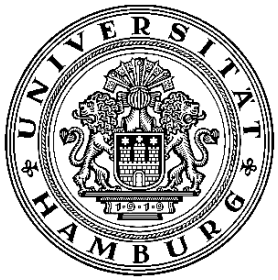
Universität Hamburg
Fachbereich Informatik
Datenbanken und Informationssysteme



STUDIENARBEIT

Polymorphe, Persistente Client/Server-Programmierung mit dynamischer, hierarchischer Adreßauflösung

April 1996

Martin Göllnitz

Betreuer:
Prof. Dr. Joachim W. Schmidt

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Ganz allgemein besteht Netzverkehr aus Datenströmen zwischen Kommunikationspartnern. Geht man davon aus, daß ein Dienst zu erbringen ist, enthalten solche Datenströme Anfragen und Antworten darauf, beziehungsweise Bestätigungen, daß der Anfrage genügt werden konnte. Durch diese Sicht ist es naheliegend, die Abstraktionsformen Prozedur beziehungsweise Funktion und Modul, wie sie in lokalen Diensten, wie zum Beispiel Funktions-Bibliotheken zu finden sind, auf das Netzwerk zu erweitern. Das sich daraus ergebende Konzept ist der *Remote Procedure Call (RPC)* [Nelson 81] (Abbildung 1.1), der den Aufruf entfernter Prozeduren logisch dem Aufruf lokaler gleichsetzen will, um auch dem Programmierer als Benutzer ein gewohntes Bild zu verschaffen und so die Komplexität der Programmierung im Netzwerk zu verbergen. In diesem Ablauf läßt sich stets eine sogenannte Client/Server-Beziehung identifizieren. Die initiierende Instanz wird *Client* beziehungsweise Dienstnehmer genannt, die ausführende *Server* oder Diensterbringer.

Damit werden aber die möglichen weiteren Vorzüge einer Netzwerkumgebung außzer der Verteilung der Software-Komponenten auf geeignete Orte, zum Beispiel wegen lokal vorhandener Rechenleistung oder großer Datenbestände, ebenfalls verborgen: Es gibt keine Ausnutzung möglicher Parallelität, da der Prozeduraufruf implizit eine synchrone Form ist, bei der der Dienstnehmer blockiert, bis das Ergebnis zur Verfügung steht. Sogar wenn keine Rückgabewerte erwartet werden, muß gewartet werden, um nur bei erfolgreicher Ausführung zum nächsten Programmschritt überzugehen.

Kommerzielle Systeme gehen daher wieder einen Schritt zurück, indem sie das strenge RPC-Konzept aufweichen und so auch sogenannte asynchrone RPC zulassen, bei denen der Dienstnehmer nach einem Aufruf wieder explizit auf die Ergebnisse wartet, aber in der Zwischenzeit andere Aufgaben wahrnehmen kann (zum Beispiel [Schill 93]). Damit bewegt man sich aber wieder deutlich in Richtung nachrichtenorientierte Kommunikation, wie es zum Beispiel die Herleitung des RPC in [Sloman, Kramer 88] zeigt. Alternativ dazu kann man das Kon-

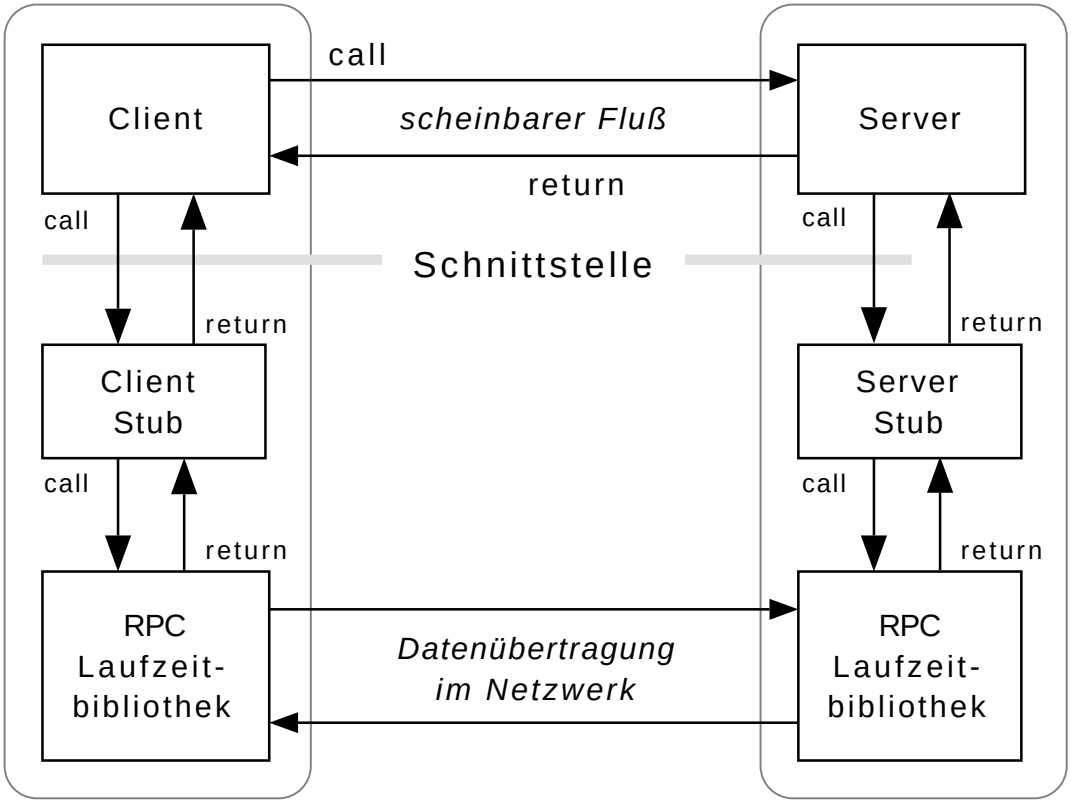


Abbildung 1.1: Die Grundstruktur eines entfernten Prozeduraufrufs (RPC)

zept des *Futures* [Walker et al. 90] sehen, das aus dem Bereich der Parallel-Programmierung für LISP-orientierte Systeme stammt [Halstead 85]. Ein Future ist ein Platzhalter für ein Ergebnis einer nebenläufigen Operation, hier also einer entfernten, und ist damit eine andere Synchronisationsform zur Verbesserung der Effizienz.

Sowohl lokale als auch entfernte Prozeduren stehen nur selten allein. Funktionen werden meist in logisch zusammengehörige Gruppen zusammengefaßt und in Modulen zum Beispiel mit verwendeten Typen abgelegt. Für den entfernten Fall stellt die Schnittstelle eines Modules dann im allgemeinen die exportierte Einheit des Diensterbringers dar. Wird sowohl auf Seite des Dienstnehmers wie des Diensterbringers die gleiche Programmiersprache eingesetzt, könnte das Interface wie im lokalen Fall aus den Schnittstellenbeschreibungen dieser Sprache abgeleitet werden. Das setzt aber voraus, daß das RPC-Subsystem die Sprache typvollständig abbildet, was gerade bei den verbreitetesten Systemen nicht der Fall ist (zum Beispiel [Corbin 91], [Schill 93]). Daraus ergibt sich die Notwendigkeit von besonderen Schnittstellen-Beschreibungssprachen den *Interface Definition Languages*,

die die Schnittstellen mit dem Typsystem des RPC darstellen. Auch für den Fall, daß unterschiedliche Implementierungs-Sprachen auf den verschiedenen kommunizierenden Seiten verwendet werden sollen, ist für mindestens eine davon eine externe Beschreibung der Schnittstellen notwendig, um den Sprachübergang abzubilden.

Betrachtet man objektorientierte Systeme wie zum Beispiel Emerald [Hutchinson 87], wird der Begriff RPC durch die *Remote Object Invocation (ROI)* ersetzt. Grundsätzlich bleibt es aber bei einer Menge entfernter Prozeduren, hier Methoden genannt, die in einem zustandsbehafteten Diensterbringer liegen, der das Objekt verwaltet. In Emerald soll die Verteilung der beteiligten Objekte vollkommen verborgen und ein uniformer Zugriff auf alle Objekte in einem verteilten System geboten werden. Es liegt hier beim Compiler der benutzten Sprache, zu erkennen, welche Zugriffe stets lokal sind und so eine Effizienz nahe derer von Systemen zu erreichen, die entfernte und lokale Zugriffe unterscheiden. Dieses Konzept zwingt aber auch zum Erstellen eines komplett neuen Systems, da es sich nicht als *Add-On*-Komponenten an bestehende Systeme und zum Beispiel deren Compiler anbinden läßt.

Weitere neue Ansätze für die Weiterentwicklung des RPC-Konzeptes dienen neben der Effizienzverbesserung der erhöhten Flexibilität und Behandlung der Heterogenität. Aufgrund der sich möglicherweise zeitlich verändernden Rolle von Dienstnehmer und Diensterbringer werden zum Beispiel im *Distributed Computing Environment (DCE)* der *Open Software Foundation (OSF)* rückwärtige Aufrufe des Diensterbringers an den Dienstnehmer ermöglicht. Ein anderer Ansatz ist die Installation von neuem Code im Diensterbringer, was aber neben der bereits bestehenden Datentyp-Problematik (s.o.) auch portable Code-Darstellungen erforderlich macht.

Neben den vielen kommerziellen und wissenschaftlichen Ansätzen hat sich zum Beispiel auch die ISO als Standardisierungs-Organisation mit Aufrufen entfernter Funktionalität befaßt. Das ISO/OSI-Referenzmodell und die dazugehörigen Standards behandeln jedoch zum gegenwärtigen Zeitpunkt nur Protokolle zwischen den Kommunikationspartnern und geben keine Schnittstellen an, was an anderer Stelle gerade ein wichtiges Problem für die hohe und portable Verbreitung von Systemen darstellt. Das *Remote Operations Service Element (ROSE)* [Kerner 89] der Anwendungsschicht berücksichtigt durch seine Operationsklassen sowohl synchrone als auch asynchrone Aufrufe und verschiedene Behandlung der Fehlerfälle der Kommunikation (Abbildung 1.2).

