is handled by different components. All policy decisions are encapsulated in a Topology Manager which, e.g., is able to detect clients that are querying the local framework for services and make use of service discovery protocols to find matching services. However, the specifications do not mandate any particular functionality of the topology manager but only specify the interfaces through which a possible topology manager can interact.

For a cloud setup with dependability requirements, the presence of an appropriate topology manager is paramount.

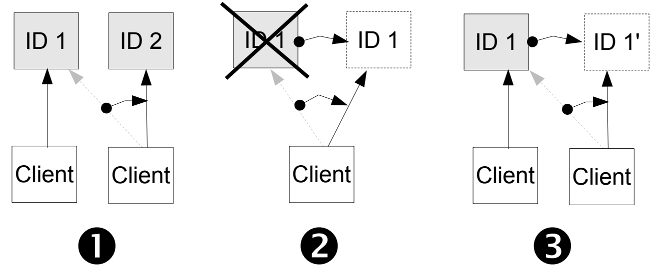

Figure 2. Different Use Cases for a Dependability Service. 1: Load balancing, 2: Failover, 3: Elastic Scalability.

In the following section, we describe our service CLOUDDEP that implements policies to ensure availability of services by acting as both a topology manager and a distributed configuration admin.

IV. TOWARDS A DEPENDABILITY SERVICE

The new service (that we call CLOUDDEP) is designed to turn OSGi services into elastic units of deployment which can be migrated and replicated across multiple cloud nodes to increase the availability and reliability of the administered service. All that is required from the administered service is that it persists all of its internal state either directly or indirectly through configurations and listens to configuration updates by implementing the Managed Service interface of the configuration admin specification. CLOUDDEP acts as a global entity in the system (which can be implemented by a centralized backend or a distributed architecture) and takes over the role of a local configuration admin and a local topology manager on each node of the cloud deployment. It also monitors the service invocations to detect failures. We illustrate the features of CLOUDDEP through three usage cases.

A. Load Balancing

We consider a simple setup with a set of identical (possibly stateful) remote service and a set of service clients as illustrated by case ❶ in Figure 2. Thanks to its facet of being a topology manager, CLOUDDEP has knowledge of the remote service relationship and was involved in the setup of the binding. It had detected the local service requests of the client bundles (e.g., through service lookup hooks as described in the remote service admin specification) and has instructed the distribution provider to create the binding to one of the known remote service instances. Furthermore, it has, either due to cooperation of the distribution provider or by wrapping its created service proxy, instrumented the local proxy to gather information about the frequencies of service invocation and response times. In the event that one remote service instance becomes overloaded, detectable by an increase of invocations and an increase of the response time, CLOUDDEP can re-bind a client to a different service

instance, thereby re-distributing the load among the service instances.

B. Failover

We now consider a setup with one or more clients and a single remote service. In the event of a service failure (e.g., an uncaught runtime exception or the node hosting the remote service becoming unavailable) CLOUDDEP has captured the identity of the faulty service through its configuration admin facet. It can thereby instantiate a new service by starting the corresponding bundle on a new node and inject the identity from the failed service into the new instance, as shown in case ❷. This process is akin to what a microreboot would do on a single node but here applied to a distributed setup. By applying the same rebinding as done for load balancing, the existing client will be forced to now interact with the new service instance. If the latency of service migration is an issue, CLOUDDEP can be configured to keep hot-spare instances of the service. These are running but unconfigured instances that have not yet received an identity. In the event of a service failure, CLOUDDEP can directly inject the identity into a hot-spare instance and create a more seamless failover.

C. Elastic Scalability

The third example combines the load balancing and the failover scenarios to enable elastic scalability of a remote service. In the event of a service over-utilization, CLOUDDEP can start a new instance of the service and bind some of the clients to the new service (see ❸ in Figure 2). A new identity is forked by CLOUDDEP from an existing configuration and can then develop independently from the original sibling. Scaling back to a smaller number of service instances can be achieved by applying the failover strategy: moving all clients of a service to a different existing instance and then dropping the service instance.

V. PROTOTYPE AND EXPERIMENTAL EVALUATION

We have implemented a prototype of CLOUDDEP that replicates configuration and manages failover for OSGi services in a cloud setup. The prototype uses R-OSGi as a distribution provider and cooperates with the protocol and the proxy handler to detect failures or service outages. The detection of failure is done through either a simple timeout or an exception caused by the termination of a connection. Since we induce failures in the experiments through terminating the connection, the latter detection is triggered. CLOUDDEP can furthermore either proactively start idle and unconfigured redundant instances of the service on a backup machine or reactively start a new instance of the service on demand. In either case, the identity of the failed service is injected into the new instance and the existing binding between the client and the no longer available service is transparently altered to point to the new instance. The prototype blocks the failed service request and replays the request to the new service instance as soon as it becomes available. We therefore assume a fail-stop model in our setup.

	Microreboot	Migration
avg. inv. time (T)	$7.23\mu s \pm 1.06\mu s$	$1.22ms \pm 0.15ms$
avg. T during failover	$266.29\mu s \pm 34.90\mu s$	$4.01ms \pm 0.66ms$
relative overhead	3581%	229%

Table I
SERVICE INVOCATION LATENCIES OF THE TREND SERVICE

For the experiment, we use a trend service that updates itself by periodically polling the most frequently used #hashtags from Twitter. The service is clustered by geographic regions, each instance of the service is configured to handle a specific region that is set through its configuration. Since the service only handles the ten most relevant results, it can afford to use the Configuration Admin interface to persist its internal state and does not rely on, e.g., an external blob storage. In a realistic setup, the service would process the results and, for instance, enrich the data with context links retrieved from Google Search or Wikipedia.

