Apache und Tomcat-Cluster im Einsatz

Mission Possible

VON RAINER JUNG UND PETER ROßBACH

Open-Source-Software in einem Hochverfügbarkeits-Projekt zu haben gilt immer noch als großes Risiko. In der Entwicklung von kleinen und mittleren Websites ist Tomcat schon lange zu Hause, aber gilt das auch für lastintensive und hochverfügbare Webanwendungen? Bisher gibt es nur wenige dokumentierte Erfahrungen und Erfolgsmeldungen sind rar. In dieser Kolumne wollen wir über den realen Einsatz eines Apache/Tomcat-Cluster in einer unternehmenskritischen Anwendung berichten.

Anfang 2004 begann bei einem großen deutschen Unternehmen das Projekt zur Erneuerung der zentralen Online-Anwendungsarchitektur. Die bisherige Anwendung war drei Jahre erfolgreich betrieben worden. In dieser Zeit gab es eine enorme Zunahme der Nutzerzahlen auf mehr als das Achtfache und eine Steigerung der Funktionalität. Getragen von diesem Erfolg ist der Betrieb des Online-Services nun unternehmenskritisch geworden.

Gefordert ist ein 24x7x365-Betrieb der Anwendung mit gleichzeitig hoher Verfügbarkeit und auch gleich bleibenden guten Antwortzeiten. Es müssen also sowohl ungeplante technische Ausfälle weitgehend abgefangen als auch Verfahren zur transparenten Wartung ohne Gesamtausfall erarbeitet werden. Dabei stellt es keine Option dar, administrative Tätigkeiten auf Randzeiten zu verlagern: Auch um ein Uhr in der Nacht arbeiten regelmäßig noch über tausend Kunden mit der Anwendung.

Gleichzeitig muss die Anwendung hochlastfähig sein und die Systemarchitektur weiteres nennenswertes Wachstum vorsehen. Aufgrund der Erfahrungen mit dem Betrieb der vorherigen Anwendung ist man in der glücklichen Lage, präzise Lastprofile für die neue Anwendung zu kennen. Die vorhandenen Informationen sind durch eine Hochrechnung des geplanten Wachstums mit der IT und Fachseite präzisiert worden. Als Lastanforderung ergeben sich

so bis zu 15.000 parallele Sitzungen (Sessions) und ein Durchsatz von 20 Anmeldungen, 200 dynamischen Seiten und 1.000 statischen Seiten bzw. Bildern jeweils pro Sekunde.

Der Auftrag

Die damals produktive Anwendung basierte im Web-Layer schon auf den Open-Source-Komponenten Apache und Tomcat. Aufgrund der sehr positiven Erfahrungen sollte zunächst untersucht werden, inwieweit diese Komponenten auch als Basis für die neue Anwendung geeignet waren. Dabei galt es, jedoch zu berücksichtigen, dass bislang die redundant vorhandenen Apache-Webserver und Tomcat-Webcontainer keine Individualität besaßen: Alle Anwendungssitzungen wurden in einer zentralen Datenbank gehalten und pro Anfrage vom angesprochenen Server gelesen und gegebenenfalls aktualisiert. Aus Sicht des Webcontainers war die Anwendung also zustandslos. Die neue Anwendung ist eine Struts-basierte J2EE-Webanwendung, die natürlich Webcontainer-Sessions verwendet. Damit der Kunde nun den Ausfall eines Servers nicht mitbekommt, muss die Sitzung repliziert werden. Es ist also sicherzustellen, dass bei einem technischen oder administrativen Ausfall die aktive Sitzung auf einem weiteren Server bereitgestellt wird.

Motiviert durch einen bestehenden Apache/Tomcat-Wartungsvertrag machte es sich die kippdata zur Aufgabe, die Einsetzbarkeit der aktuellen Tomcat 5-Cluster-Implementierung für dieses und ähnliche Projekte zu prüfen. Für die genaue Evaluierung standen vier Wochen Zeit zur Verfügung, nach weiteren vier Wochen sollte das Konzept für eine produktionsgeeignete Container-Infrastruktur vorgelegt werden. Wie sich schon bald zeigte, war dieser zeitliche Rahmen nicht zu üppig bemessen.

Warum Apache und Tomcat gewählt werden?

- Ein Internet-Webserver muss eine besondere Stabilität haben. Apache hat die meisten Produktionsstunden weltweit und ist „Marktführer".
- Tomcat hat die beste Integration in Apache und eine hohe Produktreife.
- Exzellentes Know-how für beide Komponenten ist im Markt verfügbar.
- Kontinuierliche Weiterentwicklung durch eine breite Community getrieben, im Notfall Möglichkeit eigener Fehleranalyse und Verbesserung, da open source.
- Vermeidung von Lizenzkosten sollte in komplexen Projekten nicht der ausschlaggebende Faktor sein.

