

mehr zum thema:
java.sun.com/j2ee/
www.omg.org/

EINE SYSTEMMIGRATION VON C++ NACH EJB

Anfang 2001 startete ein Projekt zur Migration eines Ticketvertriebssystems von C++ nach EJB: Das System wurde mit der EJB-Technologie neu realisiert, nachdem akute Betriebsprobleme aufgetreten waren. Die Performance der Anwendung verbesserte sich dabei um das Dreifache. Der Artikel gibt die Erfahrungen aus dem Projekt wieder.

In den Vorverkaufsstellen der CTS Eventim AG können Tickets für nationale und internationale Veranstaltungen reserviert und gebucht werden. Der Verkauf von Tickets in den Bereichen Rock/Pop, Theater/Musical, Varieté/Revue, Klassik, Volksmusik und Sport wird mit dem Ticketvertriebssystem Euroticket abgewickelt, ebenso wie die Abrechnung von Veranstaltungen. Die Ticket- und Veranstaltungsdaten werden in einem zentralen Datenbankserver gespeichert. Das Datenvolumen beträgt zur Zeit etwa 80 GByte.

Auf die zentralen Daten kann mit zwei Arten von Clients zugegriffen werden:

- Für Backoffice-Aufgaben (Pflege von Veranstaltungen und Vorverkaufsstellen, Reports, Abrechnungen) steht der **Veranstalter-Client** zur Verfügung.
- Für Frontoffice-Aufgaben (Reservierung und Verkauf von Tickets, Ticketdruck, Verteilung von Vorverkaufsinformationen an die Vorverkaufsstellen) ist der **Vorverkaufs-Client** in allen Vorverkaufsstellen installiert.

Sowohl die Clients als auch der Anwendungsserver sind mit C++ realisiert (siehe Abb. 1). Der Server enthält einen Dispatcher und die Ticketverwaltung: Der Dispatcher ermöglicht die Socket-Kommunikation zwischen Clients und Server über ein TCP/IP-Netzwerk. In der Ticketverwaltung werden alle fachlichen Aufgaben der Anwendung ausgeführt.

Mit den insgesamt ca. 1.500 Clients des Euroticket-Systems werden in Spitzenzeiten ungefähr 20.000 Tickets pro Stunde verkauft, dabei erfolgen etwa 100.000 Serveraufrufe. Diese Zahlen verdeutlichen bereits die besonderen Anforderungen beim Ticketverkauf:

- Verzögerungen in der Anwendung führen zu Warteschlangen an den Vorverkaufsstellen und wirken sich unmittelbar auf den Umsatz aus.
- Die grafische Darstellung von Saalplänen ist sehr aufwändig; pro Zugriff müssen Hunderte bis Tausende von Datensätzen gelesen und verarbeitet werden.
- Konkurrierende Zugriffe auf begehrte Plätze einer Veranstaltung sind die Regel und nicht die Ausnahme – das gilt insbesondere bei hoher Last.
- An die Sicherheit werden erhöhte Anforderungen gestellt, da die gedruckten Tickets einen Geldwert darstellen.
- Schließlich müssen Reports für Verkaufsanalysen und Abrechnungen aktuell und sekundengenau verfügbar sein.

Nachdem Doppelverkäufe und Performance-Engpässe zu Betriebsproblemen geführt hatten, beauftragte die CTS Eventim AG im Sommer 2000 die sd&m AG mit einer Analyse des Euroticket-Systems. Die Analyse zeigte, dass die Wartbarkeit und Betriebssicherheit des Systems mit einem hohen Risiko verbunden waren. Für diese Einschätzung gab es mehrere Gründe:

- Fehler bei der Nachrichtenübertragung zwischen Client und Server führten zu Instabilitäten der Anwendung.
- Ineffiziente Datenbankzugriffe verringerten die Performance; die Antwortzeiten waren zu hoch.
- Die Grenze der Wartbarkeit war erreicht; fachliche und technische Erweiterungen konnten nur mit hohem Aufwand realisiert werden und enthielten das Risiko einer weiteren Destabilisierung.
- Zudem waren die Schichten der Systemarchitektur nicht deutlich von-

die autoren



Dr. Karl-Heinz Wichert (E-Mail: Karl-Heinz.Wichert@sdm.de) ist Technischer Berater bei der sd&m AG. Er hat mehrjährige Erfahrungen im Design von Client/Server-Systemen und Internet-Anwendungen mit objektorientierten Technologien.



