03.04

Deutschland € 5,50 Österreich € 6,50 Schweiz SFr 10,70 / Luxemburg € 6,50

S&S

Javamagazin

Java magazin

Internet & Enterprise Technology

XML extra included

Maximale Leistung

Performance, Skalierbarkeit, Ausfallsicherheit durch J2EE Clustering

Host-Integration

Anschluss an die Java-Welt: Strategien und Tools

www.javamagazin.de

„Direktbank“
Implementierung mit BEA Workshop

Apache Ant 1.6
Build Manager mit neuen Features

Tom C@ – Die Kolumne
Luxus Access Logging mit Tomcat

Server-Management
JMX mit Weblogic und JBoss

Java Security
JDBC-Kommunikation absichern

mit CD!

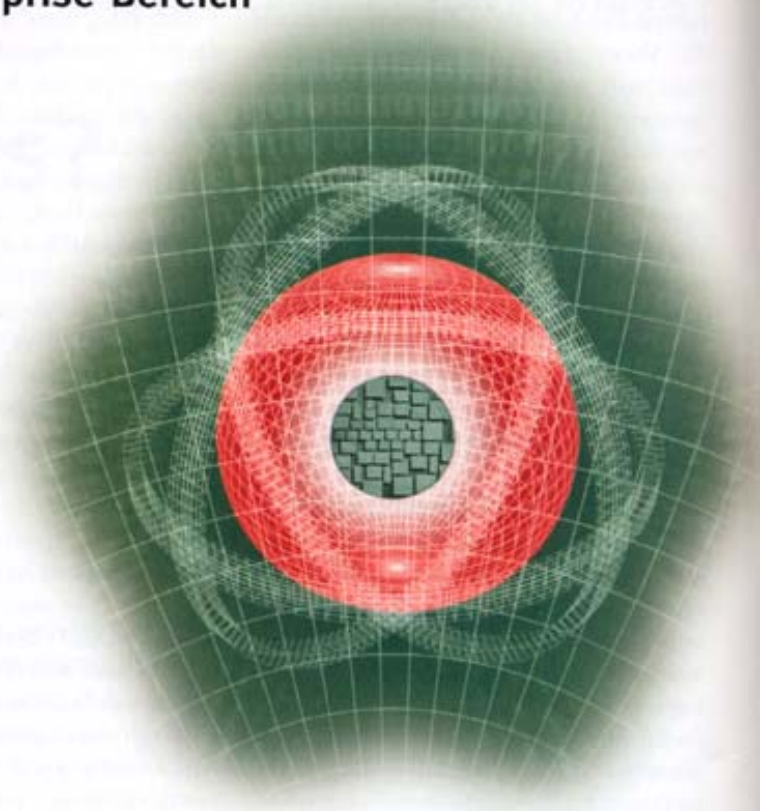
D 45867

4 194586 705506 03

Architekturen und Topologien im Enterprise-Bereich

von Roger Zacharias

Ein Herz für Cluster



Wie komme ich zur performanten, skalierbaren und hochverfügbaren Applikation? Zurzeit entstehen verschiedene Systeme, welche das Kürzel „Enterprise" im Namen tragen und die sich rühmen, hochverfügbar, hochperformant und fehlertolerant zu sein. Doch was steckt hinter diesen Konzepten?

Zunächst handelt es sich hierbei um nichtfunktionale Anforderungen, auch als Systemqualitäten bezeichnet, die es abzudecken gilt. J2EE bzw. die Implementierung dieser Spezifikation – der J2EE Application Server – bietet hierbei, wie uns suggeriert wird, alles, was das Herz begehrt. Der Entwickler muss sich nur noch um die Implementierung der Geschäftslogik kümmern, alle technischen Belange hat der Server im Griff. Doch leider ist dies nicht der Fall und für die Umsetzung einer performanten, skalierbaren und hochverfügbaren Applikation sind auch weiterhin detaillierte Überlegungen notwendig, die im Idealfall von der Berufsgruppe der Softwarearchitekten durchgeführt werden. In diesem Artikel sollen sowohl ein Überblick gegeben als auch eine Einzelbetrachtung der angesprochenen Konzepte durchgeführt werden, sodass der Leser nach der Lektüre die Grundkenntnisse besitzt, um sich intensiver mit dieser hochinteressanten Materie zu befassen.

Zu Beginn soll auf die nichtfunktionalen Anforderungen (NFA) eingegangen werden. Anschließend wird auf die Umsetzung der NFA mithilfe des Clustering eingegangen. Im dritten Schritt wird dieses Wissen auf die J2EE angewendet. Abschließend werden einige weitergehende Überlegungen und Besonderheiten dargestellt.

Nichtfunktionale Anforderungen

Folgende nichtfunktionale Anforderungen, welche auch gern als „die -ilities" bezeichnet werden, spielen im Enterprise-Bereich neben den funktionalen (Business-)Anforderungen eine besondere Rolle:

- Performance – garantiert, dass das System in den geforderten Zeiträumen antwortet
- Availability – garantiert, dass ein Systemdienst bzw. eine Systemkomponente immer verfügbar ist
- Security – garantiert, dass das System nicht „bloßgestellt" wird
- Scalability – garantiert, dass die Systemqualität aufrechterhalten bleibt, wenn die Systemlast steigt
- Extensibility – garantiert, dass Systemfunktionalität geändert oder hinzugefügt werden kann, ohne die bestehende Funktionalität zu beeinträchtigen
- Reliability – garantiert die Integrität und Konsistenz der Anwendung und ihrer Transaktionen
- Manageability – garantiert, dass das System die Möglichkeit des Managements (Überwachung & Administration) mit sich bringt, um die restlichen Systemqualitäten dauerhaft zu gewährleisten
- Maintainability – garantiert, dass bestehende Fehler korrigiert werden können, ohne dass die bestehende Funktionalität beeinträchtigt wird

Für jede dieser nichtfunktionalen Anforderungen, aus denen sich letztendlich dezidierte Zusicherungen in der Systemspezifikation ergeben, existieren verschiedene Konzepte zur Umsetzung. An dieser Stelle sollen vor allem diejenigen herausgegriffen werden, welche insbesondere durch die Topologie beeinflusst werden: Performance, Availability und Scalability. Die Topologie beziehungsweise die Technik, welche hier zum Tragen kommt, ist das Clustering.

Clustering

Sicher hat jeder schon einmal den Begriff Clustering gehört, doch was ist genau darunter zu verstehen? Nehmen wir uns nacheinander die drei angesprochenen Systemqualitäten vor.

