DEPENDABLE SYSTEMS - VERGLEICH ZWEIER J2EE CLUSTER LÖSUNGEN

1. EINLEITUNG

Ein Cluster beschreibt eine Gruppe von Rechnern, die über eine Netzwerkarchitektur sehr miteinander verbunden sind, um eine gemeinsame Aufgabe zu erfüllen. Für den Nutzer bleibt diese Rechnergruppierung transparent, sodass nach außen der Eindruck einer leistungsstarken Einzelmaschine entsteht.

Der Betrieb von Clustern begründet sich vor allem in der deutlich gesteigerten Leistungsfähigkeit und Zuverlässigkeit im Vergleich zum Betrieb von einzelnen, getrennten Maschinen. Aus betriebswirtschaftlicher Sicht befürworten oft die Kostenvorteile im Zuge einer Konsolidierung im IT-Bereich den Betrieb oder die Nutzung eines Clusters.

Dieser Projektbericht thematisiert die Ausfallsicherheit von J2EE Application Servern. Am Beispiel von JBoss und GlassFish wurden die eingesetzten Techniken im Bereich der Fehlertoleranz untersucht und mit einander verglichen. Ein besonderer Schwerpunkt liegt im Bereich der Cluster-internen Kommunikation und den Verfahren sowie Architekturen zur Zustandsreplikation innerhalb des Cluster-Systems.

Im Folgenden werden nun die grundlegenden Funktionen und Einstellungsmöglichkeiten sowie die Installation der beiden Referenzprodukte vorgestellt. Der zweite Teil beschreibt die jeweils eingesetzten Techniken zum Aufbau eines hochverfügbaren Systems.

2. CLUSTER-ARCHITEKTUR

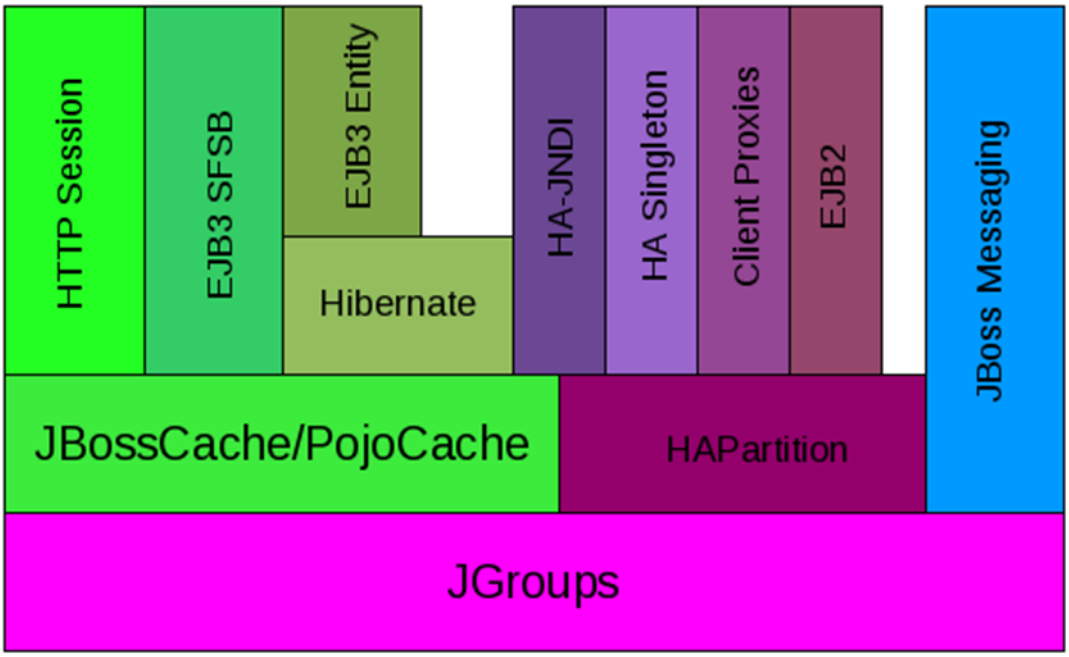

Abbildung 1: Cluster-Architektur im JBoss-Application-Server

Abbildung 1 zeigt die im JBoss-Application-Server (JBoss) implementierte Cluster-Architektur. Als Gruppenkommunikationsframework wird JGroups verwendet, was sämtliche Clusterkommunikation zwischen Knoten innerhalb eines Clusters regelt. JGroups bietet verlässliche Punkt-zu-Punkt oder Punkt-zu-Multipunkt Kommunikation, sowie eine automatische Gruppenmitgliedschaftsverwaltung, inklusive der Erkennung von gecrashten Knoten. HAPartition ist ein Service, welcher als Adapter die JGroups-Funktionalität höheren Services zur Verfügung stellt. Über diesen Service können zum Beispiel der aktuelle Clusterview, sowie Knoten auf den bestimme Services laufen erfragt werden. Sämtliche Zustandsreplikation – HTTP Session, Entity Beans, Statefull Session Beans - wird von JBoss Cache verwaltet. Dies ist ein transaktionaler, replizierter, in-memory Cache, wobei die einzelnen Cache-Instanzen wiederum über JGroups kommunizieren. Über Cacheloader kann der Cacheinhalt zusätzlich persistiert werden.