Martin Masuch (E-Mail: Martin.Masuch@sdm.de) ist Software-Ingenieur bei der sd&m AG. Sein Schwerpunktthema ist die Softwareentwicklung mit objektorientierten Technologien.

einander abgegrenzt und enthielten keine klar definierten Schnittstellen; deshalb wirkten sich Änderungen in einer Schicht in der Regel auf mindestens eine weitere Schicht aus.

Die sd&m AG erarbeitete ein Konzept für die stufenweise Migration des Anwendungsservers auf eine neue Zielarchitektur und realisierte seit Anfang 2001 die neue Architektur mit EJB-Komponenten. Ein Entwicklungsteam von ca. 15 Mitarbeitern erzielte eine Stabilisierung des Systems und verbesserte die Performance um das Dreifache.

Aspekte der Migration

Eine Umstrukturierung des bestehenden Systems war vor allem wegen der mangelhaften Schichtenarchitektur wirtschaftlich nicht vertretbar. Deshalb zogen wir zunächst eine Neurealisierung der gesamten Anwendung in Erwägung. Während der Phase der Neuentwicklung hätte jedoch ein zusätzliches Entwicklungs-►



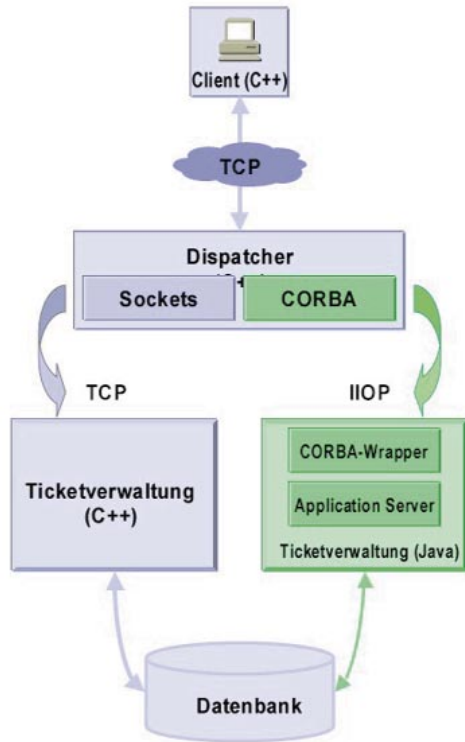


Abb. 1: Architektur Euroticket: Altsystem (grau), neue Systemkomponenten (grün)

team den Betrieb des Altsystems und die Anpassung fachlicher Änderungen sicherstellen müssen. Eine neue fachliche Spezifikation hätte weitere Zeit gekostet.

Daher entschieden wir uns für eine stufenweise Migration: Diese Vorgehensweise erhöht zwar kurzfristig das Betriebsrisiko, weil versteckte Abhängigkeiten und die große Zahl von fachlichen und technischen Fehlern zu nicht vorhersehbaren Effekten führen können. Andererseits war unter den gegebenen Bedingungen zu erwarten, dass sich eine stufenweise Migration in einem ähnlichen Kostenvolumen wie eine vergleichbare Neuimplementierung bewegen würde. Gleichzeitig ermöglichte die stufenweise Vorgehensweise eine schnelle Reaktionsfähigkeit auf kurzfristige Anforderungen. In den einzelnen Migrationsschritten sollte jede neu ausgelieferte Version einen Teil des bestehenden Systems ersetzen. Mit den einzelnen Releases waren folgende Maßnahmen verbunden:

- Portieren des Servers auf die EJB-Architektur unter Verwendung eines Applikationsservers,
- Neu-Implementierung und Optimierung derjenigen Server-Transaktionen, die besonders häufig aufgerufen werden,

- Entlastung der Datenbank durch Einführung einer Auswertedatenbank, die ausschließlich für das Erstellen von Reports verwendet wird (Lese-Zugriffe) und
- Vermindern der Konfliktwahrscheinlichkeit bei Datenbankzugriffen durch transientes Speichern reservierter Tickets und durch Serialisieren von konkurrierenden Schreibzugriffen.

