The Clustering Concept of the SAP® Web Application Server

Torben Schreiter
Hasso-Plattner-Institute for Software Engineering,
University of Potsdam, Germany
torben.schreiter@hpi.uni-potsdam.de

June 2005

Abstract

The Java 2 Enterprise Edition (J2EE) has become a widely-used platform for the development of Enterprise Applications, which are often responsible for many business processes within a company. So, Enterprise Applications have to be more reliable than any other kind of software.

The concept of combining many physical machines into one cluster offers this reliability in two important ways: Failover mechanisms guarantee a maximum degree of availability to the users and load balancing results in scalability of the applications.

This paper aims to describe the specific architecture of a cluster of SAP® Web Application Servers, concentrating on their Java-personality. The cluster architecture explained provides a sophisticated technological basis for the efficient execution of Enterprise Applications.

Keywords: Clustering, J2EE, SAP Web AS, Failover, Load balancing, SDM

1 Introduction

A clustering architecture, in general, is motivated by two primary aims. These are *high availability* and *scalability* of the software system. Due to this, the applications have to be highly distributed among a number of cluster nodes. The whole distribution should be transparent to the users of the services [AK2].

Typically, these services are mission-critical, which means that they have to be highly available. High availability correlates with fail-safeness. To provide fail-safe services, the relevant components of the system need to be kept redundant. Additionally, fault tolerance (*failover*) mechanisms have to be implemented in order to prevent any loss of transient data. So, we want the work left open by a crashed node to be taken over by another cluster node. The processing cluster node should return the expected result to the user, leaving him unaware of the incident as such.

Scalability means the ability to adapt to an increasing number of requests (in a certain period of time) in order to minimize the response times for the users requesting the services. Mostly, this is achieved by adding more machines to the cluster. A basic requirement is an effective *load balancing* mechanism, which is responsible for a fair delivery of requests to the connected cluster nodes. The total workload should be evenly distributed between the cluster nodes and the integration of new machines just as the disappearance[1] of Application Servers has to be handled by the load balancer(s) as well.

To SAP, the development of distributed

[1]There are different possibilities for an Application Server or whole cluster node to 'disappear'. It can disappear e.g. in consequence of a failure but it can also disappear because of maintenance concerns. This supported facility last mentioned is called '*hot-swapping*'.

