GeoTransporter

Design und Implementierung eines Objekttransportes für das Geo-System

Roger Brudermann

Ubilab

Ubilab Technical Report 97.3.1

Ubilab (UBS Information Technology Laboratory)

Ubilab is the corporate information technology innovation center of Union Bank of Switzerland. It pursues a small number of attractive, highly competitive research projects with the aim of maintaining the status of a recognized research institution.

It is the task of Ubilab to actively assist UBS with its goal of becoming a leader in the mastery of modern IT equipment, techniques, and methods. This task is pursued by means of intimate involvement in application-oriented research and advanced development projects. Furthermore, UBS contributes to fostering the interaction between IT research and application, that is between academia in its search for new methods, and business in its application of them. To this purpose Ubilab cooperates with universities and other research institutions on a worldwide basis. The goal is to expand the scope of the Laboratory by carrying out common projects, thus guaranteeing—provided top-notch partners are found—the quality of the research. The impact of Ubilab is therefore focussed on both the IT departments of UBS Group and the IT research community at large.

For further information about Ubilab, staff, projects, or publications see our World Wide Web (WWW) pages. Feel free to contact the staff personally via e-mail or to write to the mailing address given below.

Location of Ubilab

Universitätsstrasse 84
CH-8033 Zurich
Switzerland

Mailing address

Union Bank of Switzerland
Ubilab
Bahnhofstrasse 45
CH-8021 Zurich
Switzerland

Electronic access

Phone: ++41 1 236 57 14
Fax: ++41 1 236 46 71
E-mail: firstname.lastname@ubs.com
WWW: http://www.ubs.com/ubilab

Zusammenfassung

Die vorliegende Arbeit beschreibt in einem ersten Teil den Entwurf und die Implementierung eines Kommunikationssubsystems für den allgemeinen Transport von Objekten zwischen Prozessen. Diese Arbeit wurde vollständig eingebettet in das Geo-System, dessen Ziel es ist, mittels einer reflexiven Architektur eine Infrastruktur für global verteilte Objekte zu schaffen. Beim Entwurf des Frameworks für das Kommunikationssubsystem wurde bewusst auf Offenheit und Erweiterbarkeit geachtet. Für die Realisierung der "Low-Level"-Kommunikation wurde OrbixWeb, eine Implementierung der Common Object Request Broker Architecture (CORBA) der Firma Iona, verwendet. Dank der Offenheit des Subsystems ist es jedoch leicht, alternative Implementierungen zu verwenden, die z.B. auf Sockets oder Java Remote Method Invocation basieren. Im zweiten Teil der Arbeit wird gezeigt, wie Methodenaufrufe auf Objekten in entfernten Prozessen als Anwendung dieses allgemeinen Objekttransportmechanismus realisiert wurden.

Diese Arbeit wurde dem Institut für Computersysteme der ETH Zürich am 3. März 1997 als Diplomarbeit vorgelegt. Die Arbeit wurde von Prof. Dr. Thomas Gross vom Institut für Computersysteme der ETH Zürich und von Dirk Riehle vom Ubilab, dem Forschungslabor der Schweizerischen Bankgesellschaft, betreut.

Roger Brudermann

Zürich, 3. März 1997

Danksagung

Ich danke Prof. Dr. Thomas Gross, meinem Diplomprofessor vom Institut für Computersysteme der ETH Zürich, und allen Ansprechpartnern am Ubilab für die Unterstützung meiner Diplomarbeit.

Mein ganz besonderer Dank gilt Dirk Riehle, meinem Betreuer in der Gruppe Object-oriented Software Engineering des Ubilabs. Zum einen möchte ich ihm danken für das Beantworten der unzähligen Fragen, mit denen ich ihn im Verlauf dieser Arbeit konfrontiert habe. Zum anderen habe ich während den zahlreichen gemeinsamen Diskussionen in diesen vier Monaten viel von ihm gelernt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Die vorliegende Diplomarbeit beschäftigt sich mit dem Entwurf und der Implementierung eines allgemeinen Mechanismuses für den Transport von Objekten zwischen Prozessen. Dieser Transportmechanismus wurde vollständig in das Geo-System eingebettet. Das Geo-System verfolgt im Rahmen des Geo-Projektes das Ziel, mittels einer reflexiven Architektur eine Infrastruktur für global verteilte Objekte zu implementieren. Ausgangspunkt für die Realisierung dieser Arbeit war eine erste Version einer Einprozessimplementierung der virtuellen Maschine (VROOM), die dem Geo-System zugrunde liegt.

Aufbauend auf dem für den Objekttransport implementierten Kommunikationssubsystem wird in einem zweiten Teil gezeigt, wie die virtuelle Maschine um die Möglichkeit ergänzt wurde, in entfernten Objekten Methoden aufzurufen.

Beim Entwurf wurde bewusst auf die Offenheit und Erweiterbarkeit des Subsystems geachtet. Es wurde Wert darauf gelegt, ein komplett objektorientiertes Design des Subsystems vollständig durchzuziehen. Die Erreichung von hoher Performanz war dabei kein Gesichtspunkt, der im Vordergrund stand.

1.1 Überblick über die Arbeit

In Kapitel 2 werden kurz die allgemeinen, grundlegenden Konzepte dieser Arbeit erläutert. Das Kapitel schliesst mit einem kurzen Überblick über einige bestehende, verteilte objektorientierte Systeme. Das Kapitel 3 stellt das Geo-System vor. In Kapitel 4 wird die Architektur des Kommunikationssubsystems vorgestellt. Darunter finden sich neben dem Objekttransport auch mehrere Dienste, die den Benutzern des Objekttransportes angeboten werden. Kapitel 5 beschäftigt sich mit der Serialisierung von Objekten und deren Transformation in eine externe Datenrepräsentation. In Kapitel 6 wird gezeigt, wie entschieden wird, wie umfangreich ganze Objektgraphen übermittelt werden. Kapitel 7 beschreibt die Realisierung von fernen Methodenaufrufen basierend auf dem Objekttransport-mechanismus und Kapitel 8 schliesslich erläutert die durch die vorliegende Arbeit gewonnen Erkenntnisse und noch offene Punkte. Des weiteren wird gezeigt, was bei einer Neuimple-mentierung anders angegangen würde.

1.2 Verwendete Notation

Für die Beschreibung der Klassen- und Objekt-Interaktionsdiagramme in dieser Arbeit wurde eine erweiterte Form der Taligent Notation verwendet, die in Anhang A kurz beschrieben wird.

2 Allgemeine Konzepte

In diesem Kapitel werden kurz die allgemeinen Konzepte vorgestellt, die für das Verständnis dieser Diplomarbeit nötig sind.

2.1 Verteilte objektorientierte Systeme

In einem verteilten objektorientierten System verteilt sich die Menge aller Objekte über mehrere Prozesse. Im traditionellen Client/Server-Modell bieten sog. Serverprozesse Objekte für den gemeinsamen Zugriff an. Clients können auf die durch die Serverprozesse bereitgestellten Objekte zugreifen. Durch den gemeinsamen Zugriff auf Objekte können Ressourcen gemeinschaftlich benutzt werden. Die klare Trennung in Client- und Server-Prozesse verschwindet jedoch immer mehr, da in einem solchen System ein entsprechender Prozess in der Tat sowohl Client (für den Zugriff auf entfernte Objekte) als auch Server (als Anbieter der eigenen Objekte) ist.

Ein Begriff, den man im Zusammenhang mit verteilten Systemen oft antrifft, ist die Transparenz. Unter Transparenz in einem verteilten System versteht man die Tatsache, dass man die Verteilung vor dem Benutzer verstecken möchte. Der Benutzer soll mit dem gleichen Mechanismus auf lokale wie auch auf entfernte Objekte zugreifen können (Transparenz bezüglich Zugriff). Zudem muss er nicht wissen, auf welchem Rechner seine Objekte sich gerade befinden (Transparenz bezüglich Ort).

Verteilte Systeme sind wesentlich komplexer als herkömmliche Einprozesssysteme. Dies wird illustriert durch die folgenden, nichttrivialen Probleme, die in verteilten Systemen auftreten können:

- Kommunikationsfehler (durch verfälschte oder verlorengegangene Nachrichten, Netzwerkpartitionierung),
- Servercrash,
- Skalierbarkeit bei vielen Clients,
- konkurrenzierender auf Ressourcen (Software, Hardware, Objekte),
- Sicherheit der Übertragung (gegenseitige Autorisierung von Client und Server).

Einen guten Einstieg in die Thematik der verteilten Objekte bzw. Systeme bieten [CDK94, OHE96].

2.2 Objekttransport

Ein Mechanismus, der beliebige Objekte oder ganze Objektgraphen vom Adressraum eines Prozesses in den Adressraum eines anderen Prozesses transportieren kann, ist die notwendige Voraussetzung, damit die folgenden Eigenschaften in einem System erreicht werden können:

Performanz	Durch das Verschieben von Objekten, die häufig miteinander kommunizieren, auf einen gemeinsamen Knoten (Clustering), erhöht sich die Performanz des Systems.
Lastbalanzierung	Die Verteilung von Objekten ermöglicht eine bessere Auslastung von bestehenden Ressourcen in einem Netzwerk.
Verfügbarkeit	Objekte können auf Knoten migriert werden, die eine höhere Verfügbarkeit garantieren. Müssen Rechner unterhalten oder ersetzt werden, können die darauf “lebenden” Objekte kurzzeitig auf einen anderen Rechner verschoben werden, damit der Zugriff auf die Objekte ohne Unterbruch gewährleistet werden kann.
Effiziente Aufrufe	Dadurch, dass Parameterobjekte für die Dauer eines fernen Methodenaufrufes mitgegeben werden, kann die Performanz in einem verteilten System massiv erhöht werden [JLHB88, Jul88, Lop96, Lux95].
Sicherheit	Um Sicherheitsansprüchen zu genügen, können Objekte auf “vertrauenswürdige” Rechner verschoben werden.

Grundsätzlich gibt es zwei Varianten des Objekttransportes: Migration und Replikation. Bei der Migration werden Objekte von einem Senderprozess zu einem Empfängerprozess verschoben. Bei der Replikation hingegen werden Objekte im Adressraum des Empfängers repliziert. Im Gegensatz zu der Replikation verfügt der Senderprozess nach der Migration nicht mehr über die transportierten Objekte. Replikation wird häufig eingesetzt, um teure, ferne Zugriffe zu vermeiden. Sie hat aber den Nachteil, dass das System die Konsistenz der Replikate mit den Originalobjekten gewährleisten muss.

Die Schwierigkeiten beim Objekttransport liegen darin, dass der Mechanismus in der Lage sein muss, jedes beliebige Objekt zwischen zwei Prozessen transportieren zu können. Dazu müssen die Objekte beim Sender für die Übertragung in eine externe, rechnerunabhängige Datenrepräsentation transformiert werden. Der Empfänger muss seinerseits die Objekte wieder korrekt herstellen können. Da ganze Objektgraphen transportiert werden können, müssen auch eventuell auftretende Zyklen behandelt werden. Dies erfordert eine für die Übertragung notwendige Serialisierung der Objektgraphen.

2.3 Methodenaufrufe in entfernten Objekten

Verteilte Objekte machen erst Sinn, wenn es auch möglich ist, in entfernten Objekten Methoden aufzurufen. Dazu wird ein Mechanismus benötigt, der das Marshalling bzw. das Unmarshalling der Methodenaufrufe übernimmt.

In der prozeduralen Welt hat sich schon lange der Remote Procedure Call (RPC) Mechanismus etabliert. Dieser implementiert den lokalen Prozeduraufruf in einer verteilten

Umgebung. Ein Prozess (Client) ruft eine von einem anderen Prozess (Server) bereitgestellte Prozedur auf. Diese Prozedur wird dann im Datenbereichs des Servers ausgeführt [CDK94].

Ein Methodenaufruf in einem entfernten Objekt ist im Prinzp sehr ähnlich wie ein Remote Procedure Call. Da aber ein Methodenaufruf auf einem Objekt erfolgt, ist dieser viel gezielter als ein RPC an gewisse Daten (an den Zustand eines Objektes) gekoppelt. In der Praxis gibt es verteilte objektorientierte Systeme, die ihre fernen Methodenaufrufe mittels RPC implementieren (einzelne CORBA Implementierungen, Microsoft's DCOM).

Bei lokalen Methodenaufrufen blockiert in der Regel der Sender solange, bis der Empfänger mit der Ausführung der Methode fertig ist (Synchrone Aufrufe). Dieses Verhalten ist aber für ferne Methodenaufrufe nicht unbedingt wünschenswert. Sind die fernen Methodenaufrufe asynchron implementiert, wird der Sender nicht blockiert und kann parallel weiterarbeiten.

Ein wesentlicher Punkt bei Methodenaufrufen in entfernten Objekten ist, dass der Serverprozess für das gewünschte Objekt am Laufen sein muss, da ansonsten keine Kommunikation zustande kommen kann. Diese Bedingung stellt im Rahmen dieser Arbeit kein Problem dar und wird nur der Vollständigkeit halber erwähnt. Falls die Bedingung aber inakzeptabel ist, muss auf sog. "Message-oriented Middleware" (MOM) ausgewichen werden. Dabei werden die Methodenaufrufe der Clients in Warteschlangen, die unabhängig sind von den Servern, gespeichert. Die Serverprozesse beginnen mit der Abarbeitung der Warteschlange sobald sie selbst verfügbar sind. Dadurch ist keine direkte Verbindung mehr nötig zwischen Client und Server. Ein weiterer Vorteil ist, dass MOM Systeme in der Regel asynchron sind.

2.4 Kurzer Überblick über bestehende Systeme

In diesem Kapitel wird ein kurzer Überblick über einige ausgewählte, verteilte objektorientierte Systeme gegeben. Dabei handelt es sich um Emerald, um die Common Object Request Broker Architecture (CORBA) und um Java Remote Method Invocation (RMI).

2.4.1 Emerald

Emerald ist eine objekt-basierte Programmiersprache und ein System für die Konstruktion von verteilten Anwendungen. Emerald wurde seit 1985 an der Universität von Washington entwickelt [JLHB88, Jul88]. Die wesentlichen Ziele, die dabei verfolgt wurden, sind:

- Mobilität von einzelnen Objekten
- effiziente lokale Methoden
- ein einziges Objektmodell für lokale und entfernte Objekte

Als einziges der hier vorgestellten Systeme bietet Emerald auch direkte Unterstützung für den Objekttransport zwischen Prozessen. Jedes Objekt in Emerald kann zu jedem Zeitpunkt über Rechnergrenzen hinweg verschoben werden. In Emerald existieren dazu verschiedene Primitiven, die es erlauben, einzelne Objekte zu lokalisieren, zu verschieben oder im Adressraum eines bestimmten Prozesses zu fixieren. Zudem implementiert Emerald den transparenten Zugriff auf Objekte. Zwischen einem lokalen oder fernen Aufruf besteht kein sprachlicher Unterschied. Dies ist möglich, weil Emerald ein einziges Objektmodell sowohl für lokale als auch für entfernte Objekte unterstützt. Diese Eigenschaft vereinfacht das Programmieren von Anwendungen extrem. So ist es möglich, dass Anwendungen entwickelt werden, ohne dass dabei eine mögliche Verteilung in Betracht gezogen werden muss. Trotz

der Tatsache, dass jeder Aufruf potentiell auf ein entferntes Objekt zugreifen kann, wurde darauf geachtet, dass die Performanz von lokalen Aufrufen dennoch möglichst hoch ist.

2.4.2 Common Object Request Broker Architecture (CORBA)

Die Common Object Request Broker Architecture (CORBA) ist ein händlerunabhängiger, offener Standard für verteilte Objekte in heterogenen Umgebungen, der von der Object Management Group (OMG) seit 1990 kontinuierlich entwickelt wurde [OMG95, OHE96, Sie96]. Die OMG ist ein Konsortium bestehend aus über 600 Firmen, die praktisch das ganze Spektrum der Computerindustrie abdecken. Die einzige erwähnenswerte Firma, die sich diesem Konsortium nicht angeschlossen hat, ist Microsoft. Microsoft setzt in diesem Gebiet voll auf ihre proprietäre Lösung ActiveX und DCOM. Der CORBA Standard garantiert die Interoperabilität von Objekten unabhängig von den verwendeten Programmiersprachen und Betriebssystemen. Zudem ist es einfach möglich, durch sog. Wrapper-Objekte "Altlasten" in CORBA zu integrieren.

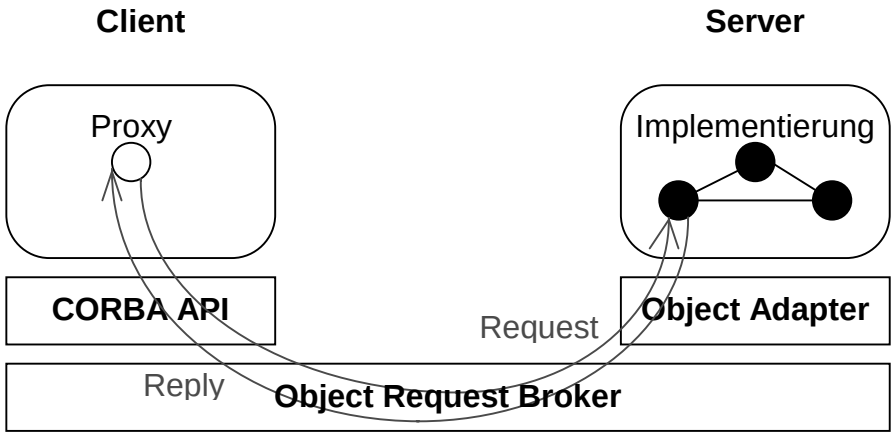


Abb. 2.1: CORBA Object Request Broker

Das Herz von CORBA ist der Object Request Broker (ORB). Er bildet die Kommunikationsinfrastruktur für die verteilten Objekte. Mittels der Interface Definition Language (IDL) ist es möglich, sprach- und betriebssystemunabhängig Schnittstellen zu spezifizieren. Der Server implementiert die Objekte, der Client muss nur deren Schnittstellen kennen. Der Client kann transparent via ein Proxy Objekt, das als Platzhalter für das entsprechende entfernte Objekt dient, Methoden aufrufen (Abb. 2.1). Dabei spielt es keine Rolle, wo die Server Objekte sich befinden. Der ORB selbst übernimmt die Lokalisierung, er übermittelt den Request bzw. Reply (inkl. Marshalling und Unmarshalling), ruft die entsprechende Methode auf und behandelt eventuell auftretende Exceptions.

Dazu enthält die CORBA 2.0 Spezifikation eine Reihe von externen Diensten, die bei Bedarf beansprucht werden können. Diese wurden im Rahmen der Common Object Service Specification (COSS) [OMG96] standardisiert, um keinen "Wildwuchs" durch die Hersteller der konkreten Implementierungen aufkommen zu lassen. Diese CORBA Services bauen auf der ORB-Infrastruktur auf und sind selbst als CORBA Objekte realisiert. Dies bedeutet, dass sie über ein IDL-Interface ansprechbar sind und alle Mechanismen des ORBs ausnützen können. Durch die IDL-Interfaces sind sie unabhängig von irgendwelchen Implementierungen und somit auch austauschbar. Die OMG hat bisher elf Services spezifiziert. Es sind dies: Naming, Event, Lifecycle, Persistent Object, Transaction, Concurrency Control, Relationships, Externalization, Query, Licensing und Property.