Erste Tests des Tomcat-Cluster

Für die Untersuchung wurden mehrere physische Testsysteme genutzt, um die Komplexität redundanter und geclusterter Tomcat-Knoten angemessen abbilden zu können. Als Plattform wurde die bei diesem Kunden für unternehmenskritische Anwendungen festgelegte Plattform Solaris Sparc gewählt, als JVM die zu diesem Zeitpunkt aktuelle Version 1.4.2. Eine erste Sichtung der Cluster-Unterstützung in Apache und Tomcat ergab die Auswahl von Apache 1.3 und Tomcat 5.0 mit mod_jk 1.2 als Verbindungsmodul. Für das Clustering wurde die mit Tomcat 5 gebündelte Implementierung eingesetzt, konkreter der so genannte DeltaManager [1]. Eine entsprechend ausführliche Beschreibung der Cluster-Grundlagen können Sie in der TomC@-Kolumne im *Java Magazin* 2.2005 finden [2].

Die verfügbare Dokumentation für das Clustern mit Tomcat erwies sich schnell als oberflächlich und teilweise inkorrekt. Sie genügt allenfalls, um schnell einen einfachen Cluster aufzusetzen, gibt jedoch keine präzisen Informationen über die genauen Abläufe im Cluster. Ebenso ist die Beschreibung der meisten Konfigurationsoptionen eher vage. Deshalb wurden alle Tests von Anfang an durch ein Code-Review der Cluster-Quellen begleitet, was sich sehr bald als Schlüssel zum wirklichen Erfolg herausstellte. Darüber hinaus wurden zur Klärung noch weitere Protokollausgaben in den vorhandenen Code eingefügt, um bessere Einsichten in die internen Abläufe der Tomcat-Cluster-Implementierung zu erhalten.

Zunächst wurden einfache Funktionstests an einem Cluster bestehend aus einem Apache-Knoten und zwei Tomcat-Knoten durchgeführt. Das Logging des Tomcat wurde mithilfe von Log4j so konfiguriert, dass das Cluster-Package *org.apache.catalina.cluster* alle Meldungen ausgab [3]. Mittels einer einzelnen Session-JSP wurde das korrekte Replizieren von Sessions und Session-Attributen verifiziert. Durch Sniffen der Verbindungspakete wurde abgeschätzt, welche Netzlast durch die Replikation entsteht [4]. Diese Tests verliefen positiv und unterstützten die Entscheidung, es weiter mit einem Tomcat-Cluster zu versuchen.

Synchron oder asynchron?

Nun galt es, die grundsätzliche Entscheidung für das einzusetzende Replikationsmodell zu fällen: synchrone Replikation mit oder ohne Senderpool oder asynchrone Replikation [1] [2]. Konzeptionell beruht der Einsatz eines Clusters auf der Idee, eine möglichst transparente Administration und Ausfallsicherung herzustellen. Im normalen Ablauf der Anwendung, also während eines stabilen Betriebs, sollte die Cluster-Replikation aber möglichst wenig die Verarbeitung der Anfragen verändern. Insbesondere sollten Probleme im Cluster selbst nicht die Betriebsstabilität einzelner Knoten gefährden. Insofern galt es, die Sitzungsreplikation als sekundäre Funktionalität nach der eigentlichen Webanwendung zu realisieren. Es sollten keinerlei Beeinträchtigungen auf die Antwortzeiten und Nebenläufigkeiten durch den Cluster messbar sein.

Diese Betrachtung schloss den Einsatz der synchronen Replikation aus, da hier eine Störung in der Cluster-Replikation sofort negativ auf die eigentliche primäre Webanwendung durchschlägt: Eine Anfrage bekommt erst dann ihre Antwort, wenn alle Knoten den korrekten Empfang der Replikationsnachricht quittiert haben. Eine Störung eines einzelnen Knotens oder des Replikationsnetzes führt schnell dazu, dass sehr viele Threads auf Time-outs warten. Deshalb wurde im Projekt auf den Einsatz asynchroner Replikation gesetzt. Im asynchronen Modus ist nicht sichergestellt, dass eine Veränderung an einer Sitzung auch sofort auf allen anderen Tomcat-Knoten zur Verfügung steht. Als Lastverteiler kommt das mod_jk-Modul des Apache deshalb im „Sticky Session"-Modus zum Einsatz [2] [5] [6]. Damit werden nach der Erzeugung einer Sitzung auf einem bestimmten Tomcat-Knoten alle späteren Anfragen, die zur selben Sitzung ge-

hören, auch an denselben Knoten gesendet. Erst wenn dieser ausfällt, wird ein anderer Cluster-Knoten genutzt.

