Verteilte Objekte

von
Andreas Joebges, Malte Teubner und Joachim Schulz

1 Einleitung

2 Verteilte Systeme

2.1 Historische Einleitung
2.2 Bedeutung von verteilte Systeme
2.3 Vorteile verteilter Systeme
2.4 Nachteile verteilter Systeme

3 Verteilte Anwendungen

3.1 Entwicklung von Verteilten Anwendungen
3.2 Die objekt-orientierte Verteiltheit

3.2.1 Gründe für oo-Verteiltheit
3.2.2 Bisherige Umsetzung
3.2.3 Vorteile oo-Verteiltheit

4 EMERALD

5 Die OO – Sprache BETA

5.1 Einleitung
5.2 Das BETA Objekt Modell
5.3 Quellcode Beispiele für einfache Objekte
5.4 Verteiltes Programmieren in BETA
5.5 BETA Quellcode Beispiele für verteilte Objekte
5.6 Zusammenfassung

6 Fallstudie „Content Management System“

6.1 Einleitung
6.2 Verteile Objekte in einem CMS
6.3 Verteilte Objekte im CMS mit BETA
6.4 BETA Quellcode Bespiel für verteilte Objekte im CMS

7 Zusammenfassung

8 Literatur

8.1 Einleitung, Verteilte Systeme, Verteilte Anwendung
8.2 EMERALD
8.3 BETA

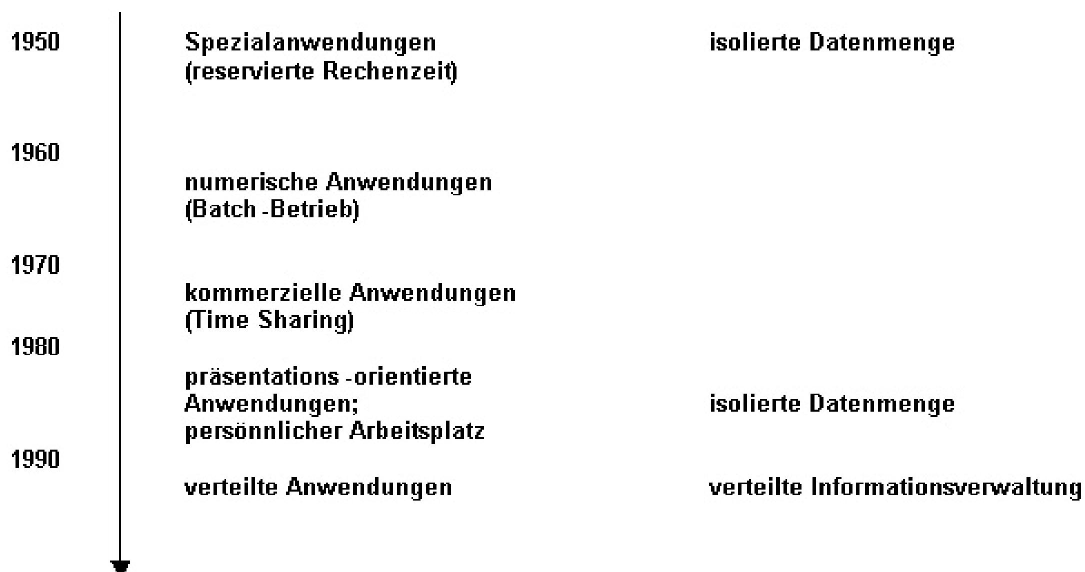
1 Einleitung

Dieser Teil des Readers behandelt das Thema „verteilte Objekte", doch um dieses Thema verständlich zu machen, werden wir erst einmal einen kleinen Einblick in verteilte Systeme geben. Nach einem kleinen Rundumblick werden wir dann auf zwei objekt-orientierte Sprachen eingehen, welche die Verteiltheit mittels Objekt-Orientiertheit realisieren. Es handelt sich dabei um die Sprachen EMERALD und BETA. Genaueres dazu später.

