International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:8, No:3, 2014

Shopping Cart System: Load Balancing and Fault Tolerance in the OSGi Service Platform

Irina Astrova, Arne Koschel, Thole Schneider, Johannes Westhuis, Jürgen Westerkamp

***Abstract*—**The main purpose of this paper was to find a simple solution for load balancing and fault tolerance in OSGi. The challenge was to implement a highly available web application such as a shopping cart system with load balancing and fault tolerance, without having to change the core of OSGi.

***Keywords*—**Fault tolerance, load balancing, OSGi, shopping cart system.

I. Introduction

THE OSGi (Open Services Gateway initiative) Service Platform [2] (or just OSGi for short) is a Java platform, which "delivers an open, common architecture to develop, deploy and manage services in a coordinated fashion." The platform is freely available and constantly developed by the OSGi Alliance. There are both commercial and non-commercial implementations of OSGi, including Eclipse Equinox, Apache Felix, Knopflerfish and ProSyst's mBedded Server. Although OSGi was originally designed for embedded systems, nowadays it finds more and more use in enterprise systems.

The core of the platform is the OSGi Framework, which helps to create loosely coupled, service-oriented, extensible applications called *bundles* that can be loaded and removed at runtime. The framework simplifies the development and deployment of bundles, by decoupling the bundle's specification from its implementation, i.e., a bundle is accessed through its 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 provides a service registry to register services, so that services can be found and used by other bundles. Also the framework provides methods and services for core functionalities such as life cycle management and security. In addition, there are many implemented services for other tasks such as event handling, logging and database access but not for load balancing and fault tolerance. As an attempt to fill a gap in this field, we present the implementation of a simple yet reasonable solution for load balancing and fault tolerance in OSGi. The solution is based on previous conceptual work from [3].

Irina Astrova is with Institute of Cybernetics, Tallinn University of Technology, Tallinn, Estonia (e-mail: irina@cs.ioc.ee).

Prof. A. Koschel, Thole Schneider, Johannes Westhuis, and Jürgen Westerkamp are with Faculty IV, Department of Computer Science, Hannover University of Applied Sciences and Arts, Hannover, Germany (e-mail: akoschel@acm.org).

The rest of the paper is organized as follows. Section II will demonstrate our own implementation of load balancing and fault tolerance in OSGi. The demo implemented a shopping cart system. Section III will give a short overview about related work. Sections IV and V will make a conclusion with a critical overview on the demo implementation.

II. Shopping Cart System

To demonstrate and test the design solutions proposed in our previous paper [3], we implemented some of them in a simple demo running in the OSGi Framework.

The demo resamples a web application, which provides a shopping cart system. The system should be 24/7 available to users. A user can log into the system, add items to the shopping cart, remove items from the shopping cart and logout of the system. After logout, the shopping cart is deleted, assuming that a transaction has completed. For simplicity, we implemented vertical load balancing.

Fig. 1 shows the demo components, including `Replica`, `Session Service`, `Db4oService`, `HTTP Server`, `HTTP Wrapper` and `Load Balancer`. All these components were implemented as OSGi bundles.

A. Replica

In the demo we used replication, which is a common approach to fault tolerance [6].

Fig. 2 shows the class diagram of a bundle `Replica`, which represents one instance of the web application. In the demo, all replicas had the same functionality and differed in their symbolic names only. Since the business logic represents the shopping cart system, a bundle `Replica` has the following methods:

- `login`: This method creates a new shopping cart. It also saves in the database a new session state with a given user name.
- `addItem`: This method adds a given item to the shopping cart. It returns the contents of the shopping cart.
- `removeItem`: This method removes a given item from the shopping cart. It returns the contents of the shopping cart, which can be empty.
- `logout`: This method deletes the session state from the database.
- `getCartContent`: This method returns the content of the shopping cart, which can be empty.

If a replica receives an update, all other replicas have to be informed about that update.

One approach is to keep a consistent state of the session among all the replicas is to use a shared persistent memory, which might be implemented by saving the session state persistently in a database. In addition, the session state has to be available globally to all the replicas.

In the demo, we used shared memory. OSGi's service registry makes it easy to integrate a bundle, which employs other services for storing the session state. This bundle can preserve the state information in a database.