Für den RPC stellt sich aber stärker als für lokale Prozeduren die Frage nach der Anbindung beziehungsweise Auffindung geeigneter Dienste. Zu dieser Vermittlungsaufgabe wird häufig eine geeignete Infrastruktur angeboten, die dynamisch oder statisch Dienstnehmer und Diensterbringer zuordnet.

In dieser Arbeit soll eine Beziehung zwischen lokal vorhandenen Konzepten des

Operationsklassen	
synchron,	Rückmeldung im Erfolgs- und Fehlerfall
asynchron,	Rückmeldung im Erfolgs- und Fehlerfall
asynchron,	Rückmeldung nur im Fehlerfall
asynchron,	Rückmeldung nur im Erfolgsfall
asynchron,	keine Rückmeldung
Dienstelemente	
RO_INVOKE	Initiieren einer Operation mit Parametern
RO_RESULT	Positive Antwort mit Parametern
RO_ERROR	Negative Antwort mit Parametern
RO_REJECT_U	Abbruch der Operation durch den Dienstnehmer
RO_REJECT_P	Abbruch der Operation durch den Diensterbringer

Abbildung 1.2: Operationsklassen und Dienstelemente des ISO ROSE

Tycoon-Systems und einer für dessen Struktur geeigneten Client/Server-Bibliothek hergestellt werden.

1.1 Zielsetzung

Ziel der Arbeit ist es, das Tycoon-System [Matthes 93] durch eine besonders leistungsfähige Bibliothek um Client/Server-Funktionalität zu erweitern. Das System soll dabei in seiner Grundstruktur nicht verändert, was bedeutet den bisher schon erfolgreichen Add-On-Ansatz ([Matthes, Schmidt 95], [Rudloff et al. 95]) weiterzuverfolgen. Wesentliche Eigenschaften wie orthogonale Persistenz, Polymorphie, generalisierte Wertbindungen und statische Typsicherheit sollen dabei sowohl für die Implementation genutzt werden, als auch bei entfernt zur Verfügung gestellten Diensten verfügbar gemacht werden.

Die vorgegebene Sprache *Tycoon Language (TL)* des Tycoon-Systems stellt zur Gliederung von Software ein Modulsystem bereit. Als Einheiten des Exports für verteilte Anwendungen bieten sich daher Module an. Lokal werden Module in den Objektspeicher geladen und wie alle Tycoon-Werte persistent gebunden. Analog sollen für entfernte Module, soweit es das Medium Netz mit seinen vielfältigen Fehlermöglichkeiten zuläßt, persistente Bindungen ermöglicht werden. Dabei darf jedoch die Autonomie kommunizierender Objekt-Speicher nicht aufgehoben werden. Außerdem sollen keine globalen Verfahren zur Konsistenzwahrung, wie zum Beispiel verteilte Garbage-Collection, erforderlich werden.

Da entfernte Dienste im Gegensatz zu lokalen Modulen nicht mit Modul-Namen aus dem Dateisystem in den Objekt-Speicher kopiert werden können, ist ein Auswahlverfahren notwendig. Dazu sollten Dienste neben ihrem Dienst-Typ auch

über geeignete Attribute verfügen, die sich über die Lebenszeit eines Diensterbringers auch ändern können sollten und die über flexible Verfahren auf Seiten des Dienstnehmers mit seinen Anforderungen verglichen werden können.

TL zeichnet sich durch ein besonders leistungsfähiges Typsystem aus. Auf externe Dienstbeschreibungen ist zu verzichten, da bestehende *Interface Definitions Languages*, wie andere externe Sprachen auch, nur einen Ausschnitt dieser Leistungsfähigkeit besitzen. Die Verwendung von TL selbst bietet auch eine übersichtlichere Integration der Verteilung in das bestehende System. Es soll ein eigener RPC entwickelt werden, der typvollständig ist.

Externe Dienste können in Tycoon lokal eingebunden und daraufhin auch für entfernte Tycoon-Systeme nutzbar gemacht werden. Externe Dienste auf direkte Weise entfernt nutzbar zu machen, ist auch deshalb nicht unbedingt notwendig, weil sich das Tycoon-System unter anderem durch seine hohe Portabilität auszeichnet.

Es ist wichtig, diese Portabilität *nicht* durch Nutzung aufwendiger externer Dienste bei der Implementation des RPC und der damit im Zusammenhang stehenden Netz-Infrastruktur zu gefährden. Eine große externe Kommunikations-Bibliothek könnte einen Großteil der geforderten Funktionalität bereitstellen, würde aber durch ihre Verfügbarkeit auf bestimmten Betriebssystem- beziehungsweise Hardware-Plattformen und durch ihre Komplexität ein Hindernis bei der Portierung darstellen. Im Gegensatz dazu bietet die hier verwendete sogenannte *Socket*-Schnittstelle eine schlanke Basis mit ausreichender Mächtigkeit und breiter Verfügbarkeit, mit der eine Übertragung beliebiger polymorpher Tycoon-Werte über heterogene Netzwerke als Kommunikations-Primitiv ermöglicht werden soll.

An die Adressierung der entfernten Dienste werden hohe Anforderungen gestellt. Neben möglichst flexiblen Mechanismen zur Auswahl geeigneter Dienste muß eine Ebene geschaffen werden, auf der Dienste weltweit eindeutig bezeichnet werden. Damit wird die Grundlage geschaffen, auf der sie für persistente Bindungen wiederauffindbar sind. Dieses Prinzip ergibt sich aus der Analogie zum Abarbeiten von langlebigen Tycoon-Prozessen in flüchtigen Betriebssystem-Prozessen [Matthes, Schmidt 94a], wobei hier zum Beispiel der diensterbringende Tycoon-Prozeß nicht mehr durch denselben Betriebssystem-Prozeß abgebildet wird. Jede direkte Netzwerkadressierung, die sich auf Maschinen und Betriebssystem-Prozesse bezieht, ist dabei als flüchtig anzusehen. Ein hierarchischer Ansatz ermöglicht es, das Verhalten zur Aufrechterhaltung der Kommunikation auf den jeweiligen Dienst abzustimmen, statt mit einer immer alle Möglichkeiten der Migration berücksichtigenden Vorgehensweise das Netz unnötig zu belasten.

Daß ein Tycoon-Prozeß nicht mehr durch denselben Betriebssystem-Prozeß abgearbeitet wird, bedeutet im Netzwerk aber auch, daß der Tycoon-Prozeß dabei seinen Standort geändert haben kann. Falls ein Diensterbringer ausfällt, kann ein Wiederanlaufen eventuell nur an einer anderen Stelle im Netz erfolgen. Da

Threads in Tycoon ganz allgemein migrationsfähige Einheiten sind, muß auf Seite des Dienstnehmers eher noch von einer größeren Mobilität ausgegangen werden.

Um die Verfügbarkeit und Sicherheit entfernter Dienste zu erhöhen, sind diese unter Ausnutzung des gegebenen Multi-Threadings im Tycoon-System zu implementieren.

1.2 Gliederung der Arbeit

Die in der Zielsetzung angeführten Konzepte und Verfahren legen eine Schichten-Architektur der zu erstellenden Bibliothek nahe. Die einzelnen Ebenen geben das Gerüst für die folgenden Kapitel vor.

In Kapitel 2 wird das persistente Objektsystem Tycoon als Basis für diese Arbeit vorgestellt. Nach einer Darstellung der Struktur des Systems wird in die Programmiersprache *Tycoon Language (TL)* eingeführt, die in dieser Arbeit zur Notation von Programmbeispielen beziehungsweise -auszügen genutzt wird.

Als Grundlage des Client/Server-Systems für Tycoon dient die Übertragung von Werten beliebigen Typs, die in Kapitel 3 über die auf vielen Plattformen weitgehend identisch vorhandene *Socket*-Schnittstelle hergestellt wird. Dabei wird in Abschnitt 3.2 ersichtlich, daß das Tycoon-System bereits eine günstige Basis zur Implementation einer typvollständigen Wertübertragung im Netz bereitstellt.

Kapitel 4 beschreibt RPC-Mechanismus für Tycoon, der bereits das gesamte Typsystem bereitstellt und nach der Bindung statische Typsicherheit garantiert, wie es auch bei der Bindung lokaler Module der Fall ist. Auch sind bereits hier die Diensterbringer potentiell multi-threaded. Lediglich die Persistenz der Bindung ist nicht garantiert.

Der Schritt hin zur vollständigen Abbildung der Tycoon-Eigenschaften für die Benutzung entfernter Dienste wird mit Kapitel 5 vollzogen. Ein geeignetes hierarchisches Adressierungsschema und eine mit den Mitteln aus Kapitel 4 erstellte Netzinfrastruktur ermöglichen es, selbst bei sich verlagernden Diensterbringern, persistente Bindungen zu realisieren.

Schließlich erfolgt mit Kapitel 6 eine Darstellung der implementierten Bibliothek und ihrer erreichten Eigenschaften. Darüberhinaus sind hier noch eine Reihe von möglichen Fortsetzungen der Arbeit aufgeführt, die zum Teil Verknüpfungen mit weiteren bestehenden oder im entstehen begriffenen generischen Komponenten des Tycoon-Systems beinhalten.

Kapitel 2

Das persistente Objektsystem Tycoon

Im Rahmen dieser Arbeit ist eine Bibliothek für das Tycoon-System entstanden, das am Arbeitsbereich Datenbanken und Informations-Systeme im Fachbereich Informatik der Universität Hamburg entwickelt wird. In diesem Kapitel wird die Architektur des Implementations-Systems und die verwendete Sprache *TL* beschrieben, die auch in den folgenden Kapiteln zur Notation von Beispielen dient.

Die Sprachumgebung soll die effiziente Programmierung von datenintensiven Anwendungen, die Korrektheit von Programmen und geringen Wartungs- und Erweiterungsaufwand gewährleisten. Darüberhinaus sollen vorhandene Software-Ressourcen wie Bibliotheken und externe Dienste genutzt werden können.

2.1 Die Architektur des Tycoon-Systems

Um die Implementation des Gesamtsystems beherrschbar zu machen, ist eine Architektur zu wählen, die die Komplexität des Systems reduziert. Das Tycoon-System ist in mehrere funktionale Schichten organisiert (Abbildung 2.1), wobei jede Schicht für einen bestimmten Aufgaben-Bereich optimiert ist.

