JBoss-Features und Tools: JBoss im Cluster

von Dirk Weil und Marcus Redeker

Viele Köche verderben den Brei, oder?

In der letzten Ausgabe des *Java Magazins* hatten wir unsere Serie über den JBoss Application Server gestartet und einen ersten Überblick über seine Möglichkeiten gegeben. In diesem Artikel werden wir uns etwas genauer anschauen, was der JBoss im Cluster bietet und wie man diese konfigurieren kann.

Clustering: Grundlagen und Architekturen

Für kleinere Web- oder Enterprise-Anwendungen reicht oft eine einzelne Instanz eines JBoss [1] auf einem Windows- oder Linux-Server aus. Allerdings kommen bei steigender Nutzerzahl rasch Transaktionsraten zusammen, die an die Grenze der einzelnen Maschine geraten. Klicken z.B. nur 100 User im Mittel alle 20 Sekunden auf einen Link oder Button der Applikation, werden immerhin schon fünf Requests pro Sekunde abgewickelt, die jeder für die Seitenresponse ggf. mehrere Datenbanktransaktionen auslösen.

Neben der Auslastung des Systems stellt auch seine Verfügbarkeit einen wichtigen Faktor für die Akzeptanz einer Anwendung dar. Defekt oder Wartung des Systems machen die Anwendung unbenutzbar, was in vielen Fällen nicht akzeptabel ist.

Durch den Einsatz mehrerer Systeme, die zu einem Cluster zusammen geschaltet sind, lassen sich beide Problemfelder behandeln. Dabei werden mehrere JBoss-Instanzen mit gleicher Anwendungskonfiguration so verknüpft, dass sie aus Sicht des Benutzers wie ein großes System erscheinen. Die Benutzeranfragen werden durch einen Load Balancer auf die JBoss-Instanzen verteilt. Bei Ausfall eines Knotens übernehmen die restlichen seine Last (Abb. 1).

Im Folgenden betrachten wir zwei übliche Architekturen für Enterprise-Anwendungen (mit Web- und EJB-Anteil) im Cluster. Die Basisarchitektur (Abb. 2) verwendet einen einzelnen JBoss-Cluster. Die darin enthaltenen Server enthalten die betrachtete Anwendung jeweils vollständig. Requests des Browsers werden über den Load Balancer auf die JBoss-Instanzen verteilt. Zugriffe der Web-Anwendungen auf EJBs verbleiben in der gleichen Instanz, da Aufrufe innerhalb einer JVM naturgemäß effizienter ablaufen als Zugriffe über das Netzwerk. Diese Architektur ist relativ einfach, da die Anwendungen „im Stück“, d.h. als EAR-File deployt werden können. Zudem werden keine unnötigen Remote-Aufrufe durchgeführt. Nachteilig ist, dass ein Serverausfall während eines länger laufenden Requests normalerweise nicht von einer anderen Instanz aufgefangen werden kann.

Will man die Verfügbarkeit des Gesamtsystems erhöhen, kann man mehrere Cluster für unterschiedliche Teilaufgaben aufbauen, indem man beispielsweise Web- und Logikanteil trennt, wie es in Ab-