CORBA bietet direkt keine Unterstützung für den Transport von Objekten. Die OMG hat jedoch genau für diesen Zweck den "Object Externalization Service" definiert. Dieser Dienst erlaubt es, Objekte in einen Stream zu externalisieren und von einem Stream zu internalisieren. Die durch die Externalisierung erhaltenen Streams können für die Objektspeicherung oder eben für den Transport von ganzen Objektgraphen zwischen Prozessen verwendet werden (Objektgraphen können in CORBA mit dem Relationships Service konstruiert werden). Zum heutigen Zeitpunkt (Februar 1997) gibt es aber noch keine CORBA Implementierung, die den "Object Externalization Service" auch wirklich anbietet.

CORBA hat den Nachteil, keine Unterstützung für die Evolution von Interfaces zu bieten. Zudem ist ein ferner Methodenaufruf nur erfolgreich, wenn der entsprechende Serverprozess am Laufen ist. In Zukunft wird jedoch erwartet, dass CORBA mehr Unterstützung für asynchrone Kommunikation (asynchrone Kommunikation ist via Dynamic Invocation Interface (DII) schon beschränkt möglich) und für die zeitliche Entkopplung von Client- und Server-Prozessen analog zu sog. Message-oriented Middleware (MOM) bieten wird. Ein weiterer Schwachpunkt von CORBA ist, dass keine standardisierte Möglichkeit existiert, um Serverprozesse zu starten. Zudem gibt es in CORBA keine Möglichkeiten, für einen Aufruf einen bestimmten “Quality of Service” garantieren zu können. Dies sind genau die Punkte, in denen sich die einzelnen Produkte durch ihre proprietären Lösungen unterscheiden und profilieren.

Die wichtigsten, kommerziell erhältlichen CORBA Implementierungen sind: Orbix(Web) von Iona, VisiBroker von Visigenics, ObjectBroker von Digital, PowerBroker von Expersoft, System Object Model (SOM) von IBM, ORB Plus von HP und Neo von Sun.

2.4.2.1 OrbixWeb - die von uns verwendete CORBA Implementierung

Das Produkt OrbixWeb der Firma Iona ist eine Implementierung des CORBA 2.0 Standards mit Java Anbindung. OrbixWeb erfüllt den Standard bezüglich Interoperabilität (Unterstützung des IIOP Protokolls) und den Standardkomponenten wie Interface Repository, Implementation Repository, etc. Ausser dem Naming Service wird von Iona aber noch keiner der standardisierten Services angeboten.

Die Funktion des Basic Object Adapters wird von einem speziellen Prozess, der auf jedem Rechner laufen muss, wahrgenommen, dem Orbix Daemon. Dieser Daemon wartet auf eingehende CORBA Requests und leitet diese an die Server weiter, die die gewünschten Objekt implementieren. Diese Architektur hat zwei wesentliche Vorteile: Zum einen müssen die Server-Prozesse nicht von Hand aufgestartet werden. Diese Aufgabe übernimmt der Daemon beim erstmaligen Zugriff auf den Server. Der Daemon stützt sich dabei auf das Implementation Repository, in dem alle Server zuvor registriert sein müssen. Zum anderen muss nicht jedes Client/Server-Paar für die Kommunikation eine eigene TCP/IP-Portnummer vereinbaren, falls das Internet Inter-ORB Protocol (IIOP) benutzt wird. Ein Client-Prozess muss nur die Portnummer des Daemons kennen; der Daemon leitet dann eingehende Requests an den Port des entsprechenden Server-Prozesses weiter. Da der Daemon die Server-Prozesse aufstartet, kennt er auch deren Portnummern.

Für die vorliegende Arbeit wurde OrbixWeb Version 2.0 (auf Solaris) zur Implementierung der “Low-Level” Kommunikation benutzt. Dabei hat sich vor allem das automatische Aufstarten der Serverprozesse durch den Orbix Daemon als äusserst nützlich erwiesen. Dies hat auch den Vorteil, dass abgestürzte Prozesse beim nächsten Zugriff automatisch wieder hochgefahren werden.

2.4.3 Java Remote Method Invocation (RMI)

Mit der Version 1.1 des Java Development Kits (JDK) bietet Java nun auch Unterstützung für verteilte Objekte. Im Gegensatz zu der CORBA Architektur, deren Ziel es ist, sprach- und betriebssystemunabhängig zu bleiben, wird kein eigenes Objektmodell definiert, sondern Java Remote Method Invocation (RMI) wird nahtlos in das Java-Sprachsystem eingebunden. Aus diesem Grund müssen sich sowohl Client als auch Server in einer homogenen Java Umgebung befinden. Für das Marshalling der fernen Methodenaufrufe beim Client sind sog. Stub-Objekte verantwortlich; Skeleton-Objekte übernehmen auf der Serverseite das Unmarshalling der eingehenden Requests und den eigentlichen Methodenaufruf. Als Ausgangspunkt für die Generierung der Stub- und Skeletonklassen dienen direkt die Java-Interfaces (präziser deren ".class" Dateien). Es ist keine separate Schnittstellenbeschreibungssprache wie in CORBA nötig. Ein Interface Repository wie in CORBA wird nicht gebraucht, weil Java die Typinformationen zur Laufzeit bereitstellt.

Jedes Java Objekt, das potentiell entfernt zugreifbar sein soll, muss die leere Schnittstelle "Remote" implementieren. Jede entfernte Methode muss die "RemoteException" in ihrer Signatur deklarieren. Diese Exception wird durch das RMI-Laufzeitsystem geworfen, wenn ein Kommunikationsfehler aufgetreten ist oder das gewünschte Objekt nicht gefunden werden konnte.

Ein sehr wesentlicher Unterschied zwischen CORBA und Java RMI besteht zudem in der Aufruf-Fehlersemantik von fernen Aufrufen. Während CORBA für seine Aufrufe "at-most-once" Semantik (siehe Kapitel 7.4.3) realisiert, implementiert Java RMI lediglich "maybe" Semantik (Kapitel 7.4.1). Diese bedeutet, dass Applikationen, die ferne Methodenaufrufe benutzen, explizit die möglichen Fehlersituationen selbst behandeln müssen. Dies ist ein gravierender Nachteil gegenüber CORBA.

Ein Pluspunkt für Java RMI ist jedoch, dass verteilte Garbage Collection unterstützt wird. Das System ist in der Lage, den Speicher für nicht mehr referenzierte Objekte freizugeben. Die dazu verwendete Technik basiert auf der Technik des Reference Counting.

Analog zum CORBA Naming Service gibt es auch in Java RMI einen Mechanismus, um Referenzen auf entfernte Objekte zu erhalten: die Registry. Die Registry, selbst ein entferntes Objekt, ist eine transiente Datenbank, die Abbildungen von Uniform Resource Locators (URL) auf entfernte Objekte verwaltet. Sämtliche Einträge in der Registry gehen jedoch nach einem Systemabsturz im Gegensatz zum CORBA Naming Service verloren. Jeder Rechner, der Objekte für den fernen Gebrauch anbietet, muss über eine Registry verfügen. Der Client muss wissen, auf welchem Rechner sein gewünschtes Objekt liegt. In CORBA übernimmt der ORB die Lokalisierung des Rechners.

Dazu kommt, dass Java analog zu CORBA (abgesehen vom Externalization Service) keine Unterstützung bietet, Objekte oder gar ganze Objektstrukturen von einer virtuellen Maschine auf eine andere zu verschieben. Des weiteren gelten sämtliche Schwächen, die CORBA aufweist (s.o.), auch für Java RMI.

Für mehr Information wird auf [Wol96] verwiesen.

3 Das Geo-System

Dieses Kapitel versucht, einen Überblick über die grundlegenden Konzepte des Geo-Systems zu geben. Das Ziel dieses Systems ist es, eine Infrastruktur für global verteilte, objektorientierte Systeme zu schaffen. An dieser Stelle kann keine vollständige Behandlung des Geo-Systems gegeben werden; es wurde vielmehr versucht, die wichtigsten Merkmale herauszufiltern, um dem Leser einen Eindruck zu vermitteln. Für weitergehende Informationen wird auf [BGR96] verwiesen.

3.1 Einführung

Die folgenden Gesichtspunkte stehen bei der Schaffung des Geo-Systems, einer Infrastruktur für global verteilte Objekte, im Vordergrund:

- Unterstützung von flexiblen Verteilungsstrategien
- Bereitstellen einer ausgereiften Metalevel-Architektur für die Introspektion und das System Management
- Evolution von Klassen
- Integration von bereits bestehenden Anwendungen (“Altlasten”)

Flexible Verteilungsstrategien werden gebraucht, um die Migration und Replikation von Objekten unterstützen zu können. Sie sind aber auch Voraussetzung für die Lastbalanzierung in einem verteilten objektorientierten System.

Die durch die Metalevel-Architektur bereitgestellte Metainformation wird einerseits für die Introspektion und andererseits für das Debugging gebraucht. Des weiteren kann sie in Zukunft als Ausgangspunkt für die Implementierung eines generischen Object Streaming-Verfahrens dienen. Zudem ist sie nötig, um neue Abstraktionen einführen zu können, die über den Konzepten der Objekte und Klassen stehen (Rollen-Objekte).

Falls ein verteiltes System ununterbrochen am Laufen sein muss, wird es schwierig, dieses um neue Klassen zu erweitern oder bestehende zu ersetzen.Das Problem der Evolution von Klassen kann nur angegangen werden, wenn das System reflexiv ist, was bedeutet, dass sämtliche Metainformationen selbst explizit (in der Form von Objekten) Bestandteile des Systems sind. Da Geo ein reflexives System ist und demzufolge Klassen auch als Objekte repräsentiert werden (analog zu Smalltalk), wird es möglich, diese Klassen-Objekte in Transaktionen einzubeziehen, um sie zu ändern oder um neue hinzuzufügen.

Da heute in den seltensten Fällen noch “auf der grünen Wiese” begonnen werden kann, muss es möglich sein, bereits bestehende Applikationen einfach in das System zu integrieren. Mittels sog. “Wrapper”-Objekten ist es einfach möglich, bestehenden Code einzubinden.

Dadurch, dass die Instanzen einer Klasse nicht zwingend mit einer einzigen Implementierung assoziiert werden, können je nach Bedürfnissen (Geschwindigkeit, Ressourcenverbrauch, etc.) unterschiedliche Implementierungen verwendet werden für die Instanzen der gleichen Klasse.

3.2 Warum nicht CORBA?

Der grosse Unterschied zwischen dem Geo-System und der Common Object Request Broker Architecture (CORBA, siehe Kap. 2.4.2) besteht darin, dass im Geo-System diverse Dienste wie Serialisierung, Transaktionen, Persistenz, etc. bereits zu der Grundfunktionalität eines jeden Objektes gehören, während diese in CORBA optional bei Bedarf als externe Dienste [OMG96] beansprucht werden können. Zudem ist in Geo - bedingt durch die geforderte Reflexivität - die gesamte Metainformation selbst explizit Bestandteil des Systems. Diese Metainformation ist zwar auch im CORBA Interface Repository enthalten; sie verfügt jedoch bei weitem nicht über den gleichen Stellenwert wie im Geo-System.

Diese Verankerung von Grundfunktionalität direkt im Basisobjekt bietet den Vorteil, das Frameworks viel einfacher entwickelt werden können. Dieser Schluss wurde aus der Verwendung der ET++ Infrastruktur und von Smalltalk im Gegensatz zur Verwendung von C++ mit seinen eher rudimentären Bibliotheken gezogen. Damit muss nicht in jedem Projekt von neuem die Grundfunktionalität implementiert werden, die eigentlich von jedem Objekt benötigt wird.

3.3 Virtual Reflexive Object-Oriented Machine (VROOM)

Das Herz des Geo-Systems besteht aus der Virtual Reflexive Object-Oriented Machine (VROOM). Diese virtuelle Machine implementiert die gewünschte reflexive Architektur mit einigen wenigen Abstraktionen, die zum Teil im folgenden hier kurz vorgestellt werden. Eine erste VROOM-Einprozessimplementierung der in [BGR96] vorgestellten Anforderungen an eine Infrastruktur für global verteilte Objekte bildete die Grundlage für die Realisierung der vorliegenden Diplomarbeit. Sowohl die VROOM als auch die Diplomarbeit wurden komplett in Java geschrieben.

Die folgenden Unterkapitel vermitteln kurz diejenigen VROOM Konzepte, die für das Verständnis der Diplomarbeit nötig sind

3.3.1 Klassen und Objekte

Alle Klassen in der VROOM erben von AnyObject. AnyObject definiert die Basisfunktionalität, über die jedes VROOM Objekt verfügen muss. Dies sind zur Zeit Externalisierung bzw. Internalisierung und Kopierbarkeit. Hinzu kommen Methoden, um auf die assoziierte Metainformation zuzugreifen. In Zukunft wird das Framework erweitert werden, um Unterstützung zu bieten für weitere Grundfunktionalität wie Transaktionen, Nebenläufigkeit, Persistenz, etc.

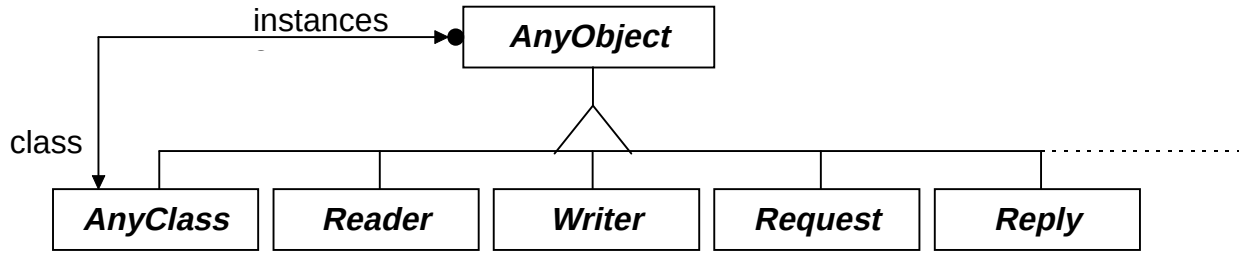


Abb. 3.1: einige Klassen des VROOM Systems

Jede Klasse im VROOM System wird durch ein Klassen-Objekt repräsentiert. Diese Klassen-Objekte sind Instanzen der Klasse AnyClass, welche eine Subklasse von AnyObject ist.

Das Metaklassen-Objekt, ein ausgezeichnetes Klassen-Objekt und damit eine Instanz von AnyClass, verwaltet alle Klassen-Objekte. Die Klassen-Objekte ihrerseits verwalten klassenspezifische Informationen wie den Klassennamen, etc. und sämtliche Instanzen der Klasse, die sie repräsentieren. In Abb. 3.2 wird anhand eines Beispiels (Kunden, Konten, Bank) gezeigt, wie das VROOM System Metainformationen und die eigentlichen Instanzen verwaltet. Unter dem Objektbezeichner wird zum besseren Verständnis der Klassennamen der jeweiligen Instanz angegeben.

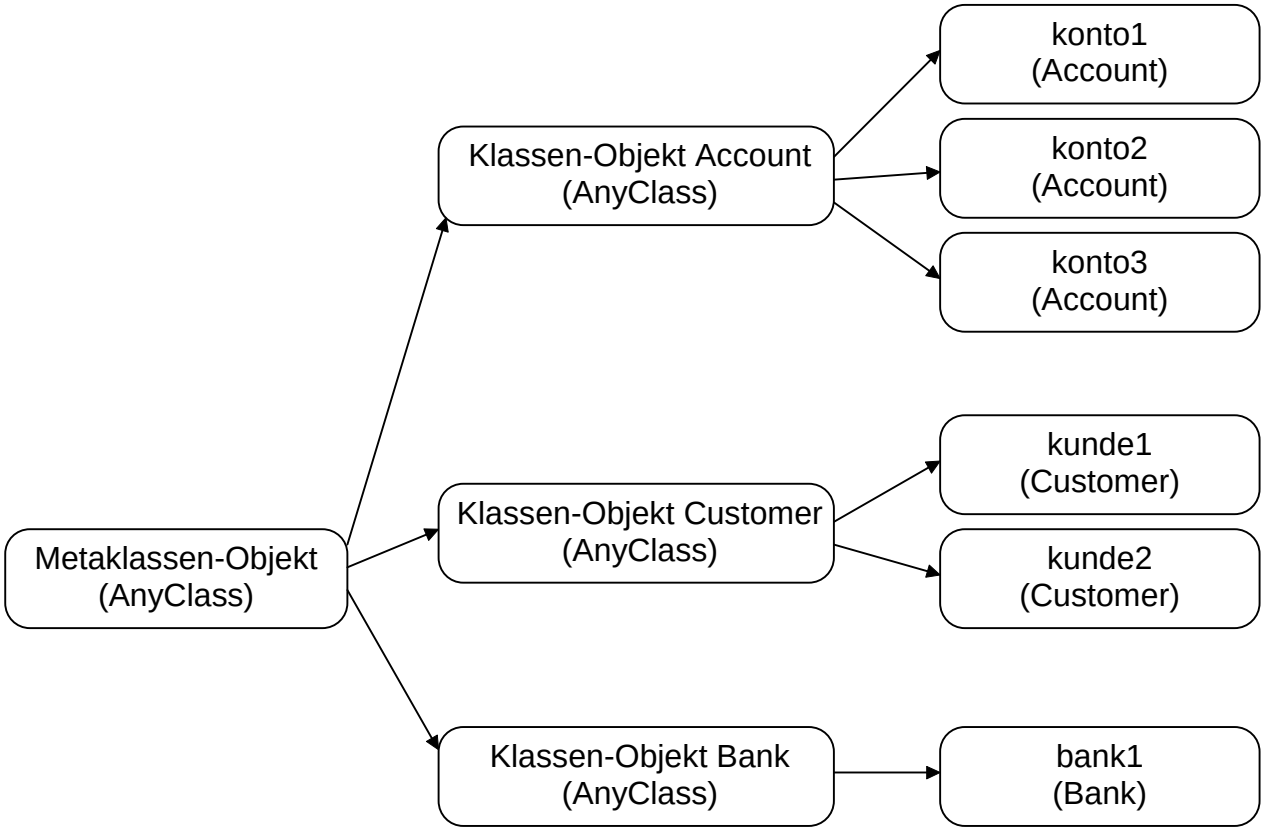


Abb. 3.2: Objektdiagramm der Metainformationen im VROOM System anhand eines Beispiels

Ein Klassen-Objekt kann zudem, wie oben bereits erwähnt, verschiedene Implementierungen für seine Instanzen verwenden. Normalerweise steht immer eine Default-Implementierung zur Verfügung, die bei Bedarf instantiiert werden kann. Das Klassen-Objekt kann aber auch

alternative Implementierungen verwalten, die z.B. optimiert sind bezüglich Ausführungsgeschwindigkeit der Methoden oder Ressourcenverbrauch. Dank dieser Fähigkeit ist es auch möglich, bestehende “Altlasten” sauber in das System zu integrieren. Die einzige Bedingung, die an die einzelnen Implementierungen gestellt wird, ist, dass sie die jeweilig geforderte Schnittstelle implementieren.