2 Verteilte Systeme

2.1 Historische Einleitung

Früher, so um 1950, waren die Computer sehr groß. Die Möglichkeit die Daten von den Rechnern zu versenden, war damals noch nicht gegeben. Wenn man nun etwas zu berechnen hatte, konnte man sich auf den Rechnern Rechenzeit reservieren. An eigene Rechner war noch nicht zu denken.
Die Rechner wurden damals meist für mathematische Berechnungen verwendet, so dass der Einzelne nicht so viel mit Computern zu tun hatte.
Doch nach 1970 begann die kommerzielle Nutzung der Computertechnik. Es war noch nicht so ausgereift allerdings konnten jetzt schon mehrer Personen die Ressourcen nutzen (Time-Sharing).
Der Schritt zum persönlichen Arbeitsplatz war dann nicht mehr weit. Allerdings waren die Daten immer noch auf den jeweiligen Rechner isoliert. Doch man kam auf die Idee, dass wenn man alle Computer mit einander vernetzt eine viel bessere Nutzung der Ressourcen erreicht und eine bessere Verbreitung von Informationen. Folgenden Abbildung soll das Geschrieben noch einmal verdeutlichen. Es handelt sich jeweils um den Durchbruch der jeweiligen Technologie in der „breiten Masse" und nicht um die Erfindung. Verteilte Anwendungen wurden auch schon wesentlich früher entwickelt (BETA 1975).



Zukunft:

- Vernetzung von *heterogenen* Rechnern, (verschiedene Plattformen)
- Anwendungen zur gemeinsamen Nutzung von Ressourcen,
- Kommunikation von Informationen(d.h. besserer Informationsfluss) und zur Koordination von Aktivitäten.

Beispiele: Flugbuchungssysteme, WorldWideWeb

2.2 Bedeutung von verteilte Systeme

Die Faktoren, die zu einer wachsenden Bedeutung verteilter Systeme geführt haben, sind schnell erklärt.

- Kosten für Rechner sind kontinuierlich gefallen.
- Verfügbarkeit von Netzwerktechnologien mit hoher Bandbreite

Heutzutage kann sich fast jeder Mensch(in einem Industriestaat) einen gut ausgerüsteten Computer leisten. Allein dadurch gibt es schon eine Unzahl von Computern die im Umlauf sind. Durch neue Technologien, wie z.B. ADSL, Satteliten, oder auch die Steckdose, ist die Bandbreiten heute kein Thema mehr.
Durch diese technologischen Fortschritte und durch die ständig „wachsende Kooperation und gemeinsame Nutzung von Informationen durch geographisch verteilte Benutzer; bedingt durch Globalisierung der Märkte und auch der Unternehmen"(Zitat des Skriptes), sind verteilte Systeme einfach nicht mehr weg zu denken.

„Ein verteiltes System ist charakterisierbar durch eine enge, direkte Kopplung von Anwendungen, die auf *heterogenen* Plattformen in einer vernetzten Umgebung ablaufen."

Bei der Verwendung von Verteilten Systemen sollte man immer auf die Konsistenz der Information achten (z.B. auch nach Absturz eines Teiles des Systems). Sind Daten im ganzen Netzt verteilt so muss gewährleistet sein, dass sie nicht verloren gehen, auch wenn ein Teil ausfällt. Beim Sharing von Informationen muss auf die Eindeutigkeit achtgegeben werden. Es sollten auf keinen Fall verschiedene Versionen eines Objektes unterwegs sein. Ein weiteres wichtiges Kriterium ist eine vernünftige Antwortzeiten auf Anfragen. Was nützt eine verteilte Umgebung, wenn man auf ein Ergebnis lange warten muss.
Das System sollte die Fähigkeit haben, falsche Eingaben von Geräten und Benutzern zu tolerieren. Hat es das nicht so kann mit einer häufigen Ausfallquote bei einigen Knoten gerechnet werden.
Es sollte möglich sein einzelne Systemteile autonom zu verwalten. Da es theoretisch möglich ist ein unendlichgroßes Netzwerk aufzubauen, müssen einzelne Teilkomponenten auch während der Laufzeit gewartet werden können (bzw. neu eingespielt, herausgenommen werden).