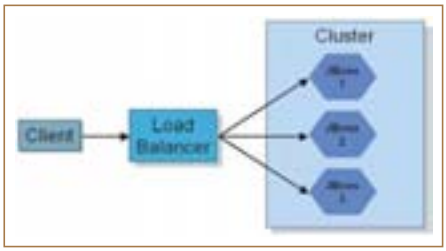

Abb. 1: Clustering-Grundprinzip

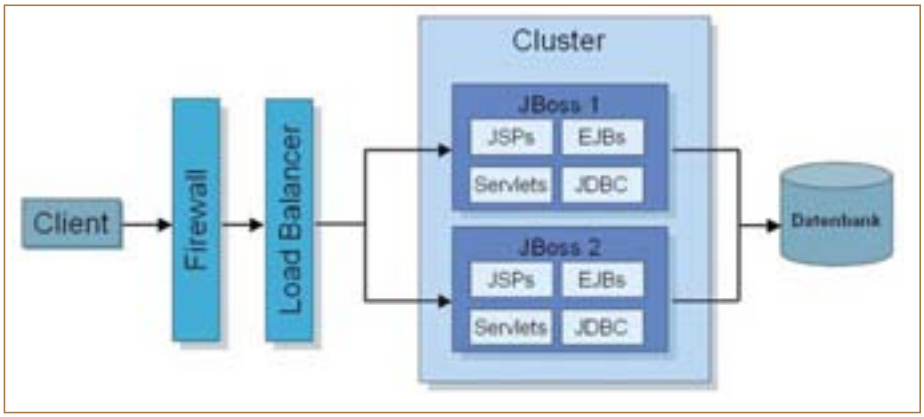

Abb. 2: Basisarchitektur

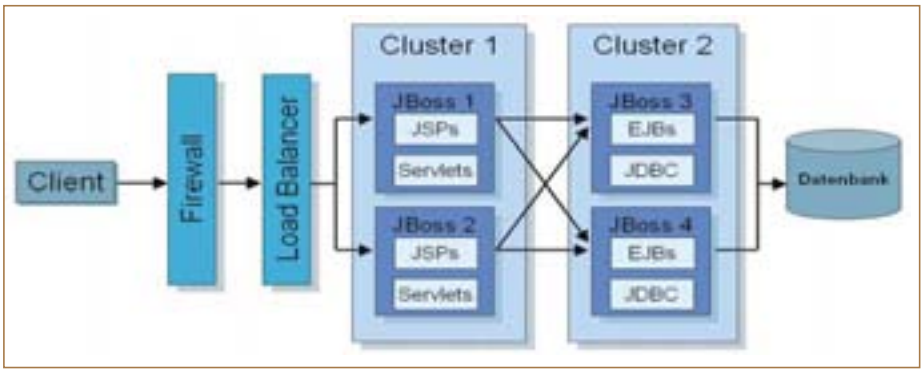

Abb. 3: Mehrstufiger Cluster

bildung 3 dargestellt ist. Vorteilhaft ist hier, dass der Ausfall einer Logikkomponente auch innerhalb der Verarbeitung eines (Web-) Requests durch ein Fail Over zu einem anderen Knoten behoben werden kann. Zudem kann die Anzahl der Cluster-Knoten in Web- und Logikanteil passend zur jeweiligen Anwendung auch unterschiedlich sein. Die höhere Verfügbarkeit und Flexibilität wird allerdings erkauft durch eine komplizierteres Deployment und einen höheren Anteil von vergleichsweise langsamen Remote-Aufrufen.

Bei allen diesen Betrachtungen muss natürlich bedacht werden, dass eine Kette nur so stark ist wie ihr schwächstes Glied. In den dargestellten Architekturen stellen Load Balancer und Datenbank jeweils einen Single-Point-of-Failure dar. Bei ihrem Ausfall ist keine weitere Verarbeitung mehr möglich. Diese Komponenten müssen also ebenfalls redundant ausgelegt werden, wenn man ein hoch verfügbares Gesamtsystem erhalten will.

Load Balancing von Client-Anfragen

Als Lastverteiler an der Front des Systems bieten sich mehrere Lösungen von simpel bis komplex an: Viele DNS-Server sind in der Lage für einen Rechnernamen der Reihe nach mehrere IP-Adressen zu liefern. Dieses „DNS-Round-Robining“ ist also auf einfachste Weise in der Lage, Requests auf die Knoten eines Clusters zu verteilen. Allerdings hat diese Einfachheit auch Ihren Preis. Ein DNS-Server hat im Allgemeinen keine Kenntnis über den Zustand der Cluster-Knoten. Ein ausgefallener Knoten wird also weiterhin gnadenlos mit Requests versorgt. Zudem findet eine evtl. vorhandene Sitzungsinformation keinerlei Beachtung. Zwei Requests einer Sitzung werden daher mit hoher Wahrscheinlichkeit von verschiedenen Cluster-Knoten verarbeitet, was eine passende Konfiguration des Clusters hinsichtlich der Replikation von Sitzungsinformationen voraussetzt.