Die Failover-Logik ist in den Proxis, welche über den speziellen HA-JNDI-Service geladen werden, implementiert. Dies ist ein clusterweiter JNDI-Service, welcher über dem lokalen JNDI-Service anzuordnen ist und einen zusätzlichen, clusterweiten Kontext anbietet. Dabei ist der Service in der Lage sowohl clusterweit gebundene, als auch auf anderen Knoten nur lokal gebundene Objekte zu ermitteln. Die registrierten Proxis beinhalten einen Clusterview mit Knoten, welche das entsprechende Objekt anbieten, sowie eine Policy, nach der Knoten ausgewählt werden. Im Fehlerfall wird der nächste Knoten der Liste benutzt.

Des Weiteren bietet JBoss noch die Möglichkeit einen Service clusterweit als Singleton zu installieren. Dabei ist der Service nur auf dem Masterknoten aktiv und auf allen anderen Knoten passiv repliziert. Im Fehlerfall wird ein Slaveknoten als neuer Masterknoten bestimmt.

3. Installation und Cluster-Deployment

GlassFish – Installation

Vollständige Cluster und High-Availablility Unterstützung bietet GlassFish in der Open Source Variante nur mit der Version 2.1.1. Die aktuelle Version 3 unterstützt diese Features lediglich in der Produktversion von Oracal.

Auf https://glassfish.dev.java.net/downloads/v2.1.1-final.html befinden sich die aktuellen Installationsprogramme für verschiedenste Plattformen. Nach dem Entpacken der JAR-Datei muss der GlassFish mittels ANT mit den entsprechenden Optionen gebaut werden. Um die Cluster-Unterstützung zu aktivieren, lässt sich die vorkonfigurierte `setup-cluster.xml` verwenden.

Wie in Abbildung 2 dargestellt, ist der Administrations-Knoten ein zentrales Konzept der Architektur eines GlassFish Clusters. Über diesen Domain Administration Server (DAS) können Zugriffsrechte und Clusterkonfigurationen verwaltet werden. Außerdem erfolgt das Deployment von Webanwendungen über diesen Knoten. Ein DAS wird standardmäßig beim Starten einer Domain erzeugt. GlassFish stellt eine vorkonfigurierte `Domain1` bereit, die per Admintool über das Kommando asadmin start-domain domain1 gestartet werden kann.

Bei Konfiguration eines Clusters unterscheidet GlassFish zwei Abstraktionsebenen Node Agents und Instanzen. Ein Node Agent identifiziert eine physikalische Maschine, die am Cluster beteiligt werden soll. Über das Admintool `asadmin` kann ein Node-Agent mit den folgenden Kommandos

erzeugt und gestartet werden. Es ist zu beachten in der host-Option den Server einzutragen, auf dem der DAS läuft, um den Node-Agent mit dem DAS bekannt zu machen.

```
asadmin create-node-agent --user admin --host adminnode nodeagent1
```

```
asadmin start-node-agent --user admin --host adminnode nodeagent1
```

