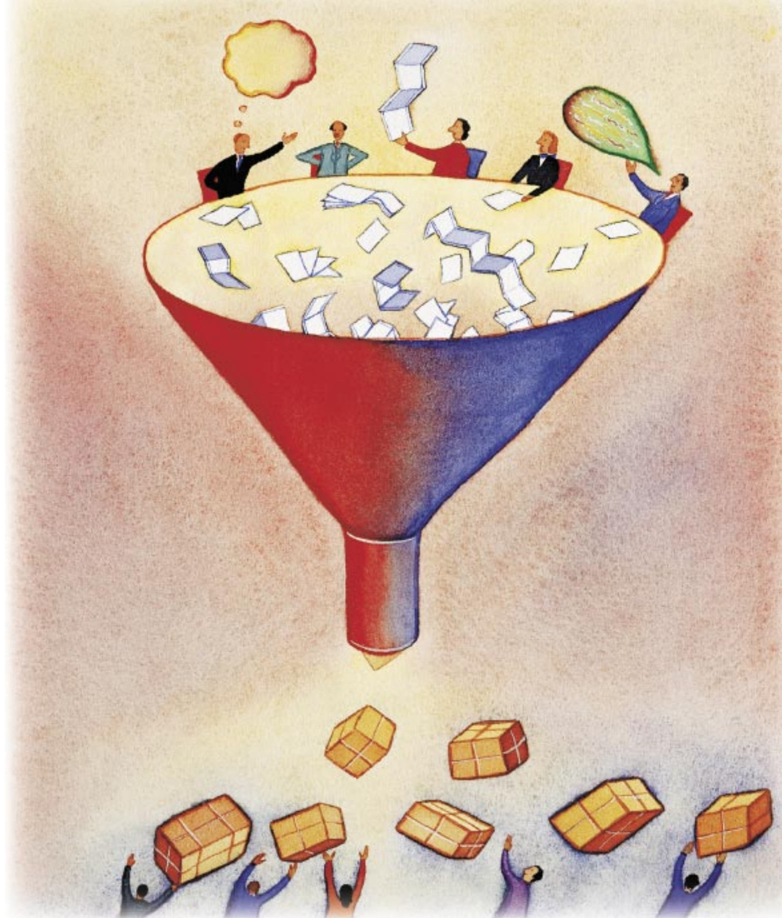

Auf vielen Schultern

Clustering mit dem JBoss Application Server

Martin Renner

Applikations-Server nach dem J2EE-Standard von Sun sind aus modernen Anwendungen kaum noch wegzudenken. Wegen dieser zentralen Rolle werden an diese Systeme besondere Anforderungen in Bezug auf Ausfallsicherheit und/oder Lastverteilung gestellt. Es ist daher eine logische Konsequenz, diese Systeme in einem Cluster zu betreiben.

Der JBoss Application Server erfreut sich großer Beliebtheit, nicht zuletzt deshalb, weil er in der J2EE-Welt zu den Trendsettern gehört, schlank in der Administration und natürlich quelloffen ist. Bezüglich der Stabilität ist er schon lange für Einsätze im Produktivumfeld prädestiniert. Im Folgenden werden die Konzepte des JBoss Clustering im Detail erläutert und um nützliche Tipps aus der Praxis ergänzt.

Kurz und bündig

Der JBoss-Server ist bereits für den Cluster-Betrieb fertig konfiguriert, es muss nur die richtige Konfiguration („all") gewählt werden. Um Probleme zu vermeiden, sollte dem JBoss-Server zusätzlich mitgeteilt werden, über welche Netzwerkkarte er erreicht werden kann. Dazu wird ihm entweder der Hostname oder die IP-Adresse des Rechners übergeben, auf dem er ausgeführt wird. Der Befehl zum Starten des JBoss-Server auf dem Rechner „MeinHost1" sieht also so aus:

```
run –c all --host=MeinHost1
```

Auf die gleiche Weise können nun beliebige weitere Knoten gestartet werden, die automatisch einen Cluster formen.

Damit der JBoss-Server eine EJB als Cluster-tauglich erkennt, muss diese im Deployment-Deskriptor („jboss.xml") entsprechend gekennzeichnet werden. Der folgende Deployment-Deskriptor macht dies für eine Stateless Session-Bean:

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>Test</ejb-name>
      <jndi-name>ejb/Test</jndi-name>
      <clustered>true</clustered>
    </session>
  </enterprise-beans>
</jboss>
```

Das fertige EAR (oder JAR) braucht nur auf einem einzigen Knoten deployt zu werden, und zwar in das JBoss-Verzeichnis „server/all/farm". Der JBoss-Server kümmert sich dann automatisch darum, dass das Archiv auch auf alle anderen Knoten verteilt wird. Dieser Mechanismus nennt sich „Farm Service" und wird über die Datei „server/all/deploy/deploy.last/farm-service.xml" gesteuert.

Im Client wird beim JNDI-Lookup als einziger Parameter die Factory gesetzt:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
```

Eine URL wird nicht angegeben. Fertig ist der Cluster. Der Client sucht nun automatisch (weil keine URL angegeben wurde) per IP-Multicast nach einem Serververbund und verteilt die Aufrufe nach dem *Round-Robin*-Verfahren auf die verfügbaren Cluster-Knoten.

