JBoss Clustering: J2EE-Cluster-Implementierung mit JGroups

Hinter den Kulissen

VON CHRISTOF KALESCHKE UND DIRK MASCHER

Ausfallsicherheit und Skalierbarkeit sind zentrale Themen im Umfeld von Enterprise-Applikationen. Zeitweise nicht verfügbare oder langsame Anwendungen sind Risiken für Unternehmen. Eine wirkungsvolle Lösung dieser Probleme verspricht Clustering. Der J2EE Application Server JBoss setzt als Basis seiner Cluster-Funktionalität auf JGroups, ein Open-Source-Framework für Gruppenkommunikation. Dieser Artikel stellt JGroups und JBoss Clustering vor und zeigt, wie man mit einfachen JGroups-Mitteln das Innerste eines JBoss-Clusters beobachten kann.

JGroups (früher JavaGroups) ist das vielleicht unbekannteste Open Source Project von JBoss [1]. Auch der Name lässt nicht sofort darauf schließen, welche Art von Software in diesem Projekt entsteht. Als erste Assoziation könnte man z.B. auf eine Community-Software für Java-Entwickler mit Chats und Diskussionsforen schließen. Aber weit gefehlt, JGroups ist zwar ein Framework für Gruppenkommunikation, aber nicht für Menschen, sondern für Prozesse in verteilten Systemen. Und das ist auch der Grund für die Unbekanntheit, JGroups zieht ja nur im Hintergrund die Fäden. Aber gerade auf dem Clustering-Gebiet hat sich JGroups durchgesetzt, so nutzen neben JBoss auch der Application Server JOnAS [3], die Servlet Engine Tomcat 4 [4] und die „Database Cluster Middleware“ C-JDBC [5] JGroups als Grundlage für ihre Cluster.

Gruppenkommunikation

Gruppen, also Mengen von verteilten Prozessen, die miteinander kommunizieren und Nachrichten austauschen, sind in vielen Szenarien anzutreffen. Neben Clustering sind z.B. (Video-)Konferenzsysteme aktuell in der Diskussion, da dort die Gruppenkommunikation besonders leistungsfähig sein muss. Gruppenkommunikation ist auch ein viel bearbeitetes Thema in den Computerwissenschaften. So entstanden an Universitäten umfangreiche Formalismen und Implementierungen. Mit die ersten und auch bekanntesten Systeme kamen von der Cornell University und hießen Horus [9] und Isis [10]. JGroups ist nun die Java-Open-Source-Antwort auf diese Systeme und nutzt sie auch als Vorbild. Nicht umsonst verbrachte Bela Ban, der Chefentwickler von JGroups, einige Zeit an eben dieser Cornell University.

Framework für Gruppenkommunikation

Grundidee eines Frameworks für Gruppenkommunikation ist es, Applikationen eben diese Kommunikation zu ermöglichen, ohne dass der Entwickler der Applikation hierfür tief greifende Implementierungsdetails kennen muss. Ähnlich wie ein Application Server, der zum Beispiel Transaktionsverwaltung, Pools von Datenbankverbindungen usw. zur Verfügung stellt, bietet das Framework bestimmte Services an. Wie unten stehender Kasten zeigt, sind die Anforderungen an diese Services, typischerweise auf den Gebieten Nachrichtenzustellung, -reihenfolge und Mitgliederverwaltung, sehr unterschiedlich [8].

JGroups

Ein Framework für Gruppenkommunikation muss also sehr flexibel und konfigu-

Aufgaben eines Frameworks für Gruppenkommunikation

Nachrichtenzustellung: Das Framework stellt Operationen zum Senden und Empfangen von Nachrichten zur Verfügung und realisiert diese auf Basis der vorhandenen Infrastruktur. Empfänger einer Nachricht kann entweder ein Mitglied oder die gesamte Gruppe sein (Eins-zu-viele-Kommunikation). Wenn gefordert, sichert das Framework einer Anwendung zu, dass entweder alle Mitglieder der Gruppe die Nachricht erhalten oder keine, man spricht dann von einer atomaren Zustellung oder der Alles-oder-nichts-Eigenschaft.

Nachrichtenreihenfolge: Das Framework garantiert Applikationen je nach deren Anforderungen eine bestimmte Reihenfolge der Nachrichtenzustellung, das geht von der totalen Ordnung aller Nachrichten aller Sender bis zu keinerlei Einschränkungen.