We use a single instance of the service and a single client that polls the service every second to receive the latest trends. After one minute, we take the service down and let CLOUDDEP handle the failure based on a strategy that we have pre-configured.

In the first experiment, we have both the client and the service running in the same Java Virtual Machine and on the same OSGi framework. Our failure recovery strategy resembles a microreboot of the service, i.e., we stop the running service and restart it. In the second experiment, we run client and service in different cloud instances and on different Java Virtual Machines and additionally keep a hot-spare VM for the failover. When the original service fails, we instantiate a new service instance on the hot-spare that becomes enabled when CLOUDDEP injects the configuration (i.e., geographic location and previously determined trends) into the instance.

Table I summarizes the measured average service invocation time during normal operation and during the failover. In the case of the microreboot, the invocation time during failover increases by a factor of 35 during failover. The reason for this high increase is that in OSGi clients get a direct reference to the service object and thereby a regular service invocation does not differ from any other method call in terms of latency. Enterprise Java Beans, in contrast, are called through reflection. Furthermore, a bundle restart is potentially more expensive than a microreboot of a single Enterprise Bean. However, the overall latency during failover is still in the range of microseconds, which is very likely acceptable for the large majority of applications.

In the distributed case the baseline of an average regular

service invocation is much higher and is in our setup in the order of milliseconds. However, the failover increases the invocation time by only a factor of three. In practice, this factor will vary with the amount of configuration data that needs to be migrated to restore the service identity and the network used but more complex services are likely to also have higher baseline invocation times. In both the local and the distributed case, only a single service invocation is temporarily blocked and experiences a higher latency. Further invocations continue to operate at normal latency.

The results illustrate the viability of our approach for a large class of applications. Microreboots are an option for recovery from failures, especially when the service is already accessed from a different host. In this case, the latency of the microreboot is within the noise of a remote service invocation. When entire nodes fail or become unreachable, migration of services to different machines are also feasible and do not affect the client significantly. We have, however, assumed that the cloud instance for failover is already instantiated. In our setup, we used an Eucalyptus cluster that requires in average 85 seconds to boot up a new instance to the point where we can interact with it. This latency is unlikely to be acceptable for cloud services but is a common problem of IaaS solutions.

VI. Conclusion

We expect dependability to become an increasing problem for cloud services. Our premise is that traditional ways of structuring such systems provide insufficient flexibility and cannot deal with failures or elasticity requirements as they occur in cloud setups. We have therefore proposed a modular, loosely-coupled approach for building cloud services and have illustrated the concrete challenges using the example of OSGi, a widely used standard for modularity for the Java language. The result of this effort is CloudDep, a dependability service for OSGi that takes care of countering service failures by migrating the identity of the service to a new machine. The experimental results of our prototype implementation show that migration to an existing node is feasible and the impact on the service invocation time is acceptable for most applications.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A view of cloud computing,” *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[3] “Cloud Computing Incidents Database.” [Online]. Available: http://wiki.cloudcommunity.org/wiki/CloudComputing:Incidents_Database

[4] K. Joshi, G. Bunker, F. Jahanian, A. van Moorsel, and J. Weinman, “Dependability in the cloud: Challenges and opportunities,” in *Dependable Systems Networks, 2009. DSN ’09. IEEE/IFIP International Conference on*, 29 2009-july 2 2009, pp. 103 –104.

[5] H. Stockinger, A. Samar, K. Holtman, B. Allcock, I. Foster, and B. Tierney, “File and object replication in data grids,” *Cluster Computing*, vol. 5, no. 3, pp. 305–314, 2002.

[6] J. Abawajy, “Fault-tolerant scheduling policy for grid computing systems,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, pp. 238–244.

[7] OSGi Alliance, *OSGi Service Platform, Core Specification Release 4, Version 4.2*, 2009.

[8] B. Meyer, *Object-Oriented Software Construction*, 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[9] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular ACTOR formalism for artificial intelligence,” in *Proceedings of the 3rd international joint conference on Artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[10] J. Liedtke, “On micro-kernel construction,” in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, ser. SOSP ’95. New York, NY, USA: ACM, 1995, pp. 237–250.

[11] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, “Microreboot - a technique for cheap recovery,” in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI’04, 2004.

[12] P. Kriens, “How osgi changed my life,” *ACM Queue*, vol. 6, no. 1, Jan. 2008.

[13] C. Matthews and Y. Coady, “Virtualized recomposition: Cloudy or clear?” *Software Engineering Challenges of Cloud Computing, ICSE Workshop on*, vol. 0, pp. 38–43, 2009.

[14] J. S. Rellermeyer, M. Duller, and G. Alonso, “Engineering the cloud from software modules,” in *ICSE-Cloud 09: Proceedings of the Workshop on Software Engineering Challenges in Cloud Computing (in conjunction with ICSE 2009)*, Vancouver, Canada, 2009.

[15] J. S. Rellermeyer, G. Alonso, and T. Roscoe, “R-osgi: Distributed applications through software modularization,” in *Middleware 07: 8th ACM/IFIP/USENIX International Middleware Conference*, Newport Beach, CA, USA, 2007.

[16] J. S. Rellermeyer, G. Alonso, and T. Roscoe, “Building, deploying, and monitoring distributed applications with eclipse and r-osgi,” in *ETX 07: Fifth Eclipse Technology eXchange (ETX) Workshop (in conjunction with OOPSLA 2007)*, Montreal, Canada, 2007.

[17] OSGi Alliance, *OSGi Service Platform, Service Compendium Release 4, Version 4.2*, 2009.

[18] Apache Foundation, “Apache zookeeper,” http://zookeeper.apache.org, 2008.

[19] OSGi Alliance, *OSGi Service Platform Enterprise Specifications, Release 4, Version 4.2*, 2010.