Fault tolerance and load balancing on the example of GlassFish

Arne Koschel, Michael Heine, Lars Knemeyer, and Irina Astrova

Abstract**—**Distributed systems are steadily gaining significance in today's IT landscape. They are increasing in size and complexity, so is their demand on high availability. While managing distributed systems, there is a major issue to be addressed: How do we manage to keep high availability even in a case where one or more parts of the system fail? To address this issue, a distributed system should efficiently balance the load that the application has to handle and should be provided with a fault tolerance mechanism. This paper describes the principles of fault tolerance and load balancing as well as their practical implementation on the example of GlassFish.

Keywords**—**Apache HTTP Server, Balance, GlassFish, fault tolerance, load balancing.

I. Introduction

FAULT tolerance and load balancing are of great importance for high availability. High availability requires 24-hour per day, 7-day per week system accessibility.

A. Fault Tolerance

Generally, the term fault tolerance means that a system is able to keep operating even if some parts of the system fail. Fault tolerance is often needed when it comes to vital systems, e.g., the engines of an airliner. Although a fault tolerant system can compensate the failure of one or more parts, its performance might be compromised; e.g., an airliner would still be able to land with only one engine working, but it might not be able to take off again.

A common approach to fault tolerance is to distribute the system's software among one or more redundant physical servers. Thus, if the primary server fails, one of the parallel running servers can take over and handle all further requests. Although a request that is processed by the primary server might get lost in the instant where the server fails, all further requests will be handled properly. In such a scenario, each redundant server would cause additional costs, but will not increase the performance of the entire system. Therefore, in practice redundant systems are often used to enhance the performance in the first place and serve as a failover only if needed.

Since it would be very inconvenient to let the user switch between the individual servers (especially for web applications), some kind of a dispatcher so-called a *load balancer* is needed to allocate the incoming requests to the servers.

B. Load Balancing

Load balancing describes the spreading of system's load (e.g., the incoming requests) among multiple server processes. Thereby the server processes might either be hosted on a single physical server or be distributed over multiple physical servers (one server per process). If all the processes are running on one physical server, this is called *vertical scaling*. In a case where each server process is hosted on a dedicated physical server, this is called *horizontal scaling*.

No matter which scaling is used, the motivation for load balancing is: avoiding (scalability) bottlenecks, achieving optimal resource utilization, avoiding overload, and minimizing response time [1]. Since the performance of one single server can be not good enough to handle all the incoming requests for a frequently used system (e.g., a web application), the scaling method of choice would usually be horizontal scaling. In addition, horizontal scaling enables to build an efficient and fault tolerant system. By contrast, vertical scaling relies on a single physical server and thus, it can provide no tolerance against hardware failures.

To make the distribution of the system's software transparent to the user, the server processes are merged logically to a so-called *cluster*. This cluster is accessed through a load balancer, which appears as a single server process to the user. The load balancer itself is not part of the cluster. Rather, it runs on its own dedicated server. For the load balancer to be able to pass the incoming requests to the instances in the cluster, the IP addresses and port numbers of all server processes should be registered in the load balancer. Once an incoming request has arrived, the load balancer routes the request to a server process in the cluster. A particular server process is selected on the basis of the used load-balancing algorithm (e.g., round-robin, random, weight-based or dynamic/pending request counting).

No matter which algorithm is used, the load balancer also needs to be able to detect whether a server process is running or not. If the server process is down, the request should automatically be redirected to the next process in the cluster according to the used load-balancing algorithm.

The load balancer should also provide support for stickiness (e.g., cookie-based or URL encoding). With stickiness, the load balancer will always route all requests coming from a particular user to the same server process as the first request.

Prof. A. Koschel, Michael Heine, and Lars Knemeyer are with Faculty IV, Department of Computer Science, Hannover University of Applied Sciences and Arts, Hannover, Germany (e-mail: akoschel@acm.org).

I. Astrova is with InVision Software OÜ, Tallinn, Estonia (e-mail: irinaastrova@yahoo.com).

This enables the software hosted by the server process to keep track of the user's actions.

II. GLASSFISH

GlassFish [2] is one of the most commonly used application servers in the field of Java applications. GlassFish uses the following terminology:

Instance: A server process that hosts the application.

Node: A physical machine hosting the GlassFish software that runs instances.

Cluster: A logical component that contains all instances on all nodes making up the cluster.

A. Fault Tolerance

GlassFish is based on a Domain Administration Architecture (DAA). This architecture enables to manage the whole cluster as if it were a single instance. The setup and configuration of the cluster with all the contained nodes and instances are done on a Domain Administration Server (DAS). The DAS is a server process. It is commonly hosted on a dedicated physical machine (see Fig. 1).