Zunächst zur Performance: Wie kann man die Performance einer Enterprise-Applikation gewährleisten? Voraussetzungen hierfür sind sicher gutes Design und guter Code, was die Performance aber nur begrenzt steigern kann. Letztendlich ist man immer auf die zugrunde liegende Hardwareplattform angewiesen. Reicht die Performance zum Beispiel bei gestiegenen Anforderungen bezüglich Systemlast einer Applikation mit optimalem Design und Code nicht mehr aus, muss man die zugrunde liegende Hardware ändern.

Hier kommen wir zur Scalability, welche garantiert, dass die Systemqualitäten, also in diesem Falle die Zusicherungen bezüglich der Performance, auch dann erhalten bleiben, wenn die Systemlast steigt. Grundsätzlich existieren zwei Arten zur Umsetzung der Skalierbarkeit – vertikal und horizontal. Unter vertikaler Skalierung versteht man das Hinzufügen von Kapazität (Speicher, CPU etc.) zu einem existierenden Server. Wo diese nicht mehr ausreicht, wird auf horizontale Skalierung zurückgegriffen, d.h. das Hinzufügen von Servern zum Gesamtsystem, also letztendlich die Aufteilung der Systemlast auf mehr als einen Server, um eine parallele Verarbeitung über Rechnergrenzen hinaus zu ermöglichen.

Der nächste Punkt ist die Availability, welche garantiert, dass ein Systemdienst bzw. eine Systemkomponente immer verfügbar sind. Dieser Aspekt ist insbesondere in geschäftskritischen Enterprise-Applikationen (z.B. im Bankenumfeld) von Bedeutung, wobei hier von High-Availability (HA), Fehlertoleranz und 24 x 7 gesprochen wird. Die Basis der Availability ist in jedem Fall die Beseitigung von Single Points of Failures (SPOF) durch Redundanz, d.h. die redundante Auslegung der hochverfügbaren Systemdienste bzw. -komponenten, sodass bei Ausfall die redundante Komponente übernehmen kann.

Diese drei Systemqualitäten – Performance, (horizontale) Scalability und Availability – und ihre Umsetzung führen zu der in Abbildung 1 dargestellten Cluster-Topologie und einer möglichen Definition des Clustering: Unter einem Cluster versteht man eine Gruppe von Komponenten, welche auf Basis von Redundanz und Lastverteilung zusammenarbeiten, um eine performantere und höher verfügbare Lösung als eine einzelne Komponente zu bieten; wobei diese Gruppe sich nach außen hin wie eine einzelne Komponente präsentiert.

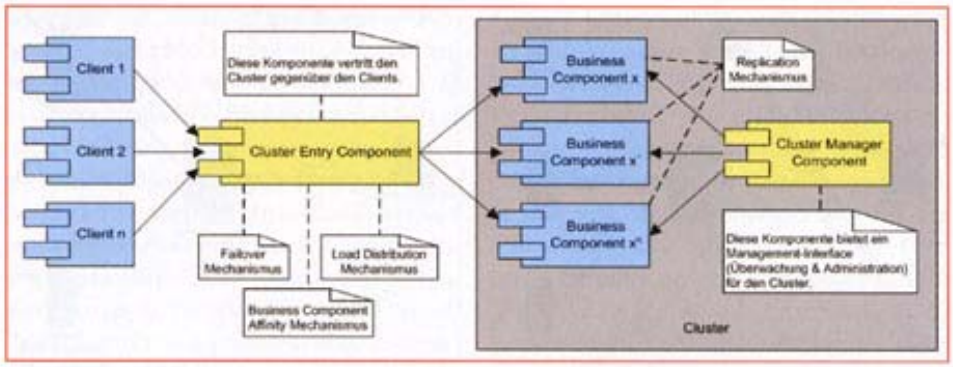


Abb. 1: Cluster-Topologie

Ziel ist der Zugriff der Client-Komponenten auf eine Geschäftskomponente, welche einen bestimmten Dienst erbringt (z.B. „Überweisung durchführen"). Diese Geschäftskomponente soll sowohl eine hohe Last von Seiten der Clients verkraften als auch hochverfügbar sein. Hierzu muss sie, wie soeben dargestellt, redundant skaliert werden. Dies sollte zudem für die Clients transparent sein, d.h., diese sollten im Idealfall weder den physikalischen Ort der Komponente kennen noch Wissen über die Anzahl der Komponenten besitzen müssen. Der Cluster stellt sich daher gegenüber den Clients als eine Einheit dar. Eine Cluster Entry Component (CEC) vertritt ihn gegenüber seinen Clients und stellt die einzige Schnittstelle zum Zugriff auf den Cluster zur Verfügung. Sie enthält das Wissen über den Cluster und entsprechende Logik für Mechanismen wie Load Distribution, Failover und Business Component Affinity, wodurch sie zur Basis der Erfüllung von Performance, Scalability und Availability wird.

Betrachten wir diese Aspekte genauer: Die Cluster Entry Component empfängt alle Anfragen an das System, ist also der Single Point of Entry. Sie agiert nun bzgl. Parallelverarbeitung so, dass sie diese Anfragen transparent auf die redundanten Komponenten verteilt, sodass diese die Requests parallel abarbeiten, wodurch eine bessere Performance erreicht wird. Diese Verteilung der Last wird als Load Distribution bezeichnet. Weiterhin beinhaltet diese Komponente einen Failover-Mechanismus, sodass sie bei Ausfall einer Komponente die Anfrage oder die Bearbeitung dieser transparent auf eine andere Komponente umleiten kann. Beide Mechanismen sind weiter unten ausführlicher beschrieben.

Solange jede Business Component vollkommen identisch ist, stellt dies kein Problem dar, da es für die Erbringung des Dienstes keine Rolle spielt, mit welcher Business Component der Client letztendlich interagiert. Muss die Komponente allerdings Client-spezifische Informationen speichern, d.h. ein Konversationsgedächtnis besitzen (wie z.B. der Warenkorb, der auch in diesem Artikel als Beispiel für die Notwendigkeit eines konversationellen Zustands herhalten muss), wird die Sache komplizierter. Hier kommen die Business Component Affinity und die Replication ins Spiel. Es ist nun wichtig, dass der entsprechende Client immer mit der Komponente spricht, die seine Statusinformationen innehat (Affinity), oder man müsste diesen Zustand auf alle Komponenten verteilen und permanent abgleichen, was generell zu aufwändig ist. Diese Affinity (Leo Dictionary übersetzt dies übrigens u.a. mit „besonderer Liebe" und „geistiger Verwandtschaft") wird in der Regel von der Cluster Entry Component gewährleistet, welche die Anfrage des Clients durch verschiedene Mechanismen (z.B. Session-ID über Cookies) immer an „eine" Business Component weiterleitet. Ein Problem würde nun allerdings beim Ausfall einer Business Component bestehen, da in diesem Fall die Statusinforma-

