„Erfahrungen mit dem Einsatz von Java und CORBA in der Entwicklung einer leistungsstarken Customer Care- Anwendung für die Telekommunikationsindustrie.“

Michael Meadows[1], Jochen Kappel[2]

LHS Verwaltungs GmbH, Abt. Research, Otto-Hahn-Strasse 36,
D-63303 Dreieich-Sprendlingen, Deutschland
[1]mmeadows@de.lhsgroup.com, [2]jkappel@de.lhsgroup.com

Abstrakt. Dieser Beitrag beschreibt die Erfahrungen mit dem Einsatz von Java und CORBA in einer verteilten Komponentenarchitektur zur Entwicklung einer Anwendung zur Kundenverwaltung und Abrechnung für den Telekommunikationsbereich. Es werden verschieden Aspekte des Projektes angesprochen, der Fokus liegt jedoch auf der Performance, da die entwickelte Anwendung für Installationen mit einigen Millionen abzurechnender Kunden und einer hohen Anzahl an Transaktionen ausgelegt wurde. Nach einer kurzen Beschreibung der Anwendungsarchitektur werden die Ergebnisse einer Reihe von Performance-Benchmarks und eine Auslegung dieser Ergebnisse vorgestellt. Die beim Einsatz dieser Technologien für die Performance kritischen Bereiche werden besonders herausgestellt.

1 Einführung

Wie dies bei vielen Herstellern von Standardsoftware der Fall ist, erreicht ein Softwareprodukt in seinem Lebenszyklus einen Punkt, an dem die Grenzen der aktuellen Technologie sichtbar werden. LHS hat daher alternative Technologien und Architekturen untersucht, die eine größere Skalierbarkeit aber auch gleichzeitig flexiblere Anwendungen ermöglichen und dabei den Qualitätsansprüchen an geschäftskritische Anwendungen Rechnung tragen. Java und CORBA wurden als grundlegende Elemente der neuen Technologie ausgewählt. Eine konkrete Implementierung erfolgte am Beispiel einer Anwendung zur Kundenverwaltung.

Dieser Beitrag adressiert vielfältige Aspekte der Anwendungsentwicklung verteilter objektorientierter Systeme. Die Architektur der Anwendung und die eingesetzten Werkzeuge werden kurz vorgestellt, der Fokus liegt aber auf der Performance dieser Technologien in einer realen Einsatzsituation. Die vorgestellte Architektur sowie die Meßergebnisse der Performance-Untersuchungen resultieren aus der Implementierung eines Prototyps. Dieser Prototyp besteht aus einem Anwendungsservers und –client und wurde im Rahmen eines industriellen Entwicklungsprojekts erstellt. Daher erheben die Meßergebnisse nicht den Anspruch einer wissenschaftlichen Untersuchung. Der Fokus dieses Beitrags liegt in der

Analyse der Meßergebnisse und den daraus für die Entwicklung eines Produkts ableitbaren Designvorgaben.

2 Anforderungen an die neue Anwendung

Das Management der LHS stellte umfangreiche Mittel für die Entwicklung einer neuen, auf verteilter Objekttechnologie basierenden Anwendung, zur Verfügung. CORBA wurde als Objektbroker-Middleware und Java als Programmiersprache ausgewählt. Die Richtlinien für den Einsatz dieser Technologien waren klar: die neue Anwendung sollte eine größere Skalierbarkeit haben als die bereits existierende Anwendung und damit auch Installationen unterstützen können, bei denen die Daten von über 10 Millionen Kunden noch zeitgerecht abgerechnet werden können. Die Technologie sollte die Entwicklung einer einerseits äußerst flexiblen Anwendung ermöglichen, die den Anforderungen der Telekommunikationsindustrie gerecht wird und die andererseits den Wartungsaufwand für das Produkt verringert und die Produktivität der Softwareentwickler steigert.

3 Werkzeuge

Für Analyse und Design wurde „Rose for Java„ von Rational eingesetzt. Dieses Werkzeug genügte den Ansprüchen zum Modellieren der Anwendung. Unter großem Aufwand wurden die Möglichkeiten des „round-trip engineering„ untersucht. Man kam jedoch zu dem Schluß, daß die im Test befindliche Version den Qualitätsansprüchen einer industriellen Softwareentwicklung nicht genügt. Als Ergebnis dieser Untersuchungen wurde lediglich das initiale (einmalige) Erzeugen von Programmrahmen vorgeschlagen. Im weiteren Verlauf des Projekts wurde auch dies verworfen und der Einsatz von Rose auf das reine Modellieren beschränkt.

Zur Programmentwicklung wurde „VisualAge for Java„ von IBM in der 1.0 Beta-Version (mit „team-support„) benutzt. Die effiziente Unterstützung fortschrittlicher graphischer Benutzerschnittstellen war bei der Entwicklung eines Anwendungsservers von untergeordneter Bedeutung (die graphische Benutzerschnittstelle der Anwendung wurde getrennt mit einem anderen Werkzeug entwickelt). Die fehlende Unterstützung von „ inner-classes„ in VisualAge war daher nicht von Relevanz. Für unsere Zwecke war das Werkzeug, im Besonderen die Unterstützung des Entwickelns im Team exzellent. In einigen wenigen Fälle, unterschied sich das Verhalten des Servers in VisualAge von dem Verhalten in der virtuellen Maschine von SUN. Diese stellten aber kein Problem für die Anwendung dar.

