Wege zum performanten App-Server

Um die Leistungsfähigkeit eines Applikations-Servers sicherzustellen, reicht ein sauberes Softwaredesign allein nicht aus. Auch die Rahmenbedingungen des Betriebs müssen vorab definiert und getestet werden.

VON WILHELM KUHN*

Die Informationsverarbeitung in Unternehmen wird immer stärker von mehrschichtigen Anwendungen bestimmt. Im Trend sind Applikationen und Web-Services, in deren Mittelpunkt Application Server stehen. Diese kommunizieren mit den IT-Ressourcen des Unternehmens, seien es Datenbanken oder Backend-Gateways zu Host-Systemen. Sie beherbergen selbst Komponenten mit Fachfunktionen, und sie bereiten Daten auf, die am Browser oder auf anderen Endgeräten angezeigt werden.

Latenz und Durchsatz

In der Anfangszeit dieser Entwicklung wurde der Begriff Application Server oft per se als Garant für Performanz angesehen – ein Fehler, wie sich in vielen Projekten schmerzhaft herausgestellt hat. Performanz auf der Basis von Application-Server-Techniken ist nicht selbstverständlich, sondern das Ergebnis einer Vielzahl von Faktoren.

Der Begriff Performanz hat mindestens zwei Aspekte: Latenz und Durchsatz. Bei der Definition von Performanzzielen sollten diese Begriffe getrennt betrachtet werden. Durchsatz bedeutet zum Beispiel die Anzahl der Transaktionen je Zeiteinheit. Latenz ist diejenige Zeit, bis eine aufgerufene Funktion antwortet, also nach einer Benutzeraktion ein Ergebnis sichtbar wird.

Unter welchen Bedingungen eine Anwendung arbeiten soll, ist aus Performanzsicht ebenso wichtig wie die fachliche Definition der Anforderungen. Die Randbedingungen bestimmen nicht nur die Dimensionierung der Server, sondern im Vorfeld der Entwicklung bereits die System- und die Softwarearchitektur. Deshalb sollte unbedingt am Anfang eines Projekts, noch vor Entwicklungsbeginn, detailliert geklärt werden, wie die nicht-funktionalen Anforderungen aussehen.

Einer der Schlüssel zu performanten Anwendungen liegt in der Softwarearchitektur. Fehler in der Architektur und im Design der Anwendung sind schwer rückgängig zu machen. Sie können nur durch Tuning nicht ausgeglichen werden und bedingen eine bereits frühzeitige und teure Skalierung der Systeme.

Hier lesen Sie …

- wie die Architektur und das Design von mehrschichtigen Anwendungen zur Performanz von Applikations-Servern beitragen;
- wann und wie man die Performanz testen sollte;
- wie mehrschichtige Anwendungen getunt und skaliert werden.

Was Performanz beeinflusst

Eine Vielzahl von Faktoren bestimmt die Performanz von Application-Server-Anwendungen – hier eine Auswahl.

Die erste Maxime für die Performanz mehrschichtiger Anwendungen lautet, die Kommunikation zwischen den Schichten einzuschränken. Application-Server-Anwendungen bestehen meist aus drei oder mehr Infrastrukturschichten: Client, Web-Server, Web-Container, EJB-Container, Datenbank etc. Die Kommunikation der Anwendungsteile über die verschiedenen Schichten hinweg ist mit einem beträchtlichen Overhead verbunden. Hier hat es sich bewährt, in der Architektur spezielle Entwurfsmuster (Design Patterns) für mehrschichtige Anwendungen zu berücksichtigen, wie sie etwa von Sun Microsystems als „J2EE Core Patterns" veröffentlicht wurden.

Connection-Pools nutzen

Ein weiteres Erfolgsgeheimnis performanter Softwarearchitekturen besteht darin, die vom Server beziehungsweise der Datenbank angebotenen Caching- und Pooling-Mechanismen zu nutzen. Dies betrifft zunächst die Nutzung von Connection-Pools für Datenbankverbindungen. Der Auf- und Abbau von Verbindungen bei jedem Datenbankzugriff ist zeitaufwändig und erhöht die Latenz. Ungleich effizienter ist es, die Connection-Pools der Application Server zu nutzen. Hierbei öffnet der Container eine Reihe von Sitzungen, hält sie offen und verteilt sie an die Anwendungen, die sie anfordern.

Checkliste Performanztest

- Gibt es ein Konzept für die Infrastruktur und den Ablauf der Tests?
- Wurde ein Verantwortlicher für Ablauf und Überwachung der Performanztests benannt?
- Wurden neben den Tests auch die Vorbereitung, Auswertung und das Code-Redesign in der Aufwands- und Terminplanung berücksichtigt?
- Wurde die Bereitstellung von Testdaten und -accounts in der Aufwands- und Terminplanung berücksichtigt?