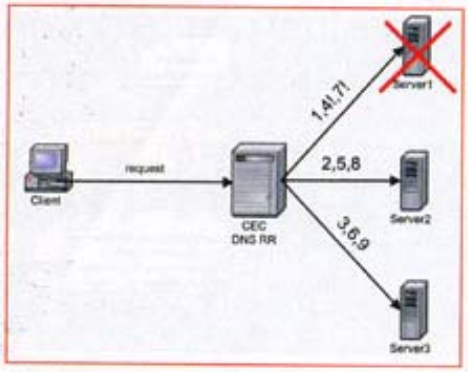


Abb. 2: Load Sharing mit DNS RR

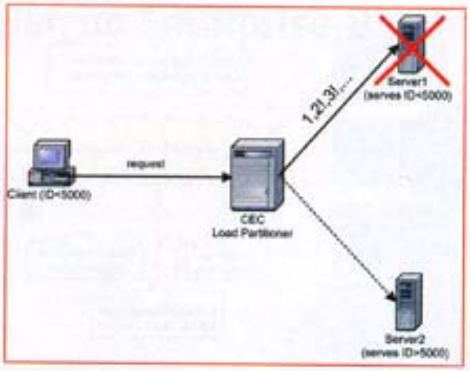


Abb. 3: Load Partitioning

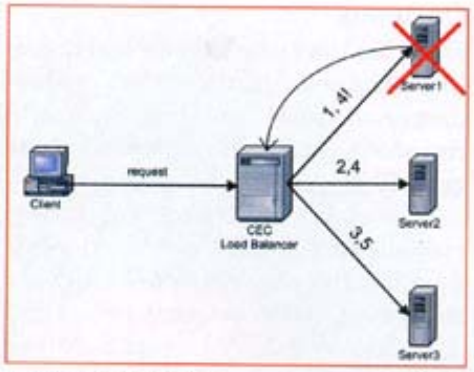


Abb. 4: Load Balancing

tionen verloren wären. Daher ist es notwendig, die Statusinformationen auf eine andere Komponente zu replizieren, sodass auch hier ein problemloses Failover möglich ist. Es existiert eine Reihe von Replikationsmechanismen, auf die später näher eingegangen werden soll.

Die dargestellte Cluster Manager Component hat, wie auch die Cluster Entry Component, das Wissen über den gesamten Cluster. Sie dient aber zum Management des Clusters, d.h., über sie ist es möglich, den Cluster zu administrieren und zu überwachen (Single Point of Administration). Weiterhin kann sie diverse andere Aufgaben wie Diagnose, Konfiguration und Recovery übernehmen.

Load Distribution

Die Verteilung der Last auf die einzelnen Cluster-Komponenten wird als Load Distribution bezeichnet. Hierfür existieren verschiedene Varianten: Die einfachste Variante ist das Load Sharing (Abb. 2), bei der die Last beliebig auf die Komponenten verteilt wird, ohne dass man von den Komponenten ein Feedback erhält. Das bekannteste Beispiel hierfür ist sicher das DNS Round Robin-Verfahren, bei dem der Domain Name Service als Cluster Entry Component genutzt wird. Er wird derart konfiguriert, dass er beim Auflösen des Servernamens keine feste IP-Adresse zurückgibt, sondern im Round Robin-Verfahren nacheinander die IP-Adressen der Cluster-Komponenten. Das Load Sharing ist damit sowohl einfach als auch kostengünstig, hat aber u.a. den Nachteil, dass durch die fehlende Rückmeldung bei einem Komponentenausfall der entsprechende Client in der nächsten Runde wieder an die ausgefallene Komponente weitergeleitet wird und damit auf einem Fehler läuft.

Die zweite Variante ist das Load Partitioning (Abb. 4), bei dem Clients abhängig von ihrem Zustand bestimmten Cluster-Komponenten zugewiesen werden. So wird zum Beispiel allen Clients, bei denen die Benutzernummer kleiner 5.000 ist, Komponente K1 und allen mit Benutzernummer größer 5.000 Komponente K2 zugewiesen. Auch hier erhält man im Allgemeinen kein Feedback von den Komponenten.

Die dritte Variante ist das fälschlicherweise oft als Oberbegriff verwendete Load Balancing (Abb. 5). Das wichtigste Kriterium für diesen Mechanismus ist das Feedback von den Business Components über deren Zustand. Fällt eine Komponente aus, so merkt dies die Cluster Entry Component und weist dieser bis zur Wiederaufnahme ihres Dienstes keine Client-Requests mehr zu. Weiterhin beherrscht eine Load Balancing-fähige Cluster Entry Component in der Regel intelligente Algorithmen für wirkliche Lastverteilung. Darunter fallen z.B. Verteilung basierend auf der unterschiedlichen Leistung der einzelnen Business Components, Zuteilung in Abhängigkeit von der Antwortzeit der Komponenten; prioritätsbasierte oder priorisierte Zu-

