Mobilität in persistenten Objektsystemen

Dissertation

Bernd Mathiske

Fachbereich Informatik

Universität Hamburg

31. Mai 1996

Zusammenfassung

Verteilte datenintensive Anwendungen, in denen komplexe, semantisch reichhaltige Strukturen migrieren, werden gegenwärtig durch aufwendige Kombinationen von Programmiersprachen und Systemdiensten erbracht, die weitgehend unabhängig voneinander entwickelt worden sind. Während persistente Systeme keine ausreichende Mobilität von Objekten bieten, unterstützen verteilte Programmiersprachen Persistenz nur unbefriedigend. Außerdem besteht ein Mangel an Techniken verteilter Programmierung, die bezüglich Bandbreite, Latenz, Zuverlässigkeit und Heterogenität von Netzwerken skalieren.

Ausgangspunkt der vorliegenden Dissertation ist die Überzeugung, daß die Qualität und der Entwicklungsaufwand verteilter datenintensiver Anwendungen wesentlich dadurch bestimmt sind, inwiefern eine direkte, unkomplizierte Abbildung von Anwendungsstrukturen in korrespondierende Programmiersprachenkonstrukte gelingt.

Das Ziel dieser Arbeit ist daher die Bereitstellung eines *einheitlichen* programmiersprachlichen und architektonischen Rahmens in offenen Systemumgebungen, der orthogonale Persistenz mit einer substantiell verbesserten *Mobilität* von Objekten beliebigen Typs verbindet. Die grundlegende Vorgehensweise besteht dabei in der konsequenten Weiterentwicklung eines *persistenten Objektsystems* durch folgende Beiträge:

- ▷ Die Gewährleistung einer hochgradigen *Plattformunabhängigkeit* des Gesamtsystems durch eine gezielt auf Portabilität ausgerichtete Laufzeitsystemarchitektur und einen modularen Kommunikationsmechanismus für den plattformunabhängigen Datenaustausch zwischen persistenten Objektspeichern.
- ▷ Polymorphe entfernte Funktionen (RPCs) höherer Ordnung mit typsicheren, mobilen und persistenten Client/Server-Bindungen.
- ▷ Migrierende persistente Threads, ein prädestiniertes Konstrukt für mobile Agenten und Workflows.
- ▷ Flexible Bindungstechniken, welche die *Autonomie* migrierender Objekte wahren und zu drastischen Reduzierungen des jeweils zu übertragenden Datenvolumens führen. Dadurch wird die Mobilität komplexer Objekte wie z.B. Funktionsabschlüsse und Threads in Weitverkehrsnetzen in der Praxis überhaupt erst ermöglicht.

Die in der Arbeit erreichten Architekturen und Methoden führen zu einer signifikanten Erhöhung der Flexibilität und Qualität bei gleichzeitiger Minderung des Aufwandes zur Erstellung datenintensiver verteilter Anwendungen. Dies wird anhand ausgewählter Problemstellungen, für die innovative Lösungen entwickelt werden, exemplarisch aufgezeigt.

Zur Validierung der Thesen der Arbeit wird ein Systemprototyp vorgestellt, der die oben genannten Beiträge in synergetischer Kombination implementiert.

Abstract

Distributed data-intensive applications featuring migrations of complex, semantically rich structures are nowadays accomplished by costly combinations of independently developed programming languages and system services. Whereas persistent systems do not provide sufficient mobility of objects, distributed programming languages have deficiencies of persistence support. Furthermore there is a lack of distributed programming techniques that scale well with respect to network bandwidth, latency, reliability and heterogenity.

This thesis is motivated by the conviction that the quality and the development expenditure of distributed data-intensive applications is essentially determined by succeeding in a direct and uncomplicated way of mapping application structures to programming language constructs.

Therefore the objective of this work is to provide a *uniform* linguistic and architectural framework in open system environments that combines orthogonal persistence with substantially improved *mobility* support for objects of arbitrary type. The basic approach consists in a consequent further development of an existing *persistent object system* by the following contributions:

- ▷ The ensurement of a high degree of *platform independence* for the overall system by a runtime system architecture which is tailored to portability and by a modular communication mechanism for platform independent data exchange between persistent object stores.
- ▷ Polymorphic higher order remote functions (RPCs) with type-safe, mobile and persistent client/server-bindings.
- ▷ Migrating persistent threads, a predestinated construct for mobile agents and workflows.
- ▷ Flexible binding techniques which preserve the *autonomy* of migrating objects and also lead to drastic reductions of data volumes to be transmitted. In practice this is crucial to ensure the mobility of complex objects like function closures and threads in wide area networks.

The architectures and methods achieved by this work lead to a significant improvement of the flexibility and quality of distributed data-intensive applications and they also reduce the required development effort. This is demonstrated by giving examples of innovative solutions for selected problems.

For validation of the contributions of this thesis a system prototype which synergetically implements them is presented.

Danksagungen

Ich danke Herrn Prof. Dr. Joachim W. Schmidt für die Organisation der ausgezeichneten Arbeitsbedingungen und für seine konstruktive Kritik. Insbesondere die überaus stimulierende Atmosphäre innerhalb der Arbeitsgruppe DBIS sowie die fruchtbaren und persönlichen Kontakte heben den Charakter dieser Arbeit bestimmt.

Herrn Prof. Dr. Friedrich H. Vogt danke ich für seine Anregungen im Hinblick auf CORBA.

Mein besonderer Dank gilt Herrn Dr. Florian Matthes für die vielfältige Förderung meiner Arbeit.

Ganz besonders herzlich möchte ich mich bei meinen Kollegen Andreas Rudloff, Thomas Sidow, Gerald Schröder, Andreas Gawecki, Sven Müßig, Rainer Müller, Michael Merz und Kay Müller-Jones für ihre tatkräftige Unterstützung und die harmonische Zusammenarbeit bedanken.

Die Implementierung der in der Arbeit vorgestellten Systemtechniken wurde durch den engagierten Einsatz von Studenten und Diplomanden im Arbeitsbereich DBIS unterstützt. Dafür danke ich Martin Göllnitz (mobilitätsorientierte RPCs uvm.), Nastaran Vaziri Pour (dynamisches Linken, automatische Replikation), Markus Breilmann (Mac-Portierungen, Demo-Programme), Andreas Piellusch (Thread-Synchronisation, Linux- und AIX-Portierung), Marcel Kornacker (persistente Sicherungspunkte, Pseudo-Persistenz), Nico Johannisson (erste Version des Tycoon-RPC, Amiga-Portierung), Hubertus Köhler (Starview-Consulting, Demo-Programme), André Willomat (Socket-Schnittstelle, OS/2-Portierung), Kai Shen (Windows-Portierung), Detlev Niemann-Bohde (NeXTSTEP-Portierung) und Oliver Schmelzle (WWW-Anbindung persistenter Threads).

Bei dieser Gelegenheit möchte ich mich bei Miguel Mira da Silva, Detlef Kreuz, Malcolm Atkinson, Chris Barter, Alan Dearle und David Maier für ihr Interesse am Thema meiner Arbeit und ihre konstruktive Kritik an meinen Ideen bedanken.

Helen Brodie danke ich für ihre Unterstützung in administrativen Angelegenheiten.

Für Christine und Annabel Fee

Inhaltsverzeichnis

1 Einleitung und Motivation **1**

1.1 Realisierungskonzepte für Verteilung und Persistenz 7

1.2 Persistente Objektsysteme als Basis der Arbeit 11

1.3 Ziele und Lösungsansätze 14

1.4 Gliederung der Arbeit 20

2 Mobilität in existierenden Programmiersystemen **21**

2.1 Kommunikationsprogrammierung in TCP/IP-Netzwerken 22

2.1.1 BSD-Sockets 23

2.1.2 Gängige RPC-Mechanismen 24

2.1.3 Externe Datenrepräsentationen und Marshalling 25

2.2 Verteilte Objektmanagementsysteme 27

2.3 Verteilte Programmiersprachen 29

2.3.1 Emerald 29

2.3.2 Obliq 31

2.3.3 Java 32

2.4 Sprachen für mobile Agenten 34

2.4.1 Telescript 35

2.4.2 FACILE 36

2.5 Persistente Objektsysteme 37

2.6 Zusammenfassung 39

3 Das persistente Objektsystem Tycoon **41**

3.1 Die Schichtenarchitektur des Tycoon-Systems 42

3.2 Die Programmiersprache Tycoon Language 44

3.2.1 Wert- und Typausdrücke 44

3.2.2 Funktionen höherer Ordnung 47

3.2.3 Polymorphie 48
3.2.4 Dynamische Typen 49
3.2.5 Modulare Spracherweiterungen 50
3.2.6 Flache Transaktionen 52
3.3 Die abstrakte Objektspeicherschnittstelle TSP 53
3.3.1 Basistypen und Tagging 55
3.3.2 Programmierschnittstellen 57
3.4 Uniforme Laufzeitrepräsentationen in persistenten Objektspeichern 58
3.5 Externe Bindungen 60
3.5.1 Funktionsaufrufe von TL nach *C* 61
3.5.2 Funktionsaufrufe von *C* nach TL 62

4 Plattformunabhängige Realisierung des Tycoon-Systems 63
4.1 Die Architektur des portablen Tycoon-Laufzeitsystems 64
4.2 Plattformunabhängige Socket-Kommunikation 66
4.3 Die plattformunabhängige lineare Datenrepräsentation TXR 69
4.3.1 Programmierschnittstellen 69
4.3.2 Die Grundalgorithmen des polymorphen Objektaustausches 71
4.3.3 Parametrische Modifikationen des Objektaustausches 73

5 Mobilitätsorientierte entfernte Funktionsaufrufe 75
5.1 Entfernte Funktionsaufrufe im Tycoon-System 76
5.1.1 Begriffe und Anforderungen 76
5.1.2 Strukturelle und algorithmische Auswahl entfernter Dienste 81
5.1.3 Programmierung mit polymorphen entfernten Funktionen höherer Ordnung 84
5.1.4 Persistenz und Mobilität kommunizierender Objekte 89
5.2 Die Kommunikationsbibliothek des Tycoon-Systems 91
5.2.1 Schnittstellen und Modulstruktur 91
5.2.2 Polymorphe Kommunikationsmechanismen mit einfacher Adressierung 95
5.2.3 Hierarchische Adressierung entfernter Dienste 103
5.2.4 Die Implementation der Netzdienste 110

6 Migrierende persistente Threads **115**

6.1 Thread-Programmierung im Tycoon-System 115

6.1.1 Threads als typisierte Objekte erster Klasse 116

6.1.2 Persistente Threads 119

6.1.3 Migrierende Threads 122

6.2 Implementationstechniken für migrierende persistente Threads 125

6.2.1 Die Repräsentation von Threads im Objektspeicher 125

6.2.2 Typisierte Thread-Migrationspforten 129

6.2.3 Transaktionale Thread-Migration 132

7 Skalierende Bindungstechniken für immobile Objekte **139**

7.1 Ursachen von Immobilität 140

7.2 Eignungskriterien für Bindungstechniken 141

7.3 Typsichere temporäre Bindungen an immobile Objekte 143

7.4 Dynamisches Linken ubiquitärer Objekte 146

7.5 Automatische Replikation 149

7.6 Pseudo-Mobilität und Pseudo-Persistenz flüchtiger Objekte 151

8 Erfahrungen und Ausblick **153**

8.1 Anwendungen 153

8.2 Resümee 155

8.3 Weiterführende Arbeiten 159

A Die polymorphe Unterstützung entfernter Funktionen **163**

A.1 Die Client-Schnittstelle 163

A.2 Die Server-Schnittstelle 165

B Die schmale Implementationsbasis der Kommunikationsbibliothek **166**

B.1 Die plattformunabhängige Socket-Schnittstelle 166

B.2 Typunsichere Systemdienste 168

C Die Standardfunktion zur Wiederauffindung entfernter Dienste **169**

D Grundoperationen migrierender persistenter Threads **170**

D.1 Grundoperationen persistenter Threads 170

D.2 Thread-Migrationspforten 174

E Operations-Logging für flüchtige Objekte **175**

E.1 Pseudo-Persistenz 175

E.2 Pseudo-Mobilität 177

F Performanz entfernter Funktionsaufrufe **178**

Abbildungsverzeichnis

1.1 Mobilitätsgrade in existierenden verteilten Programmiersystemen 3
1.2 Horizontale Verteilungs- und vertikale Persistenzabstraktion 7
1.3 Die grundlegende Struktur persistenter Objektsysteme [Matthes 93] 11
1.4 Der kombinatorische Raum von Datentypen, Persistenz und Mobilität 14

2.1 Die Rolle von Ports bei der Netzwerkkommunikation 22
2.2 Ein typisches Szenario der Socket-Kommunikation 23
2.3 Das Grundprinzip des RPC 24
2.4 Das interne Ablaufschema eines RPC 25
2.5 Generierungsschemata des ONC-RPC und des DCE-RPC 26
2.6 Transparente Erzeugung von Netzwerkreferenzen bei der Übertragung von Objekten 30
2.7 Ersetzen von Client/Server-Anfragen durch eine Agentenreise 34

3.1 Die Schichtenarchitektur des Tycoon-Systems 43
3.2 Das *Tagging*-Schema des TSP 55
3.3 Das interne Objekt-Layout in Tycoon-Objektspeichern 57
3.4 Die interne Repräsentation eines TL-Wertes 58
3.5 Der Objektgraph eines komplexen TL-Wertes 59
3.6 Die Grundstruktur des offenen persistenten Objektsystems Tycoon [Matthes et al. 95b] 60

4.1 Die auf Portabilität ausgerichtete Architektur des Tycoon-Laufzeitsystems . . 64
4.2 Funktionen der plattformunabhängigen *C*-Schnittstelle für Sockets 66
4.3 Die objektspeicherunabhängige Ansiedlung des Linearisierungs- und Delinearisierungsmoduls 70
4.4 Linearisierung der transitiven referentiellen Hülle eines komplexen Objektes . 71
4.5 Die TXR-Linearisierung im Detail 72

5.1 Grundbegriffe der RPC-Programmierung im Tycoon-System 77

5.2 Die drei Kriterien der Dienstauswahl 79
5.3 Korrespondierende Begriffe im Tycoon-System und im ODP-Trading 79
5.4 Aufruf einer entfernten Funktion höherer Ordnung 86
5.5 Entfernte dynamische Optimierung und Ausführung einer Datenbankanfrage . 87
5.6 Ein Funktionsabschluß als RPC-Ergebnis 88
5.7 Schnittstellen der Kommunikationsbibliothek des Tycoon-Systems 91
5.8 Modulstruktur der Tycoon-Bibliothek zur Kommunikationsprogrammierung . 92
5.9 Die Anordnung der Dienstprogramme im Netzwerk 93
5.10 Aufbau eines Kommunikationspaketes 95
5.11 Einfaches Senden eines Tycoon-Objektes 96
5.12 Senden eines Tycoon-Objektes mit Rückantwort 97
5.13 Anfrageverarbeitung im RPC-Server 99
5.14 Die Hierarchie der drei Adressierungsebenen 104
5.15 Auffinden eines Dienstes im LAN 105
5.16 Auffinden eines Dienstes in einem entfernten LAN 106
5.17 Funktionen des Mapping Daemons (MAD) 111
5.18 Funktionen des Tycoon LAN Daemons (TYLAND) 113

6.1 Bestandteile des Ausführungszustandes eines Threads 116
6.2 Modellierung des Reisekosten-Workflows im Client/Server-Stil 122
6.3 Modellierung des Reisekosten-Workflows durch einen migrierenden Thread . . 123
6.4 Die Repräsentationsstruktur eines Threads im persistenten Objektspeicher . . 125
6.5 Ausführungszustände persistenter Threads 126
6.6 Schematischer Ablauf des 2-Phasen-Commit-Protokolls 133
6.7 Durchführung der 2PC-Teilnahme durch einen speziellen Thread 135
6.8 Überbrückung der durch ein 2PC bedingten Kommunikationspause 138

7.1 Die umfangreiche transitive referentielle Hülle eines Threads, der graphische Ausgaben erzeugt 140
7.2 Übertragung der transitiven referentiellen Hülle eines Objektes 146
7.3 Dynamisches Linken ubiquitärer Objekte 147
7.4 Die drei Schritte der automatischen Replikation 149
7.5 Dynamisches Linken eines automatisch replizierten Objektes 149

8.1 WWW-Anbindung persistenter Threads 154
8.2 Eine Softwarearchitektur für interaktive mobile Internet-Agenten 155

8.3 Die erreichten Mobilitätsgrade im Vergleich zu anderen Systemen 156
8.4 Antizipation der Weiterentwicklung von Programmiersystemen 158

Kapitel 1

Einleitung und Motivation

Im Zuge der zunehmenden globalen Vernetzung von Unternehmen, öffentlichen Einrichtungen und Privathaushalten zeichnet sich ein deutlicher Trend zu Softwaresystemen ab, in denen Objekte[1] *migrieren*, d.h. ihren Ort wechseln. Dies gilt zum Beispiel für folgende innovativen Anwendungsszenarien:

Workflow-Management-Systeme, in denen sich Geschäftsvorgänge als eigenständige Einheiten durch Unternehmensnetze bewegen, wobei sie die Kooperation menschlicher Benutzern unterstützen [Leymann, Altenhuber 94; Mc Cready, Palermo 94; Bach et al. 95; Jablonski 95; WFMC 95].

Agentensysteme, in denen autonome Softwareagenten das Internet bereisen, um jeweils vor Ort im Auftrag ihres Benutzers Handlungen unterschiedlichster Art durchzuführen [Wayner 94; Reinhardt 94; White 94; Thomsen et al. 95].

Migratorische Anwendungen, die ihren Benutzer von Rechner zu Rechner begleiten, indem sie sich quasi selbst im Netzwerk verpflanzen [Zayas 87; Bharat, Cardelli 95].

Die Realisierung verteilter Aktivitäten, die über Stunden, Tage, Wochen oder sogar noch länger andauern, ist mit konventionellen Programmierumgebungen jedoch nur bedingt durchführbar. Zum einen ist es sehr kompliziert, entfernte Bindungen zwischen langlebigen Aktivitäten über längere Zeiträume hinweg zu unterstützen. Zum anderen erfordern insbesondere heutige Client-Server-Systeme die Verteilung der Ablaufbeschreibungen und Zustände von Aktivitäten. Dies bedeutet, daß aus Sicht der jeweiligen Anwendung in sich geschlossene Kontexte aufgrund softwaretechnischer Zwänge in Abhängigkeit der Verfügbarkeit und Zuverlässigkeit großer Anzahlen von Netzwerkkomponenten (Leitungen und Knoten, Hard- und Software) gebracht werden. Außerdem werden unnötig hohe Kommunikationskosten verursacht.

[1] Die Bezeichnung „Objekt“ wird in der vorliegenden Arbeit in einem recht allgemeinen Sinne verwendet. Sie impliziert weder, daß irgendeine Form der programmiersprachlichen Objektorientierung vorliege, noch schließt sie dies aus. Unter einem „Objekt“ wird im folgenden jegliche Art von Datum verstanden, das in einer Programmiersprache als eigenständige Einheit identifizierbar und manipulierbar ist. Dies entspricht dem weitgefaßten Objektbegriff persistenter Objektsysteme [Matthes 93]. Die sonst üblicherweise ebenfalls als „Objekte“ bezeichneten Speicherungseinheiten persistenter Objektspeicher werden im folgenden zur besseren Unterscheidung der Abstraktionsebenen „Speicherobjekte“ genannt.

In den oben genannten Anwendungsszenarien Klasse stellt die Information, an welcher Stelle im Netzwerk ein Objekt sich befindet, eine *anwendungsrelevante* Information dar. Bezeichnend ist ein hoher Bedarf an *Mobilität* (Migrationsfähigkeit) komplexer und semantisch reichhaltiger Strukturen. Hierbei handelt es sich insbesondere um rekursive Beziehungen, Ablaufbeschreibungen, gekapselte Zustände und Programmausführungszustände. Diesen Strukturen entsprechen in Programmiersprachen folgende, in Bezug auf Mobilität nach „steigendem Schwierigkeitsgrad" geordnete Konstrukte:

1. **Rekursive Datenstrukturen** modellieren rekursive Beziehungen.
2. **Programmcode** ist aus Sicht des Programmierers die direkteste Manifestation von Ablaufbeschreibungen.
3. **Funktionsabschlüsse** kapseln Zustände und ordnen sie dynamisch (ggf. parametrisiertem) Programmcode zu. Die in objektorientierten Systemen gegebene Zustandskapselung durch Objektinstanzen wird in diesem Zusammenhang als prinzipiell äquivalent angesehen.
4. **Threads** sind Objekte, die nebenläufige Programmausführungen innerhalb eines gemeinsamen Betriebssystemprozesses repräsentieren.[2]

Desweiteren ist die Mobilität von Objekten, die direkt oder indirekt Kommunikationsverbindungen identifizieren, von Bedeutung. Die unterschiedlichen Ausprägungen (Sockets, Kanäle, Client-Handles, Server-Handles, entfernte Objektreferenzen, Stubs, Proxys, etc.) solcher Objekte seien im folgenden unter der Bezeichnung „Kommunikationsidentifikator" zusammengefaßt.

Eine wichtige Grundannahme der vorliegenden Arbeit besteht darin, daß der Programmieraufwand für obige Anwendungen sowie deren Softwarequalität wesentlich dadurch bestimmt sind, inwiefern eine Abbildung von Anwendungsstrukturen in korrespondierende Programmiersprachenkonstrukte gelingt, bzw. ob sie überhaupt möglich ist.

Um in diesem Zusammenhang die Eignung heutiger Sprachen und Bibliotheken für verteilte Programmierung zu analysieren, werden diese in Bezug auf die Mobilität bestimmter Objektkategorien klassifiziert. Abbildung 1.1 zeigt eine Übersicht relevanter Systeme[3], die nach obigen Kriterien aufgeschlüsselt ist.

Es ist zu beobachten, daß genau diejenigen Systeme, welche über eine Garbage-Collection verfügen, eine überzeugende Unterstützung rekursiver Strukturen bieten. Die Mobilität von Code ist nur relativ selten anzutreffen, die von Funktionsabschlüssen noch seltener. Lediglich in Emerald und in Telescript sind auch Threads mobil. Während die Mobilität von Kommunikationsidentifikatoren in objektorientierten Systemen durchgängig unterstützt wird, sind hier bei RPC-Systemen deutliche Defizite festzustellen.

Die bisherigen Aussagen betreffen nur die *potentielle* Mobilität aufgrund einer Datentypklassifikation. In realen Anwendungen ist die *tatsächliche* Mobilität eines bestimmten Objektes von vielen zusätzlichen Faktoren abhängig:

[2] Eine eingehende Erläuterung des Thread-Begriffes erfolgt in Abschnitt 6.1.

[3] Auf die in Abbildung 1.1 genannten Systeme wird in Kapitel 2 näher eingegangen, wobei auch die zugrundeliegenden Informationsquellen angegeben werden.

Mobilität:	Rekursive Strukturen	Code	Funktions-abschlüsse	Threads	Kommunikations-identifikatoren
ONC/DCE-RPC	(+)	-	-	-	-
CORBA	(+)	-	-	-	++
Modula-3	++	-	-	-	++
Erlang	++	+	-	-	-
Java	++	++	-	-	-
SOS, DC++	+	++	oo	-	++
Napier88	+++	++	++	-	-
Phantom	++	++	+++	-	++
Obliq	++	++	+++ oo	-	+++
FACILE	+++	+++	+++	-	++
Telescript	+	+++	oo	++	-
Emerald	++	+++	oo	+	+++

+++	durchgängig unterstützt	(+)	nur bedingt und sehr unpraktisch
++	etwas unpraktisch/eingeschränkt	-	nicht unterstützt
+	simuliert oder ziemlich eingeschränkt		
oo	Funktionsabschlüsse indirekt in Objekten enthalten (bei objektorientierten Sprachen)		

Abbildung 1.1: Mobilitätsgrade in existierenden verteilten Programmiersystemen

- ▷ Ist der Gesamtumfang der zu transferierenden Daten so groß, daß die Übertragung zeitkritisch wird?
- ▷ Steht auf der Empfängerseite genügend Speicherplatz zur Verfügung?
- ▷ Wieviel späterer Netzverkehr wird indirekt verursacht?
- ▷ Bestehen Bindungen an ortsfeste Soft- oder Hardware?
- ▷ Wie werden andere Objekte durch eine Migration beeinflußt?
- ▷ Werden Referenzen auf andere Objekte angepaßt?
- ▷ Welche und wie viele Laufzeitfehler (insbesondere Typfehler) sind wahrscheinlich, bzw. ausgeschlossen?

Bei näherer Untersuchung[4] anhand solcher Fragestellungen stellt sich heraus, daß die heutigen Mittel der verteilten Programmierung für lokale Netzwerke weiter fortgeschritten sind als die für Weitverkehrsnetze. Der wesentliche Grund dafür ist, daß die verfügbaren Programmierschemata meist nicht ausreichend bezüglich Bandbreite, Latenz und Zuverlässigkeit von Netzwerken skalieren.

Von den vorhandenen Möglichkeiten verteilter Programmiersprachen würden nicht nur die eingangs genannten Anwendungsarten profitieren, sondern die Entwicklung von Informationssystemen im allgemeinen. *Informationssysteme* sind Programmsysteme, deren Aufgabe es ist, große Mengen langlebiger Daten sicher, flexibel und problemadäquat zu verwalten, zu manipulieren und darzustellen [Matthes 93]. Programmierumgebungen für Informationssysteme müssen einer Vielzahl anspruchsvoller Anforderungen gerecht werden. Besonders wichtig sind:

Persistenz: die *Eigenschaft*[5] *von Objekten*, daß ihre Repräsentation unabhängig von der Ausdehnung der für ihre Benutzung vorgesehenen Zeitspanne erhalten bleibt. Objekte, auf die diese Eigenschaft zutrifft, werden im folgenden als *persistent* bezeichnet.

In Informationssystemen treten besonders ausgedehnte „Lebensdauern“ (Zeiträume der Benutzbarkeit) von Objekten auf. Sie erstrecken sich über systemtechnische Grenzen wie Beendigungen von Programmaktivierungen (Prozessen) und nicht selten sogar über Versionsänderungen von Programmen hinweg.

Softwareintegration: Informationssysteme werden in immer größeren organisatorischen Zusammenhängen eingesetzt. Ihre Entwicklung ist im allgemeinen nur dann wirtschaftlich, wenn fremde Investitionen in Form präexistenter Softwarebausteine eingesetzt werden [Schmidt, Matthes 95]. Dabei kann es sich um Dienste unterschiedlichster Art handeln: Gerätetreiber (z.B. für Bildschirme, Drucker, Tastaturen, Zeigeinstrumente, Scanner,

[4] Diese erfolgt in den Kapiteln 2 und 7.

[5] Die Bezeichnung „Persistenz“ wird in der Literatur zwar recht uneinheitlich definiert, doch ihre tatsächliche Verwendung läuft davon unberührt meist auf die oben angegebene Interpretation hinaus. In [Atkinson, Morrison 95] finden sich sogar zwei unterschiedliche Definitionen innerhalb derselben Abhandlung. Zum einen wird Persistenz als der *Sachverhalt* des Unterstützens von Datenwerten (Objekten) über ihre gesamte Lebensdauer definiert, zum anderen sei Persistenz (mit Bezug auf [Atkinson et al. 83]) die *Zeitspanne* während derer ein Objekt existiert und benutzbar ist.

Speichergeräte, Modems), Funktionsbibliotheken (z.B. für Graphik, Mathematik, Datenkommunikation) oder Programme (z.B. Datenbanken, Tabellenkalkulationen, Editoren). Umgekehrt ist es erforderlich, aus verschiedenen Programmiersprachen und Werkzeugen heraus Methoden eines Informationssystems zu aktivieren, um externe Aktionen basierend auf dem persistenten Zustand des Informationssystems zu steuern.

In beiden Richtungen hängt die Qualität der Integration entscheidend von den verwendeten Programmiersprachen und insbesondere von deren externen Schnittstellen ab.

Generische Programmierung: Bei der Erstellung von Informationssystemen treten immer wieder hoch repetitive algorithmische und strukturelle Muster auf (mengenorientierte Anfragen, Integritätsbedingungen, generische Datenstrukturen). Viele Programmierumgebungen bieten Werkzeuge an, die stereotype Programmfragmente (Formulardefinitionen, Formatkonvertierungen, Sortierroutinen, Iteratoren, etc.) anhand parametrisierter textueller oder graphischer Beschreibungen auf hoher Abstraktionsebene automatisch generieren. Seit moderne Typsysteme den typsicheren Einsatz polymorpher Funktionen gewährleisten können, sind diese in den meisten Fällen als überlegene Alternativen anzusehen.

Flexible Verhaltensorganisation: Informationssysteme sind heutzutage nicht mehr nur passive Datenansammlungen, auf denen Anwendungsprogramme operieren. In zunehmendem Maße stellen auch ausführbare Beschreibungen von Teilvorgängen wesentliche Inhalte dar. Nicht zuletzt deshalb ist eine hohe Flexibilität in der Organisation von Programmverhalten wichtig für die Effizienz von Programmentwicklungsumgebungen.

Aufgaben wie die Reaktion auf Ereignisse, Integritätsüberwachung, Ausnahmebehandlung oder auch Präsentationselementen aus Anwendungscode zu eliminieren, fördert die Gesamtkonsistenz von Programmsystemen und mindert den Aufwand ihrer Erstellung. Außerdem kann aus getrennten Programmen herausfaktorisierter Code, auf Teilprobleme bezogen organisiert und für gemeinsame Zugriffe zur Verfügung gestellt werden.

Objektorientierte und aktive Datenbanken sind heute in der Lage, neben statischen Datenstrukturen auch Verhaltensinformation in Form von Methoden, bzw. Regeln persistent zu erfassen. Als erster Schritt in die gleiche Richtung sind in kommerziellen relationalen Datenbanksystemen sogenannte „Stored Procedures“ zu verzeichnen. Dabei handelt es sich jeweils um Code für Datenbankabfragen, der getrennt von Anwendungsprogrammen in einer Datenbank gespeichert werden kann. Hier sind bereits ansatzweise Kombinationen der Eigenschaften von Programmiersprachen der dritten und der vierten Generation zu erkennen. Diese Verbindung wird jedoch in der Entwicklung integrierter Datenbankprogrammiersprachen [Schmidt 77; Koch et al. 83; Matthes, Schmidt 89; Schmidt, Matthes 92], welche in die Entwicklung persistenter Objektsysteme mündet [Matthes et al. 92a; Matthes 93], wesentlich konsequenter vollzogen [Matthes, Schmidt 91; Matthes, Schmidt 93].

Aktivitätsorientierung: Nicht zuletzt unter dem Eindruck des modernen Business Process Reengineering (BPR) [Hammer, Champy 94] ist in jüngerer Zeit ein Trend zu aktivitätsorientierten Informationssystemen entstanden. Workflow-Management-Systeme verwalten gleichzeitig große Mengen nebenläufiger Benutzerprozesse. Der Entwicklungsaufwand ist im Vergleich zu nicht aktivitätsorientierten Informationssystemen jedoch

aufgrund unzureichender Implementationstechniken noch unverhältnismäßig hoch. Typischerweise wird sowohl die persistente Speicherung als auch die Migration von Aktivitäten explizit auscodiert, wobei nicht nur vom Benutzer eingegebene Informationen, sondern auch interne Ablaufzustände betroffen sind.

Ein weiteres Feld der Aktivitätsorientierung moderner Informationssysteme ist der Einsatz mobiler Agenten: aktiver Objekte, die autonom im Netzwerk migrieren und dabei ohne ständige interaktive Kontrolle des Benutzers in dessen Auftrag Handlungen vollziehen.

Plattformunabhängigkeit: Für einen wirtschaftlichen Einsatz von Informationssystemen sind hohe Freiheitsgrade bei der Wahl der einzusetzenden Hardware und Betriebssysteme erforderlich. Typischerweise sind aus organisatorischen Gründen bestimmte Plattformen bereits vorgegeben und zusätzliche Software „muß sich anpassen“. Daher ist die Portabilität von Informationssystemkomponenten von großer praktischer Bedeutung.

Im Zuge der zunehmenden Vernetzung von PCs und Workstations sowie der fortschreitenden Ablösung reinen Terminal-Betriebs durch Client/Server-Architekturen entstehen immer mehr heterogene Umgebungen für Informationssysteme. Dadurch gewinnt deren Portabilität zusätzlich an Bedeutung. Hinzu kommt die Anforderung der plattformunabhängigen Kommunikation, also der Mobilität von Objekten zwischen unterschiedlichen Plattformen.

Auf den meisten dieser Gebiete sind relationale und objektorientierte Datenbanksysteme sowie persistente Objektsysteme recht erfolgreich, wobei letztere besonders vielen Anforderungen gleichzeitig gerecht werden [Matthes 93; Schmidt et al. 93]. Allerdings unterstützen weder Datenbanksysteme noch persistente Objektsysteme die nunmehr geforderte Mobilität von Objekten. Da sich andererseits die Unterstützung von Persistenz durch verteilte Programmiersysteme vergleichsweise rudimentär darbietet, wird festgestellt:

> Es besteht eine Kluft zwischen persistenten Programmiersystemen und verteilten Programmiersystemen mit expliziter Mobilität.

Die vorliegende Dissertation ist von der Überzeugung motiviert, daß durch eine Zusammenführung der Stärken dieser beiden Systemklassen sowie einige Verbesserungen im Hinblick auf die Einsatzfähigkeit in Weitverkehrsnetzen eine zukunftsweisende Infrastruktur für moderne Informationssysteme entsteht.

Um einen ersten Ansatz für eine in diesem Sinne adäquate Systemarchitektur zu erarbeiten, folgt im nächsten Abschnitt eine Gegenüberstellung wichtiger Verteilungs- und Persistenzkonzepte. Durch technische Abgrenzungen, die Feststellung von Synergien und die Befolgung des Orthogonalitätsprinzips wird dabei eine grundlegende Entwurfsentscheidung begründet: die Verwendung eines persistenten Objektsystems als Basis der vorliegenden Arbeit.

Diese Wahl wird im darauffolgenden Abschnitt im Hinblick auf weitere wichtige Eigenschaften persistenter Objektsysteme erläutert. Nach dieser Klärung der Ausgangslage werden in Abschnitt 1.3 die konkreten Ziele der Arbeit spezifiziert. Die Einführung schließt mit einer Übersicht der weiteren Kapitel.

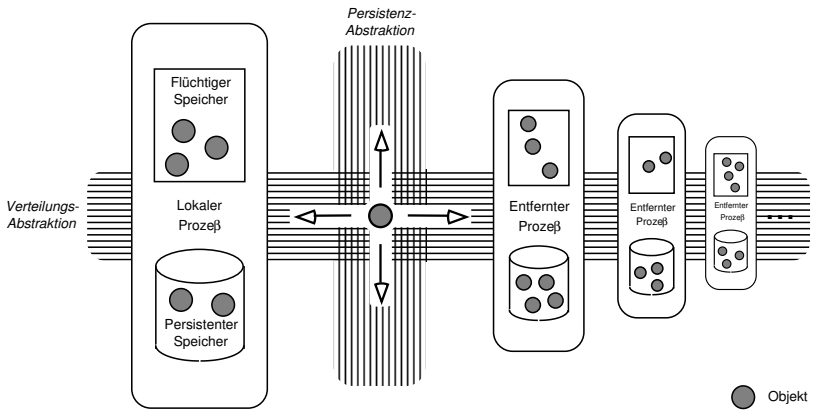

Abbildung 1.2: Horizontale Verteilungs- und vertikale Persistenzabstraktion

1.1 Realisierungskonzepte für Verteilung und Persistenz

Verteilungs- und Persistenzabstraktion ist gemeinsam, daß sie auf sehr ähnliche Weise sehr einfache Programmiermodelle ermöglichen.

- Das Konzept der transparenten Verteilung (auch Verteilungs- oder Ortstransparenz genannt) abstrahiert „horizontal“ von der Lokalität von Objekten: Programmierer brauchen sich nicht darum zu kümmern, ob sich bestimmte Objekte auf dem jeweils lokalen oder einem entfernten Netzwerkknoten befinden.

- In Analogie abstrahieren persistente Systeme „vertikal“ von der Lokalität von Objekten: Programmierer brauchen sich nicht darum zu kümmern, ob sich bestimmte Objekte im Primär- oder Sekundärspeicher befinden.

Abbildung 1.2 illustriert die Orthogonalität dieser Konzepte. Die naheliegende Idee, sie beide in einem System zu vereinen, ist auf unterschiedlichen Systemebenen realisiert worden:

- Verteilte Datenbanken [Ceri, Pelagatti 85; Özsu, Valduriez 91; Bell, Grimson 92] und die auf einem persistenten Objektspeicher basierende Programmiersprache DPS-algol [Wai 88; Wai 89] bieten weitgehende Persistenz- und Ortstransparenz auf der Sprachebene.

- Um die Komplexität höherer Systemschichten zu verringern, wurde Verteilungstransparenz in einigen Arbeiten vollständig innerhalb der Architektur persistenter Objektspeicher angesiedelt [Liskov et al. 90; Gruber et al. 92; Gruber 92; Liskov et al. 92].

- Noch einen Schritt weiter geht der Ansatz, die gesamte Handhabung von Persistenz und Verteilung bereits im Betriebssystem zu leisten [Koch et al. 90; Dearle et al. 94].

Diese Systeme haben jedoch, gerade weil sie von allen Verteilungsaspekten abstrahieren, stark eingeschränkte Anwendungsmöglichkeiten. Im Gegensatz zur Persistenztransparenz erweist sich Verteilungstransparenz nämlich in vielen Fällen als problematisch.

- Verteilungstransparenz *skaliert schlecht*. Sie ist sehr nützlich in Netzwerken mit geringen Latenzzeiten und hoher Zuverlässigkeit. Wenn die Anzahl von Netzwerkknoten sowie die Entfernung zwischen ihnen wächst, treten jedoch vermehrt Kommunikationsverzögerungen und komplexe Fehlersituationen auf, die einer dedizierten Behandlung bedürfen. Kein System ist in der Lage, dies für beliebige Anwendungen zu leisten. Daher werden hier Modelle benötigt, die eine explizite Kontrolle der Verteilung erlauben.

- Wie bereits zuvor erwähnt, stellt die Lokalität eines Objektes, etwa die Zuordnung zu einem bestimmten Netzwerkknoten, in vielen Anwendungssituationen eine *anwendungsrelevante Information* dar. Diese sollte folglich in der Programmierung erfaßbar sein.

- Um Verteilungstransparenz zu realisieren, sind auf tieferliegenden Systemebenen explizite Verteilungsmaßnahmen erforderlich. Wenn eine Anwendungsprogrammiersprache diesbezüglich nicht über adäquate Möglichkeiten verfügt, erzwingt dies eine *sprachliche Trennung* zwischen System- und Anwendungsprogrammierung. In komplexen Anwendungen, die Systemprogrammierung involvieren, entstehen dadurch schwerwiegende architekturelle Brüche.

In der konventionellen Programmierung mit Mehrzweckprogrammiersprachen der dritten Generation (z.B. *C/C++*, Pascal, Modula-2) sind für die Realisierung von Persistenz und Verteilung folgende jeweils analoge Maßnahmen zu treffen:

- Die Entwicklung externer Datenrepräsentationen,

- die Entwicklung von Algorithmen zur Umwandlung von Objekten in ihre externen Repräsentationen,

- die Entwicklung von Algorithmen, die anhand externer Repräsentationen interne Objekte, bzw. deren Zustände wiederherstellen,

- die Einbindung der Algorithmen in übergeordnete Mechanismen wie z.B. globale Zustandsspeicherung oder Kommunikationsprimitive (z.B. RPC).

Aus dieser Analogie resultiert eine Synergie. Diese machen sich einige primär nicht persistente, verteilte Systeme wie z.B. Emerald [Hutchinson 87a; Hutchinson, Jeffery 89] und Modula-3 [Nelson 91; Birell et al. 93b; Horning et al. 93] zu Nutze, indem sie ein und dieselbe externe Repräsentation sowie die dazugehörigen Algorithmen sowohl für die Datenübertragung im Netzwerk als auch für die persistente Speicherung in Dateien verwenden. Die erreichbare Sekundärspeicherperformanz aufgrund der Umwandlung in externe Repräsentationen und fehlender Optimierungen ist um Größenordnungen geringer als in Datenbanksystemen oder in persistenten Objektspeichern. Außerdem bieten diese Systeme keine durchgängige Persistenzabstraktion (vgl. [Atkinson, Bunemann 87]), da die Speicherung bestimmter Objekte nach wie vor explizit angefordert werden muß. Zusammenfassend wird festgestellt:

> Der Versuch, auf Objektmobilität spezialisierten heutigen Programmiersprachen mit ihren originären Mitteln Persistenz zu verleihen, führt nicht zu überzeugenden Ergebnissen.

Unter verändertem Blickwinkel bietet sich eine Synergie an, die mit der oben genannten verwandt ist. Objektsysteme, die eine vollkommene Persistenztransparenz realisieren, weisen aufgrund ihres generalisierten Bindungskonzeptes stets interne Strukturen (insbesondere markierte Objektgraphen) auf, die es auf einfache Weise erlauben, lineare externe Datenrepräsentationen beliebiger Objekte zu generieren (siehe Abschnitt 4.3). Der Einsatz solcher Linearisierungen verspricht insofern, ein genereller Lösungsansatz für den expliziten Datenaustausch zwischen persistenten Systemen zu sein.

Nun fehlt noch die Betrachtung des Ansatzes, Verteilung mit den betriebssystemnahen Techniken persistenter Systeme zu unterstützen. Dieses Konzept wird von Systemen, in denen die Einheit einer Datenübertragung zumindest virtuell jeweils einen gesamten Adreßraum darstellt, erfolgreich umgesetzt. Als Beispiele können hier die Prozeßmigration in fortschrittlichen Microkernel-Betriebssystemen wie Charlotte [Artsy, Finkel 89], Chorus [Rozier 92] und Mach [Milojicic 93] oder UNIX-Erweiterungen wie Condor [Litzkow, Solomon 92], MOSIX [Barak et al. 93] und Sprite [Douglis, Ousterhout 87; Douglis, Ousterhout 91] genannt werden. Wichtige Motivationen dieser Systeme sind das Load-Balancing wenig interaktiver Anwendungen in lokalen Netzwerken und die benutzerbegleitende Migration interaktiver Prozesse [Zayas 87].

Das hier zugrundeliegende Prinzip skaliert jedoch nicht gut in Richtung geringerer Granularität, was anhand der Programmiersysteme SOS [Shapiro et al. 89; Shapiro, Ferreira 93], Soul [Shapiro 91] und COOL [Amaral et al. 92; Lea et al. 93], die auf oben genannten Betriebsystemen aufsetzen, ersichtlich wird. Für die eingangs vorgestellten modernen Anwendungen sind sie aus folgenden Gründen nicht geeignet:

- Sie setzen eine homogene Hardware- und Betriebssystemarchitektur der Rechnerknoten in einem Netzwerk voraus.
- Da zusammenhängende Speichersegmente versendet werden, hängt die Menge der jeweils zu übertragenen Daten sowie die daraus am jeweiligen Zielort resultierende Speichereffizienz kritisch von der Anordnung der Objekte im Ursprungsadreßraum ab. Eine ad-hoc-Anordnung würde jedoch einer Linearisierung gleichkommen. Zudem ist vorsorgliches Clustering in Systemen, die rekursive Datenstrukturen zulassen, wenig effektiv.

Ein entscheidendes Argument zugunsten des Linearisierungsansatzes ist, daß sich die Generierung externer Repräsentationen in Netzwerkanwendungen typischerweise *nicht* wesentlich in Performanzverlusten niederschlägt. Diese im Gegensatz zu den Verhältnissen im Problemfeld der Persistenz stehende Aussage ist auf die relativ geringe Performanz anderer involvierter Komponenten zurückzuführen:

- Die Übertragungsraten der heutzutage installierten Netzwerke sind in aller Regel noch recht gering. Nur in modernen Hochgeschwindigkeitsnetzen verlagert sich das „Performanz-Bottleneck“ zur Zeit manchmal vom Netz in die Rechner, deren Leistung dann nicht mehr genügt, um die volle Netzbandbreite zu nutzen. Es ist sehr schwierig, die weitere Entwicklung dieses Verhältnisses abzuschätzen.
- Auch in Netzwerken mit hoher Bandbreite ändert sich nichts an der durch große Entfernungen gegebenen natürlichen Latenz, die nicht selten einige Hundertstel Sekunden beträgt - in Computerbegriffen sind dies „Ewigkeiten“. Hinzu kommen die typischerweise ähnlich großen Zeitverluste in Vermittlungsrechnern.

- ▷ Die etablierten Übertragungsprotokolle, vor allem TCP/IP, sind sehr aufwendig und damit zeitraubend. Neuere Hochleistungsprotokolle sind zwar schneller, doch auch sie können nicht beliebig weit vereinfacht werden.

In Zusammenfassung der vorangehenden Argumentation wird festgestellt:

> Eine Programmierumgebung für moderne Informationssysteme sollte Persistenz durch betriebssystemnahe Speicherungsmechanismen, Mobilität hingegen mit Hilfe externer linearer Repräsentationen verwirklichen.

Unter diesem Gesichtspunkt hat sich daher als Grundlage für die vorliegende Arbeit ergeben, daß persistente Objektsysteme die vielversprechendsten Voraussetzungen bieten. Im nachstehenden Abschnitt wird nun spezifischer auf deren Eigenschaften eingegangen, wobei deutlich wird, daß sie auch den übrigen oben genannten Anforderungen für die Konstruktion moderner Informationssysteme genügen.

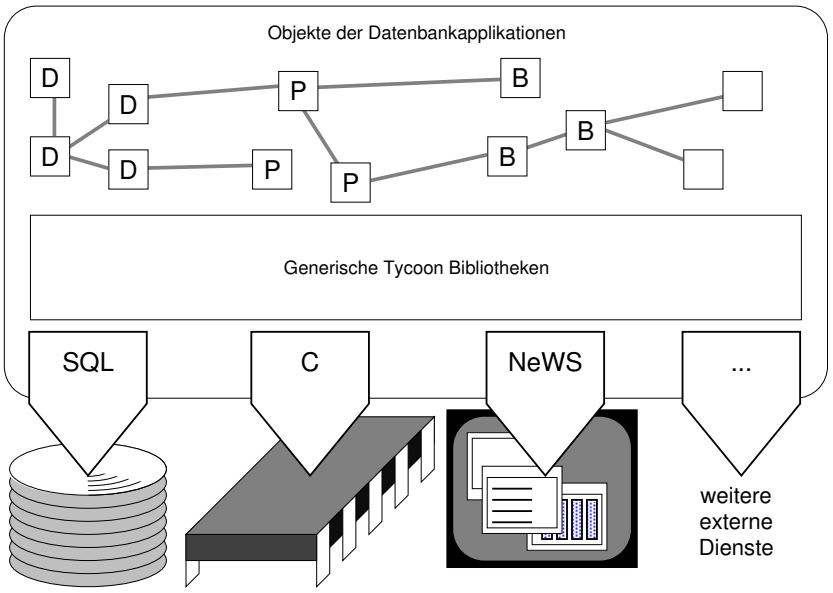

Abbildung 1.3: Die grundlegende Struktur persistenter Objektsysteme [Matthes 93]

1.2 Persistente Objektsysteme als Basis der Arbeit

Persistente Objektsysteme integrieren durch konsequente Anwendung des bekannten Orthogonalitätsprinzips Modellierungs- und Manipulationsmechanismen, die in historisch unabhängig voneinander entstandenen Datenmodellen getrennt vorliegen. Sie überwinden insbesondere die in Sprachen der vierten Generation besonders ausgeprägte statische Trennung zwischen System- und Applikationskomponenten von Informationssystemen.

Die aus [Matthes 93] stammende Abbildung 1.3 stellt schematisch die grundlegende Struktur persistenter Objektsysteme dar: Datenbankanwendungen werden durch die koordinierte Manipulation zahlreicher heterogener Objekte auf verschiedenartigen Medien (D: Datenbankobjekte, P: Programmobjekte, B: Bildschirmobjekte, etc.) in einem unifizierenden sprachlichen Rahmen realisiert, wobei die Funktionalität existierender oder neu zu definierender generischer Diensterbringer ausgenutzt wird.

Persistente Objektsysteme der in [Matthes 93] beschriebenen Qualität vereinheitlichen die Benennung semantischer Objekte wie etwa Relationen, Datenbanken, Sichten, Variablen, Funktionen, Transaktionen, Programme, Bibliotheksfunktionen, Typen, Methoden und Klassen. Sie erreichen ferner eine weitgehende Orthogonalisierung ihrer Bindungs-, Lebensdauer- und Typisierungskonzepte, insbesondere Typorthogonalität [Hull, Su 89], orthogonale Persistenz [Atkinson, Bunemann 87] und uniforme Abstraktion durch Parametrisierung [Harland 84]. Durch Typsysteme höherer Ordnung wird gewissermaßen eine Gleichbehandlung von Datenstrukturen und algorithmischen Elementen (Funktionen) bezüglich der Parametrisierung erreicht, sodaß besonders häufig repetitive Aufgaben datenintensiver Anwendungen schematisiert werden können.

Persistente Objektsysteme sind bewußt in der Absicht entwickelt worden, alle Anforderungen, die die Entwicklung von Informationssystemen stellt, gerecht zu werden [Matthes 93; Schmidt et al. 93]. Diese Bemühungen waren auf fast allen Gebieten anerkanntermaßen sehr erfolgreich [Atkinson 95]. Nur auf dem Gebiet der Mobilität ist bisher noch keine hinreichende Leistungsfähigkeit zu verzeichnen. Erste Arbeiten [Munro 93; Mira da Silva 95a; Mira da Silva 95b] in diesem Bereich decken nur reine Kopiersemantik ab und können daher mit originär verteilten Sprachen (z.B. Emerald [Jul 88], Obliq [Cardelli 94]) nicht konkurrieren. Aufgrund folgender Eigenschaften sind persistente Objektsysteme jedoch geradezu prädestiniert für verteilte Programmierung:

- Ihre Speicherverwaltung und Datenrepräsentation basiert auf Garbage-Collection-Mechanismen. Durch transitive referentielle Erreichbarkeit ergeben sich klar definierte Objektgraphen. Diese Tatsache läßt sich in natürlicher Weise für die Linearisierung beliebiger Daten zum Zwecke ihrer Übertragung ausnutzen.

- Analog zu LISP-Systemen gilt ein generalisierter Bindungsbegriff, der es Anwendungsprogrammen gestattet, dynamisch Bindungen an bereits existierende Datenstrukturen auf der (persistenten) Halde zu erzeugen und bestehende Bindungen wieder zu lösen. Es ist zu erwarten, daß über ein Netzwerk in ein persistentes Objektsystem übertragene Daten in analoger Weise mit bereits vorhandenen lokalen Daten verknüpft werden können. Dadurch ergäbe sich eine bruchlose, natürliche Integration mobiler Daten.

- Da referentielle Erreichbarkeit Persistenz impliziert, verwenden persistente Objektsysteme „sparsame“ Laufzeitrepräsentationen, insbesondere auch für Funktionsabschlüsse [Mathiske 92]. Es wird streng darauf geachtet, daß nur Daten mit effektivem Nutzen referenziert werden. Die Folge wäre sonst ungewollte Persistenz obsoleter Daten und damit dauerhafte, praktisch irreversible Speicherplatzverschwendung. Die persistenten Objektsystemen eigene Sparsamkeit ist auch im Falle von Datenübertragungen günstig. Sie schützt vor überflüssigem Übertragungsvolumen und vor nutzlosen Datenkopien in entfernten Adreßräumen.

- Statische Typsysteme höherer Ordnung mit zusätzlichen dynamischen Typrepräsentationen, die zur Laufzeit manipuliert werden können, erlauben ein äußerst flexibles Typmanagement unter Wahrung maximaler Typsicherheit.

- Es sind portable Code-Repräsentationen vorhanden.

- Die Exekution wird von einer virtuellen Maschine durchgeführt. Wie am Beispiel Java [Sun 95c; Yellin 95] ersichtlich, ist dies eine gute Voraussetzung für gezielte Authentisierung und Autorisierung.

- Multi-Threading wird unterstützt. Dies ist eine wichtige Voraussetzung für effektive Client/Server-Kommunikation.

Integrierte Datenbankprogrammiersprachen, zu denen insbesondere auch persistente Objektsysteme gezählt werden, haben die Qualität datenintensiver Anwendungen bisher durch Beiträge auf zwei Ebenen entscheidend verbessert. Auf der Sprachebene liefern sie flexible Benennungs-, Typsierungs- und Bindungsmechanismen zwischen allen Objekten, die für eine auf einem integrierten Modell für persistente Daten, Code und Threads basierende datenintensive

Anwendung relevant sind [Atkinson, Bunemann 87; Matthes, Schmidt 94]. Auf der Systemebene liefern sie eine passende integrierte Technologie (z.B. markierte Objektrepräsentationen, Iterationsabstraktionen über multiple Massendatenstrukturen, Garbage-Collection, virtuelle Maschinen, portable Coderepräsentationen), die schwerwiegende Brüche nicht-integrierter Umgebungen überwindet.

Es ist eine entscheidende Herausforderung für die Datenbankgemeinde, sich nun dem schwierigen Problem zu widmen, den erfolgreichen Begriff eines uniformen, typsicher persistenten Objektspeichers zu skalieren [Mathiske et al. 95b], um *kommunikationsintensiven* Anwendungen gerecht zu werden. Besonders hohe Anforderungen stellen langandauernde Aktivitäten, die sich über mehrere autonome Netzwerkknoten in einem weltweiten Netzwerk erstrecken [Mathiske et al. 95a].

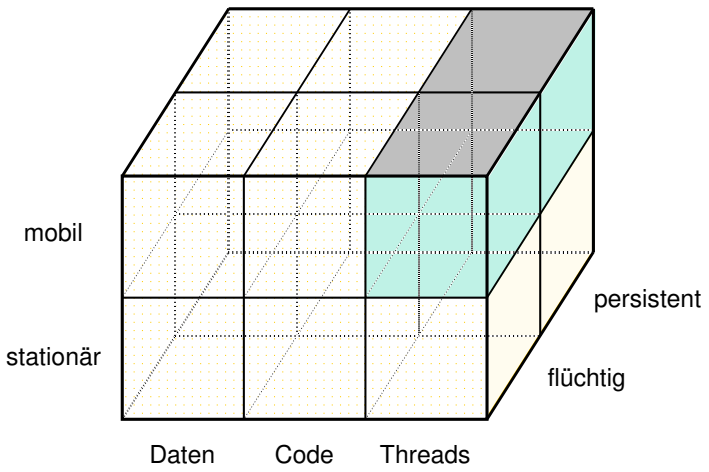

Abbildung 1.4: Der kombinatorische Raum von Datentypen, Persistenz und Mobilität

1.3 Ziele und Lösungsansätze

Das Ziel der im Rahmen dieser Arbeit entworfenen und prototypisch implementierten Erweiterungen eines persistenten Objektsystems ist die Bereitstellung eines *einheitlichen* programmiersprachlichen und architektonischen Rahmens in offenen Systemumgebungen, der orthogonale Persistenz mit einer substantiell verbesserten *Mobilität* von Objekten beliebigen Typs verbindet. Durch die Unterstützung möglichst direkter, unkomplizierter Abbildungen verteilter Anwendungsstrukturen in korrespondierende Programmiersprachenkonstrukte soll eine signifikante Erhöhung der Flexibilität und Qualität bei gleichzeitiger Minderung des Aufwandes der Entwicklung datenintensiver verteilter Anwendungen erreicht werden.

Der in Abbildung 1.4 idealisierend dargestellte, von Datentypen, Persistenz und Mobilität aufgespannte, kombinatorische Raum wird von heutigen Programmiersystemen für Informationssysteme ungenügend abgedeckt. Insbesondere werden die rechten oberen Quadranten, in denen die Mobilität von Code und Threads angesiedelt ist, vernachlässigt. In der vorliegenden Arbeit wird nun eine verbesserte Annäherung an das *Ideal*, Mobilität als zusätzliche Dimension zu unterstützen, angestrebt.

Mobilität ist in Bezug auf Datentypen allenfalls insofern eine weitere orthogonale Eigenschaft, als daß sich von jedem Datentyp irgendein Objekt instanzieren läßt, das in irgendeiner Art und Weise mobil ist. In der Praxis ist nämlich nicht erreichbar, daß beliebige Objekte jeden Datentyps in jeder beliebigen Situation mobil sind (siehe die in der Einleitung auf Seite 4 genannten, in der Praxis auftretenden Fragestellungen). Allerdings eröffnen Fortschritte in dieser Richtung zusätzliche Möglichkeiten, Anwendungsstrukturen direkt und unkompliziert auf Programmiersprachenkonstrukte abzubilden und so eine große Klasse von Anwendungen zu vereinfachen.

Um zukunftsweisende Fortschritte in der Entwicklung von Infrastrukturen für Informationssysteme in Weitverkehrsnetzen zu erarbeiten, konzentriert sich die vorliegende Dissertation auf die Lösung folgender Teilaufgaben:

1. Die Realisierung der hochgradigen Plattformunabhängigkeit eines persistenten Objektsystems und seiner Kommunikationsmechanismen.

2. Die Bereitstellung eines mobilitätsorientierten RPC-Mechanismus.

3. Die Realisierung aktiv migrierender peristenter Threads als Konstrukte für autonome migrierende Aktivitäten (mobile Agenten, Workflows), die langlebig sind.

4. Die Bereitstellung in Weitverkehrsnetze skalierender Techniken zur Bindung immobiler Objekte durch migrierende Objekte.

Bei allen vier Teilaufgaben besteht ein sehr vielversprechender Ansatz darin, die vorteilhaften Eigenschaften persistenter Objektsysteme auch für mobile Objekte nutzbar zu machen und so auf die Domäne der verteilten Programmierung zu übertragen. Der Ausgangspunkt der vorliegenden Arbeit ist das in vielerlei Hinsicht als besonders fortschrittlich angesehende [Atkinson 95] persistente Objektsystem *Tycoon* (siehe Abschnitt 1.2 und Kapitel 3), das im Arbeitsbereich DBIS des Fachbereiches Informatik der Universität Hamburg entwickelt worden ist.

Die konkret verfolgten Problemstellungen und Lösungsansätze lassen sich wie folgt näher charakterisieren:

1. **Die Realisierung der hochgradigen Plattformunabhängigkeit eines persistenten Objektsystems und seiner Kommunikationsmechanismen.**

   Eine wichtige Grundsatzentscheidung zu Beginn der vorliegenden Arbeit besteht darin, von vornherein streng auf Portabilität zu achten und *nicht* erst nach Fertigstellung weiter Teile des Systems mit Portierungen zu beginnen. Diese radikale Portabilitätsorientierung betrifft nicht nur das Laufzeitsystem. Um die Netzwerkkommunikation zwischen allen Plattformen, auf die Portierungen vorgenommen worden sind, zu ermöglichen, ist eine hochgradige Plattformunabhängigkeit der Kommunikationsmechanismen erforderlich. Diese ist am besten durch eine möglichst schmale und daher wenig wartungsintensive Implementationsbasis zu gewährleisten.

2. **Die Bereitstellung eines mobilitätsorientierten RPC-Mechanismus.**

   Für die im Tycoon-Projekt vorgegebene funktionale Sprache TL (*Tycoon Language*) mit imperativen Konstrukten bieten sich synchrone[6] RPCs (*Remote Procedure Calls* [Nelson 81; Nelson, Birrell 84; Schill 92], zu deutsch „entfernte Prozeduraufrufe") als Paradigma für verteilte Programmierung an. RPCs fassen Kommunikationsprimitive geringeren Abstraktionsgrades in einer handlichen Form zusammen und sie lassen sich nahtlos in die Wirtssprache einbetten. Im Sinne einer einheitlichen Benennung lokaler und entfernter Objekte gleichen Typs, die sich an den Gegebenheiten in TL orientiert, ist im folgenden von „entfernten Funktionen" und „entfernten Funktionsaufrufen" die Rede. Ferner wird davon ausgegangen, daß jeweils eine oder mehrere Funktionen in einem sogenannten RPC-Dienst oder „entfernten Dienst" aggregiert sind.

[6] *Synchrone* RPCs stellen hier keine besondere Einschränkung gegenüber *asynchronen* Alternativen dar, da letztere dank Multi-Threading im Tycoon-System bedarfsweise mit geringem Aufwand konstruiert werden können.

Das Ziel, eine substantiell verbesserte *Mobilität* von Objekten beliebigen Typs zu erreichen, konkretisiert sich im Umfeld der vorliegenden Arbeit zu der Anforderung, daß bei Bedarf möglichst jedes beliebige Tycoon-Objekt als RPC-Argument oder -Rückgabewert verwendbar sein sollte.

Ein durchgängig zu beobachtendes, gravierendes Defizit früherer Arbeiten über RPCs [Tanenbaum, van Renesse 88; Corbin 91; Schill 93; Mira da Silva 95a; Mira da Silva 95b] besteht darin, aufgrund in den jeweiligen Systemen unüberwindbarer technischer Schwierigkeiten die Menge der als Parameter für RPCs zulässigen Typen zu beschränken. Besonders häufig betroffen sind Objekte, die Code enthalten (Funktionen), bzw. Zustände und Code kapseln (Funktionsabschlüsse), sowie Programmausführungszustände (Threads) und Kommunikationsidentifikatoren. Jede dieser Einschränkungen wirft erhebliche Probleme für die Anwendungsprogrammierung auf, indem direkte Abbildungen von Anwendungsstrukturen auf Programmierkonstrukte, die sich bei lokaler Programmierung in natürlicher Weise ergeben, quasi an RPCs zerbrechen. Ein wichtiges Teilziel der vorliegenden Dissertation ist daher, die Problematik der Mobilität so weit wie möglich von Typabhängigkeiten zu befreien.

Ein weiterer Aspekt persistenter Objektsysteme, der verbesserte Qualitäten eines RPC-Mechanismus nahelegt, ist die orthogonale Persistenz [Atkinson, Bunemann 87; Matthes 93; Atkinson, Morrison 95]. Zwar läßt sich der lokale Persistenzbegriff persistenter Objektsysteme nicht ohne weiteres auf (weit-) verteilte Umgebungen übertragen [Dearle et al. 91; Munro 93; Atkinson, Morrison 95], denn dies würde die unbedingte Bereitschaft verteilter Anwendungskomponenten bei gegenseitiger Abhängigkeit bedeuten. Es sollte jedoch möglich sein, Bindungen an entfernte Dienste insofern Persistenz zu verleihen, daß zum einen das Ausprogrammieren des Wiederanlaufens von Servern entfällt und zum anderen Clients einmal eingegangene Bindungen nach zeitweiligen Unterbrechungen von Prozessen oder Netzwerkverbindungen nicht erneuern müssen.

Genauer betrachtet geht es also darum, die *Adressierung* entfernter Dienste persistent zu gestalten. Systemnahe Basiskonstrukte zur Netzwerkprogrammierung (etwa Sockets [Corbin 91]) sind jedoch flüchtig. Ein erster Lösungsansatz besteht darin, nach dem Vorbild der *port mapper* für den ONC-RPC [Santifaller 93] an wohlbekannten Adressen im Netzwerk spezielle Dienstprogramme bereitzustellen, um aktuelle Netzwerkadressen entfernter Dienste an RPC-Clients zu liefern. Daraufhin ist ein flexibler Client-Stub zu entwickeln, der in der Lage ist, *selbsttätig* Anfragen an Netzdienste zu stellen und dynamisch Netzwerkverbindungen aufzubauen.

Auf diese Weise ist außerdem die Mobilität kommunizierender Objekte zu unterstützen. Objekte sollen ihre Bindungen an entfernte Objekte (Dienste) bei Migrationen „mitnehmen“ können, ohne sie erneuern zu müssen. Die Erfahrung mit verteilten Programmiersprachen wie Emerald [Jul 88] und Obliq [Cardelli 94] zeigt, daß dies eine unbedingte Voraussetzung für die effiziente Erstellung verteilter Anwendungen mit mobilen Komponenten (gleich welcher Granularität) ist.

Eine weitere Aufgabe von Netzdiensten besteht darin, die Auswahl entfernter Dienste zum Zwecke der Bindung durch Clients zu vermitteln [ISO-ODP 95]. Dabei sind Diensttypen besonders wichtige Auswahlkriterien, denn durch dynamische Typvergleiche [Matthes 93; Geisler 95] kann eine typsichere Dienstverwendung gewährleistet werden. Allerdings bieten Typen nur eine eingeschränkte, rein syntaktische Beschreibung

von Diensteigenschaften. Deshalb müssen zusätzlich je nach Anwendung verschiedenartig gestaltbare Attribute zur Dienstauswahl herangezogen werden können.

3. **Die Realisierung aktiv migrierender peristenter Threads als Konstrukte für autonome migrierende Aktivitäten (mobile Agenten, Workflows), die langlebig sind.**

   Die angestrebte Typunabhängigkeit des zu entwickelnden RPC-Mechanismus impliziert insbesondere die Mobilität von Threads, wodurch sich weitreichende Perspektiven für die aktivitätsorientierte Programmierung eröffnen. Dies gilt um so mehr in persistenten Objektsystemen, in denen sich auf natürliche Weise das Konzept persistenter Threads ergibt [Matthes, Schmidt 94].

   Threads [POSIX 90] sind ein natürliches Sprachkonstrukt zur Modellierung konkurrenter Aktivitäten. In konventionellen Systemen können Threads jedoch weder migrieren noch sind sie persistent. Um Ausführungszustände dauerhaft zu erfassen und zwischen Adreßräumen zu übertragen, muß in fast allen Programmiersprachen auf andere Sprachkonstrukte zurückgegriffen werden. Dann erfolgt die Codierung von Ausführungszuständen zwangsläufig durch *explizite* Ausprogrammierung, was im allgemeinen extrem aufwendig ist. Dementsprechend hoch ist das Potential migrierender persistenter Threads [Mathiske et al. 95a] für *elegante* (d.h. einfache und unaufwendige) Lösungen komplexer Anwendungsprobleme.

   Die Mobilität von Threads ist zwar im wesentlichen eine Konsequenz der oben genannten Anforderungen an einen RPC-Mechanismus, jedoch bedarf es der Konkretisierung der speziell für Threads erforderlichen internen Repräsentationen. Diese sowohl persistent als auch plattformunabhängig mobil zu gestalten, erscheint für derartig maschinennahe Konstrukte keineswegs trivial.

   Um *autonome* Objekte wie mobile Agenten [Wayner 94] oder Workflows [WFMC 95] zu implementieren, sollen Threads nicht (nur) passiv als RPC-Parameter fungieren, sondern vielmehr selbst ihre eigenen Migrationen veranlassen. Ferner sollen sie in der Lage sein, dynamische Bindungen an jeweilige lokale Objekte einzugehen und wieder zu lösen, um lokale Ressourcen zu nutzen und mit anderen Aktivitäten zu kooperieren [Mathiske et al. 95a].

   In eine Thread-Migration involvierte Objektsysteme müssen *dauerhaft* eine *systemübergreifende* Übereinstimmung über den Verbleib des jeweiligen Threads erzielen. Dies bedeutet, daß Thread-Migrationen als *verteilte Transaktionen* [Gray, Reuter 93] durchgeführt werden müssen. Andernfalls könnten Threads verloren gehen oder in mehreren Instanzen auftreten.