Instead of using services, the communication can also be done through events. For example, a bundle `Replica` can send one of the following events to a bundle `Session Service`:

- REQUEST: This event asks for the current session state.
- SAVE: This event asks to save a given session state.
- REMOVE: This event asks to remove a given session state.

B. Session Service

Fig. 3 shows the class diagram of a bundle `Session Service`. The main task of this bundle is to distribute a consistent state of the session to all the (functional) replicas. Therefore, the bundle has to listen to all the events, which are sent by replicas, and reacts on them, by sending one of the following events:

- REPLY: This event contains the current state for a given session.
- NOTFOUND: This event notifies that no state is available for a given session.
- EXCEPTION: This event indicates that the saving of the session state failed.

To store the session state, db4o was used.

C. Db4oService

Db4o is an object-orientated database, which is primarily designed for embedded systems. We selected db4o because of its support of OSGi – specifically db4o is delivered with a bundle `Db4oService`, which can create a database, handle constraints and perform transactional CRUD (create, read, update and delete) operations. In other words, this bundle provides all the operations needed for storing the session state.

D. HTTP Server

Fig. 4 shows the class diagram of a bundle `HTTP Server`. This is a component, which accepts HTTP requests from outside the system. In the demo, the bundle is just listening for HTTP requests and redirects them to a bundle `HTTP Wrapper`.

E. HTTP Wrapper

Fig. 4 shows the class diagram of a bundle `HTTP Wrapper`. The task of this bundle is to perform the mapping of HTTP requests and responses from outside the system into OSGi service calls and vice versa. The main reason to insert this component was to make the demo extensible. With the bundle `HTTP Wrapper`, replicas have no dependency on the HTTP protocol. So it should be easy to integrate a new interface, e.g., a web service endpoint. In this case, only an extra bundle `HTTP Wrapper` needs to be added, which performs the mapping from web services to OSGi services.

In the demo, Pax Web [4] was used. It is an implementation of OSGi's `HttpService`. Pax Web supports servlets, filters, listeners and some other functionality. Important for the demo was the servlet support, because the bundle `HTTP Wrapper` depends on servlets. The following servlets were implemented within the demo:

- `IndexServlet`: This servlet represents a static index page. It returns a page with a form where the user has to enter the user name. On submission, the user name is given over as a parameter to the login page. Additionally, a cookie with the session ID of a type `java.util.UUID` is set up in the response from the `IndexServlet`.
- `ReplicaServlet`: This servlet represents all other pages. The constructor from the `Replica Servlet` expects the reference to the bundle `Load Balancer`. The `ReplicaServlet` stores all the parameters in a map. Then a method `doAction` from the bundle `Load Balancer` is called with the session ID, the name of the requested service and the map. As a return value, a map with the response parameters is expected. Based on them, the HTML response is generated, which presents the shopping cart of the user.

When the service from the bundle `Load Balancer` is present, the servlets can be registered. At first, the `IndexServlet` is registered under an alias `index` as shown below:

```
httpService.registerServlet("/index",       new
IndexServlet(), null, null);
```

After that, the `ReplicaServlet` is registered in a loop with different aliases for all other pages as shown below:

```
String[]  urls  =  {"login",  "add",  "remove",
"logout", "get"};
...
for (int i=0; i<this.urls.length; i++) {
httpService.registerServlet("/"+this.urls[i],
new ReplicaServlet (this.service), null, null);
}
```

When a request arrives, the `HttpService` looks up for an URL and calls a registered servlet, whose alias matches the requested URL. Then the selected servlet is started and the HTTP request is given over. To achieve fault tolerance, the `ReplicaServlet` catches exceptions from the bundles `Load Balancer` and `Replicas`. If an exception comes up, an individual failure page is sent to the user. This makes system failures transparent to the user. When the bundle `HTTP Wrapper` is stopped, all the servlets are unregistered.

F. Load Balancer

Since it would be very inconvenient to let users switch between replicas (especially in web applications), some kind of a dispatcher called *load balancer* is needed to allocate the incoming requests to the replicas [5].