Architektur

Um eine ernsthafte Applikation im Cluster betreiben zu können, ist ein tieferes Verständnis der Architektur und der Konzepte erforderlich.

Die einzelnen Knoten, die einen JBoss-Cluster formen, sind logisch in einer so genannten *Partition* zusammengefasst. Ohne Änderungen an der Konfiguration gehört ein Knoten automatisch zur Partition `DefaultPartition`. Die einzelnen Knoten kommunizieren untereinander über IP-Multicast und UDP. Pro Partition gibt es einen Master, der automatisch bestimmt wird. Dieser Master sendet alle paar Sekunden per IP-Multicast ein „Ping" aus. Alle Knoten, die dieses „Ping" empfangen und zur selben Partition gehören, melden sich beim Master und bekommen von ihm eine Liste aller anderen Cluster-Knoten geliefert. Fortan kennt jeder Knoten jeden anderen und kann direkt mit jedem einzelnen kommunizieren (per UDP). Da der Master sein „Ping" alle paar Sekunden aussendet, kann er schnell feststellen, ob neue Knoten hinzugekommen sind, oder ob sich ein Knoten nicht mehr meldet. Er kann dann die anderen Knoten entsprechend benachrichtigen. Falls das „Ping" vom Master ausbleibt, bestimmen die verbleibenden Knoten einen neuen Master.

Es ist ersichtlich, dass IP-Multicast eine Grundvoraussetzung für einen funktionierenden Cluster ist. Werden die einzelnen Knoten über mehrere Subnetze verteilt (beispielsweise an unterschiedlichen Standorten, um die Ausfallsicherheit zu erhöhen), so muss auch gewährleistet sein, dass auf allen dazwischen liegenden Routern das Multicast-Routing korrekt konfiguriert ist. Bei manchen Unix-Systemen ist darauf zu achten, dass bei der entsprechenden Netzwerkkarte das Multicast-Flag gesetzt ist. Ebenso ist bei manchen Unix-Systemen das Multicast-Routing explizit zu setzen. Unter Linux und Windows sind diese beiden Punkte im Normalfall schon richtig konfiguriert.