2.3 Vorteile von verteilten Systemen

Ein verteiltes System, wenn es die oben angesprochenen Regeln einhält (Bandbreite etc.), führt zu bessere Performanz und Flexibilität von Arbeitsplatzrechnern. Man erreicht durch die Verteilung einen Lastenausgleich, d.h. z.B. rechenintensive Prozesse können auf anderen Rechnern betrieben werden oder auf verschiedene Rechner verteilt werden. Dadurch erhält man eine wesentlich höhere Leistung.
Darüber hinaus können Ressourcen von allen Netzwerkinternen genutzt werden. Wie es zum Beispiel bei Reisebüros der Fall ist. Hier werden Daten(Flug, Hotelzimmer) gemeinsam genutzt. Sowohl vom Reisebüro A als auch vom Reisebüro B. Aber nicht nur Daten können gemeinsam benutzt werden, auch Geräte (z.B. Hardware [Drucker]) fallen unter die gemeinsame Nutzung von Ressourcen.
Durch die Vernetzung kann man allerdings nicht nur gemeinsame Mittel nutzen, eine Netzwerk interne Kommunikation wird möglich (z.B. E-Mail im WWW).
Der Benutzer allerdings merkt nichts von dieser Verteiltheit, da seine Programme ihm einfach ein großes System vorstellen. Er hat sich nicht um das Ansprechen der einzelnen Knoten zu kümmern. Man spricht von einer hohen Transparenz, wenn der User nur ein Ganzes zu Gesicht bekommt. Je höher die Transparenz, um so einfacher für den Benutzer.

2.4 Nachteile von verteilten Systemen

Bei einem verteilten System ist man immer abhängig von der Leistung und der Zuverlässigkeit des Netzes. Je kleiner die Bandbreite um so müheseliger (zeitaufwendiger) werden die verteilten Aktionen, die man in dem System ausführen will.

Durch die starke Vernetzung entsteht ein erhöhtes Sicherheitsrisiko, da Eindringlinge mehr Angriffspunkte an das System haben.
Die Software für ein großes Netz ist sehr komplex. Bei der Entwicklung müssen viele Punkte bedacht werden. Allein die Berücksichtigung der Kommunikation ist schon ein großes Stück Arbeit. Noch schwerer wird es eine verteilten Anwendung zu testen ohne den Rest des Systems zu beeinflussen.
Die Transparenz wird von uns als Vor- und Nachteil gesehen. Denn fällt nun ein Teil des Systems aus, merkt der Benutzer das etwas nicht läuft. Er kann allerdings nichts dagegen machen, da der Fehler nicht bei ihm liegt, sondern ein entfernter Knoten streikt. In diesem Fall ist die Transparenz sehr nachteilig.

Dazu ein Zitat nach Lamport[1]:
„ein verteiltes System ist ein System, in dem ich durch den Ausfall einer Komponente, von der ich bisher nicht wusste, in meiner Arbeit beeinträchtigt werde"

Urton:
„A distributed system is a system that prevents you from getting the work done when a computer you never heard of breaks down."

3 Verteilte Anwendungen

3.1 Entwicklung von verteilten Anwendungen

Entwickelt man eine verteilte Anwendung, so ist die Spezifikation einer geeigneten Softwarestruktur noch wichtiger als bei nicht verteilten Anwendungen. Man sollte dabei die Strukturierung der Anwendung in kleinere, verteilbare Teilkomponenten vornehmen. Und sich dich Frage stellen: „Welche Funktionen sollen lokal, und welche sollen entfernt verfügbar sein?".
Hat man diese Punkte geklärt, ist der zu verwendende Kommunikationsmechanismus festzulegen. Dabei geht es vorrangig um die Auswahl der gewünscht Art bzgl. des Kommunikationsmodells, z.B. Client-Server Architektur oder Gruppenkommunikation.

Auch hier ist die Bewahrung der Konsistenz oberstes Gebot. Werden die Daten konsistent gehalten, insbesondere auch bei replizierten Daten und ist die Konsistenz der Benutzerschnittstellen für die einzelnen Teilkomponenten vorhanden.

Als letztes muss man nun noch die Benutzeranforderungen kontrollieren. Die Funktionalität und Rekonfigurierbarkeit sollte dem Benutzer entgegenkommen. Andernfalls geht man das Risiko ein, dass der Nutzer das Programm ablehnt. Das Programm muss auf die Plattformen abgestimmt werden, mit dem es in Kontakt kommt. Und die Sicherheit der eingegebenen Daten muss gewährleistet werden.

---

[1] Für weitere Informationen zu Lamport sehen sie auf die Webpage: "http://www.research.compaq.com/SRC/personal/lamport/pubs/pubs.html"

3.2 Die objekt-orientierte Verteiltheit

3.2.1 Gründe für objekt-orientierte Verteiltheit

In der traditionellen, nicht verteilten Anwendung dienen Prozeduren bzw. Module zur Strukturierung der Funktionalität von Daten. Prozeduren ermöglichen die Einkapslung von Algorithmen und unterstützen die Wiederverwendbarkeit (allerdings nur durch andere Komponenten dieser Anwendung). Das Binden von allen Komponenten zu einem vollständigen SW-System ist oft nur ein statischer Vorgang.
Diese statischen monolithischen Anwendungen werden durch eine Menge von autonomen Teilkomponenten ersetzt, die miteinander interagieren und im verteilten System migrieren können. Dadurch erhält man die oben angesprochenen Vorteile (Flexibilität, Lastenausgleich, etc.).
Eine Zusammenfassung von Objekten zu einer autonomen Teilkomponente, die jeweils als ein Stück Software erzeugt und gewartet werden kann, wird möglich. Es beinhaltet standardisierte Schnittstellen, um mit anderen Komponenten zusammenarbeiten zu können.