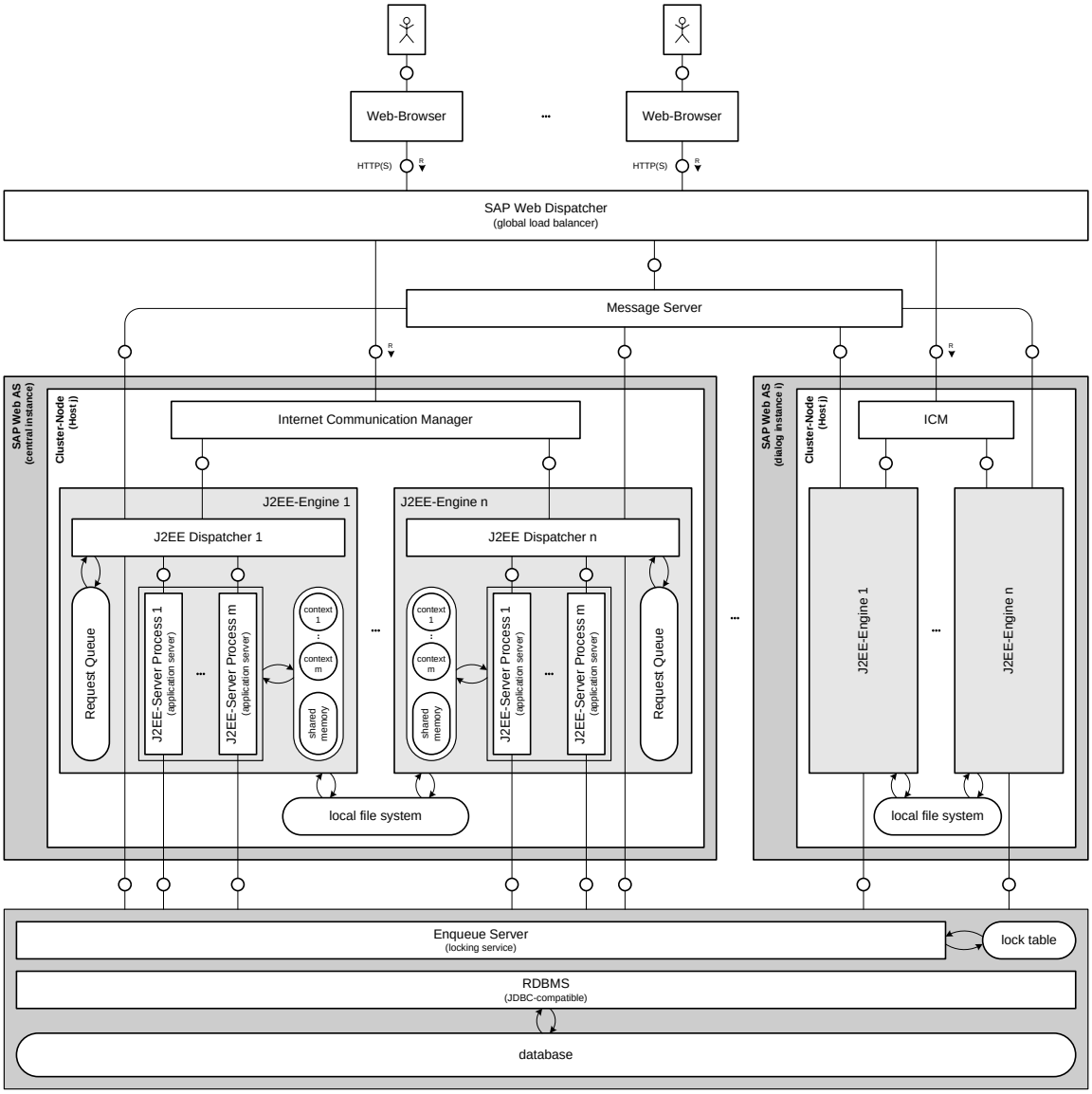

Figure 1: Structure of an operational SAP Web Application Server cluster

high-performance ERP[2] software systems is nothing new. In fact, the SAP Web Application Server (SAP Web AS) is more a SAP R/3-Application Server extended by a Java-personality than an entirely new designed system. Whilst implementing this Java-personality, SAP could revert to experiences that were gained during the development of the R/3-architecture.

In the first part of this article, the structure of a cluster of SAP Web AS will be explained. Part two and three deal with detailed explanations of the load balancing and failover mechanisms within a SAP Web AS cluster. Finally, several aspects concerning the *Software Deployment Manager* (SDM) as a central cluster component responsible for the integrity of all deployed software in the cluster will be examined in the fourth part.

[2]abbreviation for 'Enterprise Resource Planning'

2 Clustering Architecture of the SAP Web Application Server

2.1 Structure of a cluster

Figure 1 shows the general structure of a cluster of SAP Web Application Servers. A single central instance of the SAP Web AS as well as multiple dialog instances can be seen in the middle. Every Web Application Server includes an Internet Communication Manager (ICM) and one or more J2EE-Engines.[3] The ICM is mainly responsible for the recognition of the type of request. It decides whether the request has to be handed over to the Java- or ABAP-personality.[4] A *J2EE-Engine*, essentially, comprises one dispatcher and several J2EE-Server Application Server Processes. All J2EE-Engines are connected to the central database as well as to the Enqueue Server (see section 2.2.2). The central instance also contains the server component of the Software Deployment Manager (not illustrated here, see section 5).

The *SAP Web Dispatcher* is situated above the cluster nodes. It receives all incoming requests from the users via HTTP or HTTPS using SSL and forwards them to a selected cluster node (see section 3.1).

The major components of the cluster will be discussed in the following chapters.

2.2 SAP Central Services

The SAP Central Services consist of the *Message Server* and the *Enqueue Server*. Often, these services run on the central instance, but there are certain possibilities to, at least, source out the Enqueue Server to a dedicated machine with regard to fault tolerance. These aspects will be explained below.

The database system will not be explained in detail since the SAP Web Application Server abstracts from the specific third-party database solution used.

2.2.1 Message Server

The Message Server is responsible for all communication (e.g. events, data transfer) within the cluster. Thus, it provides a powerful API[5], which enables the cluster components to communicate with each other.

The concept of using a Message Server to accomplish the cluster communication between the Java-nodes was first implemented in the release 6.40 of the SAP Web AS. The earlier releases implemented a point-to-point communication between the Java-nodes instead of a *star-architecture* as in 6.40. [WASJ6.40] The point-to-point approach is problematic, because the cluster has to handle $n * (n-1)/2$ connections (where n is the total number of nodes). In other words, every node has to hold one connection to each of the other nodes. If the cluster grows, the number of connections rises nearly quadratically. This of course affects the network performance and, consequently, the cluster performance. In contrast to this, the star-architecture only needs n connections altogether: One from every node to the central Message Server.