Zum Abbilden der Geschäftsklassen auf die Tabellen einer relationalen Datenbank wurde „TOPLink for Java„ von The Objectpeople in der Version 1.0 eingesetzt. Die Rolle dieses Werkzeugs und die damit gemachten Erfahrungen werden in einem eigenen Kapitel diskutiert.

4 Die Architektur

Was die Verwendung von CORBA angeht, basiert die Anwendungs-Architektur auf etwas, das man "die momentan gültige Weisheit" nennen könnte [2], [5]. Es wurde versucht, die grundlegenden Schnittstellen zu den Servern einfach zu halten und die Komplexität des Verhaltens in einer Mehr-Schichten-Architektur hinter diesen Schnittstellen zu verbergen. Anstelle einer Beschreibung dieser Architektur mit Hilfe der gängigen Diagramme, die "Stufen" und "Schichten" darstellen und die oft wenig konkrete Aussagekraft besitzen, verwenden wir zur Beschreibung der Architektur vereinfachte Objektdiagramme.

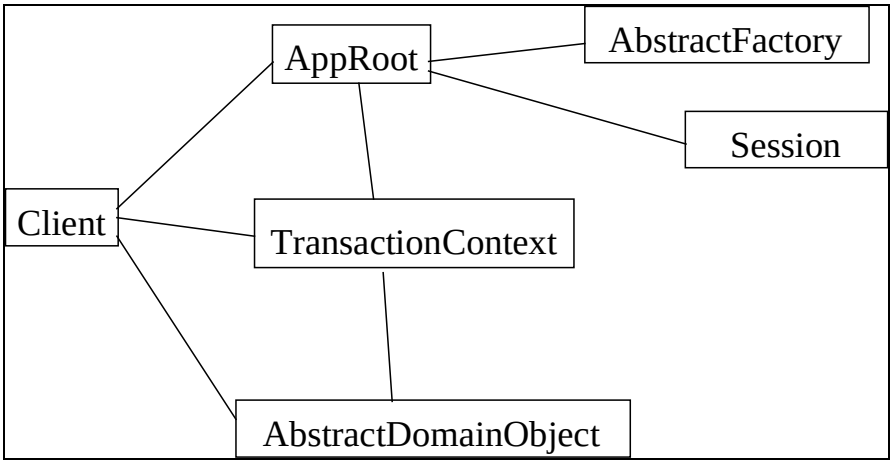

Bild 1. Ein vereinfachtes Klassenmodel der Basisklassen des Anwendungsservers.

Jeder Client besitzt eine Referenz zu einem Server in einer Instanz eines "Application Root" Objektes. Dieses Objekt erhält der Client durch Anbinden an ein „Application Dispenser“ Objekt, welches das einzige benannte Objekt innerhalb eines Servers ist. Das Benennungsschema für diese Dispenserobjekte basiert auf den Verantwortlichkeiten eines Servers, die momentan auf Klassen und nicht auf Instanzen begründet sind (die Verteilung des Verhaltens auf mehrere Server wird auf Basis von Klassen durchgeführt, z.B. Server 1 enthält alle Instanzen der Kundenklasse, Server 2 alle Instanzen der Vertragsklasse etc.). Instanzbasierte Verteilung ist eine mögliche, recht einfache Erweiterung.

Alle primären allgemeinen Einrichtungen eines Servers werden über eine Instanz der Application Root erreicht. Sie enthält Referenzen zu

- einem Exception-"Dictionary", welches die Definition aller Exceptions enthält und das auch für die Definition neuer Exceptions genutzt wird
- Benutzerinformationen, wie etwa eine Beschreibung der Zugriffsrechte innerhalb einer willkürlichen Hierarchie von Benutzern
- einem Meta-Level-"Dictionary", das Meta-Informationen über alle Geschäftsklassen und ihre Attribute enthält
- fundamentalen Schnittstellen für die Handhabung von Transaktionen.

Eine grundlegende Designentscheidung war, den gültigen Status der Geschäftsobjekte in die Datenbank zu legen. Die Persistenzschicht legt Objektinstanzen pro Server im Cache ab, aber alle Clients arbeiten auf ihren eigenen Kopien der Geschäftsobjektinstanzen. Die Geschäftsobjekte besitzen deshalb "Status"- Informationen, der tatsächliche Status der Anwendung liegt jedoch in der Datenbank.

Diese Entscheidung wurde durch zwei wesentliche Faktoren begründet. (1) Die Unterstützung von Transaktionen auf verteilten Objekten ist im Java Umfeld noch nicht verfügbar bzw. nicht produktiv einsetzbar. (2) Da die Daten zeitgleich von Altanwendungen verändert werden können, ist die Transaktionssicherheit nur auf der Datenbankebene gewährleistet.

Operationen auf Geschäftsobjekten werden unter Zuhilfenahme von Instanzen einer Transaktionskontext-Klasse durchgeführt, die dazu dient, den Transaktionsrahmen zu definieren. Alle neuen oder geänderten Objekte werden mit einer Instanz eines Transkationskontext registriert und alle Objekte werden innerhalb eines Transaktionskontext gelesen. Geschäftsobjekte werden manipuliert, indem eine Instanz eines Transaktionskontext (durch die Application Root und die Abstract Factory) erzeugt wird. Das Erstellen und Lesen dieser Objekten wird mit Hilfe der Abstract Factory, an die dieser Transaktionskontext weitergegeben wird, durchgeführt (auch über die Application Root referenziert). Der „Commit“ eines Transaktionskontext aktualisiert den Status aller damit in Verbindung stehender Geschäftsobjekte in der Datenbank bzw. dem Cache. Alle Server lesen über eine gemeinsame Session-Instanz und alle Clients erstellen und aktualisieren über ihre eigene Client-Session (das Pooling von Client Sessions wird in einem späteren Schritt realisiert).