Fig. 5 shows the class diagram of a bundle `Load Balancer`. The task of this bundle is to connect replicas with the system – specifically it forwards service requests from the bundle `HTTP Wrapper` to a bundle `Replica` and returns the response. The bundle `Load Balancer` consists of the three main classes: `LoadBalancer`, `Scheduler` and `Facade`.

To map service requests to one or more replicas, the bundle `Load Balancer` implements a class `LoadBalancer`, which tracks the state of all the service events from the replicas. This is done using OSGi's class `ServiceTracker`, which is registered with the interface name of a replica. If a replica is registered, the bundle `LoadBalancer` caches the reference and the corresponding object. When the service is unregistered, the reference and the object are removed from the cache. The cache was internally implemented as a hash map. To provide access to the cached objects and spread the requests over all the (functional) replicas, the class `LoadBalancer` uses a scheduler implemented in a class `Scheduler`. The scheduler has access to the cache and is updated by the class `LoadBalancer` whenever a replica is registered or unregistered. In the demo, the round robin algorithm was implemented in a class `RoundRobin` as the scheduler. This class uses a simple counter to schedule the access to the objects in the cache. When an object is requested but no replica is available, the scheduler throws a `ReplicasUnvailableException`.

To provide access to replicas from the bundle `HTTP Wrapper`, the class `LoadBalancer` registers a service with an instance of a class `Facade`. This class assures that every service request is redirected to another replica. For this reason, the class `Facade` provides a method `doAction` to send service requests to the replicas. For each call of this method, the class `Facade` gets a replica from an instance of the class `LoadBalancer`, interprets the service request and calls the corresponding method of the replica (i.e., `login`, `logout`, `addItem`, `delItem` or `getCartContent`).

With the class `Facade`, it also becomes possible to keep the core of the bundle `Load Balancer` independent from the managed service. Any kind of service can be scheduled and load balanced by this class. Only the class `Facade` has to be customized to the managed service. A simplified communication among the bundles `HTTP Wrapper`, `Replicas` and `Load Balancer` is illustrated in Fig. 6.

G. Testing

To test the demo, we used the life cycle management functionality from the OSGi Framework. At first, a replica was stopped within the running system. The load balancer reacted to this change, by deleting the replica from the list. The system continued working. After restart of the replica, it was automatically added to the list.

Next, all the replicas were stopped. The load balancer threw an exception, which then was displayed to the user. After restart of a replica, the system was working again. The user session was not affected, even with the complete loss of all of the replicas.

Finally, we tested that all the exceptions and errors were transferred to the client in case of failures. After a failure has been detected, the failed bundle has to be isolated. Otherwise, the failure can spread to the other bundles in the system. In OSGi a bundle can be isolated by stopping it. Later the bundle can be restarted and recovered if needed. If the bundle has a permanent failure, the bundle has to be uninstalled and checked manually.

III. Related Work

Ahn, Oh and Hong [1] proposed a proxy-based fault tolerant approach to OSGi. The authors used a proxy, which wraps a service object, intercepts service requests and routes the requests to the best service at runtime. To provide reliability, the proxy monitors faults. When a fault is detected, the proxy recovers and isolates the failed services.

Another approach to providing high availability and fault tolerance is the Virtual OSGi Framework. Papageorgiou [8] presented a global OSGi Framework, which acts as a virtualization layer on top of local OSGi Frameworks. The global OSGi Framework connects two or more local OSGi Frameworks running on different nodes and can handle dynamic changes of the network. Maragkos [7] described the replication and migration mechanisms of bundles in the Virtual OSGi Framework. These features can be used to provide high availability and fault tolerance.

Ahn, Oh and Hong's approach assumes that all the parts of the system are running in the same OSGi instance. For solutions with more than one OSGi instance, the Virtual OSGi Framework should be used. However, the Virtual OSGi Framework requires modification of the OSGi Framework, thus breaking compatibility with already existing bundles. Moreover, the Virtual OSGi Framework does not cover monitoring with the life cycle management over different framework instances. A possible solution is to use an individual management agent for every framework. This agent could communicate over the Virtual OSGi Framework with a central management server. But such a solution would become much more difficult to implement and therefore, is not recommended.