However, since the Message Server exists only once per cluster it might prove to be a possible bottleneck for the overall performance of the whole cluster, but tests revealed that clusters of 50 and more nodes pose no problem for this architecture [WASJ6.40].

The abovementioned API helps to accomplish an adequate performance of the cluster communication by abstracting from the concrete form of communication. If e.g. a node wants to send a large amount of data to another node, it would slow down the Message Server because all traffic would have to pass the Message Server as an intermediary first

[3] The SAP Web AS certainly consists of more than the components shown. However, this article focuses on the Java-personality of the Web AS and therefore only the relevant components related to this context are mentioned.

[4] Since this paper focuses on the Java-personality, the structure of the ABAP-personality is not shown in Figure 1.

[5] Application Programming Interface

until it is redirected to its destination (*Message Server communication*). To avoid this, the API - transparently - decides to send messages of a certain size upwards through a temporary point-to-point connection (via sockets) directly to the destination instead of using the persistent connection to the Message Server (*lazy communication*). [JPWAS]

Additionally, the Message Server collects and stores information about the whole cluster and its components such as the available J2EE-Dispatcher, J2EE-AS Processes, the Web Dispatcher, etc.

2.2.2 Enqueue Server

As a part of the Central Services, the Enqueue Server merely exists once per cluster, too. It provides the ability to synchronize access to common cluster resources.

When a cluster node wants to use a shared resource such as the central database it has to request a lock for this resource at first. If the resource is available, the lock is granted by the Enqueue Server. This means, that the Enqueue Server appends an entry with information about the granted lock to the *lock table*. After receiving the confirmation from the Enqueue Server, the cluster node can access the resource. When the node has finished its operation, it has to release the lock again.

The Enqueue Server, generally, locks so-called 'logical objects'. Even though most of these objects are related to the database (e.g. table rows or cells), the whole locking concept realized by the Enqueue Server is not related to the locking mechanisms provided by the vendor of the database system.

2.3 Expanding the cluster

When the demand for the services offered increases, it is desired to be able to improve the performance of the cluster by expanding it by additional machines. Relatively little effort is needed to accomplish the integration of new cluster nodes. Basically, two steps affect the integration:

- integration into the load balancing mechanisms of the cluster
- synchronization of all deployed applications

These steps are automated, which means that often it is enough to specify the correct address of the Message Server and configure several other initialization settings in order to integrate the new cluster node.

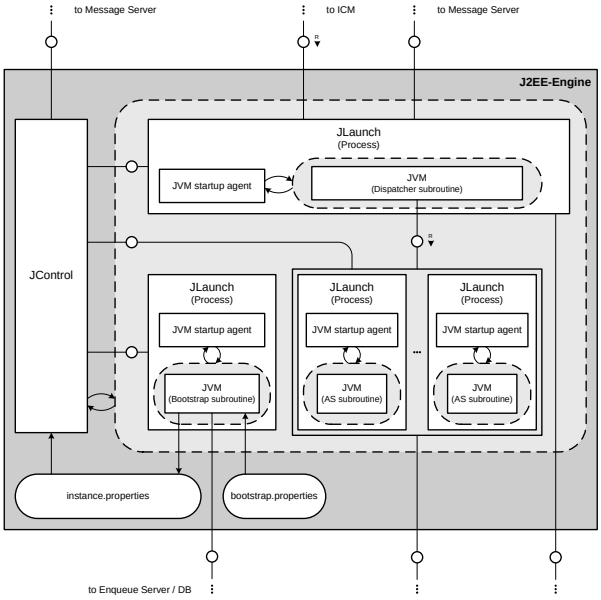

Figure 2: Structure of a J2EE-Engine during start-up

Prior to the two steps mentioned above, the J2EE-Engines have to start up. This start-up is managed by two different kinds of (operating system) processes shown in Figure 2. The *JControl* process is responsible for starting up and monitoring several *JLaunch* processes, which offer the actual functionality for the role of the J2EE-Engine.