Das Verwalten der Instanzen wird auf der Serverseite mit Hilfe der Application Root und der Transaktionskontext-Instanzen durchgeführt. Jede Application Root (pro Client) enthält eine Zusammenstellung aller Transaktionskontext-Instanzen, die gerade von einem Client verwendet werden, und jeder Transaktionskontext enthält eine Zusammenstellung aller Geschäftsobjektinstanzen innerhalb diese Kontexts. Wenn ein Transaktionskontext von einem Client geschlossen wird, werden alle assoziierten Geschäftsobjekte innerhalb dieses Kontexts vom ORB entfernt (und anschließend die Transaktionskontext-Instanz selbst). Wenn sich ein Client "abmeldet", werden alle offenen Transaktionskontexte in der gleichen Weise

behandelt. Mit Hilfe eines Ereignismechanismus des Visibroker können anormal geschlossene Client-Verbindungen geprüft und verbliebene Instanzen auf ähnliche Weise bereinigt werden.

Die Abstract Factory auf dem Server dient sowohl als Factory Finder als auch als Factory. Aufrufe der Factory schließen immer eine Information über die Eingangsklasse ein. Diese Information wird zusammen mit dem Naming-Service verwendet, um den Aufruf zu einem passenden Knoten im Netzwerk zu leiten (mit dem Namen des Dispensers/Servers und relevanter Konfigurationsinformation). Referenzen auf fremde Knoten werden mit Hilfe spezieller Platzhalterinstanzen behandelt. Diese Referenzen werden nur bei Bedarf ausgewertet - sie werden durch Aufruf der Factory Instanz erstellt.

Alle generischen Einrichtungen für Geschäftsobjekte werden in einer fundamentalen Basisklasse, dem Abstract Domain Object, implementiert. Dies schließt generische Set- und Get-Methoden ein, die es den Clients ermöglichen, Messages im Netzwerk zu bündeln (mit Hilfe von Wertepaaren mit Information über das Attribut und den Wert, der gesetzt werden soll [4]). Es schließt auch generische Überprüfungsmethoden ein, die Informationen aus dem Meta-Level Dictionary verwenden, um die Objekte und Attribute zu überprüfen - Informationen, die während der Laufzeit dynamisch geändert werden können.

5 Untersuchungen zur Performance

Die Untersuchungen zur Performance der Anwendung wurden in zwei Phasen durchgeführt. Die erste Phase des Benchmarks wurde vor Ort bei einem Kunden durchgeführt. Dieser Kunde stellte eine Umgebung zur Verfügung, welche die wesentlichen Elemente der Produktionsumgebung umfaßte. An diese Phase schloß sich eine weitere Reihe kontrollierter Tests an, welche im firmeneigenen Testlabor in Atlanta stattfanden.

In beiden Tests wurde das Anwendungsverhalten über alle Server repliziert. Das heißt die Instanzen der Geschäftsobjekte wurden in jedem Server nach Bedarf erzeugt. Es waren keine serverübergreifenden Referenzen erforderlich (die Server kommunizierten nur mit dem Client und der Datenbank).

Der erste Test diente zum Bestimmen der Skalierbarkeit der eingesetzten Technologie. Zur Simulation des Benutzerverhaltens und der Client-Anwendungen wurde eine Testanwendung erstellt. Diese Testanwendung unterstütze zwei Benutzerprofile nämlich das der Auftragsannahme (Rapid Data Entry) sowie ein Call Center Profil. Geschäftstransaktionen beschränkten sich auf das Anlegen und Ändern von Kundeninformationen sowie beliebige Abfragen und Suchen über Kunden und dazu aggregierte Objekte. Jeder Test-Client konnte eine beliebige Anzahl von „realen„ Clients in unterschiedlichen Threads simulieren.

Um das Lastprofil des Servers möglichst genau dem von realen Benutzern erzeugten anzunähern, wurde eine zufällige Wartezeit zwischen zwei Geschäftstransaktionen implementiert. Der Wertebereich dieser Wartezeit war pro Test-Client konfigurierbar. Für die durchgeführten Tests wurden Zeiten zwischen 60

und 240 Sekunden eingestellt. Diese Zeitdauer entspricht der Zeit, die ein wirklicher Endbenutzer benötigt, um andere anfallende (manuelle) Tätigkeiten auszuführen oder den als nächstes anstehenden Vorgang vorzubereiten. Ebenso war die Zeitdauer zwischen der Kundensuche und der nachfolgenden Änderung der Kundendaten konfigurierbar.

Für die Tests der zweiten Serie wurde zusätzlich die Testsoftware „LoadRunner“ eingesetzt. Mit ihrer Hilfe wurden Endbenutzer-Aktionen mit der graphischen Benutzerschnittstelle des Clients simuliert und die Antwortzeiten der unterschiedlichen Geschäftstransaktionen aufgezeichnet. Bei diesen Tests wurde ein Client mit graphischer Oberfläche für die Messung eingesetzt, die Test-Clients dienten dazu, die Last weiterer Endbenutzer zu simulieren.