3.2.2 Bisherige Umsetzung

Die wohl häufigste Art der Umsetzung erfolgt mittels MIDDLEWARE. Da unser Referat allerdings im Sprachenblock war, und die MIDDLEWARE-Ansätze schon besprochen wurden, nenne ich hier nur die bekanntesten beim Namen. Ein zweiter Ansatz, der in diesem Reader noch später besser erklärt wird ist der Sprachen-Ansatz. Hier geht es darum direkt für objekt-orientierte Verteiltheit eine eigene Sprache zu entwickeln, und nicht die Middleware zu verwenden.

MIDDLEWARE-ANSATZ

DCOM (Distributed Component Object Model) von Microsoft

RMI und JavaBeans von Sun

CORBA von OMG

Voyager von ObjectSpace
Da wir über Voyager nichts gehört haben nur ein kleiner Anriss. Voyager ist eine Sammlung von Java-Klassen und Schnittstellen, die eine komplette Kommunikationsinfrastruktur liefern, die neben der entfernten Nutzung von Java Objekten auch deren Mobilität und den Ausbau zu Agenten erlaubt. (URL:http://www.objectspace.com\voyager)

SPRACHEN-ANSATZ

EMERALD

BETA

3.2.3 Vorteil objekt-orientierter Verteiltheit

Objekte sind entfernt referenzierbar und lokalisierbar. Außerdem ist es Objekten möglich zwischen logischen Knoten zu migrieren. Die Uniformität bei der Objekt-Kommunikation ist gegeben. Man benutzt die gleiche Semantik für lokale als auch für entfernte Objektaufrufe und die Platzierung von Objekten ist absolut flexibel.

Die meisten lösen die Problemstellung, verteilte Objekte, durch Erweiterungen vorhandener Sprachen (Java, Eiffel, C++, etc.), welche durch die Nutzung vorhandener Standards (RMI, CORBA, DCOM) gezeichnet sind.
Doch es gibt auch Ansätze zu neuen Sprachen, die diese Standards nicht benutzen.
In den folgenden zwei Kapiteln wird auf BETA und EMERALD eingegangen, die diese Aufgabe zu lösen versuchen.

4 EMERALD

5 Die OO-Sprache BETA

5.1 Einleitung

Beta ist eine objektorientierte Sprache, deren Entwicklung 1975 an der „Scandinavian School of object orientation" begann. An der Entwicklung waren Bent Bruun Kristensen, Birger Møller-Pedersen, Ole Lehrmann Madsen und Kristen Nygaard hauptsächlich beteiligt. Kristen Nygaard entwickelte zusammen mit Ole-Johan Dahl die erste objektorientierte Programmiersprache Simula ebenfalls an der jetzt genannten „Scandinavian School of object orientation".
Hauptziel war bei der Entwicklung der Sprache Beta war, sie sehr einfach zu halten. Beta selber wird durch den hohen Abstraktionsmechanismus, den die Sprache besitzt, als sehr zukunftsträchtig eingestuft. Sie hat enge Bezüge zu den objektorientierten Sprachen Eifel, Smalltalk und C++. Sie wurde konzipiert, um einfaches objektorientiertes verteiltes Programmieren zu ermöglichen. Im Zusammenhang mit verteiltes Programmieren hat Beta weitere Bezüge zu der Sprache Emerald. Erste Applikationen wurden in CSCW (Computer Supported Cooperative Work) realisiert. Beispiele für CSCW Applikationen (Anwendungen von rechnerunterstützten Gemeinschaftsarbeit) sind zum Beispiel Videokonferenzsysteme, die eine direkte Kommunikation mit Bild und Ton zwischen räumlich entfernten Personen ermöglicht oder Gruppeneditoren, die das gemeinsame Arbeiten an einem Text möglich machen.