In der neuen Architektur sorgt der Dispatcher für die korrekte Zusammenarbeit von alten und neuen Systemteilen: Er regelt, welche Aktionen noch über das Altsystem auszuführen und welche Aktionen bereits im Neusystem zu bearbeiten sind (siehe Abb. 1). Client und Dispatcher sind nach wie vor mit C++ realisiert, während ein großer Teil der C++-Ticketverwaltung mit Java neu implementiert wurde. Der Dispatcher enthält zusätzlich eine CORBA-Schnittstelle, um die Verbindung zwischen C++, Java und EJB zu ermöglichen.

Bei der Auswahl geeigneter Technologien für die Realisierung des neuen Systems stand vor allem die Forderung nach Performance, Stabilität und Sicherheit im Vordergrund. Letztlich fiel die Entscheidung auf Java, EJB und den Applikationsserver von Borland („Borland AppServer“).

Warum Java?

Java ermöglicht eine sehr robuste Programmierung, da bestimmte sprachliche Mittel nicht existieren. Zum Beispiel ist der direkte Zugriff auf Speicherbereiche über Zeiger nicht möglich und das Überschreiten von *Array*- und *Stack*-Grenzen wird in *Exceptions* abgefangen. Darüber hinaus hängt die Performance von der Effizienz der Datenbankzugriffe ab, nicht von der Geschwindigkeit des Anwendungscode. Mit Java 2 und der *JIT*-Technologie (*Just-in-Time*) kommt es daher nicht zu Leistungseinschränkungen gegenüber C++ – dies konnte mit einem Prototypen nachgewiesen werden. Schließlich war mit Java eine deutliche Trennung vom Altsystem möglich; das Ziel war eine neue Implementierung mit Java, nicht einfach das Nachprogrammieren des C++-Codes.

Warum EJB?

EJB spart Entwicklungszeit, weil die *Low-Level*-Mechanismen der Verteilung, Transaktionsverwaltung und Sicherheit vom EJB-Container bereitgestellt werden. Zudem hat es die Vorteile einer Standardtechnologie. Mit der *CMP*-Engine (*Container-Managed Persistence*) des Borland AppServers existieren gute Erfahrungen: optimistisches Sperrkonzept, Schlüsselgenerierung, *tuned Updates* (d. h. nur die modifizierten Felder der Datenbank werden aktualisiert) und schnelles Laden. *Failover*-Mechanismen erhöhen die Sicherheit bei Ausfällen. Aspekte wie Komponentenarchitektur oder Verteilbarkeit waren in der Euroticket-Architektur hingegen weniger wichtig.

Kapselung von Entity-Beans

In einer klassischen, service-orientierten EJB-Anwendung werden die Methoden der Use-Cases von Session-Beans implementiert. Diese greifen dabei auf persistente Business-Objekte zu, die durch Entity-Beans realisiert werden. Ein direkter Zugriff von den Anwendungs-Clients auf Entity-Beans wird üblicherweise nicht gestattet.

Auch die Euroticket-Architektur orientiert sich an diesem Konzept. Ein Problem ist der enorme Overhead, der mit jedem Zugriff auf eine Entity-Bean verbunden ist: Der EJB-Container serialisiert vor jedem Aufruf die Übergabeparameter, greift auf den Transaktionsmanager zu und prüft Berechtigungen. Teuer ist auch das Erzeugen einer Entity-Bean. Mit EJB 2.0 sind einige dieser für die meisten Anwendungen überflüssigen Performance-Killer beseitigt worden. Als das Euroticket-Projekt entwickelt



wurde, gab es jedoch für die Version 2.0 der EJB-Spezifikation noch keine stabilen Implementierungen. Es ist auch zweifelhaft, ob mit einer Implementierung der EJB-Version 2.0 die gewünschte Performance erzielt werden kann.