5.1 Messungen

Die Test-Client Anwendung bot zusätzliche Dienste um verschiedenste Aspekte der Server-Performance messen und Statistiken über alle Client-Prozesse für einen Lauf zu sammeln. Die bedeutendste Messung war die der Dauer der Geschäftstransaktionen. Der Test-Client lieferte Zahlen über minimale, maximale und mittlere Dauer für jede der definierten Transaktionen. Die Meßdaten erhielt man aus einer Reihe von Testläufen. Bei jedem neuen Testlauf wurde die Zahl der Client-Prozesse erhöht, bis die mittlere Dauer einer Transaktion eine vorher definierte Zeitdauer überschritten hatte. Eine gleiche Testreihe wurde mit mehreren Anwendungsservern durchlaufen. Ziel war es dabei, den Gewinn an Performance durch das Einführen zusätzlicher Anwendungsserver (bei gleicher Zahl Test-Clients) zu bestimmen.

Zusätzlich zu den oben beschriebenen Testreihen wurden Messungen durchgeführt, die dem Entwicklungsteam Aufschluß über weitere Aspekte der Anwendung bieten sollten, oder aber notwendig waren, um Informationen zur Erklärung der Ergebnisse der ersten Testreihe zu gewinnen.

Die weiteren Messungen umfaßten die Laufzeit bestimmter Methodenaufrufe im Server, die Laufzeit verschiedener Arten von Methodenaufrufen (z.B. generische gebündelte Set- und Get-Methoden auf Basis der Java Reflektion, klassenspezifische Strukturen, individuelle Set- und Get-Methoden), der Einfluß des Einsatzes von SSL auf die Transaktionsdauer (SSL wurde für die primären Messungen nicht aktiviert) und die Auswirkung der unterschiedlichen Thread-Mechanismen der virtuellen Maschine.

5.2 Hardware/Software Konfiguration

Im folgenden ist die Konfiguration von Hardware und Software sowohl für die externen Tests beim Kunden als auch für den internen Test beschrieben.

5.2.1 Externe Testumgebung

Die Datenbank lief auf einem HP T600 12 CPU Rechner mit 4GB Hauptspeicher und 1.5 TB Plattenspeicher. Die Clients waren auf Windows NT Rechnern mit Intel 266MHz Pentium Prozessoren, 96MB Hauptspeicher und 4GB Festplattenkapazität installiert. Die virtuelle Maschine von SUN (inklusive JIT Übersetzter) wurde eingesetzt. Der Anwendungsserver wurde zum Einen auf einem HP T600 Rechner (HP-UX 10.20, 4CPUs) mit 4GB Hauptspeicher und 120 GB Plattenspeicher zusammen mit dem „HP Native compiler for Java„, zum Anderen auf den Client-Rechnern (Windows NT) getestet. Auch die Datenbank spiegelte die produktive Umgebung wieder. Sie enthielt mehr als 1 Million Kundeneinträge.

5.2.2 Interne Testumgebung

Die Datenbank war auf einem HP T600 12 CPU Rechner mit 4GB Hauptspeicher und 1.5 TB Plattenspeicher installiert. Die Clients liefen auf 3 Windows NT Rechnern mit Intel 266MHz Pentium II Prozessoren, 300MB Hauptspeicher und 4GB Plattenkapazität. Ein Intel 200MHz Pentium Rechner mit 64 MB Hauptspeicher und 1.2GB Plattenspeicher wurde für die LoadRunner-Tests eingesetzt. Die Anwendungsserver liefen alternativ auf den drei NT Servern (wie die Clients), zusätzlich konnte eine Maschine mit einem 333MHz Pentium II Prozessor genutzt werden. Die Datenbank für den internen Test war mit 10.000 Kunden geladen.

5.2.3 Allgemeines

Bei allen Tests wurde Version 1.1.6 des Java JDK, Visibroker for Java 3.1.0, TOPLink 1.0 und Oracle 8.0 eingesetzt. Für beide Testreihen stand ein 100MB Netzwerk zur Verfügung. Von Oracle werden zwei JDBC Treiber für die Oracle Datenbank angeboten – der „Thin Client„- und der OCI- Treiber. Der OCI-Treiber ist zwar für Serveranwendung ausgelegt, lief aber zu keinem Zeitpunkt fehlerfrei. Aus den anderen am Markt erhältlichen OCI-Treibern wurde der von WebLogic ausgewählt und für die endgültigen Tests eingesetzt.

6 Ergebnisse

Die Ergebnisse der Untersuchungen des Performance-Verhaltens können anhand der Informationen aus Bild 2 zusammengefasst werden. Es beschreibt die grundlegende Messung zur Performance des Servers – die mittlere Laufzeit einer Geschäftstransaktion in Abhängigkeit von der Anzahl der Clients bei genau einem verfügbaren Anwendungsserver.