Mitgliederverwaltung: Eine Gruppe ist ein dynamisches Gebilde, in das neue Mitglieder ein- und aus dem alte Mitglieder austreten können. Das Framework stellt einem neuen Mitglied ab dessen Eintritt alle Nachrichten zu, während ein austretendes Mitglied keine Nachrichten mehr erhält. Es ist ebenfalls Aufgabe des Frameworks, Ausfälle von Mitgliedprozessen festzustellen und an alle verbleibenden Mitglieder zu propagieren.

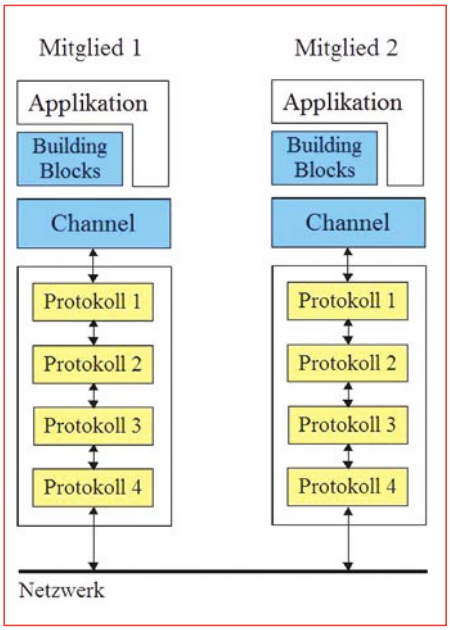

Abb. 1: Aufbau von JGroups

rierbar sein, um für verschiedene Anwendungen verschiedene Rahmenbedingungen zu schaffen. JGroups trägt dem durch seine Architektur Rechnung und unterscheidet, wie Abbildung 1 zeigt, die drei Komponenten Channel, Protokoll-Stack und Building Blocks [6].

Channel

Die zentrale Schnittstelle, über die eine Anwendung die Services von JGroups nutzt, ist das *Channel*-Interface mit der *JChannel*-Implementierungsklasse. Weitere Implementierungsdetails von JGroups bleiben für die Anwendung transparent. Ein Prozess erzeugt (wie später im Beispielcode ersichtlich) ein *JChannel*-Objekt und ruft dann die *connect()*-Methode mit einem Gruppennamen als Parameter auf. Dadurch wird er Mitglied der Gruppe aller Prozesse, die den gleichen Namen angegeben haben. Danach versendet er Nachrichten mit *send()* und empfängt Nachrichten mit *receive()*. Die Methode *getView()* liefert dem Prozess jederzeit eine Liste der aktuellen Gruppenmitglieder. *getState()* fordert die anderen Mitglieder auf, Zustandsinformation zu senden. Das kann beispielsweise verwendet werden, um ein neues Gruppenmitglied auf den gleichen Stand wie die anderen Mitglieder zu bringen. So ist die erste Aktion eines neuen JBoss-Cluster-Mitglieds der Aufruf von *getState()*. Durch *disconnect()* wird die Gruppe verlassen.

Protokoll-Stack

Der Protokoll-Stack ist eine Anordnung von Protokollschichten, die mit dem Channel verbunden ist. Ganz ähnlich wie beim TCP/IP-Stack werden alle gesendeten und empfangenen Nachrichten durch diesen Stack gereicht. Eine gesendete Nachricht durchläuft die Schichten von oben nach unten ausgehend vom Channel, bis dann schließlich die unterste Schicht die Nachricht über das Netzwerk verschickt. Eine empfangene Nachricht wird von der untersten Schicht entgegengenommen und durch alle Schichten nach oben zum Channel gereicht, der sie dann in einer Queue dem verbundenen Prozess zur Verfügung stellt. Jede Protokollschicht kann dann die Nachrichten verändern, zurückhalten, verwerfen oder ganz einfach nur durchreichen.

Der Aufbau des Protokoll-Stacks wird dem *JChannel*-Konstruktor ab der Version 2.2.5 in Form eines XML-Dokuments übergeben. In früheren Versionen musste man sich mit einem relativ kryptischen String herumschlagen. Mit dem Aufbau werden die konkreten Eigenschaften der Gruppenkommunikation bzgl. Nachrichtenzustellung, -reihenfolge und Mitgliederverwaltung festgelegt (siehe obigen Kasten) und das Framework erfüllt die Anforderungen der Anwendung.