Die Stunde der Wahrheit

In der nächsten Phase wurden die einfachen funktionalen Tests unter Last wiederholt. Ein Lasttreiber eröffnete kontinuierlich parallele Sitzungen über die Test-JSP und veränderte Attribute in diesen Sitzungen. Es wurden die Korrektheit der Replikationsergebnisse kontrolliert und das Verhalten beim Starten oder Stoppen von Webanwendungen und beim Ausfall der Knoten betrachtet. Bei diesen fortgeschrittenen Tests zeigten sich leider erste Merkwürdigkeiten:

- Bei stabiler Verfügbarkeit aller Knoten wurden unter Last einige Replikationsnachrichten nicht korrekt verarbeitet.
- Die Erhöhung des *tcpSelectorTimeout*-Parameters im Receiver der Replikationsnachrichten hatte auch im Normalbetrieb unerwartet negative Auswirkungen auf die Antwortzeiten.
- Nach Herunterfahren eines Knotens wurden auf den verbliebenen Knoten nicht alle Sitzungen korrekt beendet (expired).
- Es gab wiederholte Ausnahmen, die im Log der Knoten vermerkt waren und auf eine ungenügende Synchronisierung der Replikation bei parallelen Zugriffen auf dieselbe Sitzung hindeuteten.

Keine Panik: Alle diese Probleme sind in Folge dieses Projektes im Tomcat 5.0.30 und 5.5 behoben. Der Umgang mit den ersten beiden Problemen soll im Folgenden näher erläutert werden.

Clever & smart

Der Verlust von Replikationsnachrichten wurde dadurch festgestellt, dass bei kontinuierlicher Belastung des ersten Knotens in einem zweiten Knoten die Anzahl der Sitzungen nicht übereinstimmte. Die Sitzungen können entweder via JMX-Adapter oder einfacher im Tomcat-Manager mit der Anfrage *http://<host>:<port>/manager/sessions* beobachtet werden. Eigentlich sollte bis auf kleine Verzögerungen, die durch die asynchrone Sitzungsreplikation entstehen, diese Sitzungsanzahl immer gleich sein.

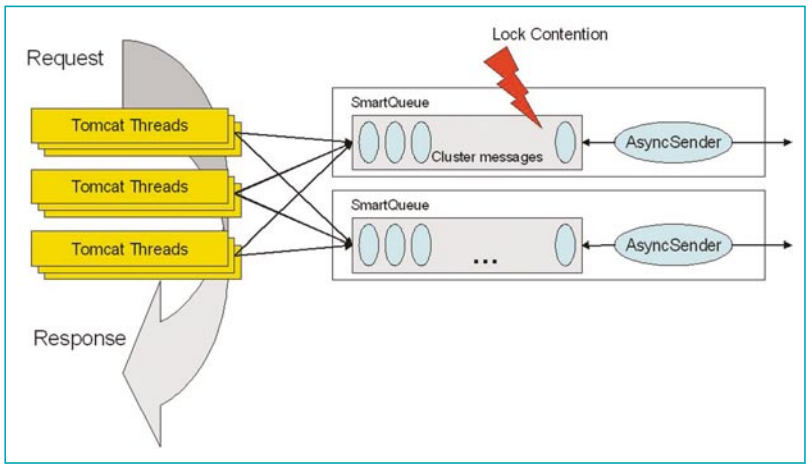

Abb. 1: Smartqueue im Tomcat 5

Um festzustellen, welche Nachrichten hier offensichtlich verloren gehen, wurden das Senden und Empfangen der Nachrichten besser protokolliert und dafür in den Tomcat-Code eingegriffen. Merkwürdigerweise kamen beim Empfänger immer wieder Nachrichten zu Attributveränderungen in Sitzungen an, deren Sitzungserzeugungsnachricht nie angekommen war. Jetzt half nur noch ein genaueres Code-Studium.

Im Tomcat 4-Cluster sind stets alle Daten einer Sitzung bei einem Zugriff auf die Sitzung repliziert worden [6]. Der Entwickler benutzte für die Replikationsnachrichten eine Queue, die er als SmartQueue bezeichnete. Die Queue verwarf beim Eintragen einer neuen Replikationsnachricht zu einer Sitzung stets alle dort noch vorhandenen – also noch nicht gesendeten – Nachrichten zu der gleichen Sitzung. Später wurden jedoch die Replikationsnachrichten auf Deltas umgestellt, d.h., eine Nachricht enthielt nicht mehr alle Daten zu einer Sitzung, sondern nur noch die geänderten Daten [2]. Nach wie vor wurden die Nachrichten aber über eine SmartQueue versendet (Abb. 1). Deshalb konnte es unter Last dazu kommen, dass die erste Nachricht zu einer Sitzung (Sitzungserzeugung) von einem nachfolgenden Sitzungs-Delta in der Sende-Queue überschrieben wurde. Auf der Empfängerseite kam dann nur noch die Delta-Nachricht an, die wegen des Fehlens der Erzeugungsnachricht verworfen wurde. Nach Analyse des Problems wurde erfolgreich ein Fix erstellt. Nach einigen Tests wurde dieser dann über die Tomcat-Bug-Datenbank Bugzilla [11] gemeldet und der Patch ist heute Bestandteil des offiziellen Release.