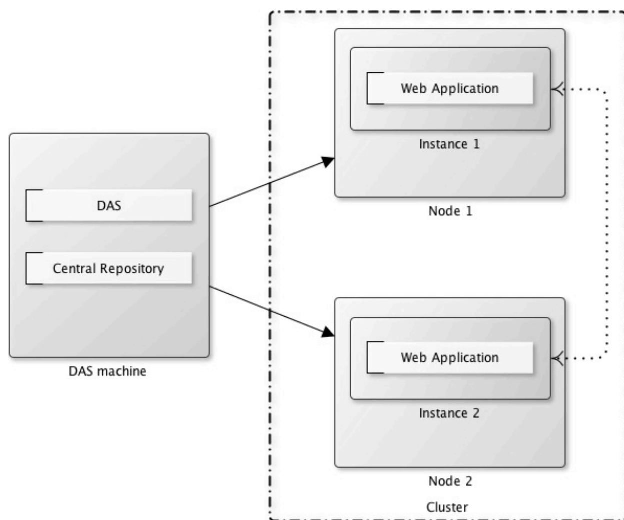

Fig. 1. GlassFish cluster

The DAS can also host web applications itself, which is usually done during the development (see Fig. 2).

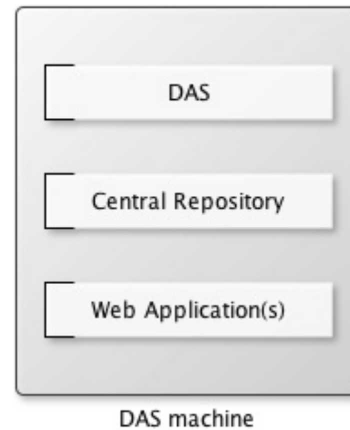

Fig. 2. Domain Administration Server (DAS)

To set up a new cluster, what is needed are a DAS (residing on its own physical machine) and one physical machine for each node in the cluster. Since the nodes should be able to run the GlassFish software, the operating system of choice for the nodes should be UNIX or Windows. For the DAS to be able to connect to the nodes, the machines should be accessible through either SSH or DCOM. Furthermore, all nodes have to be part of the same subnet as the DAS because the instances in the cluster need to communicate with one another via UDP multicasts. The setup and administration procedure can be done either from the command line interface or from the GlassFish administration console of the DAS.

At runtime, the DAS is used to manage the instances and acts as a central repository for all domain specific information (e.g. configuration information, resources and applications). If a new instance is added to the cluster, this instance will receive all necessary information from the DAS and cache it locally. Thus, once an instance has been failed, it can be reintegrated into the existing cluster without the DAS using the cached domain information.

Although a cluster can keep working properly without the DAS, it is common practice to implement a failover or at least a recovery strategy for the DAS as well. There are three basic approaches to this [3]. One is to periodically create backups of the domain data on the central repository and recreate the DAS on another GlassFish installation directory. Another approach is to periodically create backups of the whole GlassFish installation directory (including the domain root directory) and transfer it to a new host that inherits the network identity from the former host. Yet another approach is to use a hardware-based high availability solution for the DAS that automatically brings up a backup system with exactly the same configuration as the primary system where the original DAS fails.

In a cluster, fault tolerance is achieved through session replication. This means that the complete session state (including an HTTP session, EJB data and sign-on information) is replicated and stored beyond the instance that is actually handling the particular session's requests. Before GlassFish version 3, the session state data could be saved in a

ISBN: 978-1-61804-215-6

Highly Available Database (HADB). Since GlassFish version 3, this approach has been replaced by session replication. Session replication has three main advantages: (1) it is much easier to manage (since it is automatically configured by the DAS), (2) the load balancer does not need to know anything about a failover mechanism in the cluster, and (3) there is no more need for a failover of the HADB.

Session replication does not store all the session state data at one point. Rather, it distributes the replicated session information of one instance among the other instances in the cluster. A hash algorithm is used to determine which instance will store the state of a particular session. Thus in a cluster with three instances, a session S1 on an instance I1 might be replicated to an instance I2, while a session S2 from an instance I1 might be replicated to an instance I3. Fig. 3 illustrates such a scenario.

The hash algorithm can also be used to determine which instance is storing the replicated data of a particular session. This becomes important if the instance that has actually handled the session fails and the load balancer redirects the session's requests to an instance that does not have any information about the session's state. Based on Figure 3, the following scenario is possible. I1 handles S2 and replicates the session's state to I3. When I1 fails, the load balancer redirects the further requests to I2. Since I2 does not know the state of S2, it uses the hash algorithm to identify I3 as the host of the replicated session data. I2 obtains the state of S2 from I3. I3 deletes its copy of the transferred session state. I2 determines the new replication target (using the hash algorithm again) and handles the requests.