Die oberste Schicht des Tycoon-Systems stellt die Programmiersprache zur Verfügung und damit die Schnittstelle zum Benutzer dar.

Das Compiler Front-End wandelt Terme der Programmiersprache in die Zwischenrepräsentation *Tycoon Machine Language (TML)* um [Gawecki, Matthes 94]. TML ist eine nichtlineare Codierung im *Continuation Passing Style (CPS)*. Ihre Baumstruktur erlaubt weitreichende Analysen und Optimierungen, wie sie aus Datenbanksystemen und anderen optimierenden Compilern bekannt sind [Gawecki, Matthes 95]. Der Sprachkern erlaubt lediglich wenige Instruktionen

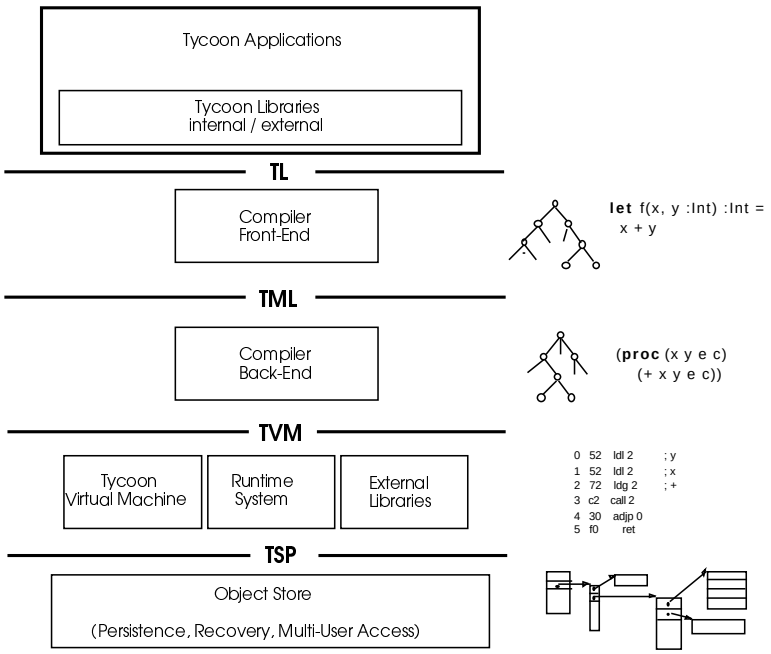


Abbildung 2.1: Die Architektur des Tycoon Laufzeit-Systems

für Funktionsdefinitionen und -applikationen, destruktive Zuweisungen, bedingte oder wiederholte Auswertungen, Ausnahmebehandlungen und den Aufruf von Funktions-Code externer Programmiersprachen. Zur späteren Anwendungen dynamischer Optimierungen ist es möglich eine zu TML isomorphe Darstellung *Persistent TML (PTML)* zu speichern [Kiradjiev 94].

Der TML-Code wird durch das Compiler Back-End in einen persistenten, portablen Byte-Code (TVM-Code für *Tycoon Virtual Machine*) umgewandelt, dessen Instruktionen an den Befehlssatz des Transputers angelehnt sind. Der Code ist von konkreter Prozessor-Hardware unabhängig und umfaßt Operationen auf elementaren Tycoon-Datenstrukturen.

Der Tycoon Byte-Code wird durch die *Tycoon Virtual Machine (TVM)* ausgeführt. Dieser virtuelle Computer bildet die Operationen auf die jeweilige Hardware-Plattform ab.

Die Daten des Systems von atomaren Werten bis zu Programmcode und Threads werden durch das *Tycoon Store Protocol (TSP)* in einem persistenten Objektspeicher verwaltet. Die Wahl der Store-Implementierung ist durch die TSP-Abstraktion weitgehend frei. Es werden neben zwei eigenen Store-Implementierungen gegenwärtig NapierStore und ObjectStore unterstützt.

Operationen in höheren Schichten werden auf Objektkonstruktoren, -modifikatoren und -destruktoren abgebildet. Dies entkoppelt die höheren Schichten bis

zu den Sprachen von konkreten Speichermodellen der zugrundeliegenden Plattform, wie es für die Code-Ebene durch die TVM geleistet wird. Der jeweilige Objektspeicher ist für eine effiziente Speicherverwaltung verantwortlich. Eine Freispeicherverwaltung durch die höheren Ebenen findet nicht statt. Mit Hilfe von Sicherungspunkten und Rücksetzmechanismen kann für das gesamte System Datenbankfunktionalität realisiert werden.

2.2 Die Programmiersprache Tycoon-Language (TL)

Neben geringen Teilen an *C*-Code ist der größte Teil dieser Arbeit in der Sprache des Tycoon-Systems selbst, TL, geschrieben worden. Dadurch konnten die Vorteile des Systems und dessen Sprache nicht nur auf die Programmierung im Netzwerk erweitert werden, sondern auch selber für die Implementation genutzt werden. TL ist damit auch die Sprache, die für alle Beispiele im folgenden Text benutzt wird und soll daher an dieser Stelle kurz vorgestellt werden. Eine detailliertere Beschreibung findet sich in [Matthes, Schmidt 92].

Werte und Typen

In TL sind sowohl statische als auch dynamische Bindungen möglich. Eine statische Wertbindung wird zum Beispiel durch

```
let a = 3
```

definiert. Nach der Evaluation des Terms ist der Bezeichner `a` an der Wert 3 statisch gebunden, dabei kann an Stelle der 3 in diesem Beispiel ein beliebiger Ausdruck stehen. Zusätzlich können in TL durch Klammerung mit *begin* und *end* Sichtbarkeitsblöcke definiert werden. Damit wäre nach Eingabe von

```
let a = 3
begin
  let a = 7
  let b = a
end
let c = a
```

der Bezeichner `a` an den Wert 3 gebunden und die lokale Bindung von `a` an 7 sowie die Bindung des Bezeichners `b` nicht mehr sichtbar.

Dynamische Bindungen werden durch Funktionen dargestellt, die in TL Elemente erster Ordnung sind. Funktionen werden durch das Schlüsselwort *fun* eingeleitet. Mit

```
let square = fun(i :Int) i*i
```

wird eine Funktion definiert, die das Quadrat des Arguments zurückgibt. Innerhalb des Funktionsrumpfes sind neben den im Sichtbarkeitsbereich der Funktion definierten globalen Bezeichnern und den eventuell definierten lokalen Bezeichnern die an die Formalparameter gebundenen Aktualparameter erreichbar. Der Doppelpunkt leitet einen Typbezeichner ein. *:Int* gibt daher an, daß das Argument `i` vom in TL vordefinierten Ganzzahl-Typ *Int* ist. Es ist nicht immer notwendig, explizite Typangaben zu machen, da der TL-Compiler in der Lage ist, teilweise fehlende Typangaben abzuleiten. Im Beispiel ist der Ergebnistyp der Funktion ebenfalls *Int*, auch wenn dies nicht angegeben ist, da das Ergebnis der Multiplikation zweier Werte vom Typ *Int* (das Symbol `*` steht immer nur für die Multiplikation von Ganzzahl-Werten) ebenfalls von diesem Typ ist.

Da in TL alle Objekte typisiert sind, hat die Funktion `square` den Typ

```
:Fun(:Int):Int
```

Solche Funktionstypen sind zum Beispiel für die Beschreibung von Funktionen höherer Ordnung notwendig. Will man eine Funktion schreiben, die eine beliebige Funktion des oben angegebenen Typs auf ein Argument zweimal anwendet, ist in der Signatur ein Funktionstyp erforderlich.

```
let twice = fun(f :Fun(:Int):Int  i :Int) f(f(i))
```

Damit wird auf das Argument `i` die Funktion `f` zweimal angewendet. Es wäre so auch denkbar, daß die Funktion `twice` eine Funktion zurückgibt, die auf ein Argument vom Typ *Int* angewendet, die übergebene Funktion `f` zweimal anwendet.

Neben den vordefinierten Typen *:Int :Bool :Char* etc. können eigene Typen definiert werden. Dazu stehen die Typkonstruktoren *Tuple*, auch mit Varianten und *Record* zur Verfügung.

- Tupel sind *geordnete* Mengen von Bindungen

```
Let Attributes = Tuple
  name     :String
  currency :String
  cost     :Real
end

let a :Attributes = tuple
  "alarm clock"
  "GBP"
  3.4
end
```

Auf die Komponenten des an den Bezeichner a gebundenen Tupels wird über

```
|> a.currency
"GBP" :String
```

zugegriffen.

- Records sind dagegen *nicht* geordnete Mengen von nicht-anonymen Bindungen

```
Let AttrRecord = Tuple
  name     :String
  currency :String
  cost     :Real
end

let a :AttrRecord = tuple
  let currency = "GBP"
  let cost     = 3.4
  let name     = "alarm clock"
end
```

Der Zugriff erfolgt analog den Tupeln über Punktnotation, jedoch können Records um weitere Bindungen erweitert werden.

```
let b = extend a with
  let time = 14.00
end
```

Subtypbeziehungen und Subtyppolymorphismus

Signaturen der Form `X :A` stellen partielle Typ-Spezifikationen dar, die aussagen, daß `x` *mindestens* `A` erfüllt. Ein an den Bezeichner `x` gebundener Wert kann aber auch eine genauere Spezifikationen `B` erfüllen. Die zugrundeliegende partielle Ordnung auf Typen „`B` ist präziser als `A`“ wird durch die induktiv definierte `B <:A` (`B` ist Subtyp von `A`) in TL explizit beschrieben [Matthes 93]. Der Supertyp aller nicht parametrisierten Typen ist `Ok`.

```
Let Person = Tuple
  name :String
  age :Int
end
Let Student = Tuple
  name :String
  age :Int
  semester :Int
end
let fred :Student = tuple
  "Fred""
  20
  1
end
let getPersonAge(P <: Person  p :P) = p.age

|>getPersonAge(fred);
20 :Int
```

Die Funktion `getPersonAge()` ist eine polymorphe Funktion, da sie keinen expliziten Argumenttyp vorschreibt, sondern für `p :P` lediglich angibt, daß `P <:Person` sein soll. Daher ist die Funktion auch auf einen Wert des Type `:Student` anwendbar. Für eine detaillierte Beschreibung der Subtypisierungsregeln sei wieder auf [Matthes 93] verwiesen. Subtypbeziehungen gelten für alle in TL definierbaren Typen einschließlich der Basistypen.