NIO blockiert

Das zweite Problem ging ebenfalls auf einen Codebug zurück. Beim Lesen von Replikationsnachrichten wird NIO des Java 1.4 eingesetzt. Hier kommt es häufig zu Fehlern wegen falscher Annahmen, welche Methoden einfach blockieren. Schon während der initialen Cluster-Implementierung war wohl aufgefallen, dass manche Aufrufe hängen bleiben, weshalb zur Umgehung ein Time-out eingebaut wurde. Dies führte aber dazu, dass das Time-out entweder sehr niedrig eingestellt werden musste und dann häufige unnötige Aktionen ausgelöst wurden oder aber es wurde höher eingestellt, sorgte dann aber für Stockungen bei der Verarbeitung von Replikationsnachrichten.

In unserem Projekt wurde wieder eine Debug-Fassung des Codes gemacht, um den blockenden Aufruf zu finden. Es wurde schnell klar, dass es sich um die Methode *java.nio.channels.SelectionKey.interestOps()* handelte, die auf das Ende eines parallelen *Selector.select()* wartete. Anhand der detaillierten Analyse konnte der Committer Filip Hanik sofort einen Fix für diesen Fehler erstellen.

Design-Review

Nachdem nun die Grundfunktionen des Clusters sichergestellt waren, wurde der Cluster-Code nochmals einem Review unterzogen. Dabei lag der Schwerpunkt der Betrachtung auf dem Replikationsmechanismus. Gemäß dem Konzept, dass die Re-

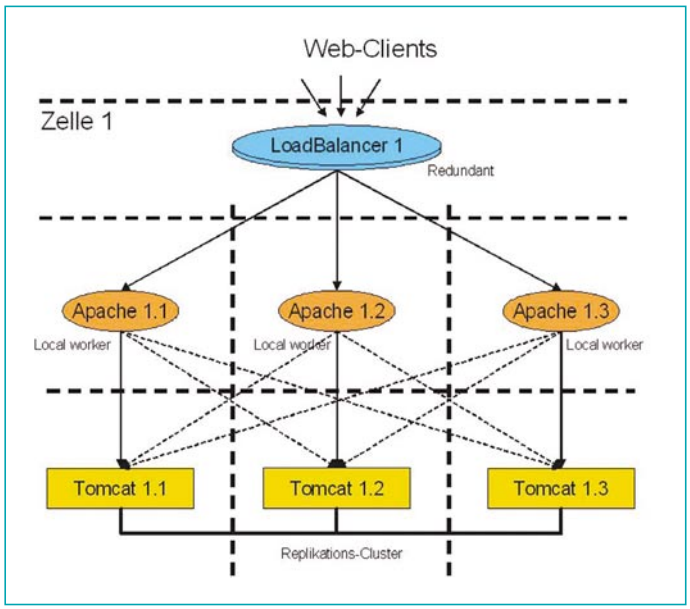

Abb. 2: Routing-Hierarchie innerhalb einer Domain (Zelle)

plikation nicht den Betrieb der eigentlichen Webanwendung gefährden soll, wurde besonders darauf geachtet, wo der Replikationsmechanismus den Betrieb des gesamten Clusters negativ beeinflussen könnte. Es fanden sich bald einige gravierende Schwächen:

- fehlende JMX-Unterstützung zum Monitoring wichtiger Cluster-Betriebszustände
- *OutOfMemory*-Gefahr bei Stau von Replikationsnachrichten in der Sende-Queue
- Sperrüberlastung unter hoher Last auf der Replikations-Queue (Lock Contention)
- nur globale Replikation auf allen Cluster-Knoten

Zum Glück bezogen sich diese Schwachstellen auf einige wenige, sehr gut abgrenzbare Teile des Cluster-Codes. Eine Abschätzung des Aufwands ergab knapp zwei Personenwochen, um diese Mängel durch eigene Alternativimplementierungen zu beseitigen. Gleichzeitig schien es möglich, mit geringem Aufwand die Implementierung auch für längere Zeit zu pflegen, da sie nur wenige Schnittstellen zum restlichen Cluster-Code benötigte [1] [7].

Asymmetrisches Locking

Die ersten drei Mängelpunkte bezogen sich alle auf den Teil des Clusters, der fertige Replikationsnachrichten an die anderen Cluster-Knoten versenden soll. Dort kommt eine Queue pro anderen Partnerknoten zum Einsatz, auf die einerseits die Tomcat Threads zugreifen, die Anfragen verarbeiten und am Ende die zugehörigen Replikationsnachrichten in die Queue einstellen wollen. Andererseits gibt es aber auch einen Hintergrund-Thread zum Senden, der bei Vorhandensein von Nachrichten in der Queue jeweils eine einzige Nachricht entfernt und über die Sockets zu den jeweiligen Empfängerknoten versendet. Dabei werden die Zugriffe der Anfrage-Threads und des Sende-Threads über einen einzigen Lock synchronisiert (Abb. 1).