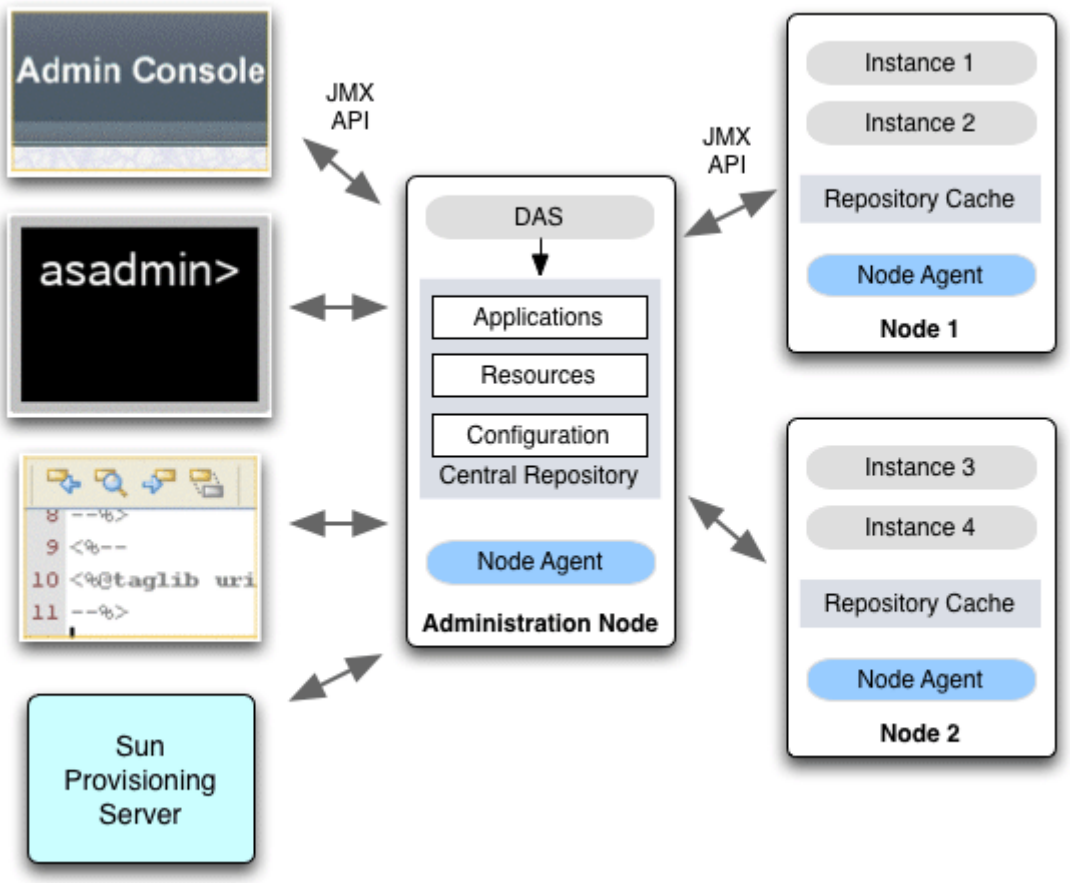

Abbildung 2: Aufbau der Verwaltungs-Domain im Cluster

Pro Node Agent ist der Betrieb mehrere Serverinstanzen möglich. Technisch betrachtet ist eine Serverinstanz ein Prozess der virtuellen Maschine von Java auf dem die Webanwendungen laufen. Die einzelnen Serverinstanzen gruppieren sich wiederum zu einem Cluster. Abbildung 3 veranschaulich Konfiguration eines Clusters an Hand einer Übersicht auf der Admin Konsole über die teilnehmenden Serverinstanzen.

GLASSFISH – CLUSTER DEPLOYMENT

Das Deployment der Webanwendungen erfolgt ebenfalls über den zentralen Administrations Knoten. Die grafische Admin Konsole ermöglicht das Bereitstellen von Anwendungen im EAR- und WAR-Containerformat.

Auf dem Administration Knoten gibt es zentrales Repository. In diesem werden neben den Konfigurationen zu jedem administrierten Cluster auch alle deployten Anwendungen vorgehalten. Jeder Knoten besitzt zusätzlich ein lokales Repository. Dieses lokale Repository synchronisiert alle für den jeweiligen Cluster relevanten Daten, wie Einstellungsparameter und Anwendungen die auf diesem Cluster deployt wurden. Die Verwendung eines lokalen Repositories soll vor dem Ausfall des zentralen Administrations Knotens schützen. Dazu werden

die lokalen Repositories bei jedem Start einer Serverinstanz und beim Deployment einer Webanwendung abgeglichen.