Module und Bibliotheken

TL-Programme setzen sich aus Modulen zusammen, die, wie zum Beispiel auch in Modula-2, durch getrennte Schnittstellen (*`interface`*) und Implementations-Teile (*`module`*) beschrieben werden. Ein Modul, das Funktionen anderer Module nutzen will, muß diese zuvor importieren. Es stehen dann die Funktionen zur Verfügung, die im *`interface`* des betreffenden Moduls exportiert werden. Die Schnittstellen stellen somit die Typen von Modulen dar, wobei ein Modul unterschiedliche Interfaces haben kann. Genauso kann eine Schnittstelle als Typ

mehrere Module beschreiben, die *mindestens* die im Interface beschriebenen Komponenten enthalten.

Die Verbindung von Modul zu Schnittstelle wird in TL über Bibliotheken (*library*) hergestellt.

```
library example with
  interface
    A
    B
  module
    a :A
    b :B
  interface
    C
  module
    c :C
    d :A
end ;
```

Persistenz

Alle TL-Bindungen sind potentiell persistent, da sie in einem Objektspeicher abgelegt werden, dessen Zustand auf einem sicheren Medium stabilisiert werden kann. Auf einen solchen gesicherten Zustand kann dann bei Neustart des Systems zurückgegriffen werden. Dazu ist es nicht notwendig, Bindungen explizit als persistent zu kennzeichnen. Ist ein Wert im Objektspeicher durch keine Bindung mehr referenziert, kann er durch den *Garbage Collector* des Tycoon-Systems entfernt werden.

Anbindung externer Dienste

Die hier gemeinten externen Dienste sind nicht mit entfernten Diensten zu verwechseln. Es sind hier nicht zum Tycoon-System selbst gehörende Dienste in Form von externen Bibliotheken gemeint, deren Funktionalität nicht durch Tycoon nachgebildet werden soll, sondern die eingebunden werden sollen.

Externe Bibliotheken, die zum Beispiel durch die Betriebssystemumgebung bereitgestellt werden oder auch zugekauft sind, sollen typsicher in TL-Programme eingebunden werden. Dazu sollten die Bibliotheken vorzugsweise als *shared libraries* vorliegen, können aber auch statisch in die Tycoon-Maschine gebunden werden. Mit dem Schlüsselwort *bind* können Funktionen aus diesen Bibliotheken in TL typisiert bekanntgegeben werden.

```
let f = bind(:Fun(:Int):Int "library" "function" "signatur")
```

Hierbei wird der Bezeichner `f` an die externe Funktion *function* in der *C*-Bibliothek *library* gebunden. Durch die Zeichenkette *signatur* wird die Typübersetzung der Parameter der externen Sprache *C* nach TL angegeben. In TL hat `f` den Typ `:Fun(:Int) :Int`.

Kapitel 3

Datenstromorientierte Kommunikation

Als Basis für die in dieser Arbeit dargestellte Umgebung ist eine Abbildung von Kommunikations-Primitiven in Tycoon erforderlich. Das elaborierte Typsystem von Tycoon [Matthes et al. 94] und die existierende Linearisierungs-Funktionalität des persistenten Objektspeichers [Matthes et al. 95] läßt die Anbindung eines externen RPC-Dienstes wie ONC-RPC [Corbin 91] oder DCE-RPC als Bestandteil eines gesamten Systems zur Verteilung von Anwendungen [Schill 93] als nicht angemessen erscheinen, da verbreitete Systeme insbesondere auch keine Persistenz zum Beispiel von Bindungen unterstützen.

3.1 Plattformunabhängige Socket-Kommunikation

Um das Tycoon-System auch inklusive der neuen Client/Server-Umgebung portabel zu halten, ist eine Netzwerk-Programmier-Schnittstelle erforderlich, die möglichst weite Verbreitung in verschiedenen Betriebssystemen, Netzwerkumgebungen und Hardwarearchitekturen gefunden hat. Außerdem ist es für einen möglichst geringen Portierungs-Aufwand nützlich, wenn diese Schnittstelle überschaubar ist.

Gerade unter dem Aspekt verteilter Systeme und Client/Server-Programmierung gibt es einige Systeme, die sich als Subsysteme an Tycoon anbinden ließen und auch einen großen Teil der benötigten Funktionalität bereitstellen. In [Johannisson 95] wird die Benutzung des ONC-RPC, der durch das auf ihm aufbauende *Network File System* (NFS) [Santifaller 93] große Verbreitung gefunden hat, für Tycoon-RPC dargestellt. Das *Distributed Computing Environment* (DCE) der Open Software Foundation (OSF) hat sich ebenfalls zum Ziel gesetzt auch einer breiten Anzahl von Plattformen weitreichende Unterstützung für verteilte

```
interface Socket                import  iNetAddress  thread

export

  SOCK_STREAM()    :Int      (* stream socket *)
  SOCK_DGRAM()     :Int      (* datagram socket *)
  SOCK_RAW()       :Int      (* raw-protocol kInterface *)

  (* Address families *)
  AF_UNSPEC      :Int      (* unspecified *)
  AF_UNIX        :Int      (* local to host (pipes, portals) *)
  AF_INET        :Int      (* kInternetwork: UDP, TCP, etc. *)

  (* Protocol families, same as address families for now. *)
  PF_UNSPEC      :Int
  PF_UNIX        :Int
  PF_INET        :Int

  Let Address = Tuple
                  addressType :Int         (* Address family *)
                  port        :Int         (* Port-Number *)
                  inAddress   :iNetAddress.T
                end

  Let T <:Ok                    (* Socket File-Handle    *)

  new(domain, type, protocol :Int) :T (* socket() *)
  destroy(sock :T) :Ok (* close() *)

  bind(sock :T name :Address) :Ok

  listen(sock :T requestQueue :Int) :Ok
  accept(sock :T var accepted :Address) :T
  (* 'abort()' aborts sockets accepting requests within thread.T 't'  *)
  abort(t :thread.T(Ok)) :Ok

  connect(sock :T toSocket :Address) :Bool

  write(sock :T data :word.T size :Int) :Ok
  read(sock :T size :Int) :word.T

end;
```

Abbildung 3.1: Die Schnittstelle Socket.ti zur Socket-Funktionalität(Ausschnitt)

Anwendungen zu geben. Leider führt dieser Funktionsumfang zu relativ großen Gesamtsystemen, die sich, wie im Laufe dieser Arbeit noch an einigen Stellen zu zeigen ist, nicht bruchlos in Tycoon einbinden lassen und außerdem die Portabilität trotz hoher Verbreitung durch den Umfang der Programmier-Schnittstellen erschweren.

Mit 4.2BSD-Unix wurde ein Satz von Systemaufrufen eingeführt, der zur Interprozeßkommunikation dient. Diese (nach dem ersten Aufruf) sogenannte *Socket*-Schnittstelle ist dahingehend generisch, daß sich neben Netzwerkprotokollen aus der *Internet-Domain* zum Beispiel auch lokale Kommunikations-Endpunkte aus der sog. *UNIX-Domain* nutzen lassen.

Sockets sind zwar von keiner Institution als Standard definiert, stellen aber dennoch einen de-facto-Standard dar und haben eine genauso hohe Verbreitung wie andere Basis-Dienste, sicherlich auch wegen ihrer einfachen Benutzbarkeit und guten Einbindung ins (Unix-) Betriebssystem, gefunden. Die *WinSock*-Schnittstelle als Variante der Sockets wurde von Microsoft in die *Windows Open Software Architecture* (WOSA) [Microsoft 95] aufgenommen und soll in einer zukünftigen Version neben größerer Flexibilität auch größere Verbreitung außerhalb der Windows-Welt (zum Beispiel für Apple-Macintosh-Rechner) erreichen [Intel Architekture Labs 95].

Ein Socket stellt einen Kommunikations-Endpunkt dar, der zunächst mit dem Systemaufruf `socket()` zu erzeugen ist. Dabei wird die Domäne des neuen Sockets, das heißt, ob es sich zum Beispiel um lokale Interprozeßkommunikation (AF_UNIX) oder Netzwerkprotokolle des Internet (AF_INET) handelt, der Typ, das heißt, ob das Socket Datagramme übertragen soll (SOCK_DGRAM) oder als Stream behandelt wird (SOCK_STREAM), und das zu verwendende Protokoll angegeben. Aus Typ und Domäne ergeben sich auch Vorgaben für das Protokoll. Zum Beispiel wird standardmäßig das *Transmission Control Protocol* (TCP) ausgewählt, wenn in der Internet-Domäne Streams benutzt werden.

Im Unix-Betriebsystem, in dem sie originär entstanden sind, stellen Sockets Datei-Deskriptoren dar und stellen die gängigen Datei-Funktionen zur Verfügung. Diese vollständige Integration wird leider nicht auf allen Plattformen erreicht. Die WinSock-Schnittstelle erlaubt nur die Benutzung von Funktionen, die ausschließlich auf Sockets arbeiten. Aufgrund der strategischen Bedeutung der Windows-Betriebssysteme ist diese Einschränkung unbedingt zu berücksichtigen. Die von WinSock bereitgestellten Funktionen finden sich auch in der originalen Socket-Schnittstelle wieder und stellen so die portable Basis dar.

Auf einem Socket können auf Empfängerseite mit `listen()` die Empfangsbereitschaft angezeigt und mit `accept()` Verbindungen angenommen werden, die auf Senderseite mit `connect()` angefordert werden (Abbildung 3.2). Da WinSock ein `read()` oder `write()` nicht erlaubt, beschränken wir uns auf `send()` beziehungsweise `recv()`, was jedoch bedeutet, daß Datenströme in Teile aufzuspalten sind,

die die maximalen Puffergrößen dieser Funktionen nicht überschreiten. Die Analogie zu Dateien ist damit leider weitestgehend aufgehoben, da man hier erwarten könnte, daß aus einem Datenstrom, der n Bytes enthält, auch n Bytes in einem Stück entnommen werden können, auch wenn n größer als 2^{12} ist. Die in Abbildung 3.2 aufgeführte Funktion `close()` ist für WinSock durch `closesocket()` zu ersetzen.