Lädt man die aktuelle JGroups-Version 2.2.7 von [2] herunter, so sind in der *jgroups-all.jar* XML-Dateien mit beispielhaften Protokoll-Stacks enthalten. Dort kann man sich einen Überblick über die Definition von Protokollen verschaffen (z.B. in der *default.xml*). Es gibt eine Vielzahl von Protokollen, die wichtigsten für das JBoss Clustering werden im Folgenden vorgestellt.

- UDP versendet Nachrichten an alle Mitglieder mittels UDP Multicast. Alternativ kann das TCP-Protokoll verwendet werden. In diesem Fall wird jedoch zu jedem Gruppenmitglied eine eigene TCP-Verbindung geöffnet. Der hinten angegebene Link [11] zeigt, dass TCP unter bestimmten Umständen aber trotzdem performanter als UDP sein kann.
- FD, VERIFY_SUSPECT und GMS implementieren die Mitgliederverwaltung. Diese Protokolle stellen mit SUSPICION-Nachrichten fest, ob Mitglieder ausgefallen sind und schließen diese aus. Das GMS-Protokoll versendet in diesem Fall, aber auch wenn neue Mitglieder beitreten oder alte Mitglieder ordnungsgemäß ausscheiden, eine VIEW-Nachricht, in der die aktualisierte Liste der Mitglieder propagiert wird.
- NAKACK und STABLE sorgen für die garantierte Zustellung von Nachrichten an alle Mitglieder und definieren die Reihenfolge der Nachrichtenzustellung. NAKACK hält sich dabei an die Single-Source-FIFO-Regel, wonach Nachrichten eines Senders sich bei der Zustellung nicht überholen dürfen. STABLE kümmert sich vor allem um die Garbage Collection von Nachrichten, die allen Mitgliedern zugestellt wurden.

Über das STATE_TRANSFER-Protokoll fordern Mitglieder von anderen Mitgliedern Zustandsinformationen an (über die *getState()*-Methode des Channels). Dazu werden GET_STATE- und SET_STATE-Nachrichten ausgetauscht.

Building Blocks

Aufbauend auf dem Channel gibt es die sog. Building Blocks, die mit der eigentlichen Gruppenkommunikation nichts zu tun haben, sondern höher stehende, für verteilte Prozesse relevante Dienste implementieren und Anwendungen zur Verfügung stellen. Sie versenden dazu über den Channel Nachrichten. Wie bei den Protokollen gibt es innerhalb von JGroups eine Vielzahl von Building Blocks, dieser Artikel beschränkt sich allerdings auf die Beschreibung des *PullPushAdapter* und des *RPCDispatcher*. Diese beiden spielen eine zentrale Rolle bei der Implementierung des JBoss Clustering und somit auch im Beispielcode für diesen Artikel.

Der *PullPushAdapter* erleichtert den Empfang verschiedener Typen von Nachrichten. Ein Prozess kann dazu beim *PullPushAdapter* Klassen registrieren, die die *MessageListener*- und *MembershipListener*-Interfaces implementieren. Der *PullPushAdapter* startet einen eigenen Thread und ruft die *receive()*-Methode des Chan-

nels auf. Geht eine Nachricht ein, wird je nach Typ der Nachricht eine spezifische Methode des registrierten *MessageListener* oder *MembershipListener* aufgerufen:

- *receive()* bei normalen Nachrichten
- *getState()* bei GET_STATE-Nachrichten
- *setState()* bei SET_STATE-Nachrichten
- *viewAccepted()* bei VIEW-Nachrichten
- *suspect()* bei SUSPICION-Nachrichten

Über den *RPCDispatcher* kann ein Gruppenmitglied wie bei einem „echten" RPC-Mechanismus eine entfernte Methode aufrufen, jedoch mit dem Unterschied, dass die Methode nicht nur auf einem Server, sondern auf allen Gruppenmitgliedern aufgerufen wird. Implementiert z.B. jedes Mitglied eine Methode *printConfiguration()*, so kann ein Mitglied über den *RPCDispatcher* den Aufruf von *printConfiguration()* bei allen Mitgliedern initiieren. Da der *RPCDispatcher* den *PullPushAdapter* erweitert, kann man bei ihm ebenfalls *MessageListener* und *MembershipListener* registrieren.