Figure 3 clarifies the dynamic structure of the start-up procedure belonging to a J2EE-Engine as depicted in Figure 2. During this procedure different instances of JLaunch are created. The characteristics of the JLaunch processes can differ from each other. There are three different types of personalities a JLaunch process can adopt:

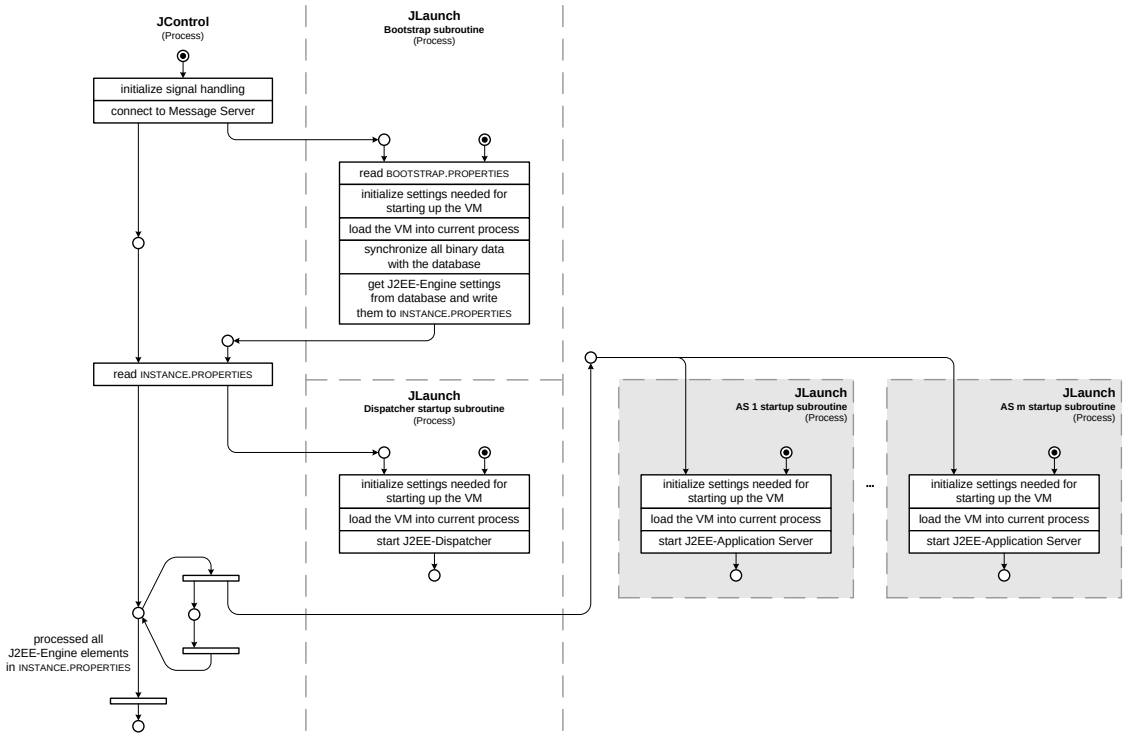

Figure 3: Petri net describing the dynamic behavior of the start-up procedure of a J2EE-Engine

- Bootstrapper
- Dispatcher
- Application Server

Before adopting one of the personalities (represented by their according subroutines), the process loads a Java-VM[6] into its own context. After that, the desired subroutine (written in Java) is executed.

At first, JControl establishes a connection to the Message Server. Then, one JLaunch process is spawned by JControl. It becomes the Bootstrapper, which synchronizes all binaries that are needed to serve client requests. Next, it writes the file INSTANCE.PROPERTIES. This file contains an exact definition of the J2EE-Engine to start up, e.g. the number of AS Processes is specified. The Bootstrapper, then, terminates and JControl starts one JLaunch process that becomes the J2EE-Dispatcher and as many JLaunch processes as specified in INSTANCE.PROPERTIES to become the J2EE-Application Servers.

3 Load balancing

The SAP Web Application Server realizes load balancing within the cluster by implementing dispatching on two different levels.

Whereas the SAP Web Dispatcher operates on cluster-level, the J2EE-Dispatchers work on level of J2EE-AS Processes within the cluster nodes. Both types of dispatchers are explained in the following.

3.1 SAP Web Dispatcher

The SAP Web Dispatcher acts as a *global load balancer* for the cluster nodes. It can be seen as the linkage between the Internet and the cluster of SAP Web AS. Hence, it is situated in the DMZ[7], where it is accessible for the clients. This is also advantageous because the cluster

[6] Virtual Machine

[7] Demilitarized Zone

nodes need not to be located inside the DMZ. Thus, they are safe behind the firewall.