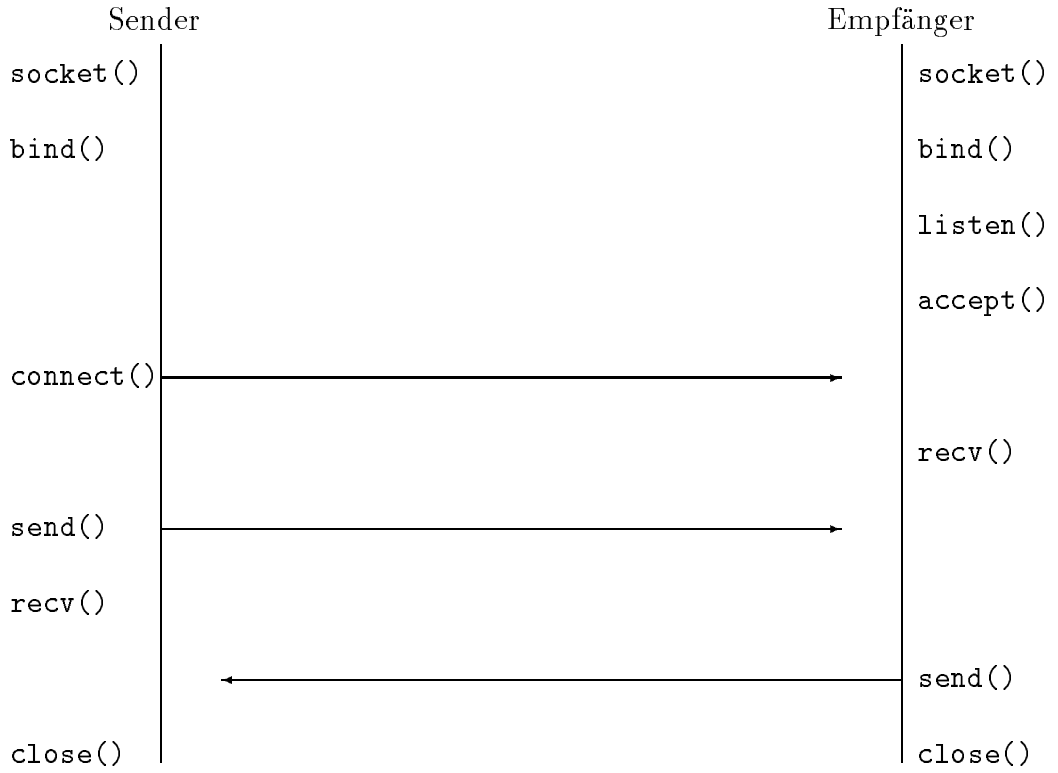


Abbildung 3.2: Ablauf einer Socket-Kommunikation

Im Tycoon-System werden Sockets als abstrakter Datentyp abgebildet, auf dem alle auf Sockets verfügbaren Funktionen definiert sind. Diese Funktionen werden zumeist direkt von der Betriebssystem-Bibliothek abgearbeitet. Lediglich die Abbildung der Werte in eine kanonische Netzwerk-Form und die Berücksichtigung des Tycoon-Multithreading sind in einem in *C* geschriebenen Modul zu behandeln. Für das Multithreading ist es nötig, daß ein `accept()` auf einem Socket nicht den gesamten Tycoon-Prozeß anhält, dennoch aber blockieren kann, wenn nur ein Thread aktiv ist beziehungsweise wird.

Da das Tycoon Socket-Modul als Basis auch für migrierende Dienstnehmer dienen soll, dürfen systemabhängige Konstanten, wie sie in *C* durch Makros abgebildet werden, *nicht* als Werte gebunden werden, damit sie an dem jeweiligen Standort wieder neu beim Tycoon-Laufzeitsystem beziehungsweise der darin enthaltenen

Socket-Bibliothek angefragt werden. Das *C*-Makro `SOCK_STREAM` wird daher auf die TL-Funktion `socket.SOCK_STREAM()` abgebildet. Die für Tycoon-Funktionen ungewöhnliche Schreibweise ist gewählt worden, damit Nutzer der Bibliothek einfach auf vorhandene Systemdokumentation über Sockets zurückgreifen können und nicht eine eigene Dokumentation für Tycoon erstellt werden muß, nur um Diskrepanzen in der Benennung zu erläutern.

Im Rahmen dieser Arbeit ist das Socket-Modul für SunOS4 und Solaris auf SPARC-Rechnern, MacOS auf PowerPC und 68k-Prozessoren, Linux auf x86 und WindowsNT auf x86-Prozessoren portiert worden (siehe auch [Breilmann 95]). Die im weiteren verwendeten Funktionen sind als Ausschnitt des TL-Interfaces in Abbildung 3.1 dargestellt. Durch diese schmale Schnittstelle und die bereits geleisteten Umsetzungen sind weitere Portierungen mit sehr geringem Aufwand verbunden, wie Versuche mit OS/2 auf x86-Rechnern oder der BSD-Unix Variante NeXTStep auf NeXT-68k-Rechnern gezeigt haben.

3.2 Verbindungslose Übertragung polymorpher Werte

Der persistente Objektspeicher von Tycoon verfügt über generische Funktionen zur Linearisierung beliebiger Tycoon-Werte. Im Gegensatz zum plattformabhängigen Objektspeicher selbst sind diese Linearisierungen portabel: In Tycoon kompilierte Module mit Werten, Funktions-Code und Typen sind zwischen allen Plattformen, auf denen Tycoon verfügbar ist, austauschbar.

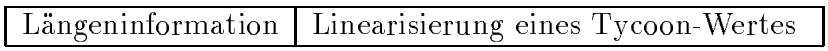


Abbildung 3.3: Einfaches Tycoon-Wert-Paket

Beliebige Tycoon-Werte lassen sich auf das Dateisystem des zugrunde liegenden Betriebssystems ausgeben und von dort wieder einlesen. Da die im letzten Abschnitt besprochenen Sockets als Datenströme genutzt werden können, liegt es nahe, auch über Sockets und damit über Netzwerke beliebige Tycoon-Werte übertragen zu können. Da allgemeinen Datenströmen keine Informationen über die Länge der darin enthaltenen Daten entnommen werden können, wie es bei Dateien im speziellen durch das Dateisystem im Betriebssystem möglich ist, muß Werten, die über Netzwerke gesendet werden, eine Längen-Information vor dem Wert-Paket mitgegeben werden (Abbildung 3.3). Für diesen Ganzzahlwert ist eine definierte Netzwerk-Repräsentation zu wählen, da er nicht als Tycoon-Wert linearisiert werden kann, ohne wieder vor demselben Problem der Paket-Länge, die auch für eine Ganzzahl unbekannt wäre, zu stehen. Für diese ein Zahl ist

aber eine Verwendung zum Beispiel der *External Data Representation* (XDR) des ONC-RPC, die im Gegensatz zur entsprechenden DCE-Schicht als Programmier-Schnittstelle zur Verfügung steht, wie sie in [Johannisson 95] an Tycoon angebunden wurde, kaum mit dem Ziel einer schmalen Basis externer Dienste vereinbar. Die Längeninformation wird daher in einem selbstdefinierten kanonischen Format über das Netz übertragen. Dieses einfache Protokoll kann dennoch vollständig in TL geschrieben werden, da lediglich ein vier Byte großer Wert mit dem höherwertigen Byte zuerst übertragen werden muß, bevor die Linearisierung gesendet wird

Interessant ist, daß so ein Problem, das innerhalb des ISO/OSI-Referenzmodells auf Schicht 6 direkt unterhalb der Anwendungsebene gesehen wird [Kerner 89], sich direkt im Transportsystem besonders einfach lösen läßt. Auch wird es damit möglich, den größten Teil der Netzwerk-Funktionalität mit der Ausdrucksmächtigkeit von TL zu formulieren.

Der grundlegende Linearisierungs-Algorithmus von Tycoon bildet zu einem Wert die transitive erreichbare Hülle. Das bedeutet unter Umständen für ein komplexes zu versendendes Objekt, daß viele Daten aus dem Objekt-Speicher, der es enthält, kopiert werden müssen. Für eine Menge von Tycoon-Systemen gilt jedoch, daß ein Teil der Werte, auf die sich andere Werte beziehen überall vorhanden ist. Solche *ubiquitären* Werte können vor dem Versenden entfernt und durch symbolische Referenzen ersetzt werden, anhand derer die Gegenseite den entsprechenden Wert im eigenen Objektspeicher wieder bindet.

Grundsätzlich ist diese Art der Datenreduktion für den Netzverkehr nicht Thema dieser Arbeit, sondern wird in [Pour 96] näher erläutert. Aus einem weiteren Aspekt dieses *dynamischen Linkens*, der *automatischen Replikation*, ergibt sich jedoch das für die Übertragung von Werten notwendige Protokoll (Abbildung 3.4).

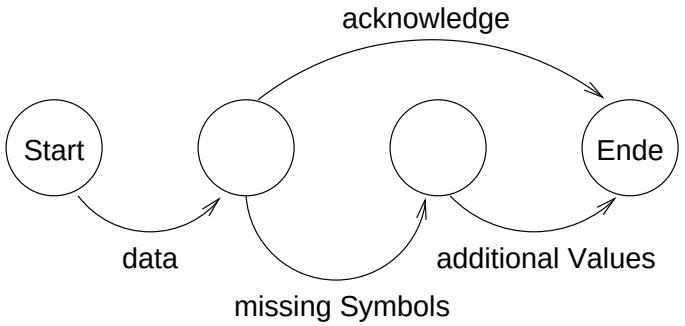


Abbildung 3.4: Protokollgraph für automatische Replikation

Über Ressourcen die als ubiquitär zu markieren sind, können entfernte Tycoon-Systeme sich nicht verständigen. Daher ist es einfach möglich, daß im Falle einer Kommunikation, also eines Austausches von Werten, gebundene Werte zuviel übertragen werden oder fehlen. Redundant übertragene Werte stellen kein inhaltliches Problem dar, füllen aber unnötig den Objektspeicher. Fehlende Werte führen zu einer Fehler-Situation. Diese läßt sich aber dadurch beheben, daß die fehlenden Module in einem weiteren Protokoll-Schritt angefordert werden (Abbildung 3.5).