IV. Conclusion

This paper presented a simple solution for vertical load balancing and fault tolerance, and demonstrated how these concepts could be implemented in OSGi, without having to change the OSGi Framework. The demo implementation showed that the OSGi Framework offers many possibilities to achieve vertical load balancing and fault tolerance. Even though it still has open issues and spaces for possible improvements, the demo implementation gives a closer look at how these important goals can be reached.

It should be noted that the demo implementation is extensible through many possible input sources and different possible applications, which can run in OSGi.

V. Future Work

Due to the time limitations, not all the solutions proposed in our previous paper [3] were implemented – specifically horizontal load balancing was omitted.

A. Load Balancing

Since the performance of one application server is normally not enough to handle all the requests for a frequently used web application such as the shopping cart system, the method of choice would usually be horizontal load balancing. Also horizontal load balancing enables to build a fault tolerant system. By contrast, vertical load balancing relies on a single application server and therefore, it can offer no tolerance against hardware failures.

In the demo implementation, all software failures were reported to the client. A useful extension would be to implement a transparent failover so that the load balancer could react on the failures of the replicas and distribute service requests to the functional ones.

B. Fault Tolerance

Moreover, the demo implementation left some single points of failure. For example, if the bundle `Db4oService` fails, the session state cannot be saved and the replicas cannot work either. Therefore, it is important to prevent single points of failure. A single point of failure indicates a bottleneck in the system. If one of the components fails, the whole system goes down. Therefore, it is significant that every critical component of the system has to be highly available and fault tolerant. Otherwise, the single point of failure is just shifted to another component of the system. The replication of the bundle `Db4oService` is an easy task, because it is loosely coupled with `Replicas` and reacts on events only. If a second database were started, this bundle would also receive all the events and thus, could store the session state.

A similar problem exists for other bundles: `Load Balancer`, `HTTP Wrapper` and `HTTP Server`. These bundles are single points of failure too. But again they are easy to replicate. For example, we can use a management bundle, which controls the life cycle and status of the bundles and restart them if needed. A management tool could be an extra application server or just a program on the server where the application is running on. This tool could react when an exception comes up but it can also monitor resources on its own. For example, monitoring could be done by a management tool, which pings the server in periodic intervals. This way the tool can discover a failure even though the system never admits an exception. This would be the case where the system crashes without the ability to signal an exception.

Another kind of monitoring is passive checking. In this case, all application servers send pings in periodic intervals. A central management server is listening to all the incoming pings. An application server, which does not send a ping, is in a failure condition. Also a proxy-based monitoring approach is possible. The proxy forwards all the requests from clients to the application servers. In this central position the proxy has the opportunity to monitor the status of the service requests and the replicas.

In OSGi a management tool can be easily integrated as a separate bundle. In addition to the use of a ping method to discover failures, it can use the life cycle management functionality from 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 replicas is also possible in OSGi. A service can act as a proxy, which provides access to a monitored bundle or service. This way the proxy has the ability to check, e.g., the correctness of the results and the response time.