Ist die jeweilige Arbeit beendet, wird die Verbindung wieder an den Pool zurückgegeben.

Deutliche Performanzgewinne lassen sich auch durch die Nutzung von „Prepared SQL Statements" erzielen. Dabei werden den SQL-Anweisungen in der Datenbank oder im Server zwischengespeichert und müssen nicht ständig neu analysiert werden.

Application Server sind kein Garant für Performanz.

Bei weiteren Aufrufen der gleichen Anweisung werden lediglich die geänderten Abfrageparameter eingesetzt, so dass Folgeaufrufe wesentlich schneller bearbeitet werden.

Bei Server-Anwendungen verdient der ökonomische Umgang mit dem Arbeitsspeicher besondere Beachtung. Moderne Programmiersprachen wie Java bieten eine automatisierte Speicherverwaltung (Garbage Collection). Dies bedeutet jedoch nicht, dass die Entwickler den Punkt Speicher-Management vernachlässigen dürfen. Ihre Aufgabe ist es, nicht mehr benötigte Speicherstrukturen (Objekte) so früh wie möglich freizugeben. Wer dies versäumt, provoziert unzuverlässige, schlecht skalierende Anwendungen. So erschweren Objekte, die nicht freigegeben wurden, ein gleichmäßiges Arbeiten des Garbage Collectors. Es kann in unregelmäßigen Abständen zum Einfrieren der Anwendung kommen. Im schlimmsten Fall führen nicht freigegebene Objekte zu Speicherlöchern (Memory Leaks): Der Arbeitsspeicher wird blockiert, und die Anwendung steht bis zum Neustart des Servers nicht mehr zur Verfügung.

Javascript nur wenn nötig

Bei einer End-to-End-Betrachtung der Performanz sind nicht nur die Server bestimmend. Die Antwortzeit beziehungsweise Latenz kann durchaus auch von den Client-Systemen beeinträchtigt werden. Immer wieder erliegen Entwickler der Verführungskraft von Javascript: Diese Skriptsprache (nicht mit der Programmiersprache Java zu verwechseln!) ermöglicht es, Code im Internet-Browser auszuführen und Web-Anwendungen mit Bedienelementen auszustatten, die HTML nicht bietet. Der Nachteil: Der Code wird im Browser interpretiert, die Verarbeitung ist daher im Vergleich zu der Verarbeitung von Java-Code im Application Server sehr ineffizient und kann die Antwortzeiten beträchtlich erhöhen. Deshalb sollten Anwendungsfunktionen soweit wie möglich auf den Application Server verlagert und die Seiteninhalte über HTML im Server erzeugt werden. Javascript sollte nur dort eingesetzt werden, wo es für Plausibilitätsprüfungen oder besondere grafische Bedienelemente unumgänglich ist.

Vertrauen gut, Kontrolle besser

Performance-Management ist klassisches Risiko-Management. Wie lassen sich dabei kritische Komponenten frühzeitig erkennen, wie kann man Fehlentwicklungen vorbeugen? Die Antwort sind Performanz- und Lasttests. Sie sollen Messwerte produzieren, anhand derer sich die begrenzenden Faktoren identifizieren lassen. Performanz- und Lasttests sollten ohnehin ein fixer Bestandteil des Vorgehensmodells für die Softwareentwicklung sein.

Die frühe Identifikation kritischer Komponenten ist besonders wichtig, weil
– späte Tests ein enormes Risiko für den Fertigstellungstermin und die Produktqualität bedeuten – im schlimmsten Fall müssen grundlegende Architekturänderungen geplant und implementiert werden; wenn daraufhin eine bereits erfolgte funktionale Abnahme hinfällig ist und viele Tests zu wiederholen sind, kann ein Projekt zu einem späten Zeitpunkt um mehrere Phasen zurückfallen;
– die Behebung architekturbedingter Schwächen in frühen Entwicklungsphasen deutlich kostengünstiger ist als später;
– früh Anhaltspunkte zur präzisen Definition einer produktionsnahen Testumgebung gewonnen werden.

Design-Fehler sind nur sehr schwer rückgängig zu machen.