Eine Grundeigenschaft, die im Zusammenhang dieser Arbeit besonders wichtig ist und über die jedes VROOM Objekt verfügen muss, ist die Möglichkeit, seinen Zustand mittels eines Writer-Objektes zu externalisieren bzw. mittels eines Reader-Objektes zu internalisieren. Die dadurch erhaltenen passiven Datenrepräsentationen der Objekte bilden die Voraussetzung für den Transport der Objekte in fremde Adressräume oder für die Speicherung der Objekte.

3.3.2 Unterscheidung Objekte und Werte

In der VROOM Architektur wird zwischen Werten und Objekten unterschieden.

Werte sind keine Objekte sondern unveränderbare Grundelemente, mit denen Objekte zusammengebaut werden können. Werte haben keine Identität, was zur Folge hat, dass sie problemlos kopiert werden können, ohne irgendwelche Integritätsbedingungen zu verletzen. Bei den Werten erfolgt eine Trennung in primitive und nicht primitive Werte. Ein primitiver Wert ist ein Wert, für den eine direkte Abbildung in die jeweilige Zielsprache (bei uns: Java) existiert wie z.B. integer, float, char, etc. Nicht primitive Werte sind Systemtypen wie Identifier, AnyException, etc. oder aber domänenspezifische Typen wie Currency, SocialSecurityNumber, etc. Da für nicht primitive Werte keine direkte Abbildung in die jeweilige Sprache existiert, müssen sie notgedrungen mit Objekten der Zielsprache realisiert werden. Aus der Sicht der VROOM sind sie jedoch keine Objekte. In unserer Java Implementierung der VROOM werden Werte auf Java-Interfaces abgebildet.

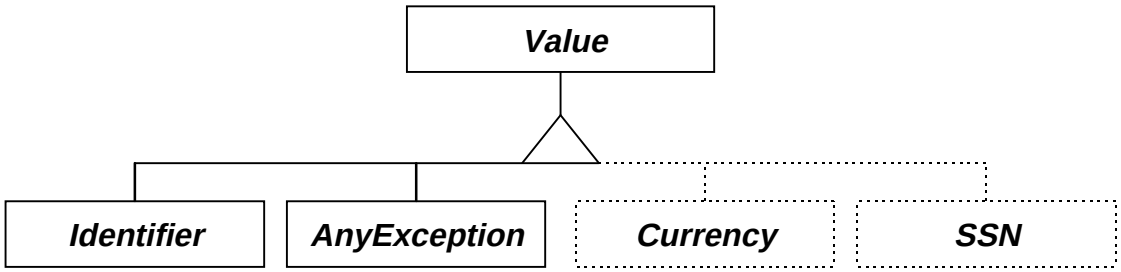


Abb. 3.3: VROOM Wertetypen sind Subtypen von Value

Objekte haben dagegen eine eindeutige Identität und verfügen über eine Lebenszeit, d.h. sie werden irgendwann instantiiert und später wieder gelöscht. Die Identifizierung der Objekte erfolgt über Identifier, welche ihrerseits Werte sind (s.o.). Objekte können an verschiedenen Stellen referenziert werden und bestehen aus Werten und Referenzen auf andere Objekte. Ein Objekt ändert nie die Werte seiner Attribute, sondern weist den Attributen bei Bedarf neue Werte zu. In unserer Java Implementierung der VROOM werden VROOM-Klassen ebenfalls durch Java-Interfaces repräsentiert. Gleichzeitig definieren diese Java-Interfaces die Schnittstelle der Instanzen dieser VROOM-Klasse. In der momentanen Implementierung der VROOM gilt die Konvention, dass alle Interfaces, die von AnyObject erben, eine VROOM-Klasse definieren. Dies hat zur Folge, dass die für ferne Methodenaufrufe nötigen Stub- und Adapterklassen generiert werden müssen (s.u.).

3.3.3 Methodenaufrufe

Prinzipiell gibt es in einem verteilten objektorientierten System zwei verschiedene Arten von Methodenaufrufen: lokale und ferne. Da es ein Ziel der VROOM Architektur ist, für den Benutzer die Verteilung der Objekte so weit wie möglich transparent zu halten, wird ein Mechanismus gebraucht, der Methodenaufrufe abfängt und je nach Art (fern oder lokal) die nötige Aufrufweise auswählt. Genau diese Aufgabe übernehmen im VROOM System die Stub-Objekte. Daraus folgt, dass ein Benutzer - obwohl es so aussieht - nie direkt Referenzen auf sein eigentliches Zielobjekt hat, sondern auf das Stub-Objekt. Zu jedem Zielobjekt gibt es mindestens ein Stub-Objekt. Die Stub-Objekte implementieren exakt die gleiche Schnittstelle wie die eigentlichen Objekte, um dem Benutzer die Illusion zu vermitteln, es handle sich um das gewünschte Zielobjekt. Erfolgt nun ein Methodenaufruf auf einem Stub-Objekt, muss dieses zuerst abklären, ob das Zielobjekt lokal oder entfernt vorhanden ist. Bei der lokalen Variante wird der Aufruf direkt an das entsprechende Zielobjekt weitergeleitet (Abb. 3.4).

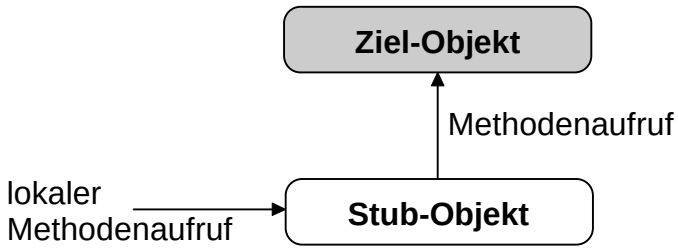


Abb. 3.4: Lokaler Methodenaufruf

Liegt das gewünschte Objekt aber auf einem entfernten Rechner, wird ähnlich wie im CORBA Dynamic Invocation Interface (DII) [OMG95, Sie96] ein Request-Objekt erzeugt, das diesen Aufruf repräsentiert. Dieses Request-Objekt wird nun via das eigene Klassen-Objekt an das entsprechende Klassen-Objekt im entfernten Adressraum übermittelt. Jedes Klassen-Objekt verfügt über ein Adapter-Objekt, das in der Lage ist, aus dem ankommenden Request-Objekt wieder einen Methodenaufruf zu generieren (Request Dispatching). Dabei wird analog zu einem lokalen Methodenaufruf (s.o.) die entsprechende Methode im Stub-Objekt und damit im gewünschten Zielobjekt aufgerufen. Die durch den Aufruf erzeugten Rückgabewerte und eine allfällige Exception werden in einem Reply-Objekt via den gleichen Weg an den Aufrufer zurückgegeben (Abb. 3.5).

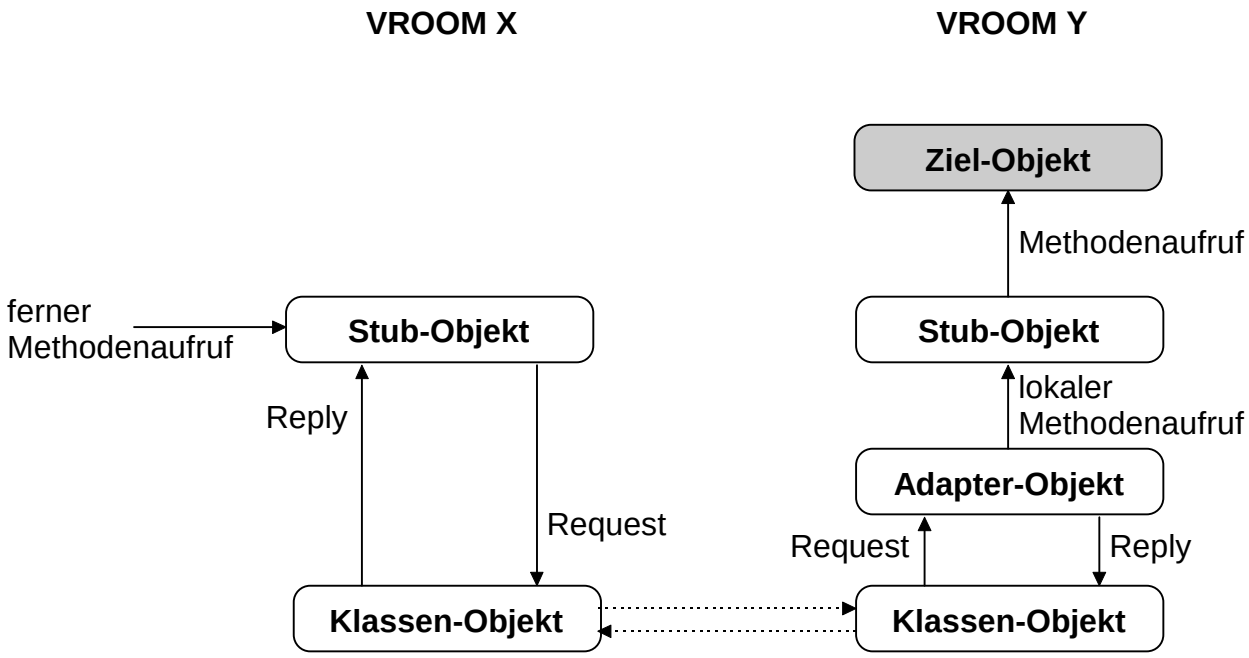


Abb. 3.5: Ferner Methodenaufruf zwischen zwei VROOMs

Ein detaillierte Erklärung der VROOM Methodenaufrufe erfolgt in Kapitel 7.

3.3.4 Erzeugung von Stub- und Adapterklassen

Für die zuvor mehrfach angesprochenen Stub und Adapter-Objekte müssen die erforderlichen Klassen analog zu CORBA vom Programmierer nicht selbst erstellt sondern können automatisch generiert werden. Statt aber wie in CORBA eine Interface Definition Language (IDL) [OMG95] als Ausgangspunkt zu nehmen, werden der Einfachheit halber in der momentanen Implementierung der VROOM die Stub- und Adapter-Klassen direkt aus den entsprechenden Java-Interfaces generiert. Jedes Java-Interface repräsentiert eine VROOM-Klasse und definiert die Schnittstelle der Exemplare dieser VROOM-Klasse.

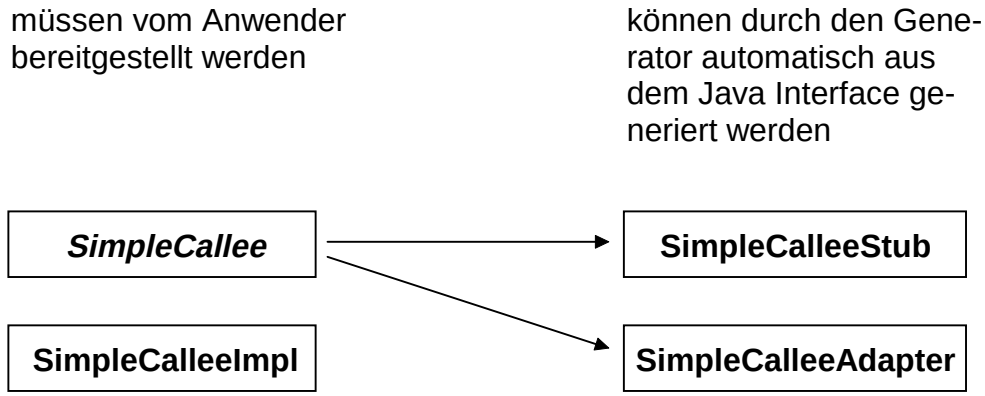


Abb. 3.6: Generierung von Stub- und Adapterklassen

Der Benutzer muss lediglich die Schnittstelle der VROOM-Klasse (ein Java Interface) und die Implementierung ihrer Instanzen (eine Java Klasse) bereitstellen. Mit dem Generator können danach aus der Schnittstelle die entsprechenden Stub- und Adapter-Klassen erzeugt werden.

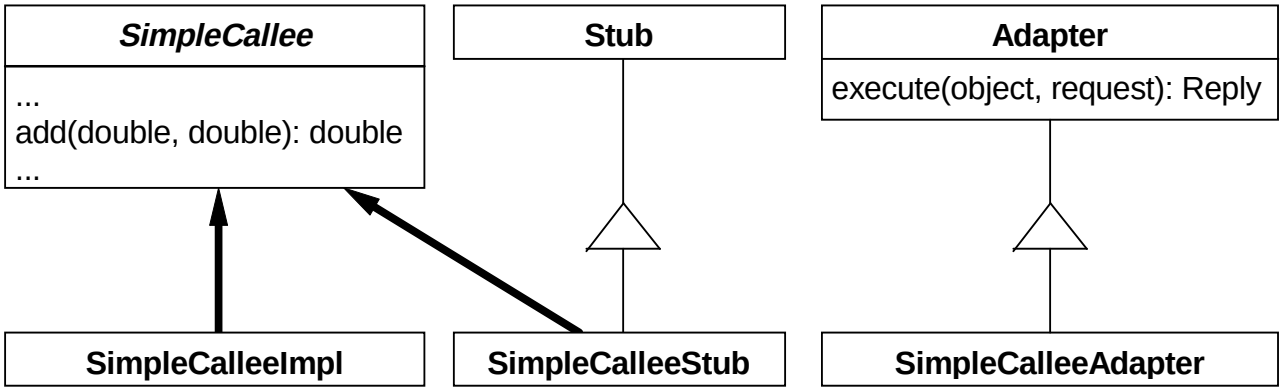


Abb. 3.7: SimpleCallee Beispiel

Abb. 3.7 illustriert noch einmal anhand der Klasse SimpleCallee in einem Klassendiagramm, wie Schnittstelle (SimpleCallee), Implementierung (SimplCalleeImpl), Stub (SimpleCallee-Stub) und Adapter (SimpleCalleeAdapter) zusammenhängen.

4 Kommunikationssubsystem

In diesem Kapitel geht es um das Kommunikationssubsystem, dessen Hauptaufgabe es ist, Objekte zwischen Prozessen zu transportieren und diesen Dienst anderen Komponenten, den sog. Listeners, zur Verfügung zu stellen. Das hier vorgestellte System ist mächtiger als die Common Object Request Broker Architecture (CORBA), da es nicht nur die für ferne Methodenaufrufe notwendigen Requests und Replys sondern ganz allgemein Objekte übermitteln kann. Das Kommunikationssubsystem stellt aber noch weitere Dienste z.B. für die Lokalisierung von entfernten Objekten zur Verfügung. Neben der Beschreibung der Architektur der Kommunikationsinfrastruktur wird darauf eingegangen, wie diese konkret basierend auf OrbixWeb, der Java CORBA Implementierung der Firma Iona, realisiert wurde.

Bei der Realisierung des Kommunikationssubsystems standen Performanz-Aspekte nicht im Vordergrund. Es ging vielmehr darum, ein offenes und erweiterbares System zu entwerfen.

4.1 Migration oder Replikation von Objekten?

Wie bereits in Kapitel 2.2 beschreiben, bieten Systeme, die einen allgemeinen Mechanismus zum Transport von Objekten zwischen Prozessen anbieten, zahlreiche Vorteile. In dem vorliegenden VROOM-System ist prinzipiell jedes Objekt transportierbar. Der Transport eines oder mehrerer Objekte in den Adressraum einer anderen VROOM bedeutet dabei eine Migration. Im Gegensatz zu der Replikation, bei der in anderen Adressräumen Kopien eines Objektes angelegt werden, die konsistent mit dem Originalobjekt gehalten werden müssen, werden bei der Migration die Objekte tatsächlich verschoben. Die Sender-VROOM verfügt danach nicht mehr über die transportierten Objekte. Ohne Zweifel wäre auch Replikation eine gute Variante; das Problem dabei ist nur, dass ein solches System in der Lage sein muss, die Konsistenz der Replikate mit dem Originalobjekt zu gewährleisten. Diese Aufgabe ist in einem verteilten System aber alles andere als trivial. Da zudem für die Diplomarbeit nur eine beschränkte Zeitspanne zur Verfügung steht, wurde der Entschluss gefasst, sich auf Migration zu beschränken.

4.2 Architektur

Dieses Kapitel befasst sich mit der Architektur des Kommunikationssubsystems. Mit der Abb. 4.1 wird versucht, zuerst einen Gesamtüberblick zu vermitteln. Weiter unten wird dann auf die einzelnen Komponenten und ihr Zusammenspiel detaillierter eingegangen.

Im Gegenteil zu CORBA bietet das Kommunikationssubsystem direkt einige wenige Dienste an, die lokal in jeder VROOM verfügbar sind: der LocalLookupService (Kap. 4.2.3), der

RemoteLookupService (Kap. 4.2.4) und der VroomLookupService (Kap. 4.2.5). In CORBA sind sämtliche Dienste nicht Bestandteil der Kommunikationsinfrastruktur sondern als externe Dienste optional bei Bedarf verfügbar (z.B. der Naming Service).

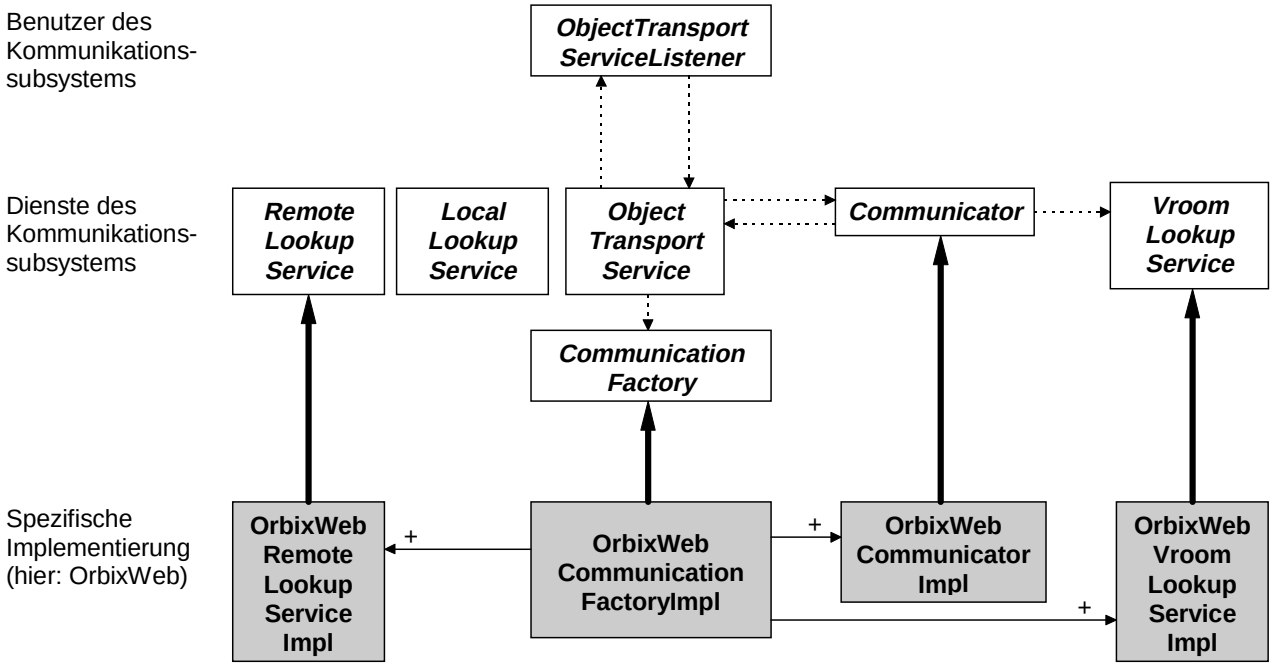


Abb. 4.1: Gesamtüberblick über das Kommunikationssubsystem

Die beteiligten Kommunikationssubsystem-Komponenten lassen sich grob in drei Kategorien aufteilen: Benutzer, Dienste und die spezifischen Implementierung (in Abb. 4.1 grau dargestellt). Beim Entwurf wurde grossen Wert darauf gelegt, die Offenheit des Systems zu bewahren und sich nicht auf eine spezifische Implementierung zu beschränken. Das Framework bietet Schnittstellen (“hooks”) an, deren Funktionalität in konkreten Subklassen basierend auf einer Kommunikationstechnologie implementiert werden muss [GHJV95]. In Abb. 4.1 sind diese Schnittstellen die Klassen: RemoteLookupService, VroomLookupService, Communicator und CommunicationFactory.