Es ist zu erwarten, dass es unter Last zu einer Lock Contention kommt, d.h., dass die vielen Anfrage-Threads die Queue häufig sperren, sodass der Hintergrund-Thread der Sitzungsreplikation nicht mehr ausreichend Nachrichten entnehmen kann. In diesem Falle, oder aber auch wenn das Senden wegen Netzstörungen lange dauert, käme es außerdem zu einem stetigen Wachstum der Queue mit potenziellen *OutOfMemory*-Problemen.

Der Neuentwurf begrenzte zum einen die maximale Länge der Queue durch einen Konfigurationsparameter. Beim Queue-Stau würden zwar Nachrichten verloren gehen, der eigene Knoten aber störungsfrei weiter funktionieren. Wichtiger noch war aber die Beseitigung der Lock Contention. Hierzu wurde einerseits eine asymmetrische Sperre auf der Queue eingeführt, die bei mehreren wartenden Threads immer dem Sende-Thread zuerst die Sperre zuteilt. Andererseits wurde das Entnehmen von Nachrichten aus der Queue so umgebaut, dass der Sende-Thread immer alle Nachrichten aus der Queue nimmt und diese dann sofort wieder freigibt, bevor er die Nachrichten tatsächlich versendet.

Die gesamte eigene Implementierung wurde mit einigen Durchsatzzählern und Statusinformationen versehen, die über ein JMX MBean für das Monitoring zur Verfügung gestellt wurden. Dieser Code ist noch nicht in die offizielle Tomcat-Codebasis eingegangen; die Cluster-Entwickler haben aber bereits ihre Bereitschaft erklärt, die Implementierung als alternativen Replikationssender mit in die offizielle Tomcat-Distribution aufzunehmen [7].

Nicht globale Replikation

Da wir in diesem Projekt aus verschiedenen Gründen (z.B. redundante Internet-Anbindungen) mit neun Tomcat-Knoten arbeiten, war es nicht vertretbar, die Sitzungen jeweils auf alle Knoten zu replizieren. Außerdem wollten wir unabhängige Problem-Domains, die etwa durch unabhängige Netze gegeben waren, nicht durch zusätzlichen Querverkehr mit Replikationsnachrichten aneinander hängen. Folgendes Konzept wurde angestrebt:

- Verwendung von Sticky Sessions: Bei stabilem Betrieb wird eine Sitzung komplett von einem festen Tomcat-Knoten verarbeitet.
- Partitionierung der Knotenmenge durch Nutzung von mehreren unabhängigen Zellen mit je eigener Internet-Verbindung und jeweils redundant ausgelegten Apache- und Tomcat-Knoten. Jede Zelle hat eine eigene Service-IP, die via DNS verbreitet wird. Störungen sollen sich nicht über Zellengrenzen hinaus ausbreiten. Transparentes Failover bei Einzelfehlern oder Wartung wird innerhalb einer Zelle durch zellenlokale Sitzungsreplikation bzw. Tomcat-Cluster garantiert.
- Korrektur von Zugriffen auf falsche Zellen durch Querverbindungen.
- Automatischer Abbau von ungenutzten Querverbindungen zur Freigabe von Tomcat Threads.

Sticky Sessions werden vom Verbindungsmodul *mod_jk* schon lange unterstützt. Die nächsten zwei Anforderungen lassen sich durch einen zellenlokalen Cluster realisieren. Innerhalb einer Zelle wird mit drei Ebenen gearbeitet (Abb. 2):

- Ein Hardware-Loadbalancer verteilt Anfragen auf die mehrfach vorhandenen Webserver-Knoten. Außerdem wirkt er als HTTPS-Endpunkt und dient dazu, nachgelagerte Webserver komfortabel aktivieren und deaktivieren zu können.
- Die zweite Ebene besteht aus drei Apache-Webserver-Knoten. Diese bearbeiten Anfragen nach statischen Informationen und leiten Anfragen nach dynamischen Informationen über *mod_jk* an die Tomcat-Webcontainer weiter. Jeder Webserver hat einen präferierten Webcontainer-Knoten, auf dem er neue Sitzungen erzeugt (local worker im *mod_jk*).
- Die dritte Ebene besteht aus ebenfalls drei Tomcat-Webcontainer-Knoten, die zum selben Cluster gehören.