In den meisten EJB-Anwendungen mit Entity-Beans werden deshalb die Entitäten so strukturiert, dass möglichst wenig Aufrufe von Entity-Beans entstehen. Zwei gängige Entwurfsmuster hierfür sind die Verwendung grob-granularer Entitäten (empfohlen in der EJB-1.1-Spezifikation) und die Beschränkung der Aufrufsstelle auf zwei Methoden (getData() und setData()). Beiden Methoden wird jeweils ein Value-Objekt übergeben. Falls eine Use-Case-Methode auf sehr viele Entitäten

modell ist in der Anwendungsarchitektur oft nur schwer wiederzuerkennen.

Viele EJB-Anwendungen verzichten deshalb ganz auf den Einsatz von Entity-Beans. Wir haben versucht, mit Entity-Beans eine leistungsfähige und strukturierte Architektur zu entwickeln: Hierzu haben wir zwischen den Use-Case-Objekten (Session-Beans) und den Datenbankobjekten (Entity-Beans) eine zusätzliche Schicht von reinen Java-Objekten eingefügt. Diese Objekte implementieren die fachlichen Schnittstellen der persistenten Datenbank-Entitäten. Wir bezeichnen sie deshalb als Business-Objekte. Das Synchronisieren der persistenten Attribute mit der Datenbank geschieht in der Regel über eine Entity-Bean. Diese enthält lediglich die zwei bereits erwähnten Metho-

Das Use-Case-Objekt arbeitet nur mit den Business-Objekten und den *Business-Factories* zusammen, nie mit den Entity-Beans. Das hat mehrere *Vorteile*:

- Zum einen haben die Business-Objekte eine fachliche Schnittstelle mit beliebigen Methoden, die ohne Performance-Overhead aufgerufen werden können.
- Zum anderen wird jede Entity-Bean pro Transaktion höchstens zweimal aufgerufen: getData() wird aufgerufen beim Erzeugen des Business-Objekts, setData() nach Änderungen der persistenten Attribute.
- Schließlich können die Entity-Beans zur Performance-Optimierung auch komplett umgangen werden. Diese Optimierung ist in der *Business-Factory* enthalten; sie ist für den Aufrufer der Business-Objekte vollkommen transparent.

Den Vorteilen stehen auch einige *Nachteile* gegenüber:

- Der Implementierungsaufwand erhöht sich und die Komplexität der Architektur steigt durch die zusätzliche Schicht von Objekten.
- Änderungen am Zustand der Business-Objekte durch den expliziten Aufruf einer update()-Methode müssen an die Entity-Beans weitergeleitet werden. Der EJB-Container sorgt dann am Ende der Transaktion dafür, dass die Änderungen in die Datenbank gelangen.
- Ein weiterer Nachteil ist, dass die Objekt-Identität bei wiederholtem Lesen derselben Entität aus der Datenbank innerhalb einer Transaktion nicht gewährleistet ist. Es werden dabei mehrere Business-Objekte erzeugt, die alle dieselbe Entity-Bean referenzieren. Das kann zu einem Problem werden, wenn innerhalb der Transaktion eine Änderung an einem oder mehreren dieser Business-Objekte erfolgt.