   Um ihre autonome Handlungsfähigkeit zu bewahren, sollen Threads *komplett* migrieren, d.h. sie sollen nicht aus *systemtechnischen* Gründen auf rückwärtsgerichtete Kommunikationsverbindungen angewiesen sein. Die grundlegende Vorgehensweise bei Thread-Migrationen muß daher in tiefem rekursiven Kopieren aller potentiell benötigten Objekte an den Zielort bestehen. Andererseits soll Thread-Migrationen durch die Übertragung von so wenig Daten wie möglich ausgeführt werden. Das bereitzustellende Repertoire von Bindungstechniken muß diese Anforderungskombination berücksichtigen.

4. **Die Bereitstellung in Weitverkehrsnetze skalierender Techniken zur Bindung immobiler Objekte durch migrierende Objekte.**

   In jedem Programmiersystem treten unvermeidlich Objekte auf, die aufgrund fester Bindungen an stationäre Systemkomponenten immobil sind. Typische Gründe für Immobilität sind: die eingeschränkte Verfügbarkeit (z.B. Plattformabhängigkeit, Lizensierung, ortsgebundene Installation) externer Software und enge Bindungen an spezielle Hardware (z.B. Drucker, Scanner, Kamera).

   Ein nicht weniger gravierendes Problem bereitet die durch schieres Datenvolumen bedingte, praktische Immobilität besonders komplexer Objekte. Da hiervon nicht nur Massendaten wie z.B. Datenbankrelationen, sondern insbesondere auch Funktionen und Threads betroffen sind, ist die Einsatzfähigkeit von Code-Mobilität und Thread-Migration zunächst ernsthaft in Frage gestellt.

   Das Problem der Immobilität wird durch seine Transitivität verschärft: Objekte, die feste Bindungen an immobile Objekte aufweisen, sind ebenfalls als immobil zu betrachten.

   Auf der Anwendungsebene können migrierende Objekte ein naheliegendes programmiersprachliches Mittel einsetzen, um sich bei Bedarf von immobilen Objekten zu lösen: *dynamische Bindungen*, d.h Zuweisungen und Parameterübergaben. Allerdings ist bei dynamischen Neubindungen, die im Zuge von Migrationen auftreten, darauf zu achten, daß sie typsicher sind und es dürfen keine *Dangling References* entstehen.

   In ausgebauten Programmiersystemen machen Anwendungsprogramme üblicherweise starken Gebrauch von Bibliotheksfunktionalität wie etwa von Fenstersystemen, Kommunikationssoftware, Massendatenstrukturen, usw. Gewöhnlich sind die betreffenden Bibliotheken auf praktisch allen in eine Anwendung involvierten Netzwerkknoten als Kopien installiert: sie sind *ubiquitäre* Objekte. Außerdem sind sie typischerweise praktisch zustandslos. Aufgrund dieser Charakteristik bietet sich *dynamisches Linken* als eine systemnahe, aber sprachlich transparente Alternative zu dynamischen Bindungen an.

   In begrenztem Umfang wird dynamisches Linken bereits durch einige verteilte Programmiersprachen und verteilte Objektmanagementsysteme genutzt [Shapiro et al. 89; Shapiro 93; Knabe 95]. Möglicherweise wird von dieser Technik in vielen Systemen auch deshalb noch nicht im größeren Umfang Gebrauch gemacht, weil es schwierig ist, ubiquitäre Installationen von Objekten zustande zu bringen.

   Die Aufgabe der Installation sollte in Zukunft nicht notwendigerweise im Bereich der Systemadministration angesiedelt sein, denn schließlich bedeutet die Mobilität von Objekten, daß sie durch Anweisungen einer Anwendungsprogrammiersprache verteilt werden können. In Emerald [Jul 88] und Java [Sun 95b; Golsing, McGilton 95] wird Code automatisch über das Netzwerk geladen und bei wiederholten Zugriffen dynamisch gelinkt. Es erscheint sehr vielversprechend, dieses elegante Konzept auf Objekte *beliebigen* Typs zu verallgemeinern und mit orthogonaler *Persistenz* zu kombinieren. Denn ohne dauerhafter Speicherung über das Netzwerk installierter Objekte sind in langlebigen Anwendungen sonst Wiederholungen der betreffenden Installationsvorgänge erforderlich.

   Durch die Einbindung externer Software in ein persistentes Objektsystem entstehen Bindungen an *externe Objekte*, die nicht unter direkten Kontrolle des persistenten Ob-

jektsystems stehen. Daher sind sie aus dessen Sicht zunächst flüchtig. Denn dafür ist hinreichend, daß zumindest die Referenzen auf sie, typischerweise sogar ihre gesamten Repräsentationen, bei einem Neustart explizit erneuert werden müssen. Außerdem sind externe Objekte immobil, weil von internen Kommunikationsmechanismen lediglich ihre Referenzen, aber nicht ihre Repräsentationen erfaßt werden. Externe Objekte stellen also ernsthafte Hindernisse für die Persistenz und die Mobilität in Anwendungen persistenter Objektsysteme dar.

Das persistente Objektsystem Napier88 [Morrison et al. 94] umgeht diese Problematik durch Verzicht auf externe Programmierschnittstellen. Diese Entscheidung wird in der vorliegenden Arbeit nicht nachvollzogen, da die Einbindung externer Software zur (ökonomischen) Erstellung komplexer Anwendungen als unverzichtbar[7] angesehen wird. Stattdessen wird die Herausforderung angenommen, Methoden zu entwickeln, die externe Objekte in die fortgeschrittenen Leistungen eines persistenten Objektsystems integrieren.

Aufgrund der Entscheidung, die Offenheit des Tycoon-Systems nicht preiszugeben, eröffnet sich die Möglichkeit, eine fortschrittliche persistente Programmiersprache höherer Ordnung als *Glue Language* zur Verknüpfung *heterogener* verteilter Softwarekomponenten einzusetzen.

[7] Siehe die Anforderungen an Programmierumgebungen für Informationssysteme in der Einleitung.

1.4 Gliederung der Arbeit

Im nächsten Kapitel wird in Form einer Bestandsaufnahme der Mobilität in repräsentativen Programmiersystemen die Ausgangssituation der vorliegenden Arbeit konkretisiert, und es werden aus den betrachteten Erfahrungen erste wichtige Entwurfsentscheidungen abgeleitet.

In Kapitel 3 wird das persistente Objektsystem Tycoon [Matthes 93] als Basis der vorliegenden Arbeit vorgestellt. Nach einem Überblick der Schichtenarchitektur des Tycoon-Systems wird in die Programmiersprache TL (*Tycoon Language*), die im folgenden als Notation für Programmauszüge und -beispiele dient, eingeführt. Dabei werden zugleich die im Zusammenhang der vorliegenden Arbeit bedeutsamen Eigenschaften von TL (orthogonale Persistenz, Funktionen höherer Ordnung, polymorphes Typsystem mit dynamischen Typrepräsentationen) erläutert. Ein erster Beitrag im Rahmen der vorliegenden Dissertation ist die Entwicklung einer *transparenten bidirektionalen* Programmierschnittstelle zwischen TL und *C*.

Die Hauptbeiträge der Arbeit beginnen mit Kapitel 4, das die plattformunabhängige Gestaltung des Tycoon-Systems beschreibt. Eine Schnittstelle zur portablen Socket-Programmierung in *C* und ein Linearisierungsmechanismus zum plattformunabhängigen Datenaustausch zwischen persistenten Objektspeichern bilden die schmale Implementationsbasis der im weiteren Verlauf der Arbeit entwickelten höheren Kommunikationsmechanismen.

Kapitel 5 stellt mobilitätsorientierte entfernte Funktionsaufrufe vor. Diese bilden den Kern der Arbeit, mit dem alle anderen Beiträge im Zusammenhang stehen. Es wird erläutert, durch welche Entwurfsentscheidungen besonders orthogonale Eigenschaften entfernter Funktionen entstehen, es wird exemplarisch gezeigt, welchen Nutzen dies in der Programmierung hat und es werden die erforderlichen Implementierungstechniken eingehend beschrieben.

In Kapitel 6 wird anhand praktischer Beispiel die direkte Abbildung von Anwendungsaktivitäten auf flüchtige, persistente und mobile Threads vorgestellt. Dabei wird unter Einsatz mobilitätsorientierter entfernter Funktionsaufrufe das Konzept migrierender persistenter Threads entwickelt. Im zweiten Teil des Kapitels werden dann Implementationstechniken beschrieben, die die Persistenz und Mobilität von Threads sowie transaktionale Migrationen von Threads verwirklichen.

Kapitel 7 beschreibt Bindungsmechanismen, die in wichtigen Anwendungsfällen die effektive Durchführbarkeit der Migration komplexer Objekte überhaupt erst ermöglichen, indem sie deren Autonomie unterstützen. Es sind dies typsichere temporäre Bindungen an stationäre Objekte und dynamisches Linken ubiquitärer Objekte. Auf der Grundlage letzterer Technik wird ferner ein Mechanismus für die automatische Replikation beliebiger Objekte entwickelt. Außerdem wird ein Logging-Verfahren zur Integration flüchtiger Daten vorgestellt.

Im letzten Kapitel werden die Beiträge und Erfahrungen der vorliegenden Dissertation zusammenfassend betrachtet. Darauf folgt ein Ausblick auf weiterführende Anschlußarbeiten.

In den Anhängen sind wichtige Aspekte technischer Realisierungen der vorgestellten Beiträge zusammengefaßt, auf die im Text Bezug genommen wird.

Kapitel 2

Mobilität in existierenden Programmiersystemen

In diesem Kapitel werden in Form einer überblicksartigen Bestandsaufnahme der Mobilität in repräsentativen Programmiersystemen die relativen Stärken und Schwächen unterschiedlicher Systemarchitekturen untersucht. Diese Analyse motiviert die Entwicklung der grundlegenden Modifikationen und Erweiterungen des persistenten Objektsystems Tycoon und die Ausgestaltung seiner Kommunikationsmechanismen.

Abschnitt 2.1 führt in knapper Form in relevante Begriffe der Internet-Programmierung ein, um die in der vorliegenden Arbeit verwendeten Bezeichnungen aus diesem Bereich zu erläutern und stellt gängige Grundtechniken der Kommunikationsprogrammierung vor. Im Anschluß daran (Abschnitt 2.2) werden komplexere Spracherweiterungen zur verteilten Programmierung behandelt. Diese sind insbesondere im Hinblick auf Systemoffenheit von Interesse.

Die vorliegende Arbeit orientiert sich vorwiegend an den in Abschnitt 2.3 vorgestellten, von vorn herein für die verteilte Programmierung konzipierte Programmiersprachen, da in diesen die höchsten Mobilitätsgrade erreicht werden.

In Abschnitt 2.4 wird auf einen aktuellen Trend in der Entwicklung verteilter Programmiersprachen eingegangen: die spezielle Ausrichtung auf mobile Agenten. In diesem Zusammenhang sind relevante Neuentwicklungen entstanden, die wichtige Ansätze der vorliegenden Dissertation bestätigen.

Nach einer Betrachtung der bisherigen Mobilität in persistenten Objektsystemen (Abschnitt 2.5) werden die in diesem Kapitel beschriebenen Beobachtungen zusammengefaßt und Entwurfsentscheidungen aus ihnen abgeleitet.

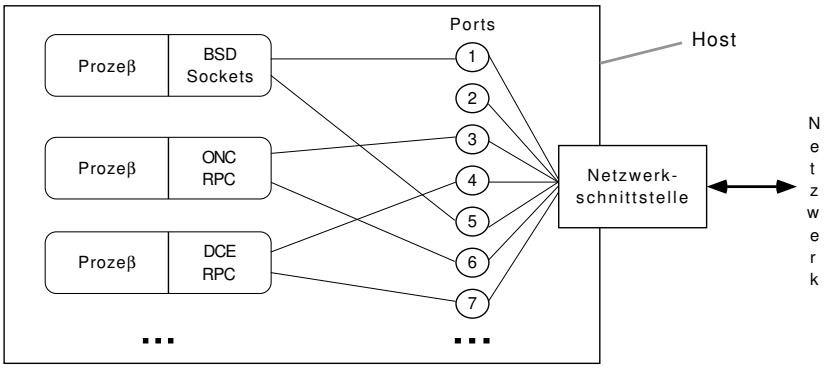

Abbildung 2.1: Die Rolle von Ports bei der Netzwerkkommunikation

2.1 Kommunikationsprogrammierung in TCP/IP-Netzwerken

Im Internet sind einige Millionen Computer durch das Kommunikationsprotokoll IP (*Internet Protocol*) miteinander verbunden. Die Netzwerkknoten als fungierenden Computer werden als *Hosts* bezeichnet. Die Verbindungsstruktur ist ein vollständiger Graph: jeder Host kann mit jedem anderen Kontakt aufnehmen. Dazu muß ein Host ggf. die Internet-Adresse (*IP-Address*) eines anderen verwenden. Internet-Adressen beziehen sich genaugenommen nicht direkt auf Hosts, sondern auf logische Netzwerkschnittstellen, die sich letztlich auf physikalische Netzanschlüsse (z.B. Steckkarten) zurückführen lassen.

Eine Internet-Adresse ist ein 32-Bit-Wert, für den zwei handliche Darstellungsformen als Zeichenkette vorgesehen sind. Allgemein üblich ist die Umsetzung der vier Bytes einer Adresse in ein durch Punkte getrenntes Quadrupel (*dotted quad*). Hier ein Beispiel:

"134.100.11.121"

Eine verteilte Datenbank, der *Domain Name Service* (DNS), ordnet den hierarchisch geordneten Teilbereichen (*Domains*) des Internet Namen zu. Eine Adresse kann dann durch eine Folge solcher Namen (*fully qualified domain name*) angegeben werden. Zum Beispiel entspricht folgende Angabe obiger Zahlenkombination:

"dbis1.informatik.uni-hamburg.de"

Damit über eine Netzwerkschnittstelle eines Hosts mehrere Prozesse und innerhalb derer mehrere verschiedene Aktivitäten gleichzeitig adressierbar sind, werden einzelne Kommunikationsverbindungen durch sogenannte Port-Nummern unterschieden. Die Kombination einer Internet-Adresse mit einer Port-Nummer bezeichnet einen logischen Kommunikationsendpunkt, einen *Port*.

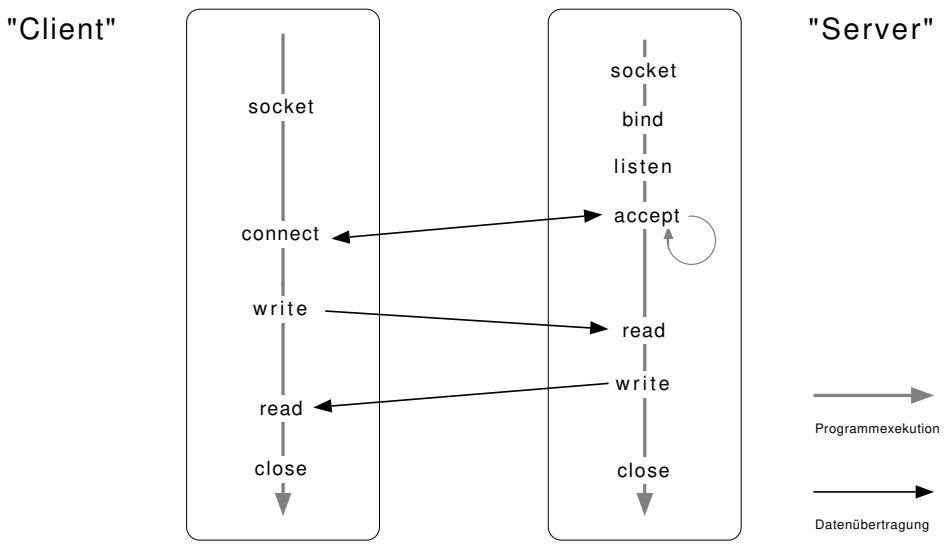

Abbildung 2.2: Ein typisches Szenario der Socket-Kommunikation

Die Datenübertragung im Internet ist paketorientiert. Auf der Basis von IP dient das verbindungslose Protokoll UDP (*User Datagram Protocol*) dient, einzelne Pakete, sogenannte *Datagramme* zu versenden. Alternativ kann das verbindungsorientierte Protokoll TCP (*Transport Control Protocol*) eingesetzt werden.

Die gebräuchlichsten[1] Programmierschnittstellen für UDP und TCP sind BSD-Sockets und ONC-RPC (auch Sun-RPC genannt) [Corbin 91; Santifaller 93] sowie DCE-RPC [Open Software Foundation 93]. Abbildung 2.1 zeigt einen Host, auf dem drei Prozesse über jeweils unterschiedliche Programmierschnittstellen Ports verwenden.

2.1.1 BSD-Sockets

Sockets sind Datenstrukturen, die logische Kommunikationsverbindungen beschreiben. Die BSD-Socket-Schnittstelle sieht eine Reihe von Standardoperationen vor, mit denen Datenströme versendet bzw. empfangen werden können. Abbildung 2.2 zeigt ein typisches Aufrufszenario für diese Operationen, die in Form von *C*-Funktionen gegeben sind.

Der linke Prozeß erzeugt ein Socket (*socket*) und initiiert dann eine Kommunikation (*connect*) und übernimmt hier die Rolle eines *Client*. Der rechte Prozeß wartet auf eingehende Kommunikationsaufforderungen (*accept*) und reagiert dann auf diese - er übernimmt die Rolle eines *Servers*. Die Datenübertragung geschieht (hier) mit Hilfe der Standardfunktionen *write* und *read*, wobei sich die verwendeten BSD-Sockets aus der Sicht des Programmierers wie geöffnete Dateien (*file handles*) verhalten. Weitere Details werden in Abschnitt 4.2 im Zusammenhang der Socket-Programmierung im Tycoon-System erläutert.

[1] Der ISO-Standard für RPCs [ISO-RPC 91] ist dem ONC-RPC und dem DCE-RPC recht ähnlich. Mangels verfügbarer Implementationen ist er in der Praxis jedoch so gut wie bedeutungslos.

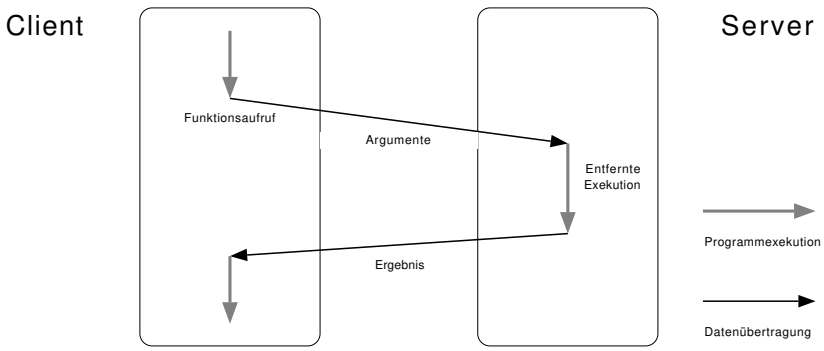

Abbildung 2.3: Das Grundprinzip des RPC

2.1.2 Gängige RPC-Mechanismen

Ein RPC *Remote Procedure Call* [Nelson 81; Nelson, Birrell 84; Corbin 91; Schill 92] faßt die obigen Vorgänge zusammen und bettet sie als Funktionsaufruf transparent in einen Programmquelltext ein. Abbildung 2.3 zeigt den Ablauf eines RPC. Ein Client ruft eine lokale Funktion auf, deren Exekution auf einem entfernten Server vorgenommen wird. Bei einem „synchronen“ RPC blockiert der aufrufende Client-Thread, bis das Ergebnis (oder eine Fehlermeldung) vorliegt.[2]

RPC-Server können mehrere RPCs parallel bearbeiten, indem sie für jeden Aufruf einen eigenen Subprozeß erzeugt. Je nach Programmiersystem und Anwendung handelt es sich hierbei um einen Betriebssystemprozeß oder einen Thread.

Der verborgene interne Ablauf eines RPC ist in Abbildung 2.4 genauer aufgegliedert. Alle gängigen RPC-Implementationen verwenden ein spezifisches Generatorprogramm, das Client-Stub, Server-Stub und zusammengefaßte Schnittstellen (*C header file*) für beide Stub-Arten generiert. Alle Stubs greifen auf eine Laufzeitbibliothek zurück, in der wiederverwendete Grundoperationen zusammengefaßt sind. Die Gestaltung der Stubs und ihrer Schnittstelle richtet sich nach einer Schnittstellenbeschreibungsdatei, die jeweils in einer eigenen Spezialsprache zu verfassen ist.

Im Fall des ONC-RPC heißt die Schnittstellenbeschreibungssprache *RPCL* (*Remote Procedure Call Language*) und das dazugehörige Stub-Generatorprogramm *rpcgen*. Abbildung 2.5 stellt die Rolle dieser Komponenten bei der Anwendungserstellung dar. Um in einem Programm Änderungen (Hinzufügen, Entfernen, Signaturmodifikationen) von RPCs vornehmen zu können, ist stets ein Editieren der RPCL-Datei und ein neuer Generierungsvorgang erforderlich. Daraufhin ist eine erneute Compilation der betroffenen Module und ein erneutes Linken des Programms erforderlich.

Gleiches gilt im Falle des DCE-RPC [Open Software Foundation 93], dessen Generierungsschema ebenfalls in Abbildung 2.5 ersichtlich ist. Das Generatorprogramm heißt hier *idl* und die Schnittstellenbeschreibungssprache *IDL* (*Interface Definition Language*). In einer separaten

[2] Sogenannte „asynchrone RPCs“ [Tanenbaum et al. 90] werden hier nicht betrachtet. Zum einen handelt es sich bei diesen eigentlich nicht um „echte“ RPCs, da sie die Einheitlichkeit von Funktionsaufrufen aufbrechen. Zum anderen lassen sie sich unter der im persistenten Objektsystem Tycoon gegebenen Voraussetzung des Multi-Threading problemlos aus synchronen RPCs oder Kanälen (siehe Abschnitt 5.2.2.1) zusammensetzen.

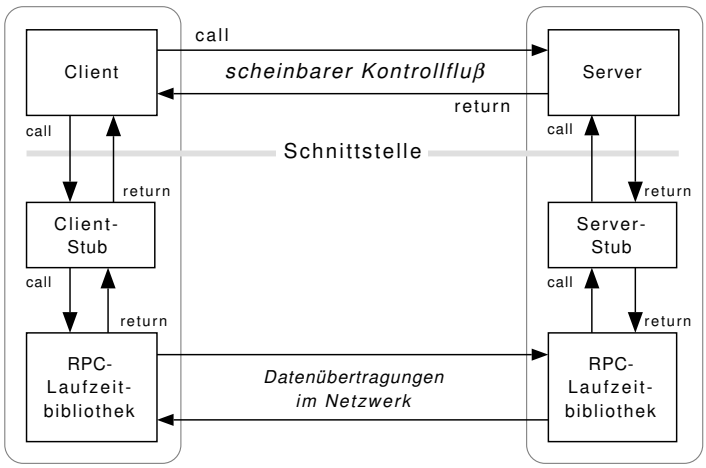

Abbildung 2.4: Das interne Ablaufschema eines RPC

Datei *ACF* sind Bindungsinformationen (RPC-Variante, Identifikationsnummern, Versionskennung) untergebracht.

Ferner ist es mit konventionellen RPC-Mechanismen umständlich, rekursive Datenstrukturen zu übertragen und es gibt keine Codemigration.

2.1.3 Externe Datenrepräsentationen und Marshalling

Der Vorgang, Objekte in einer zur Netzwerkübertragung geeigneten externen Repräsentationsform abzulegen, wird gemeinhin als *Marshalling* bezeichnet, die Rekonstruktion anhand einer solchen Datenrepräsentation als *Unmarshalling*. In diesem Zusammenhang sind folgende Anforderungen maßgeblich:

Serialisierung: Die Datenübertragung in Netzwerken ist immer seriell. Systemnahe Programmierschnittstellen zur Datenübertragung erwarten als maßgebliche Parameter zusammenhängende Hauptspeicherblöcke (*Puffer*), die sie als uninterpretierten Byte-Strom behandeln. Also bedeutet Marshalling, Byte-Ströme zu erzeugen.

Das Hauptproblem ist hierbei die Darstellung rekursiver Datenstrukturen, bzw. interner Objektreferenzen. Daher spricht man auch von *Linearisierung*.

Plattformunabhängigkeit: Je nach Hardware und Betriebssystem (manchmal auch Codegenerator) variieren maschinennahe interne Datenrepräsentationen wie z.B. die von Ganzzahlen (*Integers*) oder Fließkommazahlen. Es ist sicherzustellen, daß solche Daten beim Unmarshalling korrekt in die Zielumgebung „übersetzt“ werden.

Durch die Festlegung eines einheitlichen externen Formates für solche Werte, das beim Marshalling auf allen Plattformen erzeugt wird, läßt sich erreichen, daß jede Marshalling-Routine nur eine Variante berücksichtigen muß. Dann sind bestehende Anwendungen ohne jede Änderung zu neu hinzukommenden Plattformen kompatibel.

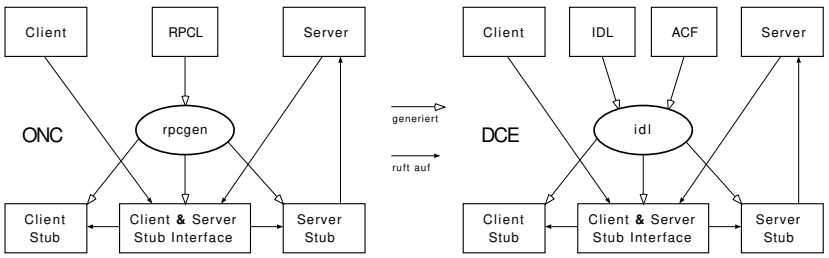

Abbildung 2.5: Generierungsschemata des ONC-RPC und des DCE-RPC

Plattformabhängige Datenformate treten auch bei BLOBs (*Binary Large Objects* wie z.B. Bildern, Bildsequenzen oder Tonfolgen) auf. Objektorientierte Sprachen (z.B. Modula-3 [Horning et al. 93]), sind durch *Overriding* dedizierter Methoden in der Lage, die Menge plattformunabhängiger Objekttypen (Klassen) ohne Modifikationen des Programmiersystems zu erweitern.

Die in Abschnitt 2.1.2 genannten RPC-Mechanismen verwenden als plattformunabhängige externe Repräsentationen zur Datenübertragung: XDR (*eXternal Data Representation*) [Corbin 91] für ONC-RPC, NDR (*Network Data Representation*) [Open Software Foundation 93] für DCE-RPC und ASN.1 [ISO-ASN 90] für ISO-RPC.

Beim Einsatz von XDR und NDR ist für jeden Datentyp eine spezifische Marshalling- und Unmarshalling-Routine erforderlich, da außer den eigentlichen Daten (fast) keine Metainformationen erfaßt werden. Im Gegensatz dazu erfaßt ASN.1 (optional) auch die Typen der repräsentierten Daten. Daher ist es möglich, eine *polymorphe* Unmarshalling-Funktion für ASN.1 zu schreiben. Um auch eine polymorphe Marshalling-Funktion zu realisieren, müßten (zumindest rudimentäre) Typinformationen auch zur Laufzeit bereitstehen. Dies ist jedoch in maschinennahen Sprache wie etwa $C/C{+}{+}$ nicht gegeben.

2.2 Verteilte Objektmanagementsysteme

Unter verteilten Objektmanagementsystemen (VOMS) werden in konventionellen Programmiersprachen (zumeist *C*/*C++*) implementierte Programmiersysteme verstanden, die entfernte Aufrufe von Objektmethoden implementieren. Dabei handelt es sich typischerweise um Kombinationen von Bibliothekserweiterungen und Stub-Generatoren.

Die bekannteste Systemarchitektur dieser Art ist CORBA[3] [OMG 91]. Sie unterscheidet sich vor allem durch folgende Eigenschaften von anderen Systemen:

Brokering: Entfernte Aufrufe gehen durch eine zusätzliche Systemschicht, den *Objekt Request Broker*, der die ausführenden Instanzen (Server, Thread, spezifisches entferntes Objekt) *dynamisch* zuordnet.

Offenheit: CORBA erlaubt durch Integration diverser konventioneller Programmiersprachen (z.B. *C*, *C++*, Fortran, COBOL, LISP, Smalltalk, Ada), eine Vielzahl bereits bestehender Anwendungen (*legacy systems*), in einem gemeinsamen verteilten System zusammenzubringen. Außerdem ist CORBA konzeptionell offen für unterschiedliche Implementationsplattformen.

CORBA-Kommunikationsschnittstellen werden in einer speziellen Sprache, der *Interface Description Language* (IDL) beschrieben. Sowohl die Syntax dieser Sprache als auch die von Genratorprogrammen dominierte Vorgehensweise zur Erstellung von Anwendungen entsprechen in groben Zügen den in Abschnitt 2.1.2 beschriebenen Gegebenheiten konventioneller RPC-Systeme. Dies betrifft auch die Probleme.

- ▷ Es werden Implementationssprachen eingesetzt, die für die verteilte Programmierung eher ungeeignet sind: *C*, *C++*, *Fortran*, etc. Die innerhalb dieser Sprachen erreichbare Mobilität ist aufgrund ihrer systemnahen Bindungsbegriffe recht gering.

- ▷ Zwischen CORBA-IDL und den gegebenen Implementationssprachen bestehen stets einige Unterschiede. Insbesondere ist es schwierig, Typsicherheit zu gewährleisten und gleichzeitig polymorph zu programmieren. Deshalb tendieren nicht nur das CORBA-Systeme selbst, sondern auch seine Anwendungen zum ad-hoc-Polymorphismus, was zu Generatorarchitekturen führt. In Bezug auf Erstellungsaufwand und Wartung bedeutet dies einen erheblichen systemimmanenten Nachteil, der sich bei wachsendem Anwendungsumfang noch verschärft.

- ▷ Die parallele Wartung von CORBA- und Implementationssprachen-Schnittstellen ist aufwendig und fehleranfällig.

- ▷ CORBA-IDL ist nicht algorithmisch. Sie dient ausschließlich zur Beschreibung von Schnittstellen. Dadurch ist eine Kluft zur Implementationssprache inklusive zusätzlichem Wartungsaufwand auch bei Systemneuentwicklungen unvermeidbar.

[3] Diese Architektur ist in einer ganzen Reihe von Implementationen verschiedener Hersteller, die untereinander in weiten Grenzen kompatibel sind, verfügbar. Detaillierte Analysen alternativer Objektmodelle und ihres Einsatzes im Rahmen von CORBA sind in [Nicol et al. 93; Manola, Heiler 93] enthalten.

- Auf der Server-Seite werden *statisch* generierte Stubs eingesetzt. Dadurch ist die Dynamik der Kommunikation eingeschränkt. Ferner sind die Konfiguration und die Wartung verteilter Programme verkompliziert.
- CORBA-IDL hat ein relativ ausdrucksschwaches Typsystem erster Ordnung. Strukturelle Typäquivalenz bezieht sich nur auf das Hinzufügen von Funktionen.

In CORBA sind einfache und rekursive Datentypen sowie CORBA-Objektreferenzen mobil. Code-Mobilität (bzw. die Mobilität von CORBA-Objekten) ist zunächst nicht gegeben.

Bedeutend weniger aufwendig als CORBA ist das VOMS *Network Objects* [Birell et al. 93b] für Modula-3 [Nelson 91], das auch in dieser Sprache implementiert ist. In der Einfachheit seiner Handhabung kommt dieses System originär verteilten Programmiersprachen (siehe Abschnitt 2.3) nahe. Es steht dabei in der Tradition von Emerald (vgl. Abschnitt 2.3.1):

- Es wird eine verteilte Garbage-Collection eingesetzt [Birell et al. 93a].
- Proxys[4] sind mobil.
- Durch das Marshalling (und Unmarshalling) werden Objekte beim Empfänger automatisch als Proxys repräsentiert.

Für letztere Bindungstechnik sind im Gegensatz zu Emerald keine Alternative vorhanden, da *Network Objects nicht* mobil sind.

In *DC++* [Schill, Mock 93] sowie einigen nicht auf Offenheit ausgerichteten VOMS wie z.B. SOS [Shapiro et al. 89; Shapiro, Ferreira 93] und Shadows [Caughey et al. 93; Caughey, Shrivastava 95], sind auch Objekte im Sinne der Objektorientierung mobil. In Shadows erstreckt sich die Mobilität sogar auf *Objektmanager*: lokale Instanzen des Shadows-Systems samt der von ihnen verwalteten Objekte.

Einige VOMS sind portabel (Shadows), bzw. relativ häufig auf verschiedenen Plattformen implementiert worden (*DC++*, CORBA). Die Kommunikation zwischen ihren jeweiligen Instanzen ist jedoch nur in VOMS ohne Objektmobilität vollkommen plattformunabhängig, da keine portablen Coderepräsentationen verwendet werden.

In den meisten Fällen wird Mobilität von einer bestimmten Klasse implementiert und weitervererbt, wodurch ihre orthogonale Geltung von vornherein ausgeschlossen ist.[5] Soweit bekannt, existiert kein VOMS mit einem *uniformen* Objektmodell, in dem die Mobilität vom Objekttyp unabhängig ist.

[4] Proxys sind transparente lokale Platzhalter entfernter Objekte (vgl. Abschnitt 5.1.1 und 2.3.1). In [Birell et al. 93b] werden sie auch als *Surrogate Objects* bezeichnet.

[5] Gleiches gilt übrigens für Persistenz.

2.3 Verteilte Programmiersprachen

In den folgenden Abschnitten werden Obliq, Emerald und Java als repräsentative Vertreter verteilter Programmiersprachen betrachtet. Diese Kombination weist alle im Zusammenhang der vorliegenden Arbeit wichtigen Merkmale auf, die heutzutage in dieser Systemklasse auftreten.

Weitere interessante verteilte Programmiersprachen sind z.B. Phantom [Courtney 95], Erlang [Armstrong et al. 93; Wikström 86] und Hermes [Strom et al. 91]. Eine Übersicht über zahlreiche ältere verteilte Programmiersprachen ist [Bal et al. 89] zu entnehmen.

2.3.1 Emerald

Emerald [Hutchinson 87b; Jul 88] ist eine objektorientierte Sprache, die ein *uniformes* Objektmodell für alle lokalen und entfernten Programmiersituationen bietet. Der wesentliche Kommunikationsmechanismus in Emerald besteht in entfernten Methodenaufrufe. Darüberhinaus kann die Verteilung von Objekten durch spezielle Kommandos (**move** ... **to** ..., **fix** ... **at** ..., **unfix**) explizit manipuliert werden.

Alle Emerald-Objekte sind mobil, wobei allerdings verschiedene Bindungstechniken zu unterscheiden sind [Jul et al. 88; Jul 88].

- Zustandslose Objekte (*immutable objects*) werden grundsätzlich kopiert.
- Veränderliche als Argument eines entfernten Aufrufes verwendete Objekte werden wahlweise durch Netzwerkreferenzen (s.u.) repräsentiert oder aber sie migrieren zum Aufrufer (*call-by-move*), wobei die Möglichkeit besteht, daß sie nach erfolgtem Aufruf automatisch zum Sender zurückkehren (*call-by-visit*).
- Objekte, die nur im lokalen Sichtbarkeitsbereich (der Methoden) eines migrierenden Objektes zugreifbar sind, begleiten dieses.
- Durch das Schlüsselwort **attach** können transitiv Cluster gemeinsam migrierender Objekte zusammengefügt werden. Allerdings bezieht sich **attach** auf (in Objekten gekapselte) Variablen und nicht direkt auf Objekte.
- Nicht mit **attach** versehene Komponenten kopierter oder migrierender Objekte bleiben jeweils auf dem sendenden Netzwerkknoten zurück. Auf dem Empfängerknoten werden sie für den Programmierer transparent durch neu erzeugte Netzwerkreferenzen (s.u.) repräsentiert. Dies ist in Abbildung 2.6 exemplarisch dargestellt: ein fiktives Objekt *a* migriert mit zwei **attach**-Objekten und hinterläßt dabei Objekt *d* und dessen transitive referentielle Hülle.

Netzwerkreferenzen sind Adreßwerte, die Objekte in entfernten Netzwerkknoten identifizieren. In Emerald bestehen Netzwerkreferenzen[6] aus 32-Bit-Maschinenworten. Davon verweisen 8 Bits auf einen Netzwerkknoten und die übrigen 24 Bits identifizieren das betreffende Objekt in

[6]Diese werden in [Jul 88] als „OIDs“ bezeichnet. In der vorliegenden Arbeit wird „Netzwerkreferenzen“ bevorzugt, um den Bezug auf entfernte Objekte hervorzuheben. Die Alternative „entfernte Referenzen“ könnte bei der Betrachtung anderer Systeme zu Verwechslungen mit lokalen Referenzen auf Proxys führen.[7]

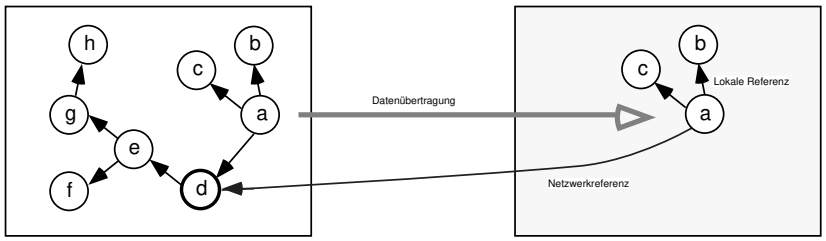

Abbildung 2.6: Transparente Erzeugung von Netzwerkreferenzen bei der Übertragung von Objekten

dessen Adreßraum. In einem nicht-experimentellen System wären natürlich mehr als 32 Bits erforderlich [Jul 88]. Eine Erweiterung führt jedoch dazu, daß Netzwerkreferenzen nicht mehr in ein Maschinenwort passen, wie dies bei *Network Objects* in Modula-3 der Fall ist (vgl. Abschnitt 2.2). In diesem System werden werden entfernte Objekte durch lokale Objekte repräsentiert: Proxys[7]. Eine vereinheitlichende Sicht wäre folgende: in Emerald sind „Proxys" durch *unboxed values* [Peyton Jones 87] und in Modula-3 durch *boxed values* repräsentiert.

Die transparente Erzeugung von Netzwerkreferenzen minimiert den Datentransfer während einer Übertragung. Außerdem bleibt die Referenzsemantik (*Sharing*) vormals lokaler Objektbeziehungen erhalten. Andererseits bewirken Lese-, Schreib- oder Ausführungszugriffe über Netzwerkreferenzen stets Netzwerkkommunikationen, bzw. entsprechende Versuche, die scheitern können. Daher kann es unter Umständen problematisch sein, wenn Netzwerkreferenzen *unkontrolliert* entstehen. Gerade dies läßt sich jedoch nicht verhindern, wenn Prozedurabstraktion, Objektabstraktion (in anderen Sprachen ggf. Modulabstraktion) oder abstrakte Datentypen eingesetzt werden.

Die **attach**-Anweisung ist nicht geeignet, die Erzeugung von Netzwerkreferenzen vollständig einzudämmen, da ihre Wirkung durch Zuweisungen unterlaufen werden kann. Außerdem würde eine durchgängige Verwendung von **attach** ein dahingehend verändertes Sprach-Design nahelegen, daß der Zusammenhang von Objekten der Normalfall und die Erzeugung von Netzwerkreferenzen die Ausnahme sei.

Aufgrund der geschilderten Umstände ergeben sich in Emerald-Anwendungen zahlreiche komplexe Interdependenzen zwischen Netzwerkknoten. Daher beschränkt sich das Einsatzgebiet von Emerald weitgehend auf Netzwerke mit geringer Latenz, hohen Übertragungsraten, und hoher Zuverlässigkeit.

Hinzu kommt, daß aufgrund der Inkaufnahme hoher Anzahlen auf unkontrollierte Weise erzeugter und im Netzwerk verteilter Netzwerkreferenzen eine verteilte Garbage-Collection unerläßlich ist. Das Funktionieren eines solchen globalen Prozesses hängt jedoch ebenfalls von den genannten Netzwerkeigenschaften ab. Emeralds Garbage-Collection ist jedenfalls nur für lokale Netzwerke konzipiert [Juul 93].

Alle genannten Einschränkungen gelten auch für verwandte Systeme wie z.B. die Sprachen Obliq [Cardelli 94] und Phantom [Courtney 95] sowie die Objektmanagementsysteme *Network*

[7] *Lokale* Referenzen auf Proxys heißen in [Birell et al. 93b] „entfernte Referenzen" (*remote references*).

Objects [Birell et al. 93b] und SOS [Shapiro, Ferreira 93]. Das Emerald-System ist darüberhinaus auf relativ eng gekoppelte Systeme angewiesen, indem es relativ häufig in für den Programmierer transparenter Weise Objekte bewegt. Es erfordert eine spezielle Anweisung (**fix** ...**at** ...), dies zu unterbinden.

Emerald-Objekte können optional einen eigenen Thread besitzen. Dieser Thread begleitet sie ggf. als fest verbundene Komponente bei allen Migrationen. Dies bedeutet, daß seine Ausführung jeweils auf dem Netzwerkknoten, auf dem sich das ihn besitzende Objekt gerade befindet, stattfindet. Also sind in Emerald im Gegensatz zu fast allen anderen Programmiersystemen[8] Threads mobil. Auf der Basis alternativer Maschinencodegenerierung wird sogar Thread-Mobilität in heterogenen Netzwerken erreicht [Steensgaard, Jul 95].

Die Implementation von Thread-Migrationen in Emerald [Jul et al. 88; Jul 88; Jul 89] basiert auf der Übertragung separater *Stack-Frames*, wodurch Thread-Stacks fragmentweise über mehrere Netzwerkknoten verteilt werden. Auch diese Lösung entspricht der Affinität von Emerald zu lokalen Netzwerken.

Emerald vermindert die Netzwerkbelastung durch dynamisches Linken von Code, der sich bereits auf dem Zielknoten einer Übertragung befindet.[9] Darüberhinaus werden Emerald-Objekte stets zunächst ohne ihren Code versendet. Wenn noch keine Kopie desselben vorhanden ist, wird er beim Sender automatisch nachgefordert.[10] Die Zuordnung zwischen Objekten und Code ist durch die Zugehörigkeit von Objekten zu *Object Constructors* gegeben. Objekte, die durch den selben *Object Constructor* entstanden sind, teilen sich dessen Code.

Es ist hervorzuheben, daß in Emerald auch Code der Garbage-Collection unterliegt, obwohl er zunächst aus Dateien geladen wird. Damit erreicht Emerald insgesamt betrachtet ein Bindungskonzept, das dem persistenter Objektsysteme nahesteht.

Emerald verfügt über einen experimentellen *Checkpoint*-Mechanismus [Hutchinson, Jeffery 89], der jedoch nicht auf Threads anwendbar ist [Larsen 92]. Außerdem ist der Checkpoint-Mechanismus von Emerald nicht sonderlich effizient, da er lineare Netzwerkrepräsentationen zur Speicherung verwendet (vgl. Abschnitt 1.1).

2.3.2 Obliq

Obliq [Cardelli 94] ist eine dynamisch typisierte, objektorientierte Sprache mit funktionalen und imperativen Konstrukten. Obliq unterstützt sowohl entfernte Methodenaufrufe als auch entfernte Funktionsaufrufe.

Unter Ausnutzung der Mobilität von Funktionsabschlüssen lassen sich auf einfache Weise zusätzliche verteilte Ausführungsmodelle realisieren.

- ▷ Entfernte Ausführung (*Remote Evaluation*, REV [Stamos, Gifford 90]) wird einfach durch eine entfernte Funktion höherer Ordnung modelliert, die Funktionsabschlüsse entgegennimmt und ausführt. Solche Funktionen werden in [Cardelli 94] als *Execution Engines* bezeichnet.

[8] So weit bekannt, liegen nur in Telescript (siehe Abschnitt 2.4.1) und TL vergleichbare Konstrukte vor [Mathiske et al. 95a].

[9] Ähnliche Techniken finden sich auch in einigen VOMS wie z.B. SOS [Shapiro et al. 89].

[10] Diese automatische Replikationstechnik wird in Abschnitt 7.5 verallgemeinert. Sie beschränkt sich in TL nicht auf Code.

- ▷ Durch verkettete Anwendung entfernter Ausführung können fortgesetzt migrierende Aktivitäten implementiert werden. Da Funktionsabschlüsse partielle Ausführungszustände kapseln, wird dabei gewissermaßen Thread-Migration simuliert [Mathiske et al. 95a]. Allerdings bringt dies einen recht unintuitiver Programmierstil[11] mit sich, bei dem sich die Programmstruktur an durch Migrationen vorgegebene Schachtelungen von Sichtbarkeitsbereichen anpassen muß. Dadurch müssen unter anderem häufig Iterationen durch Rekursion ausgedrückt werden. Außerdem ist zu beachten, daß Migrationen nur auf Veranlassung der Aktivität selbst hin und nur an vorbestimmten Stellen möglich sind.

Bei Netzwerkübertragungen wird zwischen Objekten und sonstigen Werten (Zahlen, Funktionen, Arrays, etc.) unterschieden. Während letztere kopiert werden, sind Objekte fest an ihren Ursprungsadreßraum gebunden. Lokale Objektreferenzen werden wie in Modula-3 transparent durch Proxys ersetzt. Tatsächlich ist Obliq in Modula-3 implementiert und basiert auf dem VOMS *Network Objects* (siehe Abschnitt 2.2).

Zusätzlich tritt in Obliq eine eingeschränkte Form der Objektmobilität auf: durch die **copy**-Anweisung können *flache* lokale Kopien entfernter Objekte erzeugt werden. Dies genügt jedoch nicht zur Vermeidung der transparenten Proxy-Erzeugung, da ständiges Ausprogrammieren tiefer Kopien in der Praxis nicht zu leisten ist. Insgesamt ist Obliq also eine weitere Sprache, deren Einsatzgebiet weitgehend auf lokale Netzwerke beschränkt ist.

Obliq bietet zwar Multi-Threading, doch Obliq-Threads sind nicht mobil.

2.3.3 Java

Die Kommunikationsprogrammierung in der objektorientierten Sprache Java [Sun 95a; Sun 95b] gestaltet sich knapp oberhalb der Transportebene: es stehen zunächst nur Sockets sowie das HTTP-Protokoll zur Verfügung [Weiss et al. 96a]. Daher ist Java noch nicht als vollwertige verteilte Programmiersprache anzusehen. Doch es ist in naher Zukunft damit zu rechnen, daß Java zu einer solchen weiterentwickelt wird, indem höherwertige Kommunikationsmechanismen wie z.B. entfernte Methodenaufrufe implementiert werden. Dieses Ziel wird bereits in einigen Projekten verfolgt, wobei insbesondere auch die Möglichkeiten einer Integration mit CORBA exploriert werden [Weiss et al. 96a].

Gegenwärtig wird Java hauptsächlich zur Gestaltung graphischer Benutzeroberflächen im WWW eingesetzt. Dabei werden nicht nur statische Elemente wie z.B. Texte, Eingabeformulare und Bilder sondern auch Animationen in Form sogenannter *Applets* jeweils von einem WWW-Server in einen WWW-Browser geladen und von einem dort residenten, *Applet-Viewer* genannten, Java-Programm zur Ausführung gebracht werden [Weiss et al. 96b].

Ein Applet besteht aus dem Objektinitialisierungscode einer Java-Klasse, die eine spezifische, an Applet-Viewer angepaßte, Schnittstelle implementiert. Der durch Applets erreichte Effekt entspricht trotz einer relativ großen Ansammlung nicht-trivialer Sprachkonzepte, die hier zusammenwirken, zunächst vergleichsweise simplen Prozedurübertragungen. Doch selbst in diesem eingeschränkten Anwendungsbereich treten bereits Vorteile der Java-Architektur, die auf einer virtuellen Maschine (Byte-Code-Interpreter) basiert [Sun 95c], deutlich in den Vordergrund:

[11] Strukturell handelt es sich hierbei um eine Variante des in der Tycoon-Zwischensprache TML eingesetzten *Continuation Passing Style* [Gawecki, Matthes 96].

Plattformunabhängigkeit: Das Java-System ist nicht nur sehr portabel, sondern es erlaubt auch den plattformunabhängigen Daten- und Codeaustausch zwischen verschiedenen Java-Portierungen. Es werden sowohl WWW-Clients als auch -Server verschiedener Hersteller auf einer Reihe unterschiedlicher Hardware- und Betriebssystemplattformen unterstützt.

Datensicherheit: In Applet-Viewers sind alle Interpreteranweisungen zur Ausführung „gefährlicher" Operationen, wie z.B. *C*-Calls und Dateizugriffe deaktiviert. Ein vorgeschalteter *Verifyer* analysiert empfangenen Byte-Code vor seiner Ausführung auf unzulässige Operationen. Da dieses Thema in der vorliegenden Arbeit nicht vertieft wird, sei auf [Yellin 95; Rudloff 96] verwiesen.

Alle primitiven Datentypen (Zahlen, Zeichenketten, etc.) sowie Code sind in Java mobil, Java-Objekte und Threads jedoch nicht.

In Java-Programmen können Klassen (bzw. deren Code) sowohl aus einem lokalen Dateisystem als auch in Folge einer Netzwerkübertragung dynamisch gelinkt werden. Diese Technik ist zwar in ähnlicher Form bereits in Emerald vorhanden (vgl. Abschnitt 2.3.1), doch vor dem Hintergrund einer kopierbasierten Übertragungssemantik eröffnet Java die Möglichkeit, sie auch in weit verteilten Netzen einzusetzen.[12]

[12] Damit wird ein parallel entwickelter [Mathiske et al. 95a], entsprechender Ansatz (vgl. Abschnitt 7.4 und 7.5) der vorliegenden Arbeit bestätigt.

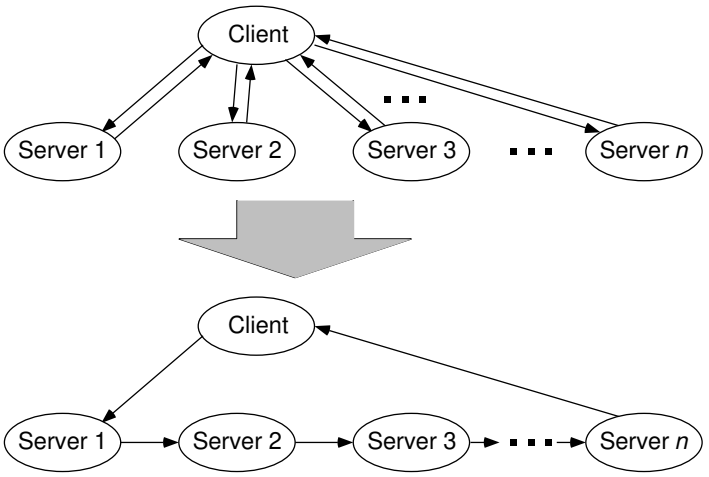

Abbildung 2.7: Ersetzen von Client/Server-Anfragen durch eine Agentenreise

2.4 Sprachen für mobile Agenten

Verteilte Softwaresysteme werden heutzutage vom Client/Server-Schema dominiert. In diesem Schema kommunizieren Paare residenter Softwarekomponenten über eine Netzwerkverbindung miteinander. Dazu müssen Client und Server permanent *online* sein und unter Umständen umfangreiche Datenströme austauschen, die Kommandos, Eingabe- und Ausgabedaten umfassen. Häufig machen die Daten, die für den Benutzer von Interesse und notwendig sind, um eine Anfrage zu spezifizieren, nur einen Bruchteil dieses Umfangs aus. Der Rest belastet nur das Netzwerk und konsumiert die Zeit des Benutzers, sofern dieser unmittelbar in die Manipulation der Daten involviert ist.

Mobile Agenten haben das Potential, Verbindungskosten erheblich zu reduzieren. Zum Beispiel könnten von einem Client ausgehende Anfragen an eine Liste von Servern in vielen Fällen durch eine Agenten-„Reise“ ersetzt werden. Diese Umstrukturierung ist in Abbildung 2.7 schematisch dargestellt. Das Client/Server-Modell erfordert hier rund doppelt so viele Datenübertragungen wie das Agentenmodell.

Im Gegensatz zu den anderen gängigen Agentenbegriffen[13], auf die in der vorliegenden Arbeit nicht weiter eingegangen wird, handelt es sich bei mobilen Agenten [Wayner 94; Reinhardt 94; White 94; Thomsen et al. 95; Knabe 95] um nicht mehr und nicht weniger als eine Softwaretechnik bzw. -architektur für verteilte Systeme, die weder an einen bestimmten Anwendungsbereich noch an einen bestimmten Interaktionsstil gebunden ist. Das Konzept mobiler Agenten befreit den Benutzer allerdings vom *Zwang* der direkten Manipulation. Es hebt auch die Notwendigkeit auf, dauernd eine Datenverbindung zu unterhalten, indem es vorsieht, daß Agenten direkt an den Ort der Datenquelle, die sie in Anspruch nehmen, wandern.

Agenten sind mächtiger als Java-Applets (vgl. Abschnitt 2.3.3). In beiden Fällen wandern Programme durchs Netz, und beide Male geht es darum, die Netzbelastung zu reduzieren.

[13] Eine differenzierende Übersicht über *Softbots*, Interface-Agenten, und diverse der KI zugerechnete Ansätze bietet [Wooldridge, Jennings 95].

Doch während Applets ausschließlich von einem Server zu einem Client wandern, migrieren Agenten von einem Client zu einem Server, zwischen Servern und ggf. schließlich von einem Server zu einem Client zurück. Darüberhinaus sind von Servern initiierte Agenten denkbar (vgl. Abschnitt 8.1).

Die Realisierung all dessen setzt eine Infrastruktur voraus, die heute in heutigen WANs nicht vorhanden ist. Es ist ein spezielles Protokoll auf der Anwendungsebene und dessen Implementierung durch ubiquitäre Server, die Agenten empfangen und beherbergen, erforderlich.

In den folgenden beiden Abschnitten werden die beiden in repräsentativer Weise mit Konstrukten zur Programmierung mobiler Agenten ausgestatteten Sprachen Telescript und FACILE betrachtet. Weitere Sprachen, die zu diesem Zweck eingesetzt werden, sind z.B. *MØ* [Tschudin 94; Tschudin 95] und Agent Tcl [Gray 95].

2.4.1 Telescript

Die objektorientierte Programmiersprache *Telescript* [Wayner 94; White 94; General Magic 95a; General Magic 95b] ist von vornherein speziell für mobile Agenten konzeptioniert worden. Ihre wichtigsten Abstraktionen sind Plätze (*Places*), Agenten (*Agents*), Reisen (*Travels*), Treffen (*Meetings*), Verbindungen (*Connections*), Besitzer (*Authorities*) und Verfügungsrechte (*Permits*).

Plätze sind logische Netzknoten, die von einem residenten Prozeß erbrachte Dienste anbieten. Agenten sind mobile Prozesse, die zwischen Plätzen migrieren, um im Auftrag ihrer Besitzer vorprogrammierte Aufgaben zu erfüllen. Ein Telescript-Prozeß umfaßt eine Prozedur und einen Ausführungszustand. Die vordefinierte Methode *go* dient zur Bewegung von einem Platz zum anderen. Sie verlangt ein sogenanntes *Ticket* als Parameter. Diese „Fahrkarte“ spezifiziert das Ziel, den Weg und den Zeitrahmen einer Reise.

Agenten, die sich an unterschiedlichen Plätzen befinden, können entfernte Kommunikationsverbindungen nur zu dem Zwecke aufnehmen, um die Bedingungen eines Treffens (*Meeting*) an einem gemeinsamen Platz untereinander auszuhandeln. Anderweitige Kommunikation kann nur während eines Meetings stattfinden.

Die Bindungskonzepte in Telescript begünstigen das Auftreten von Laufzeitfehlern:

- Während Klassen bei Migrationen je nach Anwendungssituation kopiert oder dynamisch gelinkt werden, ist für die Übertragung von Objekten entscheidend, ob sie einem migrierenden Agenten „gehören“. Lokale Kopien von Objekten, die ein migrierender Agent zunächst hinterlassen hat, werden vom System automatisch gelöscht. Lokale Referenzen auf solche Objekte sind dann ungültig (*dangling*) und erzeugen bei ihrer (versehentlichen) Verwendung Laufzeitfehler.

- Objekte und Klassen werden durch globale Namen identifiziert. Es sind ohne weiteres Duplikate, Versionskonflikte und vollkommen fehlerhafte Zuordnungen möglich. Insbesondere ist das dynamische Linken von Klassen durch migrierende Agenten typunsicher.

- Agenten lösen erst nach ihrer Ankunft an Plätzen Ausnahmen aus, wenn sie eine Klasse nicht vorfinden. Genauer gesagt geschieht dies sogar erst dann, wenn auf Methoden der Klasse zugegriffen wird.

Die Integration externer Dienste ist in der Sprache Telescript nur über den Weg der Socket-Kommunikation möglich [General Magic 95b].

2.4.2 FACILE

Das ML-Derivat FACILE ist in der Dissertation von Knabe um Kommunikations- und Bindungsmechanismen erweitert worden, die es zu einer Agentensprache machen sollen [Knabe 95]. Gleichzeitig liefert diese Arbeit einen Überblick jüngerer Entwicklungen in den Bereichen portable Codemobilität und mobile Agenten.

Knabe postuliert einige essentielle Eigenschaften von Agenten, die im wesentlichen die in [Mathiske et al. 95a] vertretenen Auffassungen im Hinblick auf den speziellen Kontext von Agenten bestätigen. Vor diesem Hintergrund ist parallel zur vorliegenden Arbeit (vgl. [Mathiske et al. 95a; Mathiske et al. 95b], Abschnitt 7.4) auch in FACILE ein Mechanismus zum dynamischen Linken ubiquitärer Objekte entstanden. Allerdings ist dynamisches Linken in FACILE auf Module eingeschränkt.

Dynamisches Linken unterscheidet sich von statischem Linken, indem es zur Laufzeit scheitern kann, wenn ein zu linkendes Objekt nicht vorliegt. In FACILE stellt ggf. der *Empfänger* eines Objektes fest, daß ein dynamischer Linkvorgang scheitert und muß dann eine Fehlerbehandlung durchführen. In der vorliegenden Arbeit wird hingegen die Auffassung vertreten, daß die Zuständigkeit für den Ablauf eines Agenten auch in Fehlerfällen möglichst bei ihm selbst liegen sollte. Es erscheint deshalb vorteilhafter, eine Migration ggf. vollständig abzubrechen und im Kontext des Agenten noch auf der *Senderseite* eine Ausnahme auszulösen.

Damit ist auch die von Knabe vorgeschlagene Alternative obsolet, einer Agentenmigrationen im Zweifelsfall einen *trial agent* oder einen *inspection agent* voranzuschicken, um die Empfangsumgebung auf vorhandene Objekte zu testen, bzw. zu untersuchen. Diese Variante würde den Vorteil der *Transparenz* dynamischen *Linkens* zunichte machen, indem Agenten über zu linkende Objekte Buch führen müßten, was vom Aufwand her der weiteren Alternative expliziten dynamischen *Bindens* gleichkommt.

FACILEs grundlegender Kommunikationsmechanismus sind Kanäle, die den Austausch nahezu beliebiger FACILE-Objekte zwischen Threads[14] erlauben. Insbesondere können Funktionsabschlüsse übertragen werden. Allerdings müssen sie dazu besonders annotiert werden und es ist stets speziell darauf zu achten, ob Funktionen übertragbar sind oder nicht.

Aktivierte Threads sind in FACILE nicht mobil. Skripte von Agenten, die mehrmals migrieren sollen, müssen daher aus mehreren Funktionen zusammengesetzt werden. Dadurch ist der Kontrollfluß häufig unübersichtlich (vgl. Abschnitt 2.3.2).

[14] Da FACILE konzeptionell auf einer Prozeß-Algebra aufbaut, werden Threads im engeren Kontext dieser Sprache als „Prozesse“ bezeichnet.

2.5 Persistente Objektsysteme

Bisher gibt es nur wenige Studien, wie Verteilung und persistente Programmiersprachen kombiniert werden können.[15]

Aus PS-algol [Atkinson et al. 81], einer persistenten Variante von Algol68, ist die verteilte Programmiersprache DPS-algol (*Distributed Persistent Algol*) [Wai 88; Wai 89] hervorgegangen. Dabei wurde zwar der Versuch unternommen, dem Programmierer die Illusion eines nicht-verteilten Objektspeichers zu bieten, doch im Ergebnis bietet DPS-algol auch explizite Mobilität. Die Konstrukte **transcopy** und **assign** erlauben den Austausch von Objekte zwischen verteilten persistenten Objektspeichern. Die Semantik des sogenannten *Transcopying* hängt vom Typ des jeweils zu kopierenden Objektes ab. Flache Kopien werden gegenüber tiefen bevorzugt, um die zu übertragenden Datenvolumina klein zu halten. Dabei entstehen auf für den Programmierer transparente Weise entfernte Objektreferenzen, was die Ausbreitung von Objektgraphen auf mehrere Objektspeicher zur Folge hat.

Bisher wurde noch keine allgemeine Antwort auf die Frage gefunden, wie mehrere persistente Objektspeicher, die aufgrund transparent entstandener entfernter Objektreferenzen in undurchschaubarer Weise voneinander abhängen, konsistent gehalten, bzw. wie Fehler und Ausfälle in solchen Systemen (transparent) überwunden werden können [Dearle et al. 91; Mira da Silva 95a; Atkinson, Morrison 95].

Daher konzentrieren sich mit Napier88 realisierte Nachfolgearbeiten zu DPS-algol auf tiefe Kopieroperationen zwischen Objektspeichern, die keine Referenzen erzeugen. Munro [Munro 93] entwickelte eine Kommunikationsschnittstelle für den datenstromorientierten Datenaustausch zwischen Objektspeichern. Als „Übertragungsmedium“ werden Sockets [Corbin 91] verwendet. Munro weist darauf hin, daß eine Kombination seines Mechanismus mit einem Zwei-Phasen-Commit-Protokoll [Gray 78] eine Grundlage für höherwertige Programmierabstraktionen bietet.

Um Datenübertragung und entfernte Ausführung zu kombinieren, wurde für Napier88 ein entfernter Ausführungsmechanismus konzipiert [Dearle et al. 91][16], der auf der Mobilität von Funktionsabschlüssen basiert. Um Bindungen an lokale Objekte eines entfernten Objektspeichers zu erlangen, müssen dorthin übertragene Funktionen jeweils explizit eine dynamische Suche in dessen globalem Sichtbarkeitsbereich durchführen (*environment lookup*). Dabei sind dann dynamische Typüberprüfungen erforderlich, weshalb es vorkommen kann, daß die Ausführung einer Anfrage auch nach der Übertragung an den Zielort aufgrund von Typinkompatibilitäten abgebrochen wird.

Hier wären Mechanismen wünschenswert, die zumindest am Übertragungsziel statische Typsicherheit garantieren. Dadurch würde sich das Spektrum möglicher Fehlerquellen und infolgedessen der notwendige Aufwand zur Fehlerbehandlung erheblich verringern. Dies ist eine wichtige Motivation für die Entwicklung der *typsicheren* entfernten Funktionsaufrufe im Tycoon-System, die in Kapitel 5 beschrieben werden.

Auch für Napier88 sind inzwischen typsichere entfernte Funktionsaufrufe entstanden. Diese jüngsten Entwicklungen des Napier-Systems [Mira da Silva 95a; Mira da Silva 95b] basieren

[15] Einen Überblick über frühe Bemühungen in dieser Richtung bietet [Dearle et al. 91].

[16] Als Vorbild diente REV [Stamos, Gifford 90]. Weder dieser Mechanismus noch die Napier-Variante wurden jedoch implementiert. Ein tatsächlich implementierter, sehr ähnlicher Mechanismus findet sich in Obliq [Cardelli 94] in Form von *Execution Engines* (vgl. Abschnitt 2.3.2)).

auf Anregungen durch im Rahmen der vorliegenden Dissertation geleisteter Arbeiten (u.a. [Matthes et al. 94; Mathiske et al. 95a]) am Tycoon-System.[17] Sie erreichen allerdings im Hinblick auf Mobilität, Flexibilität sowie sprachlicher und systemtechnischer Effizienz ihr in Kapitel 5 beschriebenes Vorbild bei weitem nicht:

- ▷ Entfernte Funktionen haben nur genau ein Argument.
- ▷ Das Multi-Threading von Clients und Servers ist während der gesamten Dauer eines RPC außer Kraft gesetzt. Dadurch entstehen häufig Deadlocks.
- ▷ Es gibt diverse Einschränkungen bezüglich der Signatur entfernter Funktionen. Insbesondere werden Typparameter und Threads nicht unterstützt.
- ▷ Es gibt keine Alternativen zum rekursiven Kopieren der gesamten transitiven referentiellen Hülle von RPC-Argumenten.
- ▷ Die Performanz ist äußerst gering [Mira da Silva 96]: ein RPC benötigt mindestens 2 Sekunden und bereits bei wenigen Tausenden zu übertragenden Objekten beträgt die Dauer mehrere Minuten (vgl. hierzu Anhang F).

Napier88-RPCs werden mit Hilfe von Quelltext-Reflektion [Kirby 93] dynamisch generiert. Daß die Generierung zur Laufzeit stattfindet, ist zwar im Vergleich zu verteilten Objektmanagementsystemen (siehe Abschnitt 2.2) ein Fortschritt, doch wäre es im Hinblick auf Effizienz und die Vermeidung von Systemfehlern erheblich vorteilhafter, hier ganz auf die Einbeziehung der Quelltextebene zu verzichten, wie dies bei originären verteilten Programmiersprachen der Fall ist.

[17] Eine entsprechende Bezugnahme ist in *Silv95a* explizit angegeben.

2.6 Zusammenfassung

Während Sockets, ONC-RPC und DCE-RPC in *C* verfügbare, grundlegende Kommunikationsmechanismen sind, die sich hauptsächlich für die Systemprogrammierung eignen, sind die weiteren in diesem Kapitel beschriebenen Systeme für die Anwendungsprogrammierung bestimmt.

Das Bestreben, eine sprachliche Offenheit an der Kommunikationsschnittstelle zu erreichen, konzentriert sich auf solche Sprachen, die nicht besonders für die Kommunikationsprogrammierung geeignet sind: *C*, *C++*, Fortran, COBOL, etc. Von vornherein kommunikationsorientierte Sprachen verwenden dagegen stets eine eigene spezielle externe Repräsentation, die nicht zu anderen Sprachen kompatibel ist. Ein hinreichender Grund für diese Entwurfsentscheidung ist, daß spezifische Leistungen in Bezug auf Typsysteme und Bindungstechniken verloren gehen, wenn sie auf einen gemeinsamen Nenner gebracht werden. Außerdem erreichen dediziert auf Verteilung ausgerichtete Programmiersprachen eine weitaus höhere sprachliche Homogenität und in Folge dessen eine höhere Generizität als verteilte Objektmanagementsysteme. Dies gilt insbesondere für die Mobilität. Andererseits besteht bei verteilten Programmiersprachen meist eine mangelnde Offenheit für externe Dienste.

CORBA ist ein Vorbild für die Offenheit verteilter Programmiersysteme. Doch es erscheint nicht zwingend notwendig, wie CORBA Schnittstellenbeschreibungssprachen und Implementationssprachen in *jedem* Anwendungsfall zu trennen. Es wird vielmehr davon ausgegangen, daß durch die Verwendung eines einheitlichen sprachlichen Rahmens, der über weitere Strecken verteilter Anwendungen trägt, systembedingte Brüche in vielen Fällen überwunden werden können.