Die hier gewählte Implementierung basiert auf der Common Object Request Broker Architecture (CORBA) (siehe Kap. 2.4.2). Es wären aber durchaus auch Lösungen denkbar, die z.B. auf Java Remote Method Invocation (siehe Kap. 2.4.3), Sockets oder anderen Technologien basieren. Wir haben uns für OrbixWeb, die Java CORBA Implementierung der Firma Iona entschieden. Die Gründe für diesen Entscheid wurden in Kapitel 2.4.2.1 bereits erläutert.

4.2.1 ObjectTransportServiceListener

Die ObjectTransportServiceListeners setzen auf dem ObjectTransportService auf. Ein Listener ist ein Client des ObjectTransportServices, der dessen Fähigkeit, Objekte von der eigenen VROOM in den Adressraum einer anderen VROOM zu transportieren, in Anspruch nimmt. Mögliche Listeners sind zum Beispiel ein MigrationService oder der RemoteRequestService (siehe Kapitel 7.1.4). Damit Clienten des ObjectTransportService nicht nur Objekte verschicken sondern auch empfangen können, müssen sie die Schnittstelle des Object-TransportServiceListeners implementieren und sich beim ObjectTransportService registrieren

lassen. Dabei kam eine leichte Abwandlung des *Observer* Entwurfsmusters zur Anwendung [GHJV95].

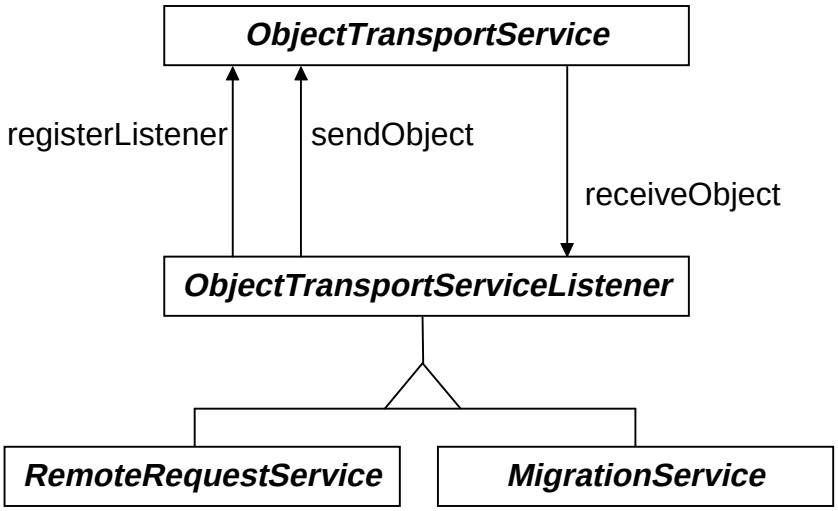


Abb. 4.2: Zusammenspiel ObjectTransportService und ObjectTransportServiceListener

Ein Objekttransport erfolgt immer zwischen zwei gleichen Listeners: z.B. der RemoteRequestService einer Sender-VROOM transportiert Request-Objekte zum RemoteRequestService einer Empfänger-VROOM (siehe Kap. 7). Schlägt der Transport fehl, weil beim Empfänger der entsprechende Listener nicht zur Verfügung steht (es erfolgte keine Registrierung beim ObjectTransportService in der Empfänger-VROOM), erhält der Sender-Listener eine “ListenerNotAvailableException”.

4.2.2 ObjectTransportService und Communicator

Die Aufgaben des ObjectTransportServices sind:

- Entgegennehmen von zu transportierenden Objekten von den Listeners (s.o.), die Serialisierung und Transformation dieser Objekte in eine externe Datenrepräsentation (siehe Kapitel 5) und das anschliessende Weiterleiten des erzeugten Datenbuffers an den Communicator.
- Entgegennehmen von ankommenden Datenbuffern (vom Communicator), die Wiederherstellung der im Datenbuffer enthaltenen Objekte und das Weiterleiten der Objekte an die Listeners (siehe auch Abb. 4.2).

Wie bereits zuvor erwähnt, entspricht der Transport von Objekten in unserem System einer Migration und nicht einer Replikation, d.h. die Objekte werden wirklich verschoben und nicht einfach kopiert. Ein transportiertes Objekt verschwindet aber nicht ganz aus der “alten” VROOM. Das Stub-Objekt bleibt weiterhin bestehen, lediglich das Implementierungsobjekt wird entfernt. Erfolgen nun weiterhin Zugriffe auf das migrierte Objekt in der “alten” VROOM, leitet das Stub-Objekt diese automatisch weiter zur “neuen” VROOM.

Beim Zusammenspiel von ObjectTransportService und Communicator kam das *Bridge* Entwurfsmuster [GHJV95] zur Anwendung. Dadurch konnte eine klare Trennung in Konzept und Implementierung erreicht werden. Der Communicator ist eine abstrakte Klasse, die für eine spezifische Implementierung erweitert werden muss (in unserem Fall: OrbixWebCommunicatorImpl).

In unserer Implementierung wurde das physikalische Transportieren eines Datenbuffers auf einen CORBA Methodenaufruf abgebildet. Jede VROOM wird auf der CORBA Ebene durch genau ein CORBA-Objekt (OrbixWebCommPortImpl) repräsentiert. Möchte ein OrbixWebCommunicatorImpl nun einen Datenbuffer an eine entfernte VROOM übermitteln, so ruft er einfach die Methode “receiveBuffer(…)” des entsprechenden OrbixWebCommPortImpl-Objektes auf. Dieses nimmt ankommende Datenbuffer entgegen und leitet sie weiter via OrbixWebCommunicatorImpl an den ObjectTransportService in der Empfänger-VROOM (Abb. 4.3). Falls beim Transportieren des Datenbuffers ein Kommunikationsfehler auftritt, was verschiedene Gründe haben kann (siehe Kap. 2.1), gibt der ObjectTransportService seinen Clienten, den Listenern, eine “Communication-FailureException” zurück.

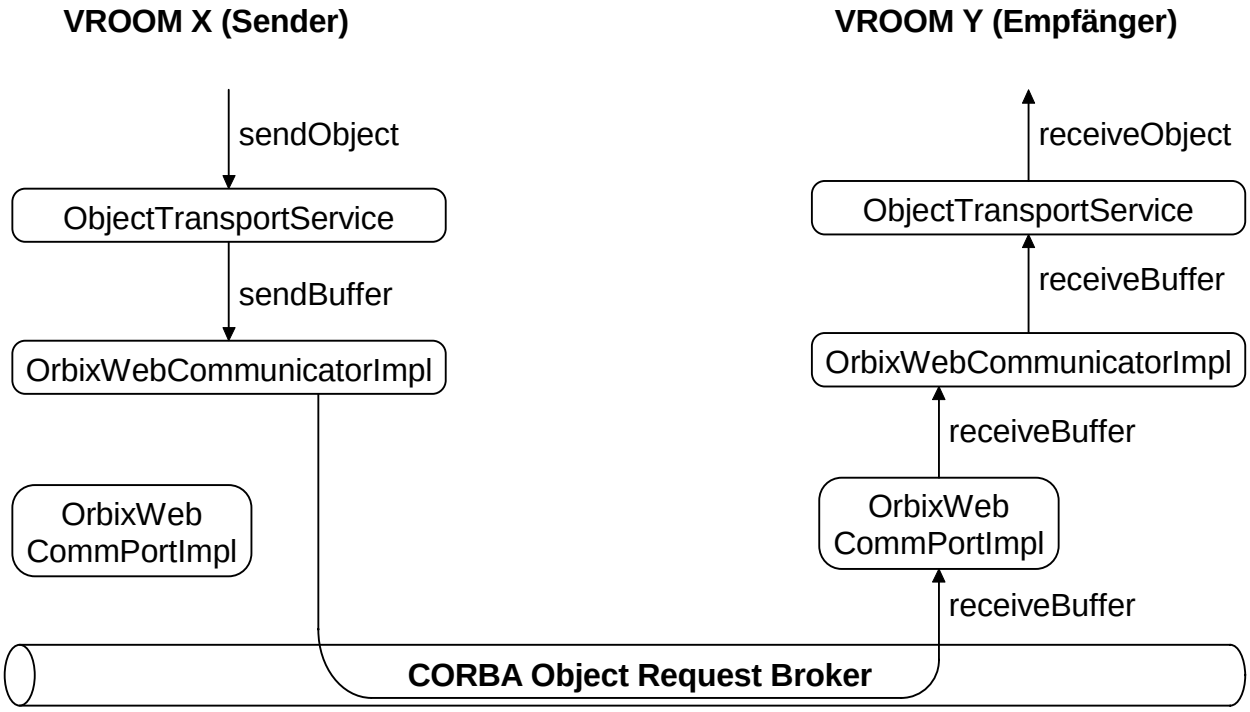


Abb. 4.3: Objekt-Interaktionsdiagramm für den Objekttransport

```
// CORBA Interface Definition Language File for OrbixWebCommPort
module OrbixWeb
{
   interface OrbixWebCommPort
   {
            // receive buffer and return an occuring exception (as string)
            string receiveBuffer(
                     in string fromVroomName,
                     in string fromHostName,
                     in string fromLocalNamingContextIdentifier,
                     in OrbixWebCommPort fromCommPort,
                     in string toListenerName,
                     in string buffer
            );

            // get identifier of local naming context (as string)
            string getLocalNamingContextIdentifier();
   };
};
```

Abb. 4.4: CORBA IDL Schnittstelle von OrbixWebCommPortImpl

Damit das Kommunikationssubsystem während des Sendens eines Datenbuffers nicht blockiert, läuft parallel ein Thread (OrbixWebEventProcessor), der ankommende Datenbuffer entgegen nimmt. Sobald der ObjectTransportService die übermittelten Objekte wiederherstellen konnte, wird ebenfalls ein Thread erzeugt, der die weitere Verarbeitung der Objekte durch die Listeners übernimmt.

4.2.3 LocalLookupService

Der LocalLookupService verwaltet alle lokalen Objekte einer VROOM. Dabei handelt es sich um eine Dienstleistung, die das Kommunikationssubsystem seinen Listeners zur Verfügung stellt.

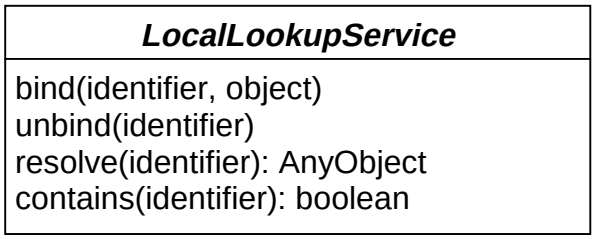


Abb. 4.5: LocalLookupService

Streng genommen ist der LocalLookupService nicht Teil des Kommunikationssubsystems. Seine Aufgabe ist es, Abbildungen von Objekt-Identifizierern auf Objekte zu verwalten. Da es sich dabei um eine allgemeine Dienstleistung handelt, ist er nicht auf eine spezifische Implementierung angewiesen (siehe auch Abb. 4.1). Seine Aufführung in diesem Kapitel wird damit begründet, dass er oft im Zusammenhang mit dem Kommunikationssubsystem gebraucht wird.

4.2.4 RemoteLookupService

Prinzipiell kann jedes VROOM Objekt während seiner Lebenszeit bei Bedarf auf andere VROOMs migriert werden. Damit Objekte überhaupt von einem Konten zu einem anderen verschoben werden können und trotzdem weiterhin auffindbar bleiben, müssen sie global eindeutig identifiziert werden können. Es braucht zudem einen Mechanismus, der es erlaubt, entfernte Objekte jederzeit zu lokalisieren, damit diese dauernd benutzbar sind. Genau diese Aufgabe erfüllt der RemoteLookupService. Er ist in der Lage, zu einem gegebenen Objekt-Identifizierer die VROOM herauszufinden, in deren Adressraum das gewünschte entfernte Objekt zur Zeit lebt. Der RemoteLookupService verwaltet im Prinzip gegen aussen eine Menge von Abbildungen der Form “Objekt-Identifizierer -> VROOM”. Um schliesslich in der entsprechenden VROOM wirklich an das gewünschte Objekt zu gelangen, wird zudem ein “Lookup” im LocalLookupService (s.o.) der dortigen VROOM benötigt.

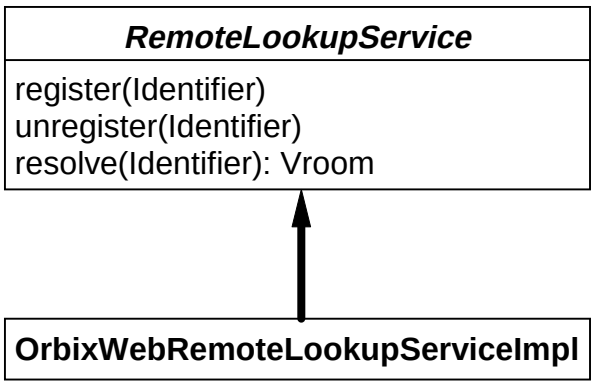


Abb. 4.6: RemoteLookupService

Ein Objekt-Identifizierer besteht in der VROOM Architektur aus folgenden drei Teilen:

- Name des Rechners, auf dem das Objekt alloziert wurde
- Name der VROOM, in deren Adressraum das Objekt alloziert wurde
- eine fortlaufende Zahl, die das Objekt innerhalb der Kombination (Rechnername, Vroomname) eindeutig identifiziert

Es ist wichtig zu bemerken, dass die Objekt-Identifizierer während der ganzen Lebenszeit der Objekte, die sie eindeutig bezeichnen, unverändert bleiben. Dies gilt insbesondere auch, wenn Objekte in den Adressraum einer anderen VROOM transportiert werden. Beim Transport eines Objektes ändert sich einzig und allein die Zuordnung des Objekt-Identifizierers von der “alten” zur “neuen” VROOM.

Die OrbixWeb spezifische Implementierung (OrbixWebRemoteLookupService) benutzt zur Implementierung des RemoteLookupServices den CORBA Naming Service. Die dreiteiligen Objekt-Identifizierer werden auf einen dreistufigen hierarchischen Namensraum abgebildet. Abb. 4.7 illustriert dies anhand eines Beispiels.

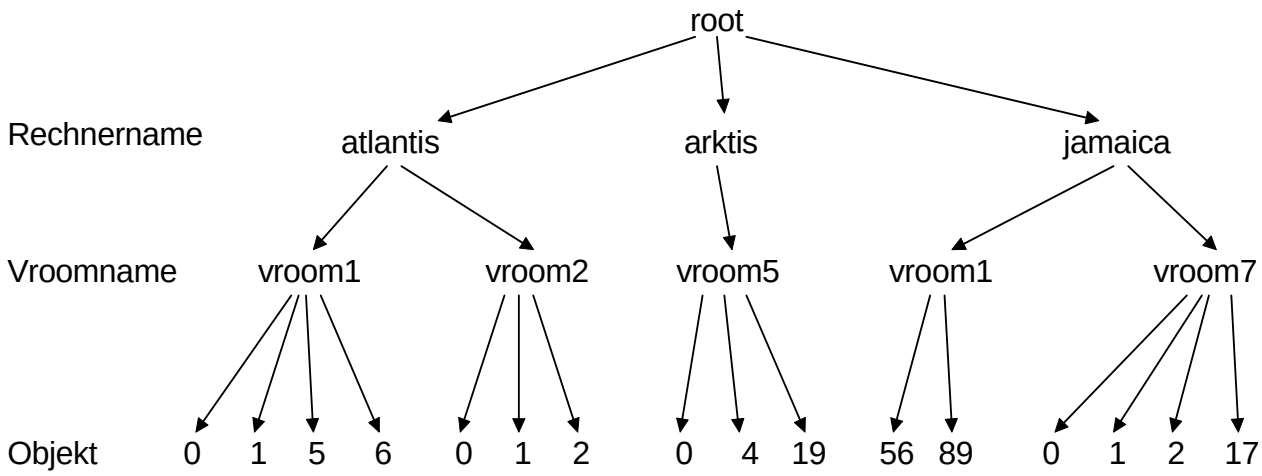


Abb. 4.7: Abbildung auf den CORBA Namensraum

Jeder Eintrag im CORBA Naming Service zeigt dabei auf das OrbixWebCommPortImpl-Objekt (s.o.) der VROOM, in deren Adressraum das Objekt zur Zeit lebt. Wird nun ein Objekt von einer VROOM zu einer anderen migriert, bleibt zwar der Name gleich, aber der Eintrag im Naming Service zeigt danach auf das OrbixWebCommPortImpl-Objekt der neuen VROOM.

Da im vorliegenden System für jede Dereferenzierung eines entfernten VROOM-Objektes ein teurer Zugriff auf den CORBA Naming Service erfolgt, um das Objekt zu lokalisieren, leidet die Performanz des Systems. Aus diesem Grund wurde ein einfacher Cache realisiert, um diese Zugriffe zu beschleunigen. Häufige gleiche Lokalisierungsanfragen können oft schon aus dem Cache heraus beantwortet werden und benötigen daher keinen Zugriff mehr auf den CORBA Naming Service. Diese Lösung hat aber den Nachteil, dass aufgrund eines veralteten Cacheeintrages einzelne Objekte in seltenen Fällen zu einer falschen VROOM transportiert werden können. Um dies einzudämmen, lebt ein Cacheeintrag nur gerade eine kurze Zeit (zur Zeit: eine Minute) bis er verworfen wird. Aufgrund der in der Regel hohen Lokalität der Zugriffe, konnte dank des Caches eine deutliche Leistungssteigerung erreicht werden.

4.2.5 VroomLookupService

Es ist sinnvoll, die einzelnen VROOMs ebenfalls durch Objekte zu repräsentieren. Die VROOM-Objekte speichern wichtige Informationen wie z.B. den Vroom- und den Rechnernamen, für die sie stehen. Beim Zugriff auf entfernte VROOMs spielen diese Objekte eine zentrale Rolle.