Mit diesen drei JGroups-Komponenten ist der Grundstein für die Gruppenkommunikation des JBoss Clustering gelegt. Bevor die konkrete JBoss-Clustering-Implementierung beschrieben wird, werden im Folgenden kurz allgemein die wichtigsten Konzepte und Begriffe von Cluster-Lösungen dargestellt.

Clustering

Wie in der Einleitung bereits beschrieben, sind Ausfallsicherheit und Skalierbarkeit von Enterprise-Applikationen für Unternehmen von entscheidender Bedeutung. Alle wichtigen J2EE Application Server bieten deshalb Clustering-Technologien an, um die auf ihnen laufenden Anwendungen ausfallsicher und skalierbar zu machen. Ein Cluster ist dabei eine Menge von Knoten (hier eine Menge von Application-Server-Instanzen), die an einer gemeinsamen Aufgabe arbeiten und sich nach außen als ein einziges System darstellen.

Da einzelne Application-Server-Instanzen aufgrund von Hardware-, Software- oder sonstigen Problemen immer ausfallen können, setzt Ausfallsicherheit Fehlertoleranz innerhalb des Clusters voraus. Der Ausfall eines Knotens wird vom Cluster erkannt und die Aufgaben des ausgefallenen Knotens werden von einem anderen übernommen. Das System als Ganzes arbeitet trotz des Ausfalls völlig korrekt weiter und der Vorgang bleibt für die Nutzer des Clusters transparent. Man spricht deshalb hier auch von Transparent Failover.

Der Ansatzpunkt für die Skalierbarkeit ist das Load Balancing innerhalb des Clusters. Anfragen an das Cluster werden für den Nutzer unbemerkt auf die Knoten des Clusters verteilt, sodass einzelne Knoten nicht zu sehr belastet werden und die Antwortzeiten insgesamt niedrig bleiben. Bei steigender Anzahl der Anfragen ist es – abhängig von der konkreten Implementierung – auch noch möglich, zusätzliche Knoten in das Cluster zu hängen. Dies ist allerdings nur bis zu einem gewissen Punkt sinnvoll, da durch eine zu große Anzahl von Knoten der Cluster-interne Kommunikations-Overhead überproportional zunehmen und die Gesamtperformance wieder sinken würde.

JBoss Clustering

Die konkrete Ausprägung eines Clusters wird bei JBoss seine Partition genannt. Hinter einer JBoss-Partition verbirgt sich eine JGroups-Gruppe, deren Mitglieder die Knoten der Partition sind. JGroups stellt der Partition die Infrastruktur für die Gruppenkommunikation zur Verfügung und ist dabei jederzeit in der Lage, neue Knoten in die Partition aufzunehmen oder alte zu entfernen. Ein Administrator kann eine JBoss-Server-Instanz also z.B. für Wartungsarbeiten jederzeit stoppen und aus der Partition nehmen, ohne den Betrieb insgesamt zu beeinflussen.

Ausfallsicherheit und Skalierbarkeit einer J2EE-Anwendung in einer JBoss-Partition werden durch Replikation der J2EE-Anwendungsobjekte erreicht (wie später noch erläutert werden wird, sind das Objekte des JNDI-Namensbaums, Stateful Session Beans und HTTP-Session-Objekte). Alle Anwendungsobjekte werden dabei über die JGroups-Gruppe (genauer über den *RPCDispatcher* Building Block) an alle Knoten der Partition verteilt. Aus Benutzersicht spielt es somit keine Rolle, welcher konkrete Knoten eine Anfrage an die Partition tatsächlich bearbeitet. Die Voraussetzung für Transparent Failover und Load Balancing ist also geschaffen.

Replikation findet immer zwischen allen Knoten der Partition statt. JBoss arbeitet derzeit an der Einführung von Subpartitionen, die die Replikation auf bestimmte Knoten der Partition einschränken. Die aktuelle JBoss-Version 4.0.1 enthält diesen Mechanismus allerdings noch nicht. In der Konsequenz bedeutet das, dass JBoss Cluster (noch) nicht beliebig skalieren. Durch die Hinzunahme von zu vielen Knoten beansprucht der nicht unerhebliche Overhead der JGroups-Kommunikation