Alternativ kann ein Hardware Load Balancer eingesetzt werden. Diese Geräte arbeiten ähnlich wie Router oder Switches, können aber Requests anhand vielfältiger Regeln auf verschiedene Knoten verteilen.

Für Architekturen mit einem Web-Frontend bietet sich zudem der Einsatz eines Webservers als Proxy-Server an. So ist z.B. der Apache Webserver mit Hilfe des Moduls *mod_jk* in der Lage, Requests auf konfigurierbare URLs an dahinter liegende JBosse zu verteilen.

Für den Fall, dass kein Web-Frontend benutzt wird, sondern es sich um eine Anwendung mit EJBs und einem Java-Client handelt, beinhalten die vom JBoss generierten Client-Stubs die Logik für das Load Balancing und Fail Over.

Abb. 4: Partitionen

Clustering-Features von JBoss

JBoss hat mittlerweile recht komfortable Clustering-Features eingebaut. Die zueinander gehörenden Knoten eines Clusters finden sich per IP-Multicast beim Starten der Server automatisch. Außerdem werden die JNDI-Bäume der Server automatisch miteinander synchronisiert. Änderungen auf einem Server werden umgehend auf die restlichen Mitglieder des Clusters repliziert. Servlets, JSPs und EJBs können im Cluster betrieben werden. Sie werden dazu auf allen Cluster-Knoten deployt und im Betrieb über einen Load Balancer lastverteilt angesprochen. Die Sitzungsinformationen von Servlets bzw. JSPs und Stateful Session EJBs können im Cluster repliziert werden, um im Falle des Ausfalls eines Knotens mit einer anderen JBoss-Instanz weiter arbeiten zu können. Für das Clusterweite Deployment steht der Farm-Service zur Verfügung, der das Deployment überwacht und dafür sorgt, dass alle im Cluster beteiligten Instanzen dieselben Anwendungen deployt haben. Für Java-Clients besteht die Möglichkeit per IP-Multicast auf einen Server im Cluster zuzugreifen, damit man nicht alle IP-Adressen der im Cluster beteiligten Rechner beim Client hinterlegen muss. Außerdem steht ein Singleton-Service zur Verfügung, sodass von einem spezifizierten Dienst im Cluster immer nur einer aktiv ist. Fällt der Server im Cluster aus, auf dem der Dienst momentan läuft, sorgt der Singleton-Service dafür, dass der Dienst auf der nächsten verfügbaren Cluster-Instanz gestartet wird. Damit kann man z.B. einen ausfallsicheren Scheduler konfigurieren.

Ein ganz besonders interessantes Feature ist auch die Cache-Invalidierung bei Entity Beans. Man kann dabei die EJBs so deployen, dass sie ihre Informationen cachen, was eine wesentlich verbesserte Performance bringt. Dies führt in der Regel allerdings dazu, dass beim Einsatz eines Clusters der Server „B“ nicht weiß, dass auf Server „A“ ein Update ausgeführt wurde. Abhilfe zu diesem Problem schafft der JBoss *Cache-Invalida-*

tion-Service, den man im Cluster so konfigurieren kann, dass er die anderen beteiligten JBoss-Instanzen informiert, wenn eine Entity Bean auf einem Server verändert oder gelöscht wurde.

Konfiguration des Clustering

JBoss benutzt intern und in der Konfiguration den Begriff der Partition. Diese sind i.w. durch einen eindeutigen Namen gekennzeichnet, der einer JBoss-Instanz zugeordnet werden kann. Alle Instanzen einer Partition tauschen Informationen über ihren Zustand und ihre Anwendungen aus. Eine Partition stellt damit einen Cluster dar. Es können beliebig viele Partitionen im Firmennetz existieren (Abb. 4).