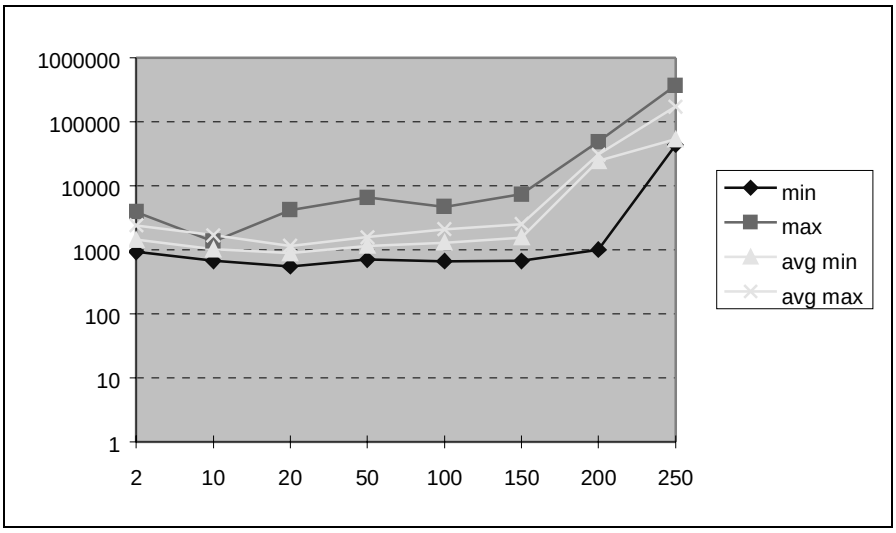

Bild 2. Laufzeit der Geschäftstransaktionen in Abhängigkeit von der Anzahl Clients für einen Server (x-Achse – Anzahl der Clients, y-Achse – Zeit in Millisekunden).

Bild 2 stellt lediglich eine der ausgeführten Testreihen, nämlich das Anlegen neuer Kunden, dar. Dies war das einzige Szenario, das sich bei jeder Ausführung gleich verhielt und damit zu reproduzierbaren Ergebnissen führte.

Die anderen Szenarien erzeugten zufällige Abfragen, was je Testlauf in stark unterschiedlichen Ergebnissen (von genau einem bis zu mehreren Tausenden von Objekten) resultierte. Diese Tatsache hatte zur Folge, daß ein Testlauf mit einem Client signifikant langsamer war, als ein weiterer mit 50 Clients. Im Durchschnitt konnte allerdings eine Verdopplung der Laufzeit bei den ‚Update'-Szenarien im Vergleich zu den ‚Insert'-Szenarien festgestellt werden. Die Streuung der Antwortzeiten der ‚Query'-Szenarien war sehr groß – Millisekunden bis mehrere Minuten (für sehr große Anzahl von Clients) – doch die Mehrheit der Messungen lag zwischen 2 und 10 Sekunden.

7 Diskussion der Testergebnisse

Dieses Kapitel diskutiert die Ergebnisse der Performance-Untersuchung im Detail und versucht die ermittelten Zahlen zu interpretieren und zu erklären. Da vielfältige Aspekte der Anwendung die Performance beeinflussen, wurde der Ansatz gewählt, diese Aspekte und ihre Bedeutung im Folgenden einzeln zu behandeln.

7.1 Allgemeines

Das wesentliche Ergebnis der Performance-Tests stellt keine Überraschung dar: Die mittlere Laufzeit einer Geschäftstransaktion auf einem Server nimmt exponentiell mit der Anzahl der Clients zu. Der Schwellwert, an dem die Kurve drastisch ansteigt wird in einem Intervall zwischen 50 und 100 Clients erreicht (einige Test hatten sogar höhere Werte zum Ergebnis). Die Ergebnisse erlauben den Schluß, daß ein einzelner Server eine Last von mindestens 50 Clients bei einer mittleren Transaktionsdauer von 2-3 Sekunden verarbeiten kann (dies gilt für Inserts und Updates, Abfragen dauern länger).

Sogar bei 100 Clients war die CPU der Server-Maschine nur zu 60% ausgelastet. Der Serverprozeß hat demzufolge seine Auslastungsgrenze noch nicht erreicht. Das liegt wohl darin begründet, daß der Serverprozeß im Leerlauf ist und entweder auf Aufträge des Client oder aber auf Antworten der Datenbank wartet. Hier ist eine weitere Analyse notwendig, um das Potential der Prozessoren vollständig ausschöpfen zu können.

Eine interessante Beobachtung war, daß die Laufzeit der Mehrheit der Geschäftstransaktionen mehr als 50 Prozent unter dem durchschnittlichen Wert lag. Die Durchschnittswerte wurden dabei von Ergebnissen einzelner Transaktionen verfälscht, welche im Vergleich zu allen anderen besonders lang andauerten. Eine genaue Analyse dieses Phänomens zeigte, daß in der virtuellen Maschine von Sun 10-30% der Transaktionen über zehnmal länger dauerten als die restlichen. Es konnte im Anschluß an die Auswertung dieser Ergebnisse gezeigt werden, daß in der virtuellen Maschine von Microsoft dieses Verhalten nicht existierte. Wie zu erwarten war, war die erste Transaktion bedingt durch den JIT Übersetzer langsamer als der Durchschnitt. Die anderen überdurchschnittlich langen Transaktionen traten jedoch zufällig und nach keinem erkennbaren Muster auf. Im Rahmen dieses Projektes war es uns nicht möglich die Ursache dieses Verhaltens weiter zu analysieren, die Problembeschreibung wurde aber an Sun gemeldet.