In Sprachen höherer Ordnung wie Obliq können Aktivitätsmigrationen, wenn auch auf umständliche Weise, durch kopierbasierte Übertragungen von Funktionsabschlüssen implementiert werden. Eine elegantere und flexiblere Möglichkeit besteht in der in Emerald gegebenen Mobilität von Objekten, die Threads enthalten. Diese ist jedoch aufgrund der Verteilung von Ausführungskontexten und häufiger impliziter Erzeugung von Proxys nur in sehr zuverlässigen Netzwerken einsetzbar. Um die Vorteile beider Ansätze zu kombinieren, sind Threads primär kopierbasiert zu übertragen.

Die Agentensysteme Telescript und FACILE machen autonome migrierende Aktivitäten zum Hauptparadigma verteilter Programmierung. So weit soll in der vorliegenden Arbeit nicht gegangen werden. Es wird zwar anerkannt, daß eine elaborierte Unterstützung des Agentenstils erforderlich ist, doch diese sollte unbedingt mit Client/Server-Elementen ergänzt sein - oder umgekehrt. Es soll möglich sein, je nach Anwendungsbedarf den einen oder den anderen Stil zu verwenden sowie beide zu vermischen. Letzteres ist zum Beispiel bei Agenten geboten, die Zwischenergebnisse einer längeren Reise nicht ständig mit sich führen, sondern an eine vorgegebene entfernte Instanz liefern sollen. Ein weiterer Mischfall ist die Fernkontrolle, bzw. -steuerung von Agenten oder Workflows.

Während VOMS und Sprachen wie Emerald und Obliq darauf ausgerichtet sind, die Konzepte Objektidentität und *Sharing* auf im Netzwerk verteilte Objekte zu übertragen, steht bei Java und Agentensprachen im Vordergrund, Ausführungseinheiten zu kopieren, die zum größten Teil von Netzwerkverbindungen unabhängig sind.

Zusammenfassend wird festgestellt, daß folgende Merkmale existierender Systeme besonders attraktiv sind: die Integration externer Dienste nach dem Vorbild eines offenen VOMS und

die sprachliche Homogenität verteilter Programmiersprachen und die Modellierung autonomer migrierender Aktivitäten nach Art mobiler Agenten. Unter den vorgefundenen Bindungstechniken erscheinen dynamisches Linken und automatische Replikation besonders vorteilhaft.

Diese Techniken gilt es nun mit orthogonaler Persistenz in Verbindung zu bringen. Aus der Betrachtung persistenter Objektsysteme in Abschnitt 2.5 geht hervor, daß die vorliegende Dissertation bezüglich der genannten Merkmale sowie der Mobilität *in persistenten Objektsystemen* zum großen Teil auf bisher unbearbeitetes Gebiet vorstößt.

Kapitel 3

Das persistente Objektsystem Tycoon

Das Tycoon-Projekt [Tycoon 92] hat eine substantiell verbesserte Produktivität der Datenbankprogrammierung unter Ausnutzung generischer Dienste zum Ziel, wobei zunächst folgende Aspekte im Vordergrund standen [Matthes 93]:

- Die Entwicklung und formale Definition eines hochgenerischen Sprachkerns, der die Benennung, Bindung und Typisierung von Objekten und Diensten in datenintensiven Anwendungen in einer Art und Weise vorsieht, die nicht an ein bestimmtes Datenmodell gebunden ist.
- Die Erweiterung dieses Sprachkerns zu einer voll ausgestatteten, streng-typisierten polymorphen Programmiersprache (TL, *Tycoon Language* [Matthes, Schmidt 92]), die auch spezielle Sprachkonzepte für die Integration, Erweiterung, Anpassung und Benutzung generischer Datenbankendienste umfaßt. TL ist eine Weiterentwicklung von Quest [Cardelli 89] mit dem Zwischenschritt P-Quest [Matthes et al. 92a].
- Die Definition der Daten- und Programmrepräsentationen und der Evaluationssemantik einer untypisierten Zwischensprache (TML, *Tycoon Machine Language* [Gawecki, Matthes 94]), die nicht nur eine effiziente Zielcodegenerierung, sondern auch dynamische Optimierungen unterstützt. Zur Zeit werden nur klassische Programmoptimierungen vorgenommen. Es gibt jedoch bereits sehr vielversprechende Ansätze, Programmoptimierungen mit effizienten Techniken der Datenbankanfrageoptimierung zu vereinen [Gawecki, Matthes 95a].

Das Tycoon-Objektsystem zeichnet sich gegenüber anderen persistenten Objektsystemen [Brown et al. 92; Bancilhon et al. 92; Atkinson 95] durch eine besonders offene, skalierbare und modifizierbare Schichtenarchitektur aus [Matthes et al. 95b], die im nachfolgenden Abschnitt vorgestellt wird. Dabei werden bereits wichtige Implikationen bestimmter Systemkomponenten für die Realisierung von Objektmobilität deutlich.

Abschnitt 3.2 beschreibt die für diese Arbeit wesentlichen Konzepte und Konstrukte der Programmiersprache TL. Bei dieser Gelegenheit wird die im folgenden verwendete Notation für Programmbeispiele eingeführt.

Nach einer Beschreibung des TSP in Abschnitt 3.3, wird auf die uniformen Objektspeicherrepräsentation von TL-Werten eingegangen (Abschnitt 3.4), deren Gestaltung entscheidend durch die Anforderungen der Polymorphie und der impliziten orthogonalen Persistenz geprägt ist. Die in diesem Zusammenhang getroffenen Design-Entscheidungen erweisen sich auch für die Realisierung von Mobilität als günstig.

Zu guter Letzt wird in Abschnitt 3.5 die bidirektionale transparente Programmierschnittstelle zwischen TL und *C* vorgestellt.

3.1 Die Schichtenarchitektur des Tycoon-Systems

Ein Hauptbeitrag für die Skalierbarkeit und Modifizierbarkeit des Tycoon-Systems besteht in einer strikten Trennung der Aufgaben der Datenmodellierung, Datenmanipulation und Datenspeicherung in verschiedenen Systemschichten. Für jede Systemschicht stehen alternative Implementationen mit unterschiedlichen operativen Qualitäten zur Verfügung. So reicht das Spektrum möglicher Tycoon-Konfigurationen von einer isolierten Hauptspeicherimplementation auf einem PC bis hin zu einem großen, vernetzten, mehrbenutzerfähigen, optimierenden und persistenten Tycoon-Objektsystem.

In Abbildung 3.1 sind die aufeinander aufbauenden Schichten des Tycoon-Systems ersichtlich: der persistente Objektspeicher, das Laufzeitsystem, der Codegenerator, das Compiler-Backend, das Compiler-Frontend und die Standard-Bibliotheken. Jeweils zwischen zweien dieser Komponenten ist eine dedizierte Repräsentationsform angesiedelt.

Tl(*Tycoon Language*): eine Programmiersprache für Benutzer, d.h. Programmierer. In dieser Sprache werden alle Tycoon-Bibliotheken und Tycoon-Anwendungen geschrieben.

Abstrakte Syntaxbäume: das Ergebnis des Compiler-Frontends ist gleichzeitig die Eingabe für das Backend.

Tml(*Tycoon Machine Language*): die vom Compiler-Backend erzeugte und (auf Anweisung) optimierte Zwischensprache. Sie ist strukturell bereits nahe am Zielcode (Byte-Code, siehe nächster Punkt) orientiert, aber nicht ausführbar.

Literale: das vom Codegenerator hervorgebrachte Endergebnis von Compilationsvorgängen. Als Literale werden alle TL-Werte bezeichnet, die zur Compilationszeit erzeugt werden. Dabei handelt es sich hauptsächlich um Byte-Code, aber auch um sonstige in TL-Quelltext vorgegebene Konstanten wie z.B. Zeichenketten sowie um Debugging-Information. Alle Literale werden im persistenten Objektspeicher angelegt.

Das Laufzeitsystem kommuniziert mit dem persistenten Objektspeicher ausschließlich über die in *C* realisierte Softwareschnittstelle *Tycoon Store Protocol* (TSP), die in Abschnitt 3.3 näher beschrieben ist.

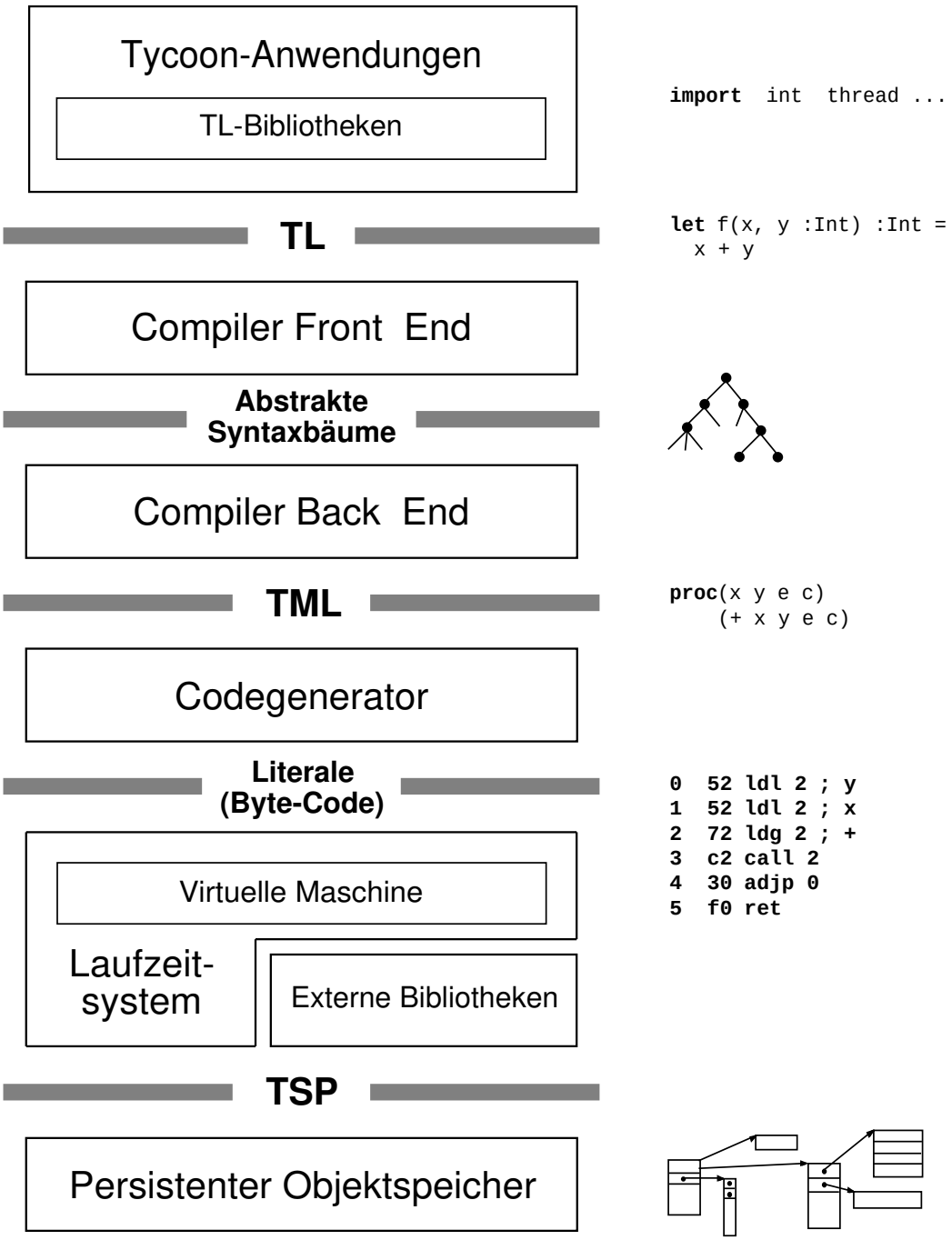

Abbildung 3.1: Die Schichtenarchitektur des Tycoon-Systems

3.2 Die Programmiersprache Tycoon Language

Zur Definition neuer und zur Einbindung bestehender externer Dienste wird die algorithmisch vollständige Programmiersprache TL [Matthes 93] verwendet. Sie ermöglicht eine flexible Benennung, Bindung und Typisierung der für datenintensive Anwendungen relevanten Objekte und Dienste. TL wird im Tycoon-System nicht nur zur Datenmodellierung und Applikationsprogrammierung eingesetzt, sondern sie ist auch die Systemprogrammiersprache, in der die frei erweiterbaren Tycoon-Bibliotheken und die Tycoon-Sprachprozessoren implementiert sind.

Die folgende (z.T. aus [Mathiske 92] übernommene) Einführung in die Programmiersprache TL konzentriert sich auf im Rahmen der vorliegenden Arbeit relevante Aspekte. Eine ausführliche Einführung bietet [Matthes et al. 94]. Detaillierte Beschreibungen der Sprache TL sind [Matthes, Schmidt 92] und Kapitel 4 in [Matthes 93] zu entnehmen. Ferner enthält [Mathiske et al. 93] eine Anleitung zur praktische Benutzung der interaktiven Systemumgebung und der Standard-Bibliotheken.

3.2.1 Wert- und Typausdrücke

TL (*Tycoon Language*) ist eine polymorphe funktionale Programmiersprache höherer Ordnung mit imperativen Eigenschaften und induktiv definierten, strukturellen Subtypisierungsregeln über Typen und Typoperatoren. Ihr Sprachkern ist erweitert um Sprachkonstrukte, die durch die Anforderungen datenintensiver Anwendungen begründet sind [Matthes, Schmidt 91].

In TL gilt eine strikte Evaluationssemantik mit streng deterministischer Evaluationsreihenfolge entsprechend der Lesart der lexikalischen Notation: „von links nach rechts“.

In TL ist weder eine automatische Wertkonvertierung noch ein Überladen von symbolischen oder alphanumerischen Bezeichnern (wie z.B. in Ada) möglich.

Folgende Basistypen sind vordefiniert: *Int*, *Real*, *Char* und *String*. Die Angabe von Werten der TL-Basistypen (s.o.) basiert auf Symbolproduktionen zur Definition von Programmliteralen. Beispiele:

```
false  true      (* Die beiden Wahrheitswerte des Typs Bool *)
42               (* Eine Ganzzahl vom Typ Int *)
3.14             (* Eine Fließkommazahl vom Typ Real *)
"Hello World!"   (* Eine Zeichenkette vom Typ String *)
```

Programmkommentare haben die Form (*...*).

TL besitzt eine LL(1) Grammatik mit einleitenden Schlüsselworten und schließendem **end**. Die wichtigsten Schlüsselworte sind **let** zur Einleitung von Wertbindungen und **Let** für Typbindungen.

```
Let T = Tupel x :Int end

let a = 7
let t = tuple a end
```

Die Konstanten a und *t* werden durch obige Anweisungen jeweils gleichzeitig deklariert, initialisiert und benannt. Die Veränderbarkeit einer Variablen wird in TL durch das Schlüsselwort **var** ausgedrückt.

```
let var x = 4710        (* x ist veränderbar. *)
let y = 4711            (* y ist nicht veränderbar. *)

x := 5003               (* Wertzuweisung *)
y := 5003               (* Hier liegt ein Typfehler vor *)
```

Wert- und Typbezeichner können in TL jederzeit mit (Meta-) Typinformationen annotiert werden. Für Bezeichner, die dynamischen oder rekursiven Bindungen unterliegen (z.B. Formalparameter in Funktionen oder exportierte Bezeichner in Modulschnittstellen) sind diese Annotationen verpflichtend, während sie ansonsten automatisch durch den Übersetzer inferiert werden.

```
let a :Int = 7
let t :T = tuple a end

(* Funktionsdefinition in ausführlicher Notation: *)
let f = fun(x :Int) :Bool x == 1

(* Die gleiche Funktionsdefinition in abgekürzter Notation: *)
let f(x :Int) :Bool = x == 1

(* Tupelzugriff und Funktionsaufruf: *)
f(t.a)
```

Da das Programmliteral 7 den Typ *Int* bereits implizit vorgibt, ist seine Angabe redundant. Die explizite Typisierung des Parameters *x* ist jedoch obligatorisch.

TL ist wertorientiert, d.h. jeder Term evaluiert zu einem (evtl. trivialen) Wert. Durch die Schlüsselworte **begin** und **end** werden „Blöcke“ eingefaßt, die Anweisungssequenzen enthalten. Die Sichtbarkeit der in einem Block deklarierten Variablen ist durch das Blockende begrenzt. Beispiel:

```
let a = 1
begin
  let a = 2
  let b = a     (* 2 *)
end
a               (* 1 *)
```

Die Konstante *b* wird mit dem Wert 2 initialisiert. Der in der letzten Zeile stehende Bezeichner a bezieht sich jedoch auf die zuerst deklarierte Variable dieses Namens. Daher ist der Wert 1 das Endergebnis. Dies gilt im nachfolgenden Beispiel analog, denn Verzweigungen in Kontrollstrukturen gelten ebenfalls als Blöcke.

```
let a = 1
if x > 7 then
  let a = 2
  ...
else
  ...
end
a                (* 1 *)
```

TL bietet alle gängigen Kontrollstrukturen wie bedingte Verzweigung (**if**), indizierte Fallunterscheidung (**case**), Schleifen (**loop**), Iterationen (**for**) und außerdem dynamische Ausnahmebehandlungen (*exception handling*, **try**). Alle Kontrollstrukturen definieren Blöcke zwischen ihren Hauptschlüsselworten.

```
loop
  ...   (* Block zu wiederholt auszuführender Anweisungen *)
end

try
  ...   (* Block für den „Normalfall“ *)
when bestimmteAusnahme then
  ...   (* Block für eine Ausnahmebehandlung *)
else
  ...   (* Block zur Behandlung aller anderen Ausnahmen *)
end
```

Es können sowohl vordefinierte als auch benutzerdefinierte Typkonstruktoren (sogenannte Typoperatoren) verwendet werden.

```
(* Aggregation in einem geordneten Tupel - ohne und mit Feldbenennung: *)
Let Info = Tuple :String :Real end
Let Person = Tuple name :String  age :Int end

(* Indizierte Aggregation: *)
Let Persons = Array(Person)

(* Spezialisierter Funktionstyp: *)
Let PersonPredicate = Fun(p :Person) :Bool

(* Definitionen von Typoperatoren: *)
Let Verein(Mitglied <:Person) = Tuple
  anzahl :Int  mitglieder :Array(Person)  ...
end
Let Statistik(Mitglied <:Person) = Fun(:Verein(Mitglied)) :Int
```

Das Zeichen <: steht für die in TL induktiv definierte Subtypbeziehung[1] (siehe Abschnitt 3.2.3).

[1] Der im nächsten Beispiel definierte Typ *Student* ist ein Subtyp des hier verlangten Typs *Person*.

Das Schlüsselwort **Rec** kennzeichnet Rekursion. Bei wechselseitiger Rekursion beginnen der zweite und ggf. nachfolgende Typausdrücke jeweils mit **and** anstelle von **Let Rec**. Gleiches gilt für rekursive Wertausdrücke.

```
(* Ein reflexiv und wechselseitig rekursiver Typausdruck: *)
Let Rec Student <:Person = Tuple
  name :String  age :Int  class :Class  buddy :Student
end
and Class <:Ok = Tuple
  students() :list.T(Student)
end

(* Ein entsprechender rekursiver Wertausdruck: *)
let rec otto :Student = tuple
  "otto"  10  classA  hugo
end
and classA = list.cons(studentList  otto)
```

3.2.2 Funktionen höherer Ordnung

In TL liefert jede Funktionsabstraktion einen Wert erster Klasse. Dies gilt insbesondere auch für geschachtelte Funktionen, die auf Objekte des Sichtbarkeitsbereiches, der sie umgibt, zugreifen.

Die folgenden Beispiele aus [Mathiske 92] entstammen ursprünglich [Abelson et al. 84], wo sie in Scheme angegeben sind. Sie simulieren jeweils den (stark vereinfachten) Vorgang des Abhebens eines Geldbetrages von einem Bankkonto. Die Variable *balance* ist global zur Funktion *withdraw*, d.h. sie befindet sich in deren äußerem Sichtbarkeitsbereich. Im ersten Beispiel ist *withdraw* eine ungeschachtelte Funktion. Diese Anordnung wird auch von Sprachen mit einem primitiven Konzept zur Repräsentation von Funktionsobjekten wie Pascal, Modula-2 oder *C* unterstützt.

```
let var balance = 100
let withdraw(amount :Int) :Int =
  if balance >= amount then
    balance := balance - amount
    balance
  else
    raise exception "error" with "Insufficient funds" end
  end

withdraw(25)
⇒ 75
```

Die Notation ⇒ *output* kennzeichnet Ausgaben des Tycoon Systems.

Im nächsten Beispiel werden zwei Instanzen einer unbenannten Funktion erzeugt, die jeweils einzeln die Aufgabe von *withdraw* erfüllen.

```
let makeWithdraw(balance :Int) =
  fun(amount :Int) :Int
    if balance >= amount then
      balance := balance - amount
      balance
    else
      raise exception "error" with "Insufficient funds" end
    end

let w1 = makeWithdraw(100)
let w2 = makeWithdraw(100)

w1(50)
⇒ 50

w2(70)
⇒ 30

w2(40)
⇒ exception "error" with "Insufficient funds" end

w1(40)
⇒ 10
```

Die Funktionen *w1* und *w2* besitzten jeweils eine eigene, unabhängige Instanz des Parameters *balance*.

In TL können Funktionen höherer Ordnung genau wie in Scheme [Abelson et al. 84; Steele 86] in beliebig tiefer Schachtelung auf globale Variablen zugreifen. Unter diesem wichtigen Aspekt der Ausdruckskraft steht TL also selbst einem modernen Lisp-Dialekt nicht nach.

3.2.3 Polymorphie

In TL gibt es eine induktiv definierte Subtypbeziehung auf Typen und Konzepte der existentiellen Typquantifizierung. Ohne das Thema vertiefen zu wollen, sei eine kurze Definition des Begriffes Subtyppolymorphismus genannt:

> Variablen können dynamisch an Objekte verschiedener Struktur gebunden werden, sofern deren Struktur eine gemeinsame generalisierte Spezifikation erfüllt.

Außerdem bietet TL universelle Typquantifizierung durch parametrischen Polymorphismus [Milner 78] und *Bounded Parametric Polymorphism* [Meyer 86]. Damit sind in TL fast[2] alle gängigen Polymorphiekonzepte vorhanden, die heutzutage im Rahmen eines *statisch typsicheren* Systems unterstützt werden können. Eine eingehende Darstellung des Typsystems höherer Ordnung von TL liefert [Matthes, Schmidt 92]. Desweiteren wird sich in Kapitel 5 zeigen, daß viele datenunabhängige Operationen, die für verteilte Anwendungen typisch sind

[2] Nicht unterstützt wird z.B. *F-Bounded Parametric Polymorphism* [Canning et al. 89].

(Datenübertragung, Aggregation, Kopieren, ...) durch eine polymorphe Typdisziplin auf natürliche Weise zu erfassen sind.

Die Signatur einer polymorphen TL-Funktion wird z.B. wie folgt angegeben:

*sort(A <:Ok a :**Array**(A) greaterEqual(:A :A):Bool) :Ok*

Die Funktion *sort* erwartet als Argumente einen Typ, einen Wert und eine Funktion. Für jedes Typargument A, das ein beliebiger Subtyp des Typs *Ok* sein kann, muß der Wert *a* ein indizierbares Feld (*Array*) mit Elementen des Typs A sein und die Funktion *greaterEqual* muß eine Vergleichsfunktion für Paare von Werten des Typs A sein. Bei *Ok* handelt es sich um den höchsten Typ in der TL-Typhierarchie. Bezüglich der Subtypbeziehung umfaßt er alle anderen Typen.

Im Hinblick auf Mobilität ist folgender implementatorischer Aspekt der genannten Polymorphismen wichtig: unnötige Spezialisierungen im Aufbau von Objekten schränken ihre polymorphe Verwendung ein. Die Unterstützung von Polymorphie erzwingt *uniforme* Speicherrepräsentationen (siehe Abschnitt 3.4) für TL-Objekte. Diese Uniformität ist wiederum vorteilhaft für die Realisierung der Mobilität von TL-Objekten: sie ermöglicht die Schaffung eines relativ unkomplizierten *polymorphen* Linearisierungsalgorithmus (siehe Abschnitt 4.3.2), der beliebige TL-Objekte in kanonischer Weise transformiert.

3.2.4 Dynamische Typen

In manchen Programmiersituationen liegt in dem Kontext, in dem ein Wert erzeugt wurde, und dem Kontext, in dem er benutzt wird, keine gemeinsame statisch überprüfbare Spezifikation vor. Ein einfaches Beispiel für dieses Problem ist die reflektive Funktion *eval* [Geisler 95], die als Parameter eine Zeichenkette übernimmt, diesen als TL-Ausdruck compiliert und auswertet und das Ergebnis zurückliefert.

eval(sourceText :String) :???

Wie durch die Fragezeichen angedeutet, läßt sich das Ergebnis dieser Funktion innerhalb des Aufrufkontextes nicht statisch typisieren. Der Typ des Rückgabewertes kann hier erst zur Ausführungszeit festgestellt werden, da er vom dynamisch gebundenen Quelltext abhängt. Auch in folgenden Situationen liegt eine entsprechend wirkende Trennung von Auswertungskontexten vor:

- beim Objektaustausch zwischen persistenten Objektspeichern mittels linearer Datenrepräsentationen (siehe Abschnitt 4.3),
- bei der Generierung und Vermittlung (*Trading)* von Schnittstellenbeschreibungen (siehe Abschnitte 5.1.2 und 5.2.3.3),
- bei der Bindung von RPC-*Stubs* (siehe Abschnitt 5.2.2.2).

Durch die Angabe des Schlüsselwortes **Dyn** wird veranlaßt, daß die zur Compile-Zeit erzeugte interne Repräsentation eines statisch deklarierten Typs als Programmliteral und damit als

Laufzeitwert zur Verfügung steht. Solche sogenannten *dynamischen Typen* können mit Hilfe der Funktion *typeRep.isSubType* zur Laufzeit bezüglich der Subtyprelation miteinander verglichen werden. In folgendem Beispiel[3] wird eine per se typunsichere Operation durch einen dynamischen Typtest „geschützt“:

```
Let Person = Tuple name :String end

let printName(Dyn P <:Tuple end  p :P)
  if (typeRep.isSubType(:P  :Person)) then
    (* Diese Typanpassung ist per se typunsicher: *)
    let person = unsafe.typeCast(p  :Person)

    print.string(person.name)
  else
    print.string("Typ paßt nicht")
  end
```

Eingehende Beschreibungen dynamischer Typen und ihrer Implementation im Tycoon-System liegen in [Geisler 95; Koehler 96] vor. Verwandte Mechanismen treten z.B. in Napier88 [Connor 88] und Modula-3 [Horning et al. 93; Birell et al. 93b] auf.

3.2.5 Modulare Spracherweiterungen

Zur Organisation auf Wiederverwendbarkeit ausgerichteter Programmteile dienen Module. Modulschnittstellen definieren die Signaturen von Werten und Typen, die von den sie implementierenden Modulen exportiert werden. Hier ist ein Beispiel [Mathiske et al. 93]:

```
interface Zaehler
export
   T <:Ok (* Abstrakter Datentyp *)
   erzeugen(init :Int) :T
   inkrementieren(z :T) :Ok
   dekrementieren(z :T) :Ok
   istNull(z :T) :Bool
   anzeigen(z :T) :Ok
end;
```

[3] Das Beispiel wurde im Hinblick auf einfache Verständlichkeit konstruiert. Es soll hier jedoch keineswegs angeregt werden, auf der Anwendungsebene in diesem Stil zu programmieren. Wirklich sinnvolle Anwendungen für eine derartige Vorgehensweise treten in leider weniger instruktiven Situationen bei der Systemprogrammierung auf.

```
module zaehler
import fmt  print
export
  Let T = Tuple var wert :Int end (* Konkreter Datentyp *)
  let erzeugen(init :Int) = tuple let var wert = init end
  let inkrementieren(z :T) = z.wert := z.wert + 1
  let dekrementieren(z :T) = z.wert := z.wert - 1
  let istNull(z :T) = z.wert == 0
  let anzeigen(z :T) = print.string("Der Zähler steht auf " <>[4] fmt.int(z))
end;
```

Im Unterschied zu konventionellen modularisierten Programmiersprachen (z.B. Modula-2) sind als Instanzen einer Modulschnittstelle *mehrere* (implementatorisch unterschiedliche) Module möglich. Sie sind außerdem Datenobjekte „erster Klasse". Modulschnittstellen werden durch Tupel-Type repräsentiert und Module durch Funktionen, die ein entsprechendes Tupel erzeugen. Die Ausführung einer solchen Erzeugerfunktion entspricht dem dynamischen „Linken" eines Moduls. Alle zu exportierenden Bestandteile des Moduls werden dabei in dem als Funktionsergebnis zurückzugebenden Tupel zusammengestellt. Die Erzeugerfunktion kapselt die importierten Werte (s.o. *print*), die in ihrem globalen Sichtbarkeitsbereich liegen.

Felder importierter Module werden durch qualifizierte Bezeichner wie z.B. *print.string* referenziert. Ein gebundenes Modul unterscheidet sich also als Wert in keiner Weise von einem gewöhnlichen Tupel. Diese Identifikation kann genutzt werden, um Proxys für entfernte Module ohne die Notwendigkeit der Einführung neuer Schlüsselworte in die Sprache TL einzubetten (siehe Abschnitt 5.1.3 und 5.2.2.2).

In dem Bemühen Mehrzweckprogrammiersprachen und Datenbankanfragesprachen, bzw. Sprachen dritter und vierter Generation zu vereinen, steht TL in der Tradition von DBPL [Matthes, Schmidt 89; Matthes et al. 92b; Schmidt, Matthes 92], Modula/R [Koch et al. 83] und Pascal/R [Schmidt 77; Lamersdorf, Schmidt 80]. Im Gegensatz zu diesen Vorgängern sind die Sprachkonstrukte zur Manipulation von Massendaten (z.B. Relationen, Listen, Mengen) bei TL (und auch bereits bei P-Quest [Matthes et al. 92a]) nicht von vornherein fest in den Sprachkern und das Laufzeitsystem "eingebaut" (*builtin*). Stattdessen werden sie nachträglich (*add-on*) mit den Mitteln (Benennung, Typisierung, Bindungen, Scoping, Rekursion, etc.) derselben Sprache, in der sie zum Einsatz kommen, definiert und implementiert. Die Überlegenheit des *add-on*-Ansatzes wird in [Matthes, Schmidt 91] ausführlich dargelegt. Sie besteht hauptsächlich in folgenden Punkten:

- Es ist nicht erforderlich, spezielle syntaktische Konstrukte zur Behandlung von Massendatenstrukturen einzuführen. Dadurch wird bei der Systemprogrammierung erheblicher Aufwand vermieden. Dies gilt insbesondere für den Fall der Einführung zusätzlicher Massendatentypen.

- Die Trennung zwischen System- und Anwendungsprogrammierung wird gemildert. Dadurch wird es einfacher, Anwendungscode herauszufaktorisieren und als Bestandteil der Programmierumgebung einer größeren Anzahl von Programmierern zur Verfügung zu

[4] Dieses Zeichen steht in TL für die Konkatenation von Zeichenketten.

stellen. Umgekehrt besteht die Möglichkeit, generischen Code aus der Programmierumgebung zu entnehmen, zu spezialisieren und an anwendungsspezifische Bedürfnisse (z.B. Performanzerhöhung in bestimmten Fällen) anzupassen.

- ▷ Es besteht keine künstliche syntaktische oder sprachliche Barriere zwischen Anfrageausdrücken und allgemeinen Programmausdrücken wie z.B. Funktionsaufrufen. Um spätere Fortschritte in der Programmoptimierung oder der Datenbankanfrageoptimierung (sowie bei Bemühungen beide zu vereinen [Gawecki, Matthes 95a]) zu nutzen, muß weder die Sprachdefinition revidiert werden, noch müssen existierende Anwendungsprogramme angepaßt werden.

Hier zeigt sich, daß die Wahl zwischen *builtin* und *add-on* nicht nur eine Entwurfsentscheidung bezüglich des Sprachdesigns, sondern auch der Systemarchitektur ist.

Beispiele für die Implementation wichtiger Basisfunktionalitäten im *add-on*-Ansatz sind in [Matthes, Schmidt 91] und [Niederée 92] beschrieben. Mit der vorliegenden Arbeit kommt nun eine generische Bibliothek zur verteilten Programmierung hinzu (siehe Abschnitt 5.2).

3.2.6 Flache Transaktionen

Das Modul *persistent* bietet Funktionen an, mit deren Hilfe flache Transaktionen gesteuert werden können. Die wesentlichen Elemente seiner Schnittstelle sind:

```
interface Persistent
export

  Via <:Ok
  (* Abstrakter Aufzählungstyp für folgende Konstanten. *)

  viaCommit, viaRollback, viaRestart :Via
  (* Ausführungsvarianten der folgenden Funktion. *)

  commit() :Via
  (* Beenden und Sichern einer flachen Transaktion. *)

  rollback() :Nok
  (* Zurücksetzen auf den Zustand for dem letzten 'commit'. *)

  ...
end;
```

Anhand des Rückgabewertes der Funktion *commit* kann unterschieden werden, aufgrund welcher Ausgangssituation der Anfang einer neuen Transaktion erreicht wird. Dadurch ist es möglich, externe Zustände mit der internen Transaktionsverarbeitung des Objektspeichers zu synchronisieren. Zum Beispiel können externe flüchtige Daten durch ein Logging-Verfahren so eingebunden werden, daß sie persistent erscheinen [Kornacker 95]. Eine entsprechende Erweiterung obiger Schnittstelle ist aus Anhang E.1 ersichtlich. Die konsequente Weiterverfolgung dieses Ansatzes der Pseudo-Persistenz führt zu einer logging-basierten Pseudo-Mobilität (siehe Abschnitt 7.6, Anhang E.2).

3.3 Die abstrakte Objektspeicherschnittstelle Tsp

Praktisch alle Systeme, die mit großen langlebigen Datenbeständen arbeiten, weisen eine Systemarchitektur auf, in der eine klare Trennung zwischen den Aufgaben der Datenmanipulation und Visualisierung sowie den Aufgaben der zuverlässigen persistenten Speicherung von Massendaten besteht. Da Aktionen in der Manipulationsschicht Datenzugriffsoperationen in der Speicherungsschicht auslösen, führt diese Trennung auf natürliche Weise zu Client/Server-Architekturen, in denen ein Speicherprotokoll mögliche Interaktionen zwischen einem Client-Programm und einem Datenbank-Server definiert.

Für Standarddatenbankanwendungen in der Wirtschaft ist SQL [ISO-SQL 89] de facto das dominierende Speicherprotokoll. SQL bietet *standardisierte* anwendungsorientierte Speicheroperationen auf hoher Abstraktionsebene, es ist weithin verfügbar und es existieren sowohl statische (*Embedded SQL*) als auch dynamische (ODBC [Pietrzyk, Radic 94]) de facto standardisierte Mechanismen zur Koppelung mit der Manipulationsschicht von Anwendungsprogrammen. Der OMG-Standard [Catell 94; Cattell 94] und die ISO STEP-Standards [ISO-EXP 92] definieren Speicherprotokolle auf ähnlicher Abstraktionsebene für objektorientierte Datenbanken sowie Repositories von CAD-Daten.

Leider sind diese kommerziell gut unterstützten Speicherprotokolle nur bedingt für die Konstruktion voll integrierter persistenter Programmierumgebungen geeignet, denn ihre relativ anwendungsorientierten Datenmodelle sind im Verhältnis zu den flexiblen Ausdrucksmöglichkeiten persistenter Programmiersprachen zu starr [Matthes et al. 92a; Müller 91; Matthes et al. 95a]:

- Vollständige Persistenzabstraktion [Atkinson, Bunemann 87] erfordert einen uniformen, effizienten elementorientierten Zugriff auf persistente und nicht-persistente Objekte. Eine Implementation durch Standarddatenbanken führt in diesem Fall zu einem zweistufigen Speichermanagement. Nicht nur aufgrund der größeren Schnittstellentiefe, sondern insbesondere aufgrund von Modellunterschieden ergibt sich eine geringe Performanz.
- Polymorphe Datenstrukturen, (polymorphe) Funktionen höherer Ordnung und benutzerdefinierte abstrakte Datentypen sind auf anwendungsnahen Ebenen nicht mit monomorphen Datenmodellen (bzw. Typsystemen) realisierbar [Matthes 93].
- Inkrementelle Programmkonstruktion und reflektive Programmiertechniken erfordern dynamische Schemaänderungen und dynamische Bindungstechniken, die von den meisten Standardspeichermodellen nicht geleistet werden können.

In Konsequenz dieser Schwierigkeiten ist praktisch jede persistente Programmierumgebung auf einen speziellen Objektspeichertyp zugeschnitten und umgekehrt. Bei der Betrachtung vorhandener Paare von Objektspeichern und persistenten Sprachen wie z.B. Exodus und E, O_2 und O_2C oder ObjectStore und Persistent *C* fällt auf, daß die internen Objektspeicherschnittstellen gewöhnlich unterhalb der Abstraktionsebene von SQL oder ODMG-Sprachen ansetzen. Außerdem sind diese Schnittstellen durch starke Interdependenzen zwischen Sprachprozessor und Speichersystem sowie durch ad-hoc-Entscheidungen bezüglich Objektrepräsentationen und -lebensdauer geprägt.

Die abstrakte Objektspeicherschnittstelle Tsp (*Tycoon Store Protocol*) basiert auf dem weithin anerkannten Begriff einer untypisierten persistenten Speicherhalde [Moss, Sinofsky 88;

Brown, Morrison 91; Kim et al. 89]. Obschon das TSP im Rahmen des Tycoon-Projektes [Matthes, Schmidt 93] entwickelt und evaluiert worden ist, handelt es sich um eine vollkommen unabhängig einsetzbare Komponente, die für unterschiedlichste Klientenprogramme und Objektspeichertypen geeignet ist. Zur Zeit wird das TSP von den Laufzeitsystemen zweier persistenter Sprachen (TL [Matthes, Schmidt 92] und TooL [Gawecki, Matthes 95b]), die an der Universität Hamburg entwickelt wurden, verwendet. Außerdem dient es als Speichermedium für eine Implementation des *Network File System* (NFS), die persistente Objektspeicher als UNIX-Dateisystem mit zusätzlichen transaktionalen Eigenschaften erscheinen läßt. Die Anknüpfung weiterer TSP-Klienten, wie z.B. das Laufzeitsystem der persistenten Sprache Fibonacci [Albano et al. 94], sind in der Entwicklung.

Jeder der genannten Klienten des TSP kann jede der folgenden TSP-Implementationen mit unterschiedlicher operationaler Charakteristik und Performanz verwenden:

Tymem ist ein portabler, komplett im Hauptspeicher residierender Objektspeicher, dessen Persistenz und Fehlererholung über Standarddateidienste gehandhabt werden.

Tysin ist ein Objektspeicher für Einzelbenutzerbetrieb, der den größten Teil des Datenbestandes auf Sekundärspeicher auslagert und außerdem performante Datensicherungs- und Fehlererholungsmechanismen bietet.

Tynap ist in Adapter für den auf SunOS beschränkten Napier-Objektspeicher [Brown 89]. Durch starke Ausnutzung sogenannter *memory-mapped files* [Rosenberg et al. 90] erreicht dieser Objektspeicher eine besonders schnelle Objektadressierung bei gleichzeitig geringem Hauptspeicherbedarf.

Tyobj ist ein Adapter für das kommerzielle Produkt ObjectStore der Firma ObjectDesign, bei dem es sich um einen Multi-User-Objektspeicher mit Client/Server-Architektur handelt.

Das TSP ist „stapelbar“: zusätzliche Dienste wie Zugriffskontrolle [Rudloff et al. 95], Versionsmanagement oder erweitertes Transaktionsmangement können auf einer bestehenden TSP-Implementation aufsetzen und nach oben wiederum eine TSP-konforme Schnittstelle anbieten. Diese kann um zusätzliche Operationen erweitert sein. Zum Beispiel existiert eine TSP-Version, die zusätzlich die Möglichkeit bietet, multiple persistente Sicherungspunkte zu etablieren sowie den Objektspeicher gezielt auf einen bestimmten Sicherungspunkt zurückzusetzen [Kornacker 95].

Im Gegensatz zur Speicherverwaltung in relationalen und objektorientierten Datenbanksprachen beruht das TSP auf einem untypisierten aber hochflexiblen Speichermodell mit relativ geringem Abstraktionsgrad, das auf die Implementation persistenter polymorpher Programmiersprachen höherer Ordnung ausgerichtet ist. Ein TSP-Klient kann dynamisch zwischen unterschiedlichen Objektspeicherimplementationen wählen und es möglich, komplexe Objektgraphen zwischen allen Objektspeichern auszutauschen, die über das TSP angesprochen werden können. Dazu dient eine plattformunabhängige, kanonische lineare externe Datenrepräsentation.

Ein wichtiges Ziel bei der Entwicklung des TSP ist, unabhängig vom Daten- und Sprachmodell möglicher TSP-Klienten, eine effiziente Datenspeicherung zu bieten. Insbesondere soll das TSP in der Lage sein, polymorph typisierte Modelle zu unterstützen, in denen die Komponenten

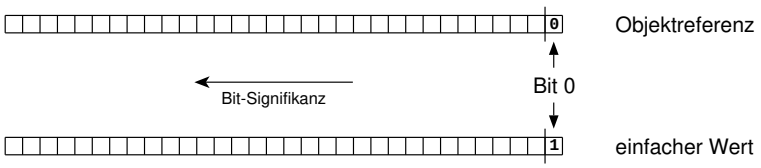

Abbildung 3.2: Das *Tagging*-Schema des TSP

von Speicherobjekten Werte mehrerer unterschiedlicher Typen enthalten, ohne daß in einer separaten Schemadefinitionsphase eine diesbezügliche Festlegung stattfinden kann. Daher weist das TSP ein weitgehend *untypisiertes* Datenmodell auf sehr niedriger Abstraktionsebene auf. Es gibt kein separates Verzeichnis und keinerlei Schemainformationen, die durch das TSP verwaltet werden. Stattdessen sind alle TSP-Speicherobjekte in Bezug auf für die Speicherungsebene relevante Aspekte *selbstbeschreibend*, was durch ein reguläres Objekt-Layout und ein uniformes *Tagging*-Schema erreicht wird.

Das TSP muß ohne Festlegung auf bestimmte Basisdatentypen (Ganzzahlen, Fließkommazahlen, Schriftzeichen, Zeichenketten, ...) oder Typkonstruktoren (Record, Array, Liste, ...) definiert sein. Algorithmen wie Zugriffskontrolle, Garbage-Collection und portabler Datenaustausch müssen mit nahezu minimalen semantischen Informationen über die Daten auskommen.

3.3.1 Basistypen und Tagging

Die TSP-Schnittstelle erwartet die Definition folgender Datentypen durch den TSP-Klienten:

***tsp_Bool*:** boolsche Werte. Zusätzlich sind die Konstanten *tsp_FALSE* und *tsp_TRUE* als mögliche Werte dieses Typs festzulegen. Daß hier "das Rad neu erfunden" wird, hat zwei Gründe. Zum einen gibt es in *C* keinen entsprechenden vordefinierten Typ. Zum anderen kann auf diese Weise die Speichergröße der Werte unter Kontrolle gebracht werden, was die Anpassung an TSP-Klienten mit eigenen diesbezüglichen Einstellungen erleichtert.

***tsp_String*:** Zeichenketten. Die Deklaration lautet stets:

*typedef char *tsp_String;*

Dementsprechend hat die Verwendung von *tsp_String* anstelle von *char ** einen reinen Kommentarcharakter.

***tsp_Store*:** ein abstrakter Typ für Objektspeicher-Handles. Jede operative TSP-Funktion hat einen Parameter dieses Typs, der angibt, auf welchen Objektspeicher sie angewendet werden soll. Die Deklaration lautet stets:

*typedef void *tsp_Store;*

***tsp_Word*:** ein Maschinenwort der CPU, typischerweise 32 Bits. Gleichzeitig werden Werte dieses Typs als positive Ganzzahlen aufgefaßt und ggf. für Berechnungen verwendet. Die *C*-Deklaration sollte daher lauten:

typedef unsigned long tsp_Word;

***tsp_Byte*:** ein Byte, also 8 Bits. Auch dieser Typ ist nicht abstrakt, d.h. es sind alle möglichen Rechenoperationen für Ganzzahlen von 0 bis 255 nutzbar. Die *C*-Deklaration lautet daher stets:

typedef unsigned char tsp_Byte;

Alle Datenstrukturen in Objektspeichern basieren auf den beiden letztgenannten Typen, wobei *tsp_Word* in einige Varianten unterschieden wird. Für markierte Worte ist als Alias *tsp_Tagged* definiert:

typedef tsp_Word tsp_Tagged;

Markierte Worte werden weiter in Objektreferenzen und in einfache ("unmittelbare", "atomare") Werte wie Ganzzahlen, Buchstaben oder Wahrheitswerte unterschieden. Das verwendete Markierungsschema (*Tagging*) ist in Abbildung 3.2 ersichtlich. Der Objektspeicher unterscheidet anhand eines zu diesem Zweck reservierten Bits Objektreferenzen von einfachen Werten (*Immediates*). Um in *C*-Programmen die Verwendung eines markierten Wertes als Objektreferenz zu dokumentieren, wird der Alias *tsp_OID* verwendet; für einfache Werte steht *tsp_Immediate*.

Alle Klientendaten müssen durch einzelne Bytes, Worte, markierte Worte oder durch in Speicherobjekten repräsentierte *homogene* Folgen jeweils einer dieser Varianten repräsentiert werden.

Array: eine Folge entsprechend des *Tagging*-Schemas diskriminierter Worte.

Byte Array: eine aus Sicht des Objektspeichers uninterpretierte Byte-Folge.

Word Array: [5] eine aus Sicht des Objektspeichers uninterpretierte Wort-Folge.

Abbildung 3.3 stellt den internen Aufbau solcher Speicherobjekte schematisch dar. Jedes Speicherobjekt besteht aus einem Kopfteil mit Meta-Informationen und einem Datenteil, der die Nutzdaten enthält. Die Elemente des Datenteils, Bytes oder Worte, werden abstrahierend als *Slots* bezeichnet.

[5] Diese Variante kommt z.Z. (noch) nicht bei der Repräsentation von TL-Werten zum Einsatz. Stattdessen werden Byte-Arrays verwendet.

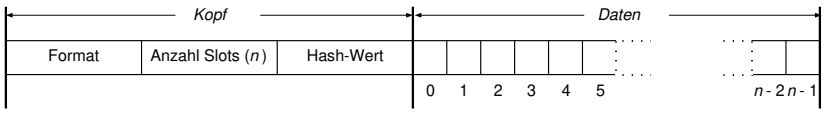

Abbildung 3.3: Das interne Objekt-Layout in Tycoon-Objektspeichern

3.3.2 Programmierschnittstellen

Programmtechnisch gesehen besteht das TSP in einer *C*-Header-Datei, in der einige Typen, Konstanten, Makros, und Funktionen definiert sind, über deren korrekte Verwendung beigefügte Kommentare Auskunft geben. Folgende Deklarationen sind typisch:

```
extern tsp_OID tsp_newArray(tsp_Store store, tsp_Format format, tsp_Word nSlots);
/* Lege ein Speicherobjekt mit nSlots Feldern der Größe von tsp_Word an.
   Ergebnis: eine Objektreferenz. */

extern tsp_OID tsp_newByteArray(tsp_Store store, tsp_Format format,
                                tsp_Word nSlots);
/* Lege ein Speicherobjekt mit nSlots Byte-Feldern an.
   Ergebnis: eine Objektreferenz. */

extern tsp_Word tsp_nSlots(tsp_Store store, tsp_OID oid);
/* Liefere die Anzahl der Felder des durch oid angegebenen Speicherobjektes. */

extern tsp_Word tsp_getWord(tsp_Store store, tsp_OID oidArray,
                            tsp_Word iIndex);
/* Liefere den Inhalt des Feldes mit Index iIndex in Speicherobjekt oidArray. */

extern void tsp_setByte(tsp_Store store, tsp_OID oidByteArray,
                        tsp_Word iIndex, tsp_Byte b);
/* Weise dem Byte-Feld mit Index iIndex in Speicherobjekt oidArray den Wert b zu. */
```

Eine komplette Auflistung dieser TSP-Schnittstelle befindet sich in [Matthes et al. 95a].

Objektspeicheradapter implementieren die so definierten Funktionen jeweils mit Hilfe der vom betreffenden Objektspeicher bereitgestellten Schnittstellen. Die Objektspeicher *Tymem* und *Tysin* sind in *C* implementiert. Sie verwenden außer dem puren ANSI-*C*-Sprachumfang nur die Standard-*C*-Bibliothek und sind dadurch sehr portabel.

Mit Hilfe des Parameters *store* kann ein TSP-Klientenprogramm wechselweise auf mehrere unterschiedliche Objektspeicher zugreifen. Das Tycoon-System nutzt diese Flexibilität jedoch nicht aus, da durch die Verwendung eines einzigen Objektspeichers ein homogener Adreßraum aller lokalen Daten von Tycoon-Anwendungen am einfachsten zu realisieren ist.

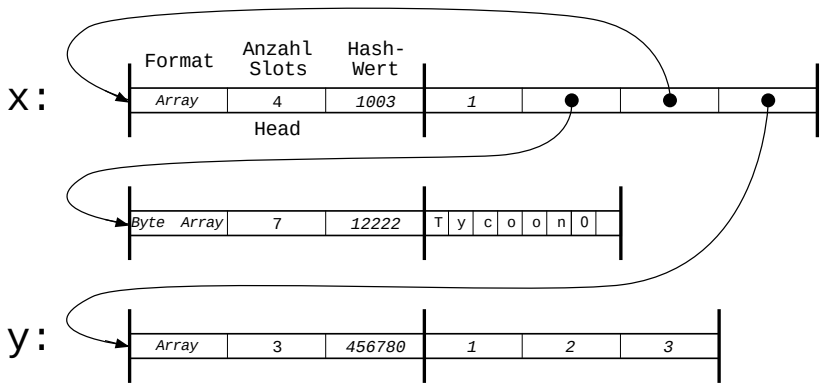

Abbildung 3.4: Die interne Repräsentation eines TL-Wertes

3.4 Uniforme Laufzeitrepräsentationen in persistenten Objektspeichern

Das TSP spezifiziert eine wohldefinierte Schnittstelle zwischen Tycoon-Sprachen (Front-End) und einem frei gewählten Objektspeicher (Back-End). Es bietet die Abstraktion eines homogenen unbegrenzten Objektspeichers, der das Anlegen, Lesen und Ändern von polymorphen, inhomogenen Datenobjekten gestattet.

Alle semantischen Objekte (Tuple, Mengen, Funktionen, dynamische Typbeschreibungen, Module, etc.) der Tycoon-Sprachen werden durch kanonische Abbildungen auf primitive Objektspeicherstrukturen im Objektspeicher realisiert.

Komplexe TL-Strukturen ergeben dabei markierte *Objektgraphen*, deren referentielle Strukturen aufgrund des *Taggings* ohne hochsprachliche Typinformation nachvollziehbar sind. Letzteres ist sowohl die Grundlage der Garbage-Collection, als auch des in Abschnitt 4.3 beschriebenen Linearisierungsverfahrens für den Objektaustausch zwischen persistenten Objektspeichern.

Der Einsatz eines persistenten Objektspeichers gewährleistet die Persistenz aller Tycoon-Objekte, die ausschließlich durch Benutzung der wohldefinierten Softwareschnittstelle TSP manipuliert werden. Jedes im Objektspeicher angelegte Objekt ist eindeutig über eine abstrakte Objektspeicherreferenz identifizierbar.

Zum Beispiel wird folgender TL-Wert wie in Abbildung 3.4 ersichtlich repräsentiert.

Let Rec *X <: Ok =* ***Tuple*** *:String :Array(Int)* ***end***

let rec *x :X =* ***tuple*** *"Tycoon" x* ***let*** *y =* ***array*** *1 2 3* ***end end***

Abbildung 3.5 illustriert den Objektgraphen eines komplexeren TL-Wertes in einer vereinfachten graphischen Notation[6].

[6]Diese graphische Notation wird auch in Abschnitt 4.3 und Kapitel 7 eingesetzt.

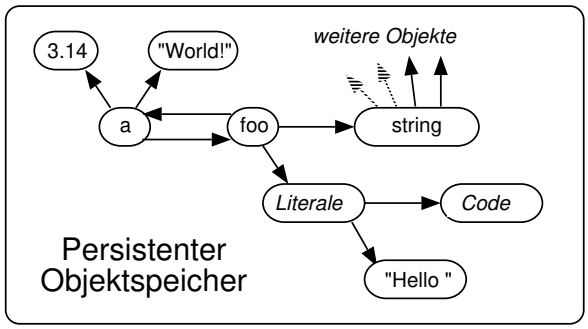

Abbildung 3.5: Der Objektgraph eines komplexen TL-Wertes

```
Let Rec A = Tuple
   r:Real  var s :String  foo(b :A):String
end

let rec a :A = tuple
   let r = 3.14
   let var s = "World!"
   let foo() :String = string.concat(a.s "Hello ")
end
```

Die Objektspeicherrepräsentation der Funktion *foo* besteht aus Referenzen auf die Literale von *foo* (konstante Bindungen an den Code von *foo* und an konstante Werte, die im Quelltext der Funktion auftreten) sowie aus Referenzen, die den Funktionsabschluß von *foo* bilden (das Modul *string* und der Wert a).

Die Ausrichtung auf Mobilität erfordert in einem noch stärkeren Maße, als es bei der Persistenz ohnehin schon der Fall ist, daß mit Objektreferenzen sparsam umgegangen wird. Um die transitive referentielle Hülle eines Objektes zu minimieren, darf seine Repräsentation keine (indirekten) Referenzen auf Objekte enthalten, die für seinen künftigen Gebrauch irrelevant sind. Ein typisches Gegenbeispiel ist eine Repräsentation von Funktionsabschlüssen, die auf verketteten Listen beruht, wie z.B. in Napier88 [Morrison et al. 94]. In diesem Fall können Funktionen unbemerkt indirekte Verweise auf Massendaten enthalten, insbesondere auch auf das Wurzelobjekt des gesamten persistenten Objektspeichers. Solch eine Situation verhindert faktisch jede Code- und Thread-Übertragung und ist außerdem ein ernstes Hindernis für Garbage-Collections.

Um diese Problematik von vornherein zu vermeiden, verwendet Tycoon flache Funktionsabschlüsse minimaler Größe, deren Zusammensetzung von einer *statischen* Bindungsanalyse vor der Codegenerierung bestimmt wird.

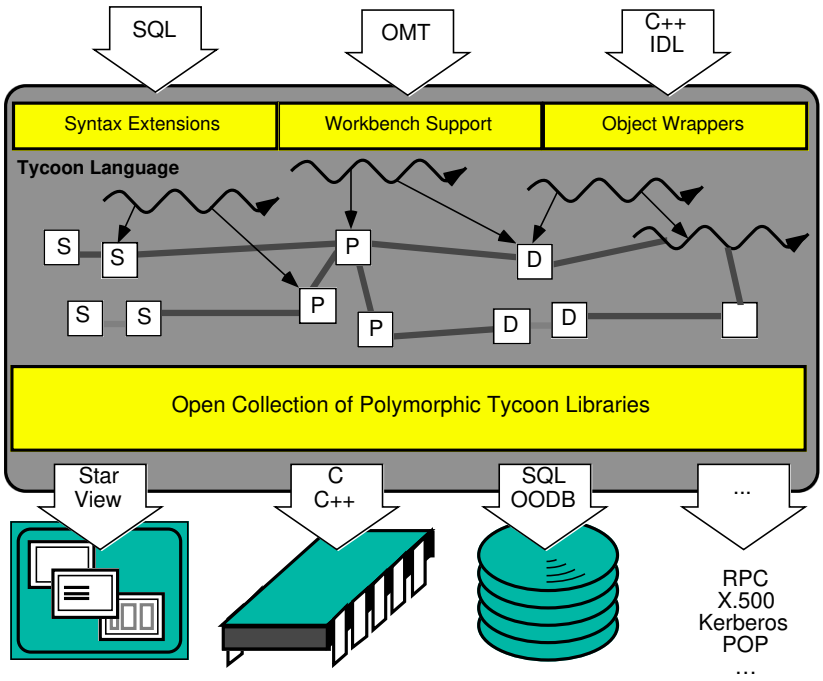

Abbildung 3.6: Die Grundstruktur des offenen persistenten Objektsystems Tycoon [Matthes et al. 95b]

3.5 Externe Bindungen

Die Grundstruktur des Tycoon-Systems ist gegenüber der in Abbildung 1.3[7] illustrierten Grundstruktur persistenter Objektsysteme wie in Abbildung 3.6 [Matthes et al. 95b] ersichtlich erweitert. Wie durch die Pfeile am oberen Rand angedeutet, kann TL-Anwendungscode von Hauptprogrammen aufgerufen werden, die in anderen Sprachen (z.B. *C*) geschrieben sind. Es kann das gesamte Tycoon-System als *Subsystem* anderer Programme fungieren.

Auf der anderen Seite werden bestehende und neu entwickelte externe Bibliotheken (z.B. für graphische Benutzeroberflächen) in uniformer Weise als TL-Bibliotheken in das Tycoon-System integriert. Dadurch ist ein typsicherer Zugriff auf externe Daten und Code sichergestellt. Integrierte externe Funktionen werden syntaktisch nicht von in TL implementierten Funktionen unterschieden.

Im Rahmen der vorliegenden Dissertation wurde für TL eine *transparente bidirektionale* Programmierschnittstelle zu *C* entwickelt [Matthes et al. 94], die eine nahtlose Integration der Funktionsparadigmen beider Sprachen erlaubt.[8] Externe *C*-Funktionen können in TL als gewöhnliche TL-Werte eingebunden werden und TL-Funktionen können dynamisch mit einem speziellen Callback-Stub ummantelt werden, sodaß sie in Form von *C*-Funktionszeigern zur Verfügung stehen.

[7] Siehe Abschnitt 1.2, auf Seite 11.

[8] In [Geisler 95] wird auf Basis der gegebenen *C*-Schnittstelle außerdem eine *C*++-Schnittstelle entwickelt.

3.5.1 Funktionsaufrufe von Tl nach *C*

Der Mechanismus zur Einbindung in externen Bibliotheken implementierter Funktionalität ist generisch. Bindungen von TL-Bezeichnern an externe Funktionswerte liefert die vordefinierte Spezialform *bind*, die syntaktisch wie eine Funktion aufgerufen wird. Sie hat folgende Signatur:

bind(Function <:Ok file, label, format :String) :Function

Die Parameter haben folgende Bedeutung. *Function* beschreibt den Typ der resultierenden TL-Funktion. Er muß die Form ***Fun**(...) :A* haben. Der Parameter *file* bezeichnet den Dateinamen der dynamischen Bibliothek (je nach Betriebssystemjargon: *dynamic link library*, *shared library*, o.ä.), welche die angeforderte *C*-Funktion enthält. Der Parameter *label* gibt den Namen der *C*-Funktion an. Zu guter Letzt enthält der Parameter *format* Steuerzeichen zur Festlegung der Parameterkonvertierung von TL nach *C*.

Um die Verwendung von *bind* anhand eines Beispiels zu verdeutlichen, werde angenommen, daß eine Bibliothek */usr/lib/libexample.so* eine Funktion *example* mit folgender *C*-Deklaration enthalte.

***extern long** example(**char** *s);*

Diese Funktion erwartet eine Zeichenkette als Argument und liefert eine Ganzzahl (**long**). Eine passende Einbindung in TL ist:

```
let cCallExample =
  bind(:Fun(:String) :Int "/usr/lib/libexample.so" "example" "si")
```

Der so entstehende Wert *cCallExample* hat den Typ ***Fun**(:String) :Int*. Diese Funktion kann wie folgt aufgerufen werden.

***let** result :Int = cCallExample("My favorite String")*

Das Aufsuchen externer Bibliotheken und der Einsprungpunkte externer Funktionen findet jeweils dynamisch beim Aufruf statt. Durch diese Entkoppelung von statischen Abhängigkeiten sind externe Bindungen persistent und im Prinzip auch mobil, d.h. sie können ohne weiteres zwischen Tycoon-Anwendungen übertragen werden. Es ist jedoch Aufgabe des Programmierers, sicherzustellen, daß die erforderlichen *C*-Bibliotheken auch tatsächlich vorhanden sind. Ein diesem Sinne ungedeckter Aufruf wird vom Laufzeitsystem als Fehlersituation erkannt, woraufhin dieses eine dynamische Ausnahme auslöst.

3.5.2 Funktionsaufrufe von *C* nach Tl

Die Programmierschnittstelle zwischen TL und *C* ist bidirektional. Es ist nicht nur möglich, *C*-Funktionen aus TL heraus aufzurufen, sondern umgekehrt können auch TL-Funktionen von *C* aus aufgerufen werden. Solche Aufrufe , die *Callbacks* genannt werden, sind zum Beispiel bei der Einbindung externer Fenstersysteme erforderlich.

Um den notwendigen Programmieraufwand zur Bereitstellung von Callbacks zu minimieren, ist es wünschenswert, TL-Funktionen in *C* möglichst exakt das gleiche Erscheinungsbild wie *C*-Funktionen zu verleihen. Dies ist in der häufigen Situation, daß externe Softwarekomponenten Verwendung finden, die nachträglich nicht mehr geändert werden können, ein entscheidender Vorteil.

Das Modul *cCallback* exportiert einen (parametrisierten) abstrakten Datentyp *T*, der *C*-Funktionszeiger, die sich auf Callbacks nach TL beziehen, repräsentiert. Werte dieses Typs erzeugt eine Funktion mit folgender Signatur:

new(Function <:Ok function :Function format :String) :T(Function)

Der erste Wertparameter (*function*) von *new* muß stets eine Funktion sein. Leider kann dieser Umstand im TL-Typsystem nicht ausgedrückt werden, weshalb der Typparameter (*Function*) lediglich die maximal unverbindliche Subtypbeziehung zum Typ *Ok* fordert. Bei diesem handelt es sich um den Supertyp aller in TL definierbaren Typen. Der Parameter *format* gibt Steuerzeichen an, welche die Konvertierungsarten der Parameter des Aktualwertes von *function* bestimmen.

Kapitel 4

Plattformunabhängige Realisierung des Tycoon-Systems

Für einen wirtschaftlichen Einsatz von Informationssystemen sind hohe Freiheitsgrade bei der Wahl der einzusetzenden Hardware und Betriebssysteme erforderlich. Dies gilt im besonderen Maße für kommunizierende verteilte Anwendungen mit mobilen Objekten. Es ist in diesem Fall nicht hinreichend, wenn bestimmte Programme auf unterschiedlichen Plattformen lauffähig sind. Zusätzlich muß gewährleistet sein, daß zwischen unterschiedlichen Plattformen übertragene Objekte entsprechend ihrer jeweils vorgesehenen Semantik verwendbar sind. Persistente Objektsysteme, die als Entwicklungsplattform für Anwendungen in heterogenen Netzen eingesetzt werden, müssen demnach zum einen möglichst portabel sein und zum anderen über einen Mechanismus zum plattformunabhängigen Objektaustausch verfügen.

Das persistente Tycoon-Objektsystem wird den genannten Anforderungen durch dedizierte Maßnahmen, die in den folgenden Abschnitten beschriebenen werden, in sehr hohem Maße gerecht. Aufgrund dieser Beiträge kann die gesamte Problematik von Plattformabhängigkeiten dann im weiteren Verlauf der Arbeit als geklärt betrachtet werden.

Für die Realisierung plattformunabhängigen Objektaustausches sind vor allem die Entwurfsentscheidungen in folgenden drei Kategorien von Bedeutung:

Netzwerkprotokolle: Aufgrund der dominierenden internationalen Verbreitung von TCP/IP (insbesondere der hohen Verfügbarkeit im akademischen Bereich) ist dieses Netzwerkprotokoll in jedem Fall zu unterstützen. Gleichzeitig genügt es, sich zum Zwecke der Erschaffung einer gemäß der Zielsetzung der Arbeit *prototypischen* Programmierinfrastruktur auf einen einzigen derartigen Standard zu beschränken.

Software-Schnittstellen: Für die Kommunikationsprogrammierung mit TCP/IP existieren nicht wenige komfortable Schnittstellen, die sowohl von Einzelheiten des Protokolls als auch von allzu spezifischen Hardware-Eigenheiten abstrahieren. In Abschnitt 4.2 wird auf der Grundlage von Sockets die bewußt schmale, plattformunabhängige Implementationsbasis aller im weiteren Verlauf der Arbeit entwickelten Kommunikationsmechanismen erarbeitet.

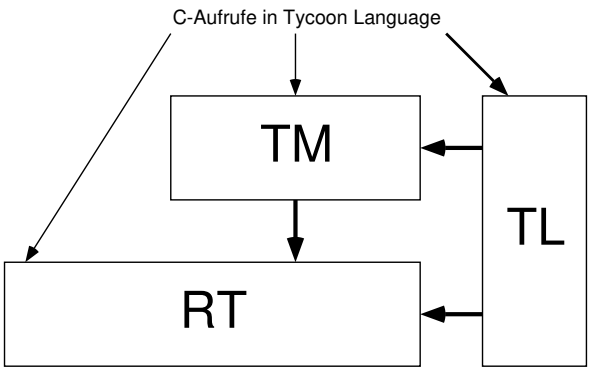

Abbildung 4.1: Die auf Portabilität ausgerichtete Architektur des Tycoon-Laufzeitsystems

Externe Datenrepräsentationen: In Abschnitt 4.3 wird eine einheitliche externe Datenrepräsentation für persistente Objekte beschrieben, die mit Hilfe recht unkomplizierter Algorithmen auf jeder relevanten Plattform erzeugt und interpretiert werden kann. Das Linearisierungsverfahren ist an dedizierten Stellen mit Funktionen parametrisiert, die es erlauben, nachträglich und ohne den eigentlichen Algorithmus zu ändern, spezielle Datenformate, Bindungstechniken (siehe Kapitel 7) und zusätzliche Systemfunktionalität wie z.B. Autorisierung (siehe Abschnitt 8.3) zu implementieren.

Es ist zu beachten, daß eine einzige *polymorphe* Funktion die Linearisierung *beliebiger* TL-Objekte leistet. Beim Einsatz als Marshalling-Funktion ist daher keine Codegenerierung erforderlich. Gleiches gilt für das Unmarshalling.

Im folgenden Abschnitt 4.1 wird die auf Portabilität ausgerichtete Architektur des Tycoon-Laufzeitsystems beschrieben. In Folge dieser portierungsfreundlichen Gestaltung konnte Tycoon mit relativ geringem Aufwand auf bislang 11 verschiedene Plattformen[1] portiert werden.

4.1 Die Architektur des portablen Tycoon-Laufzeitsystems

C ist bei weitem die verfügbarste zur portablen Systemprogrammierung geeignete Programmiersprache.[2] Daher ist das Tycoon-Laufzeitsystem in *C* geschrieben. Aufgrund wesentlich verbesserter statischer Typsicherheit wurde ANSI *C* gegenüber K&R *C* der Vorzug gegeben. Es wird keine weitere Implementationssprache benötigt.[3]

[1] AIX, Amiga OS, Linux, Macintosh 68k, NeXTSTEP, OS/2, Power Macintosh, Solaris, SunOS 4, Windows 95, Windows NT.

