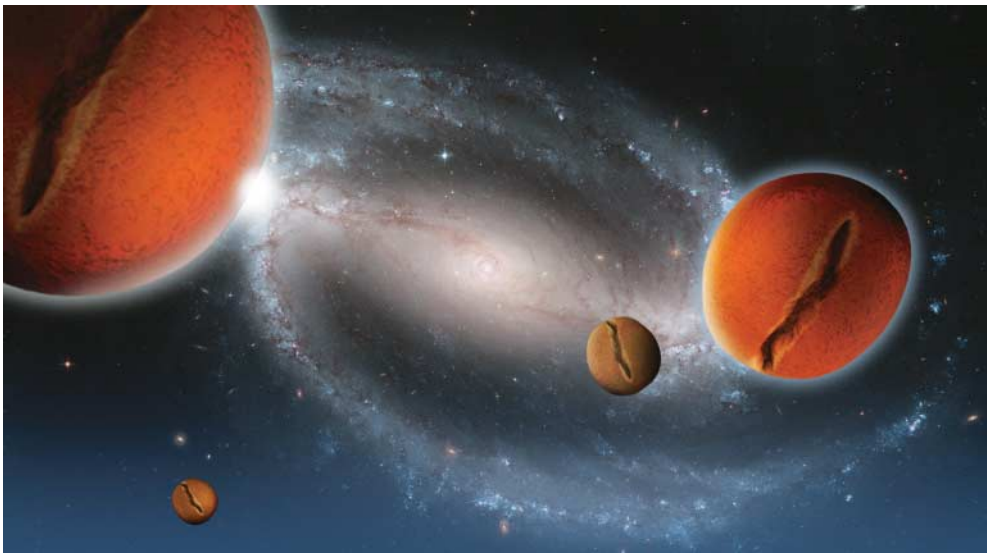

Bessere Software mit Spaces

Ohne Raum und Zeit

Bernhard Angerer

Obwohl seit Jahren bekannt, sind Spaces noch immer eine Nischentechnik. Die Spaces-Community hingegen spricht von einem zukunftsweisenden und nachhaltigen Ansatz. Diese Meinung teilt die Grid-Gemeinde, die das Konzept in einigen Projekten nutzbringend umsetzt.

Der Zeitreisende, von dem Michael Stal in seinem Artikel „Aus der Zukunft“ [1] 2006 berichtete, wäre sicherlich enttäuscht, dass sich in der aus seiner Sicht „prähistorischen Informatik“ auch gute zweieinhalb Jahre später nicht viel geändert hat. Noch immer schätzen Marktanalysten und IT-Experten, dass nur 20 Prozent aller Softwareprojekte zu einem erfolgreichen Abschluss kommen. Die restlichen 80 Prozent erleiden im schlimmsten Fall einen vorzeitigen Abbruch, der Normalfall sind Projekte mit Zeit- und Budgetüberschreitung oder unvollständigem Auslieferungsvolumen. Die Frage lautet daher nach wie vor: Wie lässt sich bessere Software effizient herstellen?

Eine Antwort liefert die Spaces-Community, die zur Verbesserung der Gesamtlage einen Ansatz vorschlägt, den man als holistisch bezeichnen kann, da er infrastrukturseitige Herausforderungen mit einem anwendungsseitigen Programmierparadigma vereint. David Gelernter gilt als der Erfinder von Spaces, seitdem er Ende der 80er-Jahre zusammen mit Nicholas Carriero in einem System namens Linda die Grundlagen gelegt hat. Die Signifikanz und zukunftsweisende Bedeutung dieses Ansatzes hat man schon damals erkannt, erst die letzten Jahre brachten jedoch Implementierungen in Produktionssystemen größeren Ausmaßes.

Die Definition von Spaces ist nicht einfach, da sich keine bekannte Schublade dafür öffnen lässt. Laut Josef Ottinger (siehe „Onlinequellen“ [b]) verbirgt sich dahinter eine Art Speicherabbild von Client/Server-Anwendungen, ein Netz (Grid), in dem Daten existieren, die gelesen und geschrieben werden können. Als Infrastruktur dient den JavaSpaces ein „Distributed Virtual Shared Memory“ zur Kommunikation und Koordination zwischen unterschiedlichen Programmen.