Wie früh ist früh? Die Messungen sollten auf keinen Fall erst dann beginnen, wenn die Anwendung als Alpha-Version vorliegt. Für kritische Anwendungsteile gilt: Anhand von explorativen Prototypen ist die Performanz bereits dann zu überprüfen, wenn die grundlegende Anwendungsarchitektur steht. Explorative Prototypen dienen der Beantwortung ganz bestimmter Fragen. Sie sollten nicht selbst zum Endprodukt weiterentwickelt werden, sondern den Softwareentwurf verbessern helfen oder Hinweise für ein Redesign liefern.

Agile Softwareentwicklung verzichtet darauf, Anwendungen durchgehend in der vollen Funktionsbreite zu entwickeln. Stattdessen werden sukzessive kleine, aber voll funktionsfähige Ausschnitte der Gesamtfunktionalität implementiert. Bedingt durch die kurzen Iterationszyklen, kann hier oft auf Prototypen verzichtet werden – unter der Bedingung, dass jede Iteration durch Performanztests begleitet und überwacht wird.

Im Lauf der Entwicklung – gleich ob agil oder klassisch – birgt jede neue oder überarbeitete Anwendungsfunktion (auch Bugfixes!) die Gefahr, dass die Performanz leidet. Daher sollten Performanztests auch bei anfänglich guten Ergebnissen über den Projektverlauf hinweg wiederholt werden.

Dazu bietet es sich an, eine Datenbank für die Testresultate anzulegen, in der die Ergebnisse der einzelnen Testdurchläufe automatisiert abgelegt werden. So können zum Beispiel die Einhaltung der Sollwerte automatisch überwacht und bei einer Überschreitung die Projektleitung und die für das jeweilige Modul zuständigen Entwickler informiert werden. Damit auch eine Reaktion auf diese Signale erfolgt, sollte man einen für das Performance-Management explizit Verantwortlichen benennen.

Kommunikation reduzieren

- **Session Facade:** Reduktion von Aufrufen über Schichtgrenzen hinweg durch Zusammenfassung zu Aufruf-Bündeln;
- **Value Objects:** Bündelung von Datenstrukturen zum gemeinsamen Transport über Schichtgrenzen;
- **Caching:** Zwischenspeicherung von Daten zur Mehrfachverwendung innerhalb eines Programms;
- **Page Iterator:** Aufteilung großer Ergebnismengen in Einzelportionen, die erst bei Bedarf abgerufen werden (ein Beispiel sind die Google-Seiten).

Testen: Vorgehen und Verlauf

Performanz- und Lasttests werden üblicherweise automatisiert betrieben. Dazu dienen spezielle Roboter- oder Treiberanwendungen, die auf den Test-Client-Rechnern installiert werden. Unter *http://www.testingfaqs.org* findet sich ein Überblick über viele verfügbare Produkte. Sowohl die kommerziellen als auch die Open-Source-Lösungen sind oft sehr leistungsfähig, aber auch komplex. Daher erfordert ihre erfolgreiche Nutzung Training und Einarbeitung, die im Projektplan berücksichtigt sein sollten. Ebenfalls einzuplanen sind Zeit und Aufwand zur Abbildung der Testfälle auf das jeweilige Testtreibersystem, das heißt zur Anfertigung der Testskripte.

Die Test-Clients selbst können an ihre Kapazitätsgrenze gelangen, wenn sie parallel und in schneller Wiederholung Anfragen an die Server-Systeme stellen – die Testergebnisse werden dann verfälscht. Wird dies bei Last- oder Stresstests befürchtet, sollten die Tests auf mehrere Client-Rechner verteilt werden. Dabei sollte der eingesetzte Treiber in der Lage sein, Test-Cluster mit mehreren Rechnern zu koordinieren.

Einzeltests empfohlen

Es hat sich als sinnvoll erwiesen, Last- und Stresstests zunächst durch Einzeltests vorzubereiten, mit denen die Reaktion des Systems auf vereinzelte Anfragen untersucht wird. Einzeltests lassen zwar nur begrenzte Aussagen über die Gesamtperformanz zu, liefern aber dennoch wertvolle Kennzahlen, die für die Dimensionierung der Server für Last- und Stresstests wichtig sind. Beispiele sind der je Transaktion beanspruchte Arbeitsspeicher, die CPU-Auslastung oder die zu übertragenden Datenmengen je Transaktion. Darüber hinaus erlauben Einzeltests die Erstellung von Zeitscheibenanalysen. Diese beschreiben den Anteil einzelner Systemteile und -schichten an der Gesamtantwortzeit.

Wichtig für die Aussagekraft der Tests ist, dass die Testumgebung der Anwendung exklusiv zur Verfügung steht. Besonders bei der Netzwerkinfrastruktur oder der Einbindung von Backends kann dies jedoch nicht immer garantiert werden. Befürchtet man dadurch eine Verzerrung der Ergebnisse, sollten mehrere Testreihen gefahren und statistische Werte gebildet werden.