[2] Diese Tatsache wird in [Mathiske 92] (im Zusammenhang portabler Codegenerierung in Kapitel 5) ausführlich dargelegt. Die heutige Situation entspricht noch der damaligen Betrachtung. Allerdings dürfte sich *C++* bei anhaltender Weiterverbreitung und bei Durchsetzung des bereits existierenden Standards recht bald zur Alternative entwickeln.

[3] Lediglich im Fall von SunOS 4 wird eine Assembler-Anweisung eingesetzt. Sie kompensiert einen „Mangel" der *setjmp*-Funktion in der *C*-Standardbibliothek von SunOS 4. Da sie als Inline-Anweisung in den *C*-Quelltext integriert ist, genügt nach wie vor ein einziges Compiler-System zur Übersetzung des Tycoon-Laufzeitsystems.

Plattform-, bzw. Compiler-Abhängigkeiten können bei disziplinierter Anwendung von *C* fast vollständig auf die Auswahl von Bibliotheksfunktionen reduziert werden. Solche Unterschiede lassen sich dann mit Hilfe des Präprozessors durch bedingte Compilation auflösen. Entscheidend für den Portierungsaufwand ist die Zusammenfassung inhaltlich zusammenhängender Fallunterscheidungen. Folglich ist dies dann auch ein vorrangiges Modularisierungskriterium.

Die Architektur des Tycoon-Laufzeitsystems ist stark von den Gesichtspunkten der Portabilität geprägt. Es werden drei Komponenten unterschieden:

RT („RunTime“ support): Diese Komponente schottet den gesamten Rest des Laufzeitsystems (und damit auch das gesamte Tycoon-System) von Betriebssystemspezifika ab. Sie stellt in plattformunabhängiger Weise Funktionen für Textein- und Ausgabe, Dateizugriffe und Speicherverwaltung zur Verfügung. Außerdem erweitert sie die Grundfunktionalität von *C* um dynamische Ausnahmebehandlung und Coroutinen. Viele Funktionen des RT werden direkt auf Betriebssystemaufrufe oder Standardfunktionen (z.B. *malloc*) abgebildet. Auf einigen Plattformen muß das gewünschte Verhalten nachgebildet werden. Derartige Anpassungen bleiben durch das RT gekapselt und auf definierte, leicht lokalisierbare Stellen beschränkt. Das RT ist nicht Tycoon-spezifisch angelegt. Es kann prinzipiell für jede Art von *C*-Programmen, die unter dem Gesichtspunkt der Portabilität zu entwickeln sind, verwendet werden.

TM („Tycoon Machine“ functionality): Dies ist der funktionale Hauptteil des Laufzeitsystems. Sein zentrales Element ist der Byte-Code-Interpreter, der auch *Tycoon Virtual Machine* genannt wird. Die weiteren Module der TM implementieren komplexe Byte-Code-Operationen, Multi-Threading und die Transaktionsverwaltung.

TL („Tycoon Language“ extensions): Hierbei handelt es sich nicht um eine geschlossene Komponente, sondern um eine lose Sammlung von Modulen, deren exportierte Funktionen von TL-Standardbibliotheken über Tycoons *C*-Programmierschnittstelle aufgerufen werden. Es ist ausschließlich Funktionalität enthalten, die für einen *Bootstrap* des Tycoon-Systems nicht benötigt wird, wie z.B.: Thread-Steuerung, typunsichere TL-Operationen, *C*-Callbacks, Symbolverwaltung für dynamisches Linken (siehe Abschnitt 7.4).

Abbildung 4.1 stellt den Zusammenhang der drei Komponenten des Laufzeitsystems dar: die TM-Schicht baut auf der RT-Schicht auf; weder TM noch RT verwenden TL; TL greift sowohl auf RT als auch auf TM zu. Von der Sprache TL aus werden via *C*-Programmierschnittstelle Funktionen aller drei Komponenten des Laufzeitsystems aufgerufen, allerdings vornehmlich Funktionen der Komponente TL.

Die Module der TL-Komponente könnten im Prinzip auch in vom eigentlichen Laufzeitsystem getrennten Bibliotheken angesiedelt sein. Ihre enge statische Ankopplung ist jedoch beabsichtigt. Durch diese rigide Maßnahme wird Tycoon-Systemprogrammierern implizit nahegelegt, die TL-Standardbibliothek relativ früh zu portieren. Diese Bibliothek wird nämlich auch für Testprogramme benötigt, die dazu dienen, den Erfolg einer Portierung zu verifizieren.

Funktionsname	*C*-Original	Maßnahme
socket_socket	socket	Parameteroptionen vereinheitlicht
socket_close	close	WinSockets: *closeSocket*
socket_bind	bind	Adreßformat vereinheitlicht
socket_connect	connect	Adreßformat vereinheitlicht
socket_listen	listen	Blockieren verhindert
socket_accept	accept	Blockieren verhindert
socket_write	write	WinSockets: Nachbildung mit *send*
socket_read	read	WinSockets: Nachbildung mit *recv*
socket_is_selected	select	Vereinfachter Aufruf

Abbildung 4.2: Funktionen der plattformunabhängigen *C*-Schnittstelle für Sockets

4.2 Plattformunabhängige Socket-Kommunikation

Um eine größtmögliche Plattformunabhängigkeit des Tycoon-Objektsystems auch bezüglich der Netzwerkkommunikation zu gewährleisten, ist eine *plattformunabhängige* Programmierschnittstelle zur Datenkommunikation auf geeignetem Abstraktionsniveau erforderlich. In einer ersten Arbeit [Johannisson 95] auf diesem Gebiet wurde der Ansatz verfolgt, auf der Basis unterschiedlicher *C*-Middleware (BSD-Sockets, ONC-RPC, DCE) alternative Implementationen für entfernte Prozeduraufrufe (RPCs: *Remote Procedure Calls*) in TL zu realisieren. In dieser Hinsicht wurde eine sehr weitgehende Portabilität der Tycoon-Kommunikationsmechanismen erreicht.

Es hat sich jedoch herausgestellt, daß es relativ umständlich ist, RPCs auf der Basis von RPCs zu implementieren. Die Verwendung von Sockets ist auf dieser Systemebene erheblich unkomplizierter.

Ferner hat sich im Zuge der zahlreichen Portierungen von Tycoon herausgestellt, daß Sockets zur Zeit verbreiteter sind als ONC-RPCs und DCE. Sie werden naturgemäß auch in Zukunft mindestens so häufig auftreten wie ONC-RPCs, weil deren Implementation auf Sockets basiert. Es kann an dieser Stelle nicht vorhergesagt werden, ob DCE noch aufholen wird.

Während DCE stets uniform auftritt, variieren die Socket-Programmierschnittstellen unterschiedlicher Plattformen meist ein wenig. Der Aufwand, DCE überhaupt einzusetzen, ist jedoch so beträchtlich, daß Sockets der Vorzug zu geben ist.

Die Socket-Schnittstellen auf UNIX-Plattformen weisen nur marginale Unterschiede auf. Die auf dem Macintosh in der Public Domain verfügbare GUSI-Bibliothek [Neeracher 95] imitiert die UNIX-Versionen nahezu perfekt. Nennenswerte Unterschiede treten bei WinSockets (unter Windows NT oder Windows 95) auf. Diese Art „Sockets“ ist im Gegensatz zu BSD-Sockets unter UNIX nicht mit Datei-Handles kompatibel. So muß z.B. anstelle von *close* die WinSocket-spezifische Funktion *closeSocket* verwendet werden, um die Benutzung eines Socket abzuschließen. Vor allem ist es nicht möglich, die Dateifunktionen *read* und*write* für Sockets einzusetzen. Ihre Funktionsweise läßt sich jedoch mit Hilfe von *recv* und *send* leicht nachbilden. Im Fall von *write* wird wie folgt eine Funktion definiert, die aufgrund bedingter Compilation auf jeder der unterstützten Plattformen wie gewünscht funktioniert.

```
int socket_write(int sock, char *buffer, int size)
{
#  ifdef rtstd_LIB_WINDOWS
#    define nBLOCK 4096
     char* p = buffer;
     int s = size;

     while (s > nBLOCK) {
       send(sock, p, nBLOCK, 0);
       p = p + nBLOCK;
       s = s - nBLOCK;
     }
     send(sock, p, s, 0);
     return size;
#  else
     return write(sock, buffer, size);
#  endif
}
```

Wenn Code für ein Windows-Betriebssystem erzeugt wird, dann ist die Konstante *rtstd_LIB_WINDOWS* definiert und die Nachbildung mit *send* tritt in Kraft. In allen anderen Fällen wird einfach die Originalversion von *write* aufgerufen. Die Funktion *read* wird analog durch ein *socket_read* ummantelt.

In Tabelle 4.2 sind die Funktionen der plattformunabhängigen Socket-Schnittstelle aufgeführt, die jeweils einer *C*-Originalfunktion entsprechen (vgl. Abschnitt 2.1.1). Um sie zu benutzen, insbesondere um Internet-Adressen in passenden *C*-Datenstrukturen zu erzeugen, werden außerdem einige Funktionen zur Adreßmanipulation benötigt, wie z.B. *gethostname*, *gethostbyname*, *gethostbyaddr*, *htonl* und *ntohl* [Corbin 91]. Während sich die Benennung und die eigentliche Funktionsweise dieser Funktionen von System zu System nicht unterscheidet, treten in WinSockets leicht abweichende Datentypen der Parameter auf. Dieses Problem wird gelöst, indem die Gegebenheiten unter UNIX als Standard festgelegt und WinSockets-Funktionen ggf. zur Angleichung ihrer Parametertypen ummantelt werden.

Die nun für die vorgesehenen Zwecke vollständige plattformunabhängige *C*-Schnittstelle für Sockets wird mit Hilfe der in Abschnitt 3.5 vorgestellten *bind*-Anweisung nach TL „geliftet“. Das prinzipielle Verfahren ist an folgendem Beispiel ersichtlich.

```
let socketLib = runtimeCore.dynamicLibraryName("commenv" "socket")

let CWrite = bind(:Fun(sock :T  data :word.T  size :Int):Int
                  socketLib  "socket_write" "wwii")

let write(sock :T data :word.T size :Int) :Ok =
  begin
    let result = CWrite(sock  data  size)
    if result != size orif result < 0 then raise error end
  end
```

Dieser Codeauszug entstammt dem Modul *socket.tm* der Tycoon-Standardbibliothek *commenv*, deren Aufgabe es ist, Kommunikationsprimitive zu implementieren. Im Verzeichnis dieser Bibliothek ist auch eine Datei namens *"socket.c"* untergebracht, welche die oben beschriebene *C*-seitige Funktionalität realisiert. In der ersten Zeile des Beispiels wird die aus dieser *C*-Datei hervorgehende *Shared Library* angesprochen (vgl. Abschnitt 3.5). Die nächste Anweisung bindet die TL-Funktion *CWrite* an die *C*-Funktion *socket_write*. Die anschließend definierte TL-Funktion *write* wird vom Modul exportiert. Dieses ist noch zusätzlich in seiner Verwendung vereinfacht, indem das umständliche Abfragen von Fehlercodes auf eine dynamische Ausnahme abbildet wird.

Eine wichtige Anforderung ist die Berücksichtigung der Mobilität kommunizierender Software-Komponenten (vgl. Abschnitt 5.1.4). Aus diesem Grunde dürfen systemabhängige Konstanten, die in *C* durch Makros abgebildet werden, *nicht* als Werte gebunden werden. Stattdessen werden sie am jeweiligen Standort stets wieder neu vom Tycoon-Laufzeitsystem bzw. der darin enthaltenen Socket-Bibliothek erfragt. Die betroffenen Makros sind zu diesem Zweck auf TL-Funktionen abgebildet. Zum Beispiel liefert die Funktion *SOCK_STREAM* den ganzzahligen Wert des gleichnamigen *C*-Makros. Die für TL zum Teil ungewöhnlichen Schreibweisen der Bezeichner in *Socket.ti* wurden absichtlich gewählt, um das Wiederfinden korrespondierender Strukturen in Dokumentationen zu *C*-Sockets erleichtern.

Die Schnittstelle *Socket.ti* des hier beschriebenen Socket-Moduls ist in Anhang B.1 aufgeführt. In Bezug auf externe Kommunikations-Software ist sie die *vollständige* Implementationsbasis aller im folgenden Kapitel beschriebenen Konstrukte. Aufgrund dieser bewußt schmal gestalteten Schnittstelle sind Portierungen mit sehr geringem Aufwand verbunden. Bisher steht das Socket-Modul für SunOS4 und Solaris auf Sparc-Rechnern, MacOS auf PowerPC und 68k-Prozessoren sowie Linux und Windows-NT auf x86-Prozessoren bereit [Göllnitz 96]. Portierungen für OS/2 auf x86-Rechnern und die BSD-Variante NeXTSTEP auf NeXT-68k-Rechnern stehen kurz bevor.

4.3 Die plattformunabhängige lineare Datenrepräsentation Txr

Ähnlich anderer moderner Sprachen wie Amber [Cardelli 86], Quest [Cardelli 89], Standard ML [Milner et al. 90] und Napier88 [Munro 93], stellt Tycoon einen Mechanismus bereit, der dazu dient, tiefe Kopien beliebig komplexer Objekte in Dateien abzulegen. Auch in einer persistenten Sprache wie Tycoon gibt es zahlreiche Anwendungen für datenstromorientierten Repräsentationen: der Austausch compilierter Modulschnittstellen, Module und Bibliotheksbeschreibungen zwischen Objektspeichern, Backups, kompakte Speicherung verschiedener Datenversionen, Datenbank-Logging, die Erzeugung eigenständiger Tycoon-Programme in sogenannten „Boot"-Dateien [Mathiske et al. 93]. Insbesondere zum Zwecke effektiver Datenkommunikation über Netzwerke ist der Linearisierungsmechanismus des Tycoon-Systems weiterentwickelt worden:

- Die Ausgabe kann nicht nur auf Dateien, sondern auf beliebige Byte-Ströme erfolgen, wie z.B. Byte-Sequenzen im Objektspeicher und Kommunikationskanäle.
- Linearisierungen sind zwischen heterogenen Hardware- und Betriebssystemarchitekturen austauschbar, wobei automatisch recht einfache und dementsprechend effiziente Konvertierungen erfolgen.
- Optional können lineare Repräsentationen mit dynamischer Typinformation versehen werden, um typunsichere Zugriffe zu vermeiden.
- Die Linearisierung unterliegt keinerlei Typbeschränkung. Es können sowohl Daten als auch Code und sogar Threads linearisiert werden.

Die in den folgenden Abschnitten näher beschriebene externe Repräsentationsform TXR (*Tycoon eXternal Representation*) hat mit ASN.1 (vgl. Abschnitt 2.1.3) gemeinsam, daß ausreichende Typinformationen bereitstehen, um eine *uninformierte* Delinearisierung leisten zu können: der Empfänger von TXR-Daten muß den ursprünglichen Typ des linearisierten Objektes nicht im Vorwege bereits kennen, um es zu rekonstruieren. Vielmehr genügt eine einzige *polymorphe* Delinearisierungsfunktion.

In TXR werden allerdings keine hochsprachlichen Typen erfaßt, sondern lediglich die primitiven Fallunterscheidungen des TSP. Dadurch ist die Komplexität der TXR besonders gering.

4.3.1 Programmierschnittstellen

Die Algorithmen zur Linearisierung und zur Rückgewinnung von Objektgraphen sind in *C* implementiert. Ihre Schnittstelle ein Teil des TSP, jedoch ist ihre Implementation in einem separaten Modul *tsplinio*) oberhalb des übrigen TSP angesiedelt. Dieses in Abbildung 4.3 dargestellte Modulschema gewährleistet, daß der Tycoon-Objektaustausch nur ein einziges Mal implementiert werden muß, woraufhin er vollkommen unabhängig von Objektspeicherarchitekturen arbeitet.

In TL stehen die bewußten *C*-Funktionen in Form von Callbacks (d.h. in diesem Fall *C*-Aufrufe) in das Tycoon-Laufzeitsystem zur Verfügung. Folgende Funktion des Moduls *unsafe* linearisiert den durch ihren Parameter *value* angegebenen Wert, der beliebigen Typs sein kann, in einem neu angelegten Hauptspeicherblock und gibt als Resultat dessen Adresse zurück.

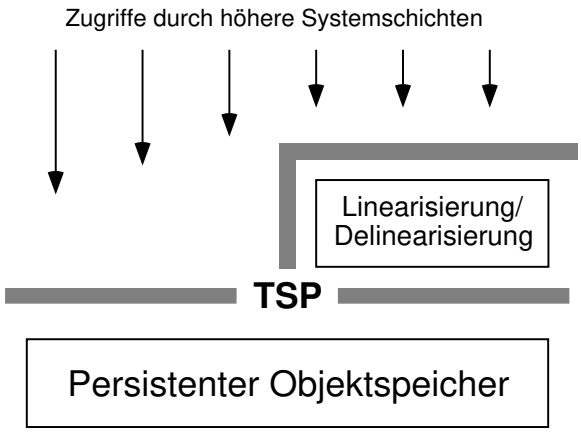

Abbildung 4.3: Die objektspeicherunabhängige Ansiedlung des Linearisierungs- und Delinearisierungsmoduls

externToNewBuffer(V <:Ok value :V **var** *nBytes :Int) :word.T*

Die Ausgabevariable *nBytes* gibt die Länge des Hauptspeicherblockes in Bytes an.

Das Pendant zu obiger Funktion rekonstruiert anhand einer in einem Speicherblock (*buffer*) bekannter Länge (*nBufferBytes*) vorliegenden externen Repräsentation einen Objektgraphen und gibt diesen als frei wählbar typisierten TL-Wert zurück.

internFromBuffer(buffer :word.T nBufferBytes :Int V <:Ok) :V

Zur Veranschaulichung der Linearisierung werde diese auf den Wert a aus Abschnitt 3.4 angewendet:

let **var** *nBytes = 0*

let *buffer = unsafe.externToNewBuffer(a nBytes)*

Wie in Abbildung 4.4 ersichtlich, wird in der externen Repräsentation die gesamte transitive referentielle Hülle von a wiedergegeben, indem Objektspeicherreferenzen im Datenstrom durch durch Indizes repräsentiert werden. Die Speichergröße dieses linearen Datenstroms beträgt ca. 30 KB.

Folgende Anweisung rekonstruiert eine isomorphe[4] Kopie des linearisierten Objektgraphen im Objektspeicher.

let *aCopy = unsafe.internFromBuffer(buffer nBytes :A)*

Die hier aufgeführten Funktionen sind zwar per se *typunsicher*, aber ihre typsichere Verwendung kann durch Kapselung in höheren Modulschichten der Tycoon-Kommunikationsbibliothek gewährleistet werden (siehe Abschnitt 5.2.2.2).

[4]Zyklen und Teilungsverhältnisse bleiben erhalten.

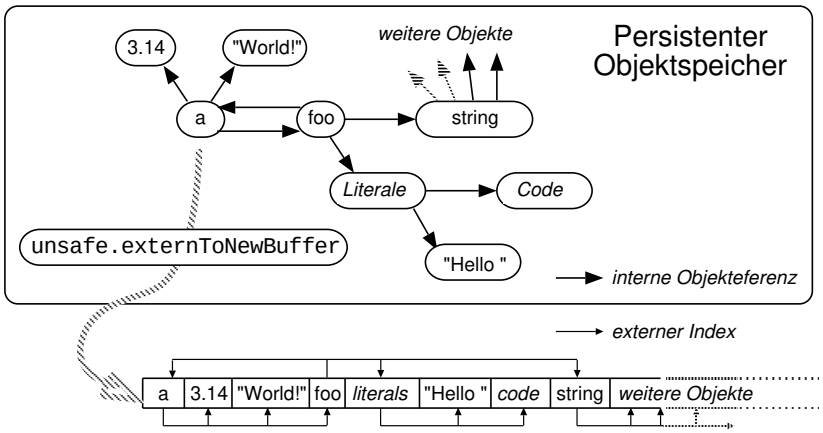

Abbildung 4.4: Linearisierung der transitiven referentiellen Hülle eines komplexen Objektes

4.3.2 Die Grundalgorithmen des polymorphen Objektaustausches

Die Linearisierung von Objektgraphen erfolgt in zwei Phasen. In der ersten Phase wird die transitive Hülle des primär zu übertragenden Objektes in einer Hash-Tabelle erfaßt, wobei mit jeder Objektreferenz ein jeweils eigener Index assoziiert wird. In der zweiten Phase werden dann über die Hash-Tabelle iterierend Repräsentationen der eingetragenen Objekte sequentiell ausgegeben. Die Objektinhalte werden dabei in eine kanonische Form gebracht:

- ▷ Jedem Objekt wird sein Index vorangestellt.
- ▷ Sämtliche Objektreferenzen werden durch entsprechende Indizes aus der Hash-Tabelle ersetzt.

Abbildung 4.5 stellt die wichtigsten Einzelheiten des Linearisierungsverfahrens am Beispiel des in Abschnitt 3.4 definierten Wertes x dar.

Um Plattformunabhängigkeit zu gewährleisten, werden außerdem folgende Maßnahmen getroffen:

- ▷ Fließkommazahlen werden in Zeichenketten umgewandelt.
- ▷ Für die Reihenfolge der Bytes innerhalb von 4-Byte-Worten gilt das Anordnungsschema *big endian*. Auf Hardware-Plattformen, für die *little endian* gilt (z.B. IBM-kompatible PCs mit Intel-CPUs), werden automatisch entsprechende lokale Umsortierungen vorgenommen.

Das Ergebnis der Linearisierung ist daher jeweils ein auf beliebigen Plattformen eindeutig interpretierbarer Byte-Strom.

Die Delinearisierung erfolgt ebenfalls in zwei Phasen. In der ersten Phase werden alle Objekte eines zu verarbeitenden Datenstroms sequentiell eingelesen. Im Zuge dessen wird ggf. die Byte-Reihenfolge der aktuellen Hardware angepaßt und aus dafür vorgesehenen Zeichenketten

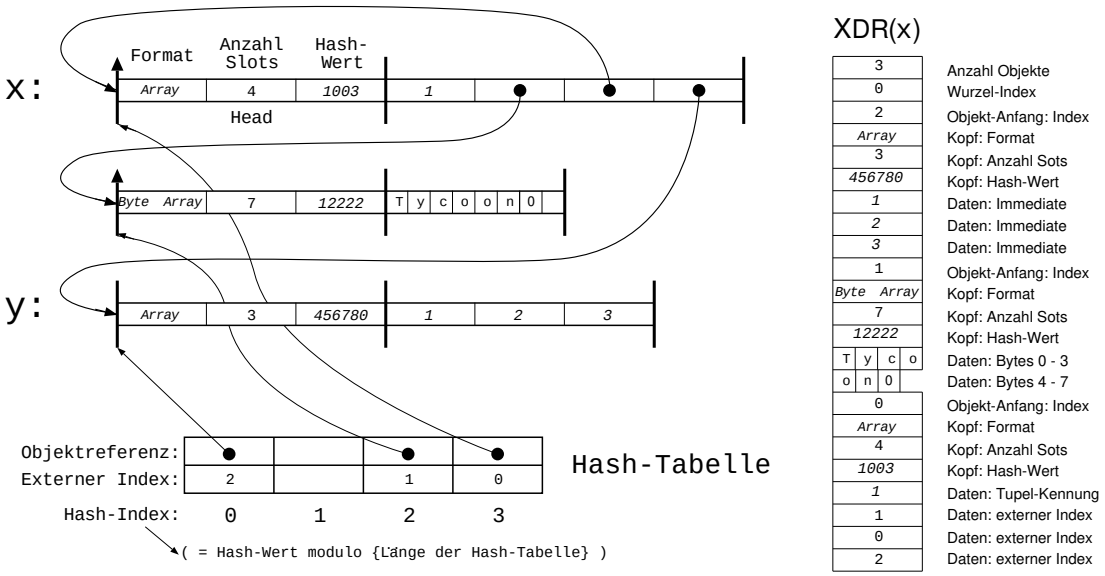

Abbildung 4.5: Die TXR-Linearisierung im Detail

werden wieder Fließkommazahlen erstellt. Alle neu entstehenden Objektreferenzen werden in einer Tabelle erfaßt, wobei die Position eines Objekteintrages in der Tabelle jeweils dem während der Linearisierung vergebenen Index entspricht. In der zweiten Phase wird sequentiell über die Tabelle iteriert, um sämtliche in Form von Indizes in den Objektinhalten auftretenden Objektreferenzen zu relozieren. Um einen Index aufzulösen wird einfach der entsprechende Tabelleneintrag aufgesucht.

Bei beiden Algorithmen werden die benötigten Hilfsdatenstrukturen (Hash-Tabelle, Objekt-Tabelle) im Objektspeicher angelegt. Dadurch steht genügend Speicherplatz zur Verfügung, um große Mengen linearisierter Daten zu verarbeiten. Außerdem sind alle Zustandsvariablen in die Garbage-Collection integriert.

Wie in Abschnitt 4.1 ausgeführt wurde, verwendet Tycoon als ausführbares Code-Format einen Byte-Code, der von einer abstrakten virtuellen Maschine interpretiert wird und damit plattformunabhängig ist. Da gemäß Abschnitt 3.4 Funktionen und Threads wie alle anderen TL-Werte auch durch uniforme Objekte im Objektspeicher repräsentiert werden, lassen sie sich auf die gleiche Weise wie diese linearisieren. Auch die Repräsentation von Threads (siehe Abschnitt 6.1.2) ist plattformunabhängig.

4.3.3 Parametrische Modifikationen des Objektaustausches

Typischerweise sind gerade Funktionen und Threads komplexe Gebilde mit einer sehr großen transitiven referentiellen Hülle. In vielen Fällen[5] ist eine rein kopierbasierte Übertragung nicht wünschenswert bzw. aufgrund exzessiven Zeit- und Speicherplatzbedarfes überhaupt nicht realisierbar.

In die im vorigen Abschnitt vorgestellten Algortihmen können nachträglich Spezialtechniken wie dynamisches Linken (siehe Abschnitt 7.4), transparente Erzeugung von Netzwerkreferenzen (siehe Abschnitt 2.3.1) und Autorisierung (siehe Abschnitt 8.3) integriert werden, indem an dedizierten Stellen benutzerdefinierte Funktionen aufgerufen werden. Diese können ihrerseits Teilaufgaben an die vorhandene Implementation zurückdelegieren, was insbesondere der Wahrung der Plattformunabhängigkeit dient.

Der Linearisierungsalgorithmus wird mit zwei Funktionsparametern aufgerufen:

writeHandler: Diese Funktion ist für die Ausgabe und damit für die Gestaltung der externen Repräsentation einzelner Objekte zuständig. Sie wird in der zweiten Phase pro Objekt einmal aufgerufen. Das Tycoon-Laufzeitsystem übergibt hier eine Funktion, die anhand des vorliegenden Objektformates entscheidet, ob an die kanonische Funktion *defaultWriteHandler* zurückdelegiert wird.

Fließkommazahlen werden erkannt und wie bereits erwähnt als Zeichenketten ausgegeben. Die jeweils vorangehende Ausgabe der Attribute des Objektkopfes geschieht allerdings in kanonischer Weise mit Hilfe der vordefinierten Funktion *externAttributes*.

Objekte, die später dynamisch gelinkt (siehe Abschnitt 7.4) werden sollen, werden ebenfalls anhand ihres Formates erkannt und gesondert behandelt: an ihrer Stelle wird einfach ein anderes Objekt, das jeweilige Linksymbol, ausgegeben. Dies übernimmt wiederum die Funktion *defaultWriteHandler*.

pruneHandler: Diese Funktion kontrolliert die Größe des zu übertragenen Teilgraphen. Da Objekte, die für dynamisches Linken vorgesehen sind, in der zweiten Phase ersetzt werden, muß die Rekursion der ersten Phase hier abbrechen. Um diese Bedingung zu testen, wird für jedes transitiv erreichte Objekt das Prädikat *pruneHandler* aufgerufen.

Der Delinearisierungsalgorithmus ist analog parametrisiert:

readHandler: Diese Funktion ist für das Einlesen und erzeugen einzelner Objekte zuständig. Sie wird nach dem Einlesen des Objektkopfes aufgerufen und erhält dessen Attribute als Argumente. Anhand dieser wird entschieden, wie der zu lesende Objektinhalt zu interpretieren ist.

Zeichenketten für Fließkommazahlen werden in solche umgewandelt.

Linksymbole werden durch lokale Objekte ersetzt.

Alle anderen Objekte werden durch die Standardfunktion *defaultReadHandler* erzeugt.

pruneHandler: Da dynamisch gelinkte Objekte lokalen Ursprungs sind, werden sie beim Relozieren in der zweiten Phase übersprungen. Das vom Laufzeitsystem gelieferte Prädikat *pruneHandler* erkennt sie anhand ihres Formates.

[5] Dies wird in Kapitel 7 näher untersucht.

Durch Erweiterungen der Funktionen *writeHandler* und *readHandler* können zusätzliche komplexe Basisdatentypen (z.B. Multimediadatentypen wie Bild, Ton oder Video) dergestalt in das Tycoon-System integriert werden, daß sie plattformunabhängig zwischen Objektspeichern austauschbar sind.

Mit obigen Parametern des Delinearisierungsalgorithmus können folgende zwei Dienste noch nicht ausreichend unterstützt werden

Autorisierung: Eine Autorisierung übertragener Objekte kann erst erfolgen, wenn diese vollständig reloziert sind. Dies ist erst nach Abschluß der zweiten Phase der Fall.

automatische Replikation: Bei dieser in Abschnitt 7.5 beschriebenen Technik ist es erforderlich, nachträglich Objekte in einem bereits vollständig übertragenen Graphen zu ersetzen. Dazu müssen Objektreferenzen innerhalb des Graphen geändert werden.

Aus diesen Gründen ist dem Delinearisierungsalgorithmus noch eine dritte Phase nachgeschaltet, in der zwei weitere Funktionsparameter angewandt werden:

updateHandler: Diese Funktion wird einmal pro Objekt aufgerufen. Sie bietet Gelegenheit zu einer gründlichen Untersuchung und Manipulation der Objektinhalte, etwa zum Zwecke der Autorisierung. Aus dieser lokalen Sicht heraus kann das jeweilige Objekt selbst jedoch nicht ausgetauscht werden.

slotHandler: Diese Funktion wird einmal pro Slot in jedem Vektorobjekt aufgerufen, d.h. einmal pro Kante im übertragenen Graphen. Durch Änderung der vorgefundenen Objektreferenzen können bestimmte Objekte im Graphen komplett gegen andere ausgetauscht werden. Zum Beispiel werden auf diese Weise bei automatischer Replikation (siehe Abschnitt 7.5) Linksymbole durch nachträglich übertragene Objekte ersetzt.

Zukünftige multimediale Anwendungen werden höhere Anforderungen stellen und es wird zunehmend Weitverkehrsnetze mit hoher Bandbreite geben. In einer Anschlußarbeit an diese Dissertation sollte daher eine segmentweise Verschränkung der Implementationen der Socket-Kommunikation und der Linearisierung erfolgen, sodaß nur eine beschränkte Puffergröße erforderlich ist.

Kapitel 5

Mobilitätsorientierte entfernte Funktionsaufrufe

Dieses Kapitel stellt einen RPC-Mechanismus vor, der im folgenden Sinn *mobilitätsorientiert* ist:

- Tycoon-Funktionen beliebigen Typs können als entfernte Funktionen (für RPCs) eingesetzt werden. Insbesondere können entfernte Funktionen polymorph sein und sie können rekursive Datenstrukturen, Funktionsabschlüsse und Threads als Argumente und Rückgabewerte verwenden. Damit ist die Mobilität von Tycoon-Werten *typunabhängig*.
- Die Auswahl und der Einsatz entfernter RPC-Dienste geschehen in einem *einheitlichen* sprachlichen Rahmen. Dabei steht für die algorithmische Bewertung von Dienstattributen der gesamte Sprachumfang zur Verfügung und für dynamische Typvergleiche wird keine zusätzliche Sprache (insbesondere keine *Interface Definition Language*) benötigt.
- Client-Bindungen an entfernte Funktionen sind sowohl mobil als auch persistent.
- Server-Prozesse können im Netzwerk verlagert werden, ohne daß Client-Bindungen deshalb explizit aktualisiert werden müssen.
- Die Bindung an entfernte Funktionen erfolgt vollkommen *dynamisch* zur Laufzeit. Es ist weder erforderlich, Stub-Generatoren zu aktivieren noch Quelltexte zu compilieren oder den Anwendungscode neu zu linken.
- Das volle Leistungsspektrum ist plattformunabhängig.

Diese Beiträge profitieren maßgeblich von den sprachlichen und architekturellen Vorgaben (vgl. Abschnitt 1.2) des Tycoon-Systems. Außer der im Rahmen dieser Arbeit erreichten Plattformunabhängigkeit spielen in diesem Kapitel besonders das polymorphe Typsystem höherer Ordnung, dynamische Typvergleiche, Funktionen höherer Ordnung und orthogonale Persistenz eine wichtige Rolle.

Im ersten Teil des Kapitels werden die genannten Eigenschaften entfernter Funktionsaufrufe näher erläutert und die aus ihnen resultierenden Möglichkeiten eleganter Programmierung vorgestellt. Der zweite Teil ist der Implementation der Systemtechniken gewidmet. Quantitative Performanzangaben entfernter Aufrufe sind aus Anhang F ersichtlich.

5.1 Entfernte Funktionsaufrufe im Tycoon-System

TL ist eine funktionale Sprache mit imperativen Konstrukten. Deshalb bieten sich synchrone RPCs (*Remote Procedure Calls*, zu deutsch „entfernte Prozeduraufrufe") [Nelson 81; Nelson, Birrell 84; Schill 92] als Paradigma für verteilte Programmierung an [Sloman, Kramer 88]. RPCs fassen Kommunikationsprimitive geringeren Abstraktionsgrades in einer handlichen Form zusammen und sie lassen sich nahtlos in die Wirtssprache einbetten. Dies hat folgende Vorteile:

- Programmierer können auf einem vertrauten Konzept der lokalen Programmierung aufbauen.
- Lokale Softwarearchitekturen können relativ einfach zu verteilten Softwarearchitekturen skaliert werden.
- Die generalisierte Abstraktion des Prozeduraufrufes interagiert hervorragend mit anderen Sprachkonzepten wie statische Typisierung, Modularisierung und Parametrisierung.

Im Sinne einer einheitlichen Benennung lokaler und entfernter Objekte gleichen Typs, die sich an den Gegebenheiten in TL orientiert, ist in der vorliegenden Arbeit mit Bezug auf RPCs von „entfernten Funktionen" und „entfernten Funktionsaufrufen" die Rede.

5.1.1 Begriffe und Anforderungen

Anstelle von RPCs wird in objektorientierten Sprachen der eng verwandte Mechanismus ROI (*Remote Object Invocation*) bevorzugt [Levy, Tempero 91]. Während zum Zwecke der Adressierung und Bindung diverse unterschiedliche Verfahren existieren, ist der Ablauf nach erfolgter vollständiger[1] Bindung bei allen Implementationen synchroner RPCs oder ROIs jedoch stets der gleiche (vgl. Abschnitt 2.1.2, Abbildung 2.4):

1. Es werden Argumentwerte über das Netzwerk zu einer Funktion in einem entfernten Adreßraum transferiert.
2. Die festgelegte Funktion wird mit den erhaltenen Argumenten aufgerufen.
3. Der Funktionsrückgabewert wird zur initiierenden Instanz des Aufrufes zurückgesendet.

In diesem Ablauf läßt sich stets eine sogenannte Client/Server-Beziehung identifizieren. Die initiierende Instanz wird „Client" genannt, die ausführende „Server". In der Literatur handelt es sich bei diesen Begriffen je nach Abstraktionsgrad der jeweiligen Betrachtung um Hosts, Prozesse, Programme, Programmteile oder näher bestimmte Objekte.[2] Um eine klare Ausdrucksweise zu gewährleisten, werden hier folgende (in Abbildung 5.1 illustrierte) Bedeutungen festgelegt:

[1] Während die aufzurufende Funktion bei RPCs fest vorgegeben ist, geht ihre Identität bei ROIs aus der späten Bindung (*late binding*) einer Methode hervor.

[2] Im vorliegenden Zusammenhang interessiert nur die abstrakte Client/Server-Beziehung, die sich in jedem einzelnen entfernten Aufruf wiederfindet. Konkrete anwendungsbezogene Aufgabenverteilungen spielen dabei keine Rolle. Deshalb wird auf Klassifikationen von Client/Server-Architekturen anhand der Schichtung unterschiedlicher System- und Anwendungskomponenten nicht eingegangen.

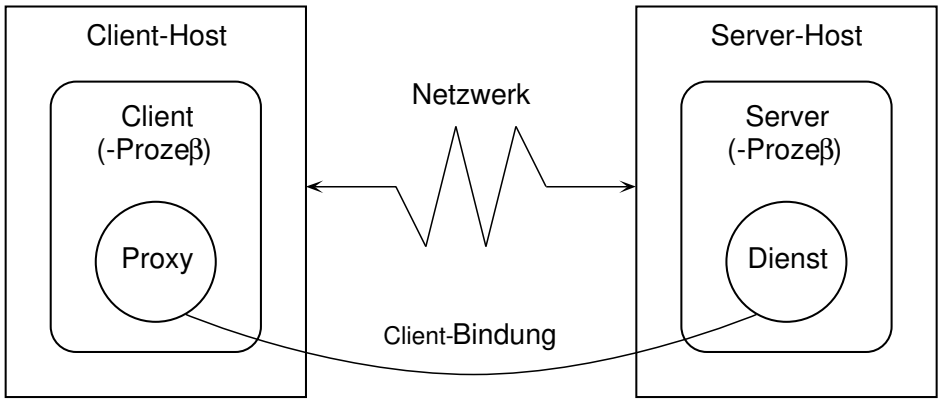

Abbildung 5.1: Grundbegriffe der RPC-Programmierung im Tycoon-System

Dienst: ein Objekt, das eine oder mehrere Funktionen für entfernte Aufrufe zur Verfügung stellt.

Server: ein Prozeß, der einen oder mehrere Dienste im Netz bekannt macht und ggf. die angebotenen Leistungen erbringt.

Client: ein Prozeß, in dem ein entfernter Dienst verwendet wird.

Proxy: die Repräsentation eines bestimmten entfernten Dienstes im Client, bzw. ein Objekt, das den Zugriff auf entfernte Funktionen für den Client bereitstellt (vgl. [Shapiro 86; Birell et al. 93b]).

Client-Bindung: Die logische Bindung eines Clients, bzw. eines Proxys, an einen bestimmten Dienst.

Ein isoliert betrachtetes Client/Server-Szenario nimmt im „Normalfall“ folgenden Verlauf:

1. Ein Server stellt einen oder mehrere Dienste für entfernte Clients zur Verfügung.
2. Ein Client geht eine (Client-) Bindung an einen bestimmten Dienst des Servers ein. Dabei entsteht ein Proxy.
3. Unter Verwendung des Proxy finden ein oder mehrere entfernte Funktionsaufrufe statt.
4. Dann wird die Bindung wieder aufgehoben, das Proxy somit „ungültig“.
5. Der Server und seine Dienste stehen entsprechend Schritt 1 bis auf weiteres für erneute Bindungen bereit.

Dieses recht allgemeine Schema bietet noch einigen Gestaltungsspielraum für die *Fehlersemantik* [Schill 92], die Eigenschaften von Client-Bindungen und den Vorgang der Dienstauswahl. Für die Fehlersemantik entfernter Aufrufe stehen mehrere Alternativen zur Auswahl [Nelson 81]:

maybe: Die Durchführung erfolgt soweit möglich, wobei die genaue Anzahl der verursachten Dienstaufrufe unbestimmt ist. Wenn von Interesse ist, ob oder wie oft ein Aufrufversuch erfolgreich war, kommt diese Variante nicht in Frage.

at-most-once: Es wird ein einziger Aufrufversuch unternommen, der fehlschlagen kann. In der Praxis ist in der Regel genau diese Variante gefordert. Zudem kann mit ihrer Hilfe jede der anderen simuliert werden.

at-least-once: Der Aufruf wird wiederholt, bis mindestens ein Aufruf erfolgreich ist. Bei zustandslosen Diensten ist dies sicherlich oft eine geeignete Strategie. Allerdings sind „interessante“ Dienste typischerweise zustandsbehaftet.

exactly-once: Es wird sichergestellt, daß genau ein Aufruf erfolgt. Diese Variante tritt in der Praxis lediglich in der Form auf, daß die gesamte Anwendung abstürzt, wenn die Middleware einen Fehler erkennt.

In der vorliegenden Arbeit wird die *at-most-once*-Semantik bevorzugt. In den weiteren Abschnitten dieses Kapitels wird deutlich, daß falls in einem speziellen Anwendungsfall einmal *maybe* oder *at-least-once* benötigt werden sollte, einige wenige TL-Zeilen genügen, um diese Varianten zu realisieren.

Folgende Eigenschaften von Client-Bindungen sind von zentraler Bedeutung für die Flexibilität und den Aufwand zur Erstellung verteilter Programme:

- ihre Toleranz gegenüber Prozeßbeendigungen, Abstürzen und Neustarts oder allgemeiner: ihre *Persistenz*.
- ihre Toleranz gegenüber topologischen Veränderungen oder anders ausgedrückt: die *Mobilität* von Anwendungskomponenten unter Berücksichtigung des Erhalts von Client-Bindungen.

In konventionellen RPC-Implementationen wie dem Sun-RPC und DCE (siehe Abschnitt 2.1.2) wird jede Bindung direkt mit einer Port-Verbindung identifiziert. *Langlebige* Client/Server-Bindungen sind mit solch fester Zuordnung von Socket-Verbindungen nicht realisierbar, denn Sockets sind flüchtig. Gleiches gilt für die Verlagerung von Programmen, da diese eine Prozeßbeendigung mit anschließendem Neustart impliziert. Zudem verändert sich die Netzwerkadresse. Letzteres ist wiederum auch bei der Migration kommunizierender Objekte zwischen Prozessen der Fall. Also sollten Sockets als *temporäre* Manifestationen von Client-Bindungen in persistenten Objektsystemen eingesetzt werden.

Nun bleibt noch zu klären, anhand welcher Attribute entfernte Dienste von Servern kenntlich gemacht und von Clients ausgewählt werden sollten, damit Client-Bindungen zustande kommen. Ein erster Ansatz besteht in einer (mehr oder weniger) einfachen Benennung entfernter Dienste (vgl. z.B. Sun-RPC [Corbin 91], DCE-RPC [Schill 93] oder Napier-RPC [Mira da Silva 95a; Mira da Silva 95b]). Diese Lösung ist jedoch recht speziell auf die Bedürfnisse relativ statischer Client/Server-Beziehungen zugeschnitten. Ein *fester* Attributtyp ist ohnehin nicht für jede beliebige Anwendung aussagekräftig genug.

Auf den ersten Blick erscheint es ideal, die Attributtypen für Dienste keinerlei Einschränkungen unterliegen zu lassen. Außerdem sollten Dienstattribute wahlweise dynamisch veränderbar sein können, um Clients eine Auswahl anhand aktueller Gegebenheiten zu ermöglichen.

Als einzige Einschränkung wird gefordert, den Diensttyp zu erfassen, denn durch die Überprüfung des Diensttypen vor der Dienstauswahl und damit noch *vor der Bindung* werden alle entfernten Aufrufe in TL *statisch* typsicher. Weitere „Standardisierungen" sollen bewußt nicht getroffen werden. So nützlich manche Standardattribute in der Regel auch sein mögen - sie erschweren nur unnötig die Programmentwicklung und Wartung, wenn sie nicht gebraucht werden.

Es bietet sich an, sowohl den Diensttyp als auch ein Attribut frei wählbaren (ggf. komplexen) Typs zur Grundlage der Dienstauswahl zu machen. Allerdings ist zu beachten, daß ein geordneter Attributvergleich die Konformität der jeweils gegebenen Attributtypen voraussetzt. Damit ergibt sich der Attributtyp als drittes Auswahlkriterium. Abbildung 5.2 faßt den resultierenden Ansatz, der maximale Ausdrucksfreiheit und maximale Typsicherheit in Einklang bringt, zusammen.

Diensttyp	Attributtyp	aktueller Attributwert

Abbildung 5.2: Die drei Kriterien der Dienstauswahl

Eine ähnliche Auffassung wird im *ODP-Trading* [ISO-ODP 95] vertreten. Abbildung 5.3 stellt die in der vorliegenden Arbeit verwendeten den korrespondierenden Begriffen des ODP-Trading gegenüber. Allerdings richten sich die Bemühungen der in diesem Bereich angesiedelten Projekte [Kosovic 95] auf sogenannte „offene Umgebungen" im Sinne von ODP, was zur Folge hat, daß stets sogenannte *Interface Definition Languages* (IDLs) zum Einsatz kommen. Diese sind weit davon entfernt, algorithmisch vollständige Sprachen mit elaboriertem Typsystem höherer Ordnung zu sein. Zudem sind die verwendeten Implementationssprachen (*C*, *C++*, Fortran, COBOL u.ä.) aufgrund ihrer mangelhaften Mobilitätsunterstützung nicht gerade prädestiniert für verteilte Programmierung.

Tycoon	**ODP-Trading**
Diensttyp	interface signature type
Attributtyp	property type
Attribut	property

Abbildung 5.3: Korrespondierende Begriffe im Tycoon-System und im ODP-Trading

Bei einer naiven Strategie zur Auswertung dieser Kriterien kann ein fataler Engpaß entstehen, wenn sehr viele entfernte Dienste auf einem Host angeboten werden. Denn der Einsatz effizienter Techniken (binäre Suche zum Vergleichen großer Mengen von Diensttypen setzt eine (zumindest annähernd) *vollständige* Ordnungsrelation als Vergleichskriterium voraus. Dieser Forderung genügt jedoch die Subtyprelation nicht; sie besitzt nur eine partielle Ordnung.

Die alternative Forderung einer Ordnung der Attributwerte würde die Menge der zulässigen Attributtypen einschränken. Zudem wäre eine Ordnung der Attributtypen vorauszusetzen.

Es existieren zwei Auswege aus dieser Situation:

- Ein oder mehrere Attribute global bekannten Typs werden als Standardattribute festgelegt. Zum Beispiel kann die Angabe der „Art“ eines Dienstes in Form einer Zeichenkette (*String*) gefordert sein. In diesem Fall sind von Dienstanbietern möglichst *diskriminierende* Bezeichnungen mit hohem *Wiedererkennungswert* zu verwenden. Da es sich hierbei um einen bei der Programmierung ohnehin ständig auftretenden Zielkonflikt handelt, erscheint diese Lösung praktikabel.

- Entweder bei den Diensttypen oder bei den Attributtypen wird die Subtyprelation durch eine effizienter einsetzbare Alternative ersetzt. Da der Diensttyp für den Gebrauch von Diensten relevant ist, der Attributtyp jedoch nicht, ist also bei letzterem anzusetzen. Wird für Attributtypen Typgleichheit gefordert, so kann Hashing auf ihre dynamischen Repräsentationen angewandt werden. Aufgrund einer sehr effizienten Implementation des Hashings beliebiger Objekte [Vaziri Pour 96] steht diese Lösung der voranstehenden in Bezug auf Performanz kaum nach.

Die Beschreibung von entfernten Tycoon-Diensten soll aufgrund der überlegenen Eigenschaften von TL auf Typ- und Wert-Ebene direkt in TL erfolgen. Daher ist keine externe Beschreibung von Dienstschnittstellen notwendig. Andererseits wird unterlassen, entfernte Dienste, die in anderen Sprachen als TL geschrieben sind, direkt in Anspruch zu nehmen. Als Lösung steht hier die Anbindung lokaler externer Funktionalität als *C*- oder *C++*-Bibliotheken zur Verfügung (siehe Abschnitt 3.5). Für einen entfernten Dienst, dessen *IDL*-Spezifikation vorliegt, kann stets mit Hilfe des gegebenen RPC-Systems (etwa DCE oder einer CORBA-Implementation) ein *C*-Stub generiert und dieser dann in TL eingebunden werden. Mit Hilfe von Callbacks können sogar CORBA-Server realisiert werden (siehe Abschnitt 8.3).

Die hier zusammengestellten, anspruchsvollen Anforderungen an einen RPC-Mechanismus sind in Form der Module *client* und *server* der Tycoon-Bibliothek *csenv* zur Kommunikationsprogrammierung realisiert. In den folgenden Abschnitten wird der Gebrauch dieser Module erklärt.

5.1.2 Strukturelle und algorithmische Auswahl entfernter Dienste

Die Dienstauswahl soll an einem Beispiel demonstriert werden, das einer Arbeit [Müller et al. 94] entstammt, die eine der wenigen *ODP-Trader*-Implementationen beschreibt.

> Es wird angenommen, daß ein Druckdienst für den Ausdruck eines relativ großen Dokumentes in Anspruch genommen werden soll. Als Randbedingung ist eine baldige und kostengünstige Ausführung erwünscht. In diesem Sinne wird festgelegt, daß die Warteschlange eines in Frage kommenden Druckers nicht mehr als fünf bereits vorhandene Druckaufträge aufweisen darf und daß die Druckkosten unter 0.5 „Geldeinheiten“ pro Seite liegen sollen.

Für den Typ der Druckdienste genügen in diesem Zusammenhang folgende Annahmen:

```
Let Printer = Tuple ... print(...) :Ok ... end
```

Die Attribute seien etwas genauer spezifiziert:

```
Let PrinterAttributes = Tuple
  service :String
  name :String
  costPerPage :Real
  queueLength :Int
  ...
end
```

Auf einem Server wird ein konkreter Druckdienst und eine auf ihn zugeschnittene Attributfunktion erzeugt:

```
let printer = tuple
  ...
   let var jobsPending = 0
   let print(...) :Ok = begin jobsPending := jobsPending + 1 ... end
  ...
end

let getPrinterAttributes() :PrinterAttributes =
  begin
    let t = time.format(time.now())    (* Aktuelle Uhrzeit *)
    tuple
      let service = "Printer"
      let name= "Laser1017"
      let costPerPage =
        if t.hours >= 8 andif t.hours < 18 then 0.7 else 0.4 end
      let queueLength = printer.jobsPending
      ...
    end
  end
```

Um den Druckdienst zu erbringen, wird (analog zur Verwendung des Moduls *portServer* in Abschnitt 5.2.2.2) mit Hilfe des Moduls *server* ggf. zuerst ein *Dispatcher* erzeugt:

```
let dispatcher = server.new(let multiThreaded[3] = true)
```

Dieses Objekt verwaltet den zur Server-seitigen Durchführung von RPCs benötigten Funktionscode (*Server-Stub*, vgl. Abschnitt 5.2.2.2) und eine initial leere Liste angebotener RPC-Dienste. In letzterer wird nun der Druckdienst registriert:

```
server.register(dispatcher :Printer :PrinterAttributes getPrinterAttributes
                server.searchFun[4] printer)
```

Schließlich wird der Dispatcher und damit auch der Druckdienst aktiviert:

```
server.start(dispatcher)
```

Ein Thread, der die Anweisung *server.start* ausführt, wird dadurch bis auf weiteres zum *Dispatcher-Thread*, der Anfragen die angesprochenen RPC-Dienste zuordnet und ggf. dafür sorgt, daß sie erbracht werden, d.h. daß die angewählte Funktion ausgeführt wird und ihr Resultat zum Client gelangt.

Ein Client kann sich entsprechend obiger Aufgabenstellung an den installierten Dienst binden, indem er z.B. folgende Anfrage stellt:

```
let cheapAndAvailable(pa :PrinterAttributes) :Bool =
  pa.costPerPage < 0.5 andif pa.queueLength < 5

let printerProxy = client.bind(:Printer :PrinterAttributes cheapAndAvailable
                               let searchDepth[5] = searchDepthLAN)
```

Zwischen 8 und 18 Uhr löst der Aufruf von *client.bind* unter der Annahme, daß keine weiteren preiswerten Druckdienste vorhanden sind, die Ausnahme *client.error* aus. Außerhalb dieser Zeit gelingt die Bindung an den oben registrierten Dienst und es kann per RPC gedruckt werden:

```
printerProxy.print(...)
```

Eine direktere und einfachere Modellierung einer nicht-trivialen Dienstauswahl als die hier demonstrierte ist kaum vorstellbar. Ein (*C*- oder *C++*-) Programm, das mit der Hilfe einer *IDL* und einem *ODP-Trader* ein äquivalentes Verhalten implementiert, ist erfahrungsgemäß um ein Mehrfaches länger, erheblich komplizierter und außerdem an diversen Stellen typunsicher.

[3] Siehe Abschnitt 5.2.2.2 an korrespondierender Stelle.
[4] Die Standardsuchfunktion *server.searchFun* wird in Abschnitt 5.2.3.2 erklärt.
[5] Der Parameter *searchDepth* wird in Abschnitt 5.2.3.3 erklärt.

Wenn mehrere Dienste der gleichen Art zur Verfügung stehen, ist es für den Client häufig sinnvoll, zuerst eine Liste passender Dienste anzufordern und dann anhand eines weiteren Vergleichs der vorliegenden Attribute eine Präferenzentscheidung zu treffen. Die Funktion *client.find*[6] hat die gleichen Eingabeparamater wie *client.bind*, anstelle eines Proxy liefert sie jedoch eine Liste (*Iter.T*) mit Elementen folgenden Typs:

```
Let Info(S <:Service  Attributes <:Ok) = Tuple
   Dyn Svc <:S              (* konkreter Diensttyp *)
   Dyn A <:Attributes       (* konkreter Attributtyp *)
   attributes :A            (* aktuelle Dienstattribute *)
   id :sid.T(S)             (* abstrakter Dienstidentifikator *)
end
```

Die ersten drei Felder eines *Info*-Tupels geben die bei einem entfernten Dienst vorliegenden konkreten aktuellen Ausprägungen der Suchkriterien an. Das Feld *id* enthält die logische Adresse des entfernten Dienstes, welche in einem späteren Schritt dazu dient, ohne erneute Suche eine Bindung herzustellen. Für diesen Zweck ist dann die Spezialfunktion *client.bindSID* vorgesehen.

Mit diesen Mitteln kann nun zum Beispiel der preiswerteste aller preiswerten Druckdienste gebunden werden:

```
let printerInfos =   (* Informationen über alle passenden Dienste einholen: *)
   client.find(:Printer  :PrinterAttributes  cheapAndAvailable  searchDepthLAN)

(* Prädikat, das durch Projektion das Kostenfeld extrahiert: *)
let costPerPage(info :Info(Printer  PrinterAttributes)) :Real =
   info.attributes.costPerPage

(* Anhand obigen Prädikates wird das Dienstinformationselement,
   welches minimale Kosten pro Seite angibt, bestimmt: *)
let cheapestInfo = getMinimum[7](printerInfos  costPerPage)

(* Erzeugung eines Proxys für den ausgewählten Dienst: *)
let printerProxy = client.bindSID(cheapestInfo.id)
```

[6] Zur Zeit ist die Funktion *client.find* zwar noch nicht implementiert, aber in einer Vorläuferversion [Johannisson 95] des Tycoon-RPC ist bereits eine in wesentlichen Teilen äquivalente Funktion vorhanden, welche die Umsetzbarkeit und die Nützlichkeit der hier angegebenen Konzepte eindrucksvoll demonstriert.

[7] Diese Funktion kann zum Beispiel wie folgt implementiert sein:

```
let rec getMinimum(X <:Ok  candidates :Iter.T(X)  value(x :X):Real) :X =
   begin
      let var m = iter.head(candidates)  (* Bei leerer Iteration: Exception iter.error *)
      foreach iter.rest(candidates) with i do
         if value(i) < value(m) then m := i end
      end
      m
   end
```

5.1.3 Programmierung mit polymorphen entfernten Funktionen höherer Ordnung

Für die Signaturen entfernter Funktionen bestehen im Vergleich zu lokalen Funktionen keinerlei Einschränkungen. Insbesondere können entfernte Funktionen von höherer Ordnung und polymorph sein. Es kann nicht nur der Inklusionspolymorphismus von Wertparametern, sondern auch parameterischer Polymorphismus genutzt werden, d.h. entfernte Funktionen können Typparameter haben.

Die weitreichenden Konsequenzen dieser orthogonalen Eigenschaften sollen am Beispiel einer fiktiven Geschäftsdatenbank aufgezeigt werden, die durch ein TL-Modul namens *geschaeft* mit folgender Schnittstellendefinition gegeben sei:

```
interface Geschaeft
import db
export

  Let rec Verkauf = Tuple
    datum :Datum
    produkt :Produkt
    kunde :Kunde
    preis :Geld
    ...
  end

  and Produkt = Tuple
    name :String
    verkaeufe :list.T(Verkauf)
    ...
  end

  and Kunde = Tuple
    name :String
    sitz :Ort
    umsatzProJahr :Geld
  ...
  end
  ...

  verkaeufe :db.T(Verkauf)
  produkte :db.T(Produkt)
  kunden :db.T(Kunde)
  ...

end;
```

Das Modul *geschaeft* stellt im wesentlichen eine Sammlung relationaler Datenbankrelationen[8] (*verkaeufe* etc.) unterschiedlichen Typs bereit. Die mit der Hilfe von RPCs zu lösende Aufgabe besteht darin, an einem (möglicherweise weit) entfernten Arbeitsplatz auf effiziente Weise Geschäftsinformationen bereitzustellen. Dazu wird zuerst ein Diensttyp definiert, der die Signaturen interessierender Anfragen zusammenfaßt:

```
let GeschaeftsInfo = Tuple

  sucheKundenPerNamen(name :String) :Kunde

  sucheKunden(praedikat(k :Kunde) :Bool) :Iter.T(Kunde)

  bearbeite(R <:Ok  anfrage(geschaeft :Geschaeft) :R  optimiert :Bool) :R

  umsatzProTag(produktName :String  von, bis :Datum) :Fun(breit, hoch :Real) :Graphik
  ...
end
```

Dieser Dienst werde durch ein Tupel, das die genannten Funktionen implementiert, instanziert:

```
let info = tuple ... (*siehe unten*) ... end

let getAttributes() = "Firma X"    (* Hier ist eine feste Benennung sinnvoll. *)

(* Dienst per RPC verfügbar machen: *)
server.register(dispatcher  :GeschaeftsInfo  :String
                getAttributes  server.searchFun  info)
```

Eine Funktion wie *info.sucheKundenPerNamen*, die den Datenbankeintrag eines per Namen angegebenen Kunden liefert, stellt einen normalen RPC, wie man ihn von anderen Systemen her kennt, dar. Hingegen sind die weiteren Funktionen „höherer Ordnung“. Sie involvieren die Mobilität von Funktionsabschlüssen.

Die Funktion *info.sucheKunden* übermittelt ein als TL-Funktion formuliertes Prädikat an den Server, um es vor Ort iterativ[9] auf die Elemente der Kundendatenbank anzuwenden.

```
let sucheKunden(praedikat(:Kunde) :Bool) :Iter.T(Kunde) =
  begin
    let var result = iter.new(:Kunde)
    foreach kunden with k do
      if praedikat(k) then result := iter.cons(result  k) end
    end
    result
  end
```

[8] Der hier angegebene Modulname *db* steht stellvertretend für eine beliebigen geeigneten Massendatentyp.
[9] Von der Ineffizienz dieser Suchmethode sei hier abstrahiert.

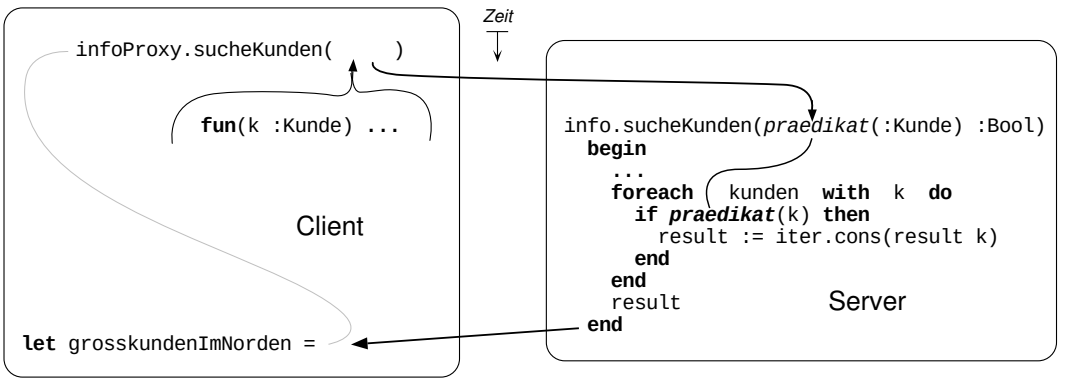

Abbildung 5.4: Aufruf einer entfernten Funktion höherer Ordnung

Ein entfernter Client kann sich wie folgt eine Liste der Datensätze aller Großkunden im Norden Deutschlands verschaffen:

```
let isFirmaX(firma :String) = string.equal(firma "Firma X")

let infoProxy = client.bind(:GeschaeftsInfo :String isFirmaX searchDepthLAN)

let grosskundenImNorden = infoProxy.sucheKunden(fun(k :Kunde)
  k.umsatzProJahr >= 1000000.0 andif k.sitz.plz < 40000
```

Abbildung 5.4 veranschaulicht den Ablauf dieses Aufrufes, bei dem das als Parameter übergebene Prädikat streng in ein auf dem Server vorliegendes Ablaufschema eingebunden ist. Letzteres muß jedoch nicht unbedingt immer der Fall sein. Zum Beispiel erlangt die Parameterfunktion von *bearbeite* vollkommene Handlungsfreiheit bezüglich des Datenbankmoduls *geschaeft*.

```
let bearbeite(R <:Ok anfrage(geschaeft :Geschaeft) :R optimiert :Bool) :R =
  if optimiert then
    optimize(fun() anfrage(geschaeft))()
  else
    anfrage(geschaeft)
  end
```

Hier besteht durch den Einsatz parametrischen Polymorphismus die freie Wahl des Ergebnistyps, und dies gilt unter vollkommener Gewährleistung statischer Tysicherheit. Ein Client kann zum Beispiel den in Geldeinheiten berechneten, durchschnittlichen Erlös aller gespeicherten Verkäufe ermitteln, indem unter Einsatz der Syntaxerweiterungen von TL [Cardelli et al. 94] in SQL-Syntax folgende Anfrage formuliert wird:

```
let durchschnittVK :Geld = infoProxy.bearbeite(
  SELECT AVG(preis) FROM geschaeft.verkaeufe (* SQL *)
  let optimiert = true)
```

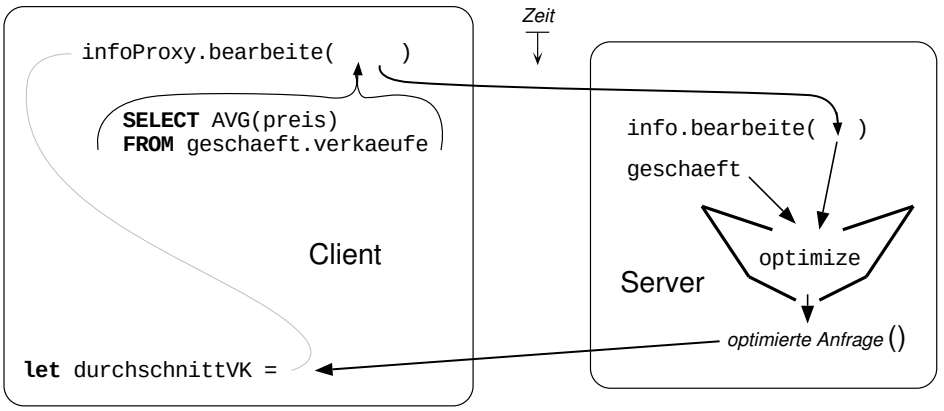

Abbildung 5.5: Entfernte dynamische Optimierung und Ausführung einer Datenbankanfrage

Wahlweise führt die entfernte Funktion *bearbeite* vor Auswertung der an sie übersandten Anfrage zuerst eine *dynamische Optimierung* [Gawecki, Matthes 95a] derselben durch. Der Ablauf eines solchen entfernten Aufrufes in Abbildung 5.5 dargestellt.

Die Einsatzmöglichkeiten der Funktion *bearbeite* sind vielgestaltig. Als Parameter kann ebenso gut eine andere SQL-Anfrage, die sich auf die Relationen der Geschäftsdatenbank bezieht, bzw. jede beliebige TL-Funktion, die die geforderte Signatur erfüllt, eingesetzt werden. Entfernte Funktionen höherer Ordnung der hier gezeigten Form sind richtungsweisend für zukünftige Datenbanktechnik, indem sie die Ausdrucksmächtigkeit einer Mehrzweckprogrammiersprache, klassische Datenbankanfragetechniken und Optimierungen mit *statischer* Typsicherheit vereinen.

Die Funktion *umsatzProTag* dient der Erstellung von Umsatz-Histogrammen. Sie liefert allerdings keine fertige Graphik, sondern eine Funktion, welche die gewünschte Graphik erstellt.

```
let umsatzProTag(produktName :String  von, bis :Datum)
                 (breit, hoch :Real) :Graphik =
  begin
    let tagesUmsaetze = arrayOp.new(...)
    let produkt = sucheProduktNamen(produkte  produktName)
    foreach produkt.verkaeufe with verkauf do
      let index = ... verkauf.datum ...
      tagesUmsaetze[index] := tagesUmsaetze[index] + verkauf.preis
    end
    (* Eine (anonyme) Funktion als RPC-Ergebnis: *)
    fun(breit, hoch :Real) :Graphik
      begin
        titel("Umsätze " <> produktName <> ": " <> zeitraum(von  bis))
        foreach tagesUmsaetze do ... end
        ...
      end
  end
```

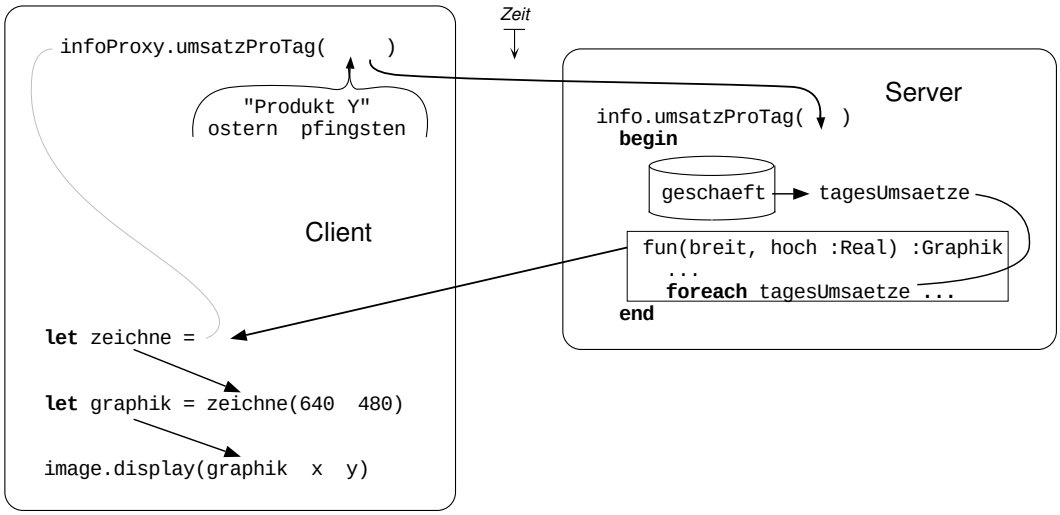

Abbildung 5.6: Ein Funktionsabschluß als RPC-Ergebnis

Eine exemplarische graphische Anzeige der Tagesumsätze zwischen Ostern und Pfingsten erfolgt dann, wie in Abbildung 5.6 dargestellt, in folgenden drei Schritten.

let zeichne = infoProxy.umsatzProTag("Produkt Y" ostern pfingsten)

let graphik = zeichne(640 480) *(* Graphik lokal erstellen. *)*

image.display(graphik x y) *(* Graphik anzeigen. *)*

Typischerweise ist das Datenvolumen der Zeichenfunktion um mehrere Größenordnungen geringer als das einer fertigen Graphik. Zudem kann der Code der Funktion, wenn er einmal übertragen wurde, im Client persistent gespeichert und bei weiteren Übertragungen dynamisch gelinkt werden (siehe Abschnitt 7.4). In Netzwerken mit geringer Bandbreite stellt diese Form der Code-Mobilität eine erhebliche Entlastung gegenüber herkömmlichen Ansätzen dar. Eine weitere Anwendung besteht in der Übertragung von Animationen zum Client, wodurch der derzeitige Hauptnutzen der Programmiersprache Java [Sun 95b; Golsing, McGilton 95] auch durch TL erbracht wird.