Der VroomLookupService - ebenfalls ein lokaler Service wie der LocalLookupService und der RemoteLookupService - merkt sich automatisch alle VROOMs, mit denen er schon einmal kommuniziert hat. Ausserdem ist er in der Lage, neue VROOMs ausfindig zu machen. Er ist quasi die Anlaufstelle für die Kommunikation mit fernen VROOMs.

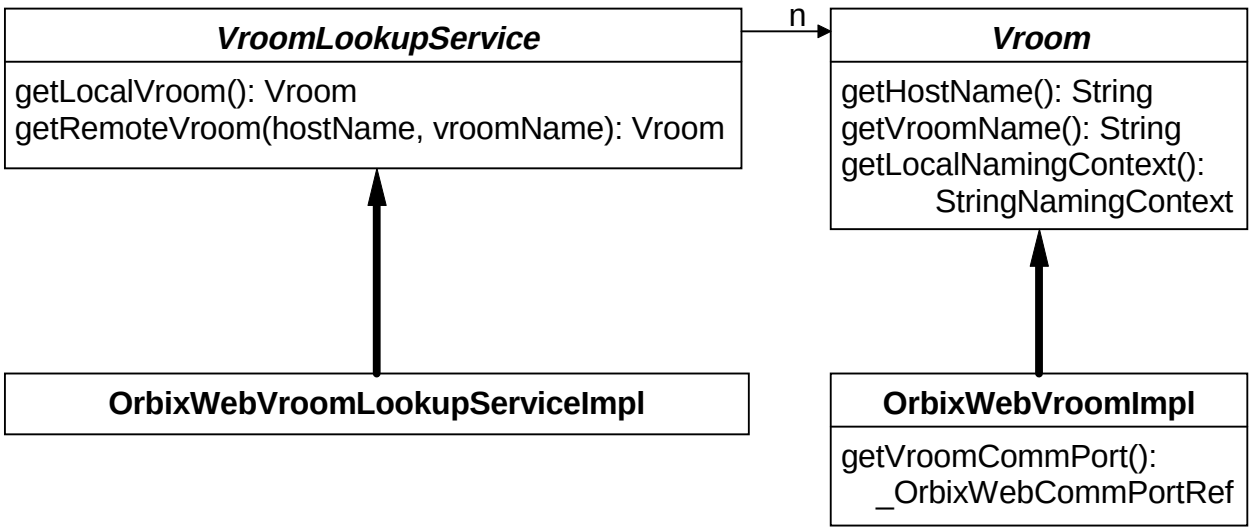


Abb. 4.8: VroomLookupService

Damit der OrbixWebVroomLookupService neue VROOMs ausfindig machen kann, benutzt er den Orbix Daemon (siehe Kap. 2.4.2.1). Dieser Daemon Prozess, der im übrigen auf jedem Rechner laufen muss, der eine VROOM besitzt, ermöglicht es, zu einer VROOM eine Verbindung aufzubauen. Falls ein benötigter VROOM Prozess noch nicht am Laufen ist, wird dieser durch den Daemon automatisch aufgestartet, was sich als sehr nützlich erwiesen hat.

4.2.6 CommunicationFactory

Die CommunicationFactory, eine Anwendung des *Abstract Factory* Entwurfsmusters [GHJV95], kapselt das ganze Konfigurationswissen des Kommunikationssubsystems zentral an einem Ort. Sie übernimmt die Erzeugung der implementationsspezifischen Klassen (s.o.). Dadurch ist es sehr einfach, das ganze System zu konfigurieren.

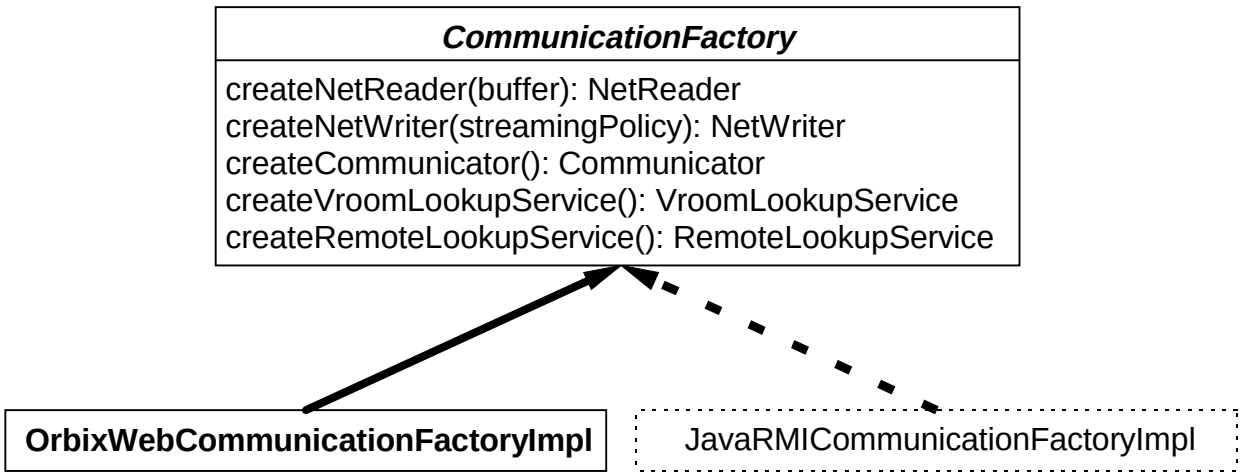


Abb. 4.9: CommunicationFactory

Wie bereits angesprochen, wäre es möglich, das Kommunikationssubsystem basierend auf einer anderen Technologie zu implementieren. Andere Varianten wären z.B. Java RMI, Sockets, DCE Remote Procedure Call, etc. In unserem Fall stellt die

OrbixWebCommunicationFactoryImpl die OrbixWeb spezifischen Implementierungen für das Kommunikationssubsystem bereit.

5 Serialisierung

Damit Objektgraphen über ein Netzwerk transportiert werden können, müssen sie zuerst serialisiert werden. Dazu kommt, dass eine geeignete, externe Datenrepräsentation ausgewählt werden muss, um die Objekte rechnerunabhängig von einem Prozess zu einem anderen transportieren zu können. Dieses Kapitel beschreibt die Grundlagen, die den Transport von ganzen Objektgraphen mit dem ObjectTransportService (siehe Kapitel 4) überhaupt erst möglich machen. Die hier beschriebene Serialisierung ist eine Anwendung des *Serializer* Entwurfsmusters [RSB+97], das sich seinerseits stark an das *Visitor* Entwurfsmuster [GHJV95] anlehnt.

5.1 NetWriter/NetReader

Die NetWriter und NetReader Schnittstellen, die Spezialisierungen der Writer bzw. Reader Schnittstellen der VROOM sind, beschreiben ein Protokoll zur Serialisierung und Transformation von ganzen Objektgraphen in eine externe Datenrepräsentation. Das ganze Wissen über Protokollformate und die Art der Serialisierung wird in den NetReadern bzw. NetWritern gekapselt. Dies hat den grossen Vorteil, dass sich die zu übertragenden Objekte nicht selbst um die Externalisierung bzw. Internalisierung kümmern müssen. Die NetWriter bzw. NetReader stellen den Objekten eine Reihe von Methoden zur Verfügung, mit denen sich sowohl primitive als auch komplexere Datentypen übertragen lassen (siehe auch Abb. 5.1). Dem NetWriter fällt die Aufgabe zu, die gewünschten Objektstrukturen zu serialisieren und in eine externe Datenrepräsentation zu bringen (Passivierung). Als Resultat liefert ein NetWriter einen Datenbuffer, der für die Übertragung über ein Netzwerk verwendet werden kann. Ein NetReader übernimmt die Wiederherstellung der übertragenen Objektstrukturen (Aktivierung), d.h. er ist in der Lage, komplexe Objektstrukturen aus einem Datenbuffer heraus wieder zu rekonstruieren. Konkrete NetWriter und NetReader treten immer in Paaren auf, die gemeinsam ein vereinbartes Protokoll implementieren. Unterschiedliche NetWriter/NetReader-Paare realisieren unterschiedliche externe Datenrepräsentationen.

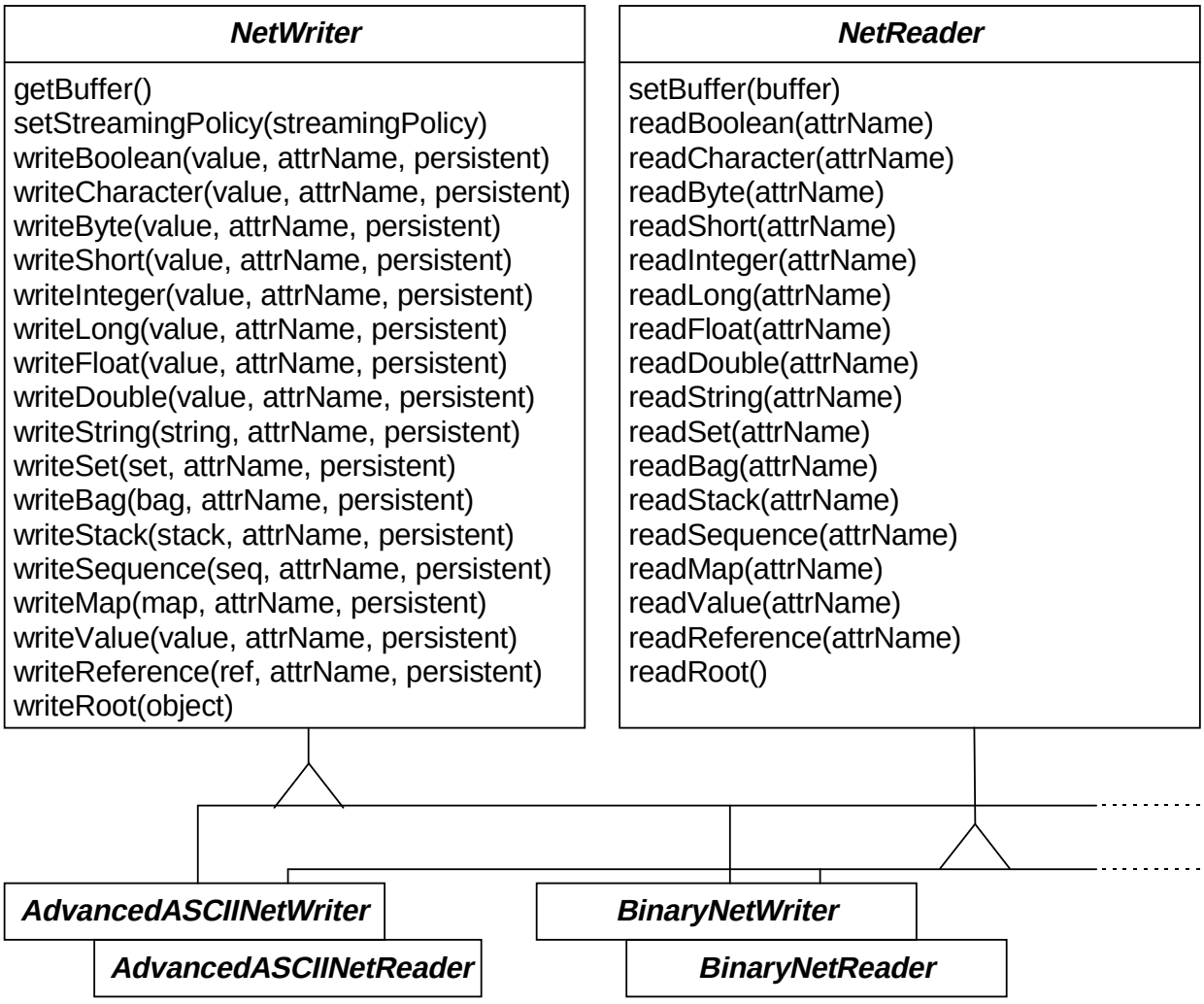


Abb. 5.1: NetWriter/NetReader-Paare

Im Rahmen dieser Diplomarbeit wurde aus Zeit- und Einfachheitsgründen in den AdvancedASCIINetWriter/-Reader-Klassen ein Übertragungsprotokoll implementiert, das auf dem American Standard Code for Information Interchange (ASCII) beruht. ASCII als Ausgangspunkt zu wählen, ist sicher keine schlechte Wahl, da ASCII ein Standard ist, der international anerkannt und weit verbreitet eingesetzt wird. Zudem hat es sich im Rahmen des Beyond-Sniff Projektes, das am Ubilab ausgeführt wurde, gezeigt, dass die Effizienz bei der Verwendung von ASCII-basierten Protokollen nicht besonders leidet. Binäre Datenrepräsentationen erzeugen zwar wesentlich kompaktere Datenbuffer, was zu schnelleren Übertragungszeiten führt, dafür müssen andere Schwierigkeiten, wie z.B. die ganze big-endian/little-endian Problematik bei unterschiedlicher Hardware gelöst werden. Ein weitere Variante wäre, einen Datenbuffer, der durch einen ASCII-basierten NetWriter erstellt wurde, nachträglich mit einem Kompressionsalgorithmus zu verkleinern.

Auf die Offenheit und Erweiterbarkeit um neue Protokolle wurde bewusst geachtet. Es ist sehr leicht möglich, das bestehende Framework um neue Protokolle zu erweitern. Dies geschieht einfach, indem neue Klassenpaare realisiert werden, die die NetWriter bzw. NetReader Schnittstellen implementieren.

5.2 Protokollerkennung auf der Empfängerseite

Das für die Kommunikation benutzte NetWriter-Protokoll wird mittels einer *Abstract Factory* [GHJV95] (siehe auch Kap. 4.2.6) beim Sender konfiguriert. Es wäre aber sehr unflexibel, wenn auch der Empfänger fix nur ein Protokoll verwenden würde. Aus diesem Grund schreibt der NetWriter zu Beginn den Protokollnamen und die Protokollversion in den Datenbuffer (siehe Abb. 5.4). Anhand dieser Informationen wird beim Empfänger des Datenbuffers die geeignete NetReader-Implementierung ausgewählt, um die enthaltenen Objektstrukturen wiederherzustellen. Trotz dieser Flexibilität muss natürlich beim Empfänger die entsprechende NetReader-Implementierung existieren. Unterstützt ein Empfänger das verwendete Protokoll nicht, erhält der Sender vom Empfänger eine "ProtocolNotSupportedException". Der Sender ist nun frei, einen anderen NetWriter auszuwählen und die Übermittlung erneut zu versuchen.

5.3 Serialisierung von einzelnen Objekten

Damit nun ein einzelnes Objekt mittels eines NetWriter/-Reader-Paares passiviert bzw. aktiviert werden kann, müssen die folgenden beiden Bedingungen erfüllt werden.

- Die entsprechende Klasse muss über einen Konstruktor ohne Argumente verfügen. Dies ist nötig, da die Objekte beim Lesen vom NetReader durch den Java-Mechanismus "class.newInstance()" erzeugt werden. Da Java-Interfaces keine Konstruktoren beinhalten können, kann der Compiler nicht überprüfen, ob solch ein argumentloser Konstruktor auch implementiert wurde. Ein Versäumnis kann daher erst zur Laufzeit erkannt werden.
- Die Objekte müssen die beiden Methoden "writeOn(Writer)" bzw. "readFrom(Reader)" implementieren. Dies ist für VROOM-Objekte automatisch gegeben, da diese die Schnittstelle von "AnyObject" implementieren müssen (siehe Bsp. "Customer" und "Account" in Abb. 5.2).

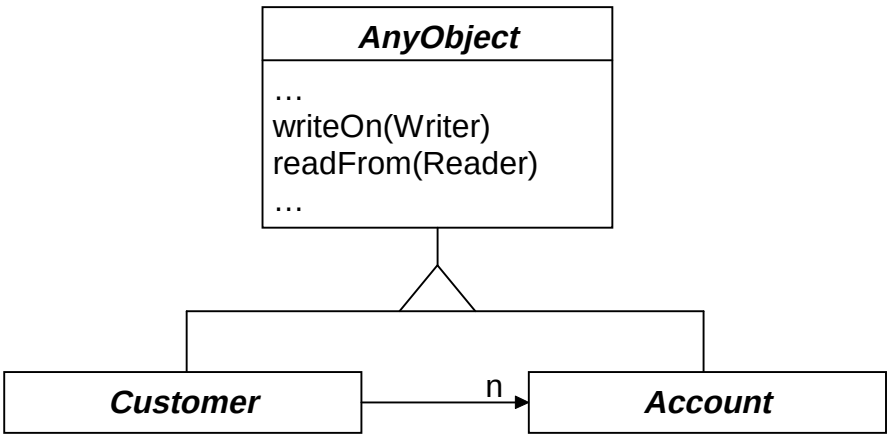


Abb. 5.2: Beispiel Customer und Account

In diesen beiden Methoden muss ein Objekt seine Instanzvariablen mittels des übergebenen NetWriters bzw. NetReaders externalisieren bzw. internalisieren. Am Beispiel von "Customer" könnten diese Methoden etwa wie folgt aussehen (Abb. 5.3):

```
public void writeOn(Writer writer)
{
    super.writeOn(writer);
    writer.writeString(“name”, name, true);
    writer.writeString(“address”, address, true);
    writer.writeSet(“accounts”, accounts, true);
}

public void readFrom(Reader reader)
{
    super.readFrom(reader);
    name = reader.readString(“name”);
    address = reader.readString(“address”);
    accounts = reader.readSet(“accounts”);
}
```

Abb. 5.3: writeOn/readFrom Methoden am Beispiel der Klasse Customer

Mit dem NetWriter und NetReader Protokoll hat ein Objekt die Möglichkeit, primitive und komplexere Wertetypen und Objektreferenzen zu schreiben bzw. zu lesen. Ob beim Schreiben einer Objektreferenz nachgegangen wird, um das referenzierte Objekt zu schreiben, oder ob nur die Objektreferenz selbst geschrieben wird, entscheidet eine sog. “StreamingPolicy”. Diese “StreamingPolicy” ist eine Anwendung des *Strategy* Entwurfsmusters [GHJV95]. Damit wird die Entscheidung über die Schreibtiefe unabhängig von der verwendeten NetWriter-Implementierung ausgelagert. Mehr dazu in Kapitel 6.

Im nachfolgenden Unterkapitel wird der AdvancedASCIINetWriter/-Reader beschrieben, eine Implementierung eines NetWriter/-Reader Paares, die im Rahmen dieser Diplomarbeit entwickelt wurde.

5.4 AdvancedASCIINetWriter/-Reader

Wie oben bereits erwähnt, externalisieren bzw. internalisieren die einzelnen Objekte mittels NetWriter- und NetReader-Implementierungen ihre Attributwerte. Damit die übermittelten Objektgraphen beim Empfänger auch wirklich wieder korrekt rekonstruiert werden können, braucht es aber noch mehr Informationen als nur die Werte der Instanzvariablen. Für jedes Objekt müssen der Objekt-Identifizierer, der VROOM-Klassenname und der Name der Implementierungsklasse in den Buffer geschrieben werden.

Die Objekt-Identifizierer werden hier quasi als persistenten Ersatz für die flüchtigen Hauptspeicheradressen zur Rekonstruktion der Objektgraphen durch den NetReader gebraucht. Sie werden aber auch dazu gebraucht, um sicherzustellen, dass zyklische Objektstrukturen korrekt serialisiert und deserialisiert werden können.