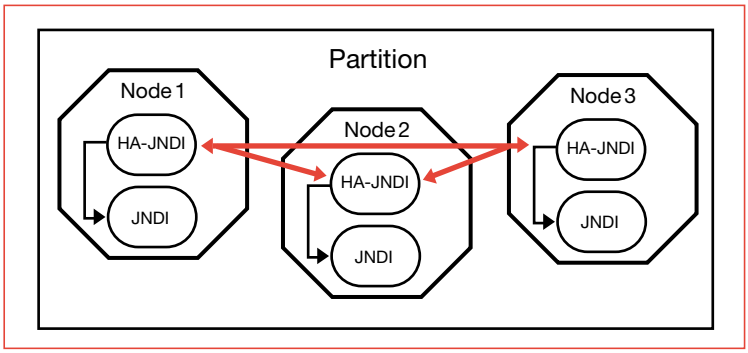

Abb. 2: Anfragen an HA-JNDI

die Partition so stark, dass die Gesamtperformance wieder sinkt. Die folgenden Abschnitte beschreiben, wie die einzelnen J2EE-Technologien unter JBoss Cluster-fähig werden, im Speziellen JNDI, EJBs, JMS und Webapplikationen.

Cluster-weites JNDI

Da praktisch jedem Zugriff auf J2EE-Funktionalität ein JNDI Lookup vorausgeht, sollte der JNDI-Naming-Service zuallererst ausfallsicher und skalierbar sein. Auf jedem Knoten einer JBoss-Partition wird deshalb der HA-JNDI-Naming-Service deployt (HA steht dabei für High Availability). Dieser Service baut durch Replikation aller Objekte auf allen Knoten (über den JGroups *RPCDispatcher*) einen globalen, Cluster-weiten JNDI-Objektbaum auf. Clients können wie gewohnt Lookups machen oder Objekte in die Struktur einhängen, ohne wissen zu müssen, mit welchem Knoten sie verbunden sind. Aus Performanzgründen und um globale und lokale Objekte klar unterscheiden zu können besitzen die einzelnen JBoss-Instanzen aber weiterhin jeweils ihren lokalen JNDI-Baum, der für andere Knoten nicht sichtbar ist.

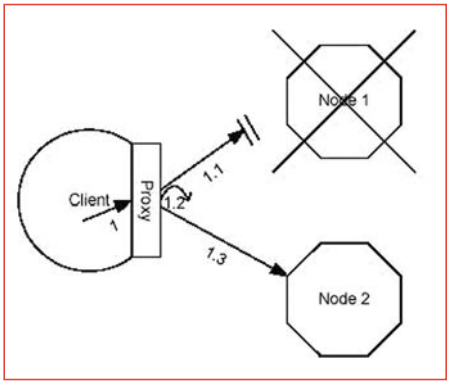

Abb. 3: Funktionsweise von HA-RMI

Der globale HA-JNDI-Baum und die lokalen JNDI-Bäume sind aber eng miteinander verbunden. Stellt der HA-JNDI-Service bei einem Lookup fest, dass das gesuchte Objekt nicht im globalen Baum vorhanden ist, so werden transparent für den Aufrufer nacheinander alle lokalen Bäume (beginnend mit dem eigenen) befragt, ob sie das Objekt enthalten. Abbildung 2 zeigt diesen Mechanismus.

Cluster-fähige EJBs

Die Entwicklung einer Cluster-fähigen EJB, sei es (Stateful oder Stateless) Session oder Entity Bean, ist denkbar einfach: Ein Eintrag *<clustered>true</clustered>* im JBoss-spezifischen EJB Deployment Descriptor (*jboss.xml*) genügt. Failover und Load Balancing finden zur Laufzeit dann clientseitig statt, d.h. auf der Seite des EJB-Aufrufers. JBoss liefert dazu mit den clientseitigen EJB-Objekten (bei JBoss werden diese Objekte Smart Proxies genannt) eine HA-RMI-Komponente mit, die eine Liste der verfügbaren Knoten in der Partition hält. Diese Liste wird bei jedem EJB-Aufruf aktualisiert und aus ihr wird der Knoten für den nächsten EJB-Aufruf bestimmt. Tritt ein Fehler beim EJB-Aufruf auf, so leitet HA-RMI den Aufruf automatisch an einen anderen Knoten weiter. Dieser Vorgang spielt sich nur im Smart Proxy ab und ist für die Client-Anwendung transparent, wie Abbildung 3 verdeutlicht.

