Providing Load Balancing and Fault Tolerance in the OSGi Service Platform

ARNE KOSCHEL, THOLE SCHNEIDER, JOHANNES WESTHUIS, JÜRGEN WESTERKAMP
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Ricklinger Stadtweg 120, 30459 Hannover
GERMANY
arne.koschel@fh-hannover.de

IRINA ASTROVA
InVision Software OÜ
Lõõtsa 2A, 11415 Tallinn
ESTONIA
irinaastrova@yahoo.com

ROMAN ROELOFSEN
WeigleWilczek GmbH
73728 Esslingen
GERMANY
roelofsen@weiglewilzek.com

Abstract: - This paper examines several design options how to provide load balancing and fault tolerance in OSGi. The aim of this provision is to make OSGi applications more scalable and highly available.

Key-Words: - OSGi, load balancing, fault tolerance, scalability, high availability

1 Introduction
The OSGi Service Platform [2] (or just OSGi for short) has made its way to numerous applications and systems, ranging from embedded systems like in-vehicle software systems up to enterprise systems like popular Java EE application servers. However, Quality of Service (QoS) is an area where OSGi still requires some extensions and where this paper comes into play. In particular, we will focus on one aspect of QoS – high availability by means of load balancing and fault tolerance.

Consequently, in this paper we will explain some fundamentals of OSGi, load balancing and fault tolerance. We will also explain some common approaches to load balancing and fault tolerance, and how they can be implemented in OSGi.

2 OSGi
OSGi is a Java platform that has arisen in the context of embedded systems. The platform is freely available and constantly developed by the OSGi Alliance.

There are several commercial and non-commercial implementations of OSGi, including Eclipse Equinox [3], Apache Felix [4], Knopflerfish [5], and ProSyst’s mBedded server [11]. Well-known applications that are based on the platform include the Eclipse IDE and Apache Service Mix [12].

The core of OSGi is the OSGi Framework. This framework simplifies the development and deployment of extensible applications called bundles, by decoupling the bundle’s specification from its implementation. This means that a bundle is accessed by the framework through an interface, which is by definition separate from the bundle’s implementation. This separation enables changing the bundle’s implementation without changing the environment itself and other bundles.

The OSGi Framework makes it possible to run multiple applications simultaneously within a single Java Virtual Machine (JVM), by dividing applications into bundles that can be loaded at runtime and also removed. For communication within the JVM, the framework provides a service registry to register services, so that services can be found and used by other bundles.

Although OSGi was originally designed for embedded systems, it is finding more and more usage in enterprise systems. Recently, OSGi has been extended with support of remote services. This extension is aggregated under the name D-OSGi (Distributed OSGi) [13]. D-OSGi is important for
load balancing because it enables to distribute an application across more than one OSGi instance.

![Fig. 1. Horizontal scalability.](image1)

![Fig. 2. Vertical scalability.](image2)

3 Load Balancing

Load balancing is the ability of a system to distribute requests across multiple resources. Load balancing has the effect of providing a virtual resource pool, which unifies all resources. A request is sent to the virtual resource pool and then forwarded to a real resource. Resources can be, for example, servers or processes running on a server. Load balancing can be performed by hardware or software components [9, 10].

Load balancing can be used to eliminate bottlenecks in a system. If, for example, one resource is temporarily not available, a request can be processed by other resources from the virtual resource pool.

With load balancing, it is also possible to provide scalability. When requests are transmitted to an overloaded server, other servers can be added to the virtual resource pool. Without load balancing, an overloaded server, which cannot handle the request, has to be replaced by a more powerful server. This type of scaling by distributing the request to multiple servers is called horizontal scalability (see Fig. 1).

Alternatively, scalability can be provided by distributing the requests across multiple processes running on a single application server. In this case, the resources are the processes and not the servers. This type of scaling over multiple processes is called vertical scalability (see Fig. 2).

Load balancing algorithms are used for mapping requests to resources. These algorithms get access to information about the resources in the virtual resource pool and the incoming requests. With this information, a load balancing algorithm selects a resource, which can handle the request.