5.1.4 Persistenz und Mobilität kommunizierender Objekte

Die mit entfernten Funktionen im Zusammenhang stehenden Objekte befinden sich im Einklang mit den originären Konzepten der orthogonal persistenten Sprache TL. Proxys, Dienste und Dispatcher verhalten sich in vielerlei Hinsicht wie andere TL-Objekte auch: sie unterliegen der lokalen Garbage-Collection, sie sind implizit persistent und sie können sowohl an lokale als auch an entfernte Funktionen als Argumente übergeben werden, d.h. sie sind mobil.

In [Tanenbaum, van Renesse 88] wird die Festlegung der Client/Server-Rollen in früheren RPC-Systemen kritisiert. Die Mobilität von Proxys in TL erlaubt nun, auf besonders einfache Weise entfernte Callbacks von Servern an Clients zu implementieren. Dazu genügt es, ein Proxy als RPC-Argument zu verwenden. Diese nahtlose sprachliche Integration entfernter Aufrufe entspricht in Kombination mit der *dynamischen* Erzeugung von Client-Stubs eher den Gegebenheiten in einer objektorientierten verteilten Programmiersprache wie z.B. Obliq (vgl. Abschnitt 2.3.2) als einem *statischen* RPC-System. In DCE sind zwar entfernte Callbacks möglich, sie sind jedoch nicht transparent eingebettet [Schill 93].

Client-Bindungen sind robust gegenüber Veränderungen, die in konventionellen Systemen unweigerlich zur Unbrauchbarkeit der betroffener Programmkomponenten führen würden: Proxy-Migrationen[10], Unterbrechungen (Absturz oder Abschaltung mit anschließendem Neustart) von Client- oder Server-Programmen sowie Verlagerungen von Client- oder Server-Programmen. Letzteres kann z.B. durch Umkopieren des Programms samt persistentem Speicher auf einen anderen Host oder durch den Wechsel des gegebenen Host (etwa eines Notebook) an einen anderen Netzanschluß geschehen. Ferner werden vorübergehende Unterbrechungen eines Dienstes von gerade passiven Proxys toleriert.

Die globalen Transaktionen von Servern erfassen automatisch alle Server-Threads, Dispatcher, und angebotenen RPC-Dienste. Deshalb ist lediglich eine einzige Programmzeile erforderlich, um das Wiederaufsetzen nach einem Programmneustart zu organisieren (vgl. Abschnitt 3.2.6):

persistent.commit()

Durch diese Anweisung werden alle Objekte eines einmal konfigurierten Servers in ihrem aktuellen Zustand gesichert. Dies gilt insbesondere für aktive Threads. Sie laufen nach einem Neustart automatisch wieder an, bzw. „weiter“ (siehe Abschnitt 6.1.2).

Ein Server-Programm kann sein Dienstangebot nicht nur jederzeit beliebig erweitern sondern auch vermindern. Für letzteren Zweck liefert die Dienstregistrierungsfunktion *server.register* stets einen Dienstidentifikator vom Typ *sid.T*. Dieser kann an die Funktion *server.unregister* übergeben werden, um den entsprechenden Dienst aus dem Angebot eines Dispatchers zu entfernen. Clients werden von der Abmeldung eines Dienstes zunächst nicht benachrichtigt. Sie erfahren davon erst bei einem anschließenden Versuch eines entfernten Aufrufes, indem in ihrem Kontext eine Ausnahme (*Exception*) ausgelöst wird. Auf diese Weise werden Proxys, die sich in aktiven Prozessen befinden und solche, die in nicht aktivierten Programmen (bzw. den dazugehörigen Objektspeichern) „ruhen“, vollkommen gleichbehandelt.

[10] Die Möglichkeit, ein Proxy unbeschadet in einen anderen Adreßraum zu transferieren, wird auch als *third party transfer* [Nelson 81; Birell et al. 93b] bezeichnet.

Bei der Beendigung (Absturz oder Abschaltung) von Server-Prozessen besteht nie Gewißheit, ob und wann ein Neustart erfolgen wird. Es ist zwar vorstellbar, eine Klassifikation vorzunehmen, die unterscheidet, ob ein Neustart ggf. überhaupt beabsichtigt wäre. Diese beträfe jedoch notwendigerweise Programme und nicht Objekte geringerer Granularität, was eine Einschränkung der Persistenz zur Folge hätte: migrierende Objekte würden unter Umständen wechselnder Persistenz unterliegen.

Damit für alle Objekte überall dieselben einheitlichen Bedingungen gelten, wird per Definition gefordert, daß Anwendungsprogramme persistenter Objektsysteme nie endgültig beendet seien. Vielmehr sei stets in absehbarer Zeit mit einer Wiederaufnahme des Betriebs zu rechnen. Ausnahmen sind auf Anwendungsebene auszuprogrammieren.

Aus Sicht eines Proxys ist in jedem Fall des Mißlingens eines Kommunikationsversuches oder der Störung eines gerade aktiven Kommunikationsvorganges die unterliegende Middleware der zuständige Ansprechpartner. Dabei wird gleichgültig, ob Programmbeendigungen, physikalische Leitungsunterbrechungen, allzu lange Latenzzeiten, Programmfehler oder sonstige Gründe vorliegen, eine dynamische Ausnahmebehandlung ausgelöst. Es ist die Aufgabe der Anwendung, rekursiv entlang der Aufrufkette die jeweils nächsthöhere Schicht mit einer aussagekräftigen Fehlermeldung zu versorgen, bis die Fehlersituation ausgewertet und behandelt ist.

Programmierschnittstelle	Sprache	Paradigma	Adressierung	Bindung	Server-Objekte	Client-Objekte
client server	TL	RPC	assoziativ: Typen Attribute	persistent	mobil	mobil
portClient portServer	TL	RPC	Ports	flüchtig	stationär	mobil
port	TL	Kanal	Ports	flüchtig	stationär	mobil
Socket	C	Kanal	Ports	flüchtig	stationär	stationär

Abbildung 5.7: Schnittstellen der Kommunikationsbibliothek des Tycoon-Systems

5.2 Die Kommunikationsbibliothek des Tycoon-Systems

Die Implementation mobilitätsorientierter entfernter Funktionsaufrufe besteht in einer TL-Bibliothek. Es wurden keine neuen Schlüsselworte in die Sprache TL eingeführt, d.h. der Compiler wurde nicht verändert, und es ist kein Generator für Stubs entstanden.

Den im ersten Teil dieses Kapitels beschriebenen Modulen *client* und *server* liegen mehrere Kommunikationsmechanismen mit einem einfachen Adressierungsschema zugrunde. Abschnitt 5.2.1 vermittelt einen Überblick über die Funktionalität der entsprechenden Module und stellt die Gesamtarchitektur der Kommunikationsbibliothek vor.

Abschnitt 5.2.2 beschreibt die Implementationen der Kommunikationsmechanismen mit einfacher Adressierung. Dabei wird insbesondere auf die generische Erzeugung von RPC-Stubs und die entfernte Ausführung von Funktionen eingegangen. Nachdem die Aspekte der Datenübertragung und der Ausführung behandelt worden sind, verbleibt noch das Problem der Adressierung zum Zwecke der Bindung entfernter Funktionen. In Abschnitt 5.2.3 werden die Anforderungen und Implementationsmethoden der dynamischen hierarchischen Adreßauflösung entfernter Funktionsaufrufe beschrieben. Dabei stellt sich heraus, daß spezielle Netzdienste benötigt werden, deren Implementation dann Abschnitt 5.2.4 gewidmet ist.

5.2.1 Schnittstellen und Modulstruktur

In der TL-Bibliothek zur Kommunikationsprogrammierung mit Namen *csenv* stehen insgesamt drei Programmierschnittstellen mit unterschiedlichen Qualitäten zur Verfügung. Abbildung 5.7 stellt die relevanten Schnittstellen der Bibliothek und die *C*-Schnittstelle für Sockets in einer vergleichenden Übersicht dar. Während *client* und *server* für die allgemeine Anwendungsprogrammierung konzipiert sind, eignen sich die weiteren Kommunikationsdienste vornehmlich für die Systemprogrammierung.

Das Modul *port* bietet einen typunsicheren, bidirektionalen Kommunikationsdienst, mit dem Tycoon-Objekte beliebigen Typs zwischen Tycoon-Anwendungen im Internet übertragen wer-

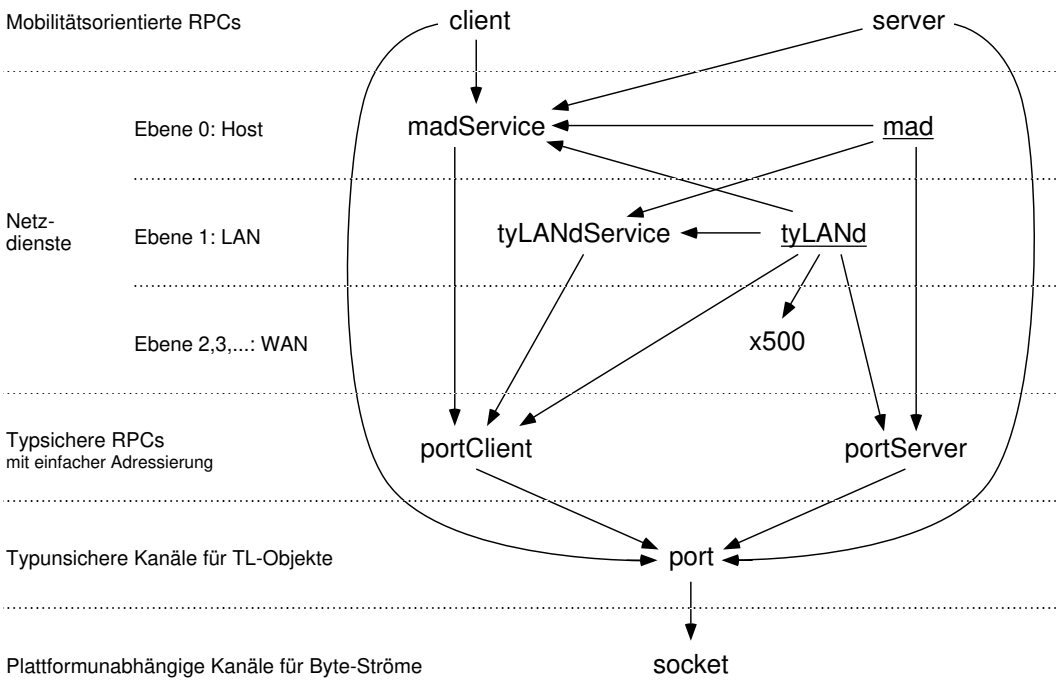

Abbildung 5.8: Modulstruktur der Tycoon-Bibliothek zur Kommunikationsprogrammierung

den können. Der Name des Moduls entstammt der Grundidee, vom Konzept der im Internet-Jargon „Ports“ genannten Kommunikationsendpunkte einen handlichen (quasi) abstrakten Datentyp (*port.T*) abzuleiten: alle wesentlichen Funktionen des Moduls *port* operieren auf Objekten, die Internet-Adressen mit explizit angegebener Port-Nummer repräsentieren und damit eindeutig bestimmte Ports identifizieren.

Die Module *portClient* und *portServer* realisieren mit Hilfe des Moduls *port* und bestimmten typunsicheren Funktionen des Moduls *unsafe* entfernte Funktionsaufrufe, die bezüglich ihrer Signatur und operativen Funktionalität den vollen Sprachumfang von TL unterstützen. Sämtliche Operationen sind statisch typsicher. Proxys sind hier bereits mobil.

Die Module *portClient* und *portServer* kommen bei der Implementation der im Netzwerk verteilten Systemdienste MAD und TYLAND (siehe unten) bevorzugt zum Einsatz, wobei sich ihre extrem unaufwendige Implementation (zusammen ca. 300 Zeilen) bereits nahezu amortisiert. Ihre Implementation ist zudem sehr instruktiv (siehe Abschnitt 5.2.4). In den folgenden Abschnitten wird dies ausgenutzt, indem zunächst der Aufrufmechanismus entfernter Funktionen nahezu isoliert betrachtet und erst danach komplexere Maßnahmen zur Adressierung erklärt werden.

Die im vorangehenden Abschnitt beschriebenen Module *client* und *server* implementieren Aufrufe entfernter Funktionen nach dem gleichen Schema wie *portClient* und *portServer*, jedoch werden weitaus elaboriertere Adressierungsmechanismen sowie flexible Reaktionen auf bestimmte Situationen integriert. Dadurch werden die Persistenz von Client-Bindungen und Server-Verlagerungen unterstützt. Außerdem müssen Anwendungsprogrammierer nicht mehr explizit mit Netzwerkadressen umgehen.

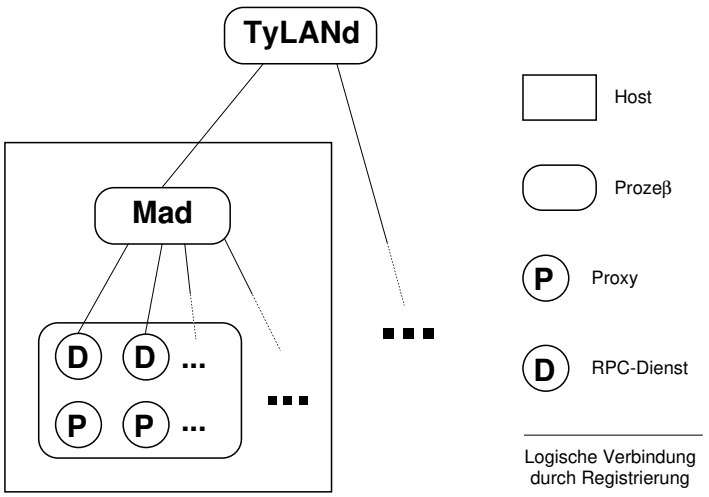

Abbildung 5.9: Die Anordnung der Dienstprogramme im Netzwerk

Die Module *client* und *server* sind das Endprodukt einer systematisch auf der portablen Implementationsbasis aus Kapitel 4 aufbauenden Modulstruktur, die aus Abbildung 5.8 ersichtlich ist. Jeder Bezeichner steht für ein Modul. Hauptmodule (*mad*, *tyLANd*) von Programmen sind unterstrichen. Die Pfeile stellen Modulimportbeziehungen dar. Sie zeigen jeweils von einem importierenden Modul auf ein bei dessen Implementation verwendetes Modul.

Die im vorangehenden Kapitel beschriebenen, plattformunabhängigen Implementationsgrundlagen befinden sich ganz unten: das plattformunabhängige Modul *socket* zur Bytestromübertragung (siehe Abschnitt 4.2) und das Modul *unsafe*, welches unter anderem Funktionen zum Ein- und Auslesen externer Repräsentationen von Tycoon-Objekten anbietet (siehe Abschnitt 4.3.1).

Darüber steht das Modul *port* gefolgt von *portClient* und *portServer*. Letztere sind bei der Implementierung von *client* und *server* nicht direkt involviert. Eine Schichtung wäre hier prinzipiell möglich, aber aufgrund der trefflichen Eignung der Funktionen von *port* sind auf diese Weise keine Verbesserungen zu erzielen. Im Gegenteil wäre es sogar etwas komplizierter, einen RPC mit einem RPC zu implementieren als mit kanalbasierten Primitiven.[11]

Intern identifizieren alle mit Hilfe des Moduls *client* erzeugten Stubs ihre korrespondierenden entfernten Dienste durch weltweit eindeutige Dienstidentifikatoren (SIDs), die durch Werte des abstrakten Datentyps *sid.T* aus dem Modul *sid* repräsentiert werden. SIDs werden bei Bedarf, d.h. direkt vor entfernten Aufrufen, mit Hilfe zweier im Netz verteilte Dienstprogramme dynamisch in aktuelle Netzwerkadressen vom Typ *port.T* übersetzt.

Abbildung 5.9 stellt die hierarchische Anordnung dieser Netzdienste dar und liefert gleichzeitig das graphische Vokabular für später folgende Abbildungen zu diesem Thema.

[11] Dies haben nicht zuletzt die praktischen Erfahrungen bei der Implementation einer früheren Version des Tycoon-RPC [Johannisson 95] ergeben.

Auf jedem Host, der potentiell für verteilte Tycoon-Anwendungen genutzt werden soll, ist eine Instanz des Dienstprogramms MAD (*Mapping Daemon*) vorgesehen. Dieses Programm läuft als eigener Betriebssystemprozeß. Es verwaltet eine Liste aller auf dem lokalen Rechner befindlichen Tycoon-Dienste, die per Modul *server* angeboten werden und beantwortet mit Hilfe des Moduls *portServer* Anfragen nach Kommunikationsendpunkten, die mit Hilfe des Moduls *portClient* zu stellen sind.

Konkret wird das MAD-Dienstprogramm durch das Modul *mad* implementiert, wobei das Modul *madService* als entfernter Dienst nach Art von *portServer* angeboten wird.

In jedem LAN, das für Tycoon-Anwendungen vorgesehen ist, muß zum Zwecke der rechnerübergreifenden Dienstauffindung eine Instanz des Dienstprogramms TYLAND (*Tycoon LAN daemon*) laufen. Dieses Programm verwaltet eine Liste aller MADs in seinem LAN. Es beantwortet Anfragen von MADs nach der Existenz und den Kommunikationsendpunkten von Diensten.

Konkret wird das TYLAND-Dienstprogramm durch das Modul *tyLANd* implementiert, wobei das Modul *tyLANdService* als entfernter Dienst nach Art von *portServer* angeboten wird. Verteilte Tycoon-Anwendungen funktionieren auch ohne TYLAND. Allerdings werden dann Server-Verlagerungen nicht mehr unterstützt.

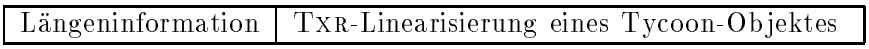

Abbildung 5.10: Aufbau eines Kommunikationspaketes

5.2.2 Polymorphe Kommunikationsmechanismen mit einfacher Adressierung

In den folgenden beiden Abschnitten werden die systematisch auf der im vorangehenden Kapitel beschriebenen plattformunabhängigen Implementationsbasis aufbauenden Implementationen und Funktionalitäten der niederen Kommunikationsmechanismen der Kommunikationsbibliothek erläutert. Es wird mit dem Modul *port* begonnen. Anschließend wird die generische Erzeugung von RPC-Stubs und die entfernte Ausführung von Funktionen beschrieben.

5.2.2.1 Typunsichere Kanäle

Das Modul *port* kombiniert das in Abschnitt 4.2 beschriebene Module *socket* mit den Funktionen *unsafe.externToNewBuffer* und *unsafe.internFromBuffer* aus Abschnitt 4.3.1 zu einem typunsicheren, bidirektionalen Kommunikationsdienst, mit dem Tycoon-Objekte *beliebigen Typs* plattformunabhängig zwischen Tycoon-Anwendungen im Internet übertragen werden können.

Das lineare Tycoon-Datenformat TXR (siehe Abschnitt 4.3) sieht zwar die Anzahl der erfaßten Speicherobjekte vor, nicht jedoch die Gesamtgröße des jeweiligen Datenstroms in Bytes. Diese für eine geordnete datenstromorientierte Netzwerkkommunikation benötigte Information liefert die Funktion *unsafe.externToNewBuffer* deshalb jeweils zusätzlich zu jeder Linearisierung (siehe Abschnitt 4.3.1). Bei der Socket-Übertragung eines linearisierten Tycoon-Objektes wird diesem dann die Länge in Bytes vorangestellt. Abbildung 5.10 zeigt den Aufbau eines solchen Kommunikationspaketes. Eine plattformunabhängige Interpretation der Längenangabe ist durch eine fest vorgeschriebene, nach Wertigkeit geordnete Reihenfolge ihrer Bits gegeben.

Zur Adressierung von Kommunikationsendpunkten werden Internet-Adressen in den beiden eher benutzerfreundlichen Darstellungsformen verwendet: DNS-Namen oder Byte-Quadrupel als Zeichenketten. Folgender Typ repräsentiert Host-Adressen in jeweils einer der beiden Varianten:

```
Let Host = Tuple
             case dnsName with
               name :String
             case ipAddress with
               address :String
           end
```

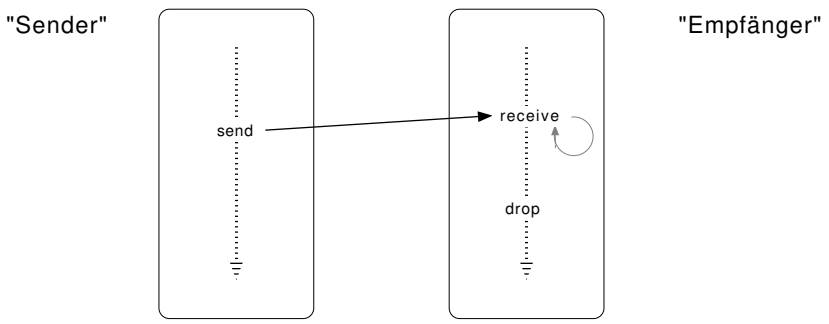

Abbildung 5.11: Einfaches Senden eines Tycoon-Objektes

Kommunikationsendpunkte ergeben sich durch die Kombination einer Host-Adresse mit einer Port-Nummer:

```
Let T = Tuple
          portNumber :Int
          host :Host
        end
```

Dies ist der Haupttyp des Moduls *port*. An Adressen diesen Typs kann mit folgender Funktion auf denkbar einfache Weise ein Wert geschickt werden.

```
send(destination :T V <:Ok v :V) :Ok
```

Diese Funktion akzeptiert einen Tycoon-Wert v beliebigen Typs V, stellt eine Socket-Verbindung zum Kommunikationsendpunkt *destination* her und überträgt den Wert dorthin. Die bidirektionale Variante *request* nimmt im Gegenzug einen Wert entgegen und gibt ihn als Funktionsergebnis zurück:

```
request(server :T V <:Ok v :V R <:Ok) :R
```

Der Ergebnistyp R ist frei wählbar. Es ist Aufgabe des Benutzers, dafür zu sorgen, daß er zu dem erhaltenen Wert paßt.

Auf der Gegenseite ist folgende Funktion zu verwenden.

```
receive(portNumber :Int var p :Promise V <:Ok) :V
```

Diese Funktion blockiert den aktuellen Thread, bis am Port mit der Nummer *portNumber* eine Anfrage eingeht, nimmt das ankommendene Wertpaket entgegen, delinearisiert es und liefert den entstehenden Objektgraphen als typisiertes Funktionsergebnis. Der Ausgabeparameter p liefert ein Handle auf die nach dem Aufruf von *receive* weiterhin bestehende Socket-Verbindung zum Anfragesender. Für die Verwendung dieses Handles gibt es zwei Alternativen.

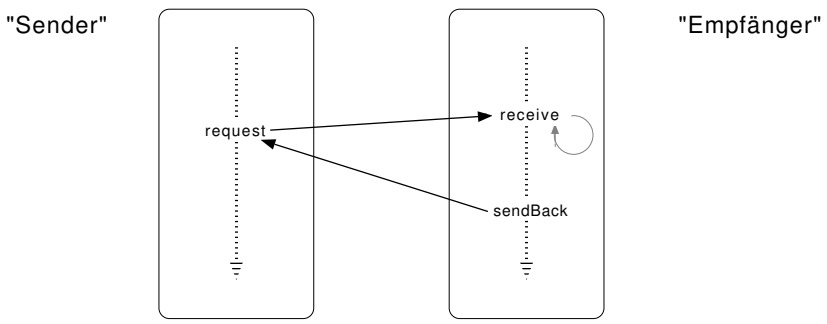

Abbildung 5.12: Senden eines Tycoon-Objektes mit Rückantwort

Die Funktion *drop* sorgt für eine umgehende, ordnungsgemäße Aufhebung der Socket-Verbindung. Dies empfiehlt sich, wenn die Anfrage per *send* erfolgte.

drop (p :Promise) :Ok

Der von *send*, *receive* und *drop* gebildete verteilte Ablauf ist aus Abbildung 5.11 ersichtlich.

Im Fall einer Anfrage per *request* ist mit einem *sendBack* zu antworten. Die Funktionsweise dieser Funktion gleicht der von *send*. Allerdings findet anstelle einer Adresse eine durch ein Handle vom Typ *Promise* identifizierte, bereits bestehende Socket-Verbindung Verwendung.

sendBack(p :Promise V <:Ok v :V) :Ok

Auf diese Weise wird das „Versprechen“ (engl.: *Promise*, vgl. [Liskov, Shrira 88]) des Empfängers, ein Objekt zurückzusenden, eingelöst. Den Ablauf dieser Kombination zeigt Abbildung 5.12. Hier handelt es sich bereits um eine recht komfortable Möglichkeit der Client/Server-Programmierung.

In Abbildung 5.11 und 5.12 ist jeweils das Layout von Abbildung 2.2 aus Abschnitt 2.1.1 wiederverwendet worden, um eine direkte Gegenüberstellung mit der Socket-Programmierung in *C* zu erzielen. Die hier entwickelten Primitive sind zwar nicht ganz so flexibel wie Sockets, aber sie sind noch erheblich einfacher zu handhaben. Es ist zu beachten, daß im Modul *port* nur scheinbar eine Beschränkung auf jeweils ein Objekt pro Funktionsaufruf vorliegt. Wenn mehrere Tycoon-Objekte zu übermitteln sind, so können diese problemlos in der Sprache TL (etwa in einem Tupel oder Array) aggregiert und dann als Gesamtobjekt versendet werden.

5.2.2.2 Typsichere entfernte Funktionsaufrufe

Ein wichtiger Aspekt des oben beschriebenen Szenarios der Objektübertragung mit Rückantwort (siehe Abbildung 5.12) besteht in der Möglichkeit des Clients nach Erhalt eines Objektes beliebige Operationen auszuführen, ohne die (Ver-)Bindung zum Server zu verlieren. Die hier ersichtliche Abfolge *Anfrage, Operation, Antwort* ist bereits das Grundschema synchroner RPCs.

Der nächste Entwicklungsschritt besteht nun in einer nahtlosen Einbettung dieser Funktionalität in die Programmiersprache TL. Als entfernte Dienste fungieren Tupel von TL-Funktionen. Da gebundene Module durch Tupel repräsentiert werden (siehe Abschnitt 3.2.5), können sie ohne weiteres als entfernte Dienste angeboten werden.

Die Bereitstellung eines entfernten Dienstes geschieht vollkommen dynamisch ohne jede Compilation oder Reflektion, einfach durch den Aufruf einer Bibliotheksfunktion. Ebenso verhält es sich mit der Client-Bindung. Die verborgenen Kommunikationsunterprogramme, der sogenannte Stub, werden im Gegensatz zu anderen RPC-Systemen nicht reflektiv generiert, sondern sie werden jeweils durch recht einfache Manipulationen einiger weniger Laufzeitwerte zusammengesetzt. Dabei spielen Funktionsabschlüsse eine wichtige Rolle.

Folgende Anweisung erzeugt einen *Dispatcher* (vgl. Abschnitt 5.1.2), der eine (initial leere) Tabelle verwaltet, deren Einträge auf als entfernte Dienste angebotene Tupel verweisen.

let *dispatcher = portServer.new(portNumber* ***let*** *multiThreaded*[12] *=* ***true****)*

An der angegebenen Port-Nummer wird mittels *port.receive* auf eingehende Anfragen „gelauscht“, sobald der Dispatcher aktiviert wird. Dies geschieht mit:

portServer.start(dispatcher)

Dank Multi-Threading können Dienste sowohl bevor, als auch während der Dispatcher aktiv ist, in der Tabelle des Servers registriert oder aus ihr entfernt werden. Zum Einfügen von Diensten dient folgende Funktion.

register(dispatcher :T serviceName :String ***Dyn*** *S <: Service service :S) :Ok*

Ein Dienst wird hier durch einen benutzerspezifizierten Namen (*serviceName*) und seinen *dynamischen* Typ (*S*) identifiziert. Dies sind genau die Attribute, anhand derer Clients Dienste auswählen können. Der Parameter *service* liefert eine Objektreferenz auf die Dienstimplementation. Dabei kann es sich um beliebiges Tupel handeln, denn es ist folgender Supertyp vorgegeben:

Let *Service =* ***Tupel end***

[12] Wenn dieser Parameter auf *true* gesetzt ist, erzeugt der Dispatcher-Thread für jeden RPC jeweils einen eigenen Thread, der die Funktionsausführung und das Antworten an den Client übernimmt. Andernfalls führt der Dispatcher-Thread alle Anfragen selbst aus.

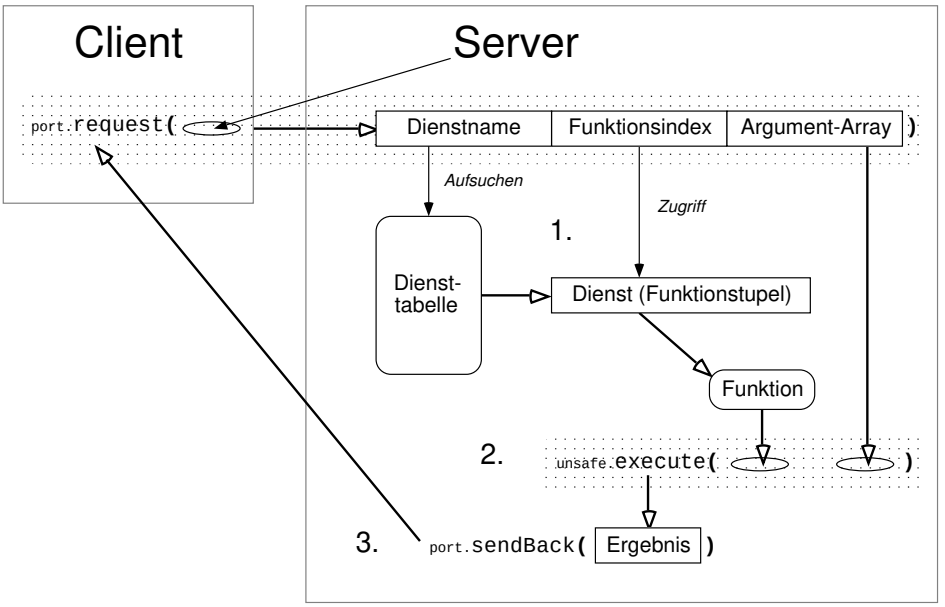

Abbildung 5.13: Anfrageverarbeitung im RPC-Server

Client-Anfragen werden per *port.request* getätigt. Sie bestehen aus einem Dienstnamen, einem Funktionsindex und einem Array von Argumenten. In informellem TL-Code bedeutet dies folgendes Szenario:

```
let call = tupel
               let serviceName = ...
               let functionIndex = ...
               let arguments = array ... end
           end
let result = port.request(serverAddress  call)
```

Auf der Grundlage dieser Informationen kann ein Dispatcher auf recht unkomplizierte Weise den angeforderten Dienst erbringen. Der kanonische Ablauf der Anfrageverarbeitung im Server ist in Abbildung 5.13 schematisch zusammengefaßt.

1. Anhand des Dienstnamens wird in der Diensttabelle das zuständige Diensttupel aufgefunden.

2. Das durch den Funktionsindex angegebene Tupelelement enthält den aufzurufenden Funktionsabschluß. Mit Hilfe der Funktion *unsafe.execute* wird die Ausführung dieser Funktion veranlaßt. Dabei werden die vom Client gelieferten Argumente übergeben.

3. Das Ergebnis des Aufrufs von *unsafe.execute* wird mittels *port.sendBack* zum Client geschickt.

In informellem TL- Code stellt sich dieser Kern des Server-Programms (ein wenig vereinfacht) wie folgt dar:

```
let request = port.receive(portNumber  promise)
let service = lookupService(tableOfServices  request.serviceName)
let function = getTupleSlot(service  request.functionIndex)
let result = unsafe.execute(function  request.arguments)
port.sendBack(promise  result)
```

Obwohl jede einzelne dieser Operationen per se typunsicher ist, besteht zu keinem Zeitpunkt die Gefahr eines Typfehlers. Dies wird durch dynamische Typvergleiche (siehe Abschnitt 3.2.4) beim Aufbau von Client-Bindungen sichergestellt. Zur Herstellung von Client-Bindungen dient die einzige Funktion des Moduls *portClient*. Sie hat folgende Signatur:

```
bind(serverAddress :Port.T  Dyn S <:Service  serviceName :String) :S
```

Diese Funktion liefert im Erfolgsfall ein lokales Tupel vom Typ S. Die Funktionen dieses Tupels führen dann für den Anwender transparent entfernte Aufrufe auf dem unter *serverAddress* angegebenen Server aus. Die Bindung kommt zustande, wenn sich auf diesem Server ein Dienst mit dem durch *serviceName* angegebenen Namen findet, der dem Typ S genügt. Letzteres wird vom Server durch einen dynamischen Subtyptest überprüft. Zu diesem Zweck wird die Laufzeitrepräsentation von S mit zum Server geschickt, sodaß obiges *request*-Paket einfach um ein Element erweitert zu verstehen ist.

Auf welche Weise wird nun erreicht, daß einem Client ein scheinbar gewöhnliches Modul vorliegt, dieses jedoch in für ihn vollkommen transparenter Weise entfernte Aufrufe tätigt? Dazu wird zunächst festgestellt, daß es auch bei lokalen Modulen überhaupt nicht ungewöhnlich ist, wenn der Benutzer einer Modulfunktion deren Funktionsweise nicht kennt. Dies gehört zum Prinzip der Funktionsabstraktion.

Der wesentliche „Trick“ der Funktion *portClient.bind* besteht darin, das lokale Diensttupel des Clients, das Proxy, ausschließlich aus Stub-Funktionen zusammenzusetzen. Dabei gelingt es unter Ausnutzung der Tatsache, daß Funktionen in TL Objekte erster Klasse sind und mit Hilfe zweier Spezialfunktionen des Moduls *unsafe*, immer ein und denselben Stub-Code zu verwenden. Der extrem kurze Quelltext zur Erzeugung von Proxys ist nachstehend fast vollständig wiedergegeben (es fehlen nur Fehlerbehandlungen).

```
let bind(serverAddress :port.T  Dyn S <:Service  serviceName :String) :S =
  begin
    let request = tuple_case request of ExecuteArgument with
      serviceName  :S
    end
    let nFunctions = port.request(serverAddress  request)
    let proxy = arrayOp.new(:Ok  nFunctions + 1  0)
    for function = 1 upto nFunctions do
      let stub() = begin
        let executeArgument = tuple_case call of ExecuteArgument with
          serviceName  function  unsafe.getArguments(1)
        end
        let result = port.request(serverAddress  executeArgument)
      end
      arrayOp.set(:Ok  proxy  function + 1  stub)
    end
    unsafe.typeCast(proxy  :S)
  end
```

Die Funktion *stub* in der Mitte des Quelltextes wird für jedes Tupelelement mit einer anderen für sie globalen Konstanten *function* versehen. Dadurch unterscheidet sich dann beim Aufruf jede Instanz gegenüber dem entfernten Server. Die Anzahl der Parameter von *stub* variiert natürlich von Fall zu Fall. Deshalb wird auf die aktuellen Argumente von *stub* stets mit Hilfe der Spezialfunktion *unsafe.getArguments*, die sie direkt vom Stack der virtuellen Maschine abholt und in einem Array aggregiert, zugegriffen.

Das Ergebnistupel der Funktion *bind* wird zunächst als Array angelegt, dann mit den Stub-Funktionen als Elementen gefüllt und erst am Ende durch die Funktion *unsafe.typeCast* mit dem passenden statischen Tupeltyp S versehen. Dieser Trick setzt zwar spezielle Kenntnisse der internen Datenrepräsentationen von Tycoon voraus, aber es ist offensichtlich, daß sich in jedem persistenten Objektsystem mit uniformen Objektrepräsentationen (vgl. Abschnitt 3.4) eine naheliegende äquivalente Vorgehensweise findet.

Der erste Aufruf von *port.request* erfragt beim Server, ob ein für eine Bindung geeigneter Dienst vorliegt. Er stellt bereits sicher, daß alle weiteren Operationen obwohl sie per se typunsicher sind, ohne Fehler ablaufen. Bei dem innerhalb der Stub-Funktionen liegenden Aufruf von *port.request* handelt es sich um die im Zusammenhang der Server-seitigen Implementation jeweils vorausgesetzte Client-Anfrage.

Das Scheitern von Bindungsversuchen und sonstige Kommunikationsfehler werden an Clients durch Auslösen einer bestimmten Ausnahme (*portClient.error*) übermittelt. Dabei kann auf den Namen, den dynamischen Typ und die Netzwerkadresse des betroffenen entfernten Dienstes zugegriffen werden. Anwendungsspezifische Ausnahmen, die durch eine entfernte Funktion ausgelöst worden sind, werden zum Client propagiert (inklusive optionaler Ausnahmeargumente).

Spätestens in diesem Abschnitt wird deutlich, welch ideale Systemvoraussetzungen ein persistentes Objektsystem mit einer Sprache höherer Ordnung für die Implementation von Kommunikationsmechanismen bietet. Zum einen dürfte der geringe Aufwand kaum zu unterbieten sein und zum anderen überzeugt das geschaffene Produkt durch Orthogonalität. Es besteht keinerlei Abhängigkeit der oben beschriebenen Mechanismen von den speziellen Typen entfernter Funktionen. Dies bedeutet insbesondere, daß Funktionen höherer Ordnung und Polymorphie in entfernten Diensten verwendet werden können. Ferner können beliebige Module dynamisch zur Laufzeit zu entfernten Diensten werden. Dies gilt sogar für Module, die vor der Entstehung des RPC-Mechanismus geschaffen wurden. Und nicht zuletzt sind alle entfernten Aufrufe statisch typsicher.

Da das unterliegende Modul *port* erst beim Aufruf von *request* eine Socket-Verbindung herstellt, sind Client-Programme ortsunabhängig und damit mobil. Sie bleiben nach Verlagerungen im Netzwerk voll funktionsfähig. Ferner sind Objekte, die Client-Stub enthalten, zwischen Tycoon-Prozessen mobil. Sie können ohne weiteres als RPC-Argument verwendet werden.

5.2.3 Hierarchische Adressierung entfernter Dienste

Die bisherige Adressierung funktioniert nur unter der Bedingung, daß Server immer unter derselben Netzwerkadresse erreichbar sind. Sie dürfen weder auf einen anderen Host verlagert werden, noch ihre Ports wechseln. Außerdem sind zur Herstellung von Kommunikationsverbindungen konkrete Netzwerkadressen erforderlich.

Um Netzwerkadressen *dynamisch* zuteilen zu können, muß eine übergeordnete Adressierungsschicht geschaffen werden. Dabei bietet es sich an, Konzepte persistenter Objektsysteme auf die Kommunikationsprogrammierung zu übertragen. Im Sinne der Unterscheidung zwischen langlebigen zwischen langlebigen Tycoon-Objekten und flüchtigen externen Objekten wird im folgenden zwischen langlebigen Tycoon-Adressen und flüchtigen Netzwerkadressen unterschieden.

Abschnitt 5.2.3.1 stellt das gewählte Adressierungsschema vor. Daraufhin werden die Implementation der dynamischen Adreßauflösung (Abschnitt 5.2.3.2) bei entfernten Aufrufen und der interne Ablauf Dienstauswahl (Abschnitt 5.2.3.3) zum Zwecke der Bindung entfernter Funktionen beschrieben.

5.2.3.1 Das dreistufige Adressierungsschema

Durch Abstraktion von Netzwerkadressen (Ports) kann das Konzept der entfernten Bindung um die feste Zuordnung bestimmter Server reduziert werden. Eine entfernte Bindung bezieht sich dann nur noch auf einen bestimmten entfernten Dienst. Als zusätzliche Möglichkeit folgt daraus die Mobilität von Diensten zwischen verteilten Servern unter Beibehaltung aller bestehenden Client-Bindungen.

Es ist von entscheidender Bedeutung, zwischen der Adressierung bereits gebundener Dienste und der Adressierung zum Zwecke einer Bindung zu differenzieren.

- ▷ Um einen bereits gebundenen Dienst zu lokalisieren genügt ein einziger eindeutiger Identifikator. Weltweit eindeutige Identifikatoren für alle Dienste im Netz können durch einen einfachen abstrakten Datentyp realisiert werden.

  Die transparente Einbindung entfernter Funktionsaufrufe in die Sprache TL bedingt, daß zum Zwecke entfernter Aufrufe keinerlei Adressen explizit vom Client-Programm manipuliert werden.

- ▷ Bei der einer Auswahl gleichkommenden Adressierung eines noch zu bindenden Dienstes kann je nach Anwendung eine unterschiedliche Menge von Attributen relevant sein. Hier ist demzufolge ein möglichst generischer Mechanismus gefordert.

  Im Gegensatz zu obigen abstrakten Dienstidentifikatoren werden die jeweiligen konkreten Dienstattribute im Client-Programm explizit verwendet.

Die Dienstlokalisierung kann im allgemeinen nicht durch eine dynamisch wiederholte Dienstauswahl simuliert werden, weil diese nicht notwendigerweise immer den *selben*, sondern lediglich einen in ihrem Sinne *gleichen* Dienst identifizieren würde. Aufgrund dieser Möglichkeit von „Verwechslungen“ wären keine wirklichen *Bindungen* an entfernte Server-Zustände gegeben. Außerdem verringern Wiederholungen der Dienstauswahl die Performanz entfernter Aufrufe.

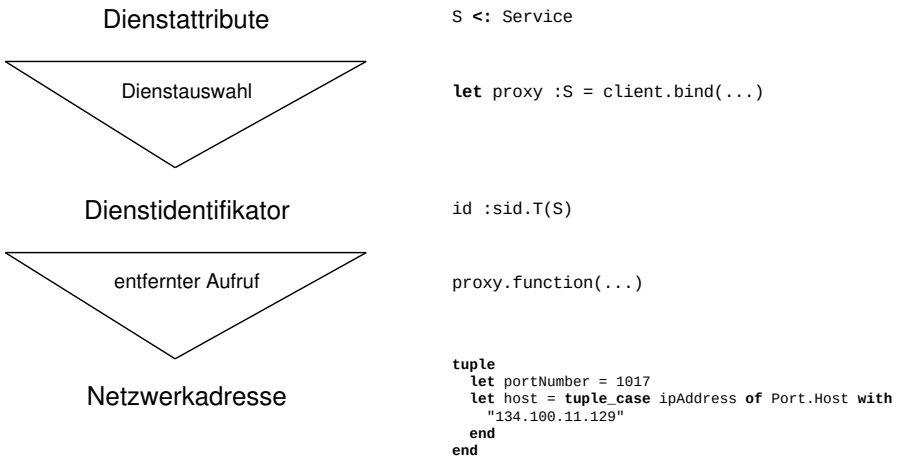

Abbildung 5.14: Die Hierarchie der drei Adressierungsebenen

Unter Berücksichtigung aller oben genannten Gesichtspunkte lassen sich drei Adressierungsebenen identifizieren, die wie in Abbildung 5.14 dargestellt, durch folgende zwei Vorgänge unterteilt, hierarchisch angeordnet sind:

1. Bei der Dienstauswahl wird anhand von Dienstattributen ein Dienstidentifikator bestimmt.
2. Beim Aufruf einer entfernten Funktion wird anhand eines Dienstidentifikators eine konkrete Netzwerkadresse bestimmt.

Diese Vorgänge werden in den nächsten beiden Abschnitten eingehend erläutert, wobei in umgekehrter Reihenfolge vorgegangen wird.

5.2.3.2 Die dynamische Adreßauflösung

Jedem Dienst wird bei seiner erstmaligen Registrierung (*server.register*) ein weltweit eindeutiger Dienstidentifikator zugeordnet. Dabei handelt es sich um einen von der Funktion *sid.new* generierten Wert des abstrakten Datentyps *sid.T*. Die Bezeichnung „sid“ steht für *Service Identificator*. Durch Parametrisierung mit dem Typ des bezeichneten Dienstes wird die Typsicherheit auch auf dieser Adressierungsebene gewährleistet: hat ein Dienst den Typ S, so ist der Typ seines Identifikators *sid.T(S)*.

Wird die Verbindung zwischen Client und Server aus einem der in Abschnitt 5.1.1 aufgeführten Gründe unterbrochen, so benötigen sie anschließend einen Dienst der die Abbildung von Dienstidentifikator auf konkrete Netzwerkadressen leistet. Dieser Abbildungs-Dienst muß zu jedem Zeitpunkt auf jedem Host, auf dem Tycoon-Client/Server-Anwendungen laufen sollen, verfügbar sein. Ferner wird ein Dienst im LAN benötigt, der die lokalen Dienste koordiniert und rechnerübergreifende Anfragen weiterleitet. Und schließlich wäre natürlich ein weltweiter Auskunftsdienst notwendig, um weltweite Mobilität zu ermöglichen.

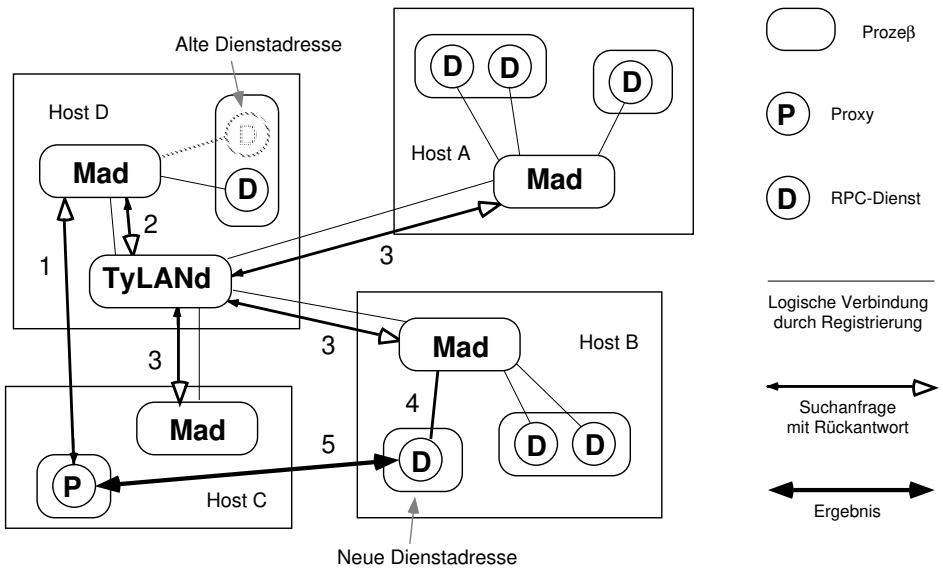

Abbildung 5.15: Auffinden eines Dienstes im LAN

Eine eigene weltweite Infrastruktur aufzubauen erscheint nicht sinnvoll, da bereits genügend generische Lösungen existieren. Hier ist vor allem der Verzeichnisdienst X.500 [Tietz 89] zu nennen. Aufgrund seiner hierarchischen Organisation ließe er sich ohne konzeptionellen Bruch auf die beiden unteren, von Tycoon selbst geleisteten, Dienstebenen aufsetzen. Da die beiden unteren Ebenen jedoch genügen, um die erarbeiteten Kommunikationsprinzipien zu demonstrieren, wurde auf eine Erweiterung der prototypischen Implementation um X.500-Zugriffe verzichtet.

Die dynamische Adreßauflösung vor einem entfernten Aufruf geht vom Client-Stub aus. Dieser versucht als erstes, den gebundenen Dienst an seinem bisherigen Standort anzusprechen. In den meisten Fällen gelingt dieser optimistische Ansatz. Wenn er fehlschlägt, setzt ein bei der Server-seitigen Registrierung des Dienstes vorbestimmtes Suchverfahren ein. Im folgenden wird die hierfür vorgesehene Standardlösung beschrieben.

Zuerst stellt der Client-Stub eine Anfrage an die auf dem Server-Host ansässige Instanz des mit Tycoon realisierten Netzdienstes MAD (*Mapping daemon*). Ähnlich dem *Port Mapper* des Sun-RPC liefert der MAD den aktuellen Port des Dienstes, sofern sich dieser weiterhin auf demselben Server befindet. Andernfalls wird die Anfrage an die nächsthöhere Dienstebene weitergeleitet.

Auf der LAN-Ebene ist der sogenannte *Tycoon LAN daemon* (TYLAND) zuständig. Er leitet die Anfrage einfach unverändert an alle MAD-Instanzen im LAN weiter, wertet deren Antworten aus und liefert das Ergebnis an den initiierenden MAD, der dann schließlich dem Client-Stub antwortet.

Dieses Schema ist in Abbildung 5.15 an einem fiktiven Beispiel ersichtlich. Als Voraussetzung für das dargestellte Szenario wird angenommen, daß das Proxy auf Host C den vergeblichen Versuch unternommen hat, eine Funktion des von Host D nach Host B verlagerten Dienstes

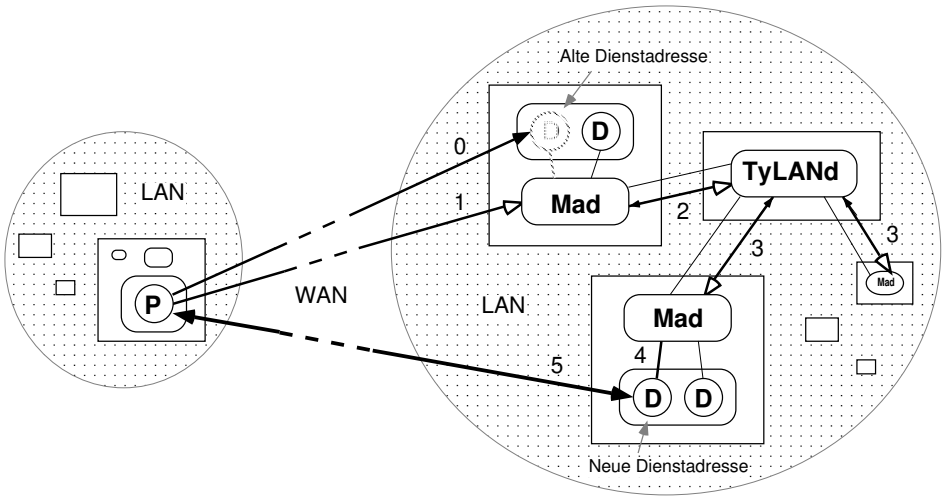

Abbildung 5.16: Auffinden eines Dienstes in einem entfernten LAN

direkt an der alten Adresse aufzurufen. Daraufhin stellt das Proxy eine Anfrage an den MAD auf dem Host (D), auf welchem sich der wiederzufindende Dienst ursprünglich befand (Schritt 1). Dieser wendet sich dann an den TyLANd auf Host D (Schritt 2). Der TyLANd propagiert die Anfrage an alle anderen MADs weiter (Schritt 3). Schließlich wird der vermißte RPC-Dienst auf Host B von dem dort ansässigen MAD wiedergefunden (Schritt 4). Diese Netzwerkadresse wird rückwärts entlang der noch bestehenden Kette entfernter Aufrufe (über den TyLANd und den MAD auf Host D) als Ergebnis an den Client zurückgereicht. Sie wird nun bis auf weiteres immer zuerst vom Client-Stub angesprochen werden. Außerdem wird sie bei der gleich anschließenden Ausführung des hier beabsichtigten entfernten Funktionsaufrufes (Schritt 5) verwendet.

Beim Wiederauffinden entfernter Dienste macht es keinen Unterschied, ob sich der Client und der entfernte Dienst im selben LAN befinden oder nicht. Für den Anfangspunkt der Suche ist nicht der Standort des Clients maßgeblich, sondern ausschließlich der vermutete Standort des entfernten Dienstes. In Abbildung 5.16 ist exemplarisch wiedergegeben, wie das Wiederauffinden eines Dienstes in einem entfernten LAN abläuft. Als Schritt 0 ist hier der erste Versuch, den gewünschten entfernten Dienst anzusprechen, mit aufgeführt. Alle weiteren Schritte entsprechen denen im oben beschriebenen lokalen Fall.

Es ist anzunehmen, daß das beschriebene Verfahren für die meisten Anwendungen ausreicht. Dafür spricht:

- Clients sind uneingeschränkt mobil.
- Server-Verlagerungen finden mit hoher Wahrscheinlichkeit jeweils innerhalb eines LAN statt.

Es sind jedoch auch Anwendungen denkbar, in denen eine höhere Server-Mobilität oder sogar die Mobilität von Diensten zwischen verschiedenen Servern auftritt. Um für alle Fälle gerüstet zu sein, ist die Suchmethode für das Wiederauffinden von Diensten nicht fest vorgegeben.

Stattdessen ist jedem Dienst einzeln eine *eigene* Suchfunktion zugeordnet. Sie wird jeweils bei der Registrierung an den zuständige Dispatcher übergeben und in dessen Diensttabelle erfaßt. Bei anschließenden Client-Anfragen zum Zwecke der Bindung an einen entfernten Dienst liefert der Dispatcher nicht nur einen Dienstidentifikator, sondern auch die korrespondierende Suchfunktion. Letztere wird ein Teil des Client-Stubs.

Die Standardsuchmethode ist in Form der Funktion *server.searchFun* verfügbar. Ihr Quelltext ist in Anhang C aufgelistet. Alternative Suchfunktionen zu schreiben, ist mit den Mitteln der Tycoon-Kommunikationsbibliothek nicht schwierig. Insbesondere sind die Anweisungen für eine Anfrage an einen durch die Angabe eines Hosts identifizierten MAD vielseitig wiederverwendbar. Zum Beispiel können unter Verwendung nachstehender Anfragefunktion folgende Alternativen für Suchfunktionen konstruiert werden:

- Die Anfrage wird stets an ein und derselben MAD gerichtet. Die dazu vorgegebene entfernte Funktion *translate* des MAD hat einen speziellen Parameter (*searchDepth*), durch den festgelegt wird, ob er sie in seinem LAN oder sogar in größere Regionen des Internet propagieren soll (siehe Abschnitt 5.2.4.1).

- Ein Dienst erstellt bei seiner Registrierung eine Liste aller Hosts, auf denen in Zukunft nach ihm gesucht werden soll und registriert eine Suchfunktion, die folgende Funktion der Reihe nach auf die Elemente dieser Liste (ggf. mit variablen Suchtiefen) anwendet.

```
let requestMadOnHost(S <:Service  serviceID :sid.T(S)  host :Port.Host
                     searchDepth :Int) :Port.T
  begin
    (* Adresse des Mad zusammensetzen: *)
    let start :Port.T = tuple madService.portNumber  host end

    (* RPC-Dienst des Mad binden: *)
    let m = portClient.bind(start  :madService.T  madService.name)

    (* Anfrage entfernt aufrufen: *)
    m.translate[13](serviceID  searchDepth)
  end
```

  Auf diese Weise wird eine dienstspezifische Suche in vorbestimmten Netzwerkregionen organisiert.

Um späte Bindungen nach Art eines *Object Request Brokers* zu realisieren, müßte zum einen der initiale Versuch, den Dienst an seiner vorherigen Adresse aufzurufen, aus dem Client-Stub entfernt werden. Zum anderen würde auf Dienstidentifikatoren verzichtet. Eine Suchfunktion für einen spät zu bindenden Dienst müßte einfach den Vorgang der im folgenden Abschnitt beschriebenen Dienstauswahl wiederholen.

[13] Die Funktionsweise dieser Funktion wird in Abschnitt 5.2.4.1 erläutert.

5.2.3.3 Der interne Ablauf der Dienstauswahl

Die Module *client* und *server* realisieren einen RPC-Mechanismus, durch den TL zu einer einheitlichen Sprache für die Dienstbeschreibung, die Client-Implementierung, die Dienstimplementierung und die Einbindung externer Dienste wird. In Bezug auf die Signaturen und die Ausführungsfunktionalität entfernter Aufrufe ändert sich dabei im Vergleich zu den Modulen *portCient* und *portServer* (siehe Abschnitt 5.2.2.2) nichts.

Auf der rechten Seite von Abbildung 5.14 sind beispielhafte TL-Ausdrücke bestimmten Adressierungsebenen und -vorgängen zugeordnet. Bindungen an entfernte RPC-Dienste entstehen durch Aufrufe folgender Art:

```
(* Diensttyp: *)
Let S = Tuple ... (* Funktionssignaturen *) ... end

(* Dienstbindung: *)
let proxy = client.bind(:S  (* Attribute, etc.: *) ...)
```

Betrachten wir zunächst die vollständige Signatur der Funktion zur Dienstauswahl und -bindung:

```
bind(Dyn S <:Service  Dyn Attributes <:Ok  predicate(:Attributes) :Bool
     searchDepth :Int) :S
```

Diese Funktion wendet sich an den MAD auf dem eigenen Host. Je nach durch den Parameter *searchDepth* vorgegebener Suchtiefe werden nur auf diesem Host, im LAN oder in größeren Regionen des Internet vorhandene Dienste mit den vorgegebenen Eigenschaften verglichen. Die Überprüfung eines einzelnen entfernten Dienstes findet jeweils in einem MAD statt. Sie erfolgt in folgenden Schritten:

1. Es wird geprüft, ob der Typ des vorliegenden Dienstes ein Subtyp des verlangten Diensttyps S ist.
2. Es wird geprüft, ob sein Attributtyp ein Subtyp des verlangten Attributtyps *Attributes* ist.
3. Das Prädikat *predicate* wird auf seine aktuellen Attribute angewendet. Wenn diese Auswertung **true** ergibt, ist ein passender Dienst gefunden worden.
4. Bei erfolgreicher Dienstauswahl wird der Dienstidentifikator des Dienstes zusammen mit der dem Dienst zugeordneten Suchfunktion an den Client geliefert und dort in neu erzeugten Stub-Funktionsabschlüssen gekapselt.

Dieses Szenario entspricht ziemlich genau der *ODP Trading Function* [ISO-ODP 95]. Tatsächlich erfüllt die hier vorgestellte Architektur alle wichtigen Grundfunktionen des *Trading*[14].

Man kann TL *als Dienstbeschreibungssprache* auffassen, die im Gegensatz zu anderen (wie z.B. CORBA-IDL) algorithmisch vollständig ist, sowie ein Typsystem höherer Ordnung aufweist.

[14] Die Tycoon-Zugriffskontrolle wird in der vorliegenden Arbeit lediglich nicht vorgestellt. Gleichwohl stehen für diesen Bereich adäquate Lösungen bereit [Rudloff et al. 95] (vgl. Abschnitt 8.3).

Denn mit Hilfe der externen Programmierschnittstellen von TL können sprachlich heterogene Komponenten mit etwa dem gleichen Aufwand integriert werden wie in sogenannten offenen verteilten Objektsystemen (z.B. CORBA-Implementationen).

Wenn durch jeweils einen MAD große Anzahlen von Diensten zu verwalten sind, kommt allerdings die Ineffizienz der iterativen *Subtyp*-Vergleiche zum Tragen. Wie in Abschnitt 5.1.1 erläutert, kann jedoch durch die Forderung der *Typgleichheit* anstelle der Subtypbeziehung von Attributtypen Abhilfe geschaffen werden. Die Dienstauswahl in einem MAD verläuft dann wie folgt:

1. Durch Hashing oder eine Suche in einer Baumstruktur (z.B. Binärbaum, B-Baum) wird die Liste derjenigen Dienste ermittelt, deren Attributtyp dem in der Anfrage des Client spezifizierten Typ *Attributes* gleicht. Als Voraussetzung für diesen Schritt speichert der MAD im Vorwege Dienste gleichen Attributtyps jeweils in einer gemeinsamen Liste und trägt diese Liste mit dem Attributtyp als Schlüssel in einen Baum, bzw. eine Hash-Tabelle ein. Die Typgleichheit kann bei dieser Methode durch wechselseitige Subtyptests überprüft werden.

2. Es wird geprüft, ob die Typen der vorliegenden Dienstes Subtypen des verlangten Diensttyps S sind.

3. Das Prädikat *predicate* wird auf die aktuellen Attribute der Dienste angewendet. Wenn diese Auswertung **true** ergibt, ist ein passender Dienst gefunden worden.

Die dynamische Auswertung der Attribute wird bis zuletzt aufgeschoben, weil sie typischerweise der aufwendigste Einzeltest ist. Sie erfordert eine Anfrage des MAD an den zuständigen Server und eine potentiell komplexe Berechnung des Servers.

Da MADs als allgemeine Netzdienste prinzipiell *anwendungsunabhängig* sind, sollten sie das auf strukturelle *Gleichheit* von Attributstypen abzielende Bewertungsverfahren verwenden. Im gegenwärtigen MAD-Prototyp ist nur das eingangs beschriebene, einfachere Protokoll implementiert.

Allerdings ist in beiden Protokollen gewährleistet, daß das Auswahlprädikat (*predicate*) vor seiner Bewertung jeweils auf den MAD übertragen wird. Ohne diese Ausnutzung der Mobilität von Funktionsabschlüssen würden bei einer weiträumigen Suche, welche eine große Anzahl von Dienstbewertungen mit sich bringt, sowohl die Rechenleistung des Client-Hosts als auch sein Netzwerkanschluß schnell zu Engpässen. Die Verteilung des Prädikats auf die MADs schafft hier entscheidende Abhilfe, indem sie die Suchanfrage parallelisiert und den Netzverkehr senkt.

5.2.4 Die Implementation der Netzdienste

Die Implementationen der Netzdienste erfolgt mit den Mitteln der Sprache TL wobei bereits oben beschriebene Neuentwicklungen wie die Module *port*, *portClient*, usw. zum Einsatz kommen. Aufgrund dieser Entwurfsentscheidung können die Netzdienste die volle Bandbreite der Mobilität im Tycoon-System nutzen, was als Verschmelzung von System- und Anwendungsprogrammierung angesehen werden kann. Besonders vorteilhaft ist die durchgängige Mobilität von Funktionsabschlüssen zwischen Anwendungsprogrammen und Netzdiensten (siehe voriger Abschnitt). Diese Aspekte werden in der folgenden Beschreibung der Implementation des *Mad* konkretisiert.

5.2.4.1 Der Mapping-Daemon

Ein MAD verwaltet eine Liste aller auf seinem Host verfügbaren Dienste und einen Pool von Ports, die er den Dispatchern, welche die Dienste ausführen, dynamisch zuweist. Er muß seinen Datenbestand aktualisieren können und auf Ausfälle von Diensten robust reagieren. Ein explizites Abmelden von Diensten gegenüber dem MAD ist nicht notwendig. Denn dieser muß ohnehin eine automatische Bestandsbereinigung auf der Basis fehlschlagender Anfragen bei den Dispatchern durchführen.

Die Implementation des MAD gliedert sich in ein Hauptprogramm (Modul *mad*) und ein Schnittstellenmodul (*madService*), das mit Hilfe des Moduls *portServer* alle wesentlichen Operationen des Hauptprogramms als RPC-Dienst verfügbar macht. An diesen Dienst binden sich alle Clients auf dem jeweiligen Host, indem sie die Funktion *madService.bind* aufrufen.

Da der Netzdienst MAD an wohlbekannten Adressen zur Verfügung stehen muß, eignen sich die Module *portClient* und *portServer* ideal zu seiner Implementation. Das Modul *madService* definiert dazu eine konkrete Netzwerkadresse und es implementiert sämtliche auf der Client-Seite notwendigen Maßnahmen, um obigen RPC-Dienst zu binden.

```
module madService
import
  :CSTypes  :Port  netDB  portClient
export
  (* Server-Unterstützung: *)
  let name = "Tycoon Mapping Daemon"
  let portNumber = 1500    (* eine beliebige unbelegte Nummer *)

  let port() :Port.T = tuple
    portNumber
    tuple_case dnsName of Port.Host with
      netDB.getHostName()  (* liefert die eigene Internet-Adresse *)
    end
  end

  (* Client-Unterstützung: *)
  let bind() = portClient.bind(port()  :MadService.T  name)
end;
```

Die lokale Funktion *port* liefert die bewußte Netzwerkadresse, welche aus der Internet-Adresse des lokalen Hosts und der festen Port-Nummer (1500) aller MADs zusammengesetzt ist. Der Typ *MadService.T* repräsentiert die Schnittstelle, auf der alle Client-Anfragen operieren.

```
Let T = Tuple
  register(Dyn S <:Service  Dyn Attributes <:Ok  :Port.T) :sid.T(S)
  put(S <:Service  :sid.T(S)  Dyn Attributes <:Ok  :Port.T) :Ok
  get(S <:Service  Dyn Attributes <:Ok  predicate(:Attributes) :Bool
      searchDepth :Int) :sid.T(S)
  translate(S <:Service  :sid.T(S)  searchDepth :Int) :Port.T
  getFreePortNumber() :Int
end
```

In dieser Schnittstelle sind der Einfachheit halber auch die Anfragemöglichkeiten für Dispatcher und den TyLAND vereint. Die Aufgaben der einzelnen Funktionen sind in Abbildung 5.17 zusammengefaßt dargestellt und werden im folgenden näher erläutert.

`register`	registriert einen neuen Dienst mit Diensttyp, Attributtyp und liefert einen Dienstidentifikator
`put`	weist einem Dienst einen neuen Port zu
`get`	sucht anhand Diensttyp, Attributtyp und Prädikat einen passenden Dienst
`translate`	übersetzt einen Dienstidentifikator in eine konkrete Netzwerkadresse
`getFreePort`	stellt einen freien Port bereit

Abbildung 5.17: Funktionen des Mapping Daemons (MAD)

Die Funktion *register* dient Dispatchern dazu, einen Dienst in die internen Diensttabelle des MAD einzutragen. Diese geben die dynamischen Typrepräsentationen des Dienstes und seiner Attribute sowie ihre eigene Port-Adresse an und erhalten einen abstrakten Dienstidentifikator zurück.

Mit Hilfe der Funktion *put* kann ein Dispatcher den Attributtyp und die Netzwerkadresse eines Diensteintrages in der Tabelle des MAD ändern. Diese Möglichkeit wird zur Zeit nur in einem Fall genutzt. Wenn ein (persistenter) Server-Prozeß neu gestartet wird, fordert er automatisch mit Hilfe von *getFreePortNumber* eine frei verfügbare Port-Nummer an und weist sie mit *put* den Diensteinträgen im MAD zu.

Die Funktion *get* dient der in Abschnitt 5.2.3.3 beschriebenen Dienstauswahl.

Die Funktion *translate* leistet die Übersetzung eines abstrakten Dienstidentifikators in eine konkrete Netzwerkadresse.