5.2 Das BETA Objekt Modell

In Beta gibt es passive und aktive Objekte. Passive Objekte sind mit denen in C++, Smalltalk oder Eiffel zu vergleichen. Sie haben eine Anzahl von Eigenschaften in Form von Referenzen auf andere Objekte. Weiterhin haben sie eine Anzahl von Pattern, deren Prinzip weiter unten erklärt wird. Aktive Objekte führen eigene Aktionen in einem eigenem unabhängigen Thread aus, der Nebenläufig oder auch als Coroutine gestartet werden kann. Dies ist ein wichtiger Aspekt für die verteiltes Programmieren in Beta. Der Zugriff auf Objekte wird durch Semaphore synchronisiert.
Der schon in der Einleitung erwähnte hohe Abstraktionsmechanismus von Beta sind **PATTERN.** Pattern ist ein allgemeiner Abstraktionsmechanismus der Klassen, Generische Klassen, Prozeduren, Prozesse, CoRoutinen, Fehler, Funktionen vereinigt. Die Vererbung von Pattern ermöglicht also auch eine Vererbung nicht nur von Klassen, sondern auch von Methoden, Prozeduren, Funktionen, Fehlern und Prozessen. Objekte in Beta werden durch Pattern beschrieben.

Deklaration eines Pattern:

```
<names> : <prefix> (#      <declarations>
                           enter  <input-list>
                           do     <imperatives>
                           exit   <output-list>
                       #)
```

Wie in jeder anderen Programmiersprachen Klassen, Attribute und Methoden hat auch in Beta jedes Pattern einen Namen, spezifiziert durch <names>. <prefix> spezifiziert das Pattern, von dem geerbt werden soll. Was zwischen „(#" und „#)" steht wird Objekt-Descriptor oder auch nur Descriptor genannt, in dem das Pattern definiert wird.

Einfache Patternbeispiele:

Funktion:

```
add :  (#      a,b: @integer
               enter  (a,b)
               exit   a+b
               #)
```

Prozedur:

```
add :  (#     a,b: @integer
                    enter  (a,b)
                    do     a+b
               #)
```

Konstante:

```
PI   :  (#
                    exit  3.1415926535897
        #)
```

Obwohl es keinen syntaktischen Unterschied zwischen einem Pattern gibt, der eine Klasse beschreiben soll und einem der eine Methode definiert, so wird doch zwischen Pattern in Ihrer Anwendung unterschieden. Das heißt, dass sie zwar optisch genau gleich aussehen , aber unterschiedliche Bedeutung haben können. Es gibt SuperPattern/ SubPattern (vergleichbar mit Superklassen und erweiterten Klassen), ClassPattern (Klassen), VirtualPattern und NonVirtualPattern (vergleichbar mit virtual functions in C++), NestedPattern (verschachtelte Pattern) und ControlPattern.
Innerhalb eines Patterns kann die Tiefe der Verschachtelung beliebig groß sein, anders als bei vergleichbaren OO – Sprachen. Es ist auch möglich, Objekte zu definieren die nicht als Instanzen eines ClassPattern generiert wurden. Solche Objekte werden Class-less Objects genannt und sind dann sehr sinnvoll wenn es nur ein Objekt geben soll. In anderen OO Sprachen muss dafür eine Klasse existieren.

5.3 Quellcode Beispiele für einfache Objekte

Im folgenden wird ein ClassPattern dargestellt, dessen NestedPattern registerWork und computeSalary eine Aktion beschreiben. Sie sind mit Funkrion in herkömmlichen Sprachen vergleichbar. Dabei ist registerWork ein NonVirtualPattern, dessen Beschreibung vollständig gegeben ist (vergleichbar mit non virtual functions in C++).
ComputeSalary stellt ein VirtualPattern dar, dessen Implementierung nur zum Teil gegeben ist. VirtualPattern erkennt man an der Syntax „:<".

```
employee: (# name: @text;
             birthday: @date;
             dept: ^Department;
```

```
                    totalHours: @integer;
                    registerWork:
                    (#  noOfHours: @integer
                            enter noOfHours
                            do noOfHours + totalhours -> totalHours
                     #);
                    computeSalary:<
                    (# salary: @integer
                            do inner
                       exit salary
                    #);
                  #);
```