Entkopplung in drei Freiheitsgraden

Anders als bei gewöhnlichem Shared Memory kommt in diesem Fall Verteilung hinzu. Auf den ersten (und zweiten Blick) zu trivial, um dem viel Aufmerksamkeit zu widmen (die Genialität liegt eben oft in der Einfachheit). Bei genauerem Hinsehen ergeben sich jedoch weitgehend neue Muster auf der Anwendungs- sowie auf der Infrastrukturseite.

Zeitreisende müssen mit Raum-, Zeit- und Referenzentkopplung zurechtkommen – so auch manche Programme. Eine anwendungsseitige „Blackboard“-Kommunikation ermöglicht das, indem sie ihnen erlaubt, Daten einfach in „den Space“ zu schreiben. Andere Programme, die Interesse an diesen Daten angemeldet haben, erhalten eine Benachrichtigung und können aus dem Space beliebig lesen (blackboard metaphore). Das bedeutet, dass jegliche Interaktion in einem Space-basierten System triadisch abläuft, wodurch sich ein neues Paradigma ergibt. Peers kommunizieren miteinander
– ohne etwas voneinander zu wissen (Referenzentkopplung),
– ohne sich am selben Ort (Maschine) befinden zu müssen (Raumentkopplung) und
– ohne zeitliche Abstimmung (Zeitentkopplung).

Diese Entkopplung in allen drei Freiheitsgraden ist ein Idealzustand aus der Sicht von Komponentendesignern (weitere Online-Informationen zu in diesem Artikel angesprochenen Themen sind über die *iX*-Links erreichbar). Je geringer die Verflechtungen und starren Abhängigkeiten, desto besser in Bezug auf den Aufwand für Entwicklung und Debugging sowie den Grad der Wiederverwendung. Ein noch entscheidenderer Punkt ist jedoch die implizit stattfindende Koordination.

Ein Programm fordert Informationen an, indem es ein *Requestobjekt* in den Space stellt. An anderer Stelle erzeugt dieser Vorgang eine Nachricht, und nachdem das Programm entsprechende Aktionen vorgenommen hat, stellt es die Antworten wiederum in den Space. Steht ein benötigtes Resultat in Form eines Space-Eintrags noch nicht zur Verfü-

Listing 1: Spaces-basierte API

```
Entry read(Entry tmpl, Transaction txn, long timeout)
Entry write(Entry entry, Transaction txn, long lease)
Entry take(Entry tmpl, Transaction txn, long timeout)
EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listener, long lease, MarshalledObject handback)
```

Die JavaSpaces-Spezifikation selbst gibt ein Beispiel für eine Spaces-basierte API.

gung, wartet der zuständige Agent auf dessen Eintreffen. Der Entwickler bleibt von typischen Workflow-Problemen (bestimmte Aufgaben müssen vor anderen erledigt werden) verschont. Die Kommunikation und Koordination löst sich geradezu auf und beschränkt sich auf Schreib- und Leseoperation in den und aus dem Space. Des Weiteren ist es vollkommen gleichgültig, ob ein Agent oder Hunderte auf das Eintreffen von Nachrichten warten. Die Komplexität für den Entwickler steigt nicht in Abhängigkeit von der Teilnehmeranzahl.

Einfachheit hat hohen Stellenwert

Beim Entwurf der Spaces-API setzten die Entwickler auf radikale Einfachheit und Verständlichkeit. Im Wesentlichen besteht sie aus vier Methoden (siehe Listing) zum Schreiben, Lesen, Lesen und gleichzeitigen Löschen (Take) sowie einem Event-Handler, den das Eintreffen von Objekten (Entries) anstößt. Die folgenden drei Techniken bestimmen das Space-basierte Programmierparadigma:

– **Template Matching**: Einer Read-Operation wird eine Vorlage (eine Instanz eines Objekts) übergeben. Objektattribute, die nicht initialisiert sind, fungieren als Wildcards, konkrete Attributwerte schränken ein und bestimmen so das zurückgelieferte Ergebnis. Auf diese Weise lässt sich einfach und flexibel in einem Objektgraphen navigieren.