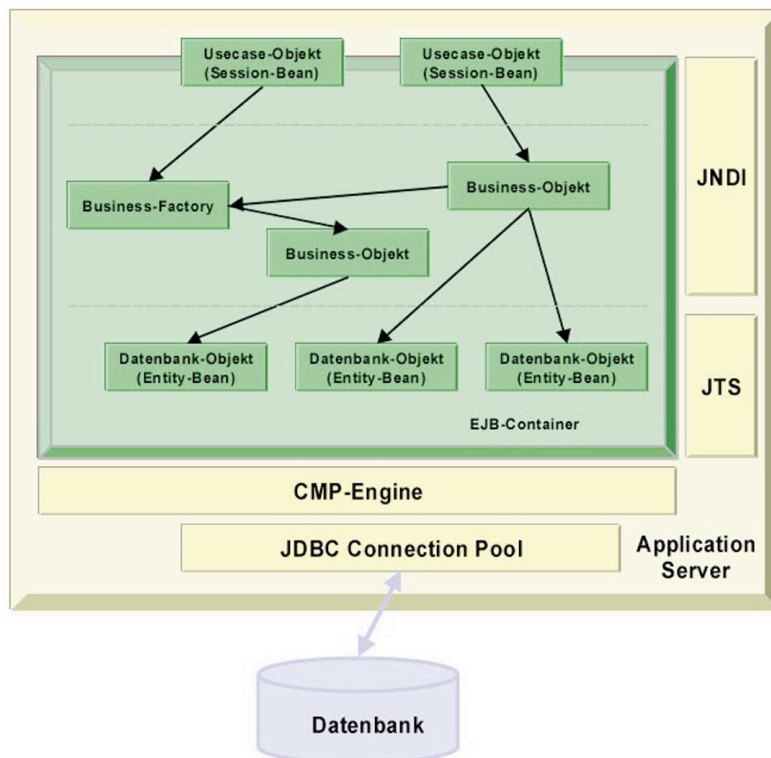


Abb. 2: EJB-Architektur im Euroticket-Server

zugreift oder extrem performance-kritisch ist, erfolgen die Datenbankzugriffe mit JDBC-Aufrufen direkt in der Session-Bean; die Verwendung von Entity-Beans wird hierbei umgangen.

Das Ergebnis dieser Tuning-Maßnahmen ist eine technisch geprägte, unstrukturierte Architektur, in der die Daten von den Methoden getrennt sind. Außerdem ist die gesamte Anwendungslogik in die Use-Case-Objekte verlagert. Das fachliche Daten-

den getData() und setData(), über die ein Value-Objekt ausgetauscht wird. Bei rein lesenden Zugriffen kann das Business-Objekt seine Daten auch direkt per JDBC aus der Datenbank lesen. Zur Verwaltung der Business-Objekte gibt es für jede Klasse eine Business-Factory, die auf das Home-Interface einer Entity-Bean zugreift. Sie kann aber auch JDBC-Anfragen direkt auf der Datenbank ausführen und mit den Ergebnisdaten Business-Objekte erzeugen (vgl. Abb. 2).

Das größte Problem ist sicherlich der hohe Implementierungsaufwand. Pro Datenbank-Entität müssen folgende Dateien realisiert werden: Remote-Interface, Home-Interface und Implementierungsklasse für die Entity-Bean, Business-Objekt, Business-Factory und Value-Objekt sowie zwei Deployment-Descriptor-Dateien, die das objekt-relationale Mapping beschreiben. Das ergibt acht zum Teil umfangreiche Dateien pro Entität. Allerdings sind deren Inhalte sehr stereotyp. Wir haben deshalb einen Perl-Generator ▶

entwickelt, der alle erforderlichen Dateien aus dem Datenbankschema generiert.

Der wichtigste Grundsatz beim Design des Generators war „keep it simple“. So wurde z. B. darauf verzichtet, einen *Merge-Mechanismus* für das Generieren von manuell veränderten Dateien zu implementieren. Manuelle Änderungen wurden stattdessen durch Vererbung der generierten Klassen realisiert. Nur wenn das nicht möglich war, erfolgte das Aktualisieren der Klassen von Hand. Der Generator erzeugt alle acht Dateien pro Datenbank-Entität. Dem Anwendungsentwickler stehen damit ein komplettes Business-Objekt mit `get()`- und `set()`-Methoden für alle Attribute, sowie eine *Business-Factory* mit `find()`- und `create()`-Methoden zur Verfügung. Keine Unterstützung gibt es für Relationen: Effizientes Lesen von abhängigen Objekten kann die Performance erheblich verbessern – daher wollten wir diese Aufgabe nicht dem Generator überlassen. Der Aufwand für die Entwicklung des Generators betrug etwa drei Bearbeiterwochen.

Dass wir Entity-Beans verwendet haben, liegt vor allem an der CMP-Engine des Borland AppServers. Diese bietet vor allem bei schreibenden Datenbankzugriffen einige interessante Möglichkeiten. Hierzu gehören ein optimistisches Sperrkonzept und das Generieren von Datenbankschlüsseln in Zusammenarbeit mit „Microsoft SQL-Server“ ebenso wie *tuned Updates*.