Nun sollen von dem ClassPattern employee erweiterte Pattern erzeugt werden, die unterschiedliche Anforderungen an das Pattern employee stellen. Dazu werden zwei Pattern definiert, die von dem Pattern employee erben und das VirtualPattern computeSalary definieren (erkennbar an der Syntax „::<“).

```
worker:employee (# seniority: @integer;
                    computeSalary::<
                    (# do noOfHours*80+seniority*4->salary;
                            0->totalHours
                      #)
                 #);

salesman:employee (# noOfSoldUnits: @integer;
                      computeSalary::<
                      (# do noOfHours*80+noOfSoldUnits*6->salary;
                          0->noOfSoldUnits->totalHours
                       #)
                    #);
```

5.4 Verteiltes Programmieren in BETA

Nachdem nun die grundlegenden Prinzipien der Sprache BETA erläutert wurden, soll nun das verteilte Programmieren mit BETA näher gebracht werden. Für das objektorientierte verteilte Programmieren stellt Beta eine **Distributionlibrary** zur Verfügung. Sie ermöglicht, das der Programmierer sich nicht um Kommunikationsaspekte kümmern muss. Methoden in entfernten Objekten können genauso aufgerufen werden wie Methoden lokaler Objekte. Dies vereinfacht eine Client/Server Implementation. In Beta sind Client und Server auch nur „einfache“ Beta Objekte. Ein Clientobjekt kann auch gleichzeitig ein Serverobjekt sein. Das Client/Server Modell ist ein asymmetrisches Modell in dem Sinn, das der Server nicht Methoden vom Client aufrufen kann und umgekehrt. In Beta ist dies möglich.
Um einfaches verteiltes Programmieren zu ermöglichen, stellt die Distributionlibrary die MainPattern/SuperPattern Remotable, Ensemble, Shell, NameServer und ErrorHandler zur Verfügung.
Das SuperPattern **Remotable** ist ein abstraktes Pattern, bei denen alle Instanzen remotable accessible sind. Jedes Objekt auf das entfernt zugreifbar sein soll, muss also von diesem Pattern erben. Dazu stellt das RemotablePattern ein NestedPattern entry zur Verfügung. Alle Methoden, die von diesem NestedPattern erben, sind remotable accessible. Der Server kann dann den Dienst zur Verfügung stellen, in dem er den Service im NameServer registriert.
Der **NameServer** mappt einen Namen und die Referenz auf ein Objekt (vergleichbar mit einer Hashtabelle) . Er ermöglicht verteilten Beta-Programmen Dienste anderer Beta-Programme zur Verfügung zu stellen. Um verteilte Beta-Programme zu starten, müssen sie von einer Plattform, dem Ensemble, laufen.

Ein **Ensemble** ist ein SuperPattern, welches ein Betriebssystem auf einen Netzwerkrechner simuliert. Es existiert immer genau eine Ensembleinstanz für einen Netzwerkrechner. Auf dem Ensemble können eine oder mehrere Shells laufen.
Eine **Shell** wird direkt vom Ensemble instanziiert und dient zur Ausführung von Beta-Programmen. Sie ist ein bisschen vergleichbar mit einer Unix- Shell. Jede Instanz von einem ShellPattern entspricht einem Betriebssystemprozess und repräsentiert einen physikalischen Adressspeicher.
Der SuperPattern **ErrorHandler** stellt einen Scope zur Verfügung, indem Fehler auftreten , die bei der Kommunikation entstehen und behandelt werden.

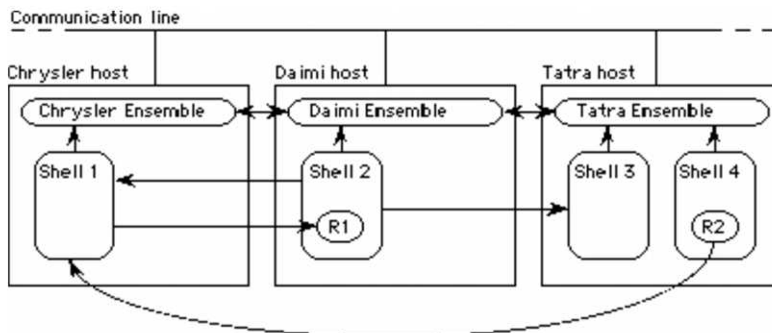


Figure 1: Ensembles, Shells and Remoteables

(Abbildung aus "Object-Oriented Distributed Programming in BETA", Søren Brandt & Ole Lehrmann Madsen)

In obigen Abbildung ist eine Übersicht gezeigt, wie die SuperPattern Ensemble, Shell und Remotable zusammenhängen. Dabei repräsentieren die Pfeile Referenzen.