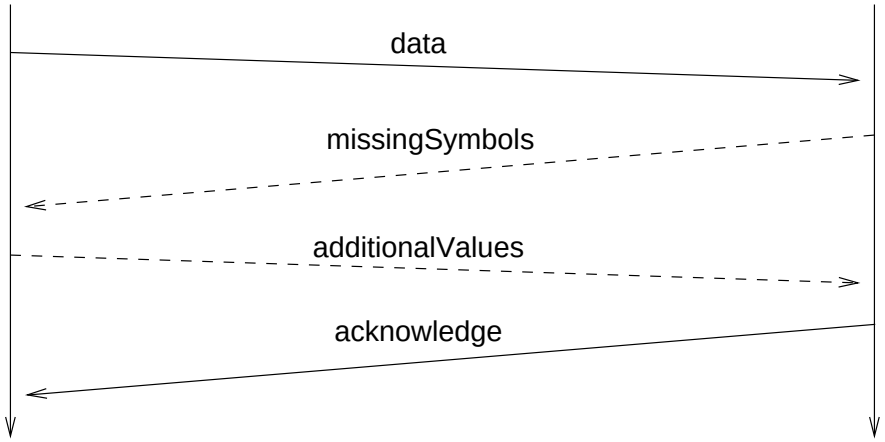


Abbildung 3.5: Ablauf einer automatischen Replikation

Die einzelnen Phasen dieses Protokolls werden durch ein variantes Tupel (Abbildung 3.6) dargestellt, den zeitlichen Ablauf gibt das Diagramm in Abbildung 3.5 beziehungsweise der Protokollgraph in Abbildung 3.4 wieder.

```
Let Packet(Data <:Ok) = Tuple
  case data with
    data :Data
  case missingSymbols with
    symbols :SymbolArray
  case additionalValues with
    values :Array(Ok)
  case acknowledge
end
```

Abbildung 3.6: Protokoll-Tupeltyp für automatische Replikation

Kapitel 4

Typsichere, polymorphe Client/Server-Programmierung

Die in Kapitel 1 diskutierten Erweiterungen des grundlegenden RPC-Konzeptes lassen sich vielfach mit einem einfachen, synchronen RPC und geeigneten weiteren generischen Systemmöglichkeiten abbilden. So ist es zum Beispiel möglich einen asynchronen RPC zu emulieren, indem in einem Kommunikations-Thread ein synchroner RPC abgesetzt wird. Damit ließe sich dann auch ein Future realisieren, da lediglich mit dem lokalen Thread synchronisiert werden muß.

Konzeptuell ist es also keine Schwäche, einen einfachen RPC zu realisieren, wenn, wie im Falle des Tycoon-Systems, andere Kommunikations-Szenarien darstellbar bleiben. Für die Implementation und Wartbarkeit der Client/Server-Bibliothek stellt es aber eine erhebliche Verbesserung dar und gibt die Möglichkeit, sich auf die wesentlichen Punkte für Tycoon wie Typvollständigkeit, Typsicherheit und Polymorphie zu konzentrieren.

Gerade diese Punkte sind der Anlaß einen eigenen Tycoon-zu-Tycoon-RPC zu realisieren, da, wie in Kapitel 3 gesehen, die Verwendung externer Kommunikations-Dienste trotz beachtlichen Umfanges TL nicht typvollständig darstellen kann.

Mit den Mitteln aus Kapitel 3 läßt sich ein von der Struktur her einfacher RPC-Mechanismus implementieren. Besonders wichtige Ergänzung der Wertübertragung, wie am Ende von Abschnitt 4.1 zu sehen ist, ist die Typsicherheit.

4.1 Entfernte Funktionen höherer Ordnung

Das Modul `port` aus Abschnitt 3.2 bietet neben der Übertragung von Tycoon-Werten auch eine Adressierungsebene, die zunächst für einen RPC-Mechanismus benutzt werden soll. Die in diesem Kapitel vorgestellten Module heißen daher

`portClient` und `portServer`, um sie von der geänderten Adressierung in Kapitel 5 zu unterscheiden.

Soll ein Tupel aus Funktionen als entfernter Dienst angeboten werden, muß zunächst eine Diensterbringer-Instanz erstellt werden.

```
let svr = portServer.new(portNumber)
```

In dieser Instanz wird der Dienst unter einem für diese Instanz eindeutigen Namen abgelegt.

```
let registeredService =
      portServer.register(svr service ServiceName)
```

Die Funktion `portServer.new()` erzeugt einen zunächst leeren Diensterbringer, der über einen Kommunikations-Thread verfügt, der auf der angegebenen Port-Nummer auf eingehende Anfragen wartet, wenn der Diensterbringer mit `portServer.start()` gestartet wird. Diese werden anhand des bei der Anmeldung eines Dienstes angegebenen Namens auf die potentiell mehreren Dienste verteilt.

Durch die in Abschnitt 3.2 entwickelte Übertragung beliebiger Tycoon-Werte ergeben sich für die Elemente der Dienste keinerlei Einschränkungen hinsichtlich des Typs von Argumenten und Funktionswerten. Für die zu übertragenden Werte steht sowohl auf Seite eines Dienstnehmers als auch auf Seite des Diensterbringers mit TL eine einheitliche Beschreibungs-Sprache der Dienste und ihrer Schnittstellen zur Verfügung. Da ein Diensterbringer potentiell mehrere Dienste erbringen kann, werden diese zur Unterscheidung benannt. Diese Benennung muß nur innerhalb des speziellen Diensterbringers eindeutig sein.

Ist ein entfernter Dienst gebunden, genügt innerhalb dessen eine einfache Ordnungszahl als Kennung der angebotenen Funktionen. Bei der Bindung wird daher im initialen Kontakt die Anzahl der Funktionen des genannten Dienstes erfragt. Ist der geforderte Dienst nicht vorhanden, kann zusätzlich so dem Dienstnehmer angezeigt werden, eine Ausnahmebehandlung auszulösen. Für einen Aufruf der entfernten Funktionen überträgt der Dienstnehmer ein TL-Tupel aus Dienstname im Diensterbringer, Funktionskennung und den Argumenten der Funktion.

```
let request = tuple
                "serviceName"
                functionNr
                arguments
              end
let result :ResultType = port.request(site request)
```

Der Funktionswert kann dann ohne weitere Ergänzungen an den wartenden Dienstnehmer rückübertragen werden. Das folgende Code-Fragment stellt bereits den vollständigen Ablauf im Kommunikations-Thread im Diensterbringer dar, der hier auch gleichzeitig der Ausführungs-Thread ist (siehe Abschnitt 4.3).

```
let promise = port.nil
let request = port.receive(portNumber promise)
let service = getService(tableOfServices request.serviceName)
let result  = execute(service request)
port.sendBack(promise result)
```

Offensichtlich kann das Dienst-Tupel `service` nicht über den Index `functionNr` angesprochen werden. Auch steht an dieser Stelle das Typsystem von Tycoon nicht zur Verfügung, da ja mit den beliebigen Argumenten beliebige Funktionen angesprochen werden müssen. Die Funktion `execute()` muß also unsicher auf die `functionNr`-te Funktion des Tupels `service` zugreifen. Typsicherheit kann an dieser Stelle *nicht* bereitgestellt werden. Über das Modul `unsafe` kann hier der Zugriff flexibel implementiert werden, und die oben benutzte Funktion `execute()` besteht im wesentlichen aus folgendem Code:

```
let execute(service :Service request :Request) = begin
  let functionArray = unsafe.typeCast(service.service :Array(Ok))
  let closure =
         arrayOp.get(functionArray  request.functionNumber+1)
  unsafe.execute(closure request.arguments :Ok)
end (* execute() *)
```

4.2 Typsichere Entfernte Prozeduraufrufe

Für gängige RPC-Dienste wie ONC-RPC und DCE-RPC ist eine externe Beschreibung der Schnittstellen der exportierten Module notwendig. Diese sogenannten *Interface Definition Languages* wie *rpcl* für den ONC-RPC und *idl* bei DCE stellen jedoch ein eigenes Typsystem bereit, das selbst gegenüber der Sprache *C* eine Einschränkung darstellt. Durch Generatoren werden diese Interfaces meist auf Versions-Nummern reduziert, die bei der Bindung verglichen werden können. Einige solcher Generator-Systeme erstellen auch leere *C*-Source-Files zur Implementation der Diensterbringer, die damit bei jedem Wechsel des Interfaces wieder *leer* erzeugt werden.

Die Beschreibung von entfernten Tycoon-Diensten erfolgt wegen der überlegenen Eigenschaften auf Typ- und Wert-Ebene in TL. Dadurch ist außerdem keine zusätzliche externe Beschreibung der Schnittstellen notwendig, was insbesondere bei der Wartung bestehender Systeme Fehlerquellen vermeidet. Auf der anderen

Seite wird die Möglichkeit ausgeschlossen, direkt entfernte Dienste in Anspruch zu nehmen, die in anderen Sprachen als TL geschrieben sind, obwohl für RPC gerne auch Interoperabilität zwischen Sprachen gefordert wird. Die sich daraus ergebenden Einschränkungen hinsichtlich Polymorphie und Typsicherheit, da zwischen den Sprachen nur der gemeinsame Teil Verwendung finden kann, sind jedoch zu groß; außerdem ist damit für mindestens eine der beteiligten Sprachen wieder eine externe Notation mit dem Problem der parallelen Wartung der Schnittstellen eingeführt.

Als Lösung steht in Tycoon die Anbindung lokaler externer Dienste als Bibliotheken in *C* oder *C++* zur Verfügung. Wurde eine *idl* verwendet, kann der Dienstnehmer-Code per *C*-Anbindung genutzt werden oder über die *C*-Callbacks eventuell sogar ein Server erstellt werden. Hierbei ergeben sich aber wieder die genannten Bruchstellen zwischen den Eigenschaften der beteiligten Sprachen.