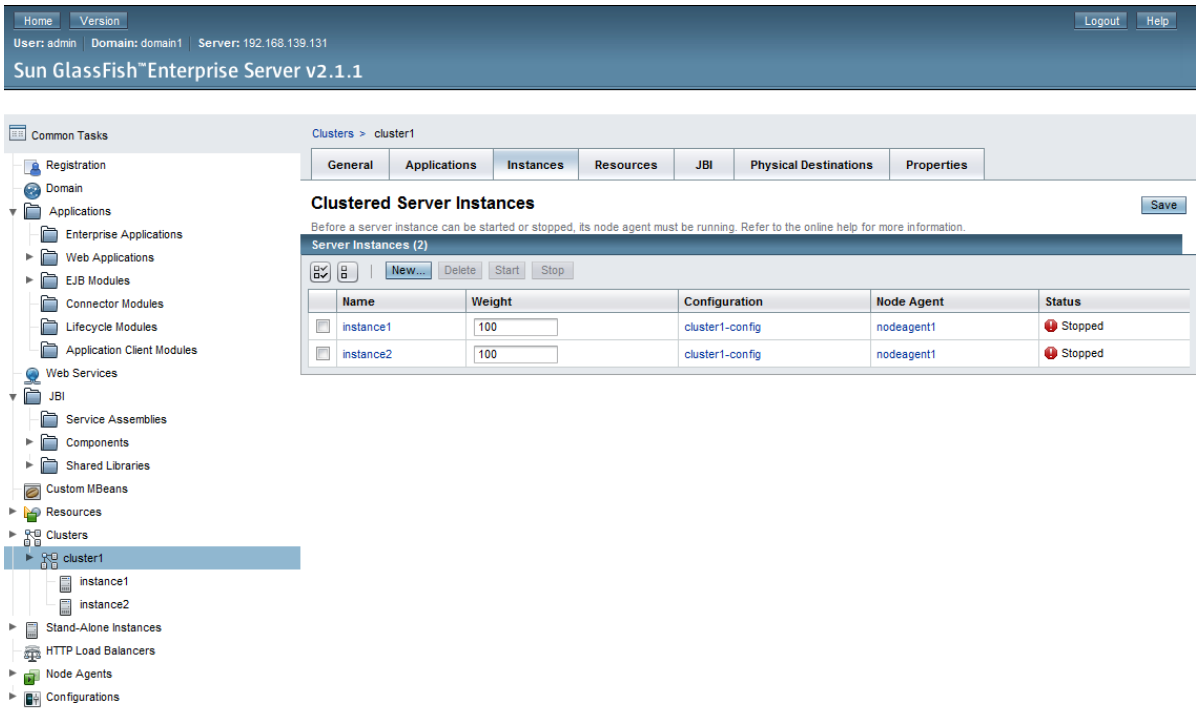

Abbildung 3: Clusterübersicht in der DAS-Oberfläche

JBoss – Installation

Im Folgenden wird die Installation eines JBoss-Servers der Version 5.1.0.GA beschrieben. Es wird davon ausgegangen, dass bereits JDK 1.5 / 1.6 installiert ist. Da der JBoss-Application-Server vollständig in Java programmiert ist, läuft dieser auf jeder Plattform auf der Java installiert ist. Es bestehen keine besonderen Hardwareanforderungen.

Unter http://sourceforge.net/projects/jboss/files/JBoss/JBoss-5.1.0.GA wird eine Binary-Zip-Datei zum herunterladen angeboten, welche den einfachsten Weg der Installation bietet. Die Zip-Datei muss lediglich entpackt werden. Zuletzt muss noch die JBOSS_HOME Umgebungsvariable gesetzt werden, welche auf das Rootverzeichnis des JBoss-Application-Servers verweisen muss.

Standardmäßig werden bereits Beispielkonfigurationen mitgeliefert, so dass unter anderem ein Cluster out-of-the-box eingerichtet werden kann.

JBoss – Cluster-Deployment

Um eine Applikation clusterweit zu installieren bieten sich zwei Möglichkeiten. Entweder man erledigt dies manuell, in dem die Applikation lokal auf jedem Knoten installiert wird. Dadurch bietet sich der Vorteil, dass die Anwendung von einzelnen Knoten entfernt werden kann, ohne das dies die Installation auf anderen Knoten beeinflusst. Jedoch kann eine Applikation auch über den Farming-Service automatisch auf allen Knoten installiert werden. Wird jedoch die Anwendung von einem Knoten entfernt, so wird diese automatisch von allen Knoten entfernt.

Im Folgenden wird die Funktionsweise des Farming-Service anhand zwei verschiedener Szenarien erläutert.

Szenario 1: Laufender Cluster, Anwendung wird auf einem Knoten installiert

Um eine Anwendung clusterweit zu installieren muss auf einem Knoten die Anwendung in den Farming-Ordner kopiert werden. Standardmäßig ist dies der Ordner „farm". Der entsprechende Deploymentscanner, welcher periodisch den Ordner auf Veränderungen überprüft, erkennt die neue Anwendung und installiert sie auf dem lokalen Knoten. Anschließend wird die Anwendung über den FarmMemberService auf alle anderen Knoten repliziert. Zunächst wird die Anwendung nur temporär auf den neuen Knoten gespeicht und gegebenfalls wird eine ältere, laufende Version der Anwendung deinstalliert. Anschließend wird die neue Anwendung in den lokalen Farming-Ordner kopiert und lokal installiert.

Szenario 2: Laufender Cluster, clusterweite Anwendung installiert, neuer Knoten

Der FarmMemberService erfragt bei dem Koordinatorknoten des Clusters Informationen über aktuell clusterweit installierte Anwendungen. Der Koordinatorknoten ist der erste Knoten innerhalb des Clusterviews. Dieser antwortet mit einer Hashmap, welche jeweils den Namen der Anwendung und das zugehörige Datum, wann die Anwendung zuletzt modifziert wurde, enthält. Im Falle einer neuen Anwendung wird diese Anwendung vom Koordinatorknoten zum neuen Knoten übertragen und installiert. Im Falle eines erfolgreich überwundenden Split-Brains, wo zeitweilig zum Beispiel zwei einzelne Cluster parallel liefen und beide eine gleiche Anwendung installiert haben, welche womöglich unterschiedlich modifiziert wurde, wird die neueste Version der Anwendung clusterweit installiert und gegebenfalls die ältere Version auf den einzelnen Knoten entfernt. Die Überprüfung erfolgt dabei anhand des Datums, welches anzeigt, wann die jeweilige Anwendung zuletzt modifziert wurde.

4. Cluster - Kommunikation

Um Hochverfügbarkeit in einem Cluster-System sicher zu stellen, muss zu jeder Zeit die Kommunikation zwischen den beteiligten Knoten im Cluster möglich sein. Dieser Anforderung kann beispielsweise eine Client Server Architektur nicht gerecht werden, da auch beim Ausfall einzelner Knoten der Cluster an sich weiter funktionsfähig bleiben soll und möglichst alle deployten Anwendungen weiterhin zur Verfügung stehen.