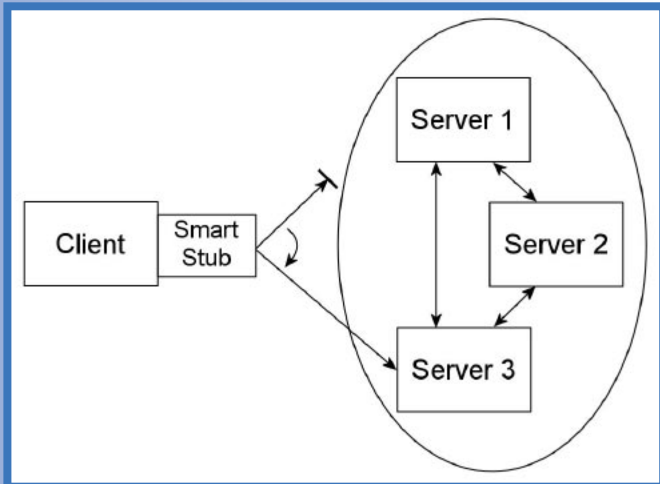

Abb. 1: Cluster-Architektur mit Smart Stubs

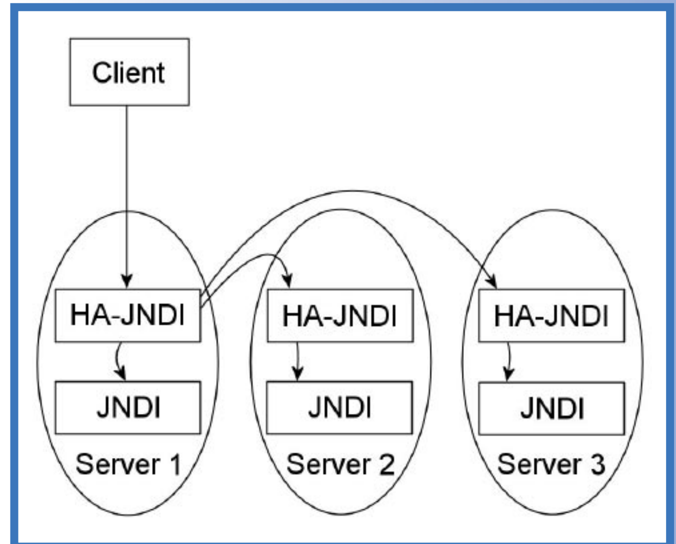

Abb. 2: JNDI-Lookup im Cluster

Die eigentliche Logik des Fail-Over und Load-Balancing befindet sich beim JBoss in so genannten *Smart Stubs* im Client. Im Gegensatz zu anderen Cluster-Lösungen kommt keine Lösung mit einem (oder mehreren) zentralen Dispatchern zum Einsatz, da diese genau so wie ein normaler Server-Knoten auch ausfallen können und somit kein höheres Maß an Sicherheit bieten. Ganz im Gegenteil: Dispatcher verursachen nur einen höheren Aufwand bei der Konfiguration und der Administration. Pro EJB gibt es einen Smart Stub für das Home-Interface und einen für das Remote-Interface. Der Client lädt den Smart Stub für ein Home-Interface automatisch bei einem JNDI-Lookup vom Server per RMI herunter. Der Smart Stub für ein Remote-Interface wird beim jeweiligen „create"-Aufruf übermittelt. Der Client erhält dabei den eigentlichen Code zum Aufruf der entfernten EJB (mit der entsprechenden Load-Balancing-Strategie) und zusätzlich eine Liste der Knoten, auf welchen diese EJB gefunden werden kann.

Bei jedem Remote-Aufruf übermittelt der Smart Stub die Versionsnummer der Cluster-Topologie, die er aktuell kennt, an den Server. Stellt der Server fest, dass es inzwischen eine Änderung gegeben hat (Knoten kamen hinzu oder fielen weg), so bekommt der Client zum eigentlichen Rückgabewert zusätzlich die neue Liste der Knoten übermittelt. Der Client bzw. der Smart Stub weiß somit von jeder EJB, auf welchen Knoten sie erreicht werden kann. Schlägt ein Aufruf fehl, weil der entsprechende Knoten ausgefallen ist, so kann der Smart Stub aus seiner Liste transparent einen neuen Knoten aussuchen, ohne dass die Client-Anwendung etwas davon mitbekommen würde.

Abbildung 1 zeigt einen Cluster, der aus drei Knoten besteht. Die drei Knoten gehören zur selben Partition und kommunizieren per IP-Multicast untereinander. Der Client kommuniziert über den Smart Stub mit dem Cluster. In der Abbildung ist skizziert, dass der Smart Stub aus irgendeinem Grund nicht mehr mit dem Knoten „Server 1" kommunizieren kann. Aus seiner Liste der möglichen Knoten wählt er daraufhin den Knoten „Server 3" zur weiteren Kommunikation aus. Würde der Knoten „Server 3" in diesem Szenario ebenfalls nicht mehr mit „Server 1" kommunizieren können, so würde er eine neue Liste der Knoten (bestehend aus „Server 3" und „Server 2") zusätzlich zum eigentlichen Funktionsergebnis an den Smart Stub zurückliefern.

JNDI

Im Cluster-Betrieb wird der bekannte JNDI-Dienst um einen Cluster-fähigen JNDI-Dienst namens HA-JNDI (*High Availability*) ergänzt. Auf jedem Knoten läuft eine Instanz dieses HA-JNDI, Änderungen an einer Instanz werden auf alle anderen Instanzen im Cluster repliziert. Somit umspannt der HA-JNDI den kompletten Cluster, während der normale JNDI auf einen Knoten begrenzt ist.

Bei einem Lookup auf den HA-JNDI prüft dieser, ob er das gewünschte Objekt kennt. Wenn nicht, delegiert er den Aufruf an den lokalen JNDI weiter. Kennt auch dieser das Objekt nicht, leitet der HA-JNDI die Anfrage an alle anderen HA-JNDIs im Cluster weiter, welche ihrerseits ihre lokalen JNDIs befragen. Abbildung 2 zeigt diesen Ablauf.

Home-Interfaces von EJBs werden von JBoss immer in den lokalen JNDI des Knotens gebunden, auf dem das EJB deployt wurde. Durch den vorgeschalteten HA-JNDI sind diese Interfaces jedoch trotzdem über den kompletten Cluster zu erreichen. Die Anweisung `Context ctx = new InitialContext()` innerhalb des Applikations-Servers arbeitet aus Performancegründen ebenfalls immer mit dem lokalen JNDI. Soll innerhalb des Applikations-Servers ein Lookup auf ein Objekt erfolgen, von dem bekannt ist, dass es lokal nicht verfügbar ist (beispielsweise, weil es nicht überall im Cluster deployt wurde), so muss explizit mit dem HA-JNDI gearbeitet werden, welcher standardmäßig auf Port 1100 läuft:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
    "org.jnp.interfaces.NamingContextFactory");
p.put(Context.PROVIDER_URL, "localhost:1100");
Context ctx = new InitialContext(p);
```

Clients werden üblicherweise den HA-JNDI benutzen, um EJBs im Cluster zu finden. Dazu kann der Client entweder direkt mit dem HA-JNDI eines Knotens kommunizieren, oder er benutzt ein Feature, welches ihn automatisch per IP-Multicast mit einem HA-JNDI des Clusters verbindet. Mehr zu diesen beiden Möglichkeiten folgt später.

Dadurch, dass ein Lookup des Clients unter Umständen über den Cluster hinweg an alle verfügbaren lokalen JNDIs delegiert werden muss (was Zeit kostet), sollte sich der Client die Ergebnisse der Lookups lokal merken. Das macht auch deshalb Sinn, da der Client – wie bereits erwähnt – bei jedem Lookup auf ein bestimmtes Home-Interface den dazu passenden Smart Stub herunter lädt.

Server-Konfiguration

Die zentrale Konfiguration für den Cluster-Betrieb befindet sich in der Datei „cluster-service.xml" im Verzeichnis „server/all/deploy". Listing 1 zeigt die wichtigsten Abschnitte dieser Datei.