Um Stateless Session Beans im Cluster zu betreiben, ist HA-RMI bereits ausreichend. Stateless Session Beans besitzen keinen Zustand und es spielt deshalb keine Rolle, auf welchem Knoten sie aufgerufen werden. Stateful Session Beans dagegen besitzen sehr wohl einen Zustand. Da dieser über JGroups nach jedem Methodenaufruf auf alle Knoten repliziert wird, funktioniert der HA-RMI-Mechanismus aber auch für Stateful Session Beans. Es ist zu beachten, dass diese Replikation eine relativ teure Operation ist und die Entscheidung für Stateful Session Beans gut durchdacht sein sollte (auch aus anderen Gründen, auf die hier nicht näher eingegangen werden kann).

Für Entity Beans gibt es keine Replikation und auch keinen verteilten Cache oder Sperrmechanismen. Eine Synchronisation der Entity Beans auf den verschiedenen Knoten kann deshalb nur über die Datenbank erfolgen. Eine Beschreibung dieses Mechanismus geht über den Rahmen dieses Artikels hinaus, kann aber in [7] nachgelesen werden.

Cluster-weites JMS

Ab JBoss 3.2.4 gibt es auch eine Cluster-fähige JMS-Implementierung (HA-JMS), die in der JBoss-„all"-Konfiguration aktiviert ist. Bei dieser Lösung läuft der JMS Provider nur auf einem Knoten der Partition. Bei dessen Ausfall wird der JMS Provider automatisch auf einem anderen Knoten deployt. HA-JMS bietet also Failover, jedoch kein Load Balancing.

HTTP-Session-Replikation

Ebenso wie Stateful Session Beans enthalten die HTTP-Session-Objekte Zustandsinformation und sie werden deshalb auf allen Knoten repliziert. Die in der aktuellen JBoss-Version 4.0.1 integrierte Servlet Engine ist Tomcat 5. Für eine detaillierte

Tipps zum JBoss-Cluster-Test

Ein JBoss Cluster zu starten ist vergleichsweise einfach. Man startet einfach die „all"-Konfiguration. JBoss erkennt automatisch die anderen Knoten der Partition. Nur unter Windows kann es aufgrund des „MediaSense"-Features Probleme mit UDP Multicast geben. Das Problem tritt jedoch nur auf, wenn der Rechner nicht an einem Netzwerk angeschlossen ist, z.B. bei lokalen Tests auf einem isolierten Entwickler-Notebook. In diesem Fall sollte man in der Datei *deploy/cluster-service.xml* in der Definition des JGroups-Protokoll-Stacks im UDP-Abschnitt die Eigenschaft *loopback="true"* setzen. Darüber hinaus ist es notwendig, über den Hardwareassistenten der Systemsteuerung den Microsoft Loopback-Adapter zu aktivieren.

Beschreibung der Themen Load Balancing und Failover für Tomcat sei an dieser Stelle auf einen früheren Artikel im *Java Magazin* verwiesen [12]. Es wird jedoch explizit darauf hingewiesen, dass die Replikation der HTTP-Session-Objekte der in JBoss integrierten Tomcat-Version auch weiterhin auf JGroups basiert. Die Standalone-Version von Tomcat setzt hier ja ab Version 5 auf eine eigene Implementierung.

JBoss-Cluster-Konfiguration

Die gesamte Cluster-Funktionalität ist in der JBoss-„all"-Konfiguration bereits komplett eingestellt. Relevant sind die Dateien *deploy/cluster-service.xml* und *deploy/jbossha-httpsession.sar/META-INF/jboss-service.xml* (für die HTTP-Session-Replikation). Eine Änderung der Konfigurationsparameter sollte wenn überhaupt nur selten (siehe nebenstehenden Kasten) notwendig sein. In der *cluster-service.xml* ist für diesen Artikel besonders der erste Service interessant, da hier der Name und der Protokoll-Stack der JGroups-Gruppe definiert werden. Die verwendeten Protokolle (UDP, FD, VERIFY_SUSPECT usw.) sind nun bereits vertraut.

Ein weiterer interessanter Service wird über die Datei *deploy/deploy.last/farm-service.xml* aktiviert. Der als Farming bezeichnete Service sorgt dafür, dass auf allen Knoten der Partiton die gleichen Applikationen laufen. Wird eine Applikation in das *farm*-Unterverzeichnis kopiert, initiiert der Farming-Service, dass die Applikation in die *farm*-Unterverzeichnisse aller Knoten kopiert wird und alle Knoten diese Applikation deployen. Analog verhält es sich beim Löschen einer Applikation aus dem *farm*-Unterverzeichnis eines Knotens. Die Applikation wird dann auf allen Knoten gelöscht und undeployt.