As shown in Figure 1, the SAP Web Dispatcher is the only way for clients to request services from the cluster. It realizes the *Single point of entry/Façade design pattern* [DPAT] regarding all incoming HTTP(S)-requests. After receiving a request, the dispatcher forwards it to a Web Application Server using a distribution algorithm. Usually, the weighted round robin algorithm is applied. This means, that the requests are distributed among the connected J2EE-Engines using the following scheme. A weight is assigned to each J2EE-Engine. Mostly, the weight is equal to the number of contained J2EE-Application Server Processes. The SAP Web Dispatcher, then, distributes the requests depending on the weights. If e.g. Engine A contains 10 processes and Engine B contains 20 processes, twice as many requests are redirected to Engine B as to Engine A.

The SAP Web Dispatcher is implemented as a software web switch, which is an operating system process usually running on a dedicated machine. The implication to fault tolerance will be discussed later on.

Alternatively, it is possible to set up a third-party hardware solution instead of using the SAP Web Dispatcher as global load balancer.

3.2 J2EE-Dispatcher

Each J2EE-Engine possesses a J2EE-Dispatcher. Within the engine, the dispatcher acts as the *local load balancer* (see Figure 1).

A J2EE-Dispatcher receives requests that were redirected by the SAP Web Dispatcher. These requests are, then, parsed and dispatched again. The J2EE-Dispatcher also makes use of a distribution algorithm to assign each of the requests to a selected J2EE-Application Server Process. The dispatcher receives a list of all Application Server Processes connected to itself from the Message Server. Therefore, the dispatcher is able to react to changes that result from crashed AS Processes or from a higher number of configured AS Processes.

The architecture of a J2EE-Engine is very much related to the *Worker Pool architectural pattern* [BG2004]. In this concrete implementation, a J2EE-AS Process can be mapped onto a *Worker* and the *Listener* is represented by the dispatcher.

4 Failover mechanisms

4.1 Definition

The deployed applications within a cluster often provide services to the clients that are *mission-critical*. Thus, these services need to be highly available, which means that the system downtime has to be reduced to a minimum. Generally, another server has to take over responding to a request if the server that currently serves the request fails. This is only possible due to redundancy of cluster components and should, of course, happen transparently to the user.

4.2 Failover services supported by the SAP Web Application Server

The failover services supported by the SAP Web AS can be divided into the two following categories (both are explained in detail later on):

- Bean session failover
- HTTP session failover

However, both of them have to prevent data loss of transient data within sessions. To achieve this, the particular session data has to be serialized and replicated to a persistent medium from where it can be restored in case of a failure.

SAP offers two possible locations for the storage of serialized sessions. They can either be replicated to the *central database* or to the *local file-system*. Accessing the database is relatively slow compared to the usage of the local file system. On the other hand, the sessions stored in the database can be restored