Middleware

Der Client für unseren Applikationsserver ist keine Java-Anwendung, sondern der oben skizzierte C++-Dispatcher. Deshalb kommt das EJB-Kommunikationsprotokoll (Java RMI bzw. RMI/IIOP) für die Client/Server-Kommunikation nicht in Frage. Die einzige praktikable Alternative ist CORBA in der Version 2.2 ohne die Möglichkeit, Objekte *by Value* übergeben zu können. Wir haben deshalb über den Session-Beans eine weitere Schicht eingefügt: Hier werden die Datenstrukturen und Methodensignaturen des RMI/IIOP-Protokolls in CORBA 2.2-kompatible, einfachere Strukturen umgewandelt. Die Objekte, in denen diese Umsetzung erfolgt, nennen wir *CORBA-Wrapper*. Diese bestehen aus rein technischem Code und können deshalb komplett generiert werden.

Als Ausgangspunkt für die Generierung kann die IDL-Definition der fachlichen CORBA-Schnittstelle dienen. Generiert werden daraus (neben den technischen CORBA-Klassen, die der kommerzielle

IDL-Generator erzeugt) die Implementierungsklassen der *Wrapper*, das *Home*- und *Remote-Interface* der Session-Beans und eventuell die Rumpf-Implementierung der Session-Bean. Wir sind allerdings etwas anders vorgegangen: Wir haben als Ausgangspunkt nicht die IDL-Schnittstelle verwendet, sondern ein von uns definiertes XML-Format mit einer vergleichbaren Semantik wie IDL. Im Gegensatz zu IDL-Parsern sind XML-Parser für Perl leicht erhältlich. Die IDL-Schnittstellendefinition wird dann aus der XML-Datei generiert.

Der Verzicht auf *Object-by-Value* hatte einen angenehmen Nebeneffekt: Die Durchsatzrate erhöhte sich deutlich. Benchmarks mit großen Datentransferaten ergaben eine Durchsaterhöhung von ein bis zwei Größenordnungen. Der Overhead durch die Transformation ist dagegen vernachlässigbar, weil der Borland AppServer die Möglichkeit bietet, *Wrapper* und EJB-Container innerhalb derselben virtuellen Java-Maschine zu betreiben.

Die CORBA-*Wrapper* waren ursprünglich als technische Objekte gedacht – sie erwiesen sich aber auch in anderer Hinsicht als wertvoll. Da sie nicht unter Kontrolle des EJB-Containers stehen, kann hier Funktionalität ausgeführt werden, die innerhalb des Containers nur schwer umsetzbar oder sogar verboten ist:

- Synchronisieren von Threads und
- Verwalten von globalen Zuständen.

Das Synchronisieren von Threads erwies sich als hilfreich, um Konflikte bei konkurrierenden Zugriffen zu vermeiden. Solche Konflikte haben zwei Ursachen: Entweder sie entstehen durch „Hot Spots“ in der Datenbank, z. B. ein Zähler für die bereits verkauften Tickets. Oder sie entstehen bei dem Versuch, in mehreren Vorverkaufsstellen zum selben Zeitpunkt denselben Platz zu reservieren. Bei Veranstaltungen mit hoher Nachfrage sind solche Konflikte fast der Normalfall, sodass spezielle Vorkehrungen notwendig sind: Zum einen müssen aufwändige Datenbank-*Rollbacks* vermieden werden, die durch zu spätes Erkennen potenzieller Konflikte entstehen; zum anderen ist es wichtig, Blockaden (*Deadlocks*) in Datenbank und EJB-Container zu verhindern. Wir haben das Problem gelöst, indem wir durch eine geeignete Thread-Synchronisierung sicherstellen, dass im Server zu einem Zeitpunkt immer nur eine Reservierung pro Veranstaltung bearbeitet wird. Außerdem wird in einem Ringbuffer im Hauptspeicher des Servers

eine Liste aller in den letzten Minuten reservierten Plätze geführt. Anhand dieser Liste kann ohne einen einzigen Datenbankzugriff entschieden werden, ob ein Platz gerade reserviert wurde. Die Kombination beider Maßnahmen – Serialisierung und Ringbuffer – entschärft das Problem konkurrierender Buchungen, das gerade in Hochlast-Zeiten das System komplett blockieren kann. Beide Maßnahmen dürfen jedoch innerhalb des EJB-Containers nicht eingesetzt werden, da sie gegen die EJB-Spezifikation verstoßen.