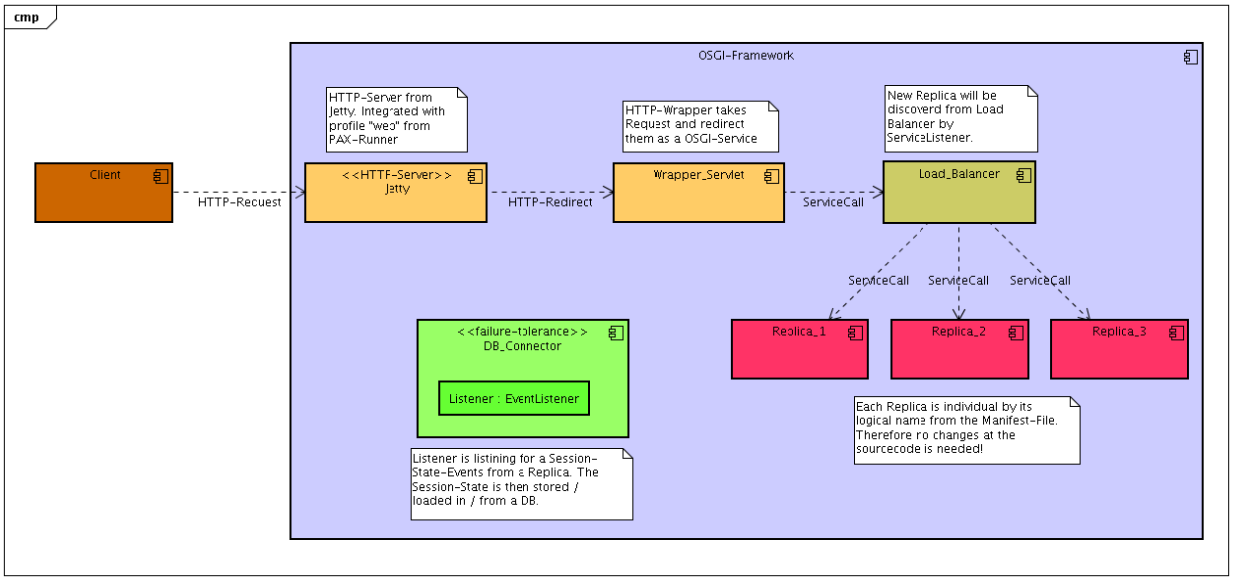

Fig. 1 Overview of shopping cart system

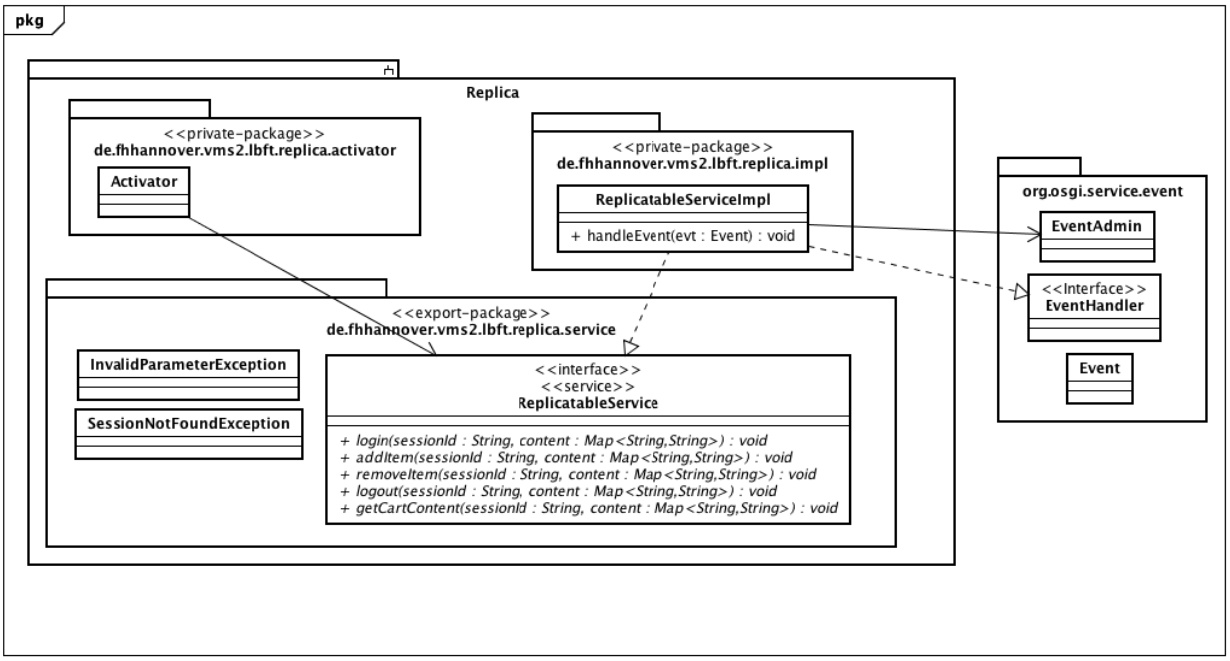

Fig. 2 Replica

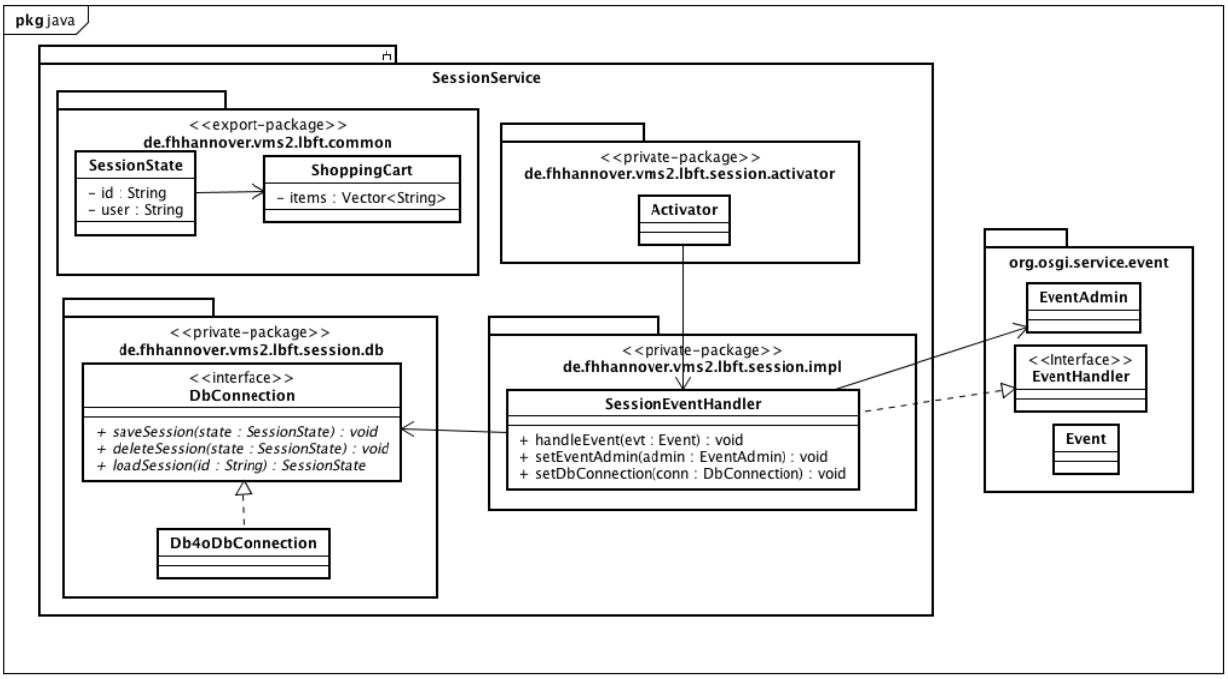

Fig. 3 Session Service

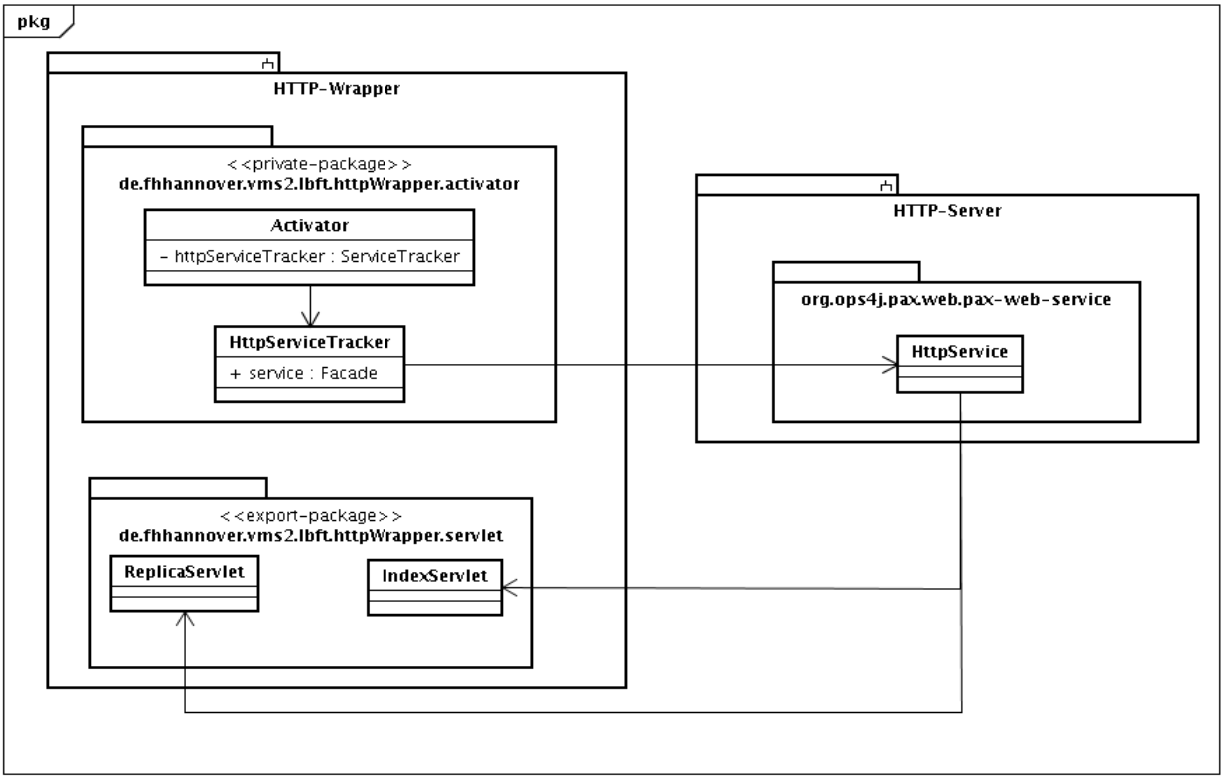

Fig. 4 HTTP Server and HTTP Wrapper

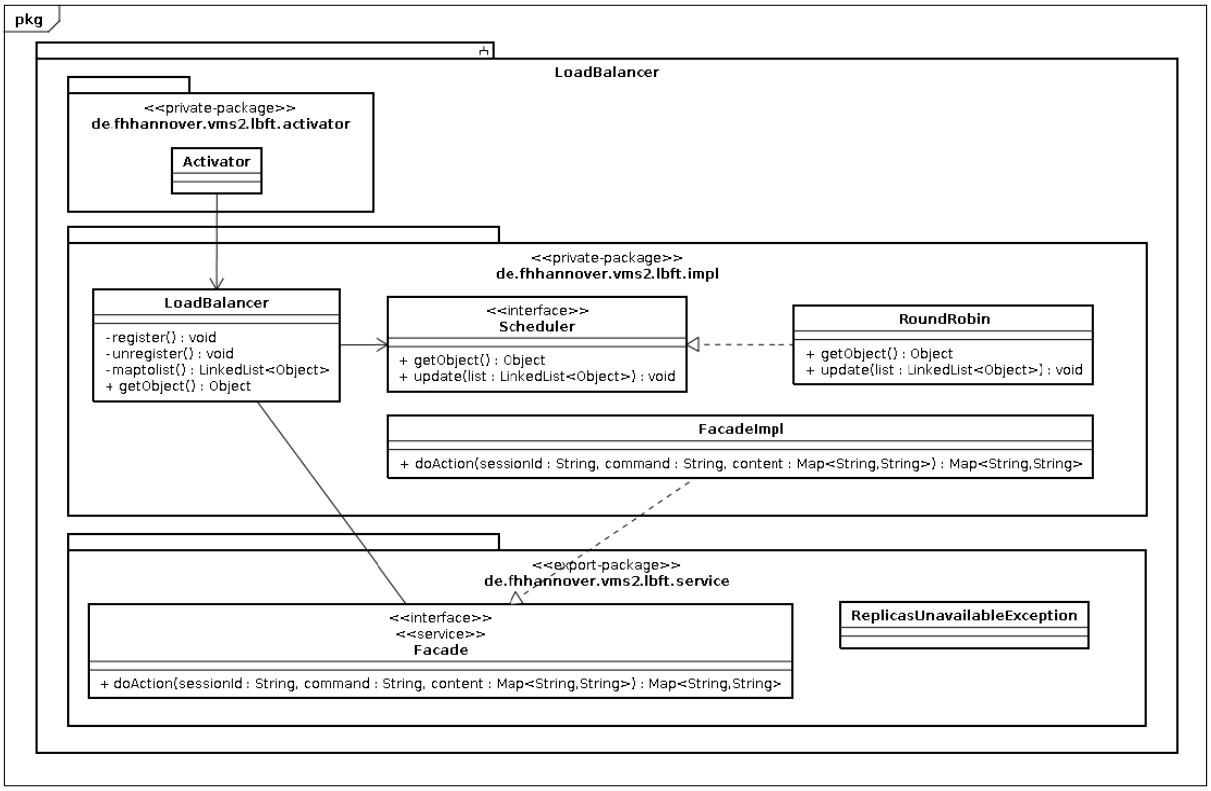

Fig. 5 Load Balancer

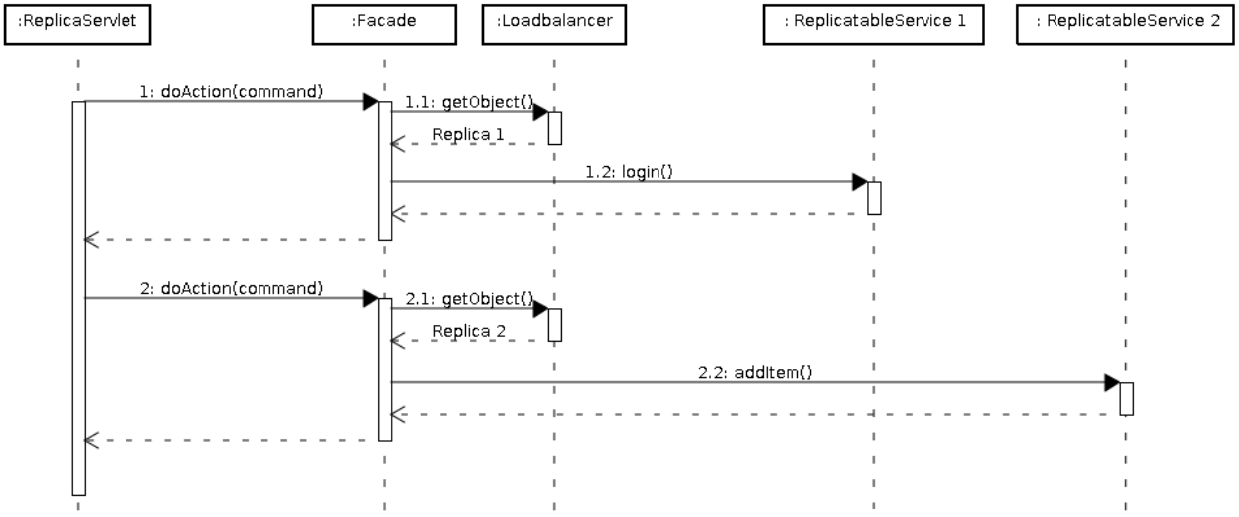

Fig. 6 Communication among Load Balancer, HTTP Wrapper and Replicas

Acknowledgment