Sowohl JBoss als auch GlassFish setzten Frameworks zur Gruppenkommunikation ein. Ziel ist es von der darunterliegenden Netzwerkschicht zur abstrahieren und alle am Cluster beteiligten Instanzen in einer Gruppe zusammenzufassen. Das Framework stellt dabei die Nachrichtenkommunikation innerhalb der Gruppe sicher und übernimmt Verwaltungsaufgaben innerhalb der Gruppe. So muss sichergestellt werden, dass bei einem Ausfall eines Knotens die interne Struktur der Gruppe erneuert wird. Gleiches gilt für das Hinzufügen eines neuen Knotens.

Um den Anforderungen eines quasi adhoc Netzwerkes gerecht zu werden, kommen aus Peer to Peer Netzwerken bekannte Techniken zum Einsatz. JBoss und GlassFish setzten frei verfügbare Frameworks zur Gruppenkommunikation ein, die im Folgenden vorgestellt werden.

GlassFish – Shoal / JXTA

GlassFish setzt das Gruppenkommunikationsframework Shoal ein, um den Clusterbetrieb zu implementieren. Der Einsatz des Peer to Peer Frameworks JXTA abstrahiert von der darunterliegenden Netzwerkarchitektur und bringt Vorteile in den Bereichen Zuverlässigkeit

und Skalierbarkeit. Shoal bietet Funktionen zum Austausch von Nachrichten und Daten innerhalb des Clusters. Ebenfalls werden Events wie Hinzufügen, Herunterfahren und Ausfall eines Knotens oder einer Gruppe von Knoten signalisiert.

Shoal setzt einen Heartbeat Mechanismus ein, um die Verfügbarkeiten der einzelnen Knoten in der Gruppe zu überprüfen. Bleibt die Antwort auf den Heartbeat aus, wird ein `Failure suspected` Signal an die Gruppe geschickt. Falls ein weiterer Knoten den vermuteten Ausfall bestätigen kann, erhält die Gruppe ein `Failure notify` Signal. Gleichzeitig wird der betroffene Knoten vom Rest der Gruppe isoliert es ist also vorerst keine beliebige Kommunikation mit diesem Knoten möglich. Dieser Mechanismus nennt sich `Recovery Failure Fencing`. Ziel ist es möglichen Race Conditions vorzubeugen, falls der ausgefallene Knoten versucht, sich wiederherzustellen.

Ein dritter Knoten kann ich Aufgaben des ausgefallenen Knotens übernehmen, falls er vorher als Recovery Node für den ausgefallenen Knoten registriert wurde. In diesem Fall wird ein Failure Recovery Signal an den Recovery Knoten geschickt der anschließend die Identität des ausgefallenen Knotens übernimmt. War jedoch der Recoveryprozess des ausgefallenen Knotens erfolgreich, kann dieser ebenfalls seine Rolle in der Gruppe wieder einnehmen. Die vorherige Isolation dieses Knotens hebt sich in dieser Situation wieder auf.

Um den Fehlerfall managen zu können übernimmt ein Knoten in dieser Situation die Rolle des Masters ein. Er entscheidet wann ein Knoten vom Rest der Gruppe isoliert wird und ein Anderer dessen Identität übernehmen soll.

JBOSS - JGROUPS

JGroups ist das zugrundeliegende Gruppenkommunikationsframework innerhalb eines JBoss-Application-Server-Clusters. Gewünschte Kommunikationseigenschaften zur Sicherstellung von Verlässligkeit können variabel konfiguriert werden. Abbildung 4 zeigt die Architektur von JGroups.

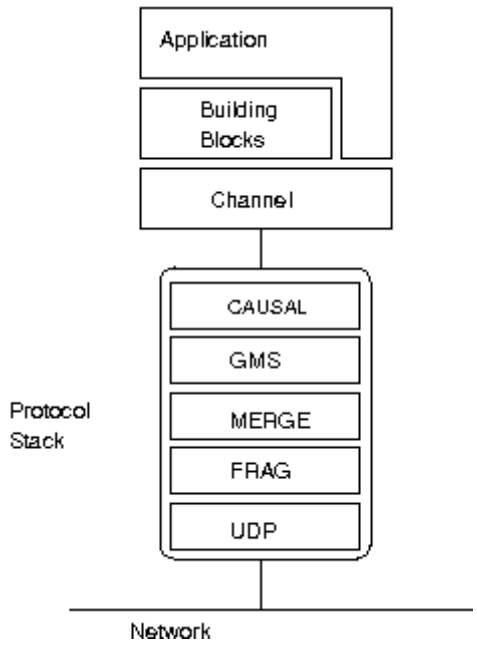

Abbildung 4 : Architektur von JGroups mit beispielhaften Protokollstack

Ein Channel bietet einer Applikation einfachste Kommunikationsmöglichkeiten innerhalb einer Gruppe. Eine Gruppe wird dabei über ihren Namen identifiziert und jeder Channel besitzt eine eindeutige Adresse. Des Weiteren gibt es in jeder Gruppe einen Koordinatorknoten (älteste Knoten im Cluster). Als Vorbild für die Entwicklung dienten BSD – Sockets. Dabei bietet ein einfacher Channel nur asynchrone Kommunikation.