Realistische Testergebnisse setzen zudem voraus, dass die Lasttreiber auf ausreichend großen Datenbeständen operieren. Bei zu kleinen Datenmengen besteht zum Beispiel die Gefahr, dass Caching-Effekte das Ergebnis verfälschen. Außerdem sollte nach Testläufen die Neu-Initialisierung der Daten nicht vergessen werden. Testdaten, die bei vorhergehenden Durchgängen verändert oder „verbraucht“ wurden, können sonst zu ungewollten Ergebnissen führen. Bei der Projektplanung sollte man deshalb daran denken, dass das Management der Testdaten einigen Aufwand erfordert.

Nichtfunktionale Anforderungen bestimmen

- Für welche Anwendungsfälle ist die Performanz besonders kritisch?
- Gibt es Spitzenzeiten (Monate, Wochentage, Tageszeiten), in denen die Anwendung besonders frequentiert ist?
- Wie ist Performanz zu präzisieren (Latenz, Durchsatz)? Mit wie vielen gleichzeitigen Sitzungen, Anfragen und Transaktionen ist in Normal- und Spitzenzeiten zu rechnen?
- Auf welchen Datenbeständen und -mengen soll eine Anwendung operieren?
- Welche Datenmengen sind zwischen den einzelnen Schichten der Anwendung zu übertragen?
- Mit welchen Wachstumsraten ist zu rechnen?
- Gibt es Infrastrukturkomponenten, die sich nicht ohne weiteres skalieren lassen?

Der Betrieb vieler Anwendungen ist mit einer gewissen Grundlast verbunden. Daher empfiehlt es sich, auch bei „einfachen“ Performanztests nach und nach mehrere Sitzungen zu simulieren und die Ergebnisse grafisch aufzutragen. Dabei soll ermittelt werden, ob der Ressourcenverbrauch linear mit der Anzahl der Sitzungen steigt oder ob er erst nach einer bestimmten Anzahl von Sitzungen einem bestimmten Trend folgend wächst.

Agile Entwicklung erfordert einen Test für jede Iteration.

Im Anschluss an die Einzeltests wird versucht, durch Hochrechnung der Messergebnisse Anhaltspunkte für die Dimensionierung der Lasttest- und Produktionssysteme abzuleiten. Dies soll zum Beispiel folgende Fragen beantworten:

– Wie sollten die Systeme für produktionsnahe Tests bezüglich Arbeitsspeicher, CPU oder Bandbreiten dimensioniert werden?

– Welche Komponente muss zuerst skaliert werden? Für welche Systemkomponente wird, bezogen auf gleichzeitige Sitzungen oder Benutzeraktionen, zuerst die Kapazitätsgrenze erreicht?

– Reichen für die zu erwartenden Anfragen die Netzwerkbandbreiten aus?

Schließlich sollen Last- und Stresstests zeigen, wie sich eine Anwendung unter produktionsnahen Bedingungen verhält: Wie lange sind die Antwortzeiten unter realistischen Bedingungen, was sind unter Last die begrenzenden Faktoren, ab welcher Nutzungsfrequenz muss die Anwendung skaliert werden?

Die Performanztests sollten mindestens für Normallast, Volllast und Überlast erfolgen. Die Dimensionierung der Szenarien sollte aus den eingangs erhobenen nichtfunktionalen Anforderungen ableitbar sein.

Sehr sinnvoll ist es, am Ende eines stufenweise ansteigenden Lasttests noch einmal Niedriglast zu erzeugen. Dabei sollte kontrolliert werden, ob die Antwortzeiten nach einer gewissen Reaktionsphase wieder in den Normalbereich zurückkehren. Ist dies nicht der Fall, könnte das ein Indiz für nicht mehr typische Testdatenbestände oder auch ein Speicherleck sein.

Tuning und Skalierung

Tuning beziehungsweise Feinabstimmung kann nur wenig ausrichten, wenn wichtige Architekturprinzipien verletzt sind. Bei klar konzipierten und sorgfältig entwickelten Anwendungen jedoch lassen sich durch Tuning die jeweiligen Server optimal anpassen. Oft ist es möglich, beträchtliche Leistungsreserven freizusetzen. Einer der wichtigsten Ansatzpunkte Java-basierender Systeme ist die Auswahl und Optimierung der virtuellen Maschine (JVM) des Application Server. Dabei ist sicherzustellen, dass der freie Speicherbereich (Heap) nicht zu eng bemessen, aber auch auf keinen Fall zu groß ist. Weitere Punkte sind die Optimierung des Connection-Pools und des Prepared Statement Caches. Einfache, aber dennoch oft vergessene Maßnahmen sind die Begrenzung des Log-Levels auf ein Mindestmaß oder die Deaktivierung von Beispielanwendungen. Datenbankseitig kann eine geeignete Indizierung der Tabellen Wunder bewirken.