Der VROOM-Klassenname wird benötigt, damit der NetReader weiss, welches VROOM-Klassenobjekt das übermittelte Objekt im eigenen Adressraum verwaltet. Der Einfachheit halber wird angenommen, dass die beim Sender verwendeten VROOM-Klassen auch in der Empfänger-VROOM vorhanden sind. Ist dies nicht der Fall, muss auch noch die VROOM-Klasse selbst übermittelt werden. Diese Ausnahmesituation wurde erkannt aber im Rahmen dieser Diplomarbeit nicht behandelt. Deren Behandlung müsste aber für den wirklich universellen Gebrauch der VROOM-Architektur noch nachimplementiert werden.

Der Name der Implementierungsklasse (bei unserer Java-VROOM-Implementierung ist dies der Klassenname des Java-Objektes) wird mitgegeben, damit der Empfänger das Objekt instantiieren kann. Der NetReader versucht immer zuerst in seiner VROOM ein Objekt dieser Implementierungsklasse zu erzeugen. Falls dies jedoch fehlschlägt, weil die konkrete Implementierungsklasse beim Empfänger nicht vorhanden ist, wird via VROOM-Klassenobjekt eine Instanz einer Default-Implementierungsklasse alloziert.

Für jedes Attribut wird neben dem Wert auch der Attributnamen übermittelt. Dies ermöglicht die Entkopplung der Lesereihenfolge der Attribute beim NetReader von der Schreibreihenfolge beim NetWriter. Zudem funktioniert das Protokoll auch noch falls - verursacht durch unterschiedliche Klassenversionen - einzelne Attribute fehlen.

Falls es sich bei einem Attribut um eine Objektreferenz handelt, werden der Objekt-Identifizierer, der VROOM-Klassenname und ein Flag in den Buffer geschrieben. Das Flag gibt an, ob nur die Referenz mitgegeben werden soll (in diesem Fall wird beim Empfänger nur ein Stub-Objekt ohne Implementierungsobjekt erzeugt) oder ob das referenzierte Objekt auch übermittelt werden soll. Wird das referenzierte Objekt auch übermittelt, wird zusätzlich der Name der Implementierungsklasse in den Buffer geschrieben.

5.4.1 EBNF des verwendeten Protokolls (vereinfachte Version)

In diesem Abschnitt wird in der Extended Backus Naur Form (EBNF) kurz das Protokoll vorgestellt, das für den AdvancedASCIINetWriter/-Reader verwendet wurde. Dabei handelt es sich zur besseren Illustration um eine vereinfachte Version ohne Symbolkompression (s.u.).

```
BUFFER      = STRING_DELIMITER ProtocolName
              STRING_DELIMITER ProtocolVersion { OBJECT }.
OBJECT      = VroomClassName ImplClassName
              STRING_DELIMITER VroomObjectIdentifier
              STRING_DELIMITER { AttributeName ATTR_DATA }
              BLOCK_END_DELIMITER.
ATTR_DATA   = BOOL | CHAR | BYTE | SHORT | INT | LONG | FLOAT
              | DOUBLE | STRING | SET | BAG | STACK |
              SEQUENCE | MAP | VALUE | REFERENCE.
BOOL        = “bo” (TRUE | FALSE).
CHAR        = “c” charAsString.
BYTE        = “by” byteAsString.
SHORT       = “sh” shortAsString.
INT         = “i” intAsString.
LONG        = “l” longAsString.
FLOAT       = “f” floatAsString.
DOUBLE      = “d” doubleAsString.
STRING      = “str” STRING_DELIMITER string STRING_DELIMITER.
SET         = “set” COLLECTION.
BAG         = “bag” COLLECTION.
STACK       = “stack” COLLECTION.
SEQUENCE    = “seq” COLLECTION.
COLLECTION  = { COLL_ELEM } BLOCK_END_DELIMITER.
COLL_ELEM   = “e” ATTR_DATA.
MAP         = “map” { KEY_ELEM VALUE_ELEM }
              BLOCK_END_DELIMITER.
KEY_ELEM    = “k” ATTR_DATA.
```

```
VALUE_ELEM  =  “v” ATTR_DATA.
VALUE       =  “v” (NULL | ValueClassName STRING_DELIMITER
               valueAsString STRING_DELIMITER).
REFERENCE   =  “r” (NULL | VroomClassName STRING_DELIMITER
               VroomObjectIdentifier STRING_DELIMITER
               (LEAVE_DANGLING | DO_NOT_LEAVE_DANGLING
               ImplClassName)).
TRUE        =  1.
FALSE       =  0.
NULL        =  “n”.
LEAVE_DANGLING          =  1.
DO_NOT_LEAVE_DANGLING   =  2.
STRING_DELIMITER        =  0X.
BLOCK_END_DELIMITER     =  “~”.
```

Abb. 5.4: AdvancedASCIINetWriter/-Reader Protokollformat in EBNF

5.4.2 Symbolkompression

Die beim AdvancedASCIINetWriter/-Reader durchgeführte Symbolkompression ist eine einfache Optimierung zur Verkleinerung der erzeugten Datenbuffer. Symbole in diesem Zusammenhang sind VROOM- und Implementierungsklassennamen sowie Attributnamen. Aufgrund der Beobachtung, dass sich die obigen Namen zum Teil oft wiederholen innerhalb eines Datenbuffers, wurde der Entschluss gefasst, statt bei jedem weiteren Vorkommen eines Symbols erneut den Symbolnamen nur noch eine Nummer zu übermitteln. Lediglich beim ersten Auftreten eines Symbols wird der Symbolname und die entsprechende Zahl übermittelt. Damit kann vor allem bei Objektgraphen mit vielen Objekten der gleichen Klasse eine beträchtliche Verkleinerung des Datenbuffers erreicht werden. Bei diesem Verfahren handelt es sich um eine einfache Komprimierung mit dem Vorteil, dass sie “on-the-fly” durchgeführt werden kann und keinen separaten Durchgang erfordert.

6 Streaming Policies

Wie bereits in Kapitel 5.3 erwähnt, entscheidet ein NetWriter nicht selbst, bis zu welcher Tiefe ein Objektgraph in den Datenbuffer geschrieben und damit übermittelt wird. Der Entscheid, ob beim Schreiben einer Objektreferenz nachgegangen wird, um das referenzierte Objekt zu schreiben oder ob nur die Objektreferenz selbst geschrieben wird, wurde unter Anwendung des *Strategy* Entwurfsmusters [GHJV95] in eine sogenannte "StreamingPolicy" ausgelagert. Diese Lösung hat die grossen Vorteile, dass jede NetWriter-Implementierung mit jeder denkbaren StreamingPolicy kombiniert und zur Laufzeit des Systems ausgetauscht werden kann. Das System ist dadurch zudem offen und erweiterbar für neue Streaming Algorithmen.

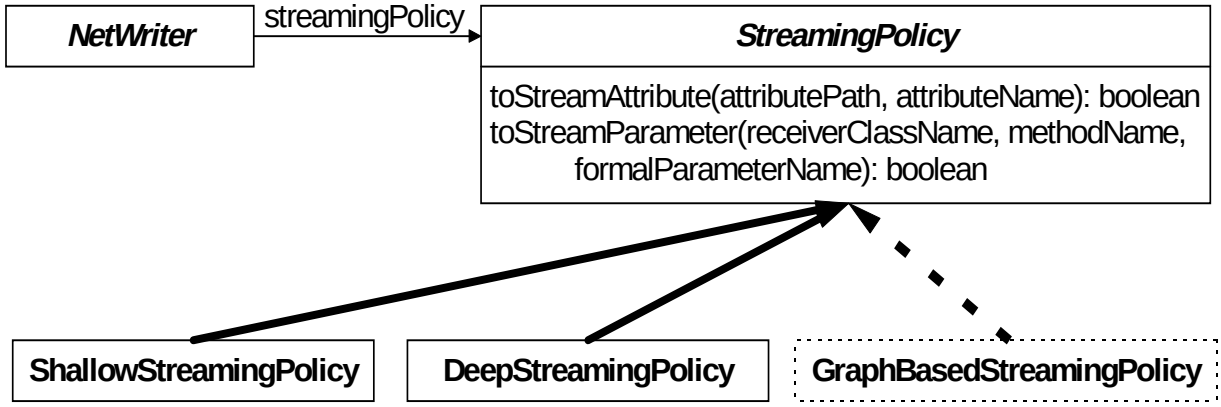


Abb. 6.1: Zusammenspiel NetWriter - StreamingPolicy

StreamingPolicies werden aber noch für mehr verwendet. Handelt es sich bei den zu übermittelnden Objekte um Request-Objekte, die für ferne Methodenaufrufe gebraucht werden, sind StreamingPolicies in der Lage zu entscheiden, ob bei Parameterobjekten das Objekt oder nur die Referenz darauf mitgegeben wird. Falls Objekte, die im Adressraum der eigenen VROOM leben, für die Dauer eines fernen Methodenaufrufes als Parameter mitgegeben werden, kann die Performanz in einem verteilten System massiv erhöht werden [JLHB88, Jul88, Lop96, Lux95], da dadurch bei der Dereferenzierung der Parameterobjekte durch das Server-Objekt kein erneuter ferner Methodenaufruf ("Callback") nötig wird.

Die folgenden zwei Kapitel gehen detaillierter auf die Möglichkeiten von Streaming Policies ein. Im dritten Unterkapitel werden einzelne, konkrete Streaming Policies behandelt.

6.1 Streaming von referenzierten Objekten

Steht nun der NetWriter vor der Entscheidung, ob er das referenzierte Objekt schreiben soll oder nicht, fragt er seine StreamingPolicy mittels der Methode "toStreamAttribute(attributePath, attributeName)". Dabei bedeuten die Argumente "attributePath" und "attributeName" folgendes:

attributePath eine Liste von Tupeln der Form (className, attributeName), die quasi den Herkunftsweg vom Ausgangsobjekt zum referenzierten Objekt beschreibt. Diese Informationen werden der StreamingPolicy übergeben, damit diese anhand des Aufrufkontextes entscheiden kann, ob das jeweilige Objekt oder nur die Referenz auf dieses übermittelt werden soll.

attributeName der Name der Instanzvariablen, die das entsprechende Objekt referenziert

6.2 Streaming von Parameterobjekten bei Methodenaufrufen

Die StreamingPolicy kann auch entscheiden, ob bei einem entfernten Methodenaufruf ein Parameterobjekt per Referenz oder - falls das Objekt im Adressraum der eigenen VROOM lebt - per Kopie übergeben werden soll. Normalerweise werden Parameterobjekte per Referenz ("call-by-reference") übergeben. Dies hat aber den Nachteil, dass - falls das gerufene Objekt das Parameterobjekt dereferenziert (dies ist nicht selten der Fall) - ein erneuter, teurer ferner Methodenaufruf nötig wird. Um diesen Fall zu umgehen, ist es oft effizienter, die Parameterobjekte für die Dauer des Aufrufes kurzfristig "auszuleihen". Dabei wird das entsprechende Objekt zusammen mit dem Request-Objekt übermittelt. Nach dem Aufruf wird das Parameterobjekt mit dem Reply-Objekt wieder zurücktransportiert. Dieses Vorgehen wurde bereits im Emerald System unter der Aufruf-Semantik "call-by-visit" vorgestellt [Jul88, JLHB88]. Mit dem Aufruf von "toStreamParameterObject(receiverClassName, methodName, formalParameterName)" entscheidet die StreamingPolicy über die Anwendung von "call-by-visit" oder "call-by-reference".

Es ist an dieser Stelle wichtig zu bemerken, dass dieses Verfahren auf den formalen und nicht auf den aktuellen Parametern beruht. Das bedeutet, dass die StreamingPolicy für alle Instanzen einer Klasse gleich entscheidet. Dieses generelle Vorgehen ist vielleicht nicht immer wünschenswert; die Spezifikationen, die der StreamingPolicy als Entscheidungsgrundlage dienen, werden dadurch jedoch massiv einfacher (und damit kann der Entscheid auch schneller gefällt werden). Zudem müsste theoretisch die StreamingPolicy aufgrund der Spezifikationen des aufgerufenen Objektes entscheiden, da nur dieses wirklich weiss, welche Parameterobjekte in seiner Methode dereferenziert werden. Dies bedingt aber, dass vor dem fernen Methodenaufruf zuerst die Spezifikation des Zielobjektes eingeholt werden müsste, was wiederum den ganzen Vorteil der "call-by-visit" Übergabe des Parameterobjektes zunichte machen würde.

6.3 Konkrete Streaming Policies

In den folgenden Unterkapiteln werden einzelne Varianten für Streaming Policies vorgestellt. Im Rahmen dieser Diplomarbeit wurden Shallow und Deep Streaming implementiert. Im Kapitel 6.3.3 über graphenbasiertes Streaming wird ein eigener Vorschlag vorgestellt, der

allerdings nicht realisiert wurde. Auch der Entscheid über die Übergabe von Parameterobjekten für effizientere ferne Methodenaufrufe (s.o.) und die Berücksichtigung des Aufrufkontextes bei Objektreferenzen wurden nicht implementiert.

6.3.1 Shallow Streaming

Beim Shallow Streaming wird nur das Ausgangsobjekt vom NetWriter in den Datenbuffer geschrieben (und damit übermittelt). Sämtliche durch Attribute des Ausgangsobjektes referenzierten Objekte oder Parameterobjekte werden nicht in den Datenbuffer geschrieben.

6.3.2 Deep Streaming

Beim Deep Streaming wird der ganze Objektgraph unabhängig von dessen Grösse und Tiefe in den Datenbuffer geschrieben (und damit übermittelt). Dies gilt auch für Parameterobjekte. Deep Streaming wird eigentlich nur in Sonderfällen angewandt (z.B. für die Speicherung aller Objekte in einer Datenbank). In der Regel braucht der Empfänger nicht alle Objekte eines Objektgraphen.

6.3.3 Graphenbasiertes Streaming

Gleich zu Beginn sei nochmals erwähnt, dass es sich bei der hier vorgestellten Streaming Policy lediglich um einen Vorschlag handelt, der aus Zeitgründen leider nicht mehr implementiert werden konnte.

Beim graphenbasierten Streaming ist die Schreibtiefe abhängig von einer Graphenspezifikation. Die Idee ist nun, dass pro VROOM eine Datei existiert, die eine textuelle Graphenspezifikation enthält. Beim Aufstarten der VROOM wird diese Datei eingelesen und geparst. Dadurch wird eine Datenstruktur erzeugt, die diese Spezifikation im Hauptspeicher repräsentiert und der graphenbasierten StreamingPolicy als Entscheidungs-grundlage dient. Diese Graphenspezifikation könnte zum Beispiel wie folgt aussehen (Abb. 6.2):

Bsp. Klasse A mit den Instanzvariablen (Objektref.) a1, a2
Klasse B mit den Instanzvariablen (Objektref.) b1, b2, b3, b4
receiver ist eine Instanz der Klasse A

für einen Methodenaufruf der Form ‘receiver.foo(A a, B b)’ könnte die Graphenspezifikation wie folgt aussehen:

```
# Spezifikation für Klasse A
class A
{
        visit parameter objects
        {
                # für die Methode ‘foo’ wird für a
                # ‘call-by-reference’ (default) und für
                # b ‘call-by-visit’ verwendet
                foo: b;
        }

        # beim Schreiben eines Objektes dieser Klasse werden
        # weder a1 noch a2 mitgegeben
        attached attributes {}
}

# Spezifikation für Klasse B
class B
{
        # keine Parameterobjekte oder alle sollen per
        # ‘call-by-reference’ übergeben werden
        visit parameter objects {}

        # beim Schreiben eines Objektes dieser Klasse werden
        # b2 und b3 mitgegeben, nicht aber b1 und b4 (default)
        attached attributes
        {
                b2, b3
        }
}
```

Dies würde bedeuten, dass bei einem Aufruf von “receiver.foo(a, b)” a per “call-by-reference” und b per “call-by-visit” übergeben werden (Wird ein Parameterobjekt in der Spezifikation nicht explizit erwähnt, wird per default “call-by-reference” angenommen.) Für das Streaming der Objektreferenzen werden die dynamischen Typen der Argumente berücksichtigt. Falls diese in unserem Beispiel A bzw. B sind, bedeutet dies, dass im Falle von b die Objekte, die durch die Attribute b2 und b3 referenziert werden, mitgestreamt werden.

Abb. 6.2: Graphenspezifikation (Vorschlag)

7 Ferne Methodenaufrufe

In diesem Kapitel wird gezeigt, wie ein ferner Methodenaufruf in die VROOM Architektur eingebettet wurde. Entgegen früherer Systeme, die ein separates Objektmodell sowohl für lokale als auch für entfernte Objekte vorsehen, verfügt das VROOM System über ein einziges Objektmodell. Dies bedeutet, dass eine Anwendung nicht wissen muss, ob sich das gerufene Objekt nun auf dem lokalen oder einem entfernten Rechner befindet. Das System ist in der Lage, dies selbst zu erkennen und die geeignete Aufrufart auszuwählen. Das Ziel war, soweit wie möglich Transparenz bezüglich der Verteilung zu erreichen. Obwohl die Aufrufe prinzipiell gleich sind, gibt es wesentliche Unterschiede zwischen lokalen und fernen Methodenaufrufen:

- ferne Methodenaufrufe dauern um Grössenordnungen länger als lokale Aufrufe bedingt durch die nötige Kommunikation
- ferne Methodenaufrufe sind um ein Vielfaches fehleranfälliger als lokale Aufrufe
- im Gegensatz zu lokalen Aufrufen, die Exactly-once Semantik unterstützen, kann für ferne Aufrufe nur At-most-once Semantik garantiert werden (s.u.); diese Tatsache kann vor der Anwendung nicht verborgen werden, da das darunterliegende Kommunikationssubsystem nicht alle Fehler maskieren kann

Das Kapitel schliesst neben der Erklärung eines VROOM Methodenaufrufes auch eine Diskussion der möglichen Fehlersituationen und deren Behandlung ein.

7.1 Funktionsweise eines VROOM Methodenaufrufes

Wie bereits in Kapitel 3 erwähnt wurde, sieht eine Anwendung nie direkt die eigentlichen Objekte sondern Stub-Objekte. Diese Stub-Objekte implementieren exakt die gleiche Schnittstelle wie die tatsächlichen Implementierungsobjekte. Da in der VROOM-Architektur potentiell jedes Objekt entfernt sein kann und sich die Lokalität der Objekte zur Laufzeit ändern kann, werden die Stub-Objekte gebraucht, um die Methodenaufrufe an die Implementierungsobjekte abzufangen. Ist das Implementierungsobjekt lokal vorhanden, wird der Aufruf direkt an dieses weitergeleitet. Ist das Implementierungsobjekt nicht lokal vorhanden, sondern liegt es zur Zeit in einer anderen VROOM, wird ein Request-Objekt erzeugt, das an das entfernte Objekt übermittelt wird. Als Antwort erhält der Stub vom entfernten Implementierungsobjekt ein Reply-Objekt. In den folgenden Unterkapiteln wird detaillierter auf die Funktionsweise von lokalen und fernen Methodenaufrufen eingegangen.