– **Leases**: Eine Leasetime spezifiziert eine Ablaufzeit und begrenzt so den Lebenszyklus eines Objekts im Space. Die Verantwortung, Objekte zwecks Schonung der Ressourcen aus dem Space zu löschen, lässt sich so an die Space Engine delegieren (Leases ist ein Jini-Mechanismus; alternativ ist eine Garbage Collection denkbar).

– **Timeouts**: Im Prinzip sind Timeouts nichts Neues, sollen hier aber nicht unerwähnt bleiben, da sie für die Workflow-Eigenschaften der Spaces-API maßgeblich sind. Ein Kommunikationsteilnehmer kann so dazu gebracht werden, auf das Eintreffen eines gewissen Ereignisses (eines Objekts) zu warten. Warten viele Teilnehmer auf von einem einzelnen Teilnehmer erzeugte Ereignisse, spricht man von dem Master/Worker Pattern.

Im Vergleich zu traditionellen Methoden kann man Folgendes zusammenfassen:

– **Write + Take**: entspricht Parallel Processing (Remote Procedure Call, RPC)

– **Write + Read**: entspricht Caching (Put, Get)

– **Write + Notify**: entspricht Messaging (Publish, Subscribe) Distributed Virtual Shared Memory

Zwei Schlagworte – In-Memory und Replication – spielen eine entscheidende Rolle dabei, wie die Space-Engine die Illusion eines verteilten, gemeinsam benutzten Gedächtnisses erzeugt.

An mehreren Orten zugleich

Ein Zeitreisender würde in einem Space-basierten Universum logisch nur einmal existieren, physikalisch jedoch an mehreren Orten. Der Space sorgt dafür, dass die Anwendung jedes Objekt als singulär erkennt, physikalisch gesehen speichert sie die Objekte jedoch auf mehreren Rechnern. Klingt einfach, ist in der Praxis jedoch schwierig, da eine Anwendung mit Ressourcen sparsam umgehen muss und daher nicht alle Objekte an alle Teilnehmer (Nodes) verteilen kann. Die Replikationsprotokolle müssen daher „intelligent“ sein und Objekte in Abhängigkeit von deren Verwendung propagieren oder partitionieren. Das heißt, ein Objekt wird nur für einen Node oder eine Anwendung repliziert, wenn diese dessen Daten tatsächlich konsumieren.

Daraus ergibt sich ein entscheidender Unterschied zu traditionellen Clustering-Ansätzen, die meist versuchen, eine Virtualisierung von Ressourcen auf transparente Art und Weise zu erreichen. Ein Space-Cluster ist hingegen „Application aware“. Die Kehrseite der Medaille ist naturgemäß der Aufwand für ein Redesign, der anfällt, wenn man ein bestehendes System umstellt (weshalb man in der Praxis fast ausschließlich neue Module Space-basiert entwickelt und Bestehendes nur zum Teil transformiert).

Skeptiker wenden üblicherweise ein, dass ein Space-System nicht skaliert, wenn viele Rechner eingebunden sind und viele Teilnehmer ein und dasselbe Objekt gleichzeitig verändern (Locking-Mechanismen sind ja bekanntlich teuer). Eine Lösung hierfür ist, nicht auf pessimistische Art einen Lock einzuführen, sondern mit Optimismus einen internen Versionszähler mitzuführen (pessimistic/optimistic locking). Kommt es zu einem Konflikt, erhält der Client eine Information über die fehlgeschlagene Write-Operation.