```
<mbean code="org.jboss.ha.framework.server.ClusterPartition"
      name="jboss:service=DefaultPartition">
  <!-- Name of the partition being built -->
  <attribute name="PartitionName">DefaultPartition</attribute>
  <!-- Determine if deadlock detection is enabled -->
  <attribute name="DeadlockDetection">False</attribute>
  <!-- The JGroups protocol configuration -->
  <attribute name="PartitionConfig">
    <Config>
      <!-- UDP: if you have a multihomed machine,
           set the bind_addr attribute to the appropriate NIC IP address -->
      <!-- UDP: On Windows machines, because of the media sense feature
           being broken with multicast (even after disabling media sense)
           set the loopback attribute to true -->
      <UDP mcast_addr="228.1.2.3" mcast_port="45566"
           ip_ttl="64" ip_mcast="true"
           mcast_send_buf_size="150000" mcast_recv_buf_size="80000"
           ucast_send_buf_size="150000" ucast_recv_buf_size="80000"
           loopback="false" />
      ...
    </Config>
  </attribute>
</mbean>
...
<mbean code="org.jboss.ha.jndi.HANamingService"
      name="jboss:service=HAJNDI">
  <depends>jboss:service=DefaultPartition</depends>
  <!-- Name of the partition to which the service is linked -->
  <attribute name="PartitionName">DefaultPartition</attribute>
  <!-- bind address of HA JNDI RMI endpoint -->
  <attribute name="BindAddress">${jboss.bind.address}</attribute>
  <!-- RmiPort to be used by the HA-JNDI service
      once bound. 0 => auto. -->
  <attribute name="RmiPort">0</attribute>
  <!-- Port on which the HA-JNDI stub is made available -->
  <attribute name="Port">1100</attribute>
  <!-- Backlog to be used for client-server RMI
      invocations during JNDI queries -->
  <attribute name="Backlog">50</attribute>