Es muß an dieser Stelle angemerkt werden, daß die Anwendung auf neu erstellter Software von Drittanbietern basiert, welche zum Einen die für Erstreleases inzwischen ‚normale' Anzahl von Fehlern enthält und zum Anderen hinsichtlich der Performance der Anwendung kaum optimiert ist. Der Servercode selbst enthält nahezu keine Optimierung. Hinzu kommen noch einige Designentscheidungen, welche sich nachteilig auf die Performance auswirkten. Diese Entscheidungen wurden bewußt getroffen und waren entweder durch den gesteckten zeitlichen Rahmen oder aber durch Unzulänglichkeiten der Basis-Software erforderlich. Mit anderen Worten: die Performance kann an diesen Stellen noch wesentlich verbessert werden.

Auf der anderen Seite gibt es auch einige Faktoren, welche die Ergebnisse zum positiven verfälschen. Die im Test eingesetzte Datenbank war zum Beispiel nicht durch andere Anwendungen belastet (der Datenbankserver war die meiste Zeit über im Leerlauf). Die Kommunikation zwischen Client und Server wurde ohne zeitintensive Sicherheitseinrichtungen wie z.B. SSL betrieben. Die SSL Testergebnisse zeigten, daß die Nutzung von SSL mit 7%-igen Anstieg der

Antwortzeiten zu Buche schlägt (durchschnittliche 1470,5 ms + 7% = 1574ms). Zudem war das zugrunde liegende Netzwerk mit 100MB sehr schnell.

7.2 Java

Die Performance von Java hat im letzen Jahr einige grundlegende Verbesserungen erfahren; dies leider nicht auf allen Plattformen. Am deutlichsten wurde diese Tatsache an den unterschiedlichen Meßergebnissen von Plattformen, Übersetzern und virtuellen Maschinen. Das Antwortzeitverhalten des Servers auf einer einfachen (kostengünstigen) NT Maschine mit SUN VM und JIT Übersetzter war signifikant besser als die Performance auf einer in der Hardware-Ausstattung weit überlegeneren HP Maschine (mit HP VM und dem HP native Übersetzer). Hier stehen 10s gegen 49s für das Anlegen eines Kunden bei 25 Clients.

Dies liegt wohl in dem hohen Forschungs- und Entwicklungsaufwand begründet, der zu Verbesserung von Java auf Intel Plattformen aufgewandt wurde. Die virtuelle Maschine von Hewlett Packard unterstützt zum Beispiel keine native Threads. Dieser Punkt ist besonders beachtenswert, denn er deckt eine der Hauptschwächen der Java Portabilität auf.

7.3 CORBA

In jeder verteilten Anwendung ist die Zeit, welche zur Kommunikation über das Netzwerk benötigt wird, einer der Schlüsselfaktoren, welche die Performance bestimmen. Dies gilt im allgemeinen für alle Arten der Kommunikation über ein Netzwerk, und für CORBA im besonderen. Der Aufruf einer Methode eines Objekts, das sich auf einer entfernten Maschine befindet benötigt erheblich länger, als ein Aufruf innerhalb der gleichen Maschine oder gar innerhalb desselben Prozesses.

Bei dem Design des Servers wurde besonders darauf geachtet, die Performance Engpässe, welche aus einer CORBA basierten Architektur resultieren, von vornherein zu Erkennen und zu Berücksichtigen. Es wurden Dienste implementiert, die es erlaubten, mehrere Messages zu einer zu bündeln, um so den Aufwand, der zum Aufbereiten, Verpacken und Übertragen einer Message notwendig ist, zu minimieren [1]. Einfache Datentypen oder Sequenzen wurden eingesetzt, der Typ ANY wurde soweit möglich vermieden (nachweislich ein äußerst kostenintensiver Datentyp).

Die Unterschiede in den Laufzeiten zur Nachrichtenübertragung in Abhängigkeit von den Strategien zum Bündeln der Nachrichten wurden während der Tests untersucht. Die Ergebnisse werden in die weiteren Entwicklungen des Servers einfließen. Für unseren Zweck war in den meisten Fällen der Ansatz generischer Get- und Set-Methoden zufriedenstellend. In einigen Fällen wurde es notwendig, klassenspezifische ‚Detail'-Strukturen zu implementieren. Nachrichten, welche auf Strukturen basieren, waren 50% schneller als jene, welche auf dem generischen Get/Set Ansatz aufsetzen. Diese wiederum waren 40% schneller als die kombinierten individuellen Get- und Set-Methoden [4].

CORBA verhindert die automatische Java Garbage-Collection, da es verlangt, daß alle Objekte, welche bei einem ORB registriert sind, explizit von dort gelöscht werden (ansonsten hält der ORB eine Referenz auf diese Objekte, welche das Einsammeln durch den Garbage-Collector verhindert). Eine der wesentlichen Herausforderungen für das Server-Entwicklungsteam war demnach das Design eines Speicherverwaltungs-Dienstes. Hier wurde ein Referenz-Zählverfahren gewählt. Die Entscheidung war bestimmt von der Anforderung, daß (um Speicherplatz zu sparen) sich alle Clients ‚Read-Only'-Objekte teilen. Diese können nur dann zum Löschen freigegeben werden, wenn sie von keinem Client mehr referenziert werden.