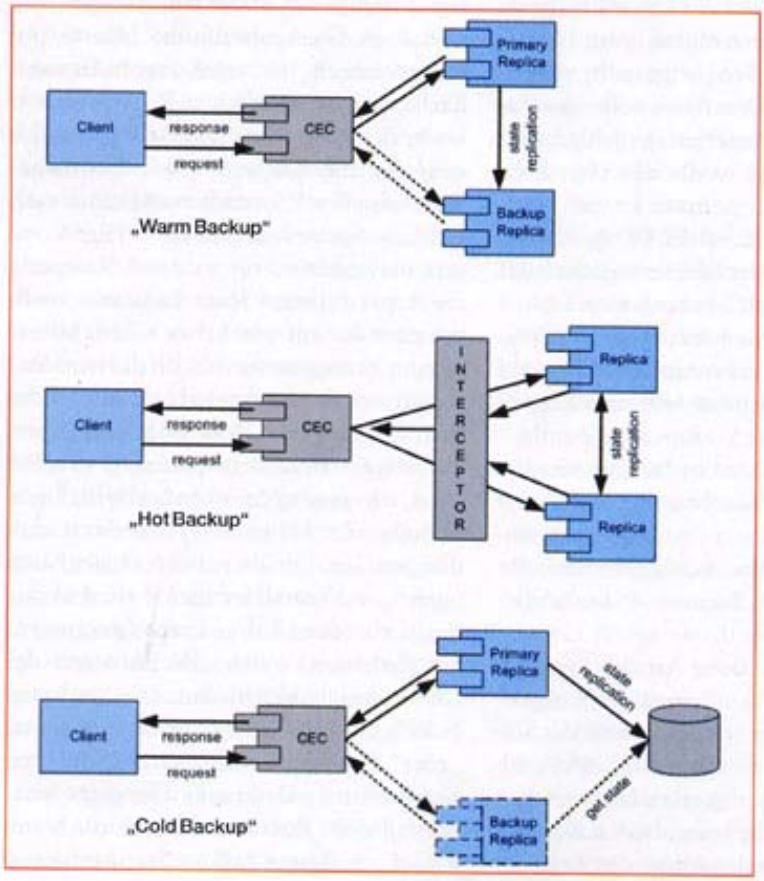


Abb. 5: Warm, Hot and Cold Backup

teilung an die zurzeit am wenigsten ausgelastete Komponente etc. Eine weitere Spezialisierung ist das Reverse Proxy Load Balancing, bei welchem je nach Art der eintreffenden Anfrage eine passende Business Component ausgewählt wird. Damit ist es zum Beispiel möglich, alle HTTPS-Anfragen an einen leistungsfähigeren Server und alle HTTP-Anfragen an einen leistungsschwächeren Server zu senden.

Replication und Failover

Wie bereits erwähnt, ist es für ein problemloses Failover bzgl. eines Konversationsgedächtnisses notwendig, die Statusinformationen auf eine andere Komponente zu replizieren. Hierzu existieren diverse Mechanismen, welche direkt auf die entsprechenden Failover-Konzepte abgebildet werden können und in die folgenden Kategorien fallen (Abb. 6):

Passive Replication: Bei der mittlerweile oft verwendeten passiven Replikation, auch als Warm Backup bezeichnet, behandelt nur die primäre Komponente alle Requests und erst wenn diese ausfällt, übernimmt eine replizierte aktive Sekundärkomponente. Die Statusinformationen werden hierbei periodisch synchronisiert.

Active Replication: Bei der aktiven Replikation, auch als Hot Backup bezeichnet, antworten alle Replikate auf alle Requests und es existiert eine Interceptor-Komponente, welche mehrfache Antworten blockt und lediglich eine Antwort an den Anfragenden zurückliefert. Auch hier werden die Statusinformationen periodisch synchronisiert.

Cold Backup: Bei dieser Variante, wie auch bei der Passive Replication, behandelt nur die primäre Komponente alle Requests und gleicht ihren Zustand periodisch mit einem passiven Datenspeicher ab. Fällt die primäre Komponente aus, wird ein Replikat erzeugt (manuell oder automatisch), welches die Statusinformationen zunächst vom Datenspeicher rekonstruieren muss und dann den Dienst übernimmt.

Abbildung auf J2EE

Die J2EE-Spezifikation bietet die Basis für verteilte Java-basierte Unternehmensanwendungen und spezifiziert hierbei u.a. TX-Management, Security, Persistenz sowie diverse Integrationsmechanismen. Sie lässt aber die behandelten Enterprise-Kriterien bzw. die Mechanismen zu deren Umsetzung fast vollständig undefiniert, was bedeutet, dass ein J2EE Application Server kein Clustering unterstützen muss. Prinzipiell unterstützen aber alle Hersteller diese Mechanismen proprietär, sodass diese nicht mehr teuer selbst entwickelt werden müssen und es hierbei nicht zu einer Verschwendung knapper Projektressourcen kommt. Die Unterstützung geschieht aber in unterschiedlichem Maßstab, wodurch dies ein sehr wichtiger Punkt bei der Auswahl eines Applikationsservers ist.

Jede Tier, inklusive der Datenhaltung, kann die Vorteile des Clusterings nutzen, wobei wir im J2EE-Bereich folgende drei Gebiete bezüglich dieser Mechanismen betrachten wollen:

- HTTP-Server/Web-Container
- EJB-Container
- J2EE-Basisdienste

HTTP-Server/Web-Container Clustering

Betrachten wir zunächst den HTTP-Server/Web-Container, bei dem die Cluster-Komponenten statische HTML-Seiten bzw. Servlets und JSPs sind und die Clients üblicherweise Web-Browser, welche über den HTTP-Server mit dem Web-Container interagieren. Hier kann die Cluster Entry Component auf verschiedene Art und Weisen realisiert werden: zum Beispiel durch DNS Round Robin, durch ein vom Applikationsserver mitgeliefertes Dispatcher-Servlet, welches in einem dem Cluster vorgeschalteten Web-Container deployed wird, einem Plugin-Modul für gängige Webserver oder durch entsprechende Hardware (z.B. Alteon, BIG-IP, Cisco Local Director), welche aber zum Teil nicht unerhebliche Kosten verursacht (bis zu $ 40.000). Bei der Wahl der Cluster Entry Component ist natürlich darauf zu achten, dass diese selbst keinen SPOF darstellt bzw. ob die dabei potenziell entstehende Downtime in Kauf genommen werden kann. Einen Vorteil haben hier die Hardware Load-Balancer, bei denen oft die Hardware selbst (z.B. Netzteile, HD, Netzwerkkarten etc.) redundant ausgelegt ist, wobei diese Features sich natürlich direkt auf die Anschaffungskosten niederschlagen. Da jeder Request durch die Cluster Entry Component geleitet wird, kann diese zudem schnell zum Bottleneck werden und ist daher entsprechend auszulegen. Da bei Servlets und JSPs jede Komponente vollkommen identisch ist, spielt es für die Erbringung des Dienstes keine Rolle, mit welcher Business Component der Client letztendlich interagiert. Die einzige Stelle für ein Konversationsgedächtnis im Web Container ist das HttpSession-Objekt, welches repliziert werden muss (siehe unten). Die Informationen über den Cluster selbst werden in der Regel in Cookies gehalten und bei jedem Request vom Client an die Cluster Entry Component weitergegeben, sodass auch hier Affinitätsmechanismen zum Zuge kommen können. Bei Änderung der Cluster-Topologie im laufenden Betrieb ist im Gegensatz zum EJB-Container Clustering in den meisten Fällen eine Rekonfiguration der Cluster Entry Component notwendig.