7.1.1 Lokaler Methodenaufruf

Damit ein Methodenaufruf lokal ist, muss das Implementierungsobjekt lokal in dieser VROOM verfügbar sein. Das Stub-Objekt verfügt in diesem Fall direkt über eine Referenz auf das Implementierungsobjekt.

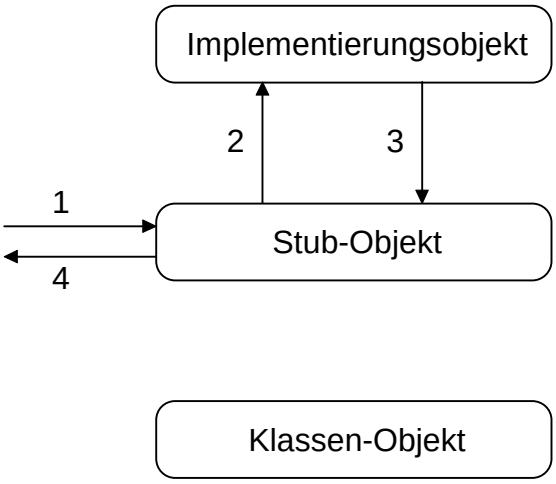


Abb. 7.1: Lokaler Methodenaufruf

Wie bereits erwähnt, sieht eine Anwendung nur Stub-Objekte. Erfolgt nun ein Methodenaufruf (Abb. 7.1) an das Stub-Objekt (1), wird dieser direkt an das Implementierungsobjekt weitergeleitet (2). Ein allfälliger Rückgabewert wird via Stub an den Aufrufer zurückgegeben (3, 4).

7.1.2 Ferner Methodenaufruf

Liegt das Implementierungsobjekt auf einer entfernten VROOM, muss mit einem fernen Methodenaufruf darauf zugegriffen werden (Abb. 7.2).

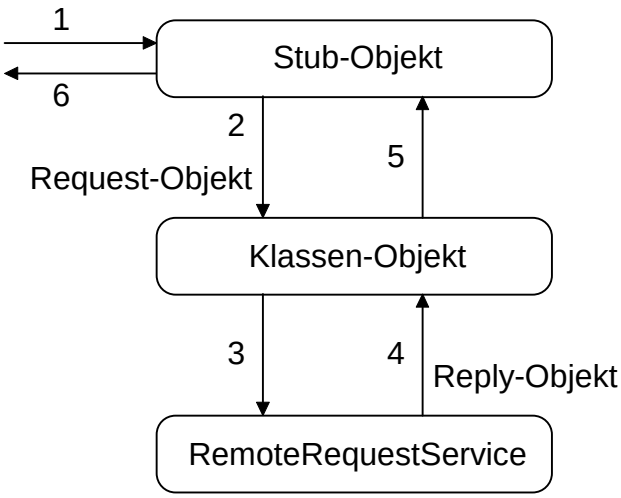


Abb. 7.2: Ferner Methodenaufruf

Erfolgt nun - analog wie beim lokalen Aufruf - ein Methodenaufruf an das Stub-Objekt (1), erzeugt dieses ein Request-Objekt und schickt es an das entsprechende Klassen-Objekt (2). Das Klassen-Objekt seinerseits leitet das Request-Objekt weiter an den RemoteRequest-Service (3). Dieser sorgt dafür, dass das entfernte Implementierungsobjekt das Request-Objekt bekommt und die gewünschte Methode aufgerufen wird (siehe Kap. 7.1.4). Als Antwort auf

den Aufruf erhält der RemoteRequestService ein Reply-Objekt, das an das Klassen-Objekt und von dort an das Stub-Objekt weitergeleitet wird (4, 5). Das Stub-Objekt extrahiert nun den Rückgabewert der Methode aus dem Reply-Objekt und gibt diesen zurück an den Aufrufer (6).

Der folgende Codeausschnitt aus einer Stub-Methode (siehe Beispiel in Kapitel 3.3.4) illustriert den Ablauf eines VROOM Methodenaufrufes (lokal oder fern).

```
public class SimpleCalleeStub
    extends StubDefImpl
    implements SimpleCallee
{
    …
    public double add(double arg0, double arg1)
    {
            if ( localTarget != null )
            {
                    // lokaler Methodenaufruf

                    SimpleCallee target = (SimpleCallee) localTarget;

                    // Aufruf weiterleiten an lokales Implementierungsobjekt
                    return target.add(arg0, arg1);
            }
            else
            {
                    // ferner Methodenaufruf

                    // Request-Objekt erzeugen
                    AnyClass requestClass =
                            (AnyClass) Sap.getMetaClass()
                            .getInstance("Vroom.Engine.Request");
                    Request request =
                            (Request) requestClass.newInstance();
                    request.setTargetIdentifier(identifier);
                    request.setOperationName
                            ("DoubleAddDoubleDouble");
                    request.addDoubleArgument(arg0);
                    request.addDoubleArgument(arg1);

                    // Request-Objekt weiterleiten an Klassen-Objekt
                    Reply reply = localClass.forward(request);
                    RuntimeException exception = (RuntimeException)
                            reply.getException();

                    // allfällige Exception werfen
                    if (exception != null) { throw exception; }

                    // Rückgabewert extrahieren und zurückgeben
                    return (double) reply.getDoubleArgument(0);
            }
```

```
    }
    …
}
```

Abb. 7.3: VROOM Methodenaufruf (Stub-Objekt für Objekte der Klasse ‘SimpleCallee’)

Ob ein Stub-Objekt über ein lokales Implementierungsobjekt verfügt, entscheidet wie bereits in Kapitel 3 beschrieben das Klassen-Objekt. Die Klassen-Objekte verwalten ihre Stub-Objekte und ihre Implementierungsobjekte. Sie sind gewissermassen Dreh- und Angelpunkt in der VROOM Architektur.

7.1.3 Request- und Reply-Objekte

Für Methodenaufrufe auf entfernten Objekten werden Request- und Reply-Objekte erzeugt.

Ein Request-Objekt beinhaltet die folgenden Informationen:

- Identifizierer des Zielobjektes, das die Methode ausführen soll
- Methodennamen
- Argumentliste (mit IN und INOUT Argumenten)

Ein Reply-Objekt beinhaltet:

- Identfizierer des Request-Objektes (s.u.)
- Argumentliste (mit OUT und INOUT Argumenten)
- allfällige Exception, die durch den Aufruf verursacht wurde

Da gleichzeitig mehrere Requests hängig sein können (verursacht durch mehrere Threads) und die Reply-Objekte in unterschiedlicher Reihenfolge zum Absenden der Request-Objekte eintreffen können, braucht es einen Mechanismus, um die Reply-Objekte den Request-Objekten eindeutig zuzuordnen. Da jedes VROOM Objekt (bei Bedarf) eindeutig identifizierbar ist, muss das Reply-Objekt einfach den Identifizierer des Request-Objektes als Attribut zurückgeben. Aufgrund dieses Attributes kann nun der RemoteRequestService (s.u.) die Zuordnung vornehmen. Diese Verwendung des Identifizierers ist eine Anwendung des *Asynchronous Completion Token (ACT)* Entwurfsmusters, wie es in [HSP97] beschrieben wird.

7.1.4 RemoteRequestService

Da die Kommunikationsprimitiven “Request” und “Reply” im VROOM System selbst Objekte sind, kann der ObjectTransportService als Grundlage für den RemoteRequestService dienen, um die fernen Methodenaufrufe zu realisieren.

Bei fernen Methodenaufrufen gelangen die Request-Objekte auf der Clientseite vom Stub-Objekt via das Klassenobjekt zum RemoteRequestService. Dieser lokalisiert mittels des RemoteLookupServices (siehe Kapitel 4.2.4) die Server-VROOM, die das gewünschte ferne Objekt enthält, und übermittelt danach das Request-Objekt unter Zuhilfenahme des ObjectTransportServices an diese VROOM (siehe Kapitel 4.2.2).

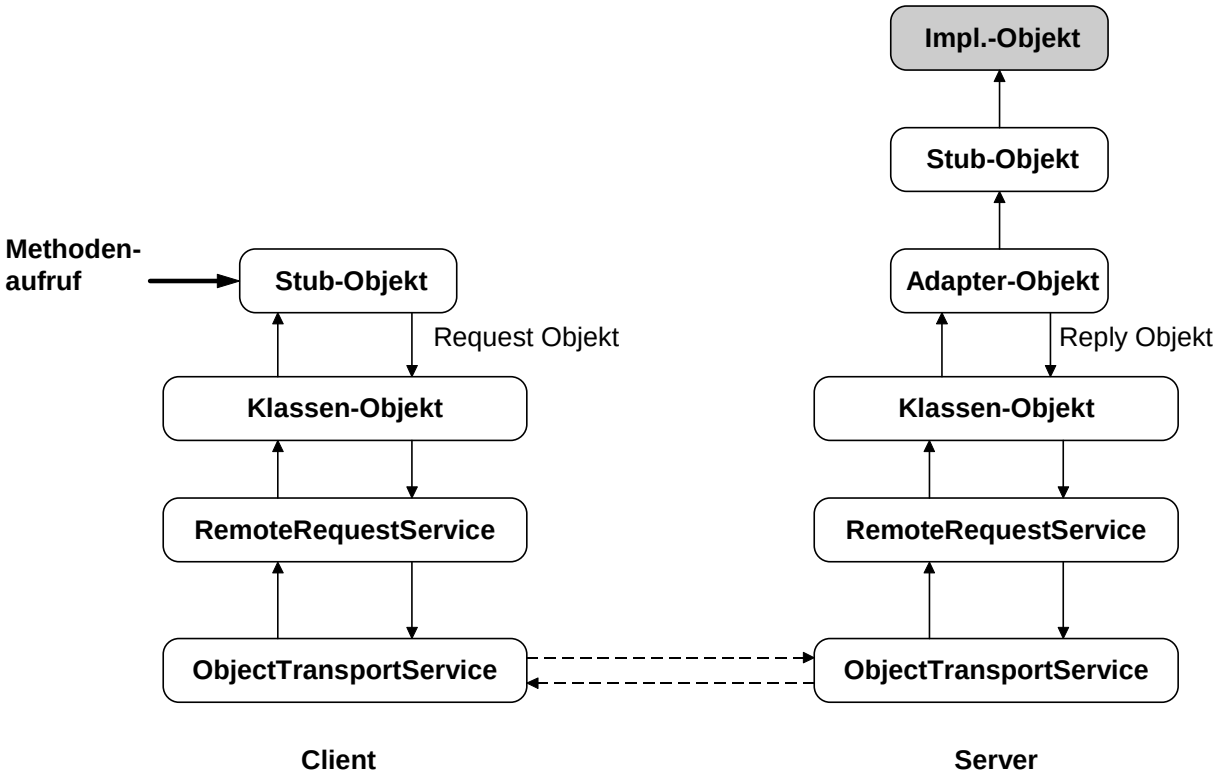


Abb. 7.4: Ablauf eines fernen Methodenaufrufes

Auf der Serverseite gelangt das Request-Objekt via ObjectTransportService, RemoteRequestService und Klassenobjekt zu einem sogenannten Adapter-Objekt. Pro Klassen-Objekt gibt es genau ein Adapter-Objekt. Dieses übernimmt das “Dispatching” des ankommenden Request-Objektes und ruft die gewünschte Methode im entsprechenden Stub-Objekt auf, welches zuvor mit dem LocalLookupService (siehe Kapitel 4.2.3) ermittelt wurde. Das Stub-Objekt seinerseits behandelt diesen Aufruf wie einen lokalen Methodenaufruf (siehe Kap. 7.1.1) und ruft die entsprechende Methode im Implementierungsobjekt auf.

Um die Rückgabewerte zurückgeben zu können, erzeugt dieser Adapter ein Reply-Objekt, das anschliessend an das aufrufende Stub-Objekt via den gleichen Weg zurückgegeben wird.

```
public class SimpleCalleeAdapter
    extends Adapter
{
    public Reply execute(AnyObject object, Request request)
    {
            SimpleCallee target = (SimpleCallee) object;
            // target ist ein Stub-Objekt und nicht direkt das Impl.objekt!

            String opName = request.getOperationName();

            // Reply-Objekt erzeugen
            AnyClass replyClass = (AnyClass) Sap.getMetaClass()
                    .getInstance("Vroom.Engine.Reply");
            Reply reply = (Reply) replyClass.newInstance();
            reply.setAppropriateRequestIdentifier(request.getIdentifier());
```

```
            …
            else if ( opName.equals("DoubleAddDoubleDouble") )
            {
                        double arg0 = request.getDoubleArgument(0);
                        double arg1 = request.getDoubleArgument(1);

                        // Methodenaufruf
                        double retValue = target.add(arg0, arg1);

                        reply.addDoubleArgument(retValue);
            }
            …

            return reply;
   }
}
```

Abb. 7.5: Request Dispatching durch das Adapter-Objekt am Beispiel der Klasse ‘SimpleCallee’

Auf der Clientseite wird der ferne Methodenaufruf - analog zu seinem lokalen Pendant - solange blockiert, bis die Methode in der Server-VROOM ausgeführt wurde und die Rückgabewerte vorliegen, d.h. sowohl lokale als auch ferne Aufrufe sind synchron. Die Kommunikationsinfrastruktur darf aber nicht blockieren, während ein oder mehrere ferne Methodenaufrufe hängig sind. Aus diesem Grund findet im RemoteRequestService eine Umsetzung der synchronen Aufrufe in asynchrone Kommunikation mittels Request- und Reply-Objekten statt. Dabei handelt es sich um eine Anwendung des *Half-Sync/Half-Async* Entwurfsmuster [SC96]. Ein Aufruf (praktisch können es - verursacht durch parallellaufende Threads - auch mehrere Aufrufe sein) wird in einer Warteschlange solange blockiert, bis dessen Rückgabewerte verfügbar sind. Erst dann wird die Blockierung aufgehoben und der synchrone Aufruf kann fortgesetzt werden. Durch diese Entkopplung ist das System in der Lage, auch während Reply-Objekte ausstehen, Request-Objekte von anderen VROOMs entgegenzunehmen, die Methoden auf Objekten in dieser VROOM ausführen möchten.

Es könnte der Fall eintreten, dass während eines lokalen oder fernen Aufrufes das gewünschte Implementierungsobjekt gerade verschoben wird. Dies ist aber nicht weiter schlimm, da prinzipiell jeder Methodenaufruf via Stub-Objekt geht. Ist das Implementierungsobjekt plötzlich nicht mehr vorhanden, erkennt das Stub-Objekt, dass es sich nun um einen fernen Methodenaufruf handeln muss und erzeugt selbst ein neues Request-Objekt, etc.

Auch beim sehr unwahrscheinlichen mehrmaligem Eintreten dieses Falles wird ein Request sein Zielobjekt früher oder später einholen, da anzunehmen ist, dass die erzeugten Datenbuffer für Request-Objekte viel kleiner sind als diejenigen, die für die komplette Migration eines VROOM Objektes benötigt werden. Diese Tatsache bewirkt, dass die Request-Datenbuffer schneller übertragen werden und damit früher oder später das gewünschte Objekt einholen.

In den folgenden Kapiteln wird auf die möglichen Fehlersituationen eingegangen, die bei fernen Methodenaufrufen enstehen können.

7.2 Fehlersituationen bei fernen Methodenaufrufen

Ferne Methodenaufrufe sind viel fehleranfälliger als lokale Methodenaufrufe, da sie auf ein Netzwerk und auf andere Prozesse zugreifen. Dies führt dazu, dass für ferne Methodenaufrufe spezielle Fehlerbehandlungsmöglichkeiten existieren müssen. Grundsätzlich versucht das System, Aufruffehler selbst zu beheben, ohne dass die Anwendung den aufgetretenen Fehler bemerkt. Erst wenn dies nicht mehr möglich ist (z.B. bei einer Unterbrechung der Kommunikationsleitung), wird eine Fehlermeldung (Exception) an die Anwendung weitergegeben. In einem verteilten System können folgende Fehlersituationen eintreten:

- Verlust einer Nachricht durch den Sender, den Empfänger oder den Kommunikationskanal
- Verfälschung von Nachrichten (wird in der Regel jedoch durch die tieferliegenden Protokollschichten behoben)
- Netzwerkpartitionierung: Unterbrechung der Kommunikationskanäle, so dass einige angeschlossene Rechner nicht mehr erreichbar sind
- Prozess(or)fehler des Senders oder des Empfängers

All diese Fehlersituationen müssen behandelt oder - falls dies nicht möglich ist - dem Benutzer mitgeteilt werden. Leider ist es nicht immer möglich, totale Fehlertransparenz zu erreichen (z.B. bei Absturz des Serverprozesses, unterbrochener Kommunikationsleitung), obwohl dies so gut als möglich angestrebt wird. In solchen Fehlerfällen kommt man nicht umhin, die Anwendung davon in Kenntnis zu setzen.

7.3 Fehlerbehandlungsmöglichkeiten bei fernen Methodenaufrufen

In diesem Kapitel werden kurz die Möglichkeiten aufgezeigt, die es gibt, um Fehlersituationen bei Methodenaufrufen von entfernten Objekten zu beheben.

7.3.1 Sendewiederholung des Requests

Wird nach einer gewissen Zeit (Timeout) nach dem Abschicken des Request-Objektes das Reply-Objekt nicht erhalten, schickt der Aufrufer das Request-Objekt erneut zum Empfänger.

7.3.2 Erkennen von Request-Duplikaten beim Empfänger (Duplikatefilterung)

Wird Sendewiederholung von Request-Objekten eingesetzt und die Kommunikation misslang, weil das entsprechende Reply-Objekt nicht bis zum Sender zurückgelangte, muss dieses Reply-Objekt nochmals erzeugt und versandt werden. Dazu müsste die Methode im Serverobjekt nochmals ausgeführt werden, um das Reply-Objekt wieder zu erzeugen. Dies ist aber nur dann sinnvoll, wenn die Methode idempotent ist, d.h. auch bei mehrmaligem Ausführen das gleiche Resultat (inkl. Seiteneffekte!) liefert. Da dies im allgemeinen nicht der Fall ist, müssen bereits erzeugte Reply-Objekte zwischengespeichert werden. Wird nun ein Request erneut erhalten und als Duplikat erkannt, wird das zugehörige Reply-Objekt ohne Neuausführen der Methode von neuem abgeschickt.

7.4 Aufruf-Fehlersemantiken

Im folgenden Abschnitt werden die Fehlersituationen, die beim Aufrufen einer fernen Methode eintreten können, mittels Aufruf-Fehlersemantiken kategorisiert. Das Entstehen der einzelnen Situationen und deren Behandlung wird erklärt. Für mehr Details wird auf die Literatur [CDK94, Wal96] verwiesen. Anschliessend wird in Kapitel 7.4.4 erklärt, welche Aufruf-Fehlersemantik für ferne Methodenaufrufe im VROOM System verwendet wurde.

7.4.1 Maybe Semantik

Wird ein Reply-Objekt vom Client bis zum Eintreten eines Timeouts nicht erhalten und wird das Request-Objekt nicht erneut abgeschickt, kann der Client keine Annahme machen, ob die entfernte Methode tatsächlich ausgeführt wurde (maybe). Falls das Request-Objekt nie beim Server ankam oder der Serverprozess abgestürzt ist, wurde die entfernte Methode nicht ausgeführt. Andererseits kann das Reply-Objekt verlorengegangen sein, d.h. die entfernte Methode wurde ausgeführt.