7.4 Architektur

Einige Teilgebiete der Serverarchitektur, welche die Performance direkt beeinflussen, sind zum heutigen Zeitpunkt nicht optimal gelöst. Eine Randbedingung für die Serverarchitektur war, daß die Anwendung parallel mit dem bereits existierenden System laufen kann. Eine grundlegende Designentscheidung war daher, den gültigen Status der Geschäftsobjekte in die Datenbank zu legen. Die Persistenzschicht legt Objektinstanzen pro Server im Cache ab, aber alle Clients arbeiten auf ihren eigenen Kopien der Geschäftsobjektinstanzen. Die Geschäftsobjekte haben deshalb "Status" Informationen, der tatsächliche Status der Anwendung liegt jedoch in der Datenbank..

Das Transaktionskonzept führt zu weiteren Kopien, da jeder Client ja bereits auf einer eigenen Kopie der Geschäftsobjekte arbeitet. Aufgrund von Designfehlern in der eingesetzten Software wurden Kopien von Objekten in Situationen angelegt und benutzt, die dies eigentlich nicht erforderten. Um die daraus resultierenden Performance-Verluste zu vermeiden, wurde das Design des Server geändert: Geschäftsobjekte mit ausschließlichem Lese-Zugriff werden nur einmal gehalten. Alle Clients teilen sich diese Objekte. Der Server verwaltet die Referenzen von Clients auf ein solches Objekt und löscht dieses Objekt dann, wenn es von keinem Client mehr referenziert wird.

Ein weiterer Bereich, dessen Architektur genauer analysiert werden mußte, war das Ausführen von Abfragen. Der Server bietet Mechanismen an, um Abfragen auf Objekte zu stellen, ohne daß diese Objekte wirklich erzeugt werden (das Ergebnis ist eine virtuelle Liste, in der jedes Listenelement wiederum eine Liste mit String-Werten der gewünschten Attribute enthält). Im Gegensatz zu diesem Ansatz erzeugt das eigentliche Query-Interface mehrere Objekte für jedes Ergebnis-Objekt. Die deutlich bessere Performance der ersten Lösung (im schlechtesten Fall 2 Sekunden gegen 8 Sekunden) zeigt hier das Potential an Verbesserungsmöglichkeiten.

Um die geforderte Flexibilität und leichte Konfigurierbarkeit der Anwendung zu erreichen, wurden die Möglichkeiten der Java Reflection-Klassen intensiv genutzt. Die Auswertung des Anwendungsprofils zeigte allerdings, daß die Benutzung dieser Funktionalität so langsam ist, daß sich die Frage des Einsatzes in einer wirklich zeitkritischen Anwendung stellen muß. Nach Abwägen zwischen Performance und Flexibilität haben wir für unsere Anwendung entschieden, im ersten Ansatz die Java

Reflection einzusetzen. Zu einem späteren Zeitpunkt werden wir Alternativen suchen, um bei gleicher Funktionalität den Durchsatz weiter erhöhen zu können.

7.5 TOPLink

Die gesamte Interaktion zwischen dem Server und der Datenbank erfolgt unter Zuhilfenahme von TOPLink, einer objekt-relationalen Mapping-Software. Für unsere Tests wurde die Version 1.0 eingesetzt. Unsere Erfahrungen zeigen, daß hier noch ein großer Spielraum für Verbesserungen besteht. Das Produkt wurde augenscheinlich bisher noch in keiner geschäftskritischen, mehrschichtigen Anwendung eingesetzt. Fehlerhafte oder wenig flexible Implementierung erzwangen einige für die Performance ungünstige Designentscheidungen auf der Serverseite. Das Projektteam ist inzwischen in engem Kontakt mit dem Hersteller der Software, der eine Lösung der dringendsten Probleme in der nahen Zukunft zugesagt hat.

Der Hauptgrund für den Einsatz von TOPLink war, den Aufwand für die Abbildung der Objekte auf das relationale Datenbank-Schema, zu minimieren. Diese Aufgabe wird von der Software zufriedenstellend gelöst. Es gibt noch einige wenige Abbildungsprobleme. Diese sind aber nur zu einem kleinen Teil in einer eingeschränkten Funktionalität von TOPLink begründet, sondern zum größeren Teil durch das Datenbankschema gegeben, das an manchen Stellen nicht optimal ist.

Die Abbildungs-Funktionalität von TOPLink ist hinsichtlich der Performance noch nicht optimiert. Wie in allen ersten Versionen lag auch hier der Schwerpunkt auf der Erfüllung der funktionalen Anforderungen. Für eine nächste Version wurde bereits eine signifikante Verbesserung der Performance zugesagt.

Einer der durchgeführten Tests sollte die Frage beantworten, wie sich der Durchsatz des Servers in der Abhängigkeit von der Größe des TOPLink Caches verhält. Die Ergebnisse waren leider nicht die erwarteten. Unsere Architektur und die Art, in der wir TOPLink benutzen, verlangt eine bestimmte Minimalgröße des Caches. Cache-Größen unterhalb dieser Schwelle führten zum Absturz des Servers, da TOPLink Instanzen aus dem Cache entfernte, obwohl diese noch von Clients referenziert wurden. Für Größen oberhalb diese Schwellwertes konnte kein Einfluß mehr auf den Durchsatz festgestellt werden.