Intern arbeitet die Space Engine mit schon aus der Datenbankwelt bekannten Primary-Copy-Verfahren. Dabei muss sie nur ein Objekt bei einer schreibenden Operation verändern (die Primary Copy). Alle anderen Replikate aktualisiert sie anschließend asynchron, das heißt, erst nach dem Ende der Transaktion. Damit bleibt die Zeit, in der der schreibende Client blockiert ist, minimal (mit dem Kompromiss, dass sich die Anpassung der Replikate verzögert).

Die Suche nach dem Gral

Dies vermeidet auf effiziente Art einen Flaschenhals, da die Speicherung der Primärkopien unterschiedlicher Objekte an verschiedenen Knoten erfolgt. Somit entsteht ein „virtueller Server“, der keine zentrale Steuerung besitzt (peer to peer virtualization). Darüber hinaus wird im Anwendungsdesign mit mehreren logischen Spaces (daher der Plural im Namen) gearbeitet, was zu einer zusätzlichen Segmentierung führt

iX-TRACT

- JavaSpaces – ein Forschungsprojekt der Sun Labs – sollen durch ihre Architektur die Komplexität verteilter Anwendungen deutlich reduzieren.
- Die Kommunikation und Koordination zwischen verschiedenen Programmen läuft über Agenten in einem verteilten virtuellen Speicher.
- Spaces-basierte Anwendungen findet man heute vornehmlich im Grid-Umfeld, da sie gegenüber herkömmlichen Clustering-Ansätzen deutliche Vorteile bezüglich der Datenhaltung bieten.

und die effiziente Abwicklung verteilter Transaktionen begünstigt.

Effiziente Transaktionsprotokolle sind bis heute ein Forschungsthema, verbirgt sich doch nach allgemeiner Meinung ein heiliger Gral der Softwareentwicklung hinter dem Thema. Jedoch hält vielerorts Pragmatismus Einzug in diesbezügliche Initiativen, nachdem man in den vergangenen Jahrzehnten viel über Abstraktionsniveaus gelernt hat. Joel Spolsky hat den Begriff der „Leaky Abstraction" [h] in seinem „Law of Leaky Abstractions" auf anschauliche Weise geprägt. Demgemäß lautet dessen Leitsatz: „All non-trivial abstractions, to some degree, are leaky."

Demzufolge sind sich Entwickler des Trade-off-Spiels (meist Performance gegen Funktionsvielfalt/Komfort) bewusst und gehen willentlich einen Kompromiss ein, indem sie die Problematik explizit im System berücksichtigen und nicht versuchen, mit allen Mitteln (fauler Kompromiss) zu abstrahieren. In Bezug auf das oben angeschnittene Thema des „Concurrent Write Resolution" bedeutet das eine Verlagerung in die Konfiguration. Der Systemdesigner informiert demnach den Space über unterschiedliche Anforderungen explizit und deklarativ. So kann er beispielsweise Objekte, die unter Write-most- oder Read-most-Einfluss stehen, dahingehend auszeichnen. Unterschiedliche Implementierungen der Engine liefern dann optimale Ergebnisse.

Um transaktionale Sicherheit zu erreichen, erfolgt die Speicherung nicht auf der Festplatte oder in der Datenbank, sondern die Daten werden speicherresident auf andere Nodes repliziert. Bei einem Ausfall weiß die Space Engine um die redundanten Daten in anderen Hauptspeichern und kann die Gesamtkonsistenz wahren (in-memory fail-over).

Entscheidend ist, dass man so mehrere Fliegen mit einer Klappe schlägt, da man ja durch das verteilte Anwendungsszenario die Daten sowieso an verschiedenen Orten zur Verfügung stellen muss (was auch auf Performance- und Verfügbarkeitsskalierung zutrifft). Die Kommunikation mit der Datenbank ist in einem Space-System üblicherweise asynchron. So überrascht es nicht, dass man mit diesem Ansatz in völlig neue Performance-Kategorien vorstößt und Gartner das Akronym XTP (Extreme Transaction Processing) für derartige Systeme prägte.