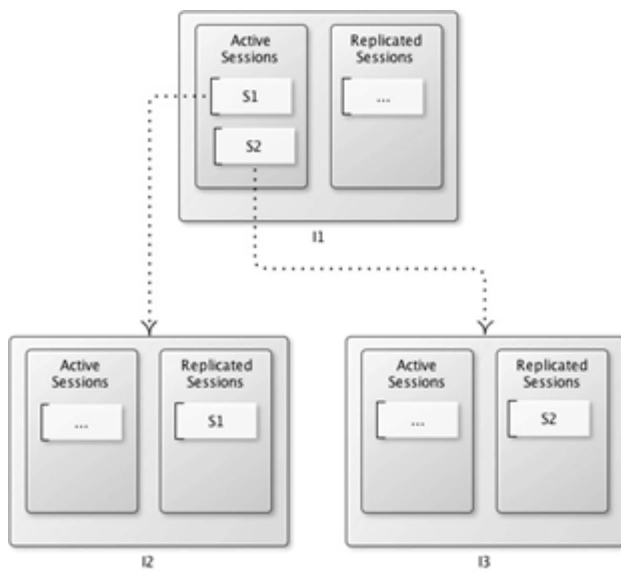

Fig. 3. Session replication

B. *Load Balancing*

GlassFish provides nearly all the functionality needed to set up a highly available and fault tolerant environment for web applications. The missing components are those that are not directly associated with the application server. Looking at the logical communication flow, the missing components are, on the one hand, those that are arranged behind the application servers (e.g., a database) and, on the other hand, those that are in front of the application servers (e.g., a load balancer).

Fig. 4 illustrates a cluster that is accessed through a load balancer and has a database connected on the backend. This cluster is an abstract representation of three instances.

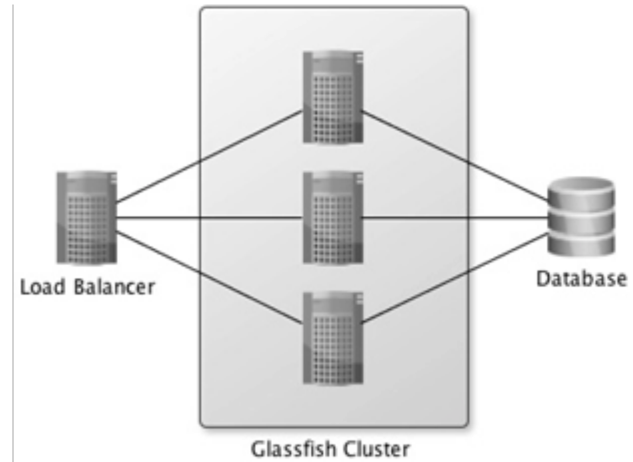

Fig. 4. GlassFish

III. EXAMPLE

To practically apply the concepts of fault tolerance and load balancing, we at first created a local cluster with two instances. By "local", we mean that all the instances as well as the DAS were running on the same machine. Each instance had three ports: a port for HTTP connections to the server, a port for HTTPS connections to the server and an admin configuration port (which became more important later when the cluster was distributed).

After setting up the local cluster, we used Balance [4] as a load balancer to enable load balancing in the cluster. The Balance was started from the command line. The resulting setup is shown in Fig. 5.

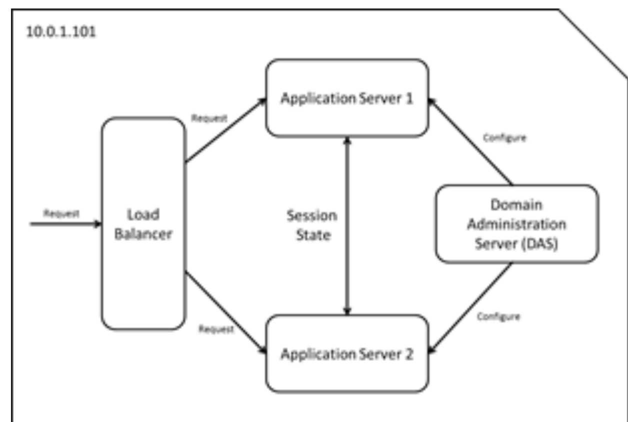

Fig. 5. Local cluster setup

We tested the local cluster by deploying a sample web

application onto the cluster and simulating different failover scenarios. The sample web application was a Java Server Faces (JSF) application. It contained three counters: one was manually stored in the HTTP session, another counter was stored inside a session scoped by JSF managed bean, and yet another counter was stored inside a session scoped by stateful session bean.