7.4.2 At-least-once Semantik

Werden Verluste von Request-Objekten erkannt, werden verlorengegangene Request-Objekte wiederholt und werden die Wiederholungen im Server nicht erkannt und herausgefiltert, wird die entfernte Methode unter Umständen mehrmals ausgeführt. Der Client kann aufgrund des erhaltenen Reply-Objektes keine Aussage darüber machen, wie oft die entfernte Methode ausgeführt wurde. Er weiss nur, das die Methode sicher zumindest einmal ausgeführt wurde (at least once).

7.4.3 At-most-once Semantik

In diesem Szenario werden Request-Wiederholungen erkannt und herausgefiltert. Verlorengegangene Reply-Objekte werden nochmals übertragen, jedoch ohne die entfernte Methode nochmals auszuführen. Erhält der Client ein Reply-Objekt, wurde die Methode genau einmal ausgeführt. Erhält er kein Reply-Objekt, weiss er nur, dass die Methode einmal oder nie ausgeführt wurde (at most once).

7.4.4 Verwendete Aufruf-Fehlersemantik im VROOM System

Der RemoteRequestService implementiert At-most-once Semantik, da weder Maybe noch At-least-once Sematik eine vernünftige Lösung darstellen. Obwohl der RemoteRequestService fehlertolerant ist und soviele Kommunikationsfehler wie möglich versucht zu beheben, können nicht alle Fehler maskiert werden (z.B. bei einer Partitionierung des Netzwerkes). Eine Benachrichtigung der daraufaufsetzenden Anwendung ist in diesen Fällen unumgänglich. Aus der Behandlung der Fehlersituationen in den obigen Kapiteln ist ersichtlich, dass es **insbesondere nicht möglich ist, für ferne Methodenaufrufe Exactly-once Semantik zu garantieren**, da der Client dazu immer ein Reply-Objekt brauchen würde, das aber potentiell verloren gehen kann.

8 Resultate und Diskussion

In der vorliegenden Arbeit ging es darum, einen allgemeinen Transport von Objekten zwischen Prozessen zu implementieren. Darauf aufbauend wurde ein Mechanismus realisiert, der es ermöglicht, Methoden in entfernten Objekten aufzurufen. Beim Entwurf und bei der Implementierung wurde bewusst auf Offenheit und Erweiterbarkeit geachtet. Performanz war dabei ganz klar ein Gesichtspunkt, der nicht im Zentrum stand. Im vorliegenden System ist es einfach, neue Übertragungsprotokolle zu implementieren und hinzuzufügen. Ausserdem stützt sich das Kommunikationssubsystem nicht auf eine spezifische Technologie der Datenübermittlung zwischen Prozessen ab. Es wurde bewusst darauf geachtet, genügend Spielraum offenzulassen, damit alternative Implementierungen möglich sind, die auf anderen Interprozesskommunikationstechnologien basieren wie z.B. Sockets, Java Remote Method Invocation, DCE Remote Procedure Call, etc.

Die hier gewählte Implementierung für das Kommunikationssubsystem basiert auf dem Common Object Request Broker Architecture (CORBA) Standard (siehe Kapitel 2.4.2). Die Realisierung mittels einer CORBA Implementierung ist aber nicht zwingend, da die Möglichkeiten des Standards nur minimal ausgenutzt werden. Im wesentlichen werden nur der Object Request Broker (ORB) und der Naming Service benutzt, um Strings zu übermitteln bzw. um entfernte Objekte zu lokalisieren. Die Möglichkeiten, die das CORBA Objektmodell bietet, werden nicht ausgenutzt. Bei der Wahl der konkreten CORBA Implementierung haben wir uns für das Produkt OrbixWeb der Firma Iona entschieden (siehe Kapitel 2.4.2.1). Die Daemon Architektur von OrbixWeb hat den Vorteil, dass sie in der Lage ist, das Auf- oder Neustarten von Serverprozessen selbst zu erledigen, falls diese noch nicht am Laufen oder zuvor abgestürzt sind. Diese Eigenschaft, die sich als sehr bequem erwiesen hat, ist leider im CORBA Standard (noch) nicht enthalten.

Relativ schnell bei der Realisierung hat sich gezeigt, dass synchrone Kommunikation für verteilte objektorientierte Systeme eine unbefriedigende Lösung darstellt. Da eine VROOM sowohl Client als auch Server ist, kann nicht hingenommen werden, dass z.B. für die Dauer von fernen Methodenaufrufen ganze Systeme blockiert werden. Eine Lösung, die auf asynchroner Kommunikation beruht, skaliert viel besser mit der Anzahl der verbundenen Systeme. Daraus folgt die Einsicht, dass Infrastrukturen für global verteilte objektorientierte Systeme nur auf asynchroner Kommunikation aufbauen können, was in [Pan96] bestätigt wurde.

Die anfängliche Forderung nach totaler Transparenz der Verteilung der Objekte ist der Erkenntnis gewichen, dass Transparenz in einem verteilten System zwar bequem nicht aber unbedingt immer wünschenswert ist. Dies hat seine Gründe: Zum einen gelten für lokale bzw. ferne Methodenaufrufe - bedingt durch mögliche Kommunikationsfehler - unterschiedliche Fehlersemantiken. Für einen fernen Methodenaufruf kann nur At-most-once Semantik und

nicht Exactly-once Semantik wie für lokale Aufrufe garantiert werden. Zum anderen dauern ferne Methodenaufrufe aufgrund der nötigen Interprozesskommunikation um Grössenordnungen länger und sind bedingt durch mögliche Übertragungsfehler, Prozessabstürze, etc. um einiges fehleranfälliger als ihre lokalen Pendants. Obwohl das System fehlertolerant ist und soviele Kommunikationsfehler wie möglich versucht zu beheben, können nicht alle Fehler maskiert werden. Eine Benachrichtigung der daraufaufsetzenden Anwendung ist in diesen Fällen unumgänglich. Eine klare Trennung in lokale und ferne Methodenaufrufe kann zudem auch Vorteile bringen. Die Anwendungen können profitieren von garantiert "schnellen" und sicheren lokalen Aufrufen ohne aufwendige Behandlungen von Ausnahmesituationen, die durch potentielle Übertragungsfehler, etc. entstehen können. Durch die Trennung wird das Bewusstsein für "langsame" und fehleranfällige ferne Methodenaufrufe gefördert. Wirklich befriedigende verteilte Systeme können vermutlich nur realisiert werden, wenn die Verteilung explizit gegen aussen sichtbar gemacht wird.

Das Prinzip, das dem VROOM System zugrunde liegt, dass jedes Objekt migrierbar sein soll, ist zwar sehr flexibel aber leider auch teuer. Der Aufwand, der für die Lokalisierung eines jeden entfernten Objektes betrieben werden muss, ist in der gegenwärtigen Implementierung trotz des Caches nicht zu vernachlässigen. Ein weiterer Punkt ist, dass man nicht selten die Aufenthaltsorte der Objekte (z.B. für "schwergewichtige" Objekte) ohnehin beschränken kann oder möchte auf die eigene oder einige wenige VROOMs. Nach all diesen Erfahrungen und Überlegungen sind wir zum Schluss gekommen, dass es effizienter wäre, statt den allgemeinen Lokalisierungsmechanismus (RemoteLookupService) für alle entfernten Objekte anzuwenden, dem Klassen-Objekt die Verantwortung über die Lokalisierung zu übertragen in der Erwartung, dass die Objekte schneller lokalisiert werden können. Ein Klassen-Objekt könnte dazu weiterhin den allgemeinen, langsamen Mechanismus oder aber eine alternative Lokalisierungsstrategie verwenden. In der momentanen Realisierung ist die Objekt-Lokalisierung viel zu tief im VROOM System verankert. Durch das Anwenden des *Strategy* Entwurfsmusters [GHJV95] könnte die ganze Lokalisierung getrennt vom Klassen-Objekt gekapselt werden. Damit wäre es auch möglich, dass ein Klassen-Objekt seine Lokalisierungsstrategie zur Laufzeit austauscht.

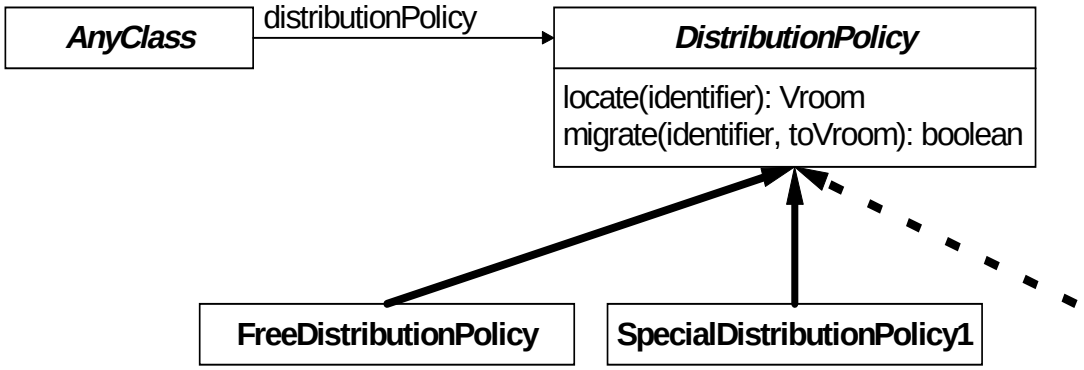


Abb. 8.1: ein Klassen-Objekt lokalisiert mittels einer DistributionPolicy seine Objekte

Das Klassen-Objekt kann neben der Lokalisierung auch zulassen oder verhindern, dass ein Objekt in den Adressraum einer anderen VROOM migriert wird (siehe "migrate(identifier, toVroom)" in Abb. 8.1). Es ist aber klar, dass die DistributionPolicies der Klassen-Objekte, die die gleiche Klasse in verschiedenen VROOMs vertreten, sich untereinander koordinieren müssen. Wie gross der dazu nötige Aufwand ist, muss weiter untersucht werden.

Bis jetzt wurde die ganze Zeit angenommen, dass bei fernen Methodenaufrufen die Klassen-Objekte des gerufenen Objektes, die während des Aufrufes eine zentrale Rolle spielen, sowohl in der Sender- als auch in der Empfänger-VROOM enthalten sind. Diese Annahme wird zwar häufig zutreffen, muss aber nicht immer der Fall sein. Falls das entsprechende Klassen-Objekt in der Empfänger-VROOM nicht existiert, muss vor dem Aufruf eine Synchronisation der beiden VROOMs geschehen.

Die vorliegende Diplomarbeit lässt aber noch viele weitere Punkte offen wie z.B.: Replikation von Objekten, Verteilte Garbage Collection, Objektpersistenz, Verteilte Transaktionen und Sicherheit.

Zum Abschluss noch ein paar Worte zu der verwendeten Programmiersprache Java. Der Einsatz dieser Programmiersprache hat sich in der Praxis bewährt. Sie verfügt über ein paar Eigenschaften, die das Leben extrem vereinfachen wie z.B. Garbage Collection und einfache Multithreading-Programmierung. Dazu schätze ich die Vorzüge, die eine Unterscheidung zwischen Schnittstellen- und Implementierungsvererbung bringt, immer mehr.

Literaturverzeichnis

[BGR96] W. Bischofberger, M. Guttman, D. Riehle. Global Business Objects: Requirements and Solutions. In Computer Science research at Ubilab: strategy and projects. Proceedings of the Ubilab conference ’96, Zurich. Edited by K.-U. Mätzel, H.-P. Frei. Konstanz, Universitätsverlag Konstanz, 1996

[CDK94] G. Coulouris, J. Dollimore, T. Kindberg. Distributed Systems - Concepts and Design, Addison-Wesley, Wokingham, 1995 (2nd Edition)

[DNX94] J. Dollimore, C. Nascimento, W. Xu. Fine Grained Object Migration. from: M.T. Öszu, U. Dayal, P. Valduriez. Distributed Object Management. Morgan Kaufmann, San Mateo, 1994

[Fla96] D. Flanagan. Java in a Nutshell: A Desktop Quick Reference for Java Programmers, O’Reilly & Associates, Sebastopol, CA, 1996

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Design. Addison-Wesley, Reading, MA, 1995

[HSP97] T.H. Harrison, D.C. Schmidt, and I. Pyarali. “Asynchronous Completion Token - An Object Behavioral Pattern for Efficient Asynchronous Event Handling”. In Pattern Languages of Program Design 3. Edited by R.C. Martin, D. Riehle, and F. Buschmann. Addison-Wesley, Reading, MA, 1997

[JLHB88] E.Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. ACM Transactions on Computer Systems 6(1). February 1988.

[Jul88] E. Jul. Object Mobility in a Distributed Object-Oriented System. PhD thesis, Departement of Computer Science, University of Washington, Seattle, Washington, 1988

[Lop96] C.V. Lopes. Adaptive parameter passing. Object Technologies for Advanced Software, Second JSSST International Symposium ISOTAS'96, Proceedings. Springer Verlag, Berlin 1996

[Lux95] W. Lux. Adaptable object migration: concept and implementation. Operating Systems Review 29(2). April 1995

[OHE96] R. Orfali, D. Harkey, J. Edwards. The Essential Distributed Object Survival Guide. John Wiley & Sons, New York, 1996

[OMG95] Object Management Group. CORBA: Common Object Request Request Broker Architecture and Specification, Revision 2.0. July 1995

[OMG96] Object Management Group. CORBAservices: Common Object Services Specification (COSS), Revised Edition. July 1996

[Pan96] M. Panttaja. Client/Server Redefined. Database Programming & Design 9(10). Oct. 1996

[RSB+97] D. Riehle, W. Siberski, D. Bäumer, D. Megert and H. Züllighoven. “Serializer”. In Pattern Languages of Program Design 3. Edited by R.C. Martin, D. Riehle, and F. Buschmann. Addison-Wesley, Reading, MA, 1997

[SC96] D.C. Schmidt and C.D. Cranor. “Half-Sync/Half-Async - An Architectural Pattern for Efficient and Well-structured Concurrent I/O”. In Pattern Languages of Program Design 2. Edited by J.M. Vlissides, J.O. Coplien, and N.L. Kerth. Addison-Wesley, Reading, MA, 1996

[Sie96] J. Siegel. CORBA Fundamentals and Programming. John Wiley & Sons, New York, 1996

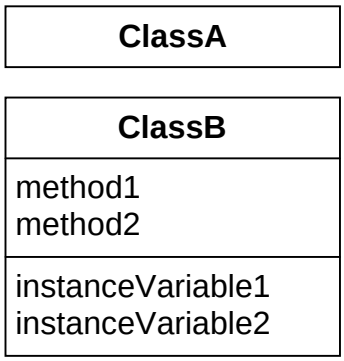
[Wal96] T. Walter. Skript zu der Vorlesung “Verteilte Systeme” (SS96). Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, Zürich, 1996

[Wol96] A. Wollrath, R. Riggs, and J. Waldo. A Distributed Object Model for the Java System. COOTS-2, Conference Proceedings. Usenix Association, 1996

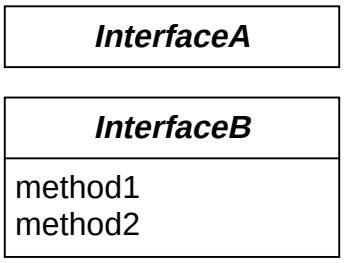
Anhang A:
Erweiterte Taligent Notation

Klassendiagramme

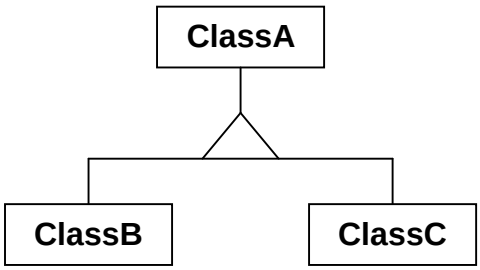
Darstellung von Klassen



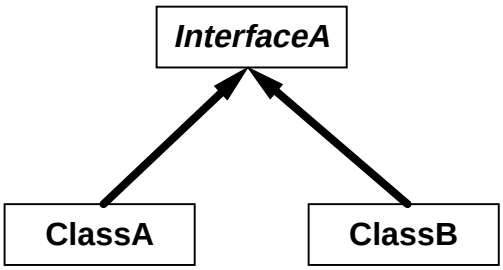
Darstellung von Schnittstellen



Vererbung (in Java: extends)

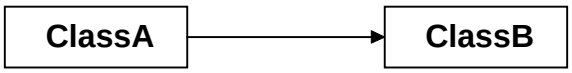


Implementierungsbeziehung
(in Java: implements)



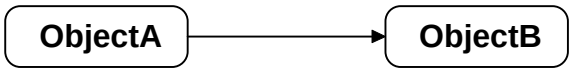
Referenzierung

Aggregation



Objekt-Interaktionsdiagramme

Methodenaufruf



Ubilab Technical Reports

94.6.1 Maffeis S, Bischofberger WR, Mätzel K-U: GTS: *A Generic Multicast Transport Service*

94.9.1 Bischofberger WR, Kofler T, Mätzel K-U, Schäffer B: *Computer Supported Co-operative Software Engineering with Beyond-Sniff*

94.9.2 Bäumer D, Bischofberger WR, Lichter H, Schneider-Hufschmitdt M, Sedlmeier-Scholz V, Züllighoven H: *Prototyping von Benutzungsoberflächen*

94.10.1 Steiger P, Ansel Suter B: *Minnelli Schlussbericht*

94.10.2 Levy N, Hornstein T: *Text-to-Speech Technology: A Survey of German Speech Synthesis Systems*

95.6.1 Riehle D: *Muster am Beispiel der Werkzeug und Material Metapher*

95.7.1 Riehle D, Schäffer B, Schnyder M: *Design and Implementation of a Smalltalk Framework based on the Tools and Materials Metaphor*

97.1.1 Riehle D: *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose.*

Paper copies of Ubilab technical reports can be ordered from the mailing address on the first page or by e-mail from its author using the scheme firstname.lastname@ubs.com. Most reports can also be obtained as PostScript files via WWW (http://www.ubs.com/ubilab).

Die vorliegende Arbeit beschreibt in einem ersten Teil den Entwurf und die Implementierung eines Kommunikationssubsystems für den allgemeinen Transport von Objekten zwischen Prozessen. Diese Arbeit wurde vollständig eingebettet in das Geo-System, dessen Ziel es ist, mittels einer reflexiven Architektur eine Infrastruktur für global verteilte Objekte zu schaffen. Beim Entwurf des Frameworks für das Kommunikationssubsystem wurde bewusst auf Offenheit und Erweiterbarkeit geachtet. Für die Realisierung der "Low-Level"-Kommunikation wurde OrbixWeb, eine Implementierung der Common Object Request Broker Architecture (CORBA) der Firma Iona, verwendet. Dank der Offenheit des Subsystems ist es jedoch leicht, alternative Implementierungen zu verwenden, die z.B. auf Sockets oder Java Remote Method Invocation basieren. Im zweiten Teil der Arbeit wird gezeigt, wie Methodenaufrufe auf Objekten in entfernten Prozessen als Anwendung dieses allgemeinen Objekttransportmechanismus realisiert wurden.