1. Durch einen Hash-Zugriff (mit dem Dienstidentifikator als Schlüssel) auf die Diensttabelle des MAD wird die vermutliche Port-Adresse des Dienstes ermittelt. Wenn kein entsprechender Eintrag vorhanden ist, wird mit Schritt 5 fortgefahren.
2. Dann wird durch eine Standardanfrage (*port.request*) an den zuständigen entfernten Dispatcher festgestellt, ob der Dienst tatsächlich noch angeboten wird.
3. Falls dem so ist, stimmt die Port-Adresse noch und sie wird als Ergebnis zurückgegeben.
4. Falls nicht, wird der Eintrag gelöscht.
5. Wenn der Parameter *searchDepth* als Suchtiefe *searchDepthHost* vorgibt, endet die Übersetzung erfolglos. Andernfalls wird die Anfrage an die nächsthöhere Instanz, den TyLAND propagiert.

Die zweite Stufe bei der Dienstsuche ist der Gegenstand des nun folgenden Abschnittes.

5.2.4.2 Der LAN-Daemon

Alle MADs eines LANs werden von einem *Tycoon LAN Daemon* (TyLAND), der als eigenständiger Prozeß auf genau einem Host des LANs zu installieren ist, in einer Liste registriert. Dessen Aufgabe ist ausschließlich die eines Vermittlers. Mit der Registrierung von RPC-Diensten ist er nicht befaßt.

Sowohl der Programmaufbau als auch die RPC-Schnittstelle des TyLAND ähneln einer Untermenge der Funktionalität des MAD. Der Diensttyp für entfernte Anfragen, *TyLANdService.T* ist wie folgt definiert:

```
Let T = Tuple
  register(madAddress :Port.T) :Ok
  get(S <:Service  Dyn Attributes <:Ok  predicate(:Attributes) :Bool
      searchDepth :Int) :sid.T(S)
  translate(S <:Service  s :sid.T(S)  searchDepth :Int) :Port.T
end
```

Die Funktionen *get* und *translate* haben die gleichen Aufgaben wie ihre Pendants im MAD (siehe Abschnitt 5.2.4.1). Sie werden einfach per RPC an alle MADs weitergeleitet. Im Fall von *translate* werden dabei für jeden einzelnen MAD je einmal folgende beiden Anweisungen ausgeführt:

```
let m = portClient.bind(addressOfCurrentMad  :madService.T madService.name)

let address :Port.T = m.translate(:S  s  searchDepthHost)
```

Die Suchtiefe *searchDepthHost* verhindert hier Endlosschleifen.

Für *get* ergibt sich analog zu *translate* folgender Aufruf:

```
let id :sid.T(S) = m.get(:S  :Attributes  predicate  searchDepthHost)
```

`register`	registriert die Netzwerkadresse eines MAD
`get`	sucht anhand Diensttyp, Attributtyp und Prädikat einen passenden Dienst
`translate`	übersetzt einen Dienstidentifikator in eine konkrete Netzwerkadresse

Abbildung 5.18: Funktionen des Tycoon LAN Daemons (TYLAND)

Die Funktion *register* wird von jedem MAD bei seinem Programmstart aufgerufen, um seine Netzwerkadresse beim TYLAND registrieren zu lassen. Abbildung 5.18 faßt die Funktionen des TYLAND-RPC-Dienstes und ihre Aufgaben zusammen (vgl. Abbildung 5.17).

Kapitel 6

Migrierende persistente Threads

Aufgrund der Mobilitätsorientierung entfernter Funktionsaufrufe sind im Tycoon-System Objekte beliebigen Typs mobil. Eine Besonderheit mit hohem Potential für elegante Lösungen komplexer Anwendungsprobleme ist die Mobilität von Threads. Dies gilt um so mehr in persistenten Objektsystemen, in denen sich auf natürliche Weise das Konzept persistenter Threads ergibt.

Im ersten Teil dieses Kapitels werden weitreichende Perspektiven aufgezeigt, die sich durch persistente migrierende Threads für die aktivitätsorientierte Programmierung eröffnen. Der zweite Teil des Kapitels ist Implementationstechniken gewidmet, die Persistenz und Mobilität von Threads unterstützen und persistente Migrationen von Threads verwirklichen.

6.1 Thread-Programmierung im Tycoon-System

Unter *Multi-Threading* wird die Möglichkeit verstanden, innerhalb eines durch eine Programmaktivierung entstandenen Betriebssystemprozesses gleichzeitig mehrere nebenläufige Programmausführungen (Evaluationen), die als *Threads*[1] bezeichnet werden, ablaufen zu lassen. Je nach Hard- und Software-Architektur kann es sich dabei um echte Parallelität handeln oder aber sie wird mittels *time slicing* einer von mehreren Threads geteilten CPU simuliert.

Abbildung 6.1 stellt die wesentlichen Bestandteile des Ausführungszustandes (Evaluationszustandes) eines Threads dar. Ein Thread wird durch eine Datenstruktur repräsentiert, die im wesentlichen die Inhalte aller relevanten Register der Ausführungseinheit wiedergibt. Diese Register, insbesondere der Stapelzeiger und der Programmzeiger, verweisen transitiv auf alle weiteren Bestandteile des Evaluationszustandes. Dabei kommt es vor, daß sich die transitiven Hüllen mehrerer Threads partiell überdecken. Auf diese einfache Weise teilen Threads Programmressourcen aller Art wie z.B. Routinen, Variablen, Dateien und Kommunikationskanäle.

Der nächste Abschnitt führt in die Programmierung mit Threads in TL ein. Zunächst werden nur Thread-Operationen und Eigenschaften betrachtet, die abgesehen von der Spracheinbet-

[1] Threads werden auch *lightweight processes* [Sun 88; Cardelli et al. 88; Keppel 93] genannt, da ihre Verwaltung (insbesondere Erzeugung und Kontextwechsel) im Vergleich zu (Betriebssystem-) Prozessen erheblich weniger Systemressourcen (vor allem Zeit und Speicherplatz) erfordert.

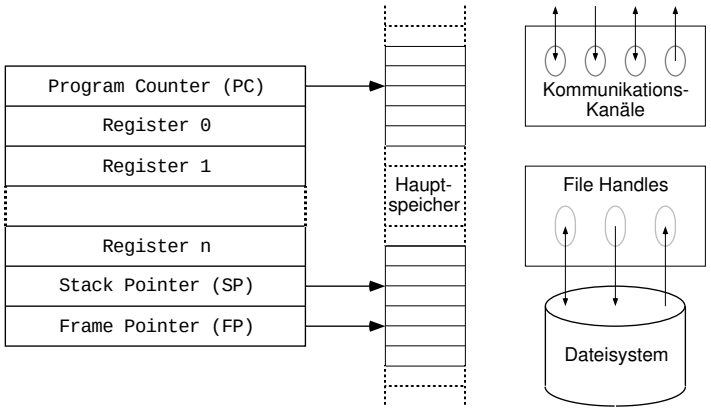

Abbildung 6.1: Bestandteile des Ausführungszustandes eines Threads

tung auch in anderen Systemen zu finden sind. In Abschnitt 6.1.2 werden dann persistente Threads vorstellt und in Abschnitt 6.1.3 folgt die Thread-Migration.

6.1.1 Threads als typisierte Objekte erster Klasse

Eine besonders wichtige Problemstellung bei konkreten Realisierungen des Thread-Konzeptes ist die Einbettung in die Programmiersprache. Durch konsequente Verfolgung des *add-on*-Ansatzes ist es im Rahmen der vorliegenden Arbeit gelungen, Threads orthogonal in das sprachliche Gefüge von TL zu integrieren. Unter Verzicht auf syntaktische Besonderheiten werden Threads einfach durch Werte eines abstrakten Datentyps repräsentiert. Daß sie „gewöhnliche" TL-Werte sind, bringt insbesondere folgende Vorteile mit sich:

- Threads sind Sprachobjekte erster Klasse. Sie können als Funktionsargumente und -rückgabewerte fungieren, (auf Identität) verglichen werden und Konstanten, Variablen sowie Elementen komplexer Datenstrukturen (Arrays, Tupel, Listen, Mengen, etc.) zugewiesen werden.
- Sämtliche auf Threads anwendbaren Operationen sind *statisch typisiert*.

Die konkrete Repräsentation von TL-Threads implementiert das Standardmodul *thread*, dessen Schnittstelle in Anhang D.1 aufgeführt ist. Es exportiert den (parametrisierten) abstrakten Datentyp *thread.T* und eine Menge von Funktionen, die allgemein übliche[2] sowie weitere nützliche Thread-Operationen (die sich in anderen Systemen nicht so leicht realisieren lassen) durchführen.

Jeder Tycoon-Thread führt jeweils eine Funktion aus, die bei seiner Erzeugung angegeben wird. In folgendem Programmfragment wird eine neuer Thread t erzeugt, der umgehend mit der Ausführung der zuvor definierten Funktion f beginnt. Ein solches „Thread-Skript" weist

[2]Hier dienten der POSIX-Standard [POSIX 90], *P-Threads* [Keppel 93] und die Bibliotheken anderer Programmiersprachen wie z.B. Modula-3 [Nelson 91; Horning et al. 93] als Vorbilder zur Bestimmung einer Ausgangsmenge nützlicher Funktionen.

stets einen Parameter auf, an den das durch die Funktion *thread.fork* involvierte Tycoon-Laufzeitsystem den ausführenden Thread als Argument übergibt, damit dieser reflexive Operationen ausführen kann.

```
let f(self :thread.T(Int)) :Int =   (* Thread-Skript *)
  begin
    ...
    thread.kill(self)   (* Beispiel für eine reflexive Operation. *)
    ...
  end

let t = thread.fork(f)    (* Erzeugung eines Threads, der f ausführt. *)
```

Der abstrakte Datentyp *thread.T* ist genaugenommen ein *Typoperator*, der mit dem Ergebnistyp der Funktion (R) parametrisiert ist.

```
T(R <:Ok) <:Ok
```

Auf diese Weise sind Thread-Ergebnisse statisch typisiert, obwohl ihr jeweiliger konkreter Typ beliebig gewählt werden kann. Durch folgenden Aufruf wird das Ergebnis obigen Threads t abgerufen:

```
let var x :Int = thread.join(t)
```

Ein Thread, der diese Anweisung ausführt, blockiert, bis die Funktion f in obigem Thread t beendet ist und dessen Ergebnis vorliegt. Daraufhin wird die Ausführung des wartenden Threads fortgesetzt, wobei als erstes der Ergebniswert von t an die Variable x zugewiesen wird. Eine mangelnde Übereinstimmung des hier verwendeten Typs (*Int*) mit dem Ergebnistyp des Threads wird bereits vom TL-Übersetzer erkannt und als Fehler gemeldet.

Threads eignen sich hervorragend für die Modellierung lokaler Agenten, die im Hintergrund verharren, bis sie bei bestimmten Anlässen aktiviert werden. Zum Beispiel kann wie folgt eine stündliche Überprüfung der eingehenden elektronischen Post (E-Mail) auf speziell gekennzeichnete Nachrichten veranlaßt werden.[3]

```
(* Eine Stunde in Sekunden: *)
let oneHour = 60.0 * 60.0


(* Ein Prädikat, das die Kennzeichnung von Nachrichten testet: *)
let urgent(m :Mail) :Bool =
  string.contains(m.title  "URGENT")
```

[3] Die Handhabung der E-Mail-Anbindung an Tycoon ist hier vereinfacht dargestellt. Für die Implementation der (noch) fiktiven Funktionen des Moduls *mail* sind jedoch bereits alle wesentlichen systemtechnischen Voraussetzungen vorhanden [Willomat 96] - insbesondere die Kommunikation mit einem entfernten POP-Server.

```
let agentScript(self :thread.T(Ok)) :Ok =
  loop
    thread.sleep(self  oneHour)

    (* Neue Mails vom entfernten POP-Server einlesen: *)
    let newMails = mail.popIncoming()

    (* Ist als dringend gekennzeichnete Mail dabei? *)
    if iter.some(newMails  urgent) then
      loop
        case alert("Urgent Mail - enter mail tool now?") of
        when yes then
          mail.readIncoming()   (* Eingegangene E-Mails bearbeiten. *)
          exit
        when no then
          exit
        when wait with answer then
          (* Noch eine Weile aufschieben: *)
          thread.sleep(self  answer.sleepInterval)
        end
      end
    end
  end

(* Die Aktivierung des Agenten: *)
thread.fork(agentScript)
```

In einem nicht orthogonal persistenten System wäre obiger Agent im Falle einer Programmbeendigung mit anschließendem Neustart durch explizit ausprogrammierte Anweisungen erneut zu etablieren. Für den Fall, daß er sich bei der Beendigung gerade in einer *sleep*-Operation befand, bereitet dies keine besonderen Umstände. Es genügt, durch die Wiederholung der Anweisung *thread.fork(agentSkript)* einen neuen Thread zu aktivieren, um ein kontinuierliches Verhalten des imaginären Agenten zu gewährleisten. Um jedoch ein Neuaufsetzen kurz vor der *if*-Anweisung zu gewährleisten, müßte das gesamte Skript umorganisiert werden. Dabei müßten sowohl die Stelle im Skript, bis zu welcher der Thread gelangt ist, als auch der Inhalt aller relevanten Variablen (hier: *newMails*) persistent erfaßt werden. Noch komplizierter wäre es, ein nahtloses Aufsetzen an beliebiger Stelle der Funktion *agentScript* zu realisieren.

Angesichts der bei diesem simplen Beispiel bereits auftretenden Schwierigkeiten wird deutlich, daß erhebliche Komplikationen auftreten können, wenn die Persistenz komplexer Aktivitäten zu organisieren ist. In diesem Zusammenhang fällt auf, daß heutige kommerzielle Anwendungsprogramme (z.B. Bürosoftware) an nicht-trivialen Stellen meist keinerlei Persistenzunterstützung von Abläufen leisten. Zum Beispiel obliegt es üblicherweise dem Benutzer, einen in einem vorangehenden Programmlauf erreichten Zustand der Benutzeroberfläche wiederherzustellen.

6.1.2 Persistente Threads

In den weitaus meisten Programmiersystemen sind Threads flüchtige Objekte. Ihre Lebensdauer ist durch die des sie umgebenden Betriebssystemprozesses begrenzt. Um nach einem Programmneustart einen bestimmten Evaluationszustand automatisch wiederherstellen zu können, sind folgende Maßnahmen *explizit* zu programmieren (vgl. Einleitung, Seite 8):

1. Eine persistent speicherbare Datenrepräsentation, die alle relevanten Einzelheiten eines Evaluationszustandes erfaßt.
2. Eine Routine, die eine solche Datenstruktur erzeugt und speichert.
3. Eine Routine, die umgekehrt anhand einer gespeicherten Datenstruktur den gewünschten Evaluationszustand reetabliert.

Es ist leicht einzusehen, daß dieser Ansatz, Thread-Zustände persistent zu machen, bei komplexeren Anwendungen zu erheblichem Aufwand führt. Die Maßnahmen sind stark anwendungsabhängig und müssen daher in vielen Fällen komplett neu getroffen werden. Außerdem wird die eigentliche Programmstruktur stark von dieser expliziten Erfassung des Evaluationszustandes von Anwendungsprozessen beeinträchtigt, da alle relevanten Zustandsvariablen gemeinsam zugreifbar sein müssen, um sie gemeinsam persistent zu speichern. Unter diesem Zwang müssen in aller Regel verschachtelte Sichtbarkeitsbereiche aufgebrochen werden, was die Programm-Modularität erheblich mindert. Auch dieser Umstand begünstigt eine hohe Fehleranfälligkeit.

Der notwendige Aufwand und die mit ihm verbundenen Komplikationen lassen sich reduzieren, indem die Menge der Stellen in einem Thread-Skript, an denen eine persistente Speicherung auftreten kann bzw. soll, eingeschränkt wird. Dies impliziert jedoch Beschränkungen der Nebenläufigkeit. Es bestehen folgende Alternativen:

- ▹ Threads werden unabhängig voneinander asynchron gespeichert, was zur Folge hat, daß sie nach einem Neustart an verschiedenen „Zeitpunkten“ aufsetzen. Dabei können sie ohne weiteres inkonsistente Annahmen über globale Zustände aufweisen.
- ▹ Gruppen von Threads werden aufgrund expliziter Anweisungen an dedizierten Stellen in ihren Skripten synchron gespeichert. Eine solche Anweisung veranlaßt einen Thread ggf. seine Ausführung zu suspendieren, bis sich auch alle anderen Threads an einer zur Speicherung geeigneten Stelle befinden. Es ist jedoch nur unter erheblichen Einschränkungen der Programmgestaltung möglich, zu garantieren, daß stets alle vorgesehenen Threads in hinreichend kurzen Intervallen (bzw. überhaupt) an solche Stellen gelangen.

In beiden Fällen ist es sehr aufwendig, ein funktionierendes Miteinander mehrerer Threads zu gewährleisten. Zusammen mit dem Aufwand für die persistente Speicherung von Evaluationszuständen ergibt sich insgesamt ein extrem hoher Anteil *systembedingter* Programmkomplexität für die Realisierung langlebiger Aktivitäten. Bezeichnend für diese Problematik ist nicht zuletzt, daß die meisten heutigen Anwendungen, auch wenn sie von Multi-Threading Gebrauch machen, gewöhnlich nur *passive* Datenstrukturen persistent speichern.

In einem persistenten Objektsystem gelangt man auf natürliche Weise zum Konzept persistenter Threads [Munro 93; Matthes, Schmidt 94]. Evaluationszustände werden im Rahmen der allgemeinen Transaktionsverarbeitung eines persistenten Objektsystems *implizit* gespeichert und wiederhergestellt. Dabei erfolgt stets eine synchrone Speicherung aller vorhandenen Threads, wodurch deren globale Konsistenz von Seiten des Systems unbeeinträchtigt bleibt. Da die Speicherung eines Evaluationszustandes für den Programmierer keinerlei Aufwand bedeutet, kann sie in jedem Thread-Skript an beliebigen Stellen auftreten.[4]

Im Tycoon-System können Anwendungsprogramme abgesehen von technischen Beschränkungen (Performanz, Speicherplatz) beliebig viele unabhängige Threads erzeugen, die auf eine geteilte Menge persistenter Daten, Module und wiederum Threads in einem gemeinsamen Objektspeicher zugreifen. Die Anzahl passiver (d.h. blockierter, suspendierter oder terminierter) Threads kann um mehrere Größenordnungen höher liegen als die Anzahl der gerade aktiven Threads, denn sie ist nur durch den im Objektspeicher verfügbaren Sekundärspeicherplatz beschränkt. Daher eignen sich persistente Threads zur Modellierung massenhaft auftretender Aktivitäten, die überwiegend ruhen und über längere Zeiträume hinweg sporadisch vorangetrieben werden.

Insgesamt sind persistente Threads eine ideale Repräsentationsformen zur Realisierung eines aktivitätsorientierten Stils der Modellierung verteilter Informationssysteme, der von Skripten in Taxis [Borgida et al. 93], prozeßorientierten Spezifikationen in Estelle, Lotos oder SDL [Turner 93] oder von (visuellen) Prozeßsprachen von Workflow-Management-Tools wie z.B. Regatta [Swenson 93] her bekannt ist.

In [Matthes, Schmidt 94] wird aufgezeigt, wie verteilte Arbeitsabläufe mit Hilfe persistenter Threads vereinfacht werden können. Dazu dient als Beispiel die Begutachtung der zu einer Konferenz eingereichten schriftlichen Beiträge (*Papers*). Letztere werden durch folgenden Datentyp, der einen persistenten Thread als Attribut vorsieht, repräsentiert:

```
Let Paper = Tuple
  contents :Tuple title, authors, abstract, text :String end
  reviewer :Person
  rating :Rating
  refereeActivity :thread.T(Ok)
end
```

Vom Standpunkt des Vorsitzenden des Programmkommitees aus muß dafür gesorgt werden, daß jedes der in einer lokalen Datenbank erfaßten Papers individuell begutachtet wird. Dementsprechend wird für jedes Paper eine eigene Begutachtungsaktivität als Thread gestartet:

```
for each p in db.submissions do
  p.refereeActivity := thread.fork(fun(self :thread.T(Ok))  reviewPaper(p))
end
joinAll(select p.refereeActivity from p in db.submissions)
```

[4] Ausnahmen ergeben sich durch die Integration externer Diensterbringer, da diese nicht der Kontrolle des Laufzeitsystems eines persistenten Objektspeichersystems unterliegen. Einzelheiten werden in Abschnitt 7.6 erläutert.

Die Funktion *joinAll* akzeptiert eine Menge von Threads als Argument und blockiert, bis all jene Threads terminiert sind.

```
let joinAll(threads :set.T(thread.T(Ok))) :Ok =
  for each t in threads do thread.join(:Ok  t)
end
```

Generell können gewöhnliche Anfragetechniken benutzt werden, um Operationen auf Kollektionen von Threads auszuführen.

Die Aktivität der individuellen Zuweisung eines Gutachters und der Erfassung der Bewertung ist wie folgt modelliert:

```
let rec reviewPaper(p :Paper) :Ok =
 begin
  p.reviewer := chooseReviewer(availableReviewers)
  sendPaperToReviewer(p.reviewer  p.contents[5])   (* Netzwerkkommunikation! *)
  try
   p.rating := waitForReview(p.reviewer)          (* Netzwerkkommunikation! *)
  when reviewerNotAvailableExc then
   reviewPaper(p)
  end
 end
```

Zweckmäßigerweise finden die einzelnen Gutachtertätigkeiten an überall in der Welt verstreuten Arbeitsplätzen statt. Der Aufruf *sendPaperToReviewer* dient dazu, die logische Begutachtungsaktivität auf einen entfernten Host zu verlagern und der Aufruf *waitForReview*, welcher die Bewertung zurückliefert, involviert ebenfalls eine Netzwerkkommunikation. An dieser Stelle deuten sich die Grenzen der Aktivitätsmodellierung auf der Basis persistenter, aber nach wie vor *stationärer* Threads an. Aus Anwendungssicht durchgängige Abläufe werden in Thread-Skripte zergliedert, die jeweils Unterbrechungen durch Netzwerkkommunikationen nach Art des Client/Server-Paradigmas vorsehen.

[5] Statt *p.contents* steht an dieser Stelle im Original [Matthes, Schmidt 94] der Bezeichner *p*. Es muß jedoch angenommen werden, daß damit *nicht* die Mobilität von Threads vorweggenommen werden sollte, da eine Fortführung des in *p* enthaltenen Threads offensichtlich keine sinnvollen Anweisungen zur Folge hätte. Der durch *p.refereeActivity* bezeichnete Thread ist nämlich eben jener, welcher gerade die Funktion *reviewPaper* mit bewußtem *p* als Argument ausführt. Er hätte demnach eine Kopie von sich selbst verschickt, die nun ebenfalls als nächstes auf eine Antwort des Gutachters *p.reviewer* warten würde...

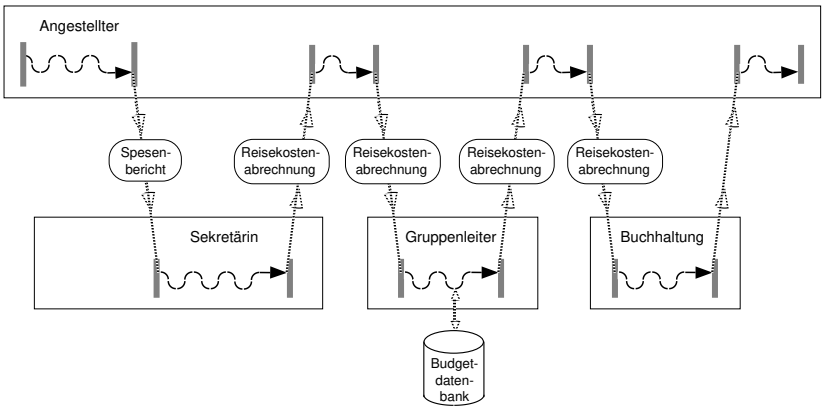

Abbildung 6.2: Modellierung des Reisekosten-Workflows im Client/Server-Stil

6.1.3 Migrierende Threads

In den weitaus meisten Programmiersprachen (besonders in den gebräuchlichsten wie z.B. *C*, *C++*, Fortran, COBOL) ist die Aufteilung migrierender Aktivitäten, die aus der gegebenen Anwendungssicht einen einheitliche logische Prozesse darstellen, in stationäre Subaktivitäten *systembedingt*. Alle relevanten Informationen müssen explizit dargestellt und notwendigerweise *statisch* verteilte Programmfragmente gezielt aufeinander abgestimmt werden. Es migrieren nur passive Daten als Argumente entfernter Anfragen, wobei der makroskopische Ablauf in einem Adreßraum verbleibt.

Dieses Verhalten ist in Abbildung 6.2 am Beispiel einer verteilten Bearbeitung einer Reisekostenabrechnung wiedergegeben. Der hier dargestellte Workflow verläuft wie folgt:

1. Er beginnt in einem Prozeß auf dem Host eines Angestellten, der einen Spesenbericht eingibt.
2. Daraufhin migriert die Aktivität zum Arbeitsplatzrechner einer Sekretärin, die anhand der informellen Angaben des Spesenberichtes eine formal korrekte Reisekostenabrechnung erstellt.
3. Nach einer weiteren Aktivitätsmigration begutachtet der Gruppenleiter die Reisekostenabrechnung und entscheidet (in diesem Fall positiv) über ihre Zulässigkeit.
4. Schließlich erfolgen in der Buchhaltungsabteilung die anfallenden finanziellen Transaktionen.

Für jede dieser Migrationen muß der Programmierer eine spezielle Codierung der im nächsten Schritt relevanten Zustände des Workflows veranlassen. Eine umfassende Codierung des gesamten Ausführungszustandes würde zwar erlauben, den Prozeß des Angestellten von der Aufgabe der makroskopischen Steuerung des Workflows zu befreien, doch dadurch würde der ohnehin schon kritische Anteil *systembedingter* Programmkomplexität noch weiter erhöht.

Die Modellierung migrierender Aktivitäten im Client/Server-Stil ist aus folgenden Gründen besonders häufig inadäquat.

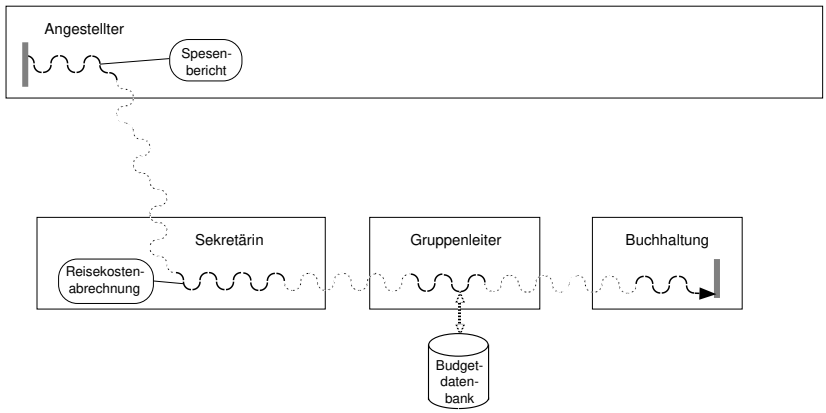

Abbildung 6.3: Modellierung des Reisekosten-Workflows durch einen migrierenden Thread

- Es ist schwierig, den Zustand eines komplexen Ablaufes (z.B. Workflow) in separate Client- und Server-Kontexte aufzusplitten. Im allgemeinen ist es natürlicher, einen einzigen Kontext, der von Host zu Host mitgeführt wird, inkrementell zu akkumulieren.
- Mit der in Kapitel 5 vorgestellten Technik ist es zwar prinzipiell möglich, Client/Server-Bindungen über Tage, Wochen oder noch länger aufrechtzuerhalten, bzw. wiederherzustellen. Doch es gibt keine Garantie für die *permanente* Bereitschaft von Netzwerkverbindungen, Programmen und kommunikativen Objekten. Deshalb wäre ein Programmierstil, der entsprechende Abhängigkeiten vermeidet, vorzuziehen.
- Für viele Aktivitäten ist es nicht erforderlich, die Ausführungskontrolle nach Beendigung einer Aufgabe an die aufrufende Seite zurückzugeben. Stattdessen ist häufig beabsichtigt, daß sie direkt auf einen oder mehrere dritte Knoten übergeht.

Programmiersprachen mit Funktionen höherer Ordnung (z.B. TL, Napier88, Obliq) können Evaluationszustände durch Funktionsabschlüsse *implizit* erfassen [Mathiske et al. 95a]. Dies ist jedoch nur an vorbereiteten Stellen und nur auf Veranlassung der Aktivität selbst hin möglich. Außerdem ist eine eingeschränkte und unintuitive Form der Programmgestaltung erforderlich.

Eine flexible und übersichtliche Gestaltung migrierender Aktivitäten ermöglicht nur die direkte Abbildung auf Threads [Mathiske et al. 95a]. Dies soll anhand obigen Workflow-Beispiels demonstriert werden. In Abbildung 6.3 ist ein Thread dargestellt, der den bewußten Workflow über mehrere Hosts hinweg migrierend ausführt. Das nachstehend aufgelistete TL-Skript dieses Threads ist kurz und bündig.

```
let var spesenBericht = dummy
repeat
  migrate to angestellter do
    let spesenBericht = erfasseBenutzerDaten()
    migrate to sekretaerin do
      abrechnung := verfasseAbrechnung(spesenBericht)
    end
  end
until formalKorrekt(abrechnung)
migrate to gruppenLeiter with remote budgetDB :BudgetDB do
  ueberpruefe(budgetDB  abrechnung.summe)
end
migrate to buchhaltung do
  ueberweise(abrechnung.summe)
end
```

Die hier verwendete Notation **migrate to ... do** ist reiner „syntaktischer Zucker“, der mit Hilfe von Tycoons erweiterbarer Syntax [Cardelli et al. 94] realisiert ist. Tatsächlich werden einige Tycoon-Bibliotheksfunktionen aufgerufen, um bestimmte Thread-Manipulationen und Kommunikationsschritte durchzuführen, die in Abschnitt 6.2.2 erläutert werden.

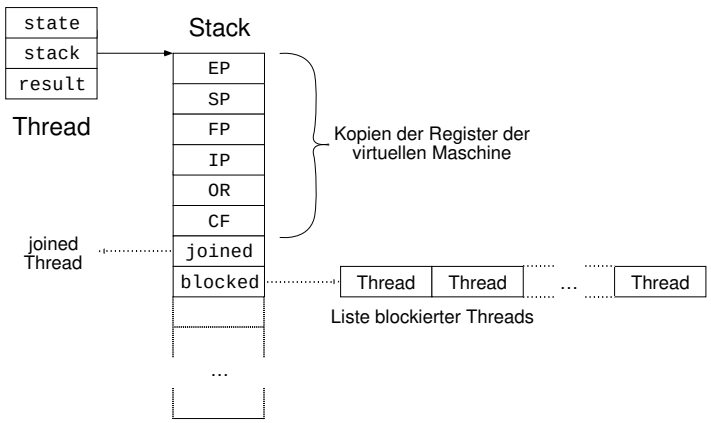

Abbildung 6.4: Die Repräsentationsstruktur eines Threads im persistenten Objektspeicher

6.2 Implementationstechniken für migrierende persistente Threads

Wenn ein Thread migriert, dann soll er anschließend auf der Empfängerseite persistent sein. Außerdem soll die Senderseite eine konsistente Sicht der Migration persistent halten, nämlich den Sachverhalt, daß der bewußte Thread nicht mehr lokal präsent ist. In den nächsten drei Abschnitten werden Implementationstechniken vorgestellt, die eine integrierte Nutzung der Persistenz und Migrationsfähigkeit von Threads erlauben:

1. eine persistente und mobile internen Datenrepräsentation für Threads (Abschnitt 6.2.1),
2. die Definition und Implementation einer typsicheren Schnittstelle für autonome Thread-Migrationen (Abschnitt 6.2.2),
3. die transaktionale Absicherung von Thread-Migrationen (Abschnitt 6.2.3).

Dabei werden auf der Implementationsebene wirksame Synergien zwischen Thread-Persistenz und Thread-Mobilität deutlich.

6.2.1 Die Repräsentation von Threads im Objektspeicher

Tycoon-Threads verhalten sich während ihrer Ausführung aus Sicht des Programmierers und des Anwenders wie normale flüchtige Threads in anderen Programmiersprachen. Jeder Tycoon-Thread wird von einer eigenen (in *C* realisierten) Coroutine[6] mit einem *eigenen CPU-Stack* exekutiert. Dadurch ist jeder Thread in der Lage, unabhängig von anderen Threads

[6]Alternativ können auch vom Betriebssystem unterstützte Threads eingesetzt werden. Allerdings ist bei Verwendung preemptiver Threads an einigen Stellen im Laufzeitsystem eine explizite Synchronisation von Zugriffen auf geteilte Ressourcen notwendig. Diese Anforderung betrifft vor allem Objektspeicherzugriffe, die durch die objektspeicherunabhängige Schnittstelle TSP gekapselt sind. Das TSP abstrahiert nicht nur „nach oben“ von den Eigenschaften des Objektspeichers, sondern auch „nach unten“ von den Eigenschaften des Laufzeitsystems. Deshalb müssen alle Maßnahmen der Thread-Synchronisation ggf. oberhalb des TSP durchgeführt werden. Bei einem Redesign der Tycoon-Architektur wäre in Betracht zu ziehen, die Basisfunktionalitätsschicht *RT* (siehe Abschnitt 4.1) des Tycoon-Laufzeitsystems für die Implementation des TSP bzw. eines Objektspeichers zu nutzen, um das gegebene Multi-Threading tiefer im System zu verankern.

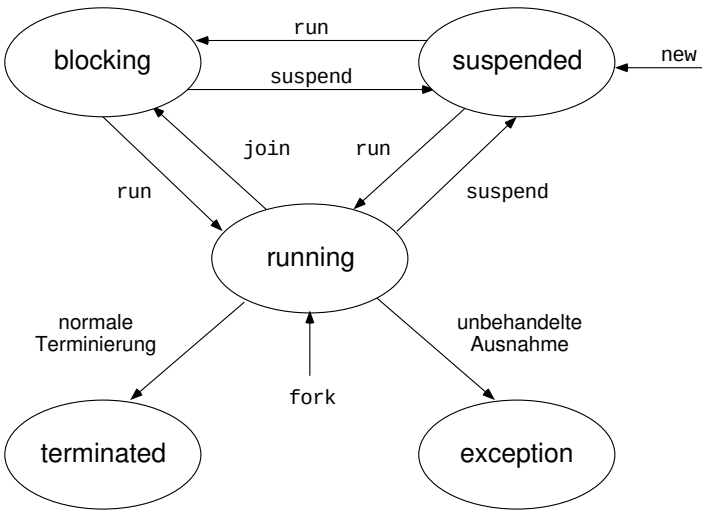

Abbildung 6.5: Ausführungszustände persistenter Threads

externe Aufrufe und Callbacks auszuführen. Kontextwechsel zwischen Coroutinen erfolgen nach Maßgabe eines Zeitgebers (*timer*). Dadurch erhält der TL-Anwender die Illusion von Parallelität.

Um Threads die Eigenschaft der Persistenz zu verleihen, gilt es, sie in das globale Transaktionsschema des persistenten Objektsystems einzugliedern. Gemäß Abschnitt 3.3 und 3.4 werden alle persistenten Tycoon-Werte mit Hilfe der Software-Schnittstelle TSP im Objektspeicher repräsentiert. Es liegt also nahe, für Threads ein analoges Vorgehen zu wählen.

Abbildung 6.4 stellt die konkrete Realisierung im Tycoon-System dar. Das Speicherobjekt links oben ist der eigentliche Thread-Wert. Es enthält als permanenten Eintrag den abstrakten Thread-Zustand. Die Inhalte der Felder *stack* und *result* sind zustandsabhängig. Es folgt eine Beschreibung der möglichen Zustände (*state*), deren Übergänge in Abbildung 6.5 ersichtlich ist.

running: Dies ist der Zustand, in dem die Exekution eines Threads vorangetrieben wird. Da es sich bei der virtuellen Maschine des Tycoon-Systems um eine Stack-Maschine handelt, weist jeder aktive Thread ein persistentes Stack-Objekt auf. Dieses wird durch das Feld *stack* referenziert. Am Anfang des Stack-Objektes sind die Register (*IP* ... *CF*) der abstrakten Maschine sowie Verwaltungsdaten (*joined*, *blocked*) gespeichert. Durch die gemeinsame Speicherung in einem Objekt kann die gemeinsame Allokation und Freigabe von Registern und Stack jeweils in einem Schritt erfolgen.

suspended: In diesem Zustand ist der Thread durch einen Aufruf der Funktion *thread.suspend* explizit angehalten worden. An seiner Speicherstruktur ändert das nichts. Dies gilt auch für folgenden Zustand.

blocking: Ein Thread, der mittels Aufruf von *thread.join* auf das Ergebnis eines anderen Threads wartet, wird so lange blockiert, bis das Ergebnis vorliegt. Anschließend wird entweder im Zustand *running* die Evaluation fortgesetzt oder aber der Thread ist *suspended*. Letzteres ist der Fall, wenn ein anderer Thread inzwischen die Operation *suspend* auf ihn angewendet hat. Das Feld *joined* verweist ggf. auf den Thread, auf welchen gewartet wird. Umgekehrt verweist das Feld *blocked* ggf. auf eine Liste wartender Threads. Natürlich kann beides gleichzeitig auftreten.

terminated: Wenn ein Thread seine Funktion ordnungsgemäß beendet hat, wird das Endergebnis im Feld *result* abgespeichert. Von dort wird es dann an die Threads in der *blocked*-Liste weiterpropagiert. Anschließend werden sämtliche durch das Feld *stack* referenzierten Datenstrukturen abgetrennt. Auf diese Weise belegt ein terminierter Thread kaum Speicherplatz, während ein auf ihn bezogener Aufruf von *join* nach wie vor das berechnete Ergebnis liefert. Außerdem bleibt der Thread ein TL-Wert erster Klasse. Dies gilt auch in folgendem Zustand.

exception: In diesen Zustand gerät jeder Thread, in dem eine dynamische Ausnahme „durchschlägt", d.h. nicht von einem geeigneten *try*-Konstrukt (siehe Abschnitt 3.2) abgefangen und behandelt wird. Das vorliegende Ausnahmepaket wird als Ersatzergebnis im Feld *result* gespeichert, sowie an wartende Threads propagiert. Auch bezüglich des Feldes *stack* wird analog zum Zustand *terminated* vorgegangen, so daß die verbleibende Thread-Struktur auf das Nötigste reduziert wird.

Die eigentliche Evaluation von Tycoon-Threads erfolgt nicht direkt auf den beschriebenen persistenten Datenstrukturen. Dies würde nämlich bedeuten, daß ausnahmslos alle Operationen der virtuellen Maschine zu Objektspeicherzugriffen führten, was um Größenordnungen zu ineffizient wäre. Deshalb gibt es für Threads jeweils auch eine komplette flüchtige Repräsentation, deren Inhalte zu wenigen wohldefinierten Zeitpunkten abgeglichen werden.

- Die Thread-Register werden in lokalen *C*-Variablen gehalten und in Folge dessen durch effiziente *C*-Anweisungen manipuliert.
- Der Stack wird mit Hilfe der in Abschnitt 3.3 beschriebenen TSP-Operation *openReadWriteLock* in den Hauptspeicher abgebildet, bzw. mit *close* zurückgeschrieben.

Für den Abgleich dieser Strukturen relevante Zeitpunkte sind: Systemneustart, Sichern und Zurücksetzen einer Transaktion, Beginn und Ende einer Garbage-Collection sowie Zustandswechsel des Threads.

Alle im Zustand *running* befindlichen Threads werden beim Sichern einer globalen Transaktion in einer Liste erfaßt, die das Wurzelobjekt des gesamten persistenten Objektspeichers bildet. Diese Maßnahme schafft auf einfache Weise klare Verhältnisse für die Speicherverwaltung:

- Alle Objekte (insbesondere auch passive Threads), die transitiv von irgendeinem aktiven Thread erreichbar sind, werden von der Garbage Collection als weiterhin benötigt erkannt und sind persistent.
- Alle anderen Objekte fallen der Garbage Collection zum Opfer, denn es gibt keine Aktivität, die in Zukunft direkt oder indirekt auf sie zugreifen könnte.

- ▷ Wenn kein Thread mehr aktiv ist, ist der Speicher faktisch leer.

Das einzige andere persistente Objektsystem, welches persistente Threads bietet, ist soweit bekannt Napier88. In diesem System werden ausnahmslos alle, auch passive Threads in einer Unterstruktur des Wurzelobjektes angesiedelten globalen Liste erfaßt [Munro 93]. Dies führt zur persistenten Erhaltung nicht mehr aktiver Strukturen, was nur in Ausnahmefällen (z.B. beim post-mortem-Debugging) erforderlich ist. In TL kann dieses Verhalten simuliert werden, indem ein Thread eine entsprechende Liste (oder Datenbank) verwaltet.

Aufgrund zahlreicher referentieller Abhängigkeiten können persistente Threads in Napier88 nicht migrieren, d.h. von einem persistenten Objektspeicher in einen anderen übertragen werden. In Tycoon hingegen wurde sorgfältig darauf geachtet, alle Referenzen zwischen Threads und anderen Objekten auf das absolut Notwendige zu reduzieren. Dies ist eigentlich auch schon allein unter dem Gesichtspunkt der Persistenz unbedingt erforderlich, denn das unabsichtliche „Aufbewahren“ obsoleter Daten führt zu faktisch irreversibler Speicherplatzverschwendung im erheblichen Ausmaß. Zum einen können beliebig große Datenstrukturen betroffen sein. Vor allem jedoch summieren sich bei *langlebigen* Anwendungen die Speicherplatzverschwendungen einer Vielzahl von Programaktivierungen.

Um eine bruchlose Programmlogik zu gewährleisten, sind alle im Tycoon-System verfügbaren Primitive[7] zur Thread-Synchronisation, *Semaphore*, *Mutexes* und *Condition Variables*, ebenfalls persistent. Zudem orientiert sich auch ihre Implementation [Piellusch 96] streng an den besonderen Anforderungen eines persistenten Objektsystems:

- ▷ Alle relevanten Zustände sind durch persistente Speicherobjekte repräsentiert.
- ▷ Es wird so sparsam wie möglich mit Objektreferenzen umgegangen.

In Zusammenfassung dieses Abschnitts ergeben sich zwei wichtige Synergien zwischen Persistenz und Mobilität:

- ▷ Die persistente Repräsentation von Tycoon-Threads durch untypisierte TSP-Objekte gleicht der aller anderen TL-Werte. Sie werden daher vollkommen unterschiedslos linearisiert, in Netzwerken übertragen und als RPC-Argumente akzeptiert. Threads sind also zwischen persistenten Objektspeichern *mobil*.
- ▷ Die notwendige Sparsamkeit der internen Repräsentation persistenter Threads kommt auch deren Mobilität zugute. Jedes unnötig referenzierte Objekt würde bei einer Thread-Migration zur Erhöhung des zu übertragenden Datenvolumens und damit zu einer längeren Übertragungszeit sowie zu einer zusätzlichen Speicherplatzbelastung am Zielort führen.

[7] Gleiches gilt damit auch für alle abgeleiteten, in TL konstruierten Mechanismen wie z.B. *Monitore* und *Rendezvous*.

6.2.2 Typisierte Thread-Migrationspforten

Während „passive“ Migrationen einfach im Form direkter oder transitiver Verwendung von Threads als RPC-Argumente geschehen, migrieren die Threads in obigen Szenarien in Entsprechung des Agenten-Paradigma [Wayner 94; Reinhardt 94; White 94] „aktiv“, d.h. sie initiieren ihre Migrationen jeweils selbst. Hierfür sind aufgrund folgender Problemstellungen einige nicht-triviale Implementationsschritte erforderlich. Denn die eigene Übertragung kann der migrierende Thread nicht komplett selbst steuern, da er nicht an mehreren Orten zugleich sein kann. Außerdem erfordert die typsichere dynamische Bindung an ein lokales Objekt auf der Emfängerseite spezielle Maßnahmen.

Das Modul *gate*, dessen operativer Quelltext (bis auf Fehlerbehandlungen) im folgenden *vollständig* wiedergegeben ist, implementiert die im vorigen Abschnitt verwendete Form der aktiven Thread-Migration. Die Kürze dieser Implementation und die Tatsache, daß hier die Systemprogrammierung vollständig im typsicheren sprachlichen Rahmen von TL erfolgt, demonstrieren die Ausdrucksmächtigkeit der in Kapitel 5 erarbeiteten RPC-Infrastruktur.

Zuerst wird im Modul *gate* ein Typoperator definiert, der mit dem Typ der dynamisch zu bindenden Objekte auf der Emfängerseite parametrisiert ist.

```
Let Parameter(Data <:Ok) = Tuple
  var data :optional.T(Data)
  agent :thread.T(Ok)
end
```

In einem solchen Paket ist jeweils ein migrierender Thread zusammen mit einem Platzhalter für die ihn erwartenden Daten angeordnet.

Der analog parametrisierte Haupttyp des Moduls repräsentiert entfernte „Pforten“ (*gates*), zu denen Threads migrieren können, um in fremde Objektspeicher „einzutreten“:

```
Let T(Data <:Ok) <:Service = Tuple
  transfer(parameter :Parameter(Data)) :Ok
end
```

Eine solche Pforte ist nichts anderes als ein RPC-Dienst, der nur eine einzige Funktion aufweist. Diese Funktion (*transfer*) wird von der Funktion *gate.migrateTo* aufgerufen, um Pakete des Typs *Parameter(Data)* zu übermitteln. Die Verwendung von *gate.migrateTo* stellt sich zusammengefaßt stets wie folgt dar:

```
let site = client.bind(...)
...
let myData = gate.migrateTo(site)
...
```

Der aktuelle Thread erhält durch eine Client-Bindung (*client.bind*) Zugang zu einer Migrationspforte (hier *site*), die er mittels *gate.migrateTo* „durchschreitet“. Am Zielort liefert die Funktion *gate.migrateTo* einen durch den dort ansässigen RPC-Dienst bestimmten Wert, den der Thread nun in seinem eigenen Sichtbarkeitsbereich manipulieren kann.

Die Implementation der Funktion *gate.migrateTo* ist kurz:

```
let migrateTo(Data <:Ok  gate :T(Data)) :Data =
  begin
    let parameter = tuple
      let var data = optional.nil(:Data)
      let agent = thread.self()
    end
    thread.launch(fun(self :thread.T(Ok))
      begin
        gate.transfer(parameter)
        thread.kill(parameter.agent)
      end)
    optional.value(parameter.data)
  end
```

Zuerst wird aus dem aktuellen Thread und einem Optionalwert vom Typ *Data* ein zu transferierendes Paket (*parameter*) aggregiert. Der Optionalwert *data* kapselt zwar den Typ *Data*, enthält jedoch anfangs noch keinen manifesten Wert dieses Typs. Die Extraktion eines solchen Wertes vollzieht der Aufruf der Funktion *optional.value* ganz am Schluß. Dazwischen liegen die eigentliche Migration des Threads und die „Füllung“ des Platzhalters *data*.

Die Durchführung der Übermittlung des Paketes *parameter* übernimmt ein nur kurzzeitig vorhandener, lokaler Hilfs-Thread, welcher mit *thread.launch* erzeugt und gestartet wird. Letztere Funktion wirkt diesbezüglich genau wie *thread.fork* (siehe Abschnitt 6.1.1). Zusätzlich suspendiert sie jedoch *zuvor* den aufrufenden Thread, d.h. sie legt den migrierenden Thread still.

Der lokale Hilfs-Thread führt die mit dem Schlüsselwort *fun* beginnende anonyme Funktion aus. Als erstes ruft er den RPC *gate.transfer* auf, wodurch das Paket versendet wird. Dann löscht er die lokale Repräsentation des nun bereits migrierten Threads. Danach ist der Hilfs-Thread beendet und da keinerlei Referenzen auf ihn bestehen, wird er letztendlich der Garbage-Collection anheim fallen.

Die Zeile *optional.value(parameter.data)*, welche das Funktionsergebnis von *migrateTo* liefert, wird von der Kopie des migrierten Threads bereits auf der Empfängerseite ausgeführt. Wie dies zustande kommt, ergibt sich aus der nachfolgend beschriebenen Implementation der Migrationspforte als RPC-Dienst.

Auf der Server-Seite ist die Funktion *new* des Moduls *gate* zu verwenden. Sie erzeugt eine neue Migrationspforte und registriert sie bei einem RPC-Dispatcher. Aufgrund parametrisch polymorpher Programmierung besteht auch ihre Implementation in wenigen Zeilen. Sie ist nachstehend vollständig aufgeführt.

```
let new(dispatcher :server.T Dyn Attributes <:Ok attributes() :Attributes
        sf :CSTypes.SearchFun Data <:Ok data() :Data) :sid.T(T(Data)) =
 begin
  let gate = tuple
   let transfer(parameter :Parameter(Data)) =
    begin
     parameter.data := optional.new(data())
     thread.run(parameter.agent)
    end
  end
  server.register(dispatcher :T(Data) :Attributes attributes sf gate)
 end
```

Die ersten vier Parameter orientieren sich an denen der in Abschnitt 5.1.2 beschriebenen Funktion *server.register*. Der dynamische Typ des RPC-Dienstes ist hier nicht anzugeben, da er sich aus den weiteren Gegebenheiten ergibt. Stattdessen sind ein Typ *Data* und eine Funktion, die einen Wert diesen Typs liefert, anzugeben. Das Ergebnis von *new* ist ein Dienstidentifikator, der mit einem manifesten Pfortentyp parametrisiert ist. Dieser Pfortentyp ist wiederum durch *Data* festgelegt.

Die erste Anweisung konstruiert ein Tupel (*gate*) als lokalen RPC-Dienst, der dann in einem zweiten Schritt durch *server.register* im Netz verfügbar gemacht wird. Die vom Dienst *gate* als einzige angebotene Funktion *transfer* nimmt von entfernten Clients einen Parameter entgegen, der wie weiter oben bereits angegeben aufgebaut ist.

Man beachte, daß die Funktion *data* einer *dynamischen Bindung* (nämlich einer Parameterübergabe) entstammt. Dadurch gelingt es, sie dynamisch in den hier vorliegenden Sichtbarkeitsbereich einzuführen. Ferner ist bedeutsam, daß TL Funktionen höherer Ordnung bietet. Die parameterlose Funktion *data* kann in ihrem Funktionsabschluß potentiell beliebige Objekte aus dem gesamten Objektspeicher kapseln.

Der durch den Aufruf von *data* erzeugte Wert dringt in den von *transfer* entgegengenommen Thread ein, indem er der im Sichtbarkeitsbereich des Threads liegende Variable *parameter.data* zugewiesen wird. Dies ist wiederum eine Form der dynamischen Bindung. Genau an dieser singulären Stelle verschmilzt quasi der lokale Sichtbarkeitsbereich mit dem des ankommenden Threads. Es ist hervorzuheben, daß dies in vollkommener statischer Typsicherheit geschieht.

In diesem Abschnitt wurde besonders deutlich, wie wertvoll ein Typsystem höherer Ordnung ist. Trotz völliger Freiheit bei der Wahl der Typen dynamisch zu bindender entfernter Objekte ist die *statische* Typsicherheit von Skripten migrierender Threads sichergestellt. Lediglich Client-Bindungen an entfernte Migrationspforten erfordern jeweils eine *dynamische* Typüberprüfung, die scheitern kann. Aber dies findet in jedem Falle *vor* der betreffenden Migration statt. Damit ist ausgeschlossen, daß erst in Folge einer Thread-Migration ein Typfehler auftritt.

Häufig ist bereits am Ursprungsort die Durchführung sämtlicher Pfortenbindungen für die „Reise“ eines Threads möglich. Daraufhin ist dann der gesamte weitere Verlauf des Thread-Skriptes statisch typsicher.

6.2.3 Transaktionale Thread-Migration

Die Einsatzdomäne der Migration persistenter Threads sind lose gekoppelten Umgebungen mit hochgradig autonomen Systemen. In solchen Umgebungen konzentriert sich die Wahrung der Konsistenz von Datenbeständen vorwiegend auf lokale Systeme. Wenn sich Integritätsbedingungen nicht über mehrere Systeme erstrecken, besteht kein Bedarf für systemübergreifende Kontrollsphären (*Spheres of Control* [Gray, Reuter 93]), um parallele Aktivitäten vor inkorrekten Wechselwirkungen zu schützen.

Durch Threads realisierte migrierende Aktivitäten sollten diese relativ einfachen Gegebenheiten nach Möglichkeit nicht verkomplizieren. Deshalb werden in der vorliegenden Arbeit nur solche Bindungstechniken (siehe nächstes Kapitel) bereitgestellt, die sowohl die Autonomie migrierender Objekte als auch die Autonomie der von ihnen besuchten Systeme unterstützen. Systemübergreifende Abhängigkeiten werden damit so weit wie möglich vermieden. Allerdings sind migrierenden Aktivitäten systemübergreifende transaktionale Problemstellungen *inhärent*.

Im Falle einer Migration sollte unter den betroffenen Objektsystemen Übereinstimmung über die Tatsache der Migration bestehen, da die Aktivität sonst ungewollt im Nichts verschwinden oder in mehreren Instanzen auftreten kann. Die Migration muß folglich als *verteilte Transaktion* [Gray, Reuter 93] durchgeführt werden, wofür sich die Technik des zweiphasigen Commit (2PC) anbietet [Gray 78; Lindsay et al. 79; Lampson 81; ISO-CCR 90].

Beim 2PC wird dem Commit eine von einem zentralen Koordinator initiierte Wahlphase (*prepare phase*) vorangestellt, in der jedes an der verteilten Transaktion teilnehmende persistente System darüber Auskunft gibt, ob die Transaktion von ihm erfolgreich beendet werden kann. Die Antworten werden an den Koordinator geschickt, der bei durchgehend positiven Antworten in der zweiten Phase allen Teilnehmern die Anweisung zur erfolgreichen Beendigung der Transaktion gibt. Bei einer oder mehreren negativen Antworten in der ersten Phase werden alle Teilnehmer zum Abbruch der Transaktion angewiesen. Durch dieses Protokoll läßt sich garantieren, daß alle Teilnehmer einen gemeinsamen Ausgang der Transaktion erreichen. Einzelheiten des Verfahrens sind in [Bernstein et al. 87] eingehend beschrieben. Eine leicht vereinfachende, an eine Darstellung in [Özsu, Valduriez 91] angelehnte Übersicht [Kornacker 95] des 2PC ist in Abbildung 6.6 ersichtlich.

Aufgrund einer Restriktion des TSP kann das 2PC leider *nicht* in TL implementiert werden. Entsprechend der Spezifikation des TSP gelangt ein Objektspeicher in Folge der TSP-Operation *tsp_prepareCommit* in den Zustand *committable*, in dem dann ausschließlich die Operationen *tsp_commit* und *tsp_abort* zulässig sind [Matthes et al. 95a]. Dies ist wiederum auf eine Restriktion des 2PC zurückzuführen: in der Phase zwischen *VOTE-COMMIT* und dem Transaktionsende darf ein 2PC-Teilnehmer nur Operationen ausführen, die die Zusicherung, das *commit* durchführen zu können, nicht untergraben. Damit verschiedenste Objektspeichertypen verwendet werden können, schließt das TSP im Zustand *committable* alle Operationen außer den beiden für das 2PC essentiellen aus.

Eine Ausführung von TL-Anweisungen zwischen der Bereiterklärung zum *commit* und der Transaktionsbeendigung hätte unzulässige TSP-Operationen zur Folge. Daher müssen zumindest diejenigen Kommunikationsvorgänge, die in dieser Phase vorgesehen sind (*VOTE-ABORT, VOTE-COMMIT, GLOBAL-ABORT, GLOBAL-COMMIT*), ohne TL realisiert werden. Hierfür bietet sich die plattformunabhängige Socket-Schnittstelle aus Abschnitt 4.2 an.

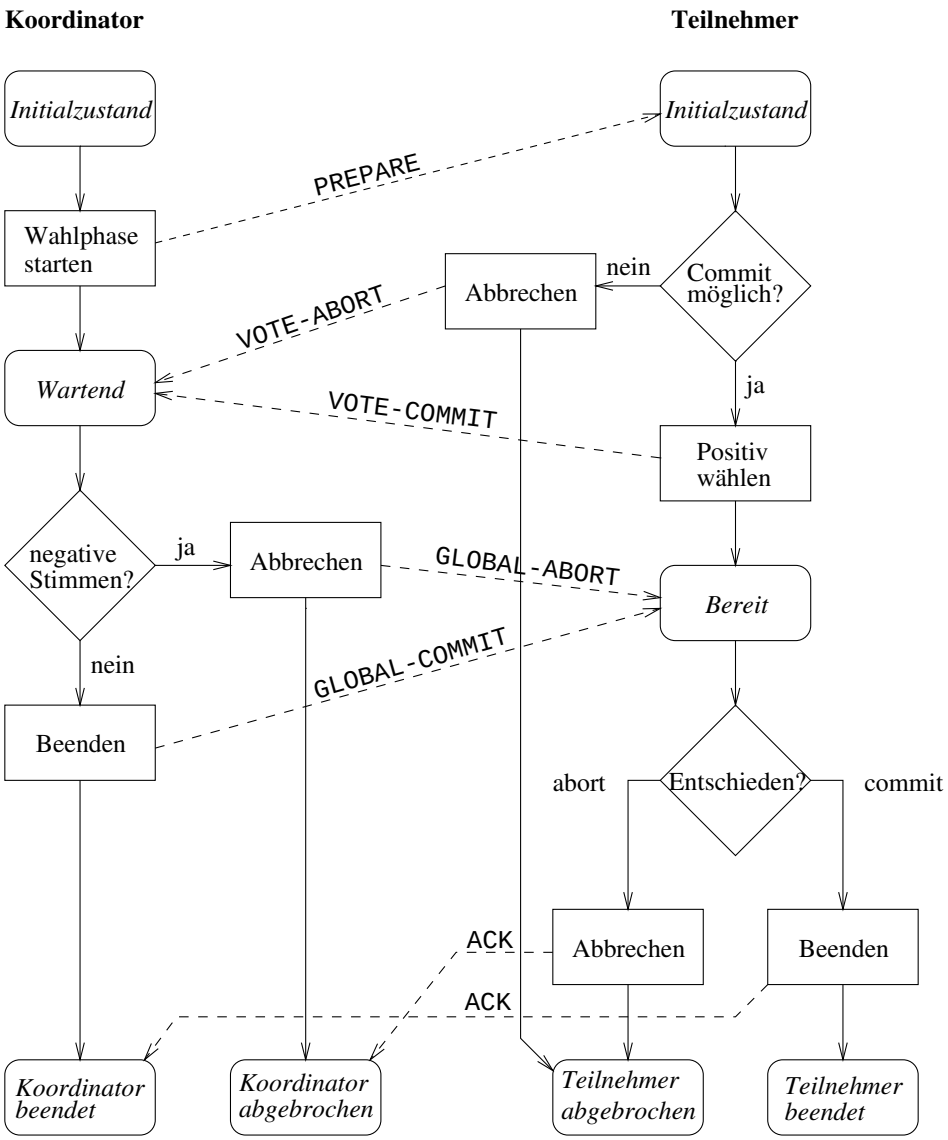

Abbildung 6.6: Schematischer Ablauf des 2-Phasen-Commit-Protokolls

Das TL-Modul *twoPhaseCommit* exportiert maßgeschneiderte Funktionen für den in TL initiierten, aber in *C* durchgeführten Einsatz des 2PC. Folgende Funktion dient der Vorbereitung eines 2PC:

newParticipation(action() :Ok) :Port.T

Diese Funktion erzeugt einen zusätzlichen Thread (2PC-Thread) und eine frische Netzwerkadresse. Der 2PC-Thread hat die Aufgabe, dem gesamten Objektsystem die Rolle als Teilnehmer in einem 2PC aufzuzwingen. Er ruft *socket.accept* auf und ruht, bis der in der Netzwerkadresse angegebene Port von einem entfernten Prozeß per Socket angesprochen wird. Wenn dies eintritt, hält er mit Hilfe der Funktion *thread.atomic* die Ausführung aller anderen Threads an und verfährt weiter wie folgt:

1. Es wird ein *C*-Call aufgerufen, der das 2PC als Teilnehmer abhandelt. Dabei wird über ein Socket mit dem Koordinator des 2PC kommuniziert. Durch die Verwendung eines *C*-Calls und das vorherige Anhalten aller anderen Threads sind unzulässige TSP-Operationen absolut ausgeschlossen.
2. Das Operations-Log zur Integration flüchtiger Objekte wird abgearbeitet (siehe dazu Abschnitt 7.6 und Anhang E.2, sowie die Funktion *log* in Anhang D.1).
3. Im Falle eines erfolgreichen Ausganges der Transaktion führt der 2PC-Thread *action* aus, sonst nicht.
4. Der 2PC-Thread terminiert. Daraufhin laufen alle anderen Threads wieder weiter.

Abbildung 6.7 stellt obiges Szenario aus der Sicht eines 2PC-Teilnehmers dar. In TL stellt sich das Skript des 2PC-Threads wie folgt dar:

```
(* Bisher unbenutzten Port belegen, IP-Adresse erzeugen: *)
let address = newAddress()

(* Am Port auf Anfrage warten: *)
let sock = createSocket(address.portNumber)
socket.listen(sock ...)
let sock2 = socket.accept(sock ...)

thread.atomic(fun() (* Alle anderen Threads vorübergehend blockieren *)
  begin
    (* In C das 2PC als Teilnehmer abhandeln: *)
    if twoPhase.participateIn2PC(sock2) then action() end
  end)
socket.close(sock2)
socket.close(sock)
```

Unter Ausnutzung des Konzeptes persistenter Threads wird der Aufruf von *action* auch im Falle eines späteren Rücksetzens (*tsp_abort*) oder nach einem Neustart des Tycoon-Prozesses ausgeführt. Der *C*-Call-Mechanismus ist so gestaltet, daß ein *commit* innerhalb eines *C*-Calls

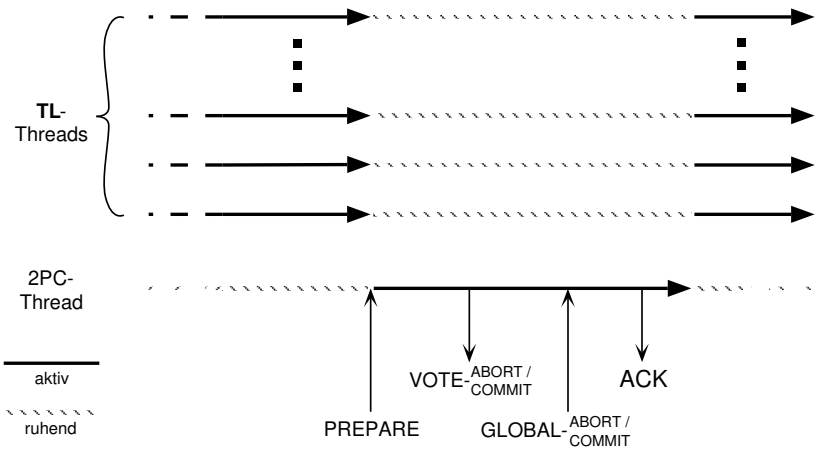

Abbildung 6.7: Durchführung der 2PC-Teilnahme durch einen speziellen Thread

ein eventuelles Wiederaufsetzen des verursachenden Threads *genau nach* dem bewußten *C*-Call vorsieht. Obiger 2PC-Thread „landet“ deshalb ggf. an der Stelle direkt nach *participateIn2PC*. Da das Ergebnis dieses Aufrufes persistent ist, führt die **if**-Anweisung dann stets in den **then**-Zweig zu *action*.

Die Weitergabe (z.B. per RPC) einer durch *twoPhaseCommit.newParticipation* erzeugten Netzwerkadresse an ein entferntes System versetzt jenes in die Lage, als Koordinator einen gemeinsamen 2PC-Vorgang zu inszenieren. Dazu dient folgende Funktion[8], die als Argument ein Array mit den Netzwerkadressen der vorgesehenen 2PC-Teilnehmer erwartet:

*coordinateAndParticipate(otherParticipants :***Array***(Port.T)) :Bool*

Eine wichtige Anwendung des Moduls *twoPhaseCommit* ist die transaktionale Sicherung der in Abschnitt 6.2.2 vorgestellten Thread-Migrationspforten. Die Rolle des Koordinators übernimmt jeweils ein Client einer Pforte, d.h. ein Prozeß, den zu verlassen ein Thread im Begriff ist. Um das Objektsystem der Pforte als 2PC-Teilnehmer fungieren zu lassen, erzeugt die Übertragungsfunktion *transfer* der Pforte mittels *twoPhaseCommit.newParticipants* einen 2PC-Thread. Außerdem liefert sie eine konkrete Netzwerkadresse vom Typ *Port.T* an den Koordinator:

```
...
let transfer(parameter :Parameter(Data)) :Port.T =
  begin
    parameter.data := optional.new(data())
    twoPhaseCommit.newParticipation(
      let action() = thread.run(parameter.agent))
  end
...
```

[8] Zusätzlich gibt es eine Funktion namens *twoPhaseCommit.coordinate*, die dem Koordinator erlaubt, nicht gleichzeitig auch Teilnehmer zu sein. Diese wird jedoch im vorliegenden Zusammenhang nicht benötigt.

Dadurch, daß der migrierende Thread erst nach erfolgreichem 2PC wieder aktiviert wird, erspart man sich ein eventuell notwendiges Rücksetzen von Operationen, die er sonst nach seiner Übertragung und vor dem 2PC hätte ausführen können.

In der Migrationsfunktion aus Abschnitt 6.2.2 ist das Skript des Hilfs-Threads, der die Migration tätigt, wie folgt ergänzt:

```
let migrateTo(Data <:Ok  gate :T(Data)) :Data =
  begin
    let parameter = tuple
      let var data = optional.nil(:Data)
      let agent = thread.self()
    end
    thread.launch(fun(self :thread.T(Ok))    (* Hilfs-Thread-Skript: *)
      begin
        (* Thread versenden und Adresse des 2PC-Teilnehmers einholen: *)
        let participant = gate.transfer(parameter)

        (* 2PC ausführen: *)
        if twoPhaseCommit.coordinateAndParticipate(array participant end) then
          (* Transaktion erfolgreich -
             lokale Kopie des migrierten Threads deaktivieren: *)
          thread.kill(parameter.agent)
        else
          (* Transaktion gescheitert -
             Ausnahme in dem Thread auslösen, der migrieren wollte: *)
          thread.throw(parameter.agent  fun() raise error with ... end)
        end
      end)
    optional.value(parameter.data)
  end
```

Von einer Migration „betroffen“ sind neben neben dem Ursprungs- und dem Zielsystem alle weiteren Systeme, für welche die Tatsache der Migration eine anwendungsrelevante Information darstellt. Dies ist zum Beispiel bei sogenannten „angeleinten“ mobilen Agenten (*tethered agents*), die auf ihrer Reise von einem bestimmten Standort aus überwacht und ggf. ferngesteuert werden, der Fall. Gleiches gilt für Workflows, deren Standort und Status von einer mit ihrer Durchführung eigentlich nicht befaßten Instanz (Manager) abgefragt und manipuliert werden kann. Es bereitet keine großen Schwierigkeiten, weitere Teilnehmer in das 2PC-Szenario von Migrationspforten einzubeziehen, denn die entscheidende Operation, *twoPhaseCommit.coordinateAndParticipate*, ist dafür von vornherein ausgelegt.