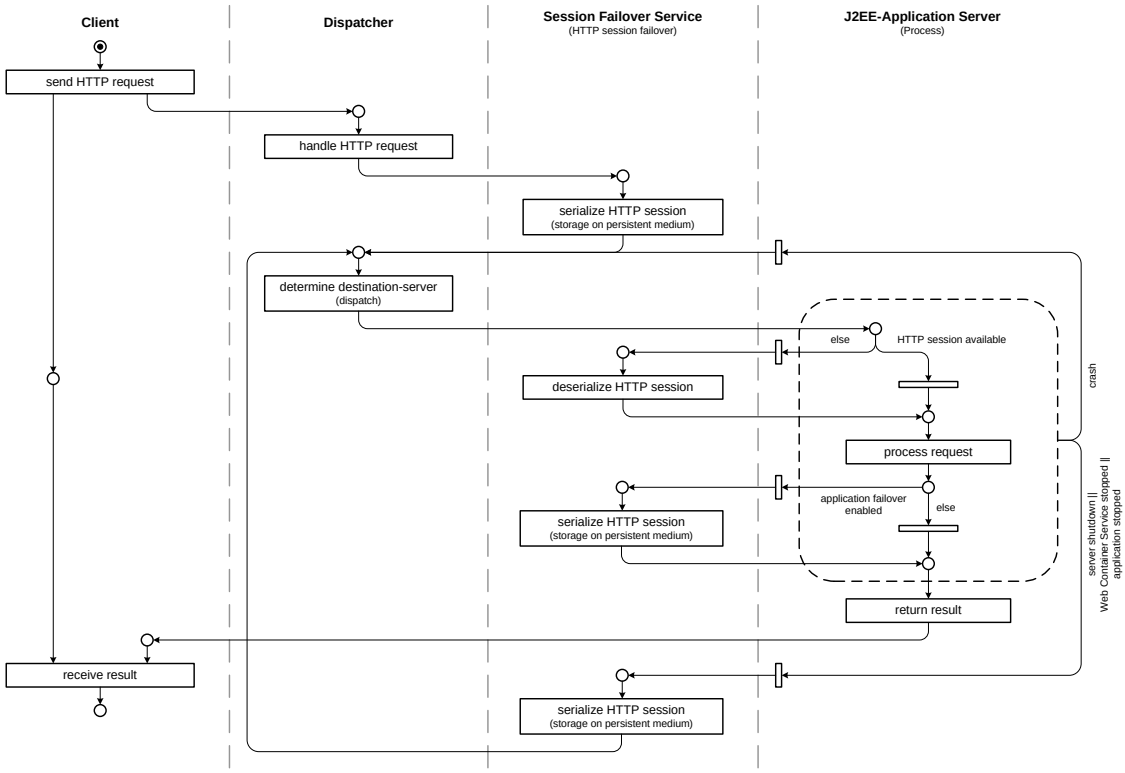

Figure 4: Petri net describing the HTTP failover mechanism

by any other cluster node whereas using the file-system is disadvantageous since the session can only be restored on this particular machine. Anyway, fault tolerance diminishes performance. It is recommended to use the file-system option because this marks a reasonable compromise between performance and security as it is relatively unlikely for a cluster node to fail completely. It is more likely that individual AS Processes crash. Replicating to the file system is sufficient in this case. [JPWAS]

4.2.1 Bean session failover

Bean session failover refers to failover regarding the state of *Enterprise Java Beans* (EJBs). Therefore, it takes place in the EJB-Container which is located inside the J2EE-AS Process.

J2EE comprises different types of Enterprise Java Beans. *Entity Beans* have a state that is already persistent per definition. Hence, there is no need to replicate them. Besides, there are two kinds of Session Beans: the *Stateful* and the *Stateless Session Beans*. The SAP Web AS supports failover mechanisms for both types of Session Beans.

When serializing a Stateless Session Bean, the EJB-Container simply stores an ID mapped to the session of that bean. Since the Bean has no state that would have to be saved, it is sufficient to create a new instance of the Bean in case of a failure.

Replicating a Stateful Session Bean[8] requires the serialization of the Bean state. Depending on the actual Bean, the amount of data that has to be serialized may become relatively large. Because of this, the developer can configure the failover options for his Beans in the Deployment Descriptor EJB-J2EE-ENGINE.XML.

4.2.2 HTTP session failover

Since most of the communication between Application Servers and the clients is based on

[8]This process is often called '*Stateful Session Failover*'.

HTTP, the data concerning the HTTP sessions has to be replicated as well. In general, the procedures and also the (dis-)advantages of Bean and HTTP session failover are comparable. The most significant difference is the level on which they operate. Whereas Bean session failover operates on level of the particular Beans, HTTP session failover acts on level of the HTTP sessions (Web Container).

Figure 4 depicts the dynamic behavior of a request/response-sequence between client and the concerned cluster components. The session has to be serialized at certain points. This happens always before dispatching the request and, in case the application failover option is enabled in the appropriate Deployment Descriptor (APPLICATION-J2EE-ENGINE.XML), after processing each request, too. It is also possible, that the server is shut down or either the Web Container Service or the application is stopped. Under these circumstances, the session is serialized again. If the server unpredictably crashes, serialization of course cannot be performed anymore.

In case that a session has to be restored, the AS Process deserializes the particular session from the persistent medium and proceeds serving the request as usual.

4.3 Overall system stability

The two abovementioned failover services can be applied in case of Application Server failures. Thus, they only focus on one possible point of failure. The SAP Web Application Server comprises other fault tolerance mechanisms that are more related to the overall stability of the cluster, which is influenced by many different kinds of cluster components.

Heterogeneous load balancing offers the possibility to isolate particular applications, which are known to be instable, on specified cluster nodes. If such an application brings down its J2EE-AS Process quite often, the isolation is able to prevent a chain reaction that would otherwise be the consequence of failover mechanisms and might bring down the whole cluster engine by engine. Therefore, the load balancing algorithms have to consider if the selected J2EE-Engine owns the respective application. Thus, it is obvious that Heterogeneous load balancing represents a combination of failover and load balancing.

The following passages briefly describe how the failure of cluster components different from the AS Processes influences the overall stability of the cluster and what possibilities exist to prevent a failure of the whole cluster.