EJB-Container Clustering

Betrachten wir nun den EJB-Container: Hierbei sind die Cluster-Komponenten EJBs und die Clients Java-, CORBA- oder Web-Container-Clients. Für den Aufenthaltsort der Cluster Entry Component, welche die Load Balancing-, Failover- und Affinity-Logik sowie das Wissen um den Cluster enthält, existieren auch hier verschiedene Möglichkeiten: auf den Knoten, auf einem zwischengeschaltetem Load Balancer oder auf dem Client innerhalb des Proxy. Jede dieser Lösungen hat Vor- und Nachteile, wobei bei der dritten Variante die Vorteile (kein SPOF, kein Bottleneck durch zusätzliche Komponente etc.) überwiegen. Daher ist die Cluster Entry Component in diesem Bereich in den meisten Fällen im Remote Proxy enthalten, welcher vom Namens- und Verzeichnisdienst erst bei Bedarf (Lazy Class Loading) durch den Client bezogen wird. Die letztendliche Implementierung eines Remote Proxy bleibt dem J2EE Vendor überlassen, wodurch hier entsprechende Logik für zum Beispiel Failover, Cluster-Awareness oder Collocation integriert werden kann. So kann der Proxy zum Beispiel erkennen, ob sich der Client in derselben JVM wie das Ziel befindet und damit eine Remote-Kommunikation überflüssig wird (siehe Collocation). Weiterhin kann überprüft werden, ob ein

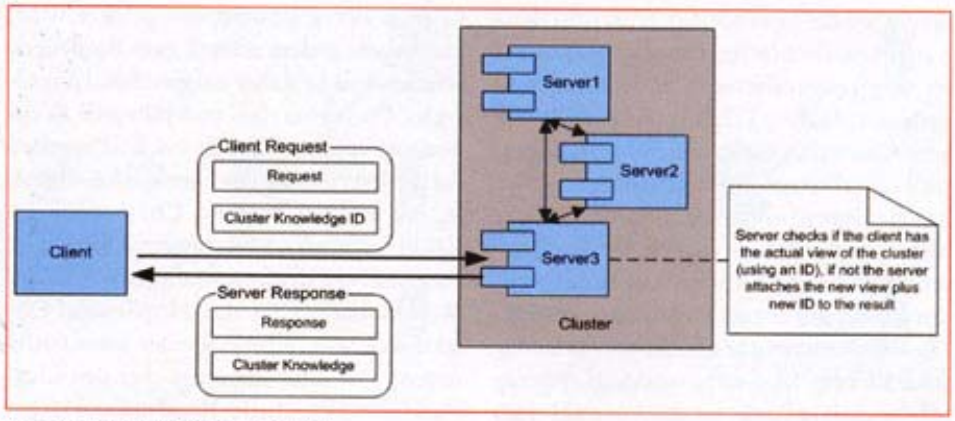


Abb. 6: Cluster-aware Remote Proxy

fehlgeschlagener Aufruf für den Client transparent ein zweites Mal durchgeführt werden kann (Silent Failover), wenn zum Beispiel der Zielserver nicht erreicht werden konnte. Wird aber zum Beispiel eine Exception vom erreichten Server zurückgeliefert, wird aus Gründen der Unwissenheit um die Idempotenz (Seiteneffektfreiheit) der Methode die Exception an den Aufrufer zurückgeliefert und die Client-Applikation muss innerhalb der Geschäftslogikregeln ihre eigene Entscheidung bzgl. der Behandlung des Problems treffen.

Was aber, wenn sich die Cluster-Topologie ändert, wenn z.B. horizontal skaliert wird oder ein Server ausfällt? Dann ist der Proxy „stale". Dieses Problem wird meist dadurch gelöst, dass der Proxy bei jeder Kommunikation mit einem Clusterknoten sein Wissen um den Cluster aktualisiert und dadurch immer den aktuellen Zustand des Clusters kennt (Abb. 6).

Statusbehaftete Komponenten wie HttpSession oder Stateful Session Beans müssen wie beschrieben repliziert werden, um einen verteilten Failover-Cache aufrechtzuerhalten. Hierzu existieren zwei grundlegende Abgleichsmöglichkeiten, In-Memory Replication oder Database Replication. Bei der ersten Variante gibt es zwei Möglichkeiten: Entweder schreiben alle Knoten ihre Session-Informationen auf einen zentralisierten Server oder jeder Cluster-Knoten besitzt einen Backup-Knoten, mit dem er die Session-Informationen permanent austauscht, indem die enthaltenen Objekte serialisiert und über das Netzwerk versendet werden. Bei der Database Replication schreibt jeder Knoten seine Session-Informationen in die zentrale Datenbank.

Clustering der J2EE-Basisdienste

Der dritte Bereich neben dem Clustering der J2EE-Komponenten ist das Clustering der J2EE-Basisdienste. Auch hier existieren verschiedenste proprietäre Realisierungsansätze oder teilweise wird das Clustering der Dienste vom Applikationsserver nicht unterstützt. Ein solcher nicht unterstützter Dienst besitzt kein Backup und stellt damit einen SPOF dar, der einen Ausfall des gesamten Clusters bzw. Teile der Anwendung verursachen kann. Dies ist oft bei JMS der Fall, wo z.B. JMS Topics immer nur auf einer Instanz existieren können und damit einen SPOF darstellen. Wenn diese Instanz ausfällt und die Applikation auf JMS Topics fußt, ist ein Ausfall der Anwendung garantiert, bis dieses Topic zum Beispiel durch ein manuelles Cold Backup auf einen anderen Server migriert wurde.

Der wichtigste Dienst, welcher zu clustern ist, ist sicher JNDI, da dieser als zentraler Einstiegspunkt für jeden Nicht-Web-Client dient. Auch hier existiert eine Reihe von Möglichkeiten:

- Independent JNDI Tree: Bei dieser Variante besitzt jeder Applikationsserver einen von allen anderen unabhängigen Naming Server. Die Vorteile bestehen in der Einfachheit einer sehr geringen Cluster-Konvergenz (Zeit, die vom Cluster benötigt wird, um alle Maschinen und ihre assoziierten Objekte im Cluster zu erkennen) sowie einer einfacheren Skalierung. Meist wird hierbei aber kein Failover unterstützt, da der geladene Remote-Proxy hier serverbezogen ist.
- Centralized JNDI Tree: Hierbei wird parallel zu den eigentlichen Business-Servern eine entsprechende Anzahl von Naming Servern aufgebaut, die den JNDI Tree verwalten. Dieser Ansatz ist bei großen Clustern nicht zu empfehlen, da hier die Anzahl der JNDI Bindings zu groß wird.
- Shared Global JNDI Tree: Bei diesem gängigen Ansatz besitzt jeder Server einen eigenen Naming Server, wobei dieser seine Informationen (JNDI Tree) aber bei jeder Änderung mit den anderen Servern abgleicht, d.h., jeder hat die gleichen Informationen. Der Vorteil liegt in der Einfachheit durch die Gleichwertigkeit der Server, einer einfachen Skalierung sowie relativ simpler Failover-Logik, der Nachteil in der hohen Konvergenzzeit aufgrund des Abgleichs sowie dem Aufwand für den permanenten Abgleich.