In practice, different load balancing algorithms are used for different tasks. All algorithms have the common goal of the effective resource utilization. Other goals of load balancing algorithms include a minimal response time and a maximal throughput. Some examples of load balancing algorithms are:

- **Round robin algorithm**: This algorithm was originally designed as a scheduling algorithm. It forwards incoming requests in a circular order to all available resources. The algorithm is easy to implement because it does not assign any priority to the resources.

- **Weight-based algorithm**: This algorithm can be used when one or more resources in the virtual resource pool handle requests differently than other resources, for example, faster ones. A weight is a priority, which is assigned to all resources. Resources with higher priorities are preferred at the request-to-resource mapping.

- **Dynamic algorithm**: The weight-based algorithm uses constant priorities for the resources. But it is also possible to set the priorities for the resources dynamically. Such an algorithm is called a dynamic algorithm. For example, a measurement of the priority could be the actual load of the resources.

- **Sticky algorithm**: This algorithm is viable for stateful applications only. In this mode of operation of an application, a request is sent to the same resource every time. The decision which of the resources handles the request will be made on the basis of a session ID.

In OSGi vertical scalability can be achieved with replicated bundles, which provide the same functionality. These bundles will vary in the symbolic bundle names only. A load balancing component implemented as an OSGi service can be used to distribute the request to the replicated bundles. Such a central load balancing component can also be used...
for other tasks, for example, monitoring of services or bundles to provide fault tolerance.

A common way for achieving horizontal scalability is to add an external load balancing component, which distributes the requests across the servers. This component can be implemented, for example, with DNS load balancing or web controllers.

Alternatively, an OSGi instance can be created to spread the client’s load over the replicated servers. This instance has to use D-OSGi methods in order to distribute the requests to services or bundles on the replicated servers.

4 Fault Tolerance
Fault tolerance is another important aspect of high availability. Load balancing only assures that a system can be used even though a part of the system is not available. But the question is what happens to the information, which is handled by a failed part? For example, how should the system react when a user request fails? Is that information lost or can it be restored? Is the system able to discover the failure or is the user just waiting for a response?

4.1 Replication
It is important to avoid single points of failure. A single point of failure represents a bottleneck in the system. If one of components in a system fails, the whole system will go down. Therefore, every critical part of the system should be fault tolerant. Otherwise, the single point of failure is just “shifted” to another component of the system.

To provide fault tolerance in a system, the first step to do is to make critical parts of the system redundant. Replication is one of the main techniques to achieve this goal. In addition to holding one or more copies of a state, there is often a requirement to share data between different replicas to keep the identical state. If a replica receives an update, all other replicas will have to be informed about that update. Techniques to keep the states of all the replicas identical are called replication techniques.

One such a technique is to use shared memory. This memory can be implemented by saving the state persistently in a file system or a database management system. For replicas on multiple servers, the state has to be available globally. This could be a database management system or a special software product like Terracotta [6].

Another replication technique is based on the primary. One primary replica is selected to accept write requests from clients and to publish state updates to the other backup replicas. When a backup replica failed, this replica will be discarded by the primary. When the primary failed, the request is taking over by another backup replica.

In OSGi we recommend to use shared memory. The service registry makes it simple to integrate a bundle, which offers services for storing and loading data. This bundle can preserve the information in memory, a file system or a database management system. Instead of using services, the communication can also be done by events.

4.2 Management Tool
The next step to do is to discover failures. This could be done by management tools.

A management tool can be an extra server or just a program on the server, where an application is running on. This tool can react when an exception comes up. It can also monitor resources on its own. For example, monitoring could be done by the management tool, which pings the application server in a specified period of time. This way the management tool can discover a failure even though the application never admits an exception. This would be the case when the application was crashed without the ability to admit an exception.

Another approach to monitoring is passive checking. In this approach, all application servers send pings in periodic intervals. The management tool is listening to all incoming pings. An application server, which does not send a ping, is considered to be failed.

Yet another approach to monitoring is based on a proxy. This proxy forwards all requests from the clients to the application servers. In this approach, the proxy has the opportunity to monitor the status of the requests and the replicas.