  <!-- Multicast Address and Group used for auto-discovery -->
  <attribute name="AutoDiscoveryAddress">230.0.0.4</attribute>
  <attribute name="AutoDiscoveryGroup">1102</attribute>
</mbean>
```

Listing 1: Cluster-Konfiguration in „cluster-service.xml“

Bei der MBean `ClusterPartition` können über das Element `UDP` die Multicast-Adresse und der Multicast-Port angegeben werden, über die die einzelnen Server im Cluster kommunizieren. Um Konflikte mit anderen JBoss-Installationen zu vermeiden, sollte zumindest der Port auf einen anderen Wert gesetzt werden.

Die MBean `HANamingService` regelt die Einstellungen für den HA-JNDI. Der Parameter „BindAddress“ legt fest, auf welche Adresse (und somit Netzwerkkarte) der HA-JNDI gebunden werden soll. Die Variable `${jboss.bind.address}` wird über den Kommandozeilenparameter `--host` gesetzt und enthält standardmäßig den Wert 0.0.0.0 (= alle Adressen des Rechners). Aufgrund eines Bugs im HA-JNDI sollte hier jedoch explizit eine Adresse angegeben werden. Entweder über den Kommandozeilenparameter `--host`, wie eingangs beschrieben, oder indem an dieser Stelle `${jboss.bind.address}` entfernt und durch eine konkrete IP-Adresse ersetzt wird.

Die Lösung mit dem Kommandozeilenparameter hat den Nachteil, dass dieser Wert über die Variable `${jboss.bind.address}` für alle Dienste verwendet wird. Setzt man diese Variable über den Kommandozeilenparameter auf eine IP-Adresse oder einen Host-Namen, so werden alle Dienste ausschließlich an diese Adresse gebunden. Ein Zugriff über „localhost“, beispielsweise `http://localhost:8080`, ist dann nicht mehr möglich.

Über die Parameter „AutoDiscoveryAddress“ und „AutoDiscoveryGroup“ der MBean `HANamingService` können die Multicast-Adresse und der Multicast-Port bestimmt werden, auf denen der HA-JNDI auf Client-Anfragen wartet. Der Client braucht dann nichts über die Cluster-Topologie zu wissen, um einen Einstiegspunkt in den Cluster zu erhalten.

In dieser Datei könnte auch der „PartitionName“ auf einen anderen Wert als „DefaultPartition“ gesetzt werden. Wird ein vom Standard abweichender Name benutzt, so muss dieser jedoch bei jeder EJB im Deployment-Deskriptor angegeben werden. Außerdem führt ein Suchen & Ersetzen in der Server-Konfiguration nicht zum Ziel, sondern zu Exceptions, da dieser Parameter nicht überall in der Konfiguration auftritt, wo er auch intern verwendet wird.

Stateless Session-Beans

Stateless Session-Beans (SLSB) können sehr einfach in einem Cluster eingesetzt werden, da kein Zustand im Cluster repliziert werden muss. Es ist egal, mit welchem Knoten ein Client bei einem EJB-Aufruf kommuniziert, die Funktionalität einer SLSB ist immer dieselbe. Im Deployment-Deskriptor braucht daher nur `<clustered>true</clustered>` angegeben zu werden.

Zusätzlich kann über den Deployment-Deskriptor gesteuert werden, wie der Client bzw. der Smart Stub die Aufrufe für das entsprechende EJB über den Cluster verteilen soll. Es ist zu beachten, dass die Strategie pro EJB und getrennt nach Home- und Remote-Interface angegeben wird. Zur Auswahl stehen hier drei verschiedene Implementierungen, welche auch für Stateful Session-Beans und Entity-Beans gelten (die gleichnamigen Klassen befinden sich im Package `org.jboss.ha.framework.interfaces`):

- RoundRobin: Die Aufrufe werden zyklisch über die vorhandenen Knoten verteilt. Der Knoten, mit dem begonnen wird, wird per Zufall aus der Liste gewählt.
- FirstAvailable: Beim ersten Aufruf wird per Zufall ein Knoten für dieses Interface ausgewählt. Der Smart Stub kommuniziert so lange mit diesem Knoten, bis dieser ausfällt. Dann wird ein neuer Knoten gewählt. Der gewählte Knoten gilt nur für den aktuellen Smart Stub. Da bei jedem Lookup bzw. bei jedem „create“-Aufruf ein neuer Smart Stub für das entsprechende Home- oder Remote-Interface angelegt wird, führt dies dazu, dass bei eben diesen Operationen jeweils ein neuer Knoten gewählt wird.
- FirstAvailableIdenticalAllProxies: Funktioniert im Prinzip wie FirstAvailable. Der gewählte Knoten gilt jedoch für alle Instanzen des Smart Stubs. Somit landen alle Aufrufe des Home- bzw. des Remote-Interfaces beim selben Knoten. Hier ist zu beachten, dass für das Home-Interface ein anderer Smart Stub verwendet wird als für das Remote-Interface. Es ist also sehr wahrscheinlich, dass alle Home-Aufruf bei einem Knoten und alle Remote-Aufrufe bei einem anderen Knoten landen. Aufrufe unterschiedlicher EJBs werden ebenfalls von unterschiedlichen Knoten bedient.

Listing 2 zeigt die Datei "jboss.xml" für eine SLSB, bei dem die Strategien explizit angegeben sind. Ohne Angabe der Strategie wird bei SLSBs für beide Interfaces *RoundRobin* verwendet.

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>Test</ejb-name>
      <jndi-name>ejb/Test</jndi-name>
      <clustered>true</clustered>      <cluster-config>
```

```
        <home-load-balance-policy>
        org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies
        </home-load-balance-policy>
        <bean-load-balance-policy>
        org.jboss.ha.framework.interfaces.FirstAvailableIdenticalAllProxies
        </bean-load-balance-policy>
      </cluster-config>
    </session>
  </enterprise-beans>
</jboss>
```

Listing 2: „jboss.xml“ für eine Stateless Session-Bean

Stateful Session-Beans

Bei Stateful Session-Beans (SFSB) hat der Container dafür zu sorgen, dass der Zustand einer SFSB-Instanz im Cluster repliziert wird. Beim JBoss erfolgt das Replizieren nach jedem Methodenaufruf, indem der Zustand der EJB über das Netzwerk zu allen anderen Knoten propagiert wird. Einige Konkurrenzprodukte erledigen dies ein bisschen eleganter, indem sie SFSBs nur auf einen einzigen Knoten des Clusters replizieren. Dadurch wird einiges an Kommunikations-Overhead gespart und die Wahrscheinlichkeit, dass genau die zwei Knoten ausfallen, die sich eine SFSB teilen, ist doch eher gering.

Damit nicht nach jedem Read-Only Aufruf der Zustand im kompletten Cluster repliziert wird, kann der Entwickler folgende Methode in SFSBs einbauen:

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

Liefert diese Methode den Wert **false** zurück, so wird der Zustand nicht repliziert.

Die Konfiguration im Deployment-Deskriptor gestaltet sich analog zu der einer SLSB. Standardmäßig werden Home-Interfaces per *RoundRobin* und Remote-Interfaces per *FirstAvailable* aufgerufen. Neben diesen beiden Elementen gibt es noch ein weiteres Element **<session-state-manager-jndi-name>**, mit dem der JNDI-Name des Services angegeben werden kann, der die Zustände repliziert. Wird dieses Element nicht explizit im Deployment-Deskriptor angegeben, so wird „/HASessionState/Default“ verwendet:

```
<cluster-config>
  <session-state-manager-jndi-name>
    /HASessionState/Default
  </session-state-manager-jndi-name>
</cluster-config>
```

Der entsprechende Service wird über die Datei „server/all/deploy/cluster-service.xml“ konfiguriert. Der Parameter „BeanCleaningDelay“ gibt an, nach wie vielen Millisekunden der replizierte Zustand einer SFSB gelöscht werden kann, wenn sich dieser nicht geändert hat *und* wenn während des Lebenszyklus dieser EJB ein Knoten ausgefallen ist. Der Wert „0“ steht für den Defaultwert von 30 Minuten.