Effizientes Lesen von Objektgeflechten

Der Schlüssel zu guter Performance liegt bei den meisten mehrschichtigen Anwendungen in der Minimierung der Datenbankzugriffe. Eine objektorientierte Implementierung eines objekt-relationalen Mappings stößt hier an ihre Grenzen: Wenn jedes persistente Objekt seine Synchronisierung mit der Datenbank selbst übernimmt, so wie es der objektorientierten Philosophie entspricht, entsteht schnell eine Inflation von Datenbankzugriffen. Die EJB-Spezifikation, vor allem in der Version 1.1, verleitet genau zu einem solchen Ansatz: In den `find()`-Methoden werden üblicherweise nur die Primärschlüssel der Ergebnismenge gelesen. Beim ersten Zugriff auf die Entitäten werden die übrigen Daten nachgeladen. Bei der Verwendung von *Bean-Managed Persistence (BMP)* ist diese ineffiziente Zugriffsstrategie kaum zu vermeiden, deshalb ist BMP immer inhärent langsam. Gute CMP-Engines lassen sich so konfigurieren, dass in den `find()`-Methoden alle Objektdaten geladen werden.

Noch problematischer ist der Umgang mit abhängigen Objekten. Für den Anwendungsprogrammierer ist es einfacher, wenn alle abhängigen Objekte automatisch geladen werden. Hierfür sind zwei Ladestrategien gebräuchlich:

- Entweder werden alle abhängigen Objekte geladen, sobald der erste Zugriff auf eines davon erfolgt (*eager loading*), oder
- die abhängigen Objekte werden jeweils einzeln beim ersten Zugriff geladen (*lazy loading*).

Für uns waren beide Strategien unbrauchbar. Wir hatten häufig mit dem Fall zu tun, dass in einer Transaktion *n* Vater-Objekte geladen wurden, die jeweils *m* abhängige Kind-Objekte haben. Wie viele Datenbankzugriffe sind dann nötig, um alle Objekte zu laden?



Die n Vater-Objekte werden mit einem einzigen Datenbankzugriff geladen (die CMP-Engine von Borland bietet diese Möglichkeit). Verwendet man die Lade-strategie *eager loading*, erfolgen danach noch n weitere Zugriffe. Mit der Lade-strategie *lazy loading* hingegen erfolgen noch $n*m$ weitere Zugriffe.

Eager loading ist also deutlich überlegen. Wir waren aber damit noch nicht zufrieden, da n bei uns oft im Bereich zwischen 100 und 1.000 lag. Unser Ziel war es, mit zwei Datenbankzugriffen auszukommen: Zunächst werden alle Vater-Objekte geladen, dann alle Kind-Objekte. Ein einzelner Zugriff ist ineffizient, weil dabei unnötig viele Daten aus der Datenbank gelesen werden.

Die Umsetzung ist nicht schwierig: Zunächst werden die n Vaterobjekte mit einer üblichen `find()`-Methode geladen. Die Primärschlüssel der Treffer-Objekte werden in einem *Array* gespeichert. Dann wird eine `find()`-Methode der Kind-Objekte mit dem Schlüssel-*Array* als Argument aufgerufen (die Primärschlüssel der Vater-Objekte sind die Fremdschlüssel in der Vater/Kind-Relation). Zuletzt werden alle Referenzen auf die Kind-Objekte in den Listen der Vater-Objekte eingefügt. Diese Lade-Algorithmen sind in den `find()`-Methoden der *Business-Factories* enthalten, sie sind jedoch nicht generiert. Für die Umsetzung mit EJB ist es erforderlich, dass die CMP-Engine auch *Arrays* als Argumente von `find()`-Methoden akzeptiert.