Once a failure has been discovered, the management tool can start a specified action to handle this failure. When an application or its server is crashed, it should be isolated. Otherwise, this failure can spread to other components in the system. Next, the failed component can be restarted and, if necessary, recovered. This could be done by the management tool itself. But often after a crash, there is information or unacknowledged request, which has to be handled as well.

The complexity of handling a failure depends on the mode in which the application is operating:

- Stateless: In this mode of operation, the application does not store any information on a request. All requests are independent from each other.
- Stateful: In this mode of operation, the application stores information on different requests in a session.

Handling a failure in a stateless application is easy because there are no dependencies between different requests. There is no loss of information and an
unacknowledged request can easily be repeated or redirected to another server by the load balancing component or the client itself. This implies that the unacknowledged request is stored until it is acknowledged.

The handling of a failure, however, becomes much more complex in case of a stateful application, where information on earlier requests has to be recovered (in order to be available to future requests), after the failure has occurred.

In OSGi a management tool can easily be implemented as a separate bundle. In addition to a ping-based approach to discovering failures, the management tool can use the life cycle functionality of the OSGi Framework. With this functionality, it becomes much easier to monitor other bundles and restart them in case of failures.

The proxy-based monitoring of replicated bundles or services is also feasible in OSGi. A service can act as a proxy, which provides access to a monitored bundle or service. Here the proxy has the capability to check, for example, the correctness of the results and the response time.

In addition to general application exceptions, the OSGi specific bundle exceptions could be used to discover failures. An alternative to bundle exceptions is the event functionality of OSGi. The management tool (implemented as a bundle) could consume special failure events, which are published by other bundles, instead of exceptions. This helps to prevent dependencies between the management tool and other components of the application.

After a failure has been discovered, the corresponding bundle has to be isolated. In OSGi a bundle can be isolated just by stopping it. If a bundle or service has a permanent failure, the bundle will have to be uninstalled and checked manually.

All the solutions above assume that all parts of the application are running in the same OSGi instance. For solutions with more than one instance, for example, in a horizontally scalable environment, D-OSGi can be used. But D-OSGi does not cover monitoring with the life cycle functionality; either events over different instances. As a possible solution to this problem, a management tool agent could be used. This agent could communicate over D-OSGi with a central server. But this solution would become much more difficult to implement.

5 Related Work
Ahn, Oh and Hong [1] proposed a proxy-based approach to fault tolerance in Service-Oriented Architecture (SOA) systems. They used a proxy, which wrapped a service object, intercepted service requests and routed the requests to the “best” service at runtime. To provide fault tolerance, the proxy monitored failures. When a failure was discovered, the proxy isolated the failed service and recovered it. This approach is also applicable to OSGi.

Papageorgiou [8] provided fault tolerance and scalability in the Virtual OSGi Framework. In particular, he presented a global OSGi Framework, which acted as a virtualization layer on the top of local OSGi Frameworks. The global OSGi Framework connected two or more local OSGi Frameworks running on different nodes, and thus could handle dynamic changes in the network.

Maragkos [7] described the replication and migration mechanisms of bundles in the Virtual OSGi Framework. These mechanisms can also be used to provide fault tolerance in OSGi.

6 Conclusion
In this paper, we have described different approaches to providing OSGi applications with high availability. In particular, we have examined several design alternatives how load balancing and fault tolerance can be implemented in OSGi.

7 Future Work
To demonstrate and test the approaches we have described, we are going to implement some of them in a demo application running on OSGi. The basic user scenario for our demo will be a vertically scalable environment with more than one application server. Each server will run in the OSGi Framework and offer a service for users from outside. Therefore, there should be a component (either internal or external) to handle load balancing of different servers. Furthermore, another component, which provides fault tolerance, will also be needed. The challenge will be to implement a highly available application by means of load balancing and fault tolerance in OSGi.

In addition to the demo application, we are going to examine whether the implementation works in real-world applications. For example, some issues might occur due to a particular JVM’s threading implementation. Also, vertical scalability from within a single OSGi instance (that runs on a single JVM) might not always help. So we need to conduct further experiments in order to address such issues.

Furthermore, checking vertical scalability of OSGi in cloud environments could also be an interesting topic for the future work.

References:
Journal of Information Science and Engineering
23 (2007), Nr. 5, S. 1379-1390