J2EE Cluster-Topologien

Da jetzt sowohl die allgemeinen Konzepte als auch deren Umsetzung im J2EE-Bereich vorgestellt sind, stellt sich die Frage: Wie ordne ich meine Komponenten innerhalb der Unternehmens-IT-Infrastruktur an, um den Mehrwert eines Clusters zu nutzen, aber gleichzeitig die vorgegebenen IT-Richtlinien einzuhalten? Hierfür existiert eine Reihe von Standard-Cluster-Topologien mit den unterschiedlichsten Bezeichnungen für unterschiedlichste Einsatzgebiete. An dieser Stelle sollen für eine J2EE-Standard-Applikation, also Web-Client, Servlets/JSPs, EJBs und RDBMS, die wichtigsten drei Cluster-Topologien herausgegriffen, benannt und dargestellt werden (Abb. 8–10):

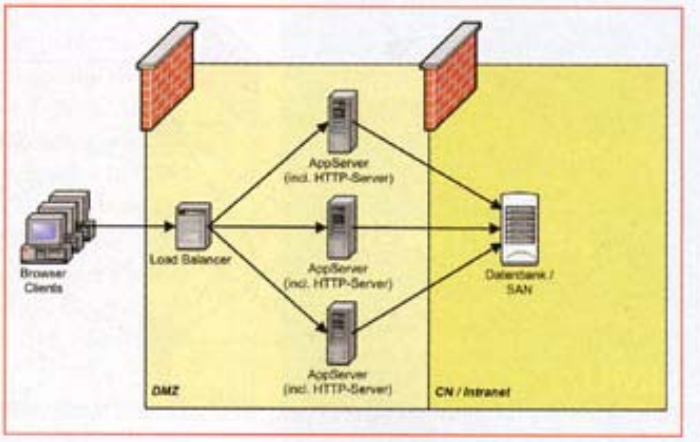
- 3-Tier Cluster-Topologie
- 4-Tier Cluster-Topologie
- 5-Tier Cluster-Topologie

Ein vom Benutzer initiierter Request (z.B. Abfrage von Artikeldaten) durchläuft im Allgemeinen folgende Tiers: Browser, Web/HTTP-Server, Web-Container, EJB-Container und Datenbank. Nun existieren verschiedene Möglichkeiten, diese obligatorischen Tiers anzuordnen, da im Produkt Applikationsserver sowohl Web/HTTP-Server als auch Web-Container und EJB-Container enthalten sind, sodass im minimalistischen Ansatz (3-Tier Cluster-Topologie) der Request lediglich Browser,

Applikationsserver und Datenbank durchläuft. Andererseits ist es, wie in der 5-Tier Cluster-Topologie zu sehen, möglich, diese Bestandteile einzeln und remote entkoppelt zu verwenden.

Wann also sollte man welche Topologie verwenden und was sind die jeweiligen Vor- und Nachteile? Generell gilt, dass Performance, Administrierbarkeit und Zuverlässigkeit umso besser sind, je weniger Einzelkomponenten und damit Netzwerkverbindungen existieren. Das heißt, hier ist die 3-Tier Cluster-Topologie, welche drei Tiers kombiniert, klar im Vorteil. Diese Topologie genügt den Anforderungen vieler Applikationen, da der Großteil der Lastverteilung zwischen Clients und Cluster stattfindet. Allerdings bestehen hier natürlich entsprechende Nachteile: Zum einen sind bei dieser Topologie die geschäftskritischen EJBs für die Außenwelt sichtbar, da sie sich in der DMZ (demilitarisierte Zone) befinden, und zum anderen betrifft ein Ausfall in einer Tier auch die beiden anderen Tiers, was in den anderen Topologien nicht der Fall ist. So können bei den 4-Tier bzw. 5-Tier Cluster-Topologien zum Beispiel bei Problemen im EJB-Container die statischen HTML-Seiten, welche auf einem entkoppelten Web/HTTP-Server liegen, weiterhin erreicht werden. Es bietet sich insbesondere bei Portalen, bei denen viel statischer Content vorliegt, eine derartige Trennung an. Hier kann natürlich auch separat skaliert werden, d.h., man verwendet zum Beispiel acht Web/HTTP-Server und drei Applikationsserver. Zum anderen besteht in großen Projekten bzw. Installationen meist eine Trennung der Verantwortungen (Separation of Concerns) bzgl. der Tiers, d.h., es existieren meist eine Web-Truppe für den statischen Content bzw. die Administration der Web/HTTP-Server und eine Appserver-Truppe. Einer der wichtigsten Punkte für die Trennung der Web/HTTP-Server und der Applikationsserver ist sicher auch das Ziehen der Grenze zwischen DMZ und dem CN (Corporate Network), die meist zwischen statischem Inhalt und eigentlicher Geschäftslogik gezogen wird. Eine Besonderheit der 5-Tier Cluster-Topologie ist die Trennung zwischen dem Web-Container und dem EJB-Container. Diese Trennung ermöglicht zusätzlich zum HTTP Load-Balancing eine Lastverteilung bzgl.

Abb. 7: 3-Tier Cluster-Topologie



der EJB-Methodenaufrufe, was in manchen Szenarien bei denen die Web-Container stark von der EJB-Container-Last abweichen, sinnvoll sein kann.

Abschließend sollen noch einige ergänzende Punkte behandelt werden, um das Thema Clustering abzurunden.

Kommunikation im Cluster

Wie angeführt, existiert eine Reihe von Kommunikationswegen zum Cluster und innerhalb des Clusters. Für den Business-Betrieb werden ausgehend von der Cluster Entry Component, welche alle Requests empfängt, die Requests an die einzelnen Server gesendet. Weiterhin kommunizieren die Cluster-Knoten auch untereinander. So werden zum Beispiel Heartbeat-Nachrichten ausgetauscht, um Serverausfälle zu registrieren, oder bei Verwendung des Shared Global JNDI Clustering wird der JNDI Tree bei jeder Änderung an alle Server übertragen. Für eine solche One-to-Many-Kommunikation wird in der Regel IP Multicast verwendet, wohingegen für Peer-to-Peer-Kommunikation IP Sockets verwendet werden. Eine Peer-to-Peer-Kommunikation tritt zum Beispiel bei In-Memory Replication mit einem Backup-Knoten für HttpSession oder SFSB Replication auf.

Collocation

Die heutigen Applikationsserver beherrschen im Cluster-Betrieb Performanceoptimierungen, die als Collocation-Mechanismen bezeichnet werden. Ein Beispiel ist die Transactional Collocation, bei der das benötigte Objekt von der Serverinstanz geladen wird, auf der die Transaktion gestartet wurde, wodurch die Netzwerklast reduziert und die Transaktionssteuerung vereinfacht werden. Ein anderes Beispiel ist die Object Collocation, bei der das Objekt von der aktuellen Serverinstanz geladen wird, um auch hier die Netzwerklast zu minimieren. Diese Optimierungsmechanismen sollte man kennen, da sich sonst beim Testen eines fertig eingerichteten Clusters schnell enorme Frustgefühle einstellen können, wenn beim Load Balancing der Request nicht wie erwartet auf eine andere Serverinstanz umgeleitet wird.