Irina Astrova's work was supported by the Estonian Centre of Excellence in Computer Science (EXCS) funded mainly by the European Regional Development Fund (ERDF). Irina Astrova's work was also supported by the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12.

References

[1] Ahn, H., Oh, H., Hong, J.: Towards Reliable OSGi Operating Framework and Applications. In: Journal of Information Science and Engineering (2007), Nr. 5, 1379-1390.
[2] OSGi Alliance: OSGi – The Dynamic Module System for Java. Last accessed: January 2014, http://www.osgi.org.
[3] Koschel, A., Schneider, T., Westhuis, J., Westerkamp, J., Astrova, I., Roelofsen, R.: Providing Load Balancing and Fault Tolerance in the OSGi Service Platform. In: Mathematical Methods and Techniques in Engineering & Environmental Science (2011), 426-430.
[4] Comunity, Pax-Runner: Pax-Runner. online. http://wiki.ops4j.org /display/paxrunner/Pax+Runner.
[5] Schlossnagle, T.: Scalable Internet Architectures. Sams Indianapolis, IN, USA, 2006.
[6] Torrao, C.: Fault Tolerance in the OSGi Service Platform. Last accessed: January 2014, http://www.gsd.inesc-id.pt/~ler/reports/ carlostorrao-midterm.pdf.
[7] Maragkos, D.: Replication and Migration of OSGi Bundles in the Virtual OSGi Framework. Last accessed: January 2014, http://e-collection.ethbib.ethz.ch/eserv/eth: 30549/eth-30549-01.pdf.
[8] Papageorgiou, D.: The Virtual OSGi Framework. 2008.