Entity-Beans

Entity-Beans werden genau gleich konfiguriert wie SLSBs. Es muss also die Zeile **<clustered>true</clustered>** in den Deployment-Deskriptor „jboss.xml“ aufgenommen werden. Das Home-Interface wird standardmäßig per *RoundRobin* und das Remote-Interface per *FirstAvailable* aufgerufen. Die Synchronisation im Cluster erfolgt ausschließlich über die zugrunde liegende Datenbank. Der Sperrmechanismus für Änderungen erfolgt ebenfalls über die Datenbank. Es gibt zwei Möglichkeiten, diesen Sperrmechanismus zu aktivieren:

- Beim JDBC-Treiber wird der Transaction Isolation Level auf **TRANSACTION_SERIALIZABLE** gesetzt (aus Performancegründen nicht besonders ratsam).
- Es werden die *Row-Locking*-Funktionen der Datenbank benutzt.

Für letzteres muss in der Datei „server/all/conf/standardjbosscmp-jdbc.xml“ geprüft werden, ob für die verwendete Datenbank das Element **<row-locking-template>** richtig gefüllt ist. Entsprechende Beispiele finden sich in eben dieser Datei, wobei es im Prinzip um den String **Select ?1 From ?2 Where ?3 For Update** geht.

Da es keinen verteilten Cache-Mechanismus für Entity-Beans gibt, sollte bei CMP Entity-Beans *Commit Option B* und nicht *Commit Option A* verwendet werden. *Commit Option B* sorgt dafür, dass die EJB bei jedem Transaktionsbeginn die Daten von der Datenbank liest und nicht auf gecachte Daten zurückgreift.

Client

Für den Client gibt es zwei Möglichkeiten, mit einem Cluster in Verbindung zu treten (wobei der Weg beides Mal über den HA-JNDI führt). Es kann entweder eine Liste von URLs zu den Cluster-Knoten übergeben werden, oder der Client sucht per IP-Multicast automatisch nach den Cluster-Knoten.

Enthält das Property **java.naming.provider.url** eine kommaseparierte Liste von URLs zu HA-JNDIs, so versucht die InitialContextFactory mit einem Server in dieser Liste in Kontakt zu treten:

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

Ist dieses Property nicht gesetzt oder kann keiner dieser Server erreicht werden, so versucht die InitialContextFactory per IP-Multicast an diese Daten zu gelangen. Es wird dabei die Adresse 230.0.0.4 mit dem Port 1102 verwendet (siehe „cluster-service.xml“). Wurden diese Parameter in „cluster-service.xml“ geändert, so müssen die entsprechenden Werte über die Properties **jnp.discoveryGroup** und **jnp.discoveryPort** der InitialContextFactory mitgeteilt werden.

Web-Applikationen

Das bisher über die Smart Stubs Gesagte lässt sich leider nicht auf Web-Applikationen übertragen. Hier kommt als Client ein Web-Browser zum Einsatz, der natürlich weit von der Eleganz der Smart Stubs entfernt ist. Deshalb müssen bei Web-Applikationen zwei unterschiedliche Aufgaben bewältigt werden:

- Load-Balancing der eingehenden HTTP-Requests und
- Replikation der HTTP-Session.

Je nach Anwendungsfall ist zu entscheiden, ob beide oder vielleicht nur einer dieser Punkte umgesetzt werden soll.

Üblicherweise wird bei Web-Anwendungen der in JBoss integrierte Tomcat 4 verwendet. Somit laufen der Servlet-Container und der EJB-Container in einer VM ab, was einige Vorteile mit sich bringt.

Load-Balancing

Das Load-Balancing wird nicht vom JBoss übernommen. Stattdessen ist eine Hardware- oder Softwarelösung erforderlich, die vor die Kombination Tomcat/JBoss geschaltet wird und die eingehenden Requests auf den Cluster verteilt. Die einfachste Lösung dürfte sein, einfach jeden Cluster-Knoten mit seiner IP-Adresse unter einem einzigen, gemeinsamen Host-Namen in den DNS einzutragen. Der DNS liefert dann bei jeder Anfrage die jeweils nächste IP-Adresse an den Client. Die Requests werden also

per Round-Robin an die Tomcat-Instanzen im Cluster verteilt. Der Nachteil dieser Lösung ist, dass auch bei einem Ausfall eines Knotens der DNS trotzdem dessen IP-Adresse an die Clients liefert. Außerdem weiß der DNS nichts von einer HTTP-Session. Da jeder Request eines Clients von einem anderen Knoten bearbeitet wird, muss entweder darauf verzichtet werden, Informationen in der Session abzulegen, oder die Session muss entsprechend über alle Cluster-Knoten repliziert werden.