Für Aufrufe entfernter Funktionen gilt das Paradigma, daß sie lokalen Funktionsaufrufen gleichwertig sein sollen. Bei entfernten Diensten in Form von TL-Modulen sollte dies auch für die Bindung des Modul-Wertes gelten. Das heißt insbesondere, daß nach Bindung eines entfernten Moduls statische Typsicherheit für das dienstnehmende Programm gewährleistet sein muß.

```
bind(p :port.T  Dyn Svc <:Service  name :String) :Svc
```

Durch die Möglichkeit der Laufzeit-Typrepräsentationen und Dynamischen Typen in Tycoon [Köhler 96] kann bei der Bindung an einen entfernten Dienst die Laufzeit-Typrepräsentation des Dienst-Typs vom Diensterbringer mit dem eigenen Typ verglichen und Anfragen, die eine Subtypbeziehung verletzen, zurückgewiesen werden, wie es zum Beispiel in ähnlicher Weise auch beim Napier88/RPC der Fall ist [Mira da Silva 95], wobei in Tycoon nicht nur Funktionen mit genau einem Argument vom Laufzeit-System akzeptiert werden. Außerdem verzichtet Tycoon auf die *Kommunikations-Handles*, in Napier *Capabilities*, da ohne Stub-Generierung keine ausführbare Funktion zur Verfügung steht. Nach der Bindung ist also keine Typüberprüfung oder ein sicherer Ersatz dafür mehr notwendig. Bei einem Aufruf von

```
let b = portClient.bind(p  :ServiceType  serviceName)
```

mit konkretem Port `p`, Dienstnamen `serviceName` und Dienst-Typ `ServiceType` wird an den Tycoon-Server an Adresse `p` der Name und die Typrepräsentation zu `ServiceType` übersendet. Der Server überprüft, ob ein Dienst mit Namen `serviceName` vorhanden ist und ob er der Beziehung

```
typeRep.isSubType(ServiceType localServiceType)
```

genügt. Ist dies der Fall, kann der Dienstnehmer die lokalen Stubs der Funktionen im Modul-Tupel erzeugen (dazu benötigt er insbesondere die Anzahl der Elemente), andernfalls wird *nur auf Dienstnehmer-Seite* eine Ausnahme-Bedingung ausgelöst, da die Typsicherheit verletzt wurde.

Diese Fehler stellen zwar Laufzeitfehler dar, die aber durch die gesonderten Ausnahme-Bedingungen immer genau den Typüberprüfungen der RPC-Programmteile zugeordnet werden können.

Ein Unterschied zu Aufrufen lokaler Funktionen ergibt sich durch die möglichen Netzwerk-Fehler. Eine Aufrufanforderung einer Funktion wird immer genau einmal befolgt. Der Aufruf einer entfernten Funktion

```
b.function(arguments...)
```

kann jedoch fehlschlagen. Die Reaktion des RPC-Laufzeit-Systems auf einen solchen Fehler bezeichnet man im allgemeinen mit *Fehlersemantik* bezeichnet [Schill 92]. Mögliche Varianten sind

maybe	Aufruf soweit möglich
at-most-once	Ein Aufruf-Versuch der fehlschlagen kann
at-least-once	Wiederholung bis ein mindestens Aufruf erfolgreich war
exactly-once	Absicherung, daß genau ein Aufruf erfolgte

Auf der Ebene der hier besprochenen Module `portClient` und `portServer` liegt eine *at-most-once*-Semantik vor, da genau ein Versuch der Übertragung unternommen wird und ein Fehlschlagen in Tycoon-Ausnahmen umgesetzt wird. Damit ist es dann möglich eine Entscheidung zu treffen, ob weitere Aufrufversuche zu unternehmen sind, da eventuelle Seiteneffekte im Diensterbringer dies nicht für jeden Dienst sinnvoll erscheinen lassen.

Ein Unterscheidungsmerkmal der Tycoon Client/Server-Bibliothek von anderen ist, daß es zu dem Bindungs-Vorgang keinen korrespondierenden Auflösungs-Vorgang `unbind()` gibt. Zum einen gibt es keine Zustandsbehafteten Bindungen, die den Diensterbringer dazu zwingen, parallel zum Dienstnehmer einen Zustand der Kommunikation zu führen. Zum anderen gibt es für Tycoon-Bindungen nie eine explizite Auflösung, da nicht mehr benötigte Werte in einem Adreßraum beziehungsweise Tycoon-Objektspeicher automatisch durch den Garbage-Collector entfernt werden. Eine globale Garbage-Collection ist für ein Netz aus kommunizierenden Tycoon-Prozessen nicht vorgesehen, aber es liegt im Bereich eines autonomen Diensterbringers, einen einmal angebotenen Dienst wieder aufzuheben, so daß auch hier eine explizite Aufhebung der entfernten Bindung nicht angestrebt ist.

4.3 Multi-Threaded Server

Für die Möglichkeit, die Diensterbringer in Tycoon multithreaded zu gestalten, gibt es mindestens drei Gründe ([Scott 86] und [Liskov et al. 84]) außerhalb der bei Betriebssystemen beobachteten höheren Geschwindigkeit durch Verwendung leichtgewichtiger Prozesse mit gemeinsamem Kontext statt Prozessen, die einen Kontextwechsel erforderlich machen:

- ▷ geringere Wartezeit auf Kommunikation
- ▷ Absicherung des Diensterbringers gegen Fehler in der Operation
- ▷ Absicherung des Diensterbringers gegen fehlerhafte Aufrufe durch einen Dienstnehmer

Im Tycoon-System gilt für einen Thread auch nicht uneingeschränkt, daß er in einem gemeinsamen Datenraum mit anderen Threads in einem Tycoon-Prozeß steht, da für Threads Bindungen an Daten definiert sind [Matthes, Schmidt 94a] und sie somit eigene Datenräume beschreiben.

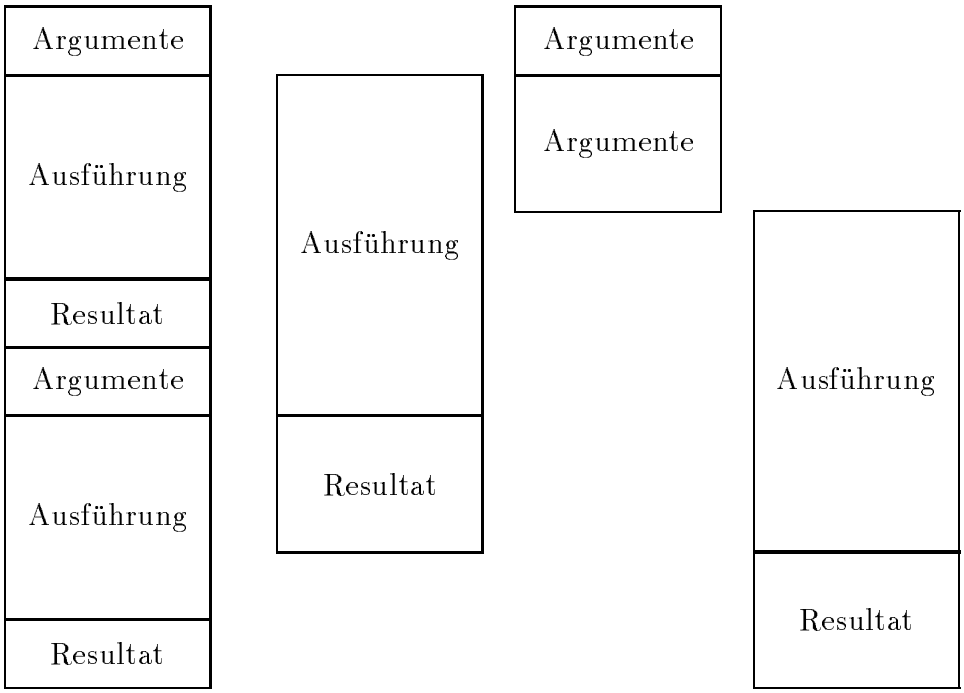


Abbildung 4.1: Multi-Threading vs. Single-Threading für Diensterbringer

Auch im Tycoon-System gilt allerdings, daß ein Diensterbringer durch einen eigenen Kommunikations-Thread die Wartezeit eines Dienstnehmers auf Kommunikation unabhängig von der eigentlichen Ausführungszeit der entfernten Funktion minimieren kann (siehe Abbildung 4.1). Dabei werden die Ausführungen der externen Anfragen an den Diensterbringer in separate Threads verlagert. Sollte durch Fehler im Dienstnehmer oder im erbrachten Dienst selbst die Ausführung fehlschlagen, wird dadurch der Diensterbringer insgesamt nicht beeinträchtigt.

Kapitel 5

Persistente Client/Server-Bindungen

Bis hierhin ist es gelungen, das Typsystem von Tycoon auf einen RPC zu übertragen und typsicher auf die entsprechenden entfernten Dienste an festgelegten Adressen zuzugreifen. Diese Adressen sind inhärent flüchtig, das heißt, sie können zu späteren Zeitpunkten und auf anderen Maschinen nicht mehr zur Verfügung stehen und eine persistente Speicherung ist nicht sinnvoll. Dennoch soll eine persistente Bindung an entfernte Dienste analog der persistenten Bindung von Modulen (wie allen Werten in Tycoon) erreicht werden.

Da es in Tycoon vorgesehen ist, daß Threads migrieren [Mathiske et al. 95], kann es zum Beispiel auch vorkommen, daß ein gesamter Diensterbringer mit den darin angemeldeten Diensten migriert. Es ist daher hier nicht sinnvoll, wie zum Beispiel beim Napier88/RPC, einen Objektspeicher zu adressieren, dessen Identität dort über die von uns als flüchtig angesehenen Eigenschaften Rechner(-name) und Objektspeicher-Datei beschrieben wird [Mira da Silva 95].

5.1 Strukturelle und algorithmische Dienstauswahl

Da die Bindung von entfernten Diensten für die Erweiterung der Bindungen aus Kapitel 4 ohne konkrete Netzadressierung, die alleine keine Persistenz ermöglicht, erfolgen muß, ist neben einer neuen Adressierungs-Architektur auch eine Dienstauswahl-Mechanismus notwendig, der über die einfache Benennung der Dienste hinausgeht. Eindeutige Benennung innerhalb eines Diensterbringers ist noch praktikabel und vielfach naheliegend, innerhalb eines lokalen Netzes oder gar weltweit kann so nicht verfahren werden [ISO/IEC JTC1/SC21 95].