Konfiguriert wird solch eine Partition und damit die Zugehörigkeit zu einem Cluster in der Konfigurationsdatei *cluster-service.xml*. In der bei JBoss mitgelieferten *all*-Konfiguration befindet sich diese Datei bereits und der Partitionsname ist auf *DefaultPartition* voreingestellt. Um also einen JBoss-Cluster zu starten, brauchen Sie nichts weiter zu tun, als in demselben Netzwerksegment auf zwei Rechnern jeweils einen JBoss mit der *all*-Konfiguration zu starten.

Im Folgenden werden sehr häufig die Buchstaben ‚HA' auftauchen. Sie stehen bei JBoss-Clustering für *High Availability* und bezeichnen alles, was mit Clustering zu tun hat.

Konfiguration des *HANaming-Service*

```
1. <mbean code="org.jboss.ha.jndi.
   HANamingService" name="jboss:
   service=HAJNDI">
2. <depends>jboss:service=
   DefaultPartition</depends>
3. <attribute name="PartitionName">
   DefaultPartition</attribute>
4. <attribute name="RmiPort">0</attribute>
5. <attribute name="Port">1100</attribute>
6. <attribute name="Backlog">50</attribute>
7. <attribute name="AutoDiscoveryAd-
   dress"> 230.0.0.4</attribute>
8. <attribute name="AutoDiscoveryGroup">
   1102</attribute>
9. </mbean>
```

Wie wir bereits im ersten Artikel unserer Serie erwähnt haben, basieren alle JBoss-Dienste auf den so genannten JMX MBeans, und auf diese Art und Weise ist natürlich auch das Clustering implementiert. Dafür muss als aller erstes die MBean namens

```
org.jboss.ha.framework.server.ClusterPartition
```

deployt werden. Ihr wird als Parameter zum einen der Partitionsname mitgegeben und zum anderen bekommt sie die Parameter zur Konfiguration des JGroups-Toolkit [2]. Das JGroups-Toolkit bildet die Grundlage der Kommunikation der einzelnen im Cluster beteiligten JBoss-Instanzen und ist für das IP-Multicasting verantwortlich.

Alle weiteren Cluster-Dienste bekommen in der Regel den MBean-Namen der *ClusterPartition*-MBean als Abhängigkeit definiert und sie benötigen auch den Partitionsnamen. Der Kasten „Konfiguration des *HANamingService*" zeigt beispielhaft, wie eine MBean, die auf der Cluster MBean aufsetzt, konfiguriert wird. Die *HANamingService*-MBean ist für das Replizieren des JNDI-Baumes im Cluster verantwortlich. In der ersten Zeile wird die Klasse der MBean angegeben und ihr eigener JMX-Name. In der zweiten Zeile steht der Name, unter dem die *ClusterPartition*-MBean beim MBean-Server registriert wurde. Die dritte Zeile gibt den Partitionsnamen an. Alle weiteren Attribute sind Parameter für die *HANamingService*-MBean, so z.B. die Multicast-Adresse und der Port für das *Auto-Discovery*.

Clustering von Session und Entity EJBs

In diesem Abschnitt zeigen wir was beachtet werden muss, wenn EJBs in einem geclusterten Umfeld benutzt werden sollen. Zunächst kann gesagt werden, dass die einfachste Architektur in der Regel die beste ist. Es ist zwar möglich sehr komplexe Architekturen, bei denen unterschiedliche EJBs auf unterschiedlichen Cluster-Instanzen deployt sind, mit JBoss Clustering zu erzeugen, aber das bedeutet auch einen sehr hohen administrativen Aufwand. Und aus Performancegesichtspunkten ist

es auch nicht zu empfehlen, wenn eine EJB Remote-Aufrufe machen muss, um ihre Arbeit zu erledigen.