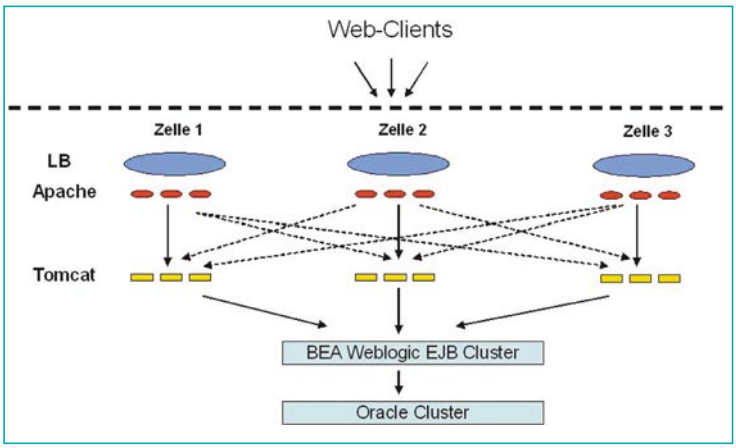

Abb. 3: Architektur eines hochverfügbaren und stabilen Apache/Tomcat-Webserver-systems

Cluster-Domains – Routing-Hierarchie

Wie aber lassen sich die lokalen Zellen nun sinnvoll zu einem Gesamtkonstrukt verbinden? Die Nutzer greifen auf den Service über HTTP zu. Da jede Zelle über eine unabhängige Internet-Anbindung verfügt, bietet es sich an, die Zugriffe über je eine eigene IP-Adresse pro Zelle zu verteilen. Dann kann es allerdings passieren, dass ein Client bzw. Browser während einer laufenden Sitzung die Serviceadresse neu im DNS auflöst und danach versucht, die Sitzung unter einer der anderen IP-Adressen (also Zellen) fortzusetzen. Nach unseren Beobachtungen passiert dies bei etwa fünf Prozent aller Sitzungen. Da unsere Cluster aber nur lokal in der gleichen Zelle replizieren sollen, benötigen wir also eine Möglichkeit, Client-Anfragen, die in einer falschen Zelle landen, auf einen Webcontainer in der richtigen Zelle weiterzuleiten (Abb. 3).

Wir untersuchten, inwieweit die Verbindungslogik zwischen Apache und Tomcat in der Lage war, solche intelligenten Routing-Entscheidungen zu treffen. Dabei zeigte sich, dass die komplette Logik in der Loadbalancer-Implementierung des mod_jk liegt, genauer in der Funktion *get_most_suitable_worker()* in der Datei *jk_lb_worker.c*. Der mod_jk Loadbalancer kannte jedoch nur die Konzepte *local worker* und *stickyness*. Sobald aber ein Webcontainer ausfällt oder in Wartung ist, gibt es keine Möglichkeit sicherzustellen, dass alle Apaches die Anfragen für diesen Container auf einen der verbliebenen Container der gleichen Zelle weiterleiten. Dies ist aber nötig, denn nur ein Tomcat in derselben Zelle besitzt die replizierte Sitzung. Wir ergänzten deshalb den mod_jk Loadbalancer um die Konfigurationsoption *worker.domain*, mit der man festlegen kann, welche Tomcat-Webcontainer im gleichen Cluster liegen. Diese Anpassungen wurden Ende 2004 vom Committer Mladen Turk in Version 1.2.7 von mod_jk offiziell aufgenommen [5].

Schließlich galt es, noch zu prüfen, inwieweit die vielen möglichen Apache-Tomcat-Verbindungen zu einem Ressourcenproblem innerhalb des Tomcat werden können. Jede Verbindung zwischen Apache und Tomcat belegt innerhalb von Tomcat einen festen Thread, der auf Anfragen auf dieser Verbindung wartet. Bei dem von uns eingesetzten Apache 1.3 werden die Verbindungen für jeden seiner vielen Arbeitsprozesse unabhängig verwaltet. Bei neun Apaches wäre also unter Last und nach einiger Betriebszeit vorstellbar, dass ca. 900 Verbindungen zu einem Tomcat laufen, obwohl nur 100 davon regelmäßig genutzt werden. Wir bauten deshalb in mod_jk eine weitere – bis dahin nicht vorhandene – Möglichkeit ein, ungenutzte Verbindungen nach einiger Zeit auch wieder abzubauen.

Letzter Feinschliff

Die Beobachtung des Replikationsdatenvolumens der echten Anwendung zeigte, dass bei manchen Anfragen recht große Datenmengen versendet wurden. Eine Dis-

kussion mit den Entwicklern ergab jedoch, dass diese Daten grundsätzlich jederzeit für den Nutzer transparent neu erzeugt werden können. Sie wurden nur aus Optimierungsgründen im Sinne eines Cache in der Sitzung gehalten.

Da wir mit Sticky Sessions arbeiteten, hielten wir es für unnötig, diese Daten kontinuierlich zu replizieren. Wir schrieben deshalb ein so genanntes Valve für Tomcat, mit dem die Entwickler aus der Webanwendung heraus feststellen konnten, ob die aktuelle Anfrage auf dem primären Knoten der Sitzung verarbeitet wird oder aber ob die Anfrage wegen Wartung oder Störung auf einem sekundären Knoten gelandet ist. Die Entwickler sorgten ihrerseits dafür, dass die Daten zwar in der Sitzung lagen, aber nicht mehr repliziert, sondern nur bei Wechsel des Knotens neu erzeugt wurden. Hierdurch wurden das Replikationsvolumen und damit die Netz- und Speicherbelastung erheblich verringert.