Building Blocks bieten einer Applikation höherwertige Funktionalitäten. Sie dienen als weitere Abstraktionsschicht und benutzen selbst intern zum Beispiel einen Channel, der jedoch unter Umständen der Applikation verborgen bleibt. So bieten Building Blocks zum Beispiel RPC – Funktionalität (RPC-Dispatcher) oder eine synchrone Kommunikation (Message – Dispatcher).

Der Protokollstack ist ein bidirektionaler Stack, wobei jede Protokollschicht eine ergänzende Funktionalität bereitstellt. Jede Nachricht die über den Channel versendet werden soll, durchläuft den Protokollstack. Dabei kann jedes Protokoll die Nachricht löschen, manipulieren, Nachrichtenreihenfolgen ändern oder einen zusätzlichen Header der Nachricht beifügen. Ein Header beinhaltet dabei, ähnlich wie ein IP-Header, Metainformationen, wie zum Beispiel die Sequenznummer, um Nachrichten in einer bestimmen Reihenfolge zu senden. Innerhalb eines Protokollstacks kommunizieren die einzelnen Protokollschichten über Events miteinander. Bei der Zusammenstellung des Stacks sind Abhängigkeiten einzelner Protokolle zu beachten.

Im Folgenden werden Funktionen und implementierte Mechanismen wichtiger Protokollschichten erläutert.

Die unterste Protokollschicht bildet das Transportprotokoll. Dadurch wird festgelegt, ob Nachrichten über UDP oder TCP über das Netzwerk versendet werden.

Die darüberliegende Protokollschicht dient dazu einen initialen Clusterview zu ermitteln. Dieser wird benötigt, um an einen ausgewählten Knoten einen Join-Request zu senden. Das entsprechende Protokoll wird bei Empfang eines FIND_INITIAL_MBRS-Events aktiv, welches von einer höheren Protokollschicht ausgelöst wird. Dabei wird entweder ein Ping an eine IP-Multicast-Adresse gesendet (PING-Protokoll) oder zum Beispiel mehrere TCP-Pings an vordefinierte Empfänger (TCPPING-Protokoll).

Die nächste Protokollschicht dient dazu, Teilgruppen, welche nach einer Netzwerktrennung entstehen, aufzuspüren und wieder zu einer Gruppe zu vereinen. Dabei erfragt der jeweilige Koordinatorknoten periodisch den initialen Clusterview. Werden dabei Inkonsistenzen (unterschiedliche Views) festegestellt, so wird ein entsprechendes Merge-Event erzeugt.

Es folgt die Protokollschicht für die Erkennung von gecrashten Knoten. Dabei wird zum Beispiel ein Heartbeat-Mechanismus eingesetzt, indem jeder Knoten periodisch einen Heartbeat an die ganze Gruppe sendet. Alternativ können die Knoten auch einen Ring von TCP-Sockets aufbauen, wo jeder Knoten zu seinem Nachbarn eine TCP-Verbindung aufbaut. Wird diese Verbindung ohne Absprache unterbrochen, ist von einem gecrashten Knoten auszugehen. Es gibt auch ein Protokoll, welches ICMP-Nachrichten zur Fehlererkennung verwendet.

Für den verlässlichen Nachrichtenaustausch ist die nächste Protokollschicht verantwortlich. Nachrichten werden in FIFO-Reihenfolge versendet. Um sicherzustellen, dass Nachrichten auch bei dem Empfänger angekommen sind, werden Sequenznummern und Agreement-Mechanismen benutzt.

Schließlich folgt die Protokollschicht welche die Gruppenmitgliedschaft verwaltet. Hier werden Join-Requests und Leave-Request, so wie zum Beispiel MERGE-Events bearbeitet. Dabei wird auf Funktionalität anderer Protokollschichten zurückgegriffen. Gegebenfalls werden andere Protokolle im Falle eines neuen Clusterviews benachrichtigt.

Es folgen Protokollschichten zur Verschlüsselung, Authentifizierung und Flusskontrolle.

5. ZUSTANDSREPLIKATION

JBOSS CACHE

Um Zustandsinformationen im Cluster zu replizieren und deren Replikation zu verwalten, wird JBoss Cache im JBoss-Application-Server eingesetzt. JBoss Cache ist ein Baum-strukturierter, transaktionaler Cache, welcher geclustered betrieben werden kann.

Als Kommunikationsframework zwischen den einzelnen Cacheinstanzen innerhalb des Clusters kommt wieder JGroups zum Einsatz. JBoss Cache bietet sowohl einen synchronen, als auch einen asynchronen Replikationsmechanismus. Dabei benutzen sämtliche geclusterte Services (Replikation von Web-Session-State, EJB3 Statefull Session Beans ...) ihren eigenen Cache. Um eine EJB3 Statefull Session Bean zu replizieren, bedarf es lediglich einer zusätlichen Annotation (@Clustered). Um die Replikation über JBoss Cache kümmert sich der entsprechende Container. Es ist jedoch auch möglich eine Cache-Instanz direkt in der Anwendung für eigene, spezielle Replikationen zu nutzen.