Stateless Session EJBs lassen sich am einfachsten clustern, da kein Status vorhanden ist, der beachtet werden muss. Aufrufe können an jede Instanz im Cluster weitergegeben werden, auf der die EJB deployed ist. Um eine EJB zu clustern, muss der *jboss.xml* Deployment Descriptor modifiziert werden. Dabei reicht es aus, nur das <*clustered*>-Tag mit aufzunehmen. Alle anderen Tags (Sub-Elemente von <*cluster-config*>) sind optional. Listing 1 zeigt alle Cluster-Tags mit ihren Default-Werten.

Stateful Session EJBs lassen sich natürlich auch clustern, allerdings ist hier der Aufwand wesentlich höher, denn der Status muss beachtet werden. Momentan benutzt JBoss eine In-Memory Replizierung des Status zwischen den beteiligten Cluster-Instanzen. Das heißt, dass bei jeder Änderung des Status einer Stateful Session EJB, dieser repliziert und im Cluster synchronisiert wird. Um diese Verteilung zu koordinieren, gibt es einen Service, der auf jeden im Cluster laufenden JBoss aktiviert sein muss. Dabei handelt es sich natürlich wieder um eine MBean, die *HASessionState*-MBean. Sie wird auch in der *cluster-service.xml*-Datei konfiguriert. Nachdem die *HASessionState*-MBean konfiguriert ist, muss man auch bei Stateful Session EJBs das <*clustered*>-Tag im *jboss.xml* Deployment Descriptor aufnehmen, um Clustering zu aktivieren. Listing 2 zeigt alle weiteren optionalen Tags für Stateful Session EJBs. Wie man an den Tags für die Load-Balance-Policy sehen kann, werden die Methoden des Home-Interface zunächst per *RoundRobin* verteilt. Wenn dann der Remote-Stub zur Verfügung steht, wird nicht mehr *RoundRobin* angewendet, sondern alle weiteren Aufrufe landen bei der ersten Instanz aus der Liste aller vorhandenen Cluster-Instanzen. Da der Replizierungsprozess sehr kostspielig ist, kann man als Bean-Entwickler eine Methode mit folgender Signatur in seine Stateful Session EJB einbauen:

```
public boolean isModified();
```

Anhand dieser Methode kann JBoss erkennen, ob der Status abgeglichen werden muss oder nicht.

Entity EJBs benötigen ebenso das <*clustered*>-Tag in ihrem Deployment Descriptor. Und auch hier sind alle anderen Cluster-spezifischen Tags optional. Listing 3 zeigt den Deployment Descriptor einer Entity EJB. Bei Entity EJBs ist zu beachten, dass JBoss momentan noch keinen im Cluster verteilten Sperrmechanismus besitzt. Das bedeutet, dass bei geclusterten Entity EJBs das row-level Locking der Datenbank genutzt werden muss, um sie zu synchronisieren. Dies wird aktiviert, indem man bei der Konfiguration der zu verwendenden Datasource bei dem JDBC-Treiber den Transaction Isolation Level auf *TRANSACTION_SERIALIZABLE* setzt. Des Weiteren besitzt JBoss kein verteiltes Caching für Entity EJBs. Hier muss man also Commit Option ‚B' verwenden, damit vor dem Start einer Transaktion die Daten wieder aus der Datenbank gelesen werden, für den Fall, dass eine andere Cluster-Instanz die Entity EJB modifiziert hat. Wie schon erwähnt, bietet JBoss an dieser Stelle allerdings den „Cache-Invalidation-Service" an, den wir hier allerdings nicht näher betrachten.