Am entgegengesetzten Ende der Preisskala befinden sich die Hardware-Load-Balancer. Hier gibt es Geräte, die speziell für HTTP-Traffic ausgelegt sind und somit dafür sorgen können, dass ein Client immer mit dem Knoten verbunden wird, der die Session-Informationen dieses Clients beherbergt. Zusätzlich können solche Load-Balancer prüfen, ob alle Knoten erreichbar sind. Fällt ein Knoten aus oder kommt ein Knoten hinzu, so wird dies bei der Verteilung der Requests berücksichtigt. Hardware-Load-Balancer sollten mindestens paarweise eingesetzt werden, denn auch diese Geräte können kaputt gehen.

Eine ähnliche Funktionalität wie bei einem Hardware-Load-Balancer kann auch mit Komponenten aus dem Opensource-Umfeld erreicht werden. Der Apache-Webserver kann in Verbindung mit dem Modul **mod_jk** bzw. **mod_jk2** das Load-Balancing abwickeln. **mod_jk** ist ein Modul, das die Verbindung zwischen dem Apache Webserver und einem Servlet-Container herstellt. Dabei kann es die Requests auf viele Servlet-Container verteilen und dafür sorgen, dass ein konkreter Client immer vom selben Servlet-Container bedient wird („Sticky Sessions“). Die im Folgenden vorgestellte Konfiguration bezieht sich auf den **mod_jk**-Nachfolger **mod_jk2** in Verbindung mit Apache 2.0.x. Konfigurationsbeispiele für das ältere **mod_jk**-Format sind auf den Webseiten des gleichnamigen Jakarta-Projekts zu finden oder können vom Autor dieses Artikels bezogen werden.

mod_jk2 kann als fertiges Binary (auch für Windows) von der URL **http://www.apache.org/dist/jakarta/tomcat-connectors/jk2/binaries/** heruntergeladen werden. Die Installation gestaltet sich recht einfach: Die Datei „mod_jk2.so“ wird in das Apache-Verzeichnis „modules“ kopiert. Die Apache-Konfiguration „httpd.conf“ (oder „apache2.conf“) wird um folgende Zeile erweitert:

```
LoadModule jk2_module modules/mod_jk2.so
```

Im Apache-Verzeichnis „conf“ wird eine Datei mit dem Namen „workers2.properties“ erstellt. Listing 3 zeigt, wie diese Datei aufgebaut ist.

Zu jedem Cluster-Knoten (bzw. zu jeder Tomcat-Instanz im Cluster) wird ein *Channel* konfiguriert. Dieser enthält den Host-Namen des Cluster-Knotens und den Port, auf dem der Tomcat einen AJP13-Listener bereitstellt. Standardmäßig ist das der Port 8009 (siehe Datei „jboss-service.xml“ im JBoss-Verzeichnis „server/all/deploy/jboss-web-tomcat41.sar/META-INF“). Im Beispiel gibt es zwei Cluster-Knoten mit den Namen **server1** und **server2**. Zu jedem Channel wird eingetragen, dass er zu einer Gruppe namens **mycluster** gehört.

Dann wird pro Channel ein AJP13-Worker definiert, was im Beispiel jeweils als Einzeiler geschehen ist. Hier werden die gleichen Host-Namen und Ports wie bei den Channels verwendet. Zusätzlich wird ein Worker definiert, der für das Load-Balancing zuständig ist (lb-Worker). Bei diesem lb-Worker wird konfiguriert, dass er alle Channels der Gruppe **mycluster** benutzen soll. Standardmäßig sorgt der lb-Worker dafür, dass ein Client, der mit einer HTTP-Session assoziiert ist, immer vom gleichen Knoten bedient wird. Auch das ist nur ein Einzeiler.

Am Schluss folgt noch das Mapping von URIs auf Worker. Hier wird definiert, welche Requests von Apache/mod_jk2 an Tomcat weitergeleitet werden sollen.

Im Vergleich zu der Lösung mit einem Hardware-Load-Balancer kann das Gespann Apache/mod_jk leider nicht redundant betrieben werden (bzw. nicht ohne manuelle Eingriffe oder entsprechende zusätzliche Software). Fällt der Apache/mod_jk-Knoten aus, so gibt es kein automatisches Fallback auf einen redundanten Knoten. Dafür handelt es sich um quelloffene Systeme.

```
# Log level
[logger.apache2]
level=INFO

#---------------------------------------

# Ein Channel beschreibt das Transportprotokoll
# zu jedem einzelnen Knoten. Hier kommen
# Sockets zum Einsatz.
[channel.socket:server1:8009]
group=mycluster

[channel.socket:server2:8009]
group=mycluster

#---------------------------------------

# Jeder der obigen Channel braucht einen Worker.
[ajp13:server1:8009]
[ajp13:server2:8009]
# Nun noch ein Load-Balancing Worker, der die
# Requests unter den Channels der Gruppe
# "mycluster" verteilt.
[lb:mycluster]

#---------------------------------------

# Uri -> Worker
[uri:/webapp/*.jsp]
group=lb:mycluster

[uri:/web-console/*]
group=lb:mycluster
```

Listing 3: „workers2.properties“ für mod_jk2

Replikation der HTTP-Session

Das Replizieren der HTTP-Session wird beim aktuellen JBoss 3.2.3 von eben diesem übernommen. Beim kommenden JBoss 3.2.4 soll Tomcat 5 integriert werden, welcher diese Funktionalität bereits mitbringt. Es bleibt also abzuwarten, wie die Session-Replikation bei JBoss 3.2.4 realisiert sein wird.