Im Zusammenhang der Fernkontrolle ist zu bedenken, daß eine asynchron gestellte Anfrage an eine entfernte Aktivität voraussetzt, daß diese permanent einen RPC-Dienst bereitstellt. Einfache migrierende Threads sind dazu nicht in der Lage, da sie nicht gleichzeitig als Server und als Skriptausführungsintanz fungieren können.

Asynchrone Anfragen ermöglicht zum Beispiel die Modellierung mobiler Agenten durch Pakete aus *mehreren* Threads. In diesem Fall ist jedoch ebenfalls keine Teilnahme dritter an einem 2PC erforderlich, weil die dynamische Adreßauflösung der Tycoon-RPCs die RPC-Dienste eines mobilen Agenten sowohl am Ursprungs- als auch am Zielort einer Migration ansprechen kann.

Das Anhalten aller operativen Threads durch einen 2PC-Thread stellt ein ernsthaftes Problem für das Antwortverhalten eines persistenten Objektsystems im Netzwerk dar. Eine Unterbrechung der Kommunikationsbereitschaft wird von weniger „beharrlichen" entfernten Kommunikationspartnern als Fehlersituation gedeutet. Sie ist von einem Prozeßabbruch nicht unmittelbar zu unterscheiden. Im der derzeitigen prototypischen Implementation des Tycoon-Systems ist noch keine Lösung für dieses Problem implementiert.

Eine Lockerung der Restriktion des TSP, nach einem *tsp_prepareCommit* keine objektspezifischen TSP-Operationen zuzulassen, ist nicht zu erwarten. Zum einen würde dadurch die Auswahl verfügbarer Objektspeicherimplementationen geschmälert und zum anderen könnte eine bestehende *Commit*-Bereitschaft durch uneingeschränkte Aktivitäten konkurrenter Threads zunichte gemacht werden.

Einen Ausweg bietet die Aufrechterhaltung der Kommunikativität während des 2PC durch einen in *C* programmierten, über Sockets kommunizierenden Thread, der dem 2PC-Thread an die Seite gestellt wird.[9] Der *C*-Thread könnte alle eingehenden Nachrichten entgegennehmen und zwischenspeichern, und nach dem 2PC an die betroffenen Threads weiterleiten. Die Implementation dieser Maßnahme wäre jedoch extrem aufwendig.

Eine wesentlich einfachere Lösung besteht darin, Kommunikationsversuche entfernter Systeme zunächst scheitern zu lassen und in Form des *C*-Threads an einem bestimmten Port einen Informationsdienst bereitzustellen, der fortlaufend Angaben über den Systemzustand macht. Dieser Informationsdienst würde ggf. die Schar der kommunikationswilligen Systeme auf das baldige Ende eines in Arbeit befindlichen 2PC „vertrösten". Aufgrund der eindeutigen Unterscheidung von einem Systemabsturz würden diese ihren ursprünglichen Versuch dann nach kurzer Zeit wiederholen. Abbildung 6.8 stellt dieses Szenario, das in alle vier Kommunikationsmechanismen (*C*-Sockets, *port*, *portClient*, *client*) des Tycoon-Systems zu integrieren ist, exemplarisch dar.

Es ist nicht immer unbedingt erforderlich, jede Migration einzeln transaktional durchzuführen. Zudem führt dies bei großen Anzahlen von Migrationen zu Performanzproblemen. In diesem Fall können durch den individuellen Einsatz des Moduls *twoPhaseCommit* größere Gruppen von Migrationen, die durch nicht-transaktionale Pforten erfolgen, mit transaktionalem Schutz versehen werden.[10]

[9] In diesem Fall ist *preemptives* Multi-Threading erforderlich.

[10] Vermutlich wäre ein Pforten-Parameter, der angibt, ob eine aktuelle Migration transaktional verlaufen soll oder nicht, eine recht praktische Erweiterung. Allerdings liegen noch keine Erfahrungen mit *massenhaften* Thread-Migrationen vor.

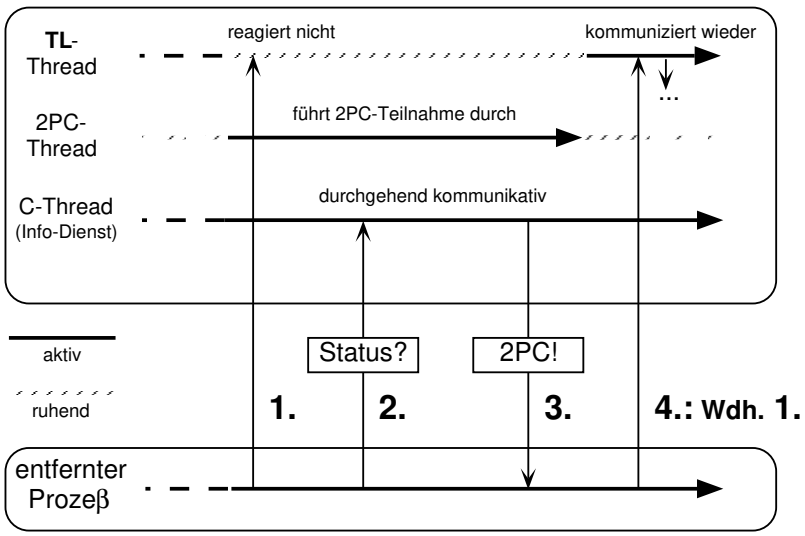

Abbildung 6.8: Überbrückung der durch ein 2PC bedingten Kommunikationspause

Kapitel 7

Skalierende Bindungstechniken für immobile Objekte

In diesem Kapitel wird das für die Mobilität von Objekten ausschlaggebende Problemfeld diskutiert: wie komplexe Objektgraphen, die transitive Bindungen an Daten aller Art inklusive Funktionsabschlüsse und Threads repräsentieren, *effizient* zwischen Adreßräumen übertragen werden können.

Mit den Beiträgen der vorangehenden Kapiteln ist die *Typunabhängigkeit* der Mobilität in persistenten Objektsystemen erreicht. Dies bedeutet jedoch nicht, daß jedes Objekt mobil wäre. Demzufolge kommt es nicht selten vor, daß eine naive Anwendung der in den vorangehenden Kapiteln vorgestellten Kommunikationsmechanismen versagt. In Abschnitt 7.1 wird erläutert, worauf die Immobilität von Objekten typischerweise zurückzuführen ist.

Die übrigen Abschnitte sind spezifischen Techniken gewidmet, mittels derer migrierende Objekte Bindungen an immobile Objekte eingehen können, ohne an Mobilität zu verlieren. Ein besonderes Augenmerk ist darauf gerichtet, inwiefern diese Techniken bezüglich Datenvolumen, Objekttyp und Einschränkungen von Netzwerkverbindungen skalieren.

In Abschnitt 7.3 wird vertiefend auf in den vorigen Kapiteln bereits angewandte Methoden der dynamischen Bindung an immobile Objekte eingegangen.

Abschnitt 7.4 beschreibt den Tycoon-Mechanismus des dynamischen Linkens ubiquitärer Objekte, der dann in Abschnitt 7.5 um die Möglichkeit automatischer Replikation erweitert wird. In Abschnitt 7.5 wird der Vorgang des dynamischen Linkens um die Möglichkeit der automatischen Replikation erweitert. Letztere Technik betrifft allerdings nicht direkt den Umgang mit primär immobilen Objekten, sondern sie setzt im Gegenteil Mobilität voraus. Ihr Zweck ist, erwünschte Immobilität zu erzeugen.

Zu guter Letzt wird in Abschnitt 7.6 ein Verfahren vorgestellt, mit dessen Hilfe auch zunächst flüchtige Objekten Mobilität und Persistenz erlangen.

```
import É
  cCallback iter thread volatile É
  brush button pen color window menu outputDevice É

let activity(self :thread.T(Ok)) =
  begin
    let var width = 100
    let var x = 0

    let paint(É) = É color É brush É
    let w = window.new(É)
    let m = menu.newPopUp(É)
    installCallbacks(É w É m É paint É)

    É
    outputDevice.drawLine(x É)
    É
  end

let t = thread.new(activity)
```

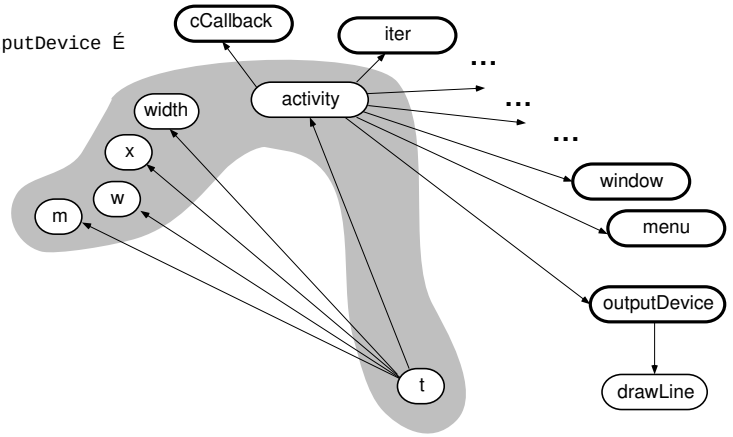

Abbildung 7.1: Die umfangreiche transitive referentielle Hülle eines Threads, der graphische Ausgaben erzeugt

7.1 Ursachen von Immobilität

Alle mit den originären Mitteln der Sprache (d.h. ohne Verwendung externer Schnittstellen) definierten Werte der Programmiersprache TL werden im persistenten Objektspeicher repräsentiert und unterliegen der direkten Kontrolle des Tycoon-Laufzeitsystems. Damit sind sie ohne weiteres Zutun persistent und mobil (vgl. Kapitel 5).

Aufgrund der Notwendigkeit der Einbindung externer Diensterbringer treten in TL auch Bindungen an *externe Objekte* auf. Dies sind durch externen (*C*- oder *C*++-) Bibliothekscode erzeugte flüchtige und immobile Laufzeitwerte, die nicht im Objektspeicher residieren und auch nicht der direkten Kontrolle des Tycoon-Laufzeitsystems unterliegen: Dateien, Kommunikationskanäle, Ein- und Ausgabegeräte, graphische Elemente (Fenster, Knöpfe, Menüs, etc.) oder Programme (E-Mail, Textverarbeitungen, etc.). Im Bedarfsfall muß die Persistenz und Mobilität externer Objekte durch explizite Anweisungen realisiert werden. Zur Vereinfachung dieser Aufgabe existiert ein kanonisches Verfahren, das in Abschnitt 7.6 vorgestellt wird.

Manche Objekte entziehen sich jedoch aufgrund harter externer Restriktionen jeglicher softwaretechnischen Bemühung um ihre Mobilität. Von einer eingeschränkten Verfügbarkeit (z.B. Plattformabhängigkeit, Lizensierung, ortsgebundene Installation) externer Software-Komponenten oder engen Bindungen an spezielle Hardware-Komponenten (z.B. Drucker, Scanner, Kamera) betroffene Objekte sind immobil.

Das Problem der Immobilität wird in der Ausgangslage zunächst dadurch verschärft, daß die grundlegende Semantik der Objektübertragung (siehe Abschnitt 4.3, [Munro 93; Matthes et al. 95a]) zwischen persistenten Objektsystemen in rekursivem Kopieren (*Deep-Copying*) der transitiven referentiellen Hülle (Objektgraph) eines Objektes besteht. Deshalb wirkt Immobilität transitiv: Objekte, die Bindungen an immobile Objekte aufweisen, sind ebenfalls immobil.

Zur Gewährleistung von Mobilität ist die Vermeidung von Referenzen auf immobile Objekte jedoch nicht hinreichend, denn nicht selten sind komplexe Objekte aufgrund folgender Problematik faktisch immobil: die schiere Größe ihrer transitiven referentiellen Hülle (Objektgraph) verhindert ihre vollständige Übertragung im Netzwerk. Im Extremfall ist dies allein schon aus Gründen der Fehlerwahrscheinlichkeit der Fall. Gewöhnlich bestehen jedoch aus Anwendungsgründen bereits Beschränkungen, die wesentlich geringere Volumina oder maximale Längen von Übertragungszeiten nahelegen. Außerdem können Speicherplatzengpässe auf der Empfängerseite eine vollständige Übertragung verhindern. Dies gilt um so mehr, wenn wiederholt komplexe Objekte zum selben Empfänger übertragen (und dort persistent gespeichert) werden.

Große Objektgraphen sind nicht nur für Massendatenstrukturen wie Listen, Mengen, Bäume, Datenbanktabellen, usw., sondern insbesondere auch für Objekte, die Code enthalten, typisch. In diesem Zusammenhang hat die Programmiertugend der Modularisierung fatale Folgen in Bezug auf Mobilität: der Aufruf einer Modulfunktion „bindet“ jeweils das gesamte betreffende Modul mit in den Objektgraphen ein - und ein Modul beinhaltet gewöhnlich Funktionen, welche wiederum andere Module ansprechen. Da jeder Thread eine Funktion als Skript enthält, sind Threads von dieser Problematik ebenfalls betroffen.

Daß durch die Verwendung von Standardmodulen ohne weiteres einige MB zusammenkommen, ist in Abbildung 7.1 anhand einer alltäglichen Situation ersichtlich: ein Thread führt eine Funktion aus, die sich einiger Standardmodule (*cCallback*, *iter*, etc.) sowie einiger Module (*brush*, *button*, etc.) der Tycoon-Graphikbibliothek (*svenv*) bedient, um eine graphische Ausgabe zu erzeugen. Auf der rechten Seite ist der so entstehende Objektgraph aufgetragen, wobei die Positionen der Teilobjekte, bzw. -graphen horizontal mit Bezeichnern im Skript korrespondieren. Die im Skript definierten Objekte sind grau unterlegt. In dem konkreten Programm, aus dem der Programmauszug in Abbildung 7.1 entstammt, nehmen ihre Speicherrepräsentationen wenige KB ein. Dem stehen ca. 6 MB der importierten Module gegenüber.

7.2 Eignungskriterien für Bindungstechniken

Um eine verbesserte Mobilität komplexer Objekte zu erreichen, muß bei der Übertragung von Objektgraphen die Möglichkeit bestehen, immobile Objekte auszulassen. Dies kann innerhalb der Anwendungssprache (TL) durch Manipulationen variabler Bindungen geleistet werden. Allerdings wird dadurch die Programmstruktur wesentlich beeinflußt. Transparente Manipulationen sind nur auf der Systemebene möglich. Hierfür bieten die in Abschnitt 4.3 beschriebenen Linearisierungs- und Delinearisierungsfunktionen, die bei RPCs zum Marshalling und Unmarshalling eingesetzt werden, einen geeigneten Angriffspunkt.

In jedem Fall ist die Beeinflussung der Kommunikationstopologie[1] zu beachten. Zu bevorzugen sind Bindungstechniken, welche durch Kolokalität die Autonomie migrierender Objekte wahren und somit bezüglich der Bandbreite, Latenz und Zuverlässigkeit von Netzwerken sowie der Verfügbarkeit von Netzwerkknoten skalieren. Um Abhängigkeiten zwischen verteilten Objekten zu vermeiden, wird in der vorliegenden Arbeit versucht, Kommunikationen *tendenziell*

[1] In theoretischen Betrachtungen von Prozeß-Algebren [Sangiorgi 93] wird die Änderung der Kommunikationstopologie sogar als das wesentliche einer Migration angesehen. Von materiellen Umständen (Entfernungen, Zeit, Speicherplatz, etc.) wird dabei allerdings abstrahiert.

auf Migrationen autonomer Objekte, insbesondere auf Threads, zu beschränken. Allerdings wird im Gegensatz zu Telescript nicht so weit gegangen, Thread-Migrationen zur *einzigen* Form der Netzwerkkommunikation innerhalb[2] der Programmiersprache zu machen.

Eine transparente durch das Marshalling vollzogene Übersetzung von Objekten in Proxys wie z.B. in Emerald [Jul 88] oder Obliq [Cardelli 94] wird den genannten Anforderungen nicht gerecht, da sie adreßraumübergreifende Abhängigkeiten produziert. Andererseits bieten Proxys den Vorteil, auf transparente Weise geteilte Zugriffe (*Sharing*) auf gemeinsam genutzte Objekte zu modellieren. Also sieht TL zwar Proxys vor, nicht jedoch ihre tranparente *Erzeugung*, die wohlgemerkt von der transparenten *Anwendung* von Proxys zu unterscheiden ist.

Ein weiteres Emerald-Konstrukt, die explizite Definition des Zusammenhangs migrierender Objekte durch die **attach**-Anweisung genügt ebenfalls nicht den hier verfolgten Zielen. Denn dabei werden allzu leicht Objekte ausgelassen, für die dann implizit die Umwandlung in Proxys in Kraft tritt. Hingegen entspricht es nicht nur dem generalisierten Bindungsbegriff persistenter Objektsysteme, den Zusammenhang von Objekten als Normalfall zu behandeln und lose Enden als Ausnahmen anzusehen, sondern auf eben diese Weise wird auch die Autonomie mobiler Objektgraphen gefördert. Hinzu kommt, daß letzteres für die Autonomie von Netzwerkknoten, bzw. Objektspeichern entscheidend ist.

Ein wichtiges Kriterium für die Konkretisierung von Bindungsmechanismen für immobile Objekte ist das *Ausmaß* ihrer Verteilung im Netzwerk. Aus der Sicht eines migrierenden Objektgraphen treten folgende Fälle auf:

Lokale Objekte sind ausschließlich in dem Objektspeicher vorhanden, wo sich der Graph gegenwärtig befindet.

Entfernte Objekte sind ausschließlich in anderen Objektspeichern vorhanden.

Ubiquitäre Objekte sind praktisch in jedem Objektspeicher vorhanden, zu dem zukünftig migriert werden soll oder könnte. Hierbei werden verteilte Objektrepräsentationen als semantisch äquivalente Instanzen des selben Objektes interpretiert.

Falls „ein“ Objekt sowohl lokal als auch entfernt, aber nicht ubiquitär ist, wird die Sicht angenommen, daß es sich um *verschiedene* Objekte handelt, von denen eines lokal ist und alle anderen entfernt sind.

Im folgenden Abschnitt wird erläutert, wie dynamische Bindungen einzusetzen sind, um lokale und entfernte immobile Objekte in den Griff zu bekommen. Die beiden darauffolgenden Abschnitte sind der Problematik ubiquitärer Objekte gewidmet.

[2]Die Einbindung externer Dienste in Telescript geschieht mittels Sockets.

7.3 Typsichere temporäre Bindungen an immobile Objekte

Aus entfernten Objekten werden in Folge von Migrationen lokale Objekte und umgekehrt. Die Übergänge zwischen diesen Beziehungen stationärer, bzw. immobiler Objekte zu migrierenden Objekten erfolgen durch dynamische Bindungen. In TL (wie in den meisten Programmiersprachen) gibt es zwei Formen der dynamischen Bindung: Parameterübergaben an Funktionen und Zuweisungen an Variablen. Diese werden wie folgt eingesetzt:

Parameterübergaben ermöglichen durch entfernte Funktionen höherer Ordnung transferierten Funktionen den Zugriff auf entfernte Objekte.

Zuweisungen implementieren temporäre Bindungen migrierender Threads.

Beide Formen sind in den vorangehenden Kapiteln bereits einige Male verwendet worden. Ein Beispiel für eine in obigem Sinne parametrisierte Funktion höherer Ordnung ist in Abschnitt 5.1.3 durch die Funktion *info.bearbeite* angegeben.

bearbeite(R <:Ok anfrage(geschaeft :Geschaeft) :R ...) :R

Die in Abschnitt 5.1.3 in SQL-Syntax aufgeführte Beispielanfrage soll nun genauer untersucht werden. Dazu werden anstelle der durch Syntaxerweiterungen [Cardelli et al. 94] ermöglichten SQL-Formulierungen äquivalente Anweisungen in reinem TL betrachtet:

let *preis(v :Geschaeft.Verkauf) = v.preis*

let *berechnung(g :Geschaeft) :Geld = geld.AVG*[3]*(db.elements*[4]*(g.verkaeufe) preis)*

let *durchschnittVK :Geld = info.bearbeite(berechnung)*

Durch den Aufruf von *info.bearbeite* wird die Anfragefunktion *berechnung* in einem entfernten System auf die *dort* vorliegende Datenbank angewendet. Letztere wird zu diesem Zweck von der Implementation von *bearbeite* an den Parameter *g* übergeben.

Ein typisches Beispiel für die temporäre dynamische Bindung eines migrierenden Threads an ein immobiles Objekt ist in Abschnitt 6.1.3 ersichtlich. Der in Abbildung 6.3 illustrierte Workflow-Thread bindet sich in Folge seiner Migration vom Host der Sekretärin zu dem des Gruppenleiters dynamisch an die immobile Datenbank *budgetDB*. Bei der darauffolgenden Migration zur Buchhaltung wird die Bindung wieder gelöst. Diese temporäre dynamische Bindung ist im Thread-Skript durch die Schlüsselworte **with remote** gekennzeichnet:

```
...
migrate to gruppenLeiter with remote budgetDB :BudgetDB do
   ueberpruefe(budgetDB abrechnung.summe)
end
...
```

[3] Für jeden Datentyp ist ggf. eine eigene Funktion zur Durchschnittsberechnung erforderlich.

[4] Die in jedem Massendatentyp-Modul standardmäßig implementierte Funktion *elements* liefert jeweils eine Iteration (*Iter.T*) über die Elemente der ihr übergebenen Struktur. Im vorliegenden Fall handelt es sich um eine Datenbanktabelle vom Typ *db.T(Geschaeft.Verkauf)*.

Dieser auf Syntaxerweiterungen [Cardelli et al. 94] basierenden Notation liegenden folgende Anweisungen in reinem TL zugrunde:

```
...
begin
  let budgetDB = gate.migrateTo(:BudgetDB  gruppenLeiter)
    ueberpruefe(budgetDB  abrechnung.summe)
end
(* Ab hier ist budgetDB nicht mehr gebunden. *)
...
```

Die dynamische Bindung findet innerhalb der Funktion *gate.migrateTo*, die in Abschnitt 6.2.2 beschrieben ist, in Form einer Zuweisung statt. Es ist also nicht erforderlich, einen zusätzlichen Mechanismus in die Sprache einzuführen. Auch das Lösen der Bindung geschieht mit gewöhnlichen Mitteln, nämlich durch die Beschränkung des Sichtbarkeitsbereiches der lokalen Variablen *budgetDB* durch einen Block.

Aufgrund dieser nahtlosen Integration verteilter und konventioneller TL-Konstrukte ergibt sich eine elegante Möglichkeit, das „Einsammeln“ entfernter Objekte durch mobile Agenten zu modellieren. Durch das Verlassen des lokalen Sichtbarkeitsbereiches eines durch **migrate to**, **do** und **end** definierten Blockes kann ein ursprünglich temporäre gebundenes Objekt zum Gegenstand einer dauerhaften Bindung gemacht werden.

Zum Beispiel kann ein mobiler Reisebuchungsagent, der einen immobilen Reisebüroagenten zur Erstellung eines Flugtickets veranlaßt, von jenem erhaltene Flugdaten fest an sich binden, bei nachfolgenden Migrationen mit sich führen und schließlich an seinen Ursprungsort liefern:

```
...
let home = gate.here[5]()   (* Ursprungsort merken *)

let flugInfo =
  migrate to reiseBuero with remote r do
    ...
    r.bucheFlug(ziel  zeitraum)
    ...
  end

...

(* Nach einigen weiteren Aktionen zurück nach Hause: *)
migrate to home do
  ...
  print.string("Für Sie gebucht ist Flug Nr.: " <> flugInfo.flugNummer)
  ...
end
```

[5] Siehe Anhang D.2

Wenn ein Knoten zu sehr vielen verschiedenartigen oder zu nicht vorhersehbaren Zwecken genutzt werden soll, können Migrationspforten wahlweise mit allgemeiner verwendbaren Objekten versehen werden. Im Extremfall wird einfach ein Namensdienst angeboten. Dadurch ist eine bestehende Pforte außerdem für jede beliebige Erweiterung ihres Leistungsangebotes offen. Allerdings erfolgen die entscheidenden dynamischen Typüberprüfungen dann erst jeweils *nach* der Migration. In folgendem Beispiel führt die Funktion *nameService.lookup* die erforderlichen Typtests durch.

```
migrate to kartenZentrale with remote nameService do
  try
    let theater = nameService.lookup(:Theater "Schauspielhaus")
    if theater.reserviereKarten(datumUhrzeit) then
      ...
    else
      let kino = nameService.lookup(:Kino "Ufa")
      kino.reserviereKarten(film datumUhrzeit)
    end
  else
    ...
  end
end
```

Ein solcher Namensdienst ist nur eine von vielen Möglichkeiten, eine Einrichtung für dynamische Bindungen allgemein zu halten. Stattdessen kann z.B. auch eine komplexe Datenbank, die mehrdimensionale Suchanfragen erlaubt, angeboten werden. Ferner muß der Namensdienst nicht notwendigerweise anwendungsspezifisch sein. Er kann auch dazu dienen, Systemdaten zu liefern, etwa um eine Laufzeitanalyse für verteilte Programme zu unterstützen. Allerdings liegt in einem solchen Fall wohl eher eine ubiquitäre Verteilung vor, was die Verwendung der im nächsten Abschnitt beschriebenen Bindungstechnik nahelegt.

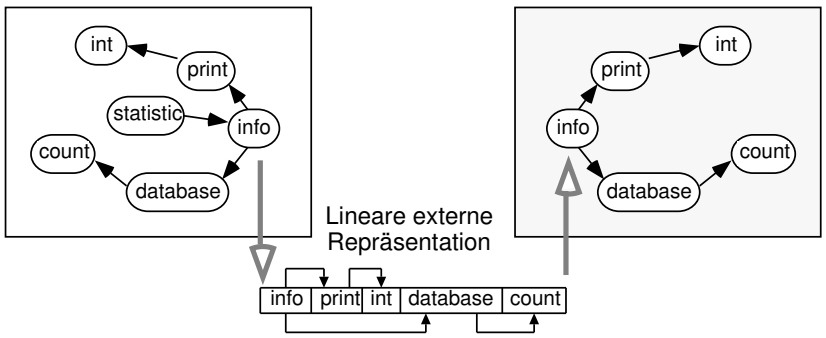

Abbildung 7.2: Übertragung der transitiven referentiellen Hülle eines Objektes

7.4 Dynamisches Linken ubiquitärer Objekte

Tycoon-Anwendungen machen in aller Regel starken Gebrauch von Bibliotheksfunktionalität wie etwa von Fenstersystemen, Kommunikations-Software, Font-Tabellen, Massendatenstrukturen und dem reflektiven Tycoon-Compiler. Viele dieser Bibliotheken können als ubiquitäre Objekte angesehen werden, da sie praktisch in allen Tycoon-Objektspeichern vorkommen. Darüberhinaus sind die weitaus meisten Bibliotheken praktisch zustandslos.

Diese Charakteristik kann dazu genutzt werden, den Netzwerkverkehr erheblich zu vermindern, bzw. die Mobilität komplexer Objekte entscheidend zu erhöhen. Die Grundidee besteht darin, eine Bindung an ein ubiquitäres Objekt beim Marshalling durch einen symbolischen Bezeichner (*Linksymbol*) zu repräsentieren. Dieser dient dann auf einer Empfängerseite dazu, beim Unmarshalling eine korrespondierende Kopie des Objektes aufzufinden und an seiner Statt einzubinden. Letzterer Vorgang entspricht genau dem bekannten Konzept des *dynamischen Linkens*.

Zur Veranschaulichung der Wirkungsweise des dynamischen Linkens wird folgendes Beispiel betrachtet.

```
module statistic
import database  print
export
   let info() = print.int(database.count())
   ...
end;
```

Der Funktionsabschluß von *info* enthält Verweise auf zwei weitere Module. Wenn er per RPC übertragen wird, dann werden diese sowie alle transitiv von ihnen referenzierte Objekte ebenfalls übertragen. Dieses in Abbildung 7.2 illustrierte Verhalten kann jedoch durch eine Registrierung der betreffenden Module als ubiquitäre Objekte verhindert werden. Dazu sind sowohl auf der Sender- als auch auf der Empfängerseite folgende Anweisungen auszuführen.

```
dynLink.registerModule(database)
dynLink.registerModule(print)
```

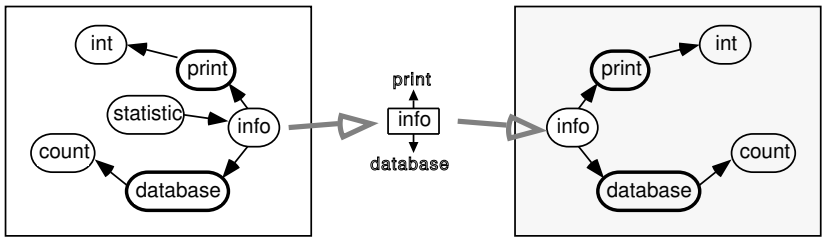

Abbildung 7.3: Dynamisches Linken ubiquitärer Objekte

Die Funktion *dynLink.registerModule* aktiviert vorbestimmte Linksymbole, die in jedem Modul bereits zur Compile-Zeit als Annotation gespeichert werden. Unter der Voraussetzung, daß die beteiligten Programme ihre ubiquitären Module aus ein und demselben Compilat (Objektdatei) bezogen haben, liegen daher automatisch übereinstimmende Linksymbole vor. Verwechslungen sind ausgeschlossen, da jeder Compilerlauf ein weltweit und allzeit eindeutiges Symbol erzeugt, das sich jeweils aus einer Hardware- und Betriebssystem-Kennung, einer Geräte-Identifikation, dem aktuellen Datum und der aktuellen Uhrzeit zusammensetzt.

Unter diesen Voraussetzungen entsteht bei einer Übertragung von *info* das in Abbildung 7.3 gezeigte Szenario. Von der Marshalling-Routine wird erkannt, daß die Module *database* und *print* speziell markiert sind. Daraufhin werden an ihrer Stelle Linksymbole übertragen. Auf der Empfängerseite vervollständigt die Unmarshalling-Routine den Objektgraphen durch lokale Instanzen der ausgelassenen Module. Die Zuordnung von Objekten und Linksymbolen geschieht durch Hash-Tabellen. Zur Effizienzsteigerung erhält jedes Linksymbol bereits bei seiner Erzeugung einen vorberechneten Hash-Index zugeteilt, der in einem speziellen Objektfeld gespeichert wird. Alle erforderlichen Erweiterungen des Marshallings und Unmarshallings sind, wie in Abschnitt 4.3.3 beschrieben, durch Funktionsparameter in die grundlegenden TXR-Linearisierungs- und Delinearisierungsroutinen integriert.

Dynamisches Linken hat folgende Vorteile:

- ▷ Das zu übertragenden Datenvolumen wird entscheidend reduziert.
- ▷ Die Lokalität des Programms bleibt erhalten.
- ▷ Übertragene Objekte bleiben vollkommen autonom.

Deshalb ist dynamisches Linken insbesondere für migrierende Threads unentbehrlich. Threads enthalten häufig eine Vielzahl von Bindungen an ubiquitäre Objekte, wobei es sich meist um Standardmodule handelt. Ein typisches Beispiel ist der in Abschnitt 7.1 erwähnte Thread, dessen Bindungsstruktur in Abbildung 7.1 dargestellt ist. Durch dynamisches Linken gelingt es in diesem Fall, den zu übertragenden Objektgraphen auf den nur wenige KB umfassenden, grau unterlegten Kernbereich einzugrenzen.

Eine besonders flexible Möglichkeit, ubiquitäre Objekte zu registrieren, bietet die Funktion *dynLink.register*, welche nicht auf Module beschränkt ist, sondern *beliebige* Objekte akzeptiert. Dies ist eine Generalisierung der eingeschränkten Fähigkeiten dynamischen Linkens in anderen verteilten Systemen (vgl. Emerald [Jul 88] und SOS [Shapiro et al. 89; Shapiro 93]). Allerdings muß nun das erforderliche Linksymbol explizit angegeben werden.

```
dynLink.register(database  "myDatabaseSymbol")
dynLink.register(print     "Modul1017")
```

Diese Methode hat ferner den Vorteil, daß unabhängig entstandene Objekte einander zugeordnet werden können. Allerdings hat sie auch gravierende Nachteile:

- Sie ist typunsicher. Im Gegensatz zu *dynLink.regsiterModule* besteht die Möglichkeit, daß zwei vollkommen unterschiedliche Objekte das gleiche Symbol erhalten.
- Der Programmierer ist für die Richtigkeit der semantischen Äquivalenz ubiquitärer Objekte verantwortlich.
- Es ist schwierig, die Installationen der als ubiquitär vorgesehenen Objekte durchzuführen.
- Es ist schwierig, Registrierungen an entfernten Orten durchzuführen.

Um letztere Administrationstätigkeiten durchzuführen, muß man sich entweder persönlich zu den jeweiligen Hosts begeben und die erforderlichen Zeilen eintippen oder aber es müssen Programme geschrieben werden, die diese Aufgaben zwar automatisch erledigen, aber dafür zusätzliche Wartungsprobleme verursachen. Im folgenden Abschnitt wird eine typsichere Alternative zur Verbreitung ubiquitärer Objekte beliebigen Typs beschrieben, die keinen besonderen Administrationsaufwand erfordert.

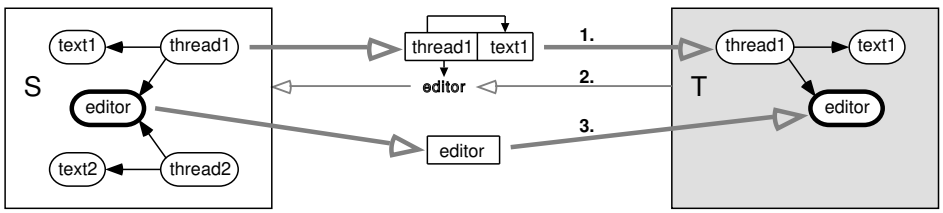

Abbildung 7.4: Die drei Schritte der automatischen Replikation

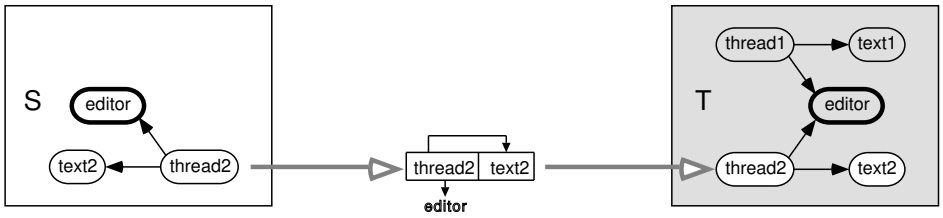

Abbildung 7.5: Dynamisches Linken eines automatisch replizierten Objektes

7.5 Automatische Replikation

Dynamisches Linken kann nicht erfolgreich durchgeführt werden, wenn die vorgegebenen Linksymbole am Zielort nicht identifiziert werden. Dies passiert, wenn lokale Instanzen ubiquitärer Objekte für die ankommenden Linksymbole nicht registriert oder nicht vorhanden sind. Dieser Fall wird auf Seiten des Senders in einem ersten Ansatz durch das Abbrechen der Übertragung und das Auslösen einer dynamischen Ausnahme gehandhabt. Der Empfänger wird in den Zustand vor Beginn der Übertragung zurückgesetzt.

Da bei allen in dieser Arbeit beschriebenen Techniken der Tycoon-Kommunikationen eine bidirektionale Verbindung besteht, kann jedoch auch wesentlich flexibler reagiert werden: der Empfänger kann durch entgegengesetzte Anfragen fehlende Objekte dynamisch beim Sender nachfordern. Dies wiederholt sich rekursiv, bis die transitive Hülle der ursprünglichen Sendung komplett ist. Alle beim Sender als ubiquitär registrierten Objekte, die zum Empfänger kopiert werden, werden dort dynamisch ebenfalls als ubiquitär registriert. Nachfolgende Anfragen verlaufen dann entsprechend abgekürzt. Dies alles verhält sich orthogonal zu den in Kapitel 5 und 6 beschriebenen verteilten Programmiermustern.

Abbildung 7.4 illustriert eine automatische Objektreplikation in einem Workflow-orientierten Szenario, in dem zwei Tycoon-Threads auf Knoten S eine geteilte Editor-Software *editor* verwenden, um Textdokumente zu editieren, die jeweils zu einem Thread lokal sind.

Das Objekt *editor* sei für dynamisches Linken registriert. Die Migration von Thread *thread1* involviert die Übertragung des Thread-Codes, des Textes *text1* und eines im Vorfeld für *editor* generierten symbolischen Bezeichners (Schritt 1 in Abbildung 7.4). Da *editor* in T noch nicht registriert ist, wird in Schritt 2 explizit die Nachsendung von *editor* angefordert und schließlich wird *editor* inklusive seiner transitiven Hülle übertragen (Schritt 3).

Eine anschließende Migration von Thread *thread2* von S nach T wird dann wie in Abschnitt 7.4 beschrieben mittels dynamischen Linkens gehandhabt (siehe Abbildung 7.5).

Offensichtlich spielt die konkrete Ausprägung eines Linksymbols im Falle automatischer Replikation keine Rolle. Die Hauptsache ist, daß überhaupt ein Symbol vergeben wurde und daß es eindeutig ist. Dies leitet folgende Variante der beiden in Abschnitt 7.4 genannten Funktionen zur Registrierung ubiquitärer Objekte, die zum Einsatz kommt, wenn ein Objekt für automatische Replikation vorgesehen werden soll, das kein Modul ist.

registerUnique(X <:Ok x :X) :Ok

Diese Funktion registriert ihr Argument x mit einem eigens erzeugten weltweit und allzeit eindeutigen Linksymbol[6], das für den Benutzer unsichtbar bleibt. Auf diese einfache und effiziente Weise bewirkt die Kombination von Persistenz und automatischer Replikation die Installation ubiquitärer Objekte beliebigen Typs.

Die Deinstallation kann automatisiert werden, indem alle Linksymbole durch automatische Replikation installierter Objekte durch *Weak References*[7] [Horning et al. 93] erfaßt werden. Jeder *Weak Reference* wird eine Funktion zugeordnet, die automatisch aufgerufen wird, wenn die Garbage-Collection erkennt, daß keine „harten“, d.h. gewöhnlichen, Referenzen mehr auf das betreffende Objekt bestehen. Diese Funktion müßte im vorliegenden Fall die Registrierung zum dynamischen Linken löschen. Daraufhin würden sowohl das Objekt als auch sein Linksymbol aus dem Objektspeicher bereinigt.

In obigem Beispiel bleibt die Registrierung des Editors so lange bestehen, wie zumindest einer der beiden Threads ihn verwendet.

[6] Dazu wird die gleiche Methode wie im Falle der Compilation eines Moduls verwendet (vgl. Abschnitt 7.4).
[7] *Weak References* sind im Tycoon-System noch nicht implementiert.

7.6 Pseudo-Mobilität und Pseudo-Persistenz flüchtiger Objekte

Flüchtige Objekte wie Fenster-Handles und Bindungen an C-Datenstrukturen außerhalb des persistenten Tycoon-Objektspeichers können mit Hilfe des Moduls *persistent*, dessen Schnittstelle in Anhang E.1 aufgelistet ist, registriert und somit „pseudo-persistent" gemacht werden. Sie werden dann nach jedem Programmneustart automatisch wiederaufgebaut und vor jedem Herunterfahren oder Zurücksetzen automatisch abgebaut. Zusätzlich können sie mit Hilfe des Moduls *persistentLog* (siehe Anhang E.2) in einem Operations-Log erfaßt werden, das bei Migrationen dazu benutzt werden kann, sie auf der Senderseite ab- und auf der Empfängerseite wieder aufzubauen. Sie sind dann „pseudo-mobil".

Die Reihenfolge, in der flüchtige Objekte registriert werden, ist signifikant, denn typischerweise existieren Bindungen „jüngerer" Objekte an „ältere". Wenn flüchtige Objekte in ihrer Erzeugungsreihenfolge registriert werden, dann folgen die automatischen Wiederherstellungsoperationen der originalen Erzeugungssequenz und der Abbau erfolgt in umgekehrter Reihenfolge. Einen ähnlichen Mechanismus weist das SOS-System auf, in dem Objektabhängigkeiten durch explizite Nennung sogenannter *pre-requisite objects* [Shapiro et al. 89], die vor allen von ihnen abhängigen Objekten erzeugt worden sein müssen, anzugeben sind.

Jeder Tycoon-Objektspeicher enthält eine globale Liste, die alle flüchtigen Objekte unter Kontrolle des Moduls *persistent* referenziert. Die Elemente dieser Liste migrieren nie; stattdessen werden sie ggf. mit Hilfe des Moduls *persistentLog* auf der jeweiligen Empfängerseite automatisch wiederaufgebaut. Um sowohl das Anlegen neuer Objekte als auch explizite Destruktionen, wie z.B. das Schließen eines Fensters effizient (d.h. mit O(1)) zu unterstützen, ist die Liste sowohl doppelt verkettet als auch in Form eines Array repräsentiert.

Der Einsatz des Moduls *persistentLog* beschränkt sich nicht auf flüchtige Objekte, denn bei den im Log erfaßten Operationen spielt es keine Rolle, ob sie externe oder interne Objekte manipulieren. Daher kann zum Beispiel auch das *Ownership*-Konzept von Telescript [General Magic 95b] simuliert werden. Es sieht vor, daß migrierende Agenten von Objekten, die ihnen „gehören", keine Kopien hinterlassen. Stattdessen werden diese ohne Rücksicht auf die referentielle Integrität des jeweils verlassenen Objektspeichers gelöscht.

Zwischen dem TL-Ansatz und Telescript bestehen folgende Unterschiede, die durch die Wahrung des generalisierten Bindungsbegriffes in TL motiviert sind:

- Der *Ownership*-Mechanismus ist fest im Telescript-System verankert (*builtin*). In TL liegt dagegen eine Bibliothekserweiterung vor (*add-on*, vgl. [Matthes, Schmidt 91]).
- In Telescript ist *Ownership* der Normalfall. In TL ist es dagegen die Ausnahme, wenn ein Objekt einer solchen Zuordnung unterliegt.
- Telescript-Objekte, die einem Agenten nicht gehören, migrieren *nicht* mit ihm.[8] In TL ist es dagegen der Normalfall, daß transitiv erreichbare Objekte mitmigrieren.

Aufgrund dieser Entwurfsentscheidungen ist TL nicht auf den Telescript-Stil festgelegt.

[8] Hierbei werden keine Klassen betrachtet. Für diese gelten eigene Regeln.

Kapitel 8

Erfahrungen und Ausblick

Bis auf wenige explizit angegebene Ausnahmen sind alle in den vorangehenden Kapiteln beschriebenen technischen Beiträge im Tycoon-System implementiert. Es sind bereits erste Anwendungen im Entstehen, von denen im nachstehenden Abschnitt drei exemplarisch aufgeführt sind.

Im Anschluß daran werden die wichtigsten Beiträge und Erfahrungen der vorliegenden Arbeit zusammenfassend bewertet. Die Dissertation schließt mit einem Ausblick auf weiterführende Arbeiten.

8.1 Anwendungen

Der Arbeitsbereich „Angewandte und Sozialorientierte Informatik“ (ASI) des Fachbereiches Informatik der Universität Hamburg nutzt die Offenheit und Kommunikativität des Tycoon-Systems zur Integration verteilter heterogener Komponenten eines komplexen Informationssystems, der Modelldatenbank für Verkehrssimulationen *MOBILE* [MOBILE 95]. In diesem Projekt wird Tycoon als plattformunabhängige Kommunikations-Middleware eingesetzt, wobei TL hauptsächlich als IDL (*Interface Definition Language*) fungiert.

Die Arbeitsgruppe „Computergestützte InformationsSysteme“ (CIS) am Institut für Kommunikations- und Softwaretechnik am Fachbereich Informatik der Technischen Universität Berlin setzt das Tycoon-System zur praktischen Umsetzung in ihrem Forschungsbereich „Objektorientierte Modellierung und Entwurf heterogener, verteilter Informationssysteme“ entwickelter Konzepte ein [Kutsche et al. 95; Busse 96].

Der Arbeitsbereich DBIS des Fachbereiches Informatik der Universität Hamburg verwendet das Tycoon-System für statistische Auswertungen von WWW-Zugriffen (*Web Site Profiling*). Dabei kommt eine im Rahmen der vorliegenden Arbeit entstandene, auf der plattformunabhängigen Socket-Schnittstelle aus Abschnitt 4.2 basierende Verknüpfung eines HTTP-Servers mit den *persistenten* Threads eines konkurrenten Tycoon-Server-Prozesses zum Einsatz.

Die interne Organisation letzteren persistenten Objektsystems ist in Abbildung 8.1 schematisch dargestellt. Entfernte WWW-Clients werden im Tycoon-System anhand sogenannter *Cookies* [Netscape 96], die ihnen jeweils vom HTTP-Server zugeteilt werden, identifiziert und es wird ihnen je ein Tycoon-Thread als „Kommunikationspartner“ fest zugeordnet. Dabei

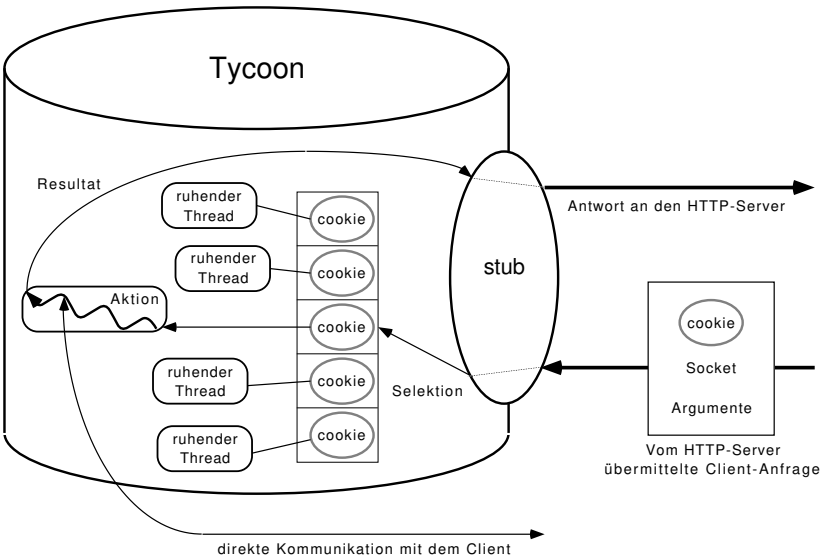

Abbildung 8.1: WWW-Anbindung persistenter Threads

erlaubt die Persistenz der Threads eine elegante Modellierung langlebiger Benutzerbeziehungen (siehe Abschnitt 6.1.2). Es gelingt somit eine langlebige Personalisierung der HTTP-Kommunikation.

Da diese Aufgabenstellung noch recht eng gefaßt ist, werden zwar bisher nur wenige Teilergebnisse der vorliegenden Arbeit genutzt, aber es deuten sich bereits weitreichende Folgeentwicklungen an. In einem nächsten Schritt werden mehrere verteilte Tycoon-Prozesse, die jeweils einem HTTP-Server an die Seite gestellt sind, per Tycoon-RPC miteinander verknüpft. Auf diese Weise entsteht eine leistungsfähige Infrastruktur für interaktive mobile Internet-Agenten:

- WWW-Clients bieten eine universelle, plattformunabhängige Benutzerschnittstelle.
- Das Tycoon-System bietet ein universelles, plattformunabhängiges Kommunikations- und Mobilitätsmedium.

Die resultierende Softwarearchitektur ist in Abbildung 8.2 ersichtlich.

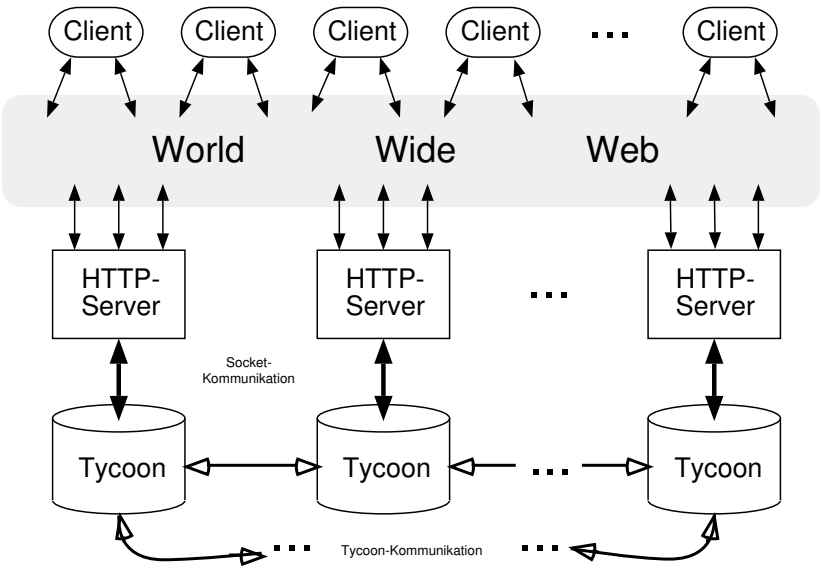

Abbildung 8.2: Eine Softwarearchitektur für interaktive mobile Internet-Agenten

8.2 Resümee

Nach den Maßstäben der in der Einleitung vorgenommenen Bewertung (Abbildung 1.1) der Mobilität in Programmiersystemen gelten für Tycoon die in Abbildung 8.3 ersichtlichen Mobilitätsgrade. Rekursive Strukturen, Funktionsabschlüsse und Threads sind im Vergleich zu anderen Systemen maximal mobil. Auch Kommunikationsidentifikatoren (hier Proxys) sind in TL maximal mobil.

Die besonders hohen Mobilitätsgrade im Tycoon-System sind vor allem auf die Entwicklung eines vollkommen *typunabhängigen*[1] RPC-Mechanismus zurückzuführen. Polymorphe entfernte Funktionen höherer Ordnung decken auf elegante Weise das gesamte Spektrum der Verteilung von Daten, Code und Threads ab.

Die einzige Einschränkung[2] besteht darin, daß in TL ausschließlich für Module, bzw. Funktionstupel Proxys erzeugt werden. Dies ist jedoch von geringerer Bedeutung, da auf eine transparente *Erzeugung* von Proxys bewußt verzichtet wird. Stattdessen stehen Bindungstechniken für immobile Objekte bereit, die bezüglich Latenz, Bandbreite und Zuverlässigkeit von Netzwerken *skalieren*[3], da sie die *Autonomie* migrierender Objekte erhalten: typsichere dynamische Bindungen und dynamisches Linken.

In Kombination dieser Bindungstechniken mit der Mobilität von Funktionsabschlüssen und Threads sowie dem generalisierten Bindungsbegriff und den lexikalischen Sichtbarkeitsregeln

[1] Eine andere Bezeichnung ist „typvollständig“ (*type-complete*) [Mira da Silva 95b]. Soweit bekannt, ist der im Rahmen der vorliegenden Dissertation entwickelte Mechanismus der erste, der diese Eigenschaft tatsächlich hat.

[2] Dies ist in Abbildung 8.3 durch die Bewertung *++* anstelle von *+++* in der Kategorie Proxys angedeutet.

[3] In dieser Hinsicht unterscheidet sich Tycoon maßgeblich von Emerald und Obliq, deren Einsatzgebiet weitgehend auf lokale Netzwerke beschränkt ist.

Mobilität:	Rekursive Strukturen	Code	Funktions-abschlüsse	Threads	Kommunikations-identifikatoren
ONC/DCE-RPC	(+)	-	-	-	-
CORBA	(+)	-	-	-	++
Modula-3	++	-	-	-	++
Erlang	++	+	-	-	-
Java	++	++	-	-	-
SOS, DC++	+	++	oo	-	++
Napier88	+++	++	++	-	-
Phantom	++	++	+++	-	++
Obliq	++	++	+++ oo	-	+++
FACILE	+++	+++	+++	-	++
Telescript	+	+++	oo	++	-
Emerald	++	+++	oo	+	+++
Tycoon	**+++**	**+++**	**+++**	**+++**	**++**

+++	durchgängig unterstützt	(+)	nur bedingt und sehr unpraktisch
++	etwas unpraktisch/eingeschränkt	-	nicht unterstützt
+	simuliert oder ziemlich eingeschränkt		
oo	Funktionsabschlüsse indirekt in Objekten enthalten (bei objektorientierten Sprachen)		

Abbildung 8.3: Die erreichten Mobilitätsgrade im Vergleich zu anderen Systemen

in TL ist im Zuge der vorliegenden Arbeit ein „autonomiebewahrender Programmierstil“ für verteilte Anwendungen entstanden. Dieser Stil erlaubt unter Gewährleistung hoher Zuverlässigkeit der Programmausführung, migrierende Anwendungsstrukturen direkt auf Konstrukte der Programmiersprache TL abzubilden. Insbesondere können Workflows und Agenten durch migrierende persistente Threads modelliert werden.

Dienstprogramme zur Vermittlung von Netzwerkadressen können unter Ausnutzung von Code-Mobilität flexibler und effizienter gestaltet werden (siehe Abschnitt 5.2.3.2 und 5.2.4). Dies erhöht die Mobilität von Proxys, die wiederum in Funktionsabschlüssen gekapselt sein können.

In der vorliegenden Arbeit wird auch die Mobilität kompletter Programme berücksichtigt: Bindungen an entfernte Funktionen bleiben sowohl bei Client- als auch bei Server-Verlagerungen erhalten. Dadurch wird wiederum die Mobilität menschlicher Benutzer unterstützt, die mit Programmen interagieren. Somit trägt diese Dissertation auf drei Ebenen zur Mobilität bei: Objekte, Programme und Benutzer.

Außer den oben genannten Beiträgen heben folgende Merkmale die geschaffene Infrastruktur zur Kommunikationsprogrammierung deutlich von anderen Programmierumgebungen (vgl. Kapitel 2) ab:

- ▷ Das gesamte Leistungsspektrum steht in plattformunabhängiger Weise zur Verfügung (siehe Abschnitt 4 und 5.2.2.1).
- ▷ Kommunizierende Objekte sind persistent:
  - Client-Bindungen tolerieren Prozeßbeendigungen, Abstürze und Neustarts (siehe Abschnitt 5.1.1 und 5.1.4).
  - Der Wiederanlauf beim Neustart von Server-Programmen muß nicht ausprogrammiert werden (siehe Abschnitt 5.1.4 und 6.1.2).
  - Thread-Migrationen sind transaktional abgesichert (siehe Abschnitt 6.2.3).
- ▷ Die Mobilität von Funktionsabschlüssen wird zur Steigerung der Flexibilität und Effizienz von Systemdiensten genutzt (siehe Abschnitt 5.1.2, 5.2.3.1 und 5.2.4).
- ▷ Die Auswahl und der Einsatz entfernter RPC-Dienste geschehen in einem *einheitlichen* sprachlichen Rahmen. Dabei steht für die algorithmische Bewertung von Dienstattributen der gesamte Sprachumfang zur Verfügung und für dynamische Typvergleiche wird keine zusätzliche Sprache (insbesondere keine *Interface Definition Language*) benötigt.
- ▷ Die Stub-Generierung für RPCs erfolgt dynamisch und ohne Reflexion; sie ist polymorph und effizient (siehe Abschnitt 5.2.2.2).
- ▷ Entfernte Funktionsaufrufe sind statisch typsicher (siehe Abschnitt 5.2.2.2).
- ▷ Lokale Operationen nach Thread-Migrationen sind statisch typsicher (siehe Abschnitt 6.2.2).
- ▷ Dynamisches Linken ist auf Objekte beliebigen Typs anwendbar und mit automatischer Replikation integriert (siehe Abschnitt 7.4 und 7.5).
- ▷ Flüchtige Objekte werden durch einen polymorphen Mechanismus in kanonischer Weise persistent und mobil (siehe Abschnitt 7.6).

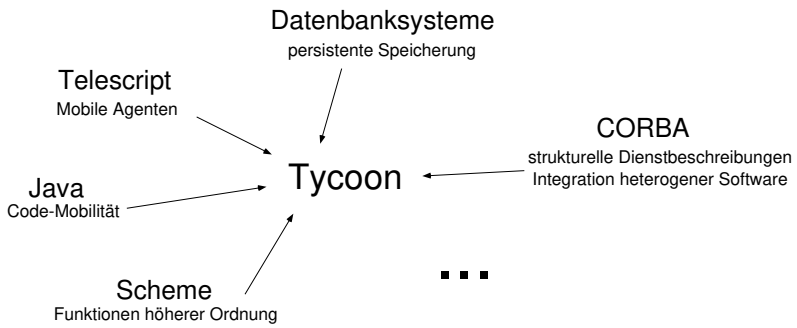

Abbildung 8.4: Antizipation der Weiterentwicklung von Programmiersystemen

Verteilte Programmierung profitiert in zweifacher Hinsicht von der sprachlichen und architekturellen Orthogonalität persistenter Objektsysteme. Zum einen verringert sich der Aufwand zur Erstellung verteilter Anwendungen und zum anderen wird die Implementierung einer leistungsfähigen Programmierinfrastruktur für verteilte Programmierung vereinfacht.

Zusammenfassend wird festgestellt:

> Es hat sich erwiesen, daß der grundlegende Ansatz, ein persistentes Objektsystem als Basis für verteilte Programmierung zu verwenden, tatsächlich zur Bereitstellung eines *einheitlichen* programmiersprachlichen und architektonischen Rahmens in offenen Systemumgebungen führt, der orthogonale Persistenz mit einer substantiell verbesserten *Mobilität* von Objekten beliebigen Typs verbindet.

Es sind nicht nur die Möglichkeiten verteilter Programmierung in persistenten Objektsystemen erheblich verbessert worden, sondern es ist auch eine generelle Perspektive für die Weiterentwicklung von Programmiersystemen deutlich geworden.

Das Tycoon-System ist aufgrund der genannten Beiträge konzeptionell inmitten einer weitaus größeren Schar unterschiedlicher Programmiersysteme als bisher positioniert. In Abbildung 8.4 sind exemplarisch einige (kommerziell) relevante Programmiersysteme aufgeführt, die dazu tendieren, Leistungsmerkmale der jeweils anderen zu integrieren. Viele dieser Entwicklungen werden durch die vorliegende Dissertation antizipiert.

8.3 Weiterführende Arbeiten

In diesem letzten Abschnitt der Arbeit werden einige vielversprechende Ansätze für weiterführende Arbeiten aufgezeigt.

Datensicherheit

Ein wichtiges Thema, das in der vorliegenden Arbeit bewußt ausgelassen wurde, da es in einer im Tycoon-Projekt parallel entstehenden Dissertation [Rudloff 96] ausführlich behandelt wird, ist das der Datensicherheit. Der wesentliche Ansatz jener Arbeit besteht darin, die Grundoperationen der Autorisierung unterhalb der Objektspeicherschnittstelle TSP, aber oberhalb der Objektspeicherimplementation anzusiedeln (vgl. Abschnitt 3.3). Auf diese Weise ist nur eine relativ geringfügige Erhöhung der Systemkomplexität erforderlich, um sämtliche Objektzugriffe unabhängig von der konkreten Objektspeicherimplementation zu kontrollieren. Es ist zu vermuten, daß sich die durch diese Entwurfsentscheidung bedingten Performanzeinbußen in Grenzen halten, da Laufzeittests einer analog konstruierten Schichtung[4] [Kornacker 95] des TSP zu keiner wesentlichen Verschlechterung geführt hat.

Die Authentisierung erfolgt unter Nutzung des *Kerberos*-Systems [Steiner et al. 88] als externer Dienst. Zum Zwecke der Autorisierung von Aktivitäten wird jedem Thread eine interne Benutzerkennung (*Principal*) zugewiesen. Migrierende Threads behalten diese Kennung bei, sodaß ihre Operationen weiterhin dem sie ursprünglich initiierenden Benutzer zugeordnet werden können.

Die Zugriffsrechte von Objekten können individuell vergeben werden. Damit migrierende Threads in ihrem jeweiligen Zielobjektspeicher auf kontrollierte, aber dennoch flexible Weise Zugriffsrechte auf lokale Objekte erhalten, ist ein an die *setuid*-Funktion in UNIX-Systemen [Curry 92] angelehnter Mechanismus vorgesehen: entsprechend markierte Funktionsabschlüsse veranlassen bei ihrem Aufruf einen temporären Wechsel des *Principals*.

Verwendung einer objektorientierten Sprache

In TL werden ausschließlich Funktionstupel (bzw. Module) durch Proxys repräsentiert. Für jeden weiteren Typkonstruktor (*Record*, *Array*) wäre ggf. ein weiterer spezieller Verteilungsmechamismus erforderlich. Außerdem sind einfache Werte (z.B. Werte der Typen *Integer*, *Bool*, *Char*) nicht durch eigene Speicherobjekte repräsentiert und können daher mit den vorhandenen Techniken überhaupt nicht durch transparente Proxys abgebildet werden. Eine diesbezüglich einheitliche Behandlung aller TL-Werte hätte weitreichende negative Konsequenzen für andere bewährte Entwurfsentscheidungen.

In einer *rein objektorientierten* Sprache würde hingegen nur noch ein einziger gemeinsamer Angriffspunkt zur Verteilung von Objekten beliebigen Typs bestehen. Allerdings müßte die betreffende Sprache den Anforderungen persistenter Objektsysteme gerecht werden (siehe Abschnitt 1.2).

[4] Im TSP angesiedeltes Logging zum Zwecke der Schaffung persistenter Sicherungspunkte.

CORBA-Integration

Durch die Fähigkeit zur transparenten Einbindung externer Dienste in Module wirkt TL wie eine IDL. Zudem sind sich TL-Modulschnittstellen und Interfaces in CORBA-IDL strukturell sehr ähnlich: der hauptsächliche Inhalt besteht jeweils aus einer Liste von Funktionssignaturen.

Um die beträchtlichen Investitionen von CORBA-Entwicklern und Anwendern zu nutzen, kann Tycoon als CORBA-Client fungieren. Dazu ist jeweils ein von einer CORBA-Implementation generierter Client-Stub als externe *C*-Bibliothek in TL einzubinden. Es liegt nahe, diesen Vorgang zu automatisieren. Dies kann mit Hilfe eines in [Geisler 95] beschriebenen Generatorprogramms für Programmierschnittstellen geschehen, das an alternative Schnittstellenbeschreibungssprachen anpaßbar ist.

Eine besondere Herausforderung besteht darin, Konzepte zu erarbeiten, wie Code-Mobilität und Thread-Mobilität in CORBA-Programme eingebracht werden können. Dazu ist Tycoon auch als CORBA-Server einzusetzen.

Wie in Abschnitt 5.1.1 schon angedeutet, ist es ohne weiteres möglich, TL-Funktionen in CORBA-Server einzubinden. Sämtliche dazu erforderlichen Systemvoraussetzungen sind vorhanden. Insbesondere gilt:

- Die externe Programmierschnittstelle von TL ist bidirektional und transparent. Es ist nicht nur möglich, in TL transparent externe *C*-Funktionen aufzurufen, sondern umgekehrt können auch in *C* (oder *C++*) transparente TL-Aufrufe stattfinden.
- Das Tycoon-System ist in der Lage, als Subsystem zu fungieren. Es kann als dynamische oder statische *C*-Laufzeitbibliothek eingebunden werden. Dies bedeutet, daß sich ein anderes Programm, insbesondere ein CORBA-Server-Stub (*Server Skeleton*), der Dienste von Tycoon bedienen kann, ohne deshalb seine Rolle als Hauptprogramm aufgeben zu müssen.

Hier ist das oben erwähnten Generatorprogramms aus [Geisler 95] so zu konfigurieren, daß es CORBA-Server-Skeletons in TL aus in IDL vorliegenden Schnittstellen zu generiert.

Für den Fall der Neuerstellung von CORBA-Servern kann eine *generische* und voll automatisierte Einbindung von TL-Funktionen in CORBA [Otte et al. 96] geschaffen werden. Dazu sind folgende Erweiterungen des Tycoon-Systems erforderlich:

- Das Tycoon-Laufzeitsystem ist um eine Schnittstelle zu einem sogenannten *Object Adapter*, einem Modul, das die Kommunikation zwischen Server-Programm und Server-Stub ver- und übermittelt, zu erweitern. Es empfiehlt sich, speziell den *Basic Object Adapter* (BOA) zu verwenden, da dieser Adapter der einzige ist, dessen Unterstützung für jede CORBA-Implementation obligatorisch ist.
- CORBA-Server müssen eine Reihe von Standardfunktionen implementieren. Dazu gehören eine Initialisierungsfunktion, Signale an den BOA, ein Anfrage-Dispatcher und Deaktivierungscode. All dies kann von einer mit Funktionen parametrisierten TL-Funktion bereitgestellt werden. Letztere würde durch Aggregation von Funktionsabschlüssen, also ohne Reflektion, jeweils ein entsprechendes Modul erzeugen.

- ▷ Es ist eine Funktion zu schreiben, die aus TL-Schnittstellen CORBA-IDL generiert und im Dateisystem oder ggf. im Interface Repository ablegt.
- ▷ Es ist eine Routine zu schreiben, die die Generierung eines CORBA-Servers steuert. Ihre Aufgaben bestehen darin, die IDL-Generierung aufzurufen und anschließend die erforderlichen externen Dienstprogramme (Server-Skeleton-Generator, *C*-Compiler, Linker) ablaufen zu lassen.[5]

Unter diesen Voraussetzungen können für in TL geschriebene Module *dynamisch* CORBA-Server generiert und gestartet werden.

Natürlich bestehen gewisse Einschränkungen der Verwendbarkeit von TL-Modulen, da CORBA nicht alle TL-Konstrukte (Funktionen höherer Ordnung, Typoperatoren, etc.) unterstützt. Im Einzelfall wird es also erforderlich sein, ein TL-Modul mit einem *Wrapper* zu versehen. Auf der anderen Seite sind in TL alle wesentlichen Datentypen, Aggregationsformen und Gestaltungsmöglichkeiten für Funktionssignaturen vorhanden, die konventionelle Sprachen (auf welche die OMG mit CORBA ja im wesentlichen abzielt) zu bieten haben.

Eine gemeinsame Basis für mobile Agenten und Workflows

Zwischen den Implementierungen von mobilen Agenten und von Workflows durch migrierende persistente Threads bestehen keine wesentlichen technischen Unterschiede. Deshalb ist eine eindeutige Zuordnung zu einer der beiden Anwendungsklassen häufig nicht mehr ohne weiteres gegeben. Ob es sich bei einem konkreten Thread eher um einen Workflow oder eher um einen Agenten handelt, obliegt letztlich der Interpretation des Anwenders.

Die im Rahmen der vorliegenden Arbeit geschaffene gemeinsame Basis für mobile Agenten und Workflows ist nun durch auf bestimmte Anwendungsklassen spezialisierte Programmiermuster auszubauen.

Vor allem ist ein Transaktionsschema zu entwickeln, das sich auf einzelne migrierende Aktivitäten, die durch Threads implementiert sind, bezieht. Eine besonders vielversprechende Grundlage für dieses Vorhaben sind "Sagas" [Garcia-Molina, Salem 87]. Auch "Contracts" [Reuter 89; Wächter, Reuter 91] bieten einige wertvolle Anhaltspunkte, obwohl sie Einschränkungen bezüglich des Kontrollflusses migrierender Aktivitäten vorschreiben.

Automatisches Software-Management

Das in Abschnitt 7.5 beschriebene Verfahren zur automatischen Installation und Deinstallation von Objekten stellt erst den Anfang einer Entwicklung dar, die auf ein vollautomatisches Software-Management hinstrebt.