Wie oben angedeutet, haben wir in vielen performance-kritischen Methoden die Entity-Beans komplett umgangen und JDBC-Aufrufe direkt in den *Business-Factories* ausgeführt. Das schnelle Laden von Objektgeflechten erfolgte dabei analog wie mit Entity-Beans.

Betrieb

Der Euroticket-Anwendungsserver läuft auf einem mehrfach redundant ausgeleg-

ten Server-Cluster mit Dual-Pentium-III-Xeon-Prozessoren unter Windows NT. Der Datenbankserver besteht aus einer Cluster-Konfiguration mit „MS SQL-Server 7.0“ und nutzt vergleichbare Hardware wie der Anwendungsserver.

Das Hardware-Cluster des Anwendungsservers dient nur der Ausfallsicherheit. Es findet zur Zeit keine Lastverteilung der Anfragen auf mehrere Server statt, obwohl die Möglichkeit dafür prinzipiell besteht. Der Durchsatz des Systems ist in dieser Konfiguration vollkommen ausreichend; eine Lastverteilung würde nur unnötige Hardware- und Lizenzkosten verursachen.

Fazit

Wir haben ein System mit extremen Anforderungen an Durchsatz, Anwenderzahl und Verfügbarkeit von C++ auf eine J2EE-Plattform migriert. Lasttests ergaben eine Verbesserung der Performance um das Dreifache gegenüber dem Altsystem. Der Server läuft stabil und mit geringem Wartungsaufwand. Die Bewährungsprobe war das Weihnachtsgeschäft 2001: Trotz deutlich höherem Umsatz gegenüber dem Vorjahr lag die Auslastung des neuen Anwendungsservers nur bei ca. 30 %. Das Altsystem war dagegen bereits im Jahr 2000 an seiner äußersten Grenze.

Diese erhebliche Performance-Steigerung wurde durch das Optimieren und Minimieren von Datenbankzugriffen erreicht. Zudem konnte mit einer geeigneten Serialisierung von Client-Aufrufen das Risiko von Datenbank-Blockaden vermindert werden.

Der Applikationsserver und die Java-Laufzeitumgebung erwiesen sich als zuverlässige Systemplattform. Durch die Verwendung des „Hot Spot“-Compilers aus dem Java SDK 1.3 war die Java-Performance gut, obwohl keine Hardware-Skalierung existierte.

Die EJB-Technologie hat sich in vielerlei Hinsicht als Einschränkung erwiesen; es

mussten einige aufwändige Maßnahmen durchgeführt werden, vor allem wegen der Performance-Probleme der Entity-Beans. Wir haben großen Wert darauf gelegt, dass diese Maßnahmen nicht die gesamte Architektur beeinträchtigen. Vor allem sollten sie in einer technischen Schicht verborgen und damit für den Anwendungsprogrammierer transparent sein. Die Systemarchitektur wurde dadurch zwar relativ komplex. Der Entwicklungsaufwand hielt sich jedoch durch den Einsatz einfacher, aber äußerst leistungsfähiger Generatoren in Grenzen. Ohne die hervorragende CMP-Engine des Borland AppServers hätte uns die EJB-Technologie keinen Nutzen gebracht – wir hätten dann eine Java-Anwendung mit CORBA vorgezogen.

Ende 2001 hat die CTS Eventim AG begonnen, ein eigenes Entwicklungsteam aufzubauen. Die Mitarbeiter wurden zeitweise in das sd&m-Team und den Projektablauf integriert, um den Transfer von Wissen in den einzelnen Phasen der Konzeption, Realisierung und Integration zu ermöglichen. ■

Literatur

- [Gam96] E. Gamma u.a., Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software, Addison-Wesley, 1996
- [Mon00] R. Monson-Haefel, Enterprise JavaBeans, O'Reilly, 2000
- [Sha98] Y-P. Shan; R. H. Earle, Enterprise Computing with Objects – From Client/Server Environments to the Internet, Addison-Wesley, 1998
- [Sun] EJB-Spezifikation 2.0, Proposed Final Draft 2, Sun Microsystems
- [Val99] T. Valesky, Enterprise JavaBeans – Developing Component-Based Distributed Applications, O'Reilly, 1999