Bei der Session-Replikation muss abgewogen werden, ob der damit verbundene Overhead (Netzwerkkommunikation zwischen den einzelnen Cluster-Knoten) in Relation zu der gewünschten Ausfallsicherheit steht. Ein Load-Balancer mit „Sticky Sessions“ (wie zuvor beschrieben) skaliert sehr gut und ist – solange kein Knoten ausfällt – nicht auf eine Session-Replikation angewiesen. Erst beim Ausfall eines Knotens geht die HTTP-Session verloren, was durch eine Session-Replikation verhindert werden kann. In vielen Fällen ist diese teure Replikation aber gar nicht erforderlich, da der eigentliche Zustand in der EJB-Logik oder sogar in der Datenbank gehalten wird. Die Session-Replikation skaliert in der aktuellen JBoss-Implementierung nicht so gut, denn mit steigender Anzahl an Knoten steigt auch der Kommunikationsaufwand.

Standardmäßig ist die Session-Replikation in der JBoss-Konfiguration „all“ bereits aktiviert. In derselben Konfigurationsdatei, in der auch der AJP13-Listener definiert wurde (siehe oben), kann eingestellt werden, ob und wie die Session repliziert werden soll. Ist das Attribut „SnapshotMode“ auf **instant** gesetzt, so wird die Session bei jeder Änderung sofort repliziert. Ist „SnapshotMode“ auf **interval** gesetzt, so kann über das zusätzliche Attribut „SnapshotInterval“ bestimmt werden, nach wie vielen Millisekunden eine geänderte Ses-

sion repliziert werden soll. Mit diesem Intervall-Modus kann die Netzwerkkommunikation unter den Cluster-Knoten reduziert werden.

Damit die Session einer Web-Applikation dann auch wirklich repliziert wird, ist folgender Eintrag in der Datei „web.xml" erforderlich:

```
<web-app>
  <distributable />
  ...
</web-app>
```

Erweiterter EJB-Load-Balancer

Wie bereits erwähnt, stellt JBoss für EJB-Aufrufe drei Strategien zur Verfügung: RoundRobin, FirstAvailable und FirstAvailableIdenticalAllProxies. Bei bestimmten Anwendungsfällen sind diese Strategien jedoch nicht ausreichend.

Bei Rich-Client Applikationen hätte man gerne, dass alle (!) EJB-Aufrufe von einem einzigen Cluster-Knoten bedient werden. Dadurch kann serverseitig sehr viel besser nachvollzogen werden, was der Client macht (bzw. warum etwas nicht korrekt funktioniert). FirstAvailableIdenticalAllProxies sorgt leider nur pro EJB dafür, dass der Client mit nur einem Knoten kommuniziert. Verschiedene EJBs (und auch das Home- und das Remote-Interface einer EJB) werden von unterschiedlichen Knoten bedient.

Sind serverseitige Prozesse im Einsatz (wie z. B. Batch-Prozesse), die nicht im EJB-Container sondern als separater Java-Prozess laufen, so hat man zusätzlich noch den Wunsch, dass diese Prozesse – sofern möglich – auf den lokalen Cluster-Knoten zugreifen und nicht übers Netzwerk arbeiten. Es bringt hier leider nichts, den Prozess auf den lokalen JNDI anstatt auf den HA-JNDI zu delegieren: Auch vom normalen JNDI wird ein Smart Stub mit Load-Balancing geliefert.

Die Implementierung eines Load-Balancer, der diese beiden Anforderungen erfüllt, umfasst 140 Zeilen an gut kommentiertem Quellcode. Dadurch ist die Implementierung für einen Abdruck etwas zu lang. Der Quellcode kann jedoch auf Anfrage beim Autor dieses Artikels bezogen werden.

Fazit

Der JBoss Application Server bringt alle Features mit, die für ein vernünftiges Clustering erforderlich sind. Das aufwändige Replizieren von SFSBs auf alle Cluster-Knoten soll schon bald optimiert werden (Stichwort: *Sub-Partitions*). Die Cluster-Funktionalität hat sich im Produktivbetrieb als sehr zuverlässig und einfach in der Handhabung erwiesen.

Links

[Apache] http://httpd.apache.org
[JBoss] http://www.jboss.org
[Tomcat] http://jakarta.apache.org/tomcat/tomcat-4.1-doc/jk2/

Dipl-Ing. (FH) Martin Renner ist als Projektleiter und Softwarearchitekt bei der eXXcellent solutions GmbH in Ulm tätig. Er beschäftigt sich seit vielen Jahren mit der Konzeption und Umsetzung komplexer Enterprise-Anwendungen, bei denen er maßgeblich an deren Architektur beteiligt ist. Sein Fokus liegt dabei auf J2EE-Anwendungen und der sinnvollen und praxistauglichen Integration neuer Technologien. E-Mail: m.renner@exxcellent.de.