Im Gegensatz zum gegenwärtigen Entwicklungsstand der Sprache Java, der die automatische Installation *flüchtiger* Objekte vorsieht, bietet die in der vorliegenden Arbeit vorgestellte Lösung Persistenz[6].

[5] Durch die Funktion *process.execute* in der TL-Standardbibliothek *stdenv* können externe Programme, d.h. Betriebssystemprozesse, gestartet werden. Dies wird z.B. auch bei der Generierung von Maschinencode für TL via *C*-Compiler und Linker [Mathiske 92] genutzt.

[6] Seit kurzem ist auch eine persistente Java-Variante in Vorbereitung [Atkinson et al. 96].

Als erster Ausbauschritt bietet sich an, das vorgeschlagene Verfahren um Verfallszeiten für replizierte Objekte zu erweitern. Die Registrierung der automatisch zu replizierenden Softwarekomponente *editor* aus Abschnitt 7.5 würde sich dann wie folgt gestalten:

dynLink.registerUnique(editor **let** *expiration = 2 * week)*

Desweiteren stellen sich interessante konzeptionelle und technische Herausforderungen:

- die Integration dynamischen Linkens und automatischer Replikation mit einem Versionsmanagement,
- die Ablösung verteilter Dateisysteme als Medium kooperativer Software-Entwicklung, Wartung und Installation durch föderierte persistente Objektsysteme.

Anhang A

Die polymorphe Unterstützung entfernter Funktionen

In diesem Anhang sind die Schnittstellen der für die Anwendungsprogrammierung konzipierten, obersten Schicht der Tycoon-Bibiliothek *csenv* aufgelistet. Ihre Verwendung ist in den Abschnitten 5.1.2 und 5.1.3 beschrieben.

A.1 Die Client-Schnittstelle

Die Funktionen der Client-Schnittstelle zur Erzeugung von Proxys sind parametrisch polymorph. Mit dem Schlüsselwort **Dyn** versehene Typparameter werden für dynamische Typvergleiche verwendet, die zum einen der strukturellen Dienstauswahl dienen und zum anderen sicherstellen, daß alle Aufrufe entfernter Funktionen *statisch* typsicher sind.

```
interface Client
import :Iter :TypeRep sid
export
  ... (* Exception-Deklarationen *)

  Let Service = Tuple end  (* Der Supertyp aller RPC-Dienste. *)

  bindToService(S <:Service  id :sid.T(S)) :S
  (* Liefert ein Proxy für einen per Dienstidentifikator (id) angegebenen RPC-Dienst.
     Erzeugung von Dienstidentifikatoren: siehe "Server.ti" (Anhang A.2) *)

  bind(Dyn S <:Service  Dyn Attributes <:Ok  predicate(:Attributes) :Bool
       searchDepth :Int) :S
  (* Stellt eine Client-Bindung an einen entfernten RPC-Dienst her, der folgende Kriterien erfüllt.
     S            : Diensttyp, gleichzeitig Proxy-Typ
     Attributes   : Typ der Dienstattribute
     predicate    : Prädikat über den aktuellen Dienstattributen
     searchDepth  : Suchtiefe im Netzwerk (0: Host, 1:LAN, 2: ...)
     Liefert ein Proxy mit der Client-Bindung. *)
```

```
Let Info(S <:Service  Attributes <:Ok) = Tuple
  Dyn Svc <:S              (* konkreter Diensttyp *)
  Dyn A <:Attributes       (* konkreter Attributtyp *)
  attributes :A            (* aktuelle Dienstattribute *)
  id :sid.T(S)             (* abstrakter Dienstidentifikator *)
end
(* Typ eines Informationselementes, das einen RPC-Dienst beschreibt. *)

find(Dyn S <:Service Dyn Attributes <:Ok  predicate(:Attributes) :Bool
     searchDepth :Int) :Iter.T(Info(S  Attributes))
(* Sucht genau wie bind nach entfernten RPC-Diensten.
   Stellt jedoch noch keine Bindung her, sondern liefert Informationen
   über in Frage kommende Dienste. *)

end;
```

A.2 Die Server-Schnittstelle

Die Server-Schnittstelle repräsentiert Dispatcher-Objekte als abstrakten Datentyp. In Bezug auf RPC-Dienste ist sie analog zur Client-Schnittstelle (siehe Anhang A.1) parametrisch polymorph.

```
interface Server
import :Iter :CSTypes sid :Sid :TypeRep
export
  ... (* Exception-Deklarationen *)

  Let Service = Tuple end  (* Der Supertyp aller RPC-Dienste. *)

  T <:Ok  (* Der abstrakte Typ aller Dispatcher. *)

  new(multiThreaded :Bool) :T
  (* Liefert einen neuen Dispatcher, der eine (initial leere) Menge
     von RPC-Diensten verwaltet und deren RPCs abarbeitet.
     Wenn multiThreaded auf true gesetzt ist,
     erzeugt der Dispatcher für jeden von ihm bearbeiteten RPC einen eigenen Thread. *)

  free(dispatcher :T) :Ok
  (* Deaktiviert einen Dispatcher, gibt all seine Ressourcen frei. *)


  Let SearchFun = Fun(S <:Service  s :sid.T(S)  var at :Port.T) :Ok
  (* Der Typ aller Dienstsuchfunktionen, die in Proxys implantiert werden. *)

  register(dispatcher :T  Dyn S <:Service Dyn Attributes <:Ok  attributes() :Attributes
           search :SearchFun  service :S) :sid.T(S)
  (* Erweitert die Menge der registrierten Dienste eines Dispatchers.
     Liefert einen abstrakten Dienstidentifikator.
     Der Dispatcher ordnet dem durch service gegebenen Dienst zu:
       S           : Diensttyp
       Attributes  : Typ der Dienstattribute
       attributes  : Funktion, die die aktuellen Dienstattribute liefert
       search      : Dienstsuchfunktion, die in Proxys zu implantieren ist *)
  unregister(S <:Service  dispatcher :T  s :sid.T(S)) :Ok
  (* Entfernt einen Dienst aus der Menge der von einem Dispatcher
     verwalteten Dienste *)


  start(dispatcher :T) :thread.T(Ok)
  (* Aktiviert einen Dispatcher, startet dazu einen eigenen Thread.  *)

  ...

end;
```

Anhang B

Die schmale Implementationsbasis der Kommunikationsbibliothek

Die beiden in diesem Anhang aufgelisteten Schnittstellen bilden die schmale Implementationsbasis, auf der alle in TL verfügbaren Mechanismen der Netzwerkkommunikation aufbauen.

B.1 Die plattformunabhängige Socket-Schnittstelle

Die Funktionen dieser *C*-Schnittstelle für plattformunabhängige Socket-Kommunikation sind in Abschnitt 4.2 beschrieben.

```
interface Socket
import iNetAddress thread
export

  error :Exception end
  (* Wird ausgelöst, wenn in einer der nachstehenden Funktionen eine Fehlersituation eintritt. *)

  (* Socket-Typen - als Funktionen zu verwenden, da die entsprechenden „Konstanten“
     je nach Plattform unterschiedliche Werte haben: *)
  SOCK_STREAM()    :Int    (* Stream socket *)
  SOCK_DGRAM()     :Int    (* Datagram socket *)
  SOCK_RAW()       :Int    (* Raw-protocol interface *)
  ...

  (* Adreßfamilien: *)
  AF_UNSPEC    :Int    (* Unspezifiziert *)
  AF_UNIX      :Int    (* Lokal auf dem selben Host (pipes, portals) *)
  AF_INET      :Int    (* Internet: UDP, TCP, etc. *)
  ...
```

```
(* Protokollfamilien - bis auf weiteres den Adreßfamilien gleich: *)
PF_UNSPEC    :Int
PF_UNIX      :Int
PF_INET      :Int
...

Let Address = Tuple
  addressType    :Int            (* Adreßfamilie *)
  port           :Int            (* Port-Nummer *)
  inAddress      :iNetAddress.T  (* Internet-Adresse *)
end

Let T <:Ok  (* Abstrakter Typ aller Socket-Handles *)

new(domain, type, protocol :Int) :T    (* in C: socket(...) *)
destroy(sock :T) :Ok                   (* in C: close(...) *)

(* Folgende Funktionen haben ein gleichnamiges C-Pendant: *)

bind(sock :T  name :Address) :Ok

listen(sock :T  requestQueue :Int) :Ok
accept(sock :T  var accepted :Address) :T

connect(sock :T toSocket :Address) :Bool

write(sock :T data :word.T size :Int) :Ok
read(sock :T size :Int) :word.T

(* Diese Spezialfunktion veranlaßt ggf. den Abbruch des aktuellen Aufrufes von accept in Thread t: *)
abort(t :thread.T(Ok)) :Ok

end;
```

B.2 Typunsichere Systemdienste

Die hier aufgeführte generische Schnittstelle zu niederen Systemschichten, wird in Abschnitt 5.2 genutzt. Mit ihrer Hilfe gelingt es, RPC-Stubs zu erzeugen, ohne eine Generierung auf Quelltextebene durchführen zu müssen.

```
interface Unsafe  (* ACHTUNG: Alle hier aufgeführten Funktionen sind TYPUNSICHER! *).
import :ArrayOp word
export

  error :Exception end
  (* Wird ausgelöst, wenn in einer der nachstehenden Funktionen eine Fehlersituation eintritt. *)

  typeCast(x :Ok  X <:Ok) :X
  (* Typumwandlung ohne jede interne Operation, liefert x mit dem Typ X. *)

  copy(X <:Ok  from, into :X) :Ok
  (* Flaches Kopieren eines Speicherobjektinhaltes.*)

  nArguments(iPreviousFrame :Int) :Int
  (* Liefert die Anzahl der Argumente der Funktion,
     deren Aufruf relativ zum aktuellen iPreviousFrame Stack-Frames zurückliegt.
     nArguments(0) ist reflexiv und liefert daher stets 1.
     Bei negativem oder zu großem Argument wird error ausgelöst. *)

  getArguments(iPreviousFrame :Int  A <:Ok) :ArrayOp.T(A)
  (* Liefert die Argumente der Funktion,
     deren Aufruf relativ zum aktuellen iPreviousFrame Stack-Frames zurückliegt als Array.
     nArguments(0) ist reflexiv und liefert daher stets array 0 end :A.
     Bei negativem oder zu großem Argument wird error ausgelöst. *)

  execute(closure :Ok  arguments :Array(Ok)  R <:Ok) :R
  (* Ruft den Funktionsabschluß closure mit den Argumenten arguments auf
     und versieht das Ergebnis per typeCast (siehe oben) mit dem Typ R. *)

  externToNewBuffer(V <:Ok  value :V  var nBytes :Int) :word.T
  (* Liefert einen Hauptspeicherblock, der die externe Repräsentation
    von value enthält, sowie die Länge des Blockes in nBytes.
     Der Block muß explizit durch word.free freigegeben werden. *)

  internFromBuffer(buffer :word.T  nBufferBytes :Int  V <:Ok) :V
  (* Liefert einen TL-Wert, der aus der an der Hauptspeicheradresse buffer befindlichen
     externen Datenrepräsentation rekonstruiert wird.
     Mit nBufferBytes ist die Länge des bei buffer beginnenden  Blockes anzugeben.
     Das Ergebnis hat per typeCast (siehe oben) den Typ V. *)

end;
```

Anhang C

Die Standardfunktion zur Wiederauffindung entfernter Dienste

Dies ist die in Abschnitt 5.1.2 und Abschnitt 5.2.3.2 erwähnte Standardsuchfunktion *server.searchFun* zur Wiederauffindung entfernter Dienste durch RPC-Proxys.

```
let searchFun(S <:Service  s :sid.T(S)  var at :port.T) :Ok =
  loop
    try
      let madOfOldSite :port.T = tuple madService.portNumber  at.host end
      (* Kontaktaufnahme mit dem MAD auf dem vorherigen Host: *)
      let m = portClient.bind(madOfOldSite  :madService.T  madService.name)
      loop
        try
          (* Versuch, den Dienstidentifikator 's' in jenem LAN zu finden: *)
          at := m.translate(s  (* Suche auf das LAN beschränken: *) 1)
          if at.portnumber ==  1 then
            raise client.error (* Ausnahmsweise nichts gefunden. *)
          else
            exit (* Heureka! *)
          end
        else
          ... (* Irgendetwas funktionierte nicht. *)
        end
      end
    else
      ... (* Der MAD ist nicht bereit. *)
    end
  end
```

Anhang D

Grundoperationen migrierender persistenter Threads

In diesem Anhang sind die elementaren Schnittstellen zur Programmierung mit migrierenden persistenten Threads aufgeführt.

D.1 Grundoperationen persistenter Threads

Tycoon-Threads sind persistent. Sämtliche hier aufgeführten Funktionen bewirken *persistente* Zustandsänderungen. Die Grundzüge der Implementation persistenter Threads sind in Abschnitt 6.1.2 beschrieben.

```
interface Thread
export

  error :Exception end
  (* Wird ausgelöst, wenn in einer der nachstehenden Funktionen eine Fehlersituation eintritt. *)

  abortion :Exception end
  (* „Standard"-Ausnahme zum Abbrechen eines Threads. *)

  T(R <:Ok) <:Ok
  (* Ein Wert vom Typ T(R) ist ein Thread, der eine Funktion berechnet,
     die einen Wert vom Typ R liefert. *)

  State <:Ok
  (* Folgende Ausführungszustände werden unterschieden. *)

  runningState, blockingState, suspendedState,
  terminatedState, exceptionState :State
```

state(R <:Ok thread :T(R)) :State
(Liefert den aktuellen Ausführungszustand von thread. *)*

new(R <:Ok f(self :T(R)) :R) :T(R)
(Erzeugt einen neuen suspendierten Thread, der dazu bestimmt ist, f auszuführen.*
Für reflexive Operationen wird der Thread selbst bei seinem Start an self übergeben.
*Außerdem wird er als Funktionsergebnis von new für seinen Erzeuger zugreifbar. *)*

fork(R <:Ok f(self :T(R)) :R) :T(R)
(Wie new, aber der Thread wird sofort gestartet. *)*

launch(R <:Ok f(self :T(R)) :R) :Ok
(Wirkt wie eine gleichzeitige Ausführung von suspend(thread.self()) und fork(f),*
*vermeidet jedoch, daß das Ergebnis von fork referentiell erreichbar wird. *)*

duplicate(R <:Ok thread :T(R)) :T(R)
(Liefert eine flache Kopie von thread. *)*

join(R <:Ok thread :T(R)) :R
(Löst error aus, falls thread der aktuelle Thread ist.*
Blockiert den aktuellen Thread bis thread terminiert.
Wenn der Endzustand von thread terminatedState ist,
nimmt der wartende Thread seine Ausführung wieder auf,
indem join das Ergebnis von thread liefert.
*Andernfalls: error. *)*

joinCatch(R <:Ok thread :T(R)) :R
(Wie join, ABER: anstelle von error wird die identische Ausnahme,*
die thread beendete, **in** *den aktuellen Thread propagiert. *)*

suspend(R <:Ok thread :T(R)) :Ok
(Wenn thread sich* **in** *runningState oder blockingState befindet, wird er suspendiert,*
d.h. seine Ausführung angehalten und sein Zustand wird auf suspendedState gesetzt.
Wenn dieser Zustand bereits vorliegt, geschieht nichts.
*Wenn ein anderer Zustand vorliegt: error. *)*

run(R <:Ok thread :T(R)) :Ok
(Wenn thread sich* **in** *runningState oder blockingState befindet, geschieht nichts.*
Wenn suspendedState vorliegt, wird die Ausführung von thread wieder aufgenommen
und sein Zustand wird runningState.
*Bei terminiertem thread: error. *)*

```
(**** Synchronisationsprimitive: ****)

atomic(R <:Ok  action() :R) :R
(* Führt einen „Kritischen Abschnitt“ als atomare Handlung aus. *)

sleep(R <:Ok  thread :thread.T(R)  timeout :Real) :Ok
(* Suspendiert thread für timeout Sekunden. *)

(* In der Bibliothek threadenv befinden sich Module, die
   verschieden Synchronisationsmechanismen als polymorphe abstrakte Datentypen anbieten:
      semaphore mutex condition event rendezvous ... *)


(**** Fortgeschrittene Ausnahmebehandlung: ****)

catch(R <:Ok  thread :T(R)) :Ok
(* Wenn state(thread) == exceptionState, dann wird die identische Ausnahme,
   die thread terminierte im aktuellen Thread ausgelöst.
   Andernfalls geschieht nichts. *)

throw(R <:Ok  thread :T(R)  raiser() :Ok) :Ok
(* Wenn thread bereits terminiert ist: error.
   Wertet raiser im aktuellen Thread aus.
   Falls eine Ausnahme den dynamischen Kontext von raiser verläßt,
   wird sie in thread „umgeleitet“ und der aktuelle Thread bleibt unbeeinflußt. *)

setThrowHandler(R <:Ok  thread :T(R)  handler(self :T(R)  catch() :Ok) :Ok) :Ok
(* Setzt eine Routine (handler), die beim Empfangen eines throw durch thread dem Auslösen
   der betreffenden Ausnahme als asynchroner Handler vorgeschaltet ist.
   Dadurch ist u.a. Signalbehandlung in TL integriert.
   Voreingestellter Defaultwert: let handler(...) = catch(). *)


(**** Vorzeitige Beendigung: ****)

abort(R <:Ok  thread :T(R)) :Ok
(* throw(thread  fun() raise abortion. *)

kill(R <:Ok  thread :T(R)) :Ok
(* Beendet thread sofort und bedingungslos im Zustand exceptionState
   und mit dem Ausnahme-Ergebnis abortion. *)
```

```
(**** Anfragen: ****)

main() :T(Int)
(* Liefert den Thread, der durch das Laufzeitsystem erzeugt wurde,
   um die Boot-Funktion (siehe Modul "stdenv:main") auszuführen. *)

self() :T(Ok)
(* Liefert den Thread, der diese Funktion ausführt.
   Leider geht dabei Typinformation verloren: der Ergebnistyp wird auf Ok projiziert. *)

running() :Array(T(Ok))
(* Liefert einen Vektor mit allen gerade aktiven Threads.
   Ihre Ergebnistypen werden auf Ok projeziert. *)



(**** Unterstützung von Pseudo-Mobilität: ****)

log() :persistentLog.T
(* Liefert eine Log-Struktur zur Registrierung flüchtiger Objekte,
   die dann bei Migrationen per gate.migrateTo (Anhang D.2) pseudo-mobil sind.
   Siehe dazu PersistentLog (Anhang E.2). *)

end;
```

D.2 Thread-Migrationspforten

Diese knappe Schnittstelle genügt, um die Migration persistenter Threads zu organisieren. Sie ist in Abschnitt 6.2.2 näher erklärt.

```
interface Gate
import CSTypes server sid
export

  T(Data <:Ok) <:Ok
  (* Der abstrakte Typ aller Migrationspforten für Threads *)

  new(dispatcher :server.T  Dyn Attributes <:Ok  attributes() :Attributes
        search :CSTypes.SearchFun  Data <:Ok  data() :Data) :sid.T(T(Data))
  (* Für die Empfängerseite: Erzeugen einer neuen Pforte
     Die Parameter entsprechen denen der Funktion server.register (siehe Anhang A.2),
     denn eine Pforte ist ein RPC-Dienst.*)

  migrateTo(Data <:Ok  gate :T(Data)) :Data
  (* Für die Senderseite, bzw. den jeweiligen Thread selbst:
     Ausführung einer Migration zur Pforte gate *)

  here() :T(Ok)
  (* Liefert die lokale Instanz der ubiquitären Standardpforte.
     Diese kann nach Migrationen ggf. zur Rückkehr verwendet werden. *)

end;
```

Anhang E

Operations-Logging für flüchtige Objekte

In diesem Anhang sind die Schnittstellen zur Registrierung pseudo-persistenter und pseudo-mobiler Objekte aufgeführt.

E.1 Pseudo-Persistenz

Die Operationen dieser in Abschnitt 7.6 erwähnten Schnittstelle dienen der Organisierung der Pseudo-Persistent flüchtiger Objekte.

```
interface Persistent
export

  T(V <:Ok) <:Tuple volatile() :V end
  (* Ein persistenter Wert vom Typ persistent.T(V) kapselt einen flüchtigen Wert vom Typ V. *)


(* – Operations-Logging: *)

  create(V <:Ok  create() :V  save(:V) :Ok  destruct(:V) :Ok) :T(V)
  (* Erzeugt mittels create einen neuen persistenten Wert und
     fügt ihn in das globale Operations-Log ein. *)

  isCreated(V <:Ok  p :T(V)) :Bool
  (* Gibt an, ob p derzeit gültig ist. *)

  destruct(V <:Ok  p :T(V)) :Ok
  (* Destruiert p.volatile() und entferne p aus dem globalen Log. *)
```

```
(* – Log-Replay: *)

  createAllVolatiles() :Ok
  (* Rekonstruiert alle gekapselten flüchtigen Werte in der originalen Erzeugungsreihenfolge. *)

  saveAllVolatiles() :Ok
  (* Führt die save-Operation auf allen gekapselten flüchtigen Werte
     in der Erzeugungsreihenfolge aus. *)

  destructAllVolatiles() :Ok
  (* Destruiert alle gekapselten flüchtigen Werte in der umgekehrten Erzeugungsreihenfolge. *)


(* – Transaktionen: *)

  Via <:Ok

  viaCommit, viaRollback, viaRestart :Via

  commit() :Via
  (* Führt saveAllVolatiles aus und beendet und sichert dann die aktuelle flachen Transaktion. *)

  rollback() :Nok
  (* Führt destructAllVolatiles aus, setzt den Objektspeicher auf den Zustand
     vor dem letzten commit zurück und führt dann createAllVolatiles aus . *)


(* – Unterstützung von persistentLog: *)

  reCreate(V <:Ok  p :T(V)):Ok
  (* Erneuert einen destruierten persistenten Wertes und
     fügt ihn an das Ende des globalen Logs an. *)

  ...

end;
```

E.2 Pseudo-Mobilität

Die Operationen dieser in Abschnitt 7.6 erwähnten Schnittstelle dienen der Organisierung der Pseudo-Mobilität flüchtiger Objekte.

```
interface PersistentLog
import persistent
export

  T <:Ok
  (* Der Typ aller lokalen Operations-Logs. *)

  new():T
  (* Erzeugt ein leeres Log. *)

  insert(V <:Ok  log :T  p :persistent.T(V)) :Ok
  (* Fügt p in log ein. *)

  create(V <:Ok  log :T  create() :V  save(:V) :Ok destruct(:V) :Ok) :persistent.T(V)
  (* Tätigt insert(log  persistent.create(create save destruct)). *)

  destruct(V <:Ok  log :T  p :T(V)) :Ok
  (* Entfernt p aus log und tätigt persistent.destruct(p). *)

(* – Migrations-Unterstützung: *)

  reCreateAll(log :T) :Ok
  (* Führt persistent.reCreate für alle Log-Einträge in origionaler Erzeugungsreihenfolge aus. *)

  destructAll(log :T) :Ok
  (* Führt persistent.destruct für alle Log-Einträge in umgekehrter Erzeugungsreihenfolge aus. *)

  ...

end;
```

Anhang F

Performanz entfernter Funktionsaufrufe

Die Dauer entfernter Aufrufe im Tycoon-System setzt sich aus fixen und variablen Anteilen zusammen. Fixe Zeitspannen sind:

- die Ausführung der in TL geschriebenen Client- und Server-Stubs,
- der Mindestaufwand der darunterliegenden Socket-Schicht: Verbindungsaufbau, ein Minimum an Netzwerkkomunikation und Verbindungsabbau.

Variable Anteile sind:

- Marshalling,
- Unmarshalling,
- die Übertragung von Nutzdaten durch die Socket-Schicht.

Während isolierte Messungen einzelner Anteile, wie z.B. von Marshalling und Umarshalling, Aufschluß darüber geben, ob ein vermeidbarer Flaschenhals vorliegt, vermitteln entfernte Aufrufe einer entfernten Funktion folgender Form einen quantitativen Eindruck der Gesamtperformanz:

```
let f(X <:OK x :X) = ok
```

Diese Funktion wurde mit folgenden Argumenten entfernt aufgerufen:

```
(* Der minimale Wert: *)
ok
```

```
(* Zeichenketten der Länge n - u.a. große Objekte: *)
let a = string.new(n  ' ')

(* Arrays mit n nicht-identischen kleinen Objekten: *)
let b = begin
  let x = arrayOp.new(n )
  for i = 1 upto n do
    x[i] := string.new(1 ' ')
  end
  b
end

(* Listen mit n kleinen Elementen: *)
let c = list.create(iter.enum(1 n))

(* Objekte mit hohem Code-Anteil (bzw. gemischt großen Teilobjekten) - Standardmodule: *)
import print client;
```

Unter Verwendung IBM-kompatibler PCs mit 90 MHz Pentium-CPU und dem Betriebssystem Linux (1.2.13) sowie eines Ethernet-LANs (10 Mbit/s) ergaben sich folgende Meßwerte.

Argument	Objekte	Bytes	Marshall	Unmarshall	RPC lokal	RPC entfernt
ok	0	4	< 0,1	< 0,1	45	130
a (String)	1	48	0,70	0,10	45	130
a	1	100	0,71	0,12	45	130
a	1	1.000	0,81	0,27	47	130
a	1	10.000	1,2	1,9	56	140
a	1	100.000	3,6	17	170	250
a	1	1.000.000	36	190	1.200	1.600
b (Array)	10	250	0,91	0,29	53	130
b	100	2.200	2,90	1,7	58	130
b	1.000	22.000	28	17	120	210
b	10.000	220.000	300	170	730	1.100
b	100.000	2.200.000	3.600	1.700	7.500	13.000
c (Liste)	10	320	1,0	0,38	50	130
c	100	2.800	3,6	2,7	57	140
c	1.000	28.000	35	25	130	270
c	10.000	280.000	320	230	900	1.400
c	100.000	2.800.000	3.200	2.300	8.500	25.000
print	1.000	37.000	34	26	140	250
client	2.600	94.000	93	63	290	430

Alle Laufzeiten sind in Millisekunden angegeben. Sämtliche Zahlen in der Tabelle sind auf zwei signifikante Stellen gerundet.

Die gemessenen Zeiten belegen, daß die Geschwindigkeit des Marshalling und Unmarshalling sowie der Tycoon-Stubs für den Einsatz in einem Ethernet (10 Mbit/s) mehr als ausreichend ist. Aufgrund bisheriger Erfahrungen werden in typischen Tycoon-Anwendungen (unter Anwendung der Bindungsmechanismen aus Kapitel 7) meist Objektgraphen mit weniger als 5.000 Objekten sowie 150.000 Bytes bewegt. In diesem Bereich haben Marshalling und Unmarshalling einen Anteil von weniger als einem Drittel an der Gesamtdauer von RPCs, bei kleineren Objektgraphen ist es sogar weniger als ein Zehntel.

Über die Performanz bezüglich multimedialer Daten in BLOBs (*Binary Large Objects*) geben die Messungen von Wert a Auskunft. Da Marshalling und Unmarshalling in solchen Fällen hauptsächlich Speicherblöcke kopieren, sind sie um ein Mehrfaches performanter als in der getesteten Netzwerkkonfiguration erforderlich.

Die minimale Dauer eines lokalen Tycoon-RPC ist durch die Ergebnisse des Argumentes *ok* bestimmt. Sie beträgt bei einem RPC via Ethernet ca. 130 ms und ist damit um Faktor 65.000 länger als die eines minimalen lokalen[1] Funktionsaufrufes (ca. 2 μs). Hingegen erfordert ein RPC zwischen zwei Prozessen auf demselben Host nur 45 ms (Faktor 22.500). Daraus geht hervor, daß die durch die Stubs bedingte Verzögerung etwa zweimal kürzer ist als die Verzögerung des Netzwerkes. Letztere ist deshalb so lang, weil für jeden Aufruf eine eigene Socket-Verbindung neu aufgebaut wird. Hier könnte durch Caching von Sockets noch eine deutliche Verbesserung erzielt werden.

[1] Die lokale Performanz von Tycoon-Code ist je nach Algorithmus um Faktor 30 bis 100 geringer als die von *C*-Programmen. Dies ist für ein System mit virtueller Maschine akzeptabel. Vergleichsmessungen haben ergeben, daß sogar Java (keine Persistenz) nur etwa 3-mal performanter ist als Tycoon (Persistenz!).

Literaturverzeichnis

Abelson et al. 84: Abelson, H., Sussman, G.J., and Sussman, J. *Structure and Interpretation of Computer Programs.* The MIT Press, 1984.

Albano et al. 94: Albano, A., Brasini, C., Diotallevi, M., Ghelli, G., Orsini, R., and Rossi, R. „A Guided Tour of the Fibonacci System". FIDE Technical Report Series FIDE/94/103, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, Juli 1994.

Amaral et al. 92: Amaral, P., Jacquemont, C., Jensen, P., Lea, R., and Mirowski, A. „Transparent Object Migration in COOL-2". In: *ECOOP 92, Proceedings of the European Conference on Object Oriented Programming*, Juni 1992.

Armstrong et al. 93: Armstrong, J., Williams, M., and Virding, R. *Concurrent Programming in Erlang.* Prentice-Hall, 1993.

Artsy, Finkel 89: Artsy, Y. and Finkel, R. „Designing a Process Migration Facility". *IEEE Computer*, Seite 47–56, September 1989.

Atkinson et al. 81: Atkinson, M.P., Chisholm, K.J., and Cockshott, W.P. „PS-algol: An Algol with a Persistent Heap". *ACM SIGPLAN Notices*, 17(7), Juli 1981.

Atkinson et al. 83: Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., and Morrison, R. „An Approach to Persistent Programming". *Computer Journal*, 26(4), November 1983.

Atkinson et al. 96: Atkinson, M.P., Jordan, M.J., Daynès, L., and Spense, S. „Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system". (submitted to the 7th Internationl Workshop on Persistent Object Systems), Februar 1996.

Atkinson, Bunemann 87: Atkinson, M.P. and Bunemann, P. „Types and Persistence in Database Programming Languages". *ACM Computing Surveys*, 19(2), Juni 1987.

Atkinson, Morrison 95: Atkinson, M. and Morrison, R. „Orthogonally Persistent Object Systems". *VLDB Journal*, 4(3), 1995.

Atkinson 95: Atkinson, M.P. *Fully Integrated Data Environments.* Springer-Verlag, 1995.

Bach et al. 95: Bach, V., Brecht, L., and Österle, H. *Marktstudie: Softwaretools für das Business Process Redesign.* FBO-Verlag, Wiesbaden (Bertelsmann Fachinformation), 1995.

Bal et al. 89: Bal, H.E., Steiner, J.G., and Tanenbaum, A.S. „Programming Languages for Distributed Systems". *ACM Computing Surveys*, 21(3):261–322, September 1989.

Bancilhon et al. 92: Bancilhon, F., Delobel, C., and Kanellakis, P. *Building an Object-Oriented Database System: The Story of O_2.* Morgan Kaufmann Publishers, 1992.

Barak et al. 93: Barak, A., Guday, S., and Wheeler, R.G. *The MOSIX Distributed Operating System: Load Balancing for UNIX*, Band 672, *Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, NY, USA, 1993.

Bell, Grimson 92: Bell, D. and Grimson, J. *Distributed Database Systems.* Addison-Wesley Publishing Company, 1992.

Bernstein et al. 87: Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

Bharat, Cardelli 95: Bharat, K. and Cardelli, L. „Migratory Applications". In: *Proceedings of the ACM Symposium on User Interface Software and Technology '95*, Seite 133–142, 1995.

Birell et al. 93a: Birell, A., Evers, D., Nelson, G., Owicki, S., and Wobber, E. „Distributed Garbage Collection for Network Objects". Technical Report 116, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Dezember 1993.

Birell et al. 93b: Birell, A., Nelson, G., Owicki, S., and Wobber, E. „Network Objects". In: *14th ACM Symposium on Operating System Principles*, Seite 217–230, Juni 1993. Außerdem erschienen als TR Nr. 115, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Februar 1994.

Borgida et al. 93: Borgida, A., Mylopoulos, J., and Schmidt, J. „The TaxisDL Software Description Language". In: Jarke, M., Hrsg., *Database Application Engineering with DAIDA*, Seite 65–84. Springer-Verlag, 1993.

Brown et al. 92: Brown, A.L., Manietto, G., Matthes, F., Müller, R., and McNally, D.J. „An Open System Architecture for a Persistent Object Store". In: *Proceedings 25th Annual Hawaii International Conference on System Sciences*, Band 2, Seite 766–776, Januar 1992.

Brown, Morrison 91: Brown, A.L. and Morrison, R. „A Generic Persistent Object Store". PPRR 2-91, Universities of Glasgow and St Andrews, 1991.

Brown 89: Brown, A.L. „Persistent Object Stores". PPRR 71-89, Universities of Glasgow and St Andrews, März 1989.

Busse 96: Busse, S. „Modellbasiertes Prototyping objektorientierter, heterogener, verteilter Informationssysteme auf der Basis des Tycoon-Systems". Diplomarbeit, Technische Universität Berlin, Fachbereich Informatik, Mai 1996.

Canning et al. 89: Canning, P.S., Cook, W.R., Hill, W.L., and Olthoff, W. „F-Bounded Polymorphism for Object-Oriented Programming". In: *Proceedings of the Conference on Functional Proramming Languages and Computer Architecture, Imperial College, London*, Seite 273–280, September 1989.

Cardelli et al. 88: Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. „Modula-3 Report". Technical Report ORC-1, Olivetti Research Center, 2882 Sand Hill Road, Menlo Park, California, 1988.

Cardelli et al. 94: Cardelli, L., Matthes, F., and Abadi, M. „Extensible Grammars for Language Specialization". In: Beeri, C., Ohori, A., and Shasha, D.E., Hrsg., *Proceedings of the Fourth International Workshop on Database Programming Languages, Manhatten, New York*, Workshops in Computing, Seite 11–31. Springer-Verlag, Februar 1994.

Cardelli 86: Cardelli, L. „Amber". In: *Combinators and Functional Programming Languages*, Band 242, *Lecture Notes in Computer Science.* Springer-Verlag, 1986.

Cardelli 89: Cardelli, L. „Typeful Programming". Technical Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Mai 1989.

Cardelli 94: Cardelli, L. „Obliq: A Language with Distributed Scope". Technical report, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Juni 1994.

Catell 94: Catell, R.G.G., Hrsg. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1994.

Cattell 94: Cattell, R.G.G., Hrsg. *Object Data Management.* Addison-Wesley Publishing Company, second edition, 1994.

Caughey et al. 93: Caughey, S. J., Parrington, G. D., and Shrivastava, S. K. „Shadows - A Flexible Support System for Objects in Distributed Systems". In: *Proceedings of the 3rd International Workshop on Object Orientation and Operating Systems*, Seite 73–82, Asheville, NC (USA), Dezember 1993.

Caughey, Shrivastava 95: Caughey, S. J. and Shrivastava, S. K. „Architectural Support for Mobile Objects in Large Scale Distributed Systems". In: *IWOOOS-95*, Lund, August 1995.

Ceri, Pelagatti 85: Ceri, S. and Pelagatti, G. *Distributed Databases: Principles and Systems.* Science Series. McGraw-Hill, 1985.

Connor 88: Connor, R. „The Napier Type-Checking Module". PPRR 58-88, Universities of Glasgow and St Andrews, März 1988.

Corbin 91: Corbin, J.R. *The Art of Distributed Applications.* Sun Technical Reference Library. Springer-Verlag, 1991.

Courtney 95: Courtney, A. „Phantom: An Interpreted Language for Distributed Computing". In: *USENIX Conference on Object-Oriented Technologies (COOTS), Monterey, CA*, Juni 1995.

Curry 92: Curry, D. *UNIX System Security.* Addison-Wesley Publishing Company, 1992.

Dearle et al. 91: Dearle, A., Rosenberg, J., and Vaughan, F. „A Remote Execution Mechanism for Distributed Homogeneous Stable Stores". In: *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.

Dearle et al. 94: Dearle, A., Bona, R. di, Farrow, J., Henskens, F., Lindström, A., and Rosenberg, J. „Grasshopper: An Orthogonally Persistent Operating System". *Computer Systems*, 7(3):289–312, 1994.

Douglis, Ousterhout 87: Douglis, F. and Ousterhout, J. „Process Migration in the Sprite Operating System". In: *Proceeddings of the 7th International Conference on Distributed Computing Systems*, Seite 18–25, 1987.

Douglis, Ousterhout 91: Douglis, F. and Ousterhout, J. „Transparent Process Migration: Design Alternatives and the Sprite Implementation". *Software Practice and Experience*, 21(8):757–785, August 1991.

Garcia-Molina, Salem 87: Garcia-Molina, H. and Salem, K. „Sagas". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, Seite 249–259, Mai 1987.

Gawecki, Matthes 94: Gawecki, A. and Matthes, F. „The Tycoon Machine Language TML: An Optimizable Persistent Program Representation". FIDE Technical Report FIDE/94/100, Fachbereich Informatik, Universität Hamburg, August 1994.

Gawecki, Matthes 95a: Gawecki, A. and Matthes, F. „Integrating Query and Program Optimization Using Persistent CPS Representations". In: Atkinson, M.P., Hrsg., *Fully Integrated Data Environments.* Springer-Verlag, 1995.

Gawecki, Matthes 95b: Gawecki, A. and Matthes, F. „TooL: A Persistent Language Integrating Subtyping, Matching and Type Quantification". FIDE Technical Report Series FIDE/95/135, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1995.

Gawecki, Matthes 96: Gawecki, A. and Matthes, F. „Exploiting Persistent Intermediate Code Representations in Open Database Environments". In: *Proceedings of the 5th EDBT Conference*, Avignon, France, März 1996.

Geisler 95: Geisler, A. „Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Juni 1995.

General Magic 95a: „General Magic's Telescript home page". http://www.genmagic.com/Telescript/, 1995.

General Magic 95b: „Telescript Programming Guide, Version 0.5 (ALPHA)". http://www.genmagic.com/Telescript/TDE/TDEDOCS_HTML/developer.html, Oktober 1995.

Göllnitz 96: Göllnitz, M. „Polymorphe persistente Client/Server-Programmierung mit dynamischer hierarchischer Adreßauflösung". Studienarbeit, Fachbereich Informatik, Universität Hamburg, April 1996.

Golsing, McGilton 95: Golsing, J. and McGilton, H. „The Java Language Environment – A Whitepaper". Technical report, Sun Microsystems, Oktober 1995.

Gray, Reuter 93: Gray, J. and Reuter, A. *Transaction Processing – Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 1993.

Gray 78: Gray, J. „Notes on Database Operating Systems". In: *Operating Systems – An Advanced Course*, Band 60, *Lecture Notes in Computer Science.* Springer-Verlag, 1978.

Gray 95: Gray, R.S. „Agent Tcl: A Transportable Agent System". In: Mayfield, J. and Finin, T., Hrsg., *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, Dezember 1995.

Gruber et al. 92: Gruber, O., Amsaleg, L., Daynès, L., and Valduriez, P. „Eos, an Envrionment for Object-Based Systems". In: Rosenberg, J., Hrsg., *Proceedings of the 25th Hawaii Internationel Conference on System Sciences*, Band I, Seite 757–768, 1992.

Gruber 92: Gruber, O. *Eos, an Environment for Persistent and Distributed Applications over a Shared Object Space.* PhD thesis, Université Pierre et Marie Curie, Paris IV, Dezember 1992.

Hammer, Champy 94: Hammer, M. and Champy, J. *Business Reengineering: Die Radikalkur für das Unternehmen.* Campus Verlag, Frankfurt, 2nd edition, 1994.

Harland 84: Harland, D.M. *Polymorphic Programming Languages, Design and Implementation.* Ellis Horwood Limited, a division of John Wiley & Sons, 1984.

Horning et al. 93: Horning, J., Kalsow, P., J. McJones, and Nelson, G. „Some Useful Modula-3 Interfaces". Technical report, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, Dezember 1993.

Hull, Su 89: Hull, R. and Su, J. „On Bulk Data Type Constructors and Manipulation Primitives: A Framework for Analyzing Expressive Power and Complexity". In: *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, Seite 396–410, Juni 1989.

Hutchinson, Jeffery 89: Hutchinson, N. and Jeffery, C.L. „An Efficient Implementation of Distributed Object Persistence". Internes Emerald-Dokument, ftp://ftp.diku.dk/pub/diku/dists/emerald/papers.tar.gz, Juli 1989.

Hutchinson 87a: Hutchinson, N.C. „Emerald: A Language To Support Distributed Programming". In: Barbacci, M.R., Hrsg., *Proceedings of the 2nd Workshop on Large-Grained Parallelism*, Seite 45–47, November 1987.

Hutchinson 87b: Hutchinson, N.C. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, University of Washington, September 1987.

ISO-ASN 90: ISO / IEC JTC 1 / SC21 / DIS 8824. *Abstract Syntax Notation One (ASN.1), Draft International Standard*, 1990.

ISO-CCR 90: ISO / IEC JTC1 / SC21 / IS 9805. *Information Technology - Open Systems Interconnection - Protocol Specification for the Commitment, Concurrency and Recovery Service Element*, 1990.

ISO-EXP 92: ISO. *Standard ISO 10303, Part1: Industrial Automation Systems – Product Data Representation and Exchange, Overview and Fundamental Principles*, 1992.

ISO-ODP 95: ISO / IEC JTC 1 / SC 21 / DIS 13235. *ODP Trading Function*, Juni 1995.

ISO-RPC 91: ISO / IEC CD 11578-1 / SC 21 / N 6561. *ISO Remote Procedure Call Specification*, November 1991.

ISO-SQL 89: ISO-SQL. *Database Language SQL.* ISO / IEC / JTC 1 / SC 21 / IS 9075, 1989.

Jablonski 95: Jablonski, S. „Workflow-Management-Systeme: Motivation, Modellierung, Architektur". *Informatik Spektrum*, 18(1):13–24, 1995.

Johannisson 95: Johannisson, Nico. „Generische Programmierung typsicherer Kommunikationsdienste". Master's thesis, Fachbereich Informatik, Universität Hamburg, Mai 1995.

Jul et al. 88: Jul, E., Levy, H., Hutchinson, N.C., and Black, A. „Fine-Grained Mobility in the Emerald System". *ACM Transactions on Computer Systems*, 6(1):109–133, Februar 1988.

Jul 88: Jul, E. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, 1988.

Jul 89: Jul, E. „Migration of Light-weight Processes in Emerald". *Operation Systems Technical Comittee Newsletter*, 3(1):25–30, 1989.

Juul 93: Juul, N.C. *Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System Emerald.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1993. Published as Technical Report DIKU-rapport 93/1.

Keppel 93: Keppel, D. „Tools and Techniques for Building Fast Portable Threads Packages". Technical Report 93-05-06, University of Washington, Department of Computer Science and Engineering, University of Washington, Seattle, Mai 1993.

Kim et al. 89: Kim, W., Kim, K., and Dale, A. „Storage Management for Objects in EXODUS". In: Kim, W. and Lochovsky, F.H., Hrsg., *Object-Oriented Concepts, Databases, and Applications*, Frontier Series. ACM Press, 1989.

Kirby 93: Kirby, G.N.C. *Reflection and Hyper-Programming in Persistent Programming Systems.* PhD thesis, University of St. Andrews, 1993.

Knabe 95: Knabe, F. *Language Support for Mobile Agents.* PhD thesis, School of Computer Science, Carnegie-Mellon University, Dezember 1995.

Koch et al. 83: Koch, J., Mall, M., Putfarken, P., Reimer, M., Schmidt, J.W., and Zehnder, C.A. „Modula/R Report, Lilith Version". Technical report, Department Informatik, ETH Zürich, Switzerland, Februar 1983.

Koch et al. 90: Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakearley, R., and Barter, C. „Cache Coherence and Storage Management in a Persistent Object System". In: Dearle, A., Shaw, G.M., and Zdonik, S.B., Hrsg., *Implementing Persistent Object Bases, Principles and Practice.* Morgan Kaufmann Publishers, 1990.

Koehler 96: Koehler, H. „Generische Daten- und Funktionsvisualisierung in einer persistenten Programmierumgebung". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, März 1996.

Kornacker 95: Kornacker, M. „Persistente Sicherungspunkte für langlebige Aktivitäten in offenen Umgebungen". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, August 1995.

Kosovic 95: Kosovic, Douglas. „Trader Information Service“. http://www.dstc.edu.au/AU/research_news/odp/trader/trader.htm, November 1995.

Kutsche et al. 95: Kutsche, R.D., Schöning, C., and Waßerroth, S. „Informationsmodellierung im Rahmen eines Umweltinformationssystems“. Forschungsbericht Nr. 95-17, Technische Universität Berlin, Fachbereich Informatik, April 1995.

Lamersdorf, Schmidt 80: Lamersdorf, W. and Schmidt, J.W. „Specification of Pascal/R“. Bericht 73, 74, Fachbereich Informatik, Universität Hamburg, Juli 1980.

Lampson 81: Lampson, B. W. „Atomic Transactions“. In: *Distributed Systems–Architecture and Implementation*, Band 105, *Lecture Notes in Computer Science*, Seite 246–265. Springer-Verlag, New York, 1981.

Larsen 92: Larsen, N.E. „An Object-Oriented Database in Emerald“. Master's thesis, DIKU, University of Copenhagen, Department of Computer Science, Juli 1992.

Lea et al. 93: Lea, R., Jacquemont, C., and Pillevesse, E. „COOL: System Support for Distributed Object-Oriented Programming“. *Communications of the ACM*, 36(9):37–46, September 1993.

Levy, Tempero 91: Levy, H. M. and Tempero, E. D. „Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation“. *Software Practice and Experience*, 21(1), Januar 1991.

Leymann, Altenhuber 94: Leymann, F. and Altenhuber, W. „Managing Business Processes as an Information Resource“. *IBM Systems Journal*, 33(2), 1994.

Lindsay et al. 79: Lindsay, B., Selinger, P., Galtieri, C., Gray, J., Lorie, R., Price, T., Putzolo, F., Traiger, I., and Wade, B. „Notes on Distributed Databases“. Technical Report TR RJ2571, IBM Almaden Research Laboratories, San Jose, CA, 1979.

Liskov et al. 90: Liskov, B., Johnson, P., Gruber, R., and Shrira, L. „A Highly Available Object Repository for Uses in a Heterogeneous Distributed System“. In: Dearle, A., Shaw, G.M., and Zdonik, S.B., Hrsg., *Implementing Persistent Object Bases, Principles and Practice*, Seite 255–266. Morgan Kaufmann Publishers, 1990.

Liskov et al. 92: Liskov, B., Day, M., and Shrira, L. „Distributed Object Management in Thor“. In: *Proceedings of the International Workshop on Distributed Object Management, Edmonton, Canada*, 1992.

Liskov, Shrira 88: Liskov, B. and Shrira, L. „Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems“. *ACM SIGPLAN Notices*, 23(7):260–267, Juli 1988.

Litzkow, Solomon 92: Litzkow, M. and Solomon, M. „Supporting Checkpointing and Process Migration outside the UNIX Kernel“. In: *In USENIX Winter Conference, San Francisco*, Seite 283–290, Januar 1992.

Manola, Heiler 93: Manola, F. and Heiler, S. „A “RISC” Object Model for Object System Interoperation: Concepts and Applications“. Technical Report TR-0231-08-93-165, GTE laboratories Inc., Waltham, MA (USA), August 1993.

Mathiske et al. 93: Mathiske, B., Matthes, F., and Müßig, S. „The Tycoon System and Library Manual“. DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Dezember 1993.

Mathiske et al. 95a: Mathiske, B., Matthes, F., and Schmidt, J.W. „On Migrating Threads“. In: *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, Juni 1995. (Außerdem erschienen als TR FIDE/95/136).

Mathiske et al. 95b: Mathiske, B., Matthes, F., and Schmidt, J.W. „Scaling Database Languages to Higher-Order Distributed Programming“. In: *Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy.* Springer-Verlag, September 1995. (Außerdem erschienen als TR FIDE/95/137).

Mathiske 92: Mathiske, B. „Kodegenerierung für Programmiersprachen mit Persistenz, Polymorphie und Funktionen höherer Ordnung". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Dezember 1992.

Matthes et al. 92a: Matthes, F., Müller, R., and Schmidt, J.W. „Object Stores as Servers in Persistent Programming Environments – The P-Quest Experience". FIDE Technical Report Series FIDE/92/48, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, Juli 1992.

Matthes et al. 92b: Matthes, F., Rudloff, A., Schmidt, J.W., and Subieta, K. „The Database Programming Language DBPL: User and System Manual". FIDE Technical Report Series FIDE/92/47, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, Juli 1992.

Matthes et al. 94: Matthes, F., Müßig, S., and Schmidt, J.W. „Persistent Polymorphic Programming in Tycoon: An Introduction". FIDE Technical Report FIDE/94/106, Fachbereich Informatik, Universität Hamburg, August 1994.

Matthes et al. 95a: Matthes, F., Müller, R., and Schmidt, J.W. „Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocol". In: Atkinson, M.P., Hrsg., *Fully Integrated Data Environments.* Springer-Verlag, 1995.

Matthes et al. 95b: Matthes, F., Schröder, G., and Schmidt, J.W. „Tycoon: A Scalable and Interoperable Persistent System Environment". In: Atkinson, M.P., Hrsg., *Fully Integrated Data Environments.* Springer-Verlag, 1995.

Matthes, Schmidt 89: Matthes, F. and Schmidt, J.W. „The Type System of DBPL". In: *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, Seite 255–260, Juni 1989.

Matthes, Schmidt 91: Matthes, F. and Schmidt, J.W. „Bulk Types: Built-In or Add-On?". In: *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.

Matthes, Schmidt 92: Matthes, F. and Schmidt, J.W. „Definition of the Tycoon Language TL – A Preliminary Report". Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, November 1992.

Matthes, Schmidt 93: Matthes, F. and Schmidt, J.W. „System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways". In: Spies, P.P., Hrsg., *Proceedings of Euro-Arch'93 Congress*, Seite 301–317. Springer-Verlag, Oktober 1993.

Matthes, Schmidt 94: Matthes, F. and Schmidt, J.W. „Persistent Threads". In: *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, Seite 403–414, Santiago, Chile, September 1994.

Matthes 93: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993.

Mc Cready, Palermo 94: Mc Cready, S.C. and Palermo, A.M. „Lotus Notes: Agent of Change, the Financial Impact of Lotus Notes on Business". IDC Special Report, International Data Corporation, Five Speen Street, Framingham, MA 01701 USA, Tel. 508/872-8200, 1994. (Erhältlich direkt von Lotus).

Meyer 86: Meyer, B. „Genericity versus Inheritance". In: *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Portland, Oregon*, Seite 391–405, Oktober 1986.

Milner et al. 90: Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

Milner 78: Milner, R. „A Theory of Type Polymorphism in Programming“. *Journal of Computer and System Sciences*, 17:348–375, 1978.

Milojicic 93: Milojicic, D.S. et al. „Task Migration on top of the Mach Microkernel“. In: *In 3rd USENIX Mach Symposium, Santa Fe*, Seite 273–289, April 1993.

Mira da Silva 95a: Mira da Silva, M. „Automating Type-safe RPC“. In: Bukhres, O., Öszu, T., and Shan, M.C., Hrsg., *Proceedings of the 5th International Workshop on Research Issues on Data Engineering: Distributed Object Management, Taipeh, Taiwan*, IEEE Computer Society Press, Seite 100–107, März 1995.

Mira da Silva 95b: Mira da Silva, M. „Programmer's Manual to Napier88/RPC 2.2“. FIDE Technical Report FIDE/95/133, ESPRIT Basic Research Action 6309, $FIDE_2$, 1995.

Mira da Silva 96: Mira da Silva, M. „RPC-Performanz in Napier88“. Persönliche Kommunikation, April 1996.

MOBILE 95: „Home Page des Projektes MOBILE: MOdel Base for an Integrative View of Logistics and Environment“.
http://www.informatik.uni-hamburg.de/ASI/MOBILE_home.html, 1995.

Morrison et al. 94: Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.J., Dearle, A., Kirby, G.N.C., and Munro, D.S. „The Napier88 Reference Manual (Release 2.0)“. FIDE Technical Report Series FIDE/94/104, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1994.

Moss, Sinofsky 88: Moss, J.E.B. and Sinofsky, S. „Managing Persistent Data with Mneme: Designing a Reliable Shared Object Interface“. In: Dittrich, K.R., Hrsg., *Advances in Object-Oriented Database Systems*, number 334 in Lecture Notes in Computer Science. Springer-Verlag, September 1988.

Müller et al. 94: Müller, K., Merz, M., and Lamersdorf, W. „Der TRADE-Trader: Ein Basisdienst offener verteilter Systeme“. In: Popien, C. and Meyer, B., Hrsg., *Neue Konzepte für die Offene Verteilte Verarbeitung*, Band 7, *Aachener Beiträge zur Informatik*, Seite 35–44. Verlag der Augustinus Buchhandlung, September 1994.

Müller 91: Müller, R. „Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung“. Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, November 1991.

Munro 93: Munro, D.S. *On the Integration of Concurrency, Distribution and Persistence.* PhD thesis, Department of Mathematical and Computational Sciences, University of St. Andrews, Scotland, 1993.

Neeracher 95: Neeracher, M. *Grand Unified Socket Interface - User's Manual, Version 1.55*
ftp://ftp.switch.ch/software/mac/src/mw_c/, April 1995.

Nelson, Birrell 84: Nelson, B. J. and Birrell, A. D. „Implementing Remote Procedure Calls“. *ACM Transactions on Computer Systems*, 2(1), Februar 1984.

Nelson 81: Nelson, B.J. *Remote Procedure Call.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1981.

Nelson 91: Nelson, G., Hrsg. *Systems programming with Modula-3.* Series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Netscape 96: „Persistent Client State - HTTP Cookies - Preliminary Specification, Netscape Communications Corporation“. http://home.netscape.com/newsref/std/cookie_spec.html, 1996.

Nicol et al. 93: Nicol, J., Wilkes, T., and Manola, F. „Object Orientation in Heterogeneous Distributed Computing Systems“. *Special Issue on Heterogeneous Processing*, Juni 1993.

Niederée 92: Niederée, C. „Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung". Master's thesis, Fachbereich Informatik, Universität Hamburg, November 1992.

OMG 91: OMG. „The Common Object Request Broker: Architecture and Specification". Document 91.12.1, Rev. 1.1, Object Management Group, Dezember 1991.

Open Software Foundation 93: Open Software Foundation. *OSF DCE Application Development Guide.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

Otte et al. 96: Otte, R., Patrick, P., and Roy, M. *Understanding CORBA: the Common Object Request Broker Architecture.* Prentice Hall, Englewood Cliffs, New Jersey, 1996.

Özsu, Valduriez 91: Özsu, M.T. and Valduriez, P. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Peyton Jones 87: Peyton Jones, S. L. *The Implementation of Functional Programming Languages.* Prentice Hall, Englewood Cliffs, New Jersey, 1987.

Piellusch 96: Piellusch, A. „Synchronisation langlebiger Aktivitäten". Diplomarbeit in Vorbereitung, Fachbereich Informatik, Universität Hamburg, 1996.

Pietrzyk, Radic 94: Pietrzyk, D. and Radic, M. *ODBC in der Praxis.* IT-Verlag, Höhenkirchen, 1994.

POSIX 90: National Bureau of Standards, NBS-FIPS-PUB-151-1. *Portable Operating System Interface for Computer Environments (POSIX)*, 1990.

Reinhardt 94: Reinhardt, A. „The Network with Smarts: Intelligent Networks from AT&T and IBM could dramatically change the Way you work and may set the Model for a Future of Mobile Software Agents". *Byte Magazine*, 19(10), Oktober 1994.

Reuter 89: Reuter, A. „ConTracts: A Means for Extending Control Beyond Transaction Boundaries". In: *Third International Workshop on High Performance Transaction Systems*, 1989.

Rosenberg et al. 90: Rosenberg, John, Henskens, Frans, Brown, Fred, Morrison, Ron, and Munro, David. „Stability in a Persistent Store Based on a Large Virtual Memory". In: Rosenberg, J. and Keedy, J.L., Hrsg., *Security and Persistence*, Seite 229–245. Springer-Verlag, 1990.

Rozier 92: Rozier, M. et al. „Overview of the Chorus Distributed Operating System". In: *In USENIX Workshop Proceedings - Microkernels and Other Kernel Architectures*, Seite 39–69, April 1992.

Rudloff et al. 95: Rudloff, A., Matthes, F., and Schmidt, J.W. „Security as an Add-On Quality in Persistent Object Systems". In: *Second International East/West Database Workshop, Klagenfurt, Austria*, Workshops in Computing, Seite 90–108. Springer-Verlag, 1995. (Außerdem erschienen als TR FIDE/95/138).

Rudloff 96: Rudloff, A. *Sicherheitsaspekte in persistenten Programmierumgebungen.* Dissertation in Vorbereitung, siehe dazu [Rudloff et al. 95], Fachbereich Informatik, Universität Hamburg, 1996.

Sangiorgi 93: Sangiorgi, D. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, Februar 1993.

Santifaller 93: Santifaller, M. *TCP/IP und ONC/NFS in Theorie und Praxis.* Addison-Wesley Publishing Company, 1993. 2. Auflage.

Schill, Mock 93: Schill, A. and Mock, M.U. „DC++: Distributed Object-oriented System Support On Top of OSF DCE". *Distributed Systems Engineering*, 1:112–125, 1993.

Schill 92: Schill, A. „Remote Procedure Call: Fortgeschrittene Konzepte und Systeme - Ein Überblick". *Informatik Spektrum*, 15, 1992.

Schill 93: Schill, A. *DCE: Das OSF Distributed Computing Environment, Einführung und Grundlagen.* Springer-Verlag, 1993.

Schmidt et al. 93: Schmidt, J.W., Matthes, F., and Valduriez, P. „Building Persistent Application Systems in Fully Integrated Data Environments: Modularization, Abstraction and Interoperability". In: *Proceedings of Euro-Arch'93 Congress*, Seite 270–287. Springer-Verlag, Oktober 1993.

Schmidt, Matthes 92: Schmidt, J.W. and Matthes, F. „The Database Programming Language DBPL: Rationale and Report". FIDE Technical Report Series FIDE/92/46, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, Juli 1992.

Schmidt, Matthes 95: Schmidt, J.W. and Matthes, F. „Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems". In: Atkinson, M.P., Hrsg., *Fully Integrated Data Environments*. Springer-Verlag, 1995.

Schmidt 77: Schmidt, J.W. „Some High Level Language Constructs for Data of Type Relation". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977.

Shapiro et al. 89: Shapiro, M., Gautron, P., and Mosseri, L. „Persistence and Migration for C++ Objects". In: *Proceedings of the European Conference on Object Oriented Programming, Nottingham, GB*, Juli 1989.

Shapiro, Ferreira 93: Shapiro, M. and Ferreira, P. „Distribution and Persistence in Multiple and Heterogeneous Address Spaces". In: *Proceedings of the 3rd IEEE International Workshop on Object-Orientation in Operating Systems, Asheville, NC (USA)*, Seite 83–92, Dezember 1993.

Shapiro 86: Shapiro, M. „Structure and Encapsulation in Distributed Systems: The Proxy Principle". In: *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, Seite 198–205, Cambridge, MA USA, 1986. IEEE Computer Society , Washington, DC.

Shapiro 91: Shapiro, M. „Soul: an Object-Oriented Frameword for Object Support". In: *Workshop on Operating Systems for the Nineties and Beyond*. Springer-Verlag, Juli 1991.

Shapiro 93: Shapiro, M. „Flexible Bindings for Fine-Grain and Fragmented Objects in Distributed Systems". Rapport de Recherche 2007, INRIA, Domaine de Voluceau, Rocquencourt 78153 Le Chesnay Cedex, France, August 1993.

Sloman, Kramer 88: Sloman, M. and Kramer, J. *Verteilte Systeme und Rechnernetze*. Coedition Carl Hanser Verlag, München, Wien und Prentice-Hall International Inc., London, 1988. Übersetzung Kurt Ackermann und Inge Haas-Ackermann.

Stamos, Gifford 90: Stamos, J.W. and Gifford, D.K. „Remote Evaluation". *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, 1990.

Steele 86: Steele, G.L. Jr. „The Revised^3 Report on the Algorithmic Language Scheme". *ACM SIGPLAN Notices*, 21(12):37–79, Dezember 1986.

Steensgaard, Jul 95: Steensgaard, B. and Jul, E. „Object and Native Code Thread Mobility Among Heterogeneous Computers". In: *15th ACM Symposium on Operating System Principles (SOSP), Copper Mountain Resort, Colorado*, Dezember 1995.

Steiner et al. 88: Steiner, J.G., Neumann, B.C., and Schiller, J.I. „Kerberos: An Authentication Service for Open Network Systems". In: *Proceedings of the Winter 1988 Usenix Conference*, Februar 1988.

Strom et al. 91: Strom, R.E., Bacon, D.F., Goldberg, A.P., Lowry, A., Yellin, D.M., and Yemini, S.A. *Hermes: A Language for Distributed Computing*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Sun 88: Sun Microsystems. *Sun OS Reference Manual*, Mai 1988.

Sun 95a: „Java: The Inside Story".
http://www.sun.com/sunworldonline/swol-07-1995/swol-07-java.html, 1995.

Sun 95b: Sun Microsystems. *The Java Language Specification*, 1.0 beta edition, Oktober 1995.

Sun 95c: Sun Microsystems. *The Java Virtual Machine Specification*, 1.0 beta edition, August 1995.

Swenson 93: Swenson, K.D. „Visual Support for Reengineering Work Processes". In: *Proceedings of the Conference on Organizational Computing Systems, COOCS'93*. ACM Press, 1993.

Tanenbaum et al. 90: Tanenbaum, A.S., Renesse, R. van, Staveren, H. van, Sharp, G.J., Mullender, S.J., J., Jansen, and Rossum, G. van. „Experiences with the Amoeba Distributed Operating System". *Communications of the ACM*, 33(12):46–63, Dezember 1990.

Tanenbaum, van Renesse 88: Tanenbaum, A.S. and Renesse, R. van. „A Critique of the Remote Procedure Call Paradigm". In: R. Speth, Hrsg., *Research into Networks and Distributed Applications*, Seite 775–783. Elsevier Science Publisher B.V., 1988.

Thomsen et al. 95: Thomsen, B., Leth, L., Knabe, F., and Chevalier, P.Y. „Mobile Agents". Technical Report ECRC-95-21, European Computer-Industry Research Centre, München, Juni 1995.

Tietz 89: Tietz, Walter, Hrsg. *Verzeichnis Systeme, Serien X.500*. CCITT-Empfehlungen der V-Serie und der X-Serie. R. V. Decker's Verlag, G Schenck, Heidelberg, April 1989.

Tschudin 94: Tschudin, C.F. „An Introduction to the MØ Messenger Language". Cahier du Centre Universitaire d'Informatique No 86, Universität Genf, Schweiz, September 1994.

Tschudin 95: Tschudin, C.F. „Protokollimplementierung mit Kommunikationsboten". In: *Proceedings of the KiVS '95 Conference*, Chemnitz, Februar 1995.

Turner 93: Turner, K., Hrsg. *Using Formal Description Techniques, An Introduction to Estelle, Lotos and SDL*. Wiley series in communication and distributed systems. John Wiley & Sons, 1993.

Tycoon 92: „WWW Home Page for the Tycoon Project". http://idom-www.informatik.uni-hamburg.de/Tycoon/entry.html, 1992.

Vaziri Pour 96: Vaziri Pour, N. „Flexible Bindungstechniken für ubiquitäre Ressourcen in verteilten Anwendungen". Studienarbeit, Fachbereich Informatik, Universität Hamburg, März 1996.

Wächter, Reuter 91: Wächter, H. and Reuter, A. „The ConTract Model". In: Elmagarmid, A.K., Hrsg., *Database Transaction Models for Advanced Applications*, Seite 219–263. Morgan Kaufmann Publishers, 1991.

Wai 88: Wai, F. *Distributed Concurrent Persistent Programming Languages: An Experimental Design and Implementation*. PhD thesis, University of Glasgow, April 1988.

Wai 89: Wai, F. „Distributed PS-algol". In: Rosenber, R. and Koch, D., Hrsg., *Proceedings of the 3rd International Workshop on Persistent Object Store Systems, Newcastle, NSW*, Seite 126–140. Springer-Verlag, Januar 1989.

Wayner 94: Wayner, P. „Agents Away". *BYTE*, Seite 113–118, Mai 1994.

Weiss et al. 96a: Weiss, M., Johnson, A., and Kiniry, J. „Distributed Computing: Java, CORBA and DCE". Technical report, Open Software Foundation Research Institute, Februar 1996.

Weiss et al. 96b: Weiss, M., Johnson, A., and Kiniry, J. „Overview of Java and HotJava". Technical report, Open Software Foundation Research Institute, Februar 1996.

WFMC 95: „WWW Home Page of the Workflow Management Coalition". http://www.www.aiai.ed.ac.uk/WFMC/, 1995.

White 94: White, J.E. „Telescript Technology: The Foundation for the Electronic Marketplace". White paper, General Magic Inc., Mountain View, California, USA, 1994.

Wikström 86: Wikström, C. „Distributed Programming in Erlang". In: *Proceedings of the 1st International Symposium on Parallel Symbolic Computation*, Linz, Österreich, September 1986.

Willomat 96: Willomat, A. „Tycoon Mail Tooolkit“. Diplomarbeit in Vorbereitung, Informationen unter: hhtp://idom-www.informatik.uni-hamburg-.de/tycoon/secure/mail.html, Fachbereich Informatik, Universität Hamburg, 1996.

Wooldridge, Jennings 95: Wooldridge, M.J. and Jennings, N.R., Hrsg. *Agent Theories, Architectures and Languages: A Survey.* Intelligent Agents. Springer-Verlag, 1995.

Yellin 95: Yellin, Frank. „Low Level Security in Java“. In: *4th International Conference on the World-Wide Web*, Massachusetts Institute of Technology, Boston, Dezember 1995.

Zayas 87: Zayas, E.R. *The Use of Copy-On-Reference in a Process Migration system.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, April 1987.

Hiermit versichere ich, die vorliegende Dissertation ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel selbständig angefertigt zu haben.

Hamburg, den 31. Mai 1996

Bernd J. W. Mathiske

Lebenslauf

Bernd Joachim Wolfgang Mathiske

Geburtsdatum und Ort	18.06.1964 in Berlin
Anschrift	Bernd Mathiske Matthias-Claudius-Weg 2 21244 Buchholz
Familienstand	ledig
10.05.1983	Abitur am Albert-Einstein-Gymnasium in Buchholz in der Nordheide
01.07.1983 – 30.09.1984	Wehrdienst in Lüneburg
01.10.1984	Immatrikulation zum Studium der Informatik mit Nebenfach Mathematik an der Universität Hamburg
22.07.1985 – 04.10.1985	Werkstudent bei der ESSO AG, Raffinerie Hamburg
11.12.1986	Vordiplom in Informatik
01.01.1987 – 31.12.1993	Freiberufliche EDV-Beratung und Software-Entwicklung
19.02.1993	Diplom in Informatik
01.03.1993 – 31.03.1996	Wissenschaftlicher Mitarbeiter der Universität Hamburg im ESPRIT-Grundlagenforschungsprojekt „FIDE2“ und Promotionsstudium im Fachbereich Informatik der Universität Hamburg

Hamburg, den 31.05.1996