Selbstverständlich sollte sein, jede Tuning-Maßnahme durch Performanztests zu kontrollieren und die Ergebnisse zum späteren Vergleich zu dokumentieren. Eine Anwendung „skaliert“, wenn ihre Antwortzeiten trotz wachsender Zahl von Anfragen und steigender Last gleich bleibt.

Skalieren heißt stabilisieren und nicht beschleunigen.

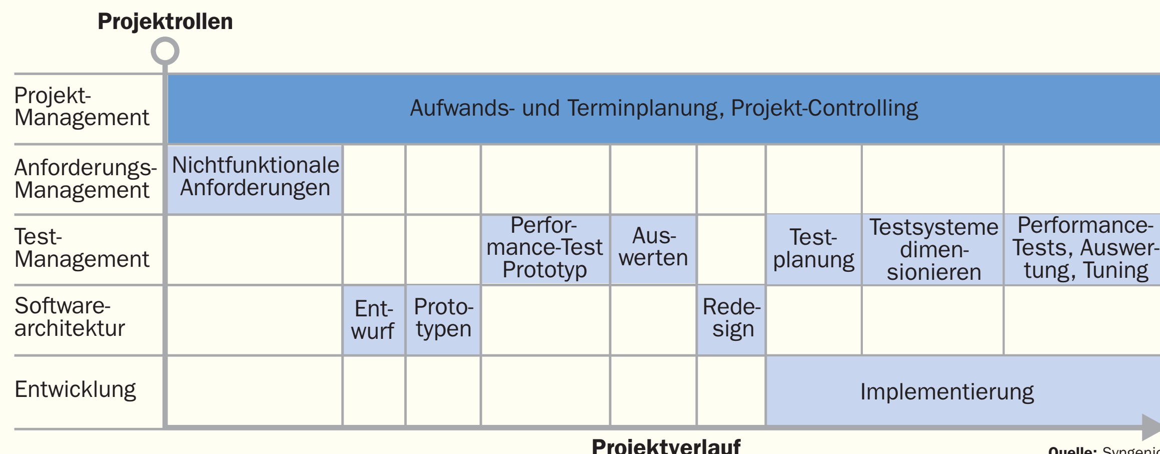
Skalierung ist also keine Maßnahme zur Beschleunigung von Anwendungen, sondern zu ihrer Stabilisierung.

Die einfachste und effizienteste Maßnahme ist die interne Skalierung: Dabei stattet man die Hardware der Server mit zusätzlichen Ressourcen aus, etwa mit Arbeitsspeicher und weiteren CPUs. Diese Art der Skalierung ist für die Anwendung selbst transparent, das heißt nicht sichtbar. Viele Java-Application-Server nutzen bei einem gemeinsamen Betrieb von Web-Containern (Servlet-Engine) und EJB-Containern auf ein- und demselben Hardwareknoten (Co-Location) besonders effiziente Kommunikationsmechanismen, die zum Beispiel Serialisierungen vermeiden. Daher ist eine interne Skalierung wirkungsvoller als eine Verteilung der Schichten auf mehrere Maschinen (externe Skalierung).

Nachdem die Grenzen der internen Skalierung erreicht sind, kommen externe Maßnahmen zum Einsatz. Diese bestehen darin, weitere Hardware hinzuzuziehen und die Anwendung auf mehrere Systeme zu verteilen.

Nach jeder Skalierung ist zu prüfen, ob die Tuning-Einstellungen angepasst werden müssen. Es ist auch sicherzustellen, dass nach der Skalierung die übrigen Ressourcen noch adäquat dimensioniert sind, damit sich nicht neue Engpässe ergeben, weil zum Beispiel nach dem Ausbau der CPU der Arbeitsspeicher oder das Input-Output-System begrenzend wirken. *(ue)*

Das Projekt



Die Performanz von Anwendungen sollte nicht dem Zufall überlassen, sondern durch systematische Rollen-, Aufgaben- und Phasenverteilung im Entwicklungsprozess sichergestellt werden.

***DR. WILHELM KUHN** ist Senior Consultant bei der Syngenio AG in Stuttgart und auf J2EE-Anwendungen sowie Service-orientierte Architekturen spezialisiert. (wilhelm.kuhn@syngenio.de)