5.5 BETA Quellcode Beispiele für verteilte Objekte

Im folgenden soll an einem kleinem Beispiel veranschaulicht werden, wie objektorientiertes verteiltes Programmieren in BETA aussieht. Das Bespiel zeigt einen Rechner, der das Pattern plus zur Verfügung stellt. Calculator erbt von dem SuperPattern Remotable der das NestedPattern entry zur Verfügung stellt. Damit wird das NestedPattern plus vom ClassPattern Remotable für entfernte Zugriffe bei dem NameServer registriert und der Dienst wird bereitgestellt.

Remoteable

```
calculator: remoteable (# plus: entry
                              (# a,b,c: @Integer;
                                  enter (a,b)
                                   do a+b -> c
                                   exit c
                               #);
                    #);
```

NameServer

```
c: ^calculator; ns: ^NameServer;
(c[],"Simple Calculator")->ns.put;
```

5.6 Zusammenfassung

Durch den Abstraktionsmechanismus Pattern, welche alle Konzepte bisheriger objektorientierte Sprachen vereinigt, ist es sehr leicht, diese Sprache zu erlernen.
Die anfänglichen Schwierigkeiten des Konzeptsverständnisses sind schnell überwunden und machen ein einfaches Programmieren möglich. Durch die bereitgestellte Distributionlibrary, die dem Programmierer die Kommunikationsaspekte der Objekte erspart, ist verteiltes Programmieren in BETA sehr leicht. Ob sich diese Sprache jedoch durchsetzten wird bleibt durch sogenannte Modeerscheinungssprachen wie Java, die ebenfalls mit RMI und CORBA Lösungen zur Realisierung von verteilten Objekten anbieten, abzuwarten.

6 Fallstudie „Content Management System"

6.1 Einleitung

Um die Bedeutung von verteilten Objekten dem Leser dieses Readers zu verdeutlichen, werden wir zeigen , wie sie in einem Content Management System auftreten können. Dazu erläutern wir zunächst den Begriff Content Management System (im folgendem CMS).
In unserem Beispiel werden wir ein CMS betrachten, welches die Planung, Anmeldung und Durchführung von Lehrveranstaltungen vereinfacht, vereinheitlicht und ins WWW bringt. Damit wird einerseits die Anmeldung und Terminplanung für die Studierenden durchsichtiger und andererseits die Organisation erleichtert. Um verteilte Objekte und deren Bedeutung in diesem CMS zu erklären, betrachten wir im CMS die Teilkomponenten Lehrveranstaltung, Teilnehmerliste, Student und Anmeldung.
Um sich in der Teilnehmerliste einer Lehrveranstaltung anzumelden, muss jeder Student eine webbasierte Anmeldung ausfüllen und abschicken (Abbildung 1.1). Die daraus resultierende Kommunikation und Interaktion zwischen den Teilkomponenten werden wir im Bezug auf verteilte Objekte im Folgendem näher beleuchten.

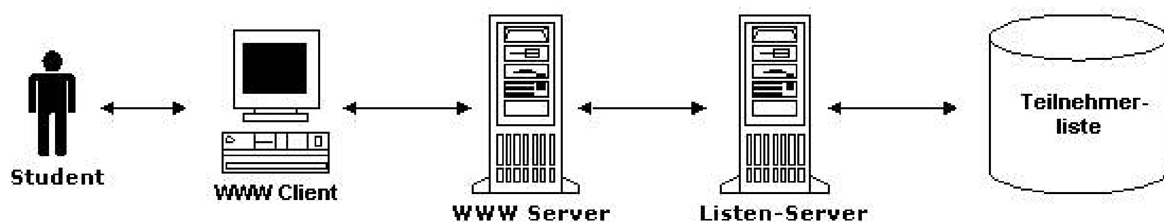


Abbildung 1.1: Interaktion der Teilkomponenten Lehrveranstaltung, Teilnehmerliste, Student und Anmeldung.

Diese Teilkomponenten des CMS sind auf verschiedenen Betriebsystemen installiert, so dass es zu einer plattformübergreifenden Kommunikation kommt.