Bisherige Produktionserfahrung

Seit Ende 2004 befindet sich die Lösung in vollem Umfang in Produktion. Die Anwendung läuft performant und stabil. Die über die JMX MBeans gesammelten Statusinfomationen der Cluster dokumentieren eine hohe Robustheit. Einige Störungen im Netzbereich wurden bereits transparent abgefangen.

Parallel wurden Verfahren zum rotierenden Update der Webanwendung erarbeitet, die die Flexibilität der Lösung zur Vermeidung von Ausfallzeiten nutzen. Da hier gemischte Versionen der Webanwendung im gleichen Cluster zum Einsatz kommen, ist es ohne Tests nicht immer einfach einzuschätzen, ob konkrete Webanwendungsversionen für diese Form des Updates geeignet sind. Natürlich wurde hier in einer frühen Phase des Projekts recht hoher Aufwand getrieben, um eine passende Systemintegration zu erarbeiten. Bislang zeigt sich aber, dass wir im Gegensatz zu vielen anderen Projekterfahrungen nahezu keine Probleme im normalen Betrieb, bei gewöhnlichen Ausfällen und Releasewechseln bekommen haben.

Nobody is perfect

Was natürlich noch fehlt, ist eine globale Administrationsschicht für solche Systeme. Ein Konstrukt aus neun Webservern und ebenso vielen Webcontainern individuell zu administrieren stellt einen nicht unerheblichen Aufwand dar, insbesondere wenn zu jedem Zeitpunkt die volle Funktionalität der Anwendung sichergestellt sein muss. Es wäre sehr nützlich, wenn es hier eine robuste Management-Oberfläche gäbe, insbesondere zur Unterstützung von Deployment-Verfahren.

Ein Teil davon könnte auch durch Verbesserung der Apache-Tomcat-Integration erreicht werden. So wäre es sehr nützlich, wenn ein Container beim Starten oder Stoppen einer Webanwendung dies den vorgelagerten Apache-Servern mitteilen könnte, damit diese nicht versuchen, eine nicht vorhandene Webanwendung anzusprechen. Zwar erkennt Apache nach dem Stoppen eines gesamten Tomcat-Knotens, dass dieser nicht mehr ansprechbar ist, und wählt dann für den Nutzer transparent einen anderen Knoten in der gleichen Cluster Domain aus. Problematisch ist jedoch die Zeit, während der der Knoten startet oder stoppt. Dann gibt es nämlich Phasen, in denen der Knoten ansprechbar, die Webanwendung aber noch nicht oder nicht mehr vorhanden ist. Dies führt für einige Sekunden zu Fehlern auf Seiten der Nutzer. Leider ist ein Versuch der Community, diese Eigenschaft im Rahmen von mod_jk2 bereitzustellen, gerade erst gescheitert – dieses Modul wird nicht mehr weiterentwickelt.

Ebenso fehlt ein Nachrichtentyp innerhalb des Tomcat Cluster, um das Stoppen eines Knotens anzukündigen. Bislang erkennen die anderen Cluster-Mitglieder das Stoppen nur durch ausbleibende Heartbeat-Informationen. Nach Ablauf einer konfigurierbaren Zeit ohne Eintreffen von Heartbeat-Informationen gilt der Partnerknoten als nicht mehr vorhanden. Die Erfahrung zeigt, dass diese Time-outs nicht zu kurz eingestellt werden dürfen. Eine vollständige Garbage Collection kann schon mal länger dauern und harmoniert damit nicht gut mit der echtzeitartigen Funktionalität des Heartbeat; andererseits bekommt man bei zu langen Time-outs Probleme, wenn man einen Knoten stoppt und sofort wieder startet. Dann versuchen die Partner weiterhin die alten – nicht mehr funktionierenden – Verbindungen zu dem durchgestarteten Knoten zu verwenden.

Die Moral von der Geschicht?

Auf den ersten Blick schienen die aktuellen Fähigkeiten von Apache/Tomcat ausreichend, die Anwendung darauf aufzubauen. Erst bei Tests unter Last zeigten sich Fehler, die in der späteren Produktion fatale Folgen gehabt hätten. Nicht nur zur Fehleranalyse, sondern auch schon zum Verständnis der exakten Funktionsweise, etwa aller Konfigurationsoptionen, war es nötig, klar abgegrenzte Teile des Codes zu studieren. Eine sehr präzise Beschreibung der Fehler sorgte dann für deren Beseitigung durch die zuständigen Tomcat Committer.

Der Aufbau eines unternehmenskritischen Clusters überstieg die bis zu diesem Zeitpunkt vorhandenen Möglichkeiten. Das Studium der einschlägigen Mailing-Listen zeigte schnell, dass wir uns auf Neu-

Tomcat-News

- Tomcat 5.5.7 Beta ist veröffentlicht
  - JMX-Support für den Cluster
  - Neues Modul für die verlässliche Speicherung der server.xml
  - mod_jk Installer für den IIS
- Tomcat 5.0.30 ist veröffentlicht
- Neues mod_jk 1.2.8 [5]
  - Neuer LoadBalancing-Algorithmus mit Unterstützung für mehrere LocalWorker
  - Domainkonzept für Worker zur Verbesserung der Cluster-Fähigkeit
  - Kompilierung unter Windows nun mit frei verfügbaren Microsoft-Umgebung möglich
  - Support für den IIS und Unterstützung weitere Betriebsysteme für den Apache
- Referenzkarte zum Tomcat 5.0.29 auf tomcat.objektpark.org/referenzkarte veröffentlicht.

Was haben wir im Projekt gelernt?

Open Source mit Apache und Tomcat bedeutet, dass wir uns in gewissen Situationen nur auf den Quellcode verlassen konnten.
Die umfangreichen Reviews, Tests und Anpassungen haben eine perfekt passende Lösung ergeben, die auch in schwierigen Situationen sehr gut verstanden ist.
Eine aktive Community steigert die Lösungsqualität, die Integration braucht aber ihre Zeit und eine gute Vorbereitung.

land bewegten. Den Entschluss, die Produkte dennoch zu verwenden und die fehlenden Funktionalitäten selbst zu entwickeln, haben wir nicht bereut. Wir wurden belohnt mit einer stabilen Umgebung, die darüber hinaus in ihrer Funktionsweise sehr gut verstanden ist. Letzten Endes machen wir jedoch den Erfolg dieses Ansatzes daran fest, ob es uns gelingt, die Committer von der Übernahme unserer Anpassungen an Apache/Tomcat in die offizielle Code-Basis zu überzeugen. Für große Teile ist dies bereits geschehen, sie stehen damit der gesamten Community zur Verfügung.

Committer von Open-Source-Anwendungen haben mit einer sehr schwankenden Qualität von Support-Anfragen zu tun. Darüber hinaus existiert keine traditionelle Trennung zwischen First-, Second- und Third-Level-Support. Deshalb sind die ersten Reaktionen auf Anfragen häufig abweisend. Das erfolgreiche Verfolgen von Support-Anfragen setzt in manchen Projekten voraus, dass man selbst in der Lage ist, die Problemstellung einzukreisen und reproduzierbar zu machen. Selbst nach Vorlage von Patches braucht es häufig bei fehlender Reputation eine erhebliche Überzeugungsarbeit, um diese in den offiziellen Code zu übernehmen.

Bei komplexen Fragestellungen müssen sich die Anwender deshalb entscheiden, ob sie diesen langwierigen und schwierigen Weg selbst gehen wollen und können oder aber ob sie den Einsatz von Open-Source-Anwendungen durch Hinzunahme kommerziellen Supports absichern.

Wir möchten uns bei den Committern Filip Hanik und Mladen Turk für die sehr gute Zusammenarbeit bedanken und hoffen, dass die guten Ergebnisse den ein oder anderen Leser dazu inspirieren, sich aktiv an der Weiterentwicklung von Apache und Tomcat zu beteiligen. Die noch nicht in die offiziellen Releases eingegangenen Code-Anpassungen haben wir als Download zur Verfügung gestellt [7]. Wir freuen uns auf Kommentare und Anregungen – besuchen Sie also unsere TomC@ Site und das TomC@-Forum oder die Tomcat-Mailing-Listen [8] [9] [10].

Peter Roßbach (pr@objektpark.de) ist als freier J2EE-Systemarchitekt, Entwickler und Trainer tätig. Er ist Committer der Apache Software Foundation und arbeitet aktiv im Projekt Tomcat.

Rainer Jung (rainer.jung@kippdata.de) hat 1998 kippdata informationstechnologie gegründet. Das Unternehmen steht für innovative Konzepte und komplette Lösungen in großen Systemlandschaften. Für Apache/Tomcat bietet kippdata professionelle Supportdienstleistungen.

Links & Literatur

[1] jakarta.apache.org/tomcat

[2] Peter Roßbach: Magische Momente. Grundlagen eines Tomcat Cluster, in *Java Magazin* 2 2005

[3] Log4J: logging.apache.org/log4j

[4] Ethereal Network Protocol Analyzer: www.ethereal.com

[5] jakarta.apache.org/tomcat/connectors-doc

[6] Peter Roßbach (Hrsg.): Tomcat 4X: Die neue Architektur und moderne Konzepte für Web-Anwendungen im Detail, Software & Support Verlag, 2002

[7] www.kippdata.de/red/tomcat

[8] www.javamagazin.de/tomcat

[9] www.mail-archive.com/tomcat-user@jakarta.apache.org, www.mail-archive.com/tomcat-dev@jakarta.apache.org

[10] tomcat.objektpark.org

[11] jakarta.apache.org/tomcat/bugreport.html