Our next step was to distribute the cluster over multiple physical nodes. The Balance and the DAS were running on one machine, whereas the other two machines hosted the two instances in the cluster (one machine per instance). The resulting setup is shown in Fig. 6. In this setup, the load balancer was connected to two servers (`10.0.1.102` and `10.0.1.103`); both were listening on port 8080 and hosting the sample web application.

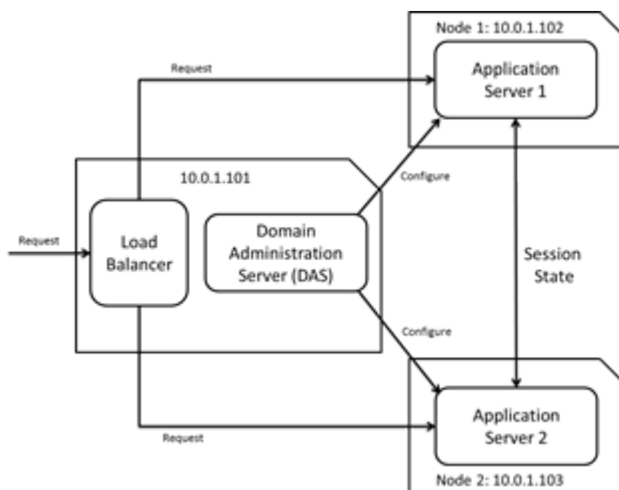

Fig. 6. Distributed cluster setup

We tested the distributed cluster against the sample web application again and particularly checked the behavior of the cluster in different failover scenarios.

To conclude the example, we critically looked at the previous setup to spot possible weaknesses of the configuration and to make suggestions for possible improvements. We identified the Balance and the DAS as single points of failure. If the Balance fails, no request will reach the instances. However, a failure of the DAS will not result in a complete failure of the cluster as the cluster can keep on running autonomously without the DAS. To solve the first problem, we suggest to have multiple Balances so that one Balance can take over if the other one fails.

IV. Conclusion and Future Work

During the tests on both the local and distributed cluster, it became apparent that in some failover scenarios the Balance redirects request from a single client to different instances in the cluster. Therefore, in the future we are going to replace the Balance with the Apache HTTP Server [5]. The biggest advantage of the Apache HTTP Server is its support for stickiness.

Although the Apache HTTP Server is more sophisticated and potent than the Balance, it is more difficult to configure. To use the Apache HTTP Server as a load balancer, it should be installed with the following modules: `mod_proxy`, `mod_proxy_http`, `mod_proxy_ajp` and `mod_proxy_balancer` [6]. After the installation, the Apache HTTP Server as well as the DAS need further configuration. In the Apache HTTP Server configuration, the lines shown in Fig. 7 should be added. The placeholder `[Web-App]` stands for the name of the hosted web application, whereas the placeholders `[Instance 1]` and `[Instance 2]` represent names of the two instances that can be chosen freely. In the DAS configuration, a new property called `INSTANCE` should be added to the cluster.

```
<Proxy balancer://myCluster>
   BalancerMember
       http://10.0.1.102:[Port]
       route=[Instance 1]
   BalancerMember
       http://10.0.1.103:[Port]
       route=[Instance 2]
</Proxy>

ProxyPreserveHost On
ProxyPass                    /[Web-App]
balancer://myCluster -
/[Web-App] stickysession=JSESSIONID
<Location /balancer-manager>
    SetHandler balancer-manager
    Order Deny,Allow
    Allow from all
</Location>
```

Fig. 7. Apache HTTP Server configuration

Acknowledgment

We would like to thank Mats Lennart Henke from Hannover University of Applied Sciences and Arts, Hannover, Germany, for his help in preparing this paper.

References

[1] A. Koschel. High availability, fault tolerance, clustering concepts and sample approaches with os/hardware, corba and java ee/j2ee products. paul.inform.fh-hannover.de: /home/daten/skripte/skripte/master/qualitaet_verteilter_systeme/ss2012/ Vorlesung/05_QVS_SoSe12_j2-co_lb-ft.pdf, 2012.

[2] Oracle Corporation. Glassfish - open source application server. http://glassfish.java.net/

[3] Oracle Corporation. GlassFish Server Open Source Edition High Availability Administration Guide, Release 3.1.2. Oracle Corporation, Redwood City, CA 94065.

[4] Inlab Software GmbH. Balance. https://www.inlab.de/balance.html

[5] Apache Software Foundation. The apache http server project. http://httpd.apache.org/

[6] Apache Software Foundation. Apache module mod proxy balancer. http://httpd.apache.org/ docs/2.2/mod/mod_proxy_balancer.html