Message Driven EJBs sind momentan nicht Cluster-fähig, da die JMS-Implementierung auch noch kein Clustering unterstützt. Hier wird allerdings momentan dran gearbeitet, sodass man in diesem

Listing 1

```
<jboss>
 <enterprise-beans>
  <session>
   <ejb-name>cluster.StatelessSession</ejb-name>
   <jndi-name>cluster.StatelessSession</jndi-name>
   <clustered>True</clustered>
   <cluster-config>
    <partition-name>DefaultPartition</partition-name>
    <home-load-balance-policy>
     org.jboss.ha.framework.interfaces.RoundRobin
    </home-load-balance-policy>
    <bean-load-balance-policy>
     org.jboss.ha.framework.interfaces.RoundRobin
    </bean-load-balance-policy>
   </cluster-config>
  </session>
 </enterprise-beans>
</jboss>
```

Listing 2

```
<jboss>
 <enterprise-beans>
  <session>
   <ejb-name>cluster.StatefulSession</ejb-name>
   <jndi-name>cluster.StatefulSession</jndi-name>
   <clustered>True</clustered>
   <cluster-config>
    <partition-name>DefaultPartition</partition-name>
    <home-load-balance-policy>
     org.jboss.ha.framework.interfaces.RoundRobin
    </home-load-balance-policy>
    <bean-load-balance-policy>
     org.jboss.ha.framework.interfaces.FirstAvailable
    </bean-load-balance-policy>
    <session-state-manager-jndi-name>
     /HASessionState/Default
    </session-state-manager-jndi-name>
   </cluster-config>
  </session>
 </enterprise-beans>
</jboss>
```

Listing 3

```
<jboss>
 <enterprise-beans>
  <entity>
   <ejb-name>cluster.Entity</ejb-name>
   <jndi-name>cluster.Entity</jndi-name>
   <clustered>True</clustered>
   <cluster-config>
    <partition-name>DefaultPartition</partition-name>
    <home-load-balance-policy>
     org.jboss.ha.framework.interfaces.RoundRobin
    </home-load-balance-policy>
    <bean-load-balance-policy>
     org.jboss.ha.framework.interfaces.FirstAvailable
    </bean-load-balance-policy>
   </cluster-config>
  </entity>
 </enterprise-beans>
</jboss>
```