Weniger Komplexität

Singularitäten sind wohl der Stoff, aus dem Zeitreisen gemacht sind. Betrachtet man die eben beschriebene dreidimensionale, anwendungsseitige Entkopplung im Zusammenspiel mit der infrastrukturseitigen Fusion von Kommunikation und Replikation, ergibt sich das holistische Bild einer Space-basierten Architektur, deren Ansatz sich durch eine Besonderheit auszeichnet: Üblicherweise wird eine höhere Abstraktionsebene ausschließlich durch eine Softwareschicht erreicht, die alle Problemstellungen dieser Ebene löst. Kein neuer Ansatz reduziert die existierende Komplexität, sondern es wird lediglich der Ort der Implementierung verschoben.

Bei einer Space-basierten Architektur jedoch kommt eine tatsächliche Komplexitätsreduktion hinzu. Durch die Zusammenlegung des Kommunikations-, Koordinations- und Replikationsmechanismus kann man geradezu von einer „technischen Singularität" (Problemraumfaltung) sprechen. Ein und dieselbe Infrastruktur erlaubt Parallelverarbeitung, Nebenläufigkeit auf Einprozessor-Rechnern sowie netzweite Kommunikation. Die folgenden Eigenschaften sind in einem Space-basierten System inhärent gegeben, greifen gegenseitig ineinander und definieren ein völlig neues Designparadigma für den Anwendungsprogrammierer:

– **Stateful Programming**: Klassische verteilte Systeme gehorchen dem Stateless Programming. Eine Anwendung arbeitet jede Anfrage unabhängig von der vorhergehenden ab. Es gibt kein „Gedächtnis" (distributed shared state/memory). Der Space hebt diesen Umstand auf und erzeugt die Illusion einer einzelnen (virtuellen) Maschine.

– **Deferred Execution**: Ein Space-basiertes System verarbeitet alle Aufrufe asynchron. Dadurch eröffnen sich völlig neue Möglichkeiten in der zeitlichen Abwicklung und Priorisierung von Aufgaben. Dinge lassen sich delegieren und entkoppeln. Somit ist es ein Leichtes, die Antwortzeiten für eine initiale Antwort gering zu halten und gleichzeitig parallel im Hintergrund an zusätzlichen Aufgaben zu arbeiten (beispielsweise beim Einsatz von Ajax).

– **Agent Based**: In der Regel sind Agents feingranulare Komponenten, die ihren eigenen Lebenszyklus besitzen und bestimmte Aufgaben erledigen. Durch die asynchrone, entkoppelte Architektur „mutiert" in einem Space-basierten System jede Komponente zu einem Agenten. Somit kann die Anwendung flexibel „zusammengestellt" werden (service orchestration) und Komponenten lassen sich voneinander losgekoppelt testen.

– **Monitoring**: Die nachrichtenbasierte Kommunikation und Koordination in einem Space-System geben dem Entwickler auf Anwendungsebene Einsicht in den aktuellen Ablauf, ohne die Gesamt-Performance nachhaltig zu beeinflussen. Monitoring und Controlling auf Anwendungsebene erledigt ein zusätzlicher Agent.

Onlinequellen

[a] Auszug aus: Michael Stal; Aus der Zukunft; Auf dem Weg zu besserer Software; *iX* 2/06, S. 38
www.heise.de/ix/artikel/2006/02/038/

[b] Joseph Ottinger; Using JavaSpaces; TheServerSide.Com 2007
www.theserverside.com/tt/knowledgecenter-gs/knowledgecenter-gs.tss?l=UsingJavaSpaces

[c] IBM; Scaling the Grid; Case Study 2006
www-03.ibm.com/servers/deepcomputing/cod/pdf/fsscasestudy.pdf

[d] David Gelernter; The Second Coming – A Manifesto; Edge.org 2004
www.edge.org/3rd_culture/gelernter/gelernter_p1.html

[e] Bernhard Angerer; Space based Programming; O'Reilly 2003; ONJava.com
www.onjava.com/pub/a/onjava/2003/03/19/java_spaces.html

[f] Grids Gets Transactional; GridToday 2006
www.gridtoday.com/grid/1150800.html

[g] GigaSpaces Releases eXtreme Application Platform; GridToday 2007
www.gridtoday.com/grid/1611358.html