4.3.1 SAP Web Dispatcher

As discussed in chapter 3.1, the SAP Web Dispatcher is an operating system process. It has to be highly available since it is the only cluster component that is directly reachable for the clients.

In UNIX-environments[9] it is possible to realize this by using a secondary so-called *watchdog-process*, which is mostly identical to the primary Web Dispatcher process. It is created as a child-process and, therefore, knows about the same shared memories, sockets and administration structures as the parental process. The watchdog monitors the primary process and is able to take over the control whenever this is necessary. After the former watchdog gains control it creates a process that becomes the new watchdog. [SLIB]

High availability on (operating system) process-level can be achieved by this. However, in case of e.g. a hardware failure of the machine the SAP Web Dispatcher is running on, the cluster is not able to receive any requests anymore. To prevent this, an additional hardware redundancy solution should be taken into consideration.

4.3.2 J2EE-Dispatcher

The J2EE-Dispatchers are so-called stateless[10] components of the cluster. If one of them crashes, it is immediately restarted by the

[9] due to the usage of the *fork()*-system call

[10] Since a J2EE-Dispatcher is a dynamic (software) system, it of course does have an internal state. But there is no state that is actually related semantically to its task as a dispatcher.

JControl-process. Due to the fact that no state is existent, no data is lost. The restarted dispatcher is able to continue dispatching immediately.

In case that the restart is not possible, the SAP Web Dispatcher recognizes the failed J2EE-Dispatcher to be absent and, thus, no more requests are redirected to it. The sessions, which were processed by this J2EE-Engine, are, then, restored by another Engine using the failover mechanisms described above.

4.3.3 Single Points of Failure (SPOF)

The whole cluster has a total of three Single Points of Failure. This means that no Application Server can work if one of the following three cluster components fails:

- Message Server
- Enqueue Server
- Database System

If the Message Server fails, all cluster components that want to communicate are blocked until the Message Server is operational again. Usually, the restart of the Message Server-process is completed within seconds. Hence, a temporary crash is not that problematically. But similarly to the SAP Web Dispatcher, this only works on level of operating system processes. To eliminate a failure of the cluster resulting from hardware malfunction of the machine that houses the Message Server, a hardware redundancy solution is inevitable. The same applies to Enqueue Server and the central Database System.

In contrast to this, SAP provides the additional facility to use a *Standalone Enqueue Server*. Since the Enqueue Service has to hold a lock table in memory, one runs the risk that the lock table is lost if the machine crashes. All information about the granted locks are gone in that case. Therefore, all applications that owned locks have to be reset. When using the Standalone solution instead, it is possible to replicate the lock table and avoid data loss. During the time the Enqueue Server is unavailable, all applications that either want to request or release a lock are blocked until the Enqueue Server is operational again. [SLIB]

5 Software Deployment Manager

As described above, the load balancing mechanisms redirect incoming requests to the connected cluster nodes. Every request is related to an application, which, then, serves the request and returns the desired result. Naturally, every of these applications has to be deployed within the cluster before they are ready to be executed.

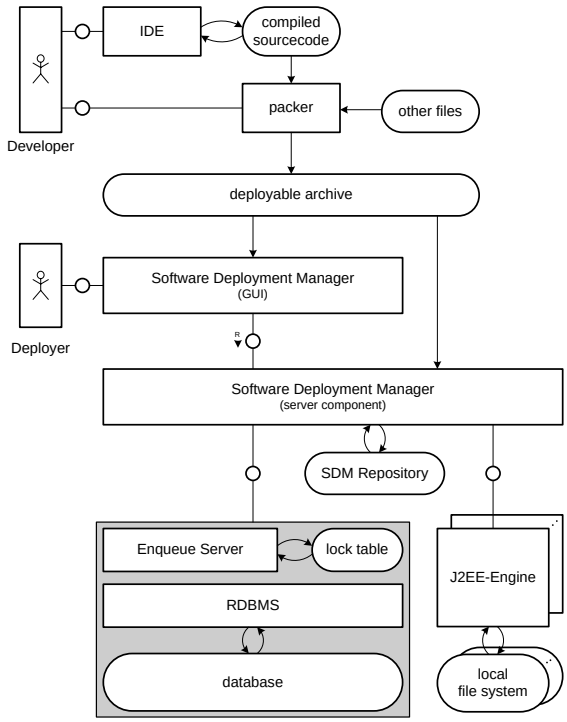

Figure 5: Compositional structure showing the participants in the deployment process

The Software Deployment Manager (SDM) is the cluster component that is responsible for the integrity of all deployed software on the cluster nodes. It consists of two parts: the *client GUI-utility* and the *server component*. Figure 5 depicts the Compositional Structure of the relevant system components taking part

in the deployment process.