JBoss-Cluster-Monitor

Nach der ganzen Theorie ist es nun Zeit, ein Cluster in Aktion zu erleben. Auf der beiliegenden CD sind zwei JBoss-Konfigurationen vorhanden, mit denen man auf einem Rechner zwei JBoss-Instanzen starten kann, die sich zu einer Partition zusammenschließen. Die zwei Konfigurationen *cluster1* und *cluster2* werden einfach in das *server*-Verzeichnis kopiert und dann mit

```
run –c cluster1
run –c cluster2
```

gestartet. Eine kleine beiliegende Enterprise Application (*Simple.ear*) enthält ein minimales Servlet, das einen String in die Session schreibt und eine minimale Stateful Session Bean, die die beiden Business-Methoden *getString()* und *setString()* implementiert. Deployt wird die Enterprise Application, indem sie in das *farm*-Unterverzeichnis in einer der beiden Konfigurationen kopiert wird. Man kann jetzt schon beobachten, wie dieser Vorgang zum Deployment auf beiden Knoten führt.

Die beiliegende Klasse *EjbCaller* erzeugt immer wieder neue Instanzen der minimalen Stateful Session Bean (die dann

von JBoss repliziert werden) und ruft die *setString()*- und *getString()*-Methoden auf. *EjbCaller* wird ganz normal weiterlaufen, wenn einer der Knoten gestoppt wird – Transparent Failover in Aktion! Diese Replikation von HTTP-Session-Objekten wird provoziert, indem man mit einem Browser die URL *http://localhost:9080/cluster/simple* auf dem einen Knoten oder die URL *http://localhost:9081/cluster/simple* auf den anderem Knoten aufruft. Ein Transparent Failover kann man hier allerdings nicht demonstrieren.

Die Replikation von HTTP-Session-Objekten und Stateful Session Beans wird mit dem *JBossClusterMonitor* beobachtet, der in Listing 1 zu sehen ist. In der *main()*-Methode wird ein *JChannel* mit der gleichen Konfiguration wie die der JBoss-Knoten erzeugt (die Datei mit der Definition des Protokoll-Stacks ist auf der CD vorhanden) und die *connect()*-Methode mit dem gleichen Gruppennamen aufgerufen, wodurch der *JBossClusterMonitor* ein neues Mitglied der Gruppe wird. *JBossClusterMonitor* täuscht also vor, ein Knoten der Partition zu sein. Die folgenden *setOpt()*-Aufrufe sorgen dafür, dass Nachrichten vom Typ VIEW, SUSPECT, GET_STATE und SET_STATE zugestellt werden. Durch den Aufruf der *channel.getState()*-Methode als letzte Aktion der *main()*-Methode werden die JBoss-Knoten in der Partition veranlasst, ihre aktuelle Zustandsinformation zu versenden. Diese wird dann in der *setState()*-Methode des *MessageListener* entgegengenommen. Das ist aus dem Grund interessant, da als Zustandsinformation alle replizierten Objekte der Partition versendet werden. Ein neuer tatsächlicher JBoss-Knoten wäre mit diesen Informationen in der Lage, am Failover und Load Balancing teilzunehmen.

Änderungen in der Mitgliederliste können in den aufgeführten Methoden *viewAccepted()* und *suspect()* des *MembershipListener* beobachtet werden. Startet/Stoppt man einen Knoten, werden eine VIEW-Nachricht versendet und in *viewAccepted()* die aktuelle Mitgliederliste ausgegeben.