JBoss Cache ist ein in-memory Cache. Über einen Cacheloader können die Zustände persistiert werden. Dabei sind verschiedene Konfigurationen möglich. Des Weiteren können auch die Orte der Replikation von clusterweit auf jedem Knoten bis zu einzelnen ausgewählten Knoten spezifiziert werden.

Die Zustandsübertragung kann dabei über Byte-Arrays oder einen Stream umgesetzt werden. Des Weiteren kann auch nur ein Teilzustand übertragen werden. Schließlich muss je nach Einsatz entschieden werden, ob der transiente in-memory Zustand oder der persistierte Zustand übertragen werden sol bzw. beidesl.

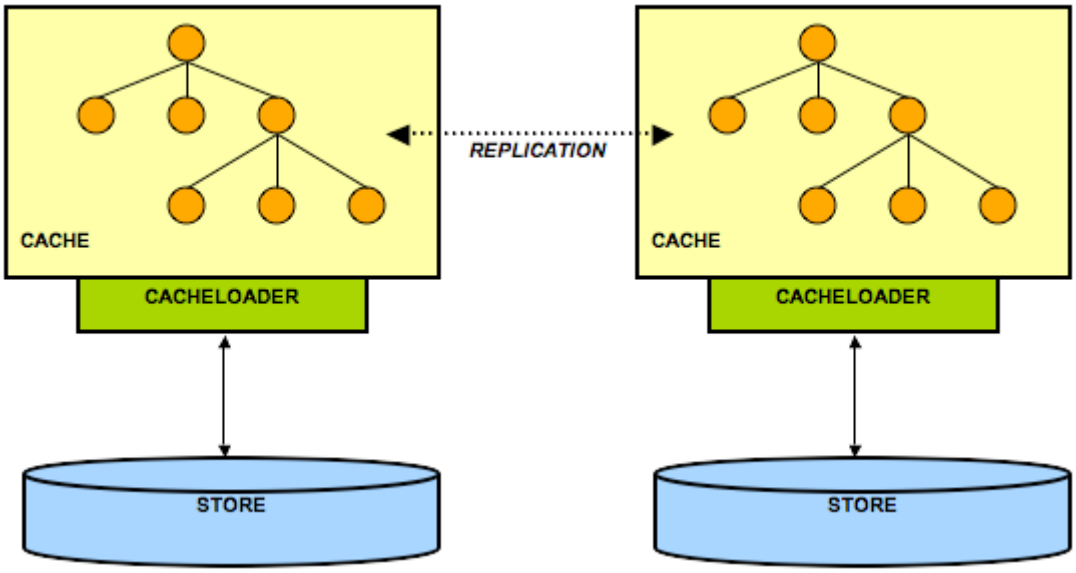

Abbildung 5: JBoss Cache - Beispielkonfiguration für 2 Knoten

Um bei Transaktionen einen konsistenten Zustand zu gewährleisten, werden Replikationen erst nach dem Commit durchgeführt. Im Falle von asynchroner Replikation, erfolgt die Replikation durch einen Aufruf. Jedoch kann nicht sichergestellt werden, ob die Replikation auf anderen Knoten auch erfolgreich ist. Im Falle des synchronen Replikationsmodus wird ein two-phase-commit-Protokoll eingesetzt. Dabei wird zuerst ein Vorbereitungsaufruf an alle weiteren Cacheinstanzen geschickt, dass die Modifizierungen durch die Transaktion jeweils lokal durchgeführt werden sollen. Können einzelne Instanzen dazu zum Beispiel ein erforderliches Lock nicht erhalten, so antworten diese mit einer negativen Response und der Initiator schickt ein clusterweites Rollback. Im Falle einer positiven Response aller Beteiligten sendet der Initiator ein globals Commit. Als Locking-Schema wird eine Multi-Version-Concurrency-Controll-Implementierung eingesetzt. Als Isolationslevel werden Repeatible_Read und Read_Committed unterstützt.

GlassFish – Memory Replikation

Der im GlassFish eingebaute Failover-Mechanismus basiert auf einer Ring-Topologie wie in Abbildung 6 dargestellt. Der Aufbau der Ring Topologie basiert auf den Namen der einzelnen Serverinstanzen. Dabei kennt jede Serverinstanz ihren rechten Nachbarn und speichert eine Kopie ihrer HTTP-Session auf der benachbarten Instanz. Zum Erzeugen und Speichern der Kopie werden jeweils zwei Einstellungsmöglichkeiten angeboten. Beim eventbasierten Sichern werden jeweils beim Senden der HTTP-Response die Kopien angelegt. Alternativ kann auch ein bestimmtes Zeitintervall für das Erzeugen der Kopie angegeben werden.

Für die Speicherung der Kopie bietet GlassFish ebenfalls zwei Alternativen an. Zum einen kann die Zustandsreplikation komplett im Speicher erfolgen. Damit sind allerdings beispielsweise Stromausfälle mehrerer benachbarter Knoten nicht im Fehlermodel enthalten. Werden dies wiederhergestellt sind die Zustandsinformationen verloren. Für diesen Fall bietet die kommerzielle Oracle Version eine spezielle High Availability Datenbank zur Zustandsspeicherung an.