All deployable content has to be merged in deployable archives. These may either be *Software Deployment Archives* (SDA) or *Software Component Archives* (SCA). The SDA is the smallest unit that can be deployed. Usually, a SDA contains one particular version of a software component whereas the SCA comprises several SDAs. Thus, the number of SDAs reflects the version-level or version-history of the contained software component. [SLIB]

The GUI-utility allows the deployer to import selected archive(s). In doing so, the client utility connects to the server component of the SDM and, then, hands over the archive. Next, the archive is placed into the *SDM Repository*, where all imported archives are stored for future use (e.g. redistribution of the archives to new cluster nodes). During the whole deployment process special dependencies between and within the archives are taken into consideration, to avoid incompatibilities.

Three different deployment actions are conceivable:

- initial deployment
- updating existing archives
- undeployment

Each of these actions refers primarily to the SDM Repository. The stored data inside the repository is, then, used to synchronize the data with the specified destination(s). All data that has to be synchronized is specified to be delivered to a particular destination. As depicted in Figure 5[11], these destinations are either the central database or the local file system of the cluster nodes. The file system content can be divided into the J2EE-applications, other files and additional libraries for the J2EE-Engines[12].

[11] Naturally, the SDM has to utilize the Message Server in order to communicate with the J2EE-Engines. Though, in the chosen level of abstraction it is possible to ignore this fact.

[12] In some cases it is necessary to restart the whole J2EE-Engine to integrate the deployed data.

Synchronization is only performed if it is required. Usually, this is the case when the content of the Repository changes. Besides, a single synchronization takes place, if e.g. a new J2EE-Engine is started up (see section 2.3).

6 Conclusion

Applying a cluster-architecture is essentially motivated by the two primary aims *scalability* and *high availability* of the offered services. This paper described how these aims are realized regarding the concrete example of the Java-personality of the SAP Web Application Server.

The two main concepts - *flexible load balancing* and *effective fault-tolerance mechanisms* - were discussed in detail. The SAP Web AS utilizes 2-level-based dispatching combined with various failover mechanisms in order to adapt quickly to changes in the availability of cluster nodes and to provide a well-balanced workload across all available nodes. Therefore, expanding the cluster is fairly easy to accomplish by adding new machines.

All of the mentioned activities are enabled due to redundancy of cluster components and aim at the users' satisfaction. Thus, they are completely transparent to the users.

The sophisticated cluster-architecture of the SAP Web AS provides a solid technical basis for the execution of Java Enterprise Applications if all eventualities are considered while setting up the local cluster. Especially the Single Points of Failure have to be secured by additional hardware redundancy solutions in order to guarantee real high availability.

References

[WAS] Frédéric Heinemann and Christian Rau, *SAP Web Application Server*, SAP Press/Galileo Press, 1^{st} Edition, 2003

[JPWAS] Karl Kessler, Peter Tillert and Panayot Dobrikov, *Java-Programmierung mit dem SAP Web Application Server*, SAP Press/Galileo Press, 1^{st} Edition, 2005

[DPAT] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 5^{th} Edition, 1996

[SLIB] *SAP Library*,
http://help.sap.com/

[SDN] *SAP Developer Network*,
https://www.sdn.sap.com/

[WASJ6.40] SAP AG, *SAP Web AS Java 6.40 - a Reliable, Scalable, Efficient J2EE Application Server*, 2005

[MK2.3] Matt Kangas, *Introduction to the SAP Web Application Server*, Version 2.3, 2005,
http://www.sap.com/mk/
get?_EC=wjqfETRbYxuv8o_fZzdmLI

[AK1] Abraham Kang, *J2EE-Clustering, Part 1*, JavaWorld.com, 2001,
http://www.javaworld.com/javaworld/
jw-02-2001/jw-0223-extremescale_p.html

[AK2] Abraham Kang, *J2EE-Clustering, Part 2*, JavaWorld.com, 2001,
http://www.javaworld.com/javaworld/
jw-08-2001/jw-0803-extremescale2_p.html

[BG2004] Bernhard Gröne, *Konzeptionelle Patterns und ihre Darstellung*, 2004

[CA2004] BEA Systems, *Clustering Architectures*, 2004,
http://e-docs.bea.com/wls/
docs81/cluster/planning.html

[JAP2004] SAP AG, *Creating a J2EE-Based Car Rental Application*, 2004