HTTPS und Clustering

Bei Verwendung von HTTPS als Eingangskommunikation kommt es zu Problemen mit der Business Component Affinity sowie dem Failover, da die Cluster Entry Component, d.h. zum Beispiel der Hardware Load Balancer, die Session-Informationen in der URL oder den Cookies aufgrund der Verschlüsselung nicht auslesen und daher kein Routing des Requests auf den entsprechenden Server durchführen kann. Um dieses Problem zu lösen, muss der Request noch vor oder in der Cluster Entry Component entschlüsselt werden, wobei hier bei der Planung besonders auf die dadurch entstehende weitere CPU-Belastung der Cluster Entry Component geachtet werden muss. Auch hier existiert wieder die Wahl zwischen einem softwarebasiertem Ansatz (Web Server Proxies) und einem hardwarebasiertem Ansatz (Hardware SSL Decoder).

Was muss bei der Programmierung beachtet werden?

Leider ist das Clustering bezüglich der Programmierung nicht transparent, d.h., es existieren bestimmte Regeln, die ein Entwickler beachten muss, sodass die ent-

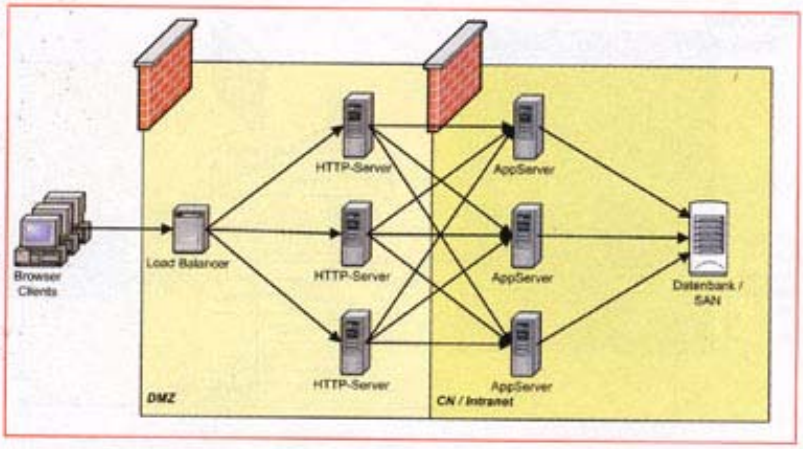


Abb. 8: 4-Tier Cluster-Topologie

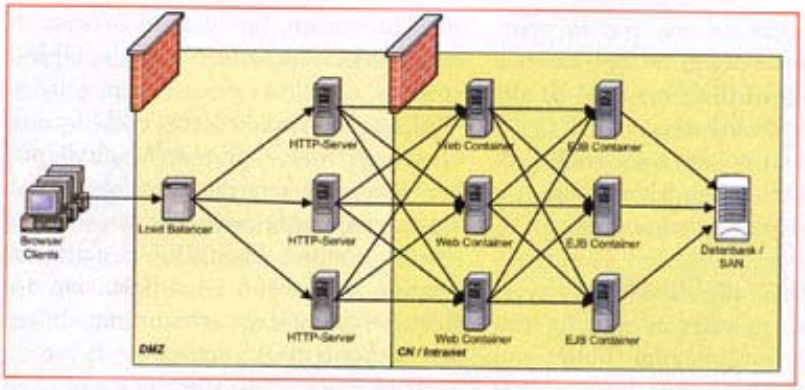


Abb. 9: 5-Tier Cluster-Topologie

stehende Applikation clusterfähig ist. Beispiele hierfür sind:

- Alle in einer HttpSession abgelegten Daten müssen serializable sein.
- Die *setAttribute()*-Methode der HttpSession muss immer aufgerufen werden, wenn sich die in der HttpSession abgelegten Daten geändert haben, erst dann erfolgt die Replikation.
- In den serverspezifischen Deployment-Deskriptoren existieren diverse Parameter zur Definition von clusterbezogenen Eigenschaften einer Deployment-Einheit.

Sizing and Planning

Die Planung eines Produktiv-Clusters ist komplex und muss entsprechend überdacht werden, da Fehler an dieser Stelle meist direkte finanzielle Auswirkungen besitzen. In vielen Fällen reichen zwei leistungsstarke Servermaschinen, die hauptsächlich dem Ziel der Hochverfügbarkeit dienen. Erst wenn die Last entsprechend groß ist oder durch einen User-Anstieg nach einer Werbekampagne größer werden kann, sollte der Cluster auf mehrere Maschinen ausgeweitet werden. Hierbei muss also immer zwischen Durchschnitts- und Spitzenbelastung unterschieden werden. Bezüglich der CPU-Auslastung der Server existieren Heuristiken, die als 70/90 Percent CPU Utilization Rules bekannt sind. Die 70-Prozent-Regel besagt, dass die CPU-Auslastung im Normalbetrieb 70 Prozent nicht überschreiten und spätestens hier eine horizontale bzw. vertikale Skalierung vorgenommen werden sollte. 90 Prozent zeigen einen kritischen Zustand auf, bei dem das System in Kürze die Dienstfähigkeit verlieren kann. In einem Cluster sollte also die jeweilige Auslastung der Server maximal 70 Prozent betragen. Betrachtet man die Kapazitätsplanung in Kombination mit Failover-Szenarien wie in Abbildung 10, wird die angesprochene Komplexität deutlich: Besteht das Cluster aus drei gleichwertigen Servern, die jeweils eine CPU-Auslastung von 75 Prozent haben, und ein Server fällt aus, laufen die beiden verbleibenden Systeme auf jeweils 112,5 Prozent, wodurch es zunächst zu einem Thrashing der Systeme und später ggf. zum kaskadierten Ausfall kommt.