Fällt nun eine Instanz aus, versucht der Load Balancer die Anfrage an einen beliebigen weiteren Knoten weiterzuleiten. Ist dieser Knoten zufällig der Benachbarte und hält eine Kopie der

aktuellen Session, kann er die Anfrage sofort beantworten. Fehlt die Kopie der Session wird ein Broadcast-Signal innerhalb des Clusters versandt. Empfängt ein Knoten mit der gewünschten Sessionkopie dieses Signal, übernimmt er die weitere Anfragebeantwortung.

In beiden Fällen muss die bestehende Ring-Topologie angepasst werden. Der ausgefallene Knoten wird entfernt und sein linker Nachbar bekommt den rechten Nachbar des ausgefallenen Knoten als neuen rechten Nachbarn zugewiesen.

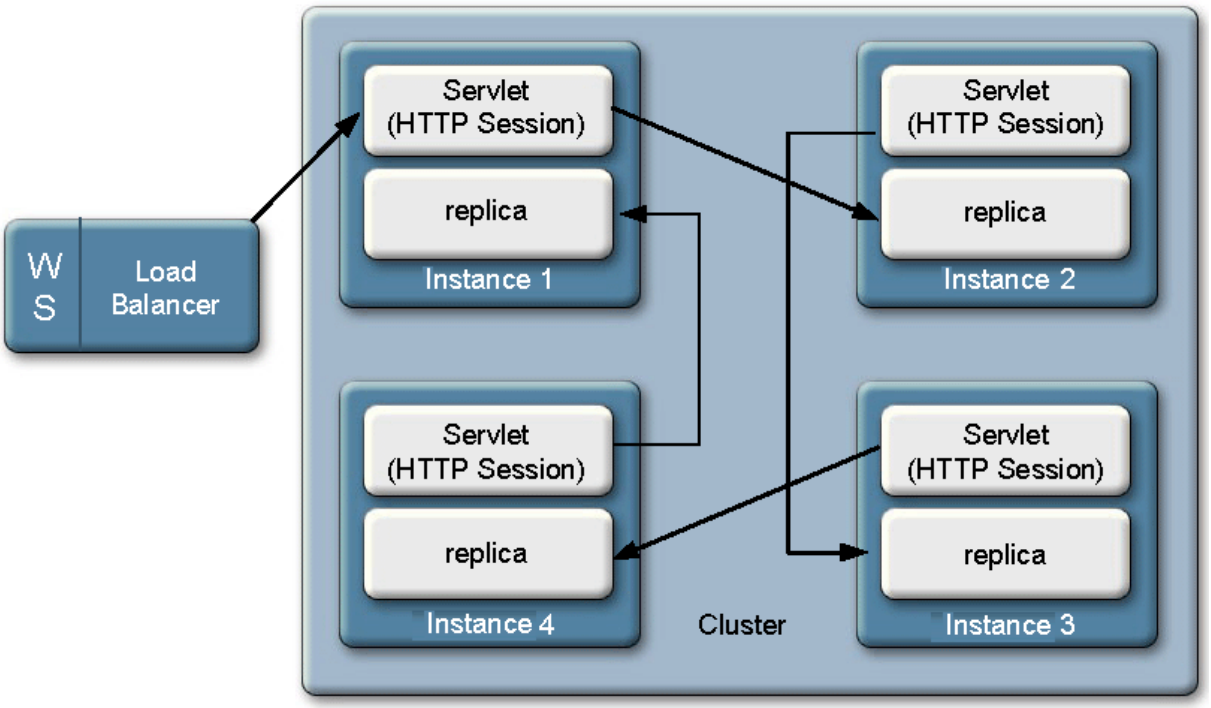

Abbildung 6: Aufbau der Ring-Topologie zur Zustandsreplikation

6. Zusammenfassung

Die jeweilige Clusterlösung beider vorgestellten J2EE Application Server versucht durch Redundanz im Raum Verlässlichkeit sicherzustellen. Dabei werden sowohl Anwendungen als auch der aktuelle Zustand repliziert. Im zugrundeliegenden Fehlermodell werden dabei Crashfaults ganzer Knoten bzw. einzelner Services betrachtet. Durch Failover-Mechanismen wird versucht auftretende Faults transparent für den Client zu halten. Zentrale Komponente eines Clusters ist die Clusterkommunikation. Beide Application Server benutzen ein Framework, welches Verlässlichkeit unter anderem durch den Einsatz von Aggreement Protokollen (Consensus Protokoll) und Heartbeats zur Erkennung nicht verfügbarer Knoten unterstützt.

Quellen

http://docs.jboss.org/jbosscache/3.2.1.GA/userguide_en/html_single/index.html

http://docs.jboss.org/jbossclustering/cluster_guide/5.1/html-single/index.html

http://www.jgroups.org/manual/html_single/index.html

http://developers.sun.com/appserver/reference/techart/glassfishcluster/