7.6 JDBC Treiber

Die Interaktion einer Java Anwendung mit einer (relationalen) Datenbank erfordert in fast allen Fällen den Einsatz eines JDBC Treibers. Oracle bietet zwei solcher Treiber an: eine „Thin-Client“ Version und eine OCI Version. Die Thin-Client Version wurde mit dem Ziel eines möglichst geringen Resourcenbedarfs entworfen, um den Treiber so schneller über ein Netz laden zu können. Das primäre Einsatzgebiet sind daher zweischichtige Anwendungen. Der OCI Treiber benötigt mehr Speicher, ist aber für mehrfache Verbindungen und große Datenvolumen ausgelegt, wie sie ein Anwendungsserver erforderlich macht.

Unglücklicherweise arbeitete der zum Testzeitpunkt von Oracle erhältliche OCI Treiber nicht ordnungsgemäß. Dies zwang uns zum Einsatz der Thin-Client Version, was sich in schlechteren Meßergebnissen niederschlug. Die fehlerhafte Implementierung des Oracle OCI Treibers wurde offensichtlich nachdem wir den OCI Treiber eines Drittanbieters (WebLogic) erfolgreich installiert hatten. Die Transaktionen liefen jetzt beinahe 30% schneller (von 1.2s auf 1.6s).

7.7 Skalierbarkeit

Das wichtigste Ziel der Tests war, die Skalierbarkeit der Technologie zu untersuchen. Erlaubt die Technologie Transkationsvolumen und eine Anzahl gleichzeitiger Benutzer, die für eine Kundenverwaltungsanwendung der nächsten Generation gefordert werden? Die in diesem Beitrag aufgeführten Ergebnisse erlauben kein enthusiastisches ‚Ja' als Antwort auf diese Frage. Die Transaktionen sind in einigen Fällen noch zu langsam. Aber die Ergebnisse weisen nach, daß der Server zufriedenstellend skaliert.

Die Testreihen mit mehreren Servern zeigen, daß zusätzliche Benutzer und somit zusätzliche Transaktionen einfach durch das Hinzufügen weiterer Server ausgeglichen werden können. Es war nicht möglich, die Obergrenze des Performance-Gewinns, der durch das Hinzufügen weiterer Server erreicht wird, zu ermitteln (die Obergrenze ist dann erreicht, wenn der Gewinn an Durchsatz durch zusätzliche Server zu teuer wird). Die Kurve des Durchsatzes in Abhängigkeit von der Anzahl der Server verläuft theoretisch nicht linear, was wir allerdings messen konnten, war annähernd linear. Das legt nahe, daß wir nur das untere Ende der Kurve bestimmen konnten.

8 Zusammenfassung und Ausblick

Aus den Ergebnissen des durchgeführten Performance-Tests kann man schließen, daß die untersuchten Technologien die Anforderungen an ein Kundenverwaltungssystem der nächsten Generation erfüllen können. Die Messungen zeigten, daß ein auf einer kostengünstigen NT Maschine installierter Server bei einer maximal zulässigen Antwortzeit von 2-3 Sekunden bereits einen Transaktionsdurchsatz von 50 Clients zufriedenstellen verarbeiten kann (bei konservativer Schätzung). Die realistische Anzahl von Clients liegt bei 100, eine optimistische Betrachtung führt zu einer Zahl von 150 Clients.

Ein Erhöhen der Anzahl von Anwendungsservern erhöht den gesamten Durchsatz des Systems. Obwohl der Durchsatz in Abhängigkeit von der Anzahl Anwendungsserver keine lineare Funktion ist, gibt es keinen Grund zur Annahme, daß ein solches System nicht mehrere Tausend Benutzer unterstützen kann. Diese Annahme wird auch durch die Art der Anwendung unterstützt, die ja im Wesentlichen Daten aus einer Datenbank liest und schreibt.

Wenn man auch einige der Meßergebnisse (im Speziellen die für Abfragen) als ‚langsam' einstufen kann, zeigt die Analyse der Tests, daß es hier noch genügend Spielraum für Verbesserungen gibt. Bereits zum heutigen Zeitpunkt liegt die Mehrheit der Transaktionen bei der Hälfte des Durchschnitts. Wir vertreten daher den optimistischen Standpunkt, daß diese Technologie in ausreichendem Maße skalierbar ist.

Der aktuelle Stand auf dem Gebiet der Entwicklungswerkzeuge für diese Technologien darf als kritischer Faktor nicht unterschätzt werden. Viele dieser Werkzeuge sind nur in sehr frühen Versionen verfügbar und damit noch sehr fehlerbehaftet. Die Integration von CORBA und Java IDEs ist weiterhin unterentwickelt. Wir sind jedoch der Meinung, daß das Potential für eine sehr hohe Produktivität gegeben ist. Unterstellt man, daß sich diese Technologien so weiterentwickeln wie bisher, kann man davon ausgehen, daß die Kombination von CORBA und Java bald zu einer der meistbenutzten Technologien in der Softwareindustrie zählen wird.

Referenzen

1. Eichern, Kamber, Murer, 'CORBA: Principles and practical experiences', Informatik/Informatique, 2/97
2. Mowbray, Malveau, 'CORBA Design Patterns', Wiley Computer Publishing 1997
3. Swainston-Rainford, 'Ringing the Changes to the IDL Interface', Application Development Advisor, Sept/Oct 1997
4. Swainston-Rainford, 'Designing Your IDeaL Interface', Application Development Advisor, Nov/Dec 1997
5. Orfali, Harkey, ‚Client/Server Programming with Java and CORBA', Wiley Computer Publishing 1998