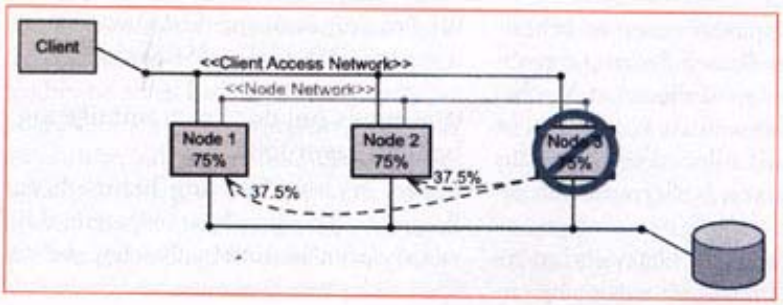


Abb. 10: Kapazitätsplanung und Failover-Effekte (vgl. [3])

Ein weiterer wichtiger Punkt, der an dieser Stelle nicht fehlen soll, ist das Problem der Machine Equivalence, d.h. das Verhältnis zwischen der Gesamtleistung des Clusters zur kumulierten Einzelleistung der Maschinen. Dieses Verhältnis, welches durch Abstimmungs- und Verteilungsaufwand entsteht, ist logarithmisch (Abb.11). Man sieht, dass zwei Maschinen in einem Cluster so viel leisten wie 1,6 Einzelmaschinen und erst drei Maschinen die doppelte Kapazität bieten.

Was muss bei der Adressierung der nichtfunktionalen Anforderungen beachtet werden? Ein Beispiel soll dies verdeutlichen: Meist fordert der Kunde eine 24x7-Verfügbarkeit des Systems. Nun kommt der Architekt ins Spiel, der u.a. die folgenden unangenehmen und konkreten Fragen stellen muss:

- Wie soll die prozentuale Verfügbarkeit aussehen? Da eine hundertprozentige Verfügbarkeit (Uptime) nicht gewährleistet werden kann, muss die 24x7-Forderung prozentual hinterlegt werden, d.h., zu wie viel Prozent soll das System verfügbar sein? 95 Prozent, 99 Prozent, 99,9 Prozent oder 99,99 Prozent, was immer noch eine Downtime von ca. zwei Stunden pro Jahr bedeutet, die natürlich auch am Stück auftreten kann. Übrigens gehören zur Berechnung der High Availability (z.B. die oft geforderten Three Nines = 99,9 Prozent; d.h. ca. neun Stunden Ausfall pro Jahr) die Wartungsarbeiten nicht dazu, eine Skalierung des laufenden Systems aber schon.

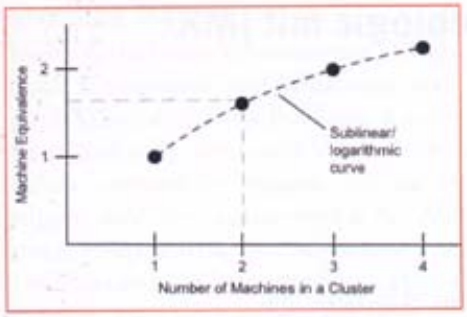


Abb. 11: Machine Equivalence (vgl. [3])

- Welche Funktionalitäten müssen hochverfügbar sein? Meist müssen Funktionen, die direkte Kundentransaktionen durchführen, hochverfügbar sein, aber zum Beispiel interne Statistik- und Recherchefunktionen auf den Daten des gleichen Systems nicht.
- Wie verfügbar ist die Systemumgebung? Die prozentuale Verfügbarkeit eines Systems kann nur gewährleistet werden, wenn alle Systeme, von denen das zu bewertende System abhängig ist, mindestens auch diese Verfügbarkeit gewährleisten. Dies entspricht dem kritischen Pfad in der Projektplanung. So macht es keinen Sinn, von einem System eine Verfügbarkeit von 99,9 Prozent zu verlangen, wenn das dieses System umgebende Netzwerk lediglich eine Verfügbarkeit von 95 Prozent zusichert. Bei entsprechend hohen Anforderungen an die Verfügbarkeit müssen neben dem Applikationsserver auch alle anderen Infrastrukturkomponenten ebenfalls mindestens zweifach vorgehalten werden.

An diesem Beispiel sieht man, dass die korrekte Analyse der nichtfunktionalen Anforderungen ein aufwändiger Prozess ist, bei dem jede einzelne Systemqualität in Verbindung mit verschiedenen Parametern (z.B. Zeit, Funktion, Komponente, Benutzer/Rolle) betrachtet werden muss. Zu bedenken ist hierbei aber, dass 80 Prozent aller gescheiterten IT-Projekte an der Nichterfüllung der nichtfunktionalen Anforderungen scheitern! Der Aufwand sollte sich also insbesondere bei großen Projekten lohnen.

Fazit

Unter einem Cluster versteht man eine Gruppe von Komponenten, welche auf Basis von Redundanz und Lastverteilung zusammenarbeiten, um eine performantere und höher verfügbare Lösung als eine einzelne Komponente zu bieten, wobei diese Gruppe sich nach außen hin wie eine einzelne Komponente präsentiert. Das heißt, Clustering wirkt sich positiv auf die nichtfunktionalen Anforderungen der Skalierbarkeit, Ausfallsicherheit und Performance aus und sollte deshalb bei entsprechenden Anforderungen das Mittel der Wahl sein.

Natürlich bringen Clustering wie auch jeder andere Mechanismus Nachteile mit sich: Es entsteht ein höherer Aufwand zur Planung und Einrichtung des Systems, es müssen bei der Programmierung der Komponenten Regeln eingehalten werden, damit diese clusterfähig sind, der Aufwand bezüglich der Wartung ist höher und auch das Management eines solchen Systems ist in den meisten Fällen aufwändiger. Weiterhin gestaltet sich die Anbindung von Fremdsystemen schwieriger, da statt einer 1:1-Beziehung zwischen System und Fremdsystem jeweils eine 1:n- bzw. bei einem geclusterten Fremdsystem eine m:n-Beziehung besteht.

Es handelt sich bei diesem Thema um ein sehr umfassendes Thema, vom dem dieser Artikel lediglich die Spitze des Eisbergs in Form eines groben Überblicks darstellen kann. Es lohnt sich aber auf jeden Fall, sich tiefer in diese interessante Materie (siehe „Replikation mit Lamport-Uhren", „Replikationsprotokolle", „Problem der byzantinischen Generäle" etc.) einzuarbeiten. ■

Links & Literatur

[1] BEA: WebLogic Server – Using WebLogic Server Clusters: edocs.bea.com/wls/

[2] IBM: WebSphere Scalability: www.ibm.com/redbooks/

[3] Sun: Architecting and Designing J2EE Application. SL425-Course

[4] Oracle: High Availability & Disaster Recovery – 2002: www.oracle.com/

[5] JCP: JSR 117 – J2EE APIs for Continous Availability: www.jcp.org/

[6] Sacha Labourey: Load Balancing and Failover in the JBoss Application Server: www.clustercomputing.org/

[7] Tom McDonough: Object Level Fault Tolerance for CORBA-based Distributed Computing: dantanner.tripod.com/

[8] Thomas Mattern: Clustering in J2EE-Umgebungen: www.eai-competence-center.de/