Die Anforderungen eines Dienstnehmers an einen Dienst können vielfältig sein. Ein spezieller Attributtyp, wie es der Dienst-Name aus Kapitel 4 ist, kann diese

Anforderungen nicht berücksichtigen. Die Attributierung der Dienste darf keine Einschränkung bezüglich der Darstellbarkeit bestimmter anzubietender Dienste mit sich bringen.

Ein Kriterium bei der Auswahl ist sicherlich der Typ des Dienstes, der allerdings um eine Menge von Dienstattributen erweitert werden muß, die eventuell über die Lebensdauer des Dienstes nicht konstant sind. Um die Menge der möglichen Dienste nicht durch ein enges Attributierungs-Schema zu begrenzen, bietet es sich an, ähnlich den Dienst-Typen beliebige Attribut-Typen zuzulassen, die ebenfalls über die dynamische Typüberprüfung verglichen werden.

```
Dyn Attributes <:Ok
```

Beispiele sind

```
Let Attributes = Tuple
  cost     :Real
  currency :Currency
end
```

für einen kostenpflichtigen Dienst oder

```
Attributes = String
```

um eindeutige Benennung zu benutzen.

Sind für einen Dienst Dienst-Typ und Attribut-Typ zur Anfrage eines Dienstnehmers passend, das heißt, sie stehen in Subtypbeziehung, kann die Anforderung des Dienstnehmers an die Attribute des Dienstes verglichen werden. Zum Beispiel

```
attribute.cost < 4
```

für den oben beschriebenen kostenpflichtigen Dienst.

Dynamische Attribute lassen sich einfach dadurch realisieren, daß das initiale Protokoll zwischen Diensterbringer und Dienstnehmer, das in Kapitel 4 zur Typüberprüfung diente, um den Austausch des Attribut-Typs und einer Attribut-Bedingung erweitert wird. Da in Tycoon Funktionen Werte erster Klasse sind, kann die Bedingung einfach als beliebige Funktion über den Attributwert mit booleschem Ergebnis formuliert werden.

```
condition(:Attributes) :Bool
```

Für das Beispielszenario könnte man die Bedingung wie folgt formulieren,

```
let condition(attribute :Attributes) =
  if attributes.currency=DM then
    attribute.cost < 4
  else
    false
  end
```

um einen Dienst zu erhalten, der in Deutscher Mark abgerechnet wird und nicht mehr als 4 DM kostet.

Auf der Seite des Diensterbringers könnte man die Attribute konstant bei der Anmeldung des Dienstes angeben.

```
let r = server.register(...:Attributes tuple DM 3.5 end ...)
```

Die geforderte Veränderbarkeit der Attribute ergibt sich durch die Angabe einer Funktion,

```
let r = server.register(...:Attributes attributes ...)
```

die im Diensterbringer bei jeder neuen Anfrage bewertet wird. Zum Beispiel würde man mit

```
let attributes() :Attributes = begin
  let t = time.format(time.now())
  if t.hours > 8 /\ t.hours < 18 then
    tuple
      DM
      4
    end
  else
    tuple
      DM
      2.5
    end
  end (* if *)
end (* attributes() *)
```

einen Dienst erhalten, der zu angenommenen Schwerpunktzeiten tagsüber teurer ist als zur übrigen Zeit.

Nach der strukturellen Auswahl durch Dienst-Typ und Attribut-Typ, die Typsicherheit der Bindung respektive Anwendbarkeit der Attribut-Prädikate sicherstellen, kann durch den Dienstnehmer eine beliebige boolesche Funktion über

den Attribut-Typ übergeben werden, anhand derer ein Dienst ausgewählt werden kann. Konkret bedeutet dies für die Implementierung, daß auf alle Dienste, die zu Dienst-Typ und Attribut-Typ passen, die Auswahlfunktion angewendet wird. Aus den Diensten, die diese Funktion erfüllen, wird beliebig einer ausgewählt und im Dienstnehmer gebunden.

5.2 Die Adressierungshierarchie

Die in Kapitel 4 benutzten Adressen sind für persistente Bindungen nicht geeignet, da sie vom jeweiligen Betriebssystem flüchtig vergeben werden und die Lebensdauer eines Tycoon-Prozesses nicht berücksichtigen. Bei einem Neustart eines Tycoon-Servers könnte selbst auf demselben Rechner das zuletzt benutzte Socket belegt sein. Außerdem sind an Sockets flüchtige Daten gebunden, die einen Betriebssystem-Prozeß nicht überdauern können.

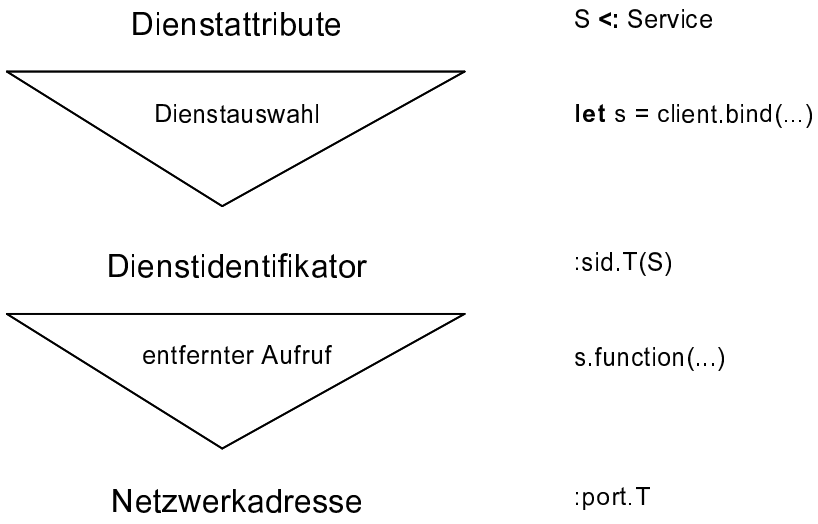


Abbildung 5.1: Adressierung-Schichten

Als Lösung bietet es sich an, analog der Unterscheidung zwischen langlebigem Tycoon-Prozeß und flüchtigem Betriebssystem-Prozeß, zwischen einer persistenten Adressierung und einer konkreten Netzwerkadresse, wie sie in Kapitel 4 verwendet wird, zu unterscheiden. Dazu ordnen wir jedem Dienst bei seiner ersten Registrierung in einem Diensterbringer einen weltweit eindeutigen *Service-Identifikator* `sid` zu. Formulieren wir diesen Identifikator typabhängig zu einem Service `service :S` mit `S <:Service` als Service-Identifikator `s :sid.T(S)`, ist bereits auf dieser Ebene Typsicherheit garantiert. Dieser Identifikator entspricht

damit zum Beispiel auch den *Capabilities*, die beim Napier88/RPC die Typsicherheit *bei jedem Aufruf* sicherstellen sollen [Mira da Silva 95].

Der in Abschnitt 5.1 beschriebene Mechanismus darf natürlich nur bei Bindung beziehungsweise Auswahl eines neuen Dienstes benutzt werden, da wir danach eine *persistente* Bindung an den einmal gewählten Dienst erreichen wollen.

Ist die Verbindung zwischen Diensterbringer und Dienstnehmer unterbrochen, benötigen sie einen Dienst der die angedeutete Abbildung von Service-Identifikator auf konkrete Netzwerkadressen leistet; bereits für die Dienstauswahl ist ein, wenn auch indirekter, Kontakt zwischen Diensterbringer und Dienstnehmer für die Auswertung der dynamischen Attribute notwendig. Dieser Abbildungsdienst kann offensichtlich *nicht* frei im Netz beweglich sein, da er zu jedem Zeitpunkt an jedem Ort, also Rechner auf dem Tycoon-Client/Server-Anwendungen laufen sollen, verfügbar sein muß.

5.3 Netzdienste zur dynamischen Dienstauffindung

Mit den in Kapitel 4 vorgestellten Modulen kann der geforderte Netzdienst erstellt werden, der es ermöglichen soll, Dienste, die nach dem beschriebenen Schema adressiert sind, aufzufinden, da dieser Dienst an festgelegten Adressen verfügbar sein muß.

Der Dienst wird analog der Struktur eines lokalen Netzwerkes in zwei Ebenen realisiert: Auf jedem in Tycoon kommunizierenden Rechner wird eine lokale Komponente installiert, die, ähnlich einem ONC-RPC-Portmapper [Santifaller 93], die konkreten Netzwerkadressen der Dienste, die in virtuellen Tycoon-Maschinen auf dem betreffenden Rechner angeboten werden, verwaltet. Innerhalb des lokalen Netzes ist dann eine einfache Komponente dafür verantwortlich, die lokal nicht zu beantwortenden Anfragen an alle bekannten lokalen Komponenten im Netz weiterzuleiten (Abbildung 5.2). Da Prozesse, die Verwaltungsaufgaben übernehmen und keinen interaktiven Benutzer aufweisen, im Unix-Umfeld gerne Daemon genannt werden und nach dem ONC-RPC Portmapper heißen die beschriebenen Komponenten *Mapping Daemon* (mad) und *Tycoon LAN Daemon* (tyLANd).

Grundsätzlich wäre für eine umfassende Implementation ein weltweiter Auskunftsdienst notwendig. Es ist allerdings wohl nicht sinnvoll, hier mit Tycoon und für Tycoon eine komplett eigene Infrastruktur zu schaffen, wenn es generische Dienste, wie zum Beispiel X500 [Tietz 89] oder vielleicht auch den *Domain Name Service* (DNS) [Liu, Albitz 92] des *Internet*, gibt. Die meisten dieser möglichen externen Dienste sind hierarchisch organisiert, so daß hier davon ausgegangen wird, daß die beschriebenen Netzdienste für Tycoon nur die notwendigen unteren zwei Ebenen einer potentiell höheren Hierarchie darstellen. Da die meisten

⋮	⋮	
1	LAN-Komponente	
0	lokale Komponente	lokale Komponente
	Dienstnehmer	Diensterbringer

Abbildung 5.2: Schichten der Netzdienste

Abläufe sich aber auf ein lokales Netz beziehen, sind diese Ebenen zur Darstellung der Technologie ausreichend. In der Implementation sind sie als Ebenen 0 und 1 bezeichnet, durch die zum Beispiel eine mögliche Suchtiefe bei der Bindung eines Dienstes angegeben wird.