[h] Joel Spolsky; The Law of Leaky Abstractions; 2002
www.joelonsoftware.com/articles/LeakyAbstractions.html

[i] Robert Tolksdorf, Lyndon Nixon, Franziska Liebsch, Duc Minh Nguyen, Elena Paslaru Bontas; Freie Universität Berlin 2004; Semantic Web Spaces
ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-04-11.pdf

– **Scaling**: Alle Aspekte der Skalierung und Verteilung werden aus dem Anwendungscode entfernt und in die Konfiguration des Space verschoben. Dies beinhaltet sowohl Performance- als auch Verfügbarkeitsskalierung. Multi-Core-Systeme (von aktueller Middleware oft nur durch explizite Multi-Threaded-Programmierung unterstützt) werden in idealer Weise ausgenutzt, da alle Agenten vollkommen asynchron laufen und somit ihre Ausführung parallel erfolgen kann.
– **Reality Check**: Unternimmt der Zeitreisende einen Reality Check im Jahr 2008, stößt er auf erste Space-basierte Systeme, die in nennenswerten Größenordnungen in Produktion sind [b]. In den meisten Fällen handelt es sich um Anwendungen, deren Komplexität die Möglichkeiten der etablierten Anwendungsserver signifikant übersteigt, arbeiten jene doch mit getrenntem Clustering für die Anwendungs-, Daten- und Nachrichtenschicht sowie separaten Modulen für paralleles Rechnen.

So mancher sieht die Zeit der Spaces bereits kommen. Allerdings zeigt der Reality Check eindeutig, dass sich das Thema ausschließlich im Umfeld des Grid Computing bewegt. Keine Rede von einer breiteren Diskussion (abgesehen vom universitären Umfeld), die Spaces als fundamentalen Ansatz erkennt und ihn in Bezug auf die Herausforderungen des Software Engineering im Allgemeinen setzt. Lediglich Bill Joys und David Gelernters Proklamationen der Neunziger stellen das Thema nach wie vor in einen größeren Zusammenhang [2]. So gibt es auch keine Organisation, keinen Titel (etwa wie die Object Management Group mit CORBA), der die internationalen Aktionen bündeln würde. Das Jini/JavaSpaces-Forschungsprojekt der Sun Labs hat diesbezüglich nie Anspruch erhoben. Tatsächlich hat es für einige Verwirrung gesorgt, da die Jini Community auf einem anderen Abstraktionsniveau angesiedelt ist als die Spaces Community, was naturgemäß zu Dissonanzen geführt hat.

Back to the Future

Wird der Zeitreisende einen Neuanfang initiieren können? In manchen Bereichen vielleicht, eher kann man jedoch mit Quantensprüngen durch die Fusion mit neuen Bereichen rechnen. Hier sind etwa die Triple Spaces zu nennen, bei denen es um die Verschmelzung des semiotischen Dreiecks (triadische Relationen) mit Spaces geht. Die gesamte Middleware-Branche betreffend zeichnet sich eher eine behutsame, inkrementelle Annäherung ab.

Space-basierte Techniken und Programmiermuster werden nach und nach in etablierte Anwendungsserver übernommen. Der Space-Effekt ist jedoch nur mit einer holistischen Sichtweise realisierbar. David Gelernter jedenfalls unternimmt nach wie vor Zeitreisen und regt in seinem Manifest zu Diskussionen über die Zukunft an [d]. (ka)

DR. BERNHARD ANGERER

ist Support Engineer bei GigaSpaces Technologies in New York.

Literatur

[1] Michael Stal; Aus der Zukunft; Auf dem Weg zu besserer Software; *iX* 2/06, S. 38; (Auszug unter www.heise.de/ix/artikel/2006/02/038/)
[2] David Gelernter; Mirror Worlds: or the Day Software Puts the Universe in a Shoebox … How It Will Happen and What It Will Mean; Oxford University Press 1992

iX-Link ix0808118