6.2 Verteile Objekte in einem CMS

Wie aus der Einleitung schon hervorgeht, sind verteilte Objekte in einem CSM von immenser Bedeutung. Ein heterogenes System mit dieser komplizierten Struktur ist ohne sie kaum zu realisieren. Gerade im Bezug auf webbasierten Anmeldungen, dezentraler Datenhaltung und hohe Interaktion innerhalb des Systems ermöglichen verteilte Systeme die Realisierung eines solchen CMS.
Wie aus Abbildung 1.1 hervorgeht, müssen die verschiedenen Teilkomponenten unseres Beispiels eine geeignete Art finden miteinander zu kommunizieren, da sie auf verschiedenen Rechner liegen (Anmeldung auf WWW-Server, Teilnehmerliste auf Listen-Server). In dem aufgeführten Beispiel wird ein Objekt von Typ Anmeldung genau dann kreiert, wenn das Formular auf dem WWW-Client von dem Studenten vollständig und richtig ausgefüllt abgeschickt wurde. Dieses Objekt, welches die Daten des Studenten

enthält, hat eine Referenz auf die Teilnehmerliste auf dem Listenserver. Nun migriert das Objekt Anmeldung vom WWW-Server auf den Listen-Server. Die Teilnehmerliste wird um das migrierte Objekt erweitert.

6.3 Verteilte Objekte im CMS mit BETA

Wir wollen nun mit Hilfe der objektorientierten Sprache Beta, über die wir einen Einblick zuvor gegeben haben, unser oben aufgeführten Beispiel verteiltes Programmieren darstellen.
Der WWW – Server erzeugt mit den Daten des Studenten ein BETA Objekt vom Typ Anmeldung. Dieses Objekt holt sich vom NameServer eine Referenz auf die Teilnehmerliste, fragt ob die maximale Teilnehmeranzahl der Lehrveranstaltung erreicht ist. Ist dies nicht der Fall, so wandert das Objekt zum Listenserver, auf dem sich die Teilnehmerliste befindet, und fügt sich selbst der Liste hinzu. Andernfalls wird eine Mail an den Studenten geschickt, das die maximale Teilnehmerzahl überschritten ist. Die Referenz auf das Objekt wird gelöscht.

6.4 BETA Quellcode Bespiel für verteilte Objekte im CMS

```
teilnehmerliste:remoteable (#   students: 20;
                                currentStudentNumber: 0;
                                liste: [students] @anmeldung;
                                add: entry(#
                                        a: @anmeldung;
                                        enter (a)
                                        do (* liste wird um ein Element erweitert *)
                                        currentStudentNumber+1->
                                        currentStudentNumber;
                                        a+liste -> liste
                                #);
                                full?: entry
                                (#
                                        (if currentStudentNumber< students )
                                                then exit false;
                                                else exit true;
                                        if)
                                #)
                           #);


anmeldung : remotable (# surname : @cstring;
                         firstname : @cstring;
                         matrikelnr: @integer;
                         email: @cstring;
                         lv : @cstring;
                         (#
                         liste: ^teilnehmerliste;
                         ns.get(liste[],lv)->liste;
                         (if liste.full?
                                then (#
                                liste.add(this(anmeldung)[]);
                                liste[] -> this(anmeldung).move;
                                #)
                                else (#
                                mailer: ^mailServer;
```

```
                    ns.get(mailer[],'mail')->mailer;
                    mailer.send(email,'Teilnehmeranzahl in der
                    Lehrveranstaltung ' + lv + ' ist überschritten');
                    this(anmeldung)[] -> NONE;
                    #)
        if)
    #);
```

7 Zusammenfassung

Die verfassten Texte zum Thema verteilte Objekten machen deutlich, wie wichtig diese in heutigen Systemen geworden sind. CSCW Applikationen konnten durch sie erst realisiert werden. Durch neue Sprachen und Interfaces ist verteiltes Programmieren sehr stark vereinfacht worden. Die größeren Bandbreiten sorgen für weitere Möglichkeiten, verteilte Objekte stärker denn je einzusetzen.

8 Literatur

8.1 Einleitung, Verteilte Systeme, Verteilte Anwendungen

Skript zur Vorlesung im Sommersemester 2001
„Verteilte Anwendungen - Grundlagen und Konzepte zur Erstellung verteilter kooperativer Anwendungen" von Prof. Dr. J. Schlichter an der TU München [http://www11.in.tum.de/lehre/lectures/ss2001/va/extension/latex/va_course_student.pdf]

8.2 Emerald

8.3 BETA

Distributed Objects in BETA
Mjølner Informatics Report MIA 93-25,Nov´00

Object-Oriented Distributed Programming in BETA
Søren Brandt & Ole Lehrmann Madsen
Computer Science Department, Aarhus University,Ny Munkegade, DK-8000 Aarhus C, Denmark ,February 24, 1994

The Mjølner BETA System
BETA Language Introduction
Mjølner Informatics Report MIA 94-26(1.2)