Listing 4

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems,
Inc.//DTD Web Application 2.3//EN""http://java.sun.
                                com/dtd/web-app_2_3.dtd">
<web-app>
<distributable/>
 ...
</web-app>
```

Bereich in naher Zukunft sicherlich etwas erwarten kann.

Zugriff auf geclusterte Web-Applikationen

Beim Zugriff auf eine Web-Applikation von einem http-Client muss man in einer geclusterten Umgebung zwei Szenarien unterscheiden. Bei dem ersten Szenario geht es darum, eine gut skalierbare Anwendung zu haben, die viele Requests verarbeiten kann und daher auf mehreren Rechnern zur Verfügung gestellt wird. Wir brauchen dafür, wie zu Beginn schon erwähnt, einen so genannten Load Balancer, der unsere Client-Anfragen verteilt. Dabei kann es allerdings zu einem Problem kommen. Nämlich dann, wenn unsere Web-Applikationen Sessions benutzen. Es könnte dann passieren, dass der erste Request auf Server „A“ landet, wo für unseren Client eine Session erzeugt wird, der zweite Request vom Load Balancer allerdings auf Server „B“ geschickt wird. Dort befindet sich für unseren Client allerdings noch keine Session und es kann zu unvorhergesehenen Problemen kommen. Dies lässt sich dann vermeiden, wenn der Load Balancer so genannte Sticky-Sessions unterstützt. Das

Load-Balancer verteilt Client-Anfragen

bedeutet, dass der erste Request noch verteilt wird, aber alle nachfolgenden Requests unseres Clients immer wieder zu dem Server geschickt werden, auf dem das erste Mal für den Client eine Session erzeugt wurde.

Das zweite Szenario verlangt neben einer guten Skalierung auch noch Ausfallsicherheit. Dies lässt sich dann erreichen, wenn unsere Session-Informationen im Cluster repliziert werden und der eingesetzte Load Balancer in der Lage ist, ausgefallene Server zu erkennen und Client-Anfragen automatisch an den nächsten Rechner im Cluster weitergibt.

Für beide Szenarien lässt sich sehr gut ein Apache Webserver mit dem *mod_jk*-Modul [3] benutzen. Für den ersten Fall braucht man auf JBoss-Seite gar nichts zu tun, da die Sticky-Sessions automatisch von dem *mod_jk*-Modul gehandelt werden. Wenn man allerdings eine Replizierung der Session-Informationen im Cluster benötigt, dann muss auf JBoss-Seite die Datei *jbossha-httpsession.sar* deployt werden. Sie befindet sich standardmäßig allerdings auch schon in der *all*-Konfiguration. Sowohl JBoss/Tomcat als auch JBoss/Jetty sind in der Lage diesen Service für die Session-Replizierung zu nutzen. Jetzt muss man nur noch seiner Web-Applikation sagen, dass sie in einer verteilten Umgebung läuft. Dafür ist bereits in den Spezifikationen das <*distributable*>-Tag vorgesehen, welches man entsprechend in dem *web.xml* Deployment Descriptor angeben muss (Listing 4).

Zugriff auf geclusterte EJBs

In dem vorherigen Abschnitt haben wir uns schon angeschaut, was man beachten

muss, wenn man per HTTP auf geclusterte Web-Applikationen zugreifen will. In diesem Abschnitt zeigen wir die Möglichkeiten, die ein Java-Client hat, um auf einen JBoss-Cluster zuzugreifen, wenn er EJBs benutzen will. Wie wir wissen, muss sich der Client zunächst den JNDI-Context besorgen und dann ein Lookup auf das zu benutzende EJB machen. Bei einem standalone Server braucht der Client dazu die IP-Adresse und evtl. die Portnummer des JNDI-Servers.

Bei einem Cluster von JBoss-Servern ist das nicht ganz so leicht. So weiß der Client z.B. nicht, welcher Server gerade läuft und welcher evtl. momentan ausgefallen ist. Der Client muss also alle Server kennen, die in dem Cluster beteiligt sind. Man kann daher bei dem ersten Zugriff auf den JNDI-Server eine kommaseparierte Liste von IP-Adressen und Ports mitgeben:

```
java.naming.provier.url=server1:1100,server2:1100,
                        server3:1100,server4:1100
```

Der Client-Code, der sich in der *InitialContextFactory* befindet, versucht jeden Server in der Liste zu erreichen und benutzt dann den ersten aktiven Server, der gefunden wird. Bei dem Stub, der dann vom Server heruntergeladen wird, handelt es sich um einen so genannten Smart-Stub. Dieser beinhaltet Logik, um auf andere Server im Problemfall auszuweichen und auch um seine Liste, der im Cluster aktiven Rechner, upzudaten.

In sehr dynamischen Umgebungen, wo Server häufig gestartet und gestoppt werden oder umgezogen und mit anderen IP-Adressen versehen werden, kann es eine frustrierende Arbeit sein, immer wieder die Liste der IP-Adressen auf den Clients aktuell zu halten. Erst recht, wenn es sich um mehrere hundert Clients handelt, die keine automatische Funktionalität für solch ein Update haben. Aus diesem Grunde beinhaltet die JNDI-*InitialContextFactory* von JBoss ein neues Feature, was die Entwickler *auto-discovery* genannt haben. Das bedeutet, dass wenn die *Property java.naming.provier.url* leer ist oder keiner der aufgelisteten Server erreicht werden kann, dann wird versucht über einen IP-Multicast Rundruf von irgendeinem Server den Smart-Stub zu bekommen. So ist der Client in der Lage Zugriff auf den Cluster zu erhalten, ohne dass irgendeine IP-Adresse beim Client konfiguriert werden muss. Man kann dieses *auto-discovery* noch mit einer Reihe von speziellen Properties für die *InitialContextFactory* beeinflussen (Tab. 1).

Farm Service

Der Farm Service ist eine interessante Erweiterung zu einem JBoss-Cluster. Er wird über die Konfigurationsdatei *farm-service.xml* deployt. In dieser Datei muss zum einen der Partitionsname der zu verwendenden Partition und zum anderen ein Ordner konfiguriert werden. Dieser Ordner dient dann als zusätzliches Deployment-Verzeichnis und er wird automatisch bei allen im Cluster beteiligten Instanzen abgeglichen. Man muss dazu den Farm Service auf allen Instanzen deployen, die an diesem Dienst teilnehmen sollen und der Ordner, der in der Konfiguration angegeben ist, muss vorhanden sein.

Wenn man dann eine zu deployende Anwendung (z.B. eine EAR-File) auf einem JBoss in das Verzeichnis des Farm Service kopiert, verteilt JBoss die Datei automatisch auf alle anderen beteiligten Instanzen im Cluster und die Datei landet in dem jeweiligen für den Farm Service konfigurierten Ordner. Dies funktioniert auch mit einem Undeployment, d.h. wenn in einem Farm Service-Verzeichnis eine Anwendung gelöscht wird, wird sie auch bei allen anderen Instanzen gelöscht.

In der *all*-Konfiguration befindet sich auch bereits eine fertige *farm-service.xml*-Datei, in der die *DefaultPartition* und als Farm-Verzeichnis ein Ordner namens *farm* konfiguriert sind. Weitere Parameter werden anhand von Kommentaren erläutert.

Fazit

Zusammenfassend kann festgehalten werden, dass JBoss im Bezug auf Clustering alle notwendigen Features mitbringt, die auch die kommerziellen Produkte aufweisen. Das Clustering lässt sich an einer zentralen Stelle konfigurieren und ist sehr einfach zu aktivieren. Dieses Feature wird mittlerweile auch schon von vielen Anwendern eingesetzt und wir haben bisher nur gute Erfahrungen damit gemacht.

In der nächsten Ausgabe des *Java Magazins* stellen wir Ihnen das Eclipse-Plugin JBoss-IDE vor.

Dirk Weil ist Geschäftsführer der Gedoplan GmbH in Bielefeld [4]. Marcus Redeker ist Senior Java Consultant im gleichen Unternehmen. Beide sind zertifizierte JBoss Consultants und seit mehr als fünf Jahren als Berater und Trainer im Bereich Java und Enterprise Java tätig.

Property	Beschreibung
jnp.disableDiscovery	Wenn dieser Wert auf *true* steht wird das *auto-discovery* Feature deaktiviert. Standard ist *false*.
jnp.partitionName	Wenn man mehrere Cluster mit unterschiedlichen Partitionsnamen im gleichen Netzwerk konfiguriert hat, kann man hiermit festlegen für welchen Cluster man sich interessiert. Wenn diese Property nicht angegeben wird (Standard) wir der erste Server benutzt, der antwortet; unabhängig von seinem Partitionsnamen.
jnp.discoveryTimeout	Wie lange soll der Client auf eine *auto-discovery* Anfrage warten. Standard ist 5000 ms.
jnp.discoveryGroup	Legt fest, welche IP-Multicast Adresse für das auto-discovery benutzt werden soll. Standard ist 230.0.0.4 (Wird in der *cluster-service.xml* für die HANamingService-MBean angegeben).
jnp.discoveryPort	Legt fest, welcher Port für die Multicast-Anfrage genutzt werden soll. Standard ist 1102.

Tab. 1: HA-JNDI-spezifische Properties für die *InitialContextFactory*

Links & Literatur

[1] www.jboss.org/

[2] www.jgroups.org/

[3] jakarta.apache.org/tomcat/tomcat-4.1-doc/jk2/

[4] www.gedoplan.de/