Listing 1

```
class JBossClusterMonitor extends RPCDispatcher{

 public static void main(...) {
   // Erzeugen des Channel
   channel = new JChannel(CONFIG);
   channel.setOpt(Channel.SUSPECT, Boolean.TRUE);
   channel.setOpt(Channel.GET_STATE_EVENTS,
                                        Boolean.TRUE);
   channel.setOpt(Channel.VIEW, Boolean.TRUE);

   // Starte den Channel
   channel.connect(groupName);
   boolean stateReceived = channel.getState(null, 5000);
 }

 /* Implementierung des MembershipListener */
 public void viewAccepted(View newView) {
   // Gib eine Liste der Mitglieder in der aktuellen View aus
   Vector members   = newView.getMembers();
   long currentViewId = newView.getVid().getId();

   log.debug("New view with viewId " + currentViewId +
                                        "; members: ");

   for (int i = 0; i < members.size(); i++) {
     Address address = (Address)members.get(i);
     log.debug("  " + address.toString());
   }
 }

 public void suspect(Address suspected) {
   // Gib den verdächtigten Knoten aus
   log.debug("Node suspected: " + suspected.toString()));
 }

 /* Implementierung des MessageListener */
 public void receive(Message msg) {
   log.debug("Message: " + msg.getBuffer());
 }

 public void setState(byte[] state) {
   // Gib den aktuellen Zustand aus,
   // das sind alle replizierten Objekte
   ...

 }

 /* Überschreiben der handle()-Methode des
    RPCDispatcher */
 public Object handle(Message msg) {
                // Gib das aktuell replizierte Objekt in msg aus
   ...
 }
}
```

Mit einem kleinen Trick wird die Replikation von Objekten über den *RPCDispatcher* sichtbar. Die Klasse *JBossClusterMonitor* erweitert nämlich einfach den *RPCDispatcher* und überschreibt dann die *handle()*-Methode, in der die replizierten Objekte ausgegeben werden können (der Code dafür ist allerdings zu umfangreich, um ihn hier abzudrucken).

Zusammenfassung

JBoss implementiert ein sehr flexibles Clustering, das bereits in vielen Projekten seine Produktionstauglichkeit unter Beweis gestellt hat. Der Artikel hat aufgezeigt, dass die JBoss-Entwickler ihre Cluster-Implementierung nicht von Grund auf entwickelt, sondern auf dem Open-Source-Projekt JGroups aufgesetzt haben. Grundle- gende Problemstellungen der Gruppen- kommunikation mussten somit nicht erneut gelöst werden. Dieser Artikel hat einen Einstieg in die generelle Thematik der Gruppenkommuniktion gegeben und die Funktionsweise von JGroups erläutert. Das Code-Beispiel hat aufgezeigt, wie vergleichsweise einfach es ist, mit diesem Verständnis ein Monitoring-Tool für JBoss Cluster zu schreiben.

Es wurde ebenfalls darauf hingewiesen, dass die Cluster-Implementierung der aktuellen JBoss-Version 4.0.1 nur begrenzt skalierbar ist, da Änderungen an HTTP-Session-Objekten oder den Zuständen von Stateful Session Beans immer auf allen Knoten im Cluster repliziert werden. Diese Restriktion will das JBoss-Team aber mit einer der nächsten Versionen und der Einführung von den erwähnten Subpartitionen aufheben.

Christof Kaleschke ist Diplom-Informatiker, zertifizierter JBoss Consultant und Java/J2EE-Entwickler der ersten Stunde. Als Mitarbeiter von Accelsis Technologies war er maßgeblich an Architektur, Design und Implementierung von mehreren erfolgreich umgesetzten J2EE-Projekten beteiligt.

Dirk Mascher ist Geschäftsführer von Accelsis Technologies und zertifizierter JBoss Consultant. Sein Spezialgebiet ist die Architektur verteilter, mehrschichtiger Systeme, insbesondere im J2EE- Umfeld.

Links & Literatur

[1] JBoss: www.jboss.org

[2] JGroups: www.jgroups.org/javagroupsnew/docs/

[3] JOnAS: jonas.objectweb.org

[4] Tomcat 4 Clustering: www.theserverside.com/articles/article.tss?l=Tomcat; Tomcat 5 Clustering: Peter Roßbach: Magische Momente. Grundlagen eines Tomcat Cluster, in *Java Magazin* 2.2005

[5] C-JDBC: c-jdbc.objectweb.org

[6] Bela Ban: JGroups User Guide

[7] S. Labourey, B. Burke: JBoss Clustering

[8] Andrew Tanenbaum: Moderne Betriebssysteme, Pearson, 2002

[9] Horus: www.cs.cornell.edu/Info/projects/horus

[10] Isis: www.cs.cornell.edu/Info/projects/isis

[11] JGroups Evaluation: jmob.objectweb.org/jgroups/JGroups-middleware-2004.pdf

[12] Achim Hügen: Doppelt hält besser. Clustering auf Basis von Tomcat 5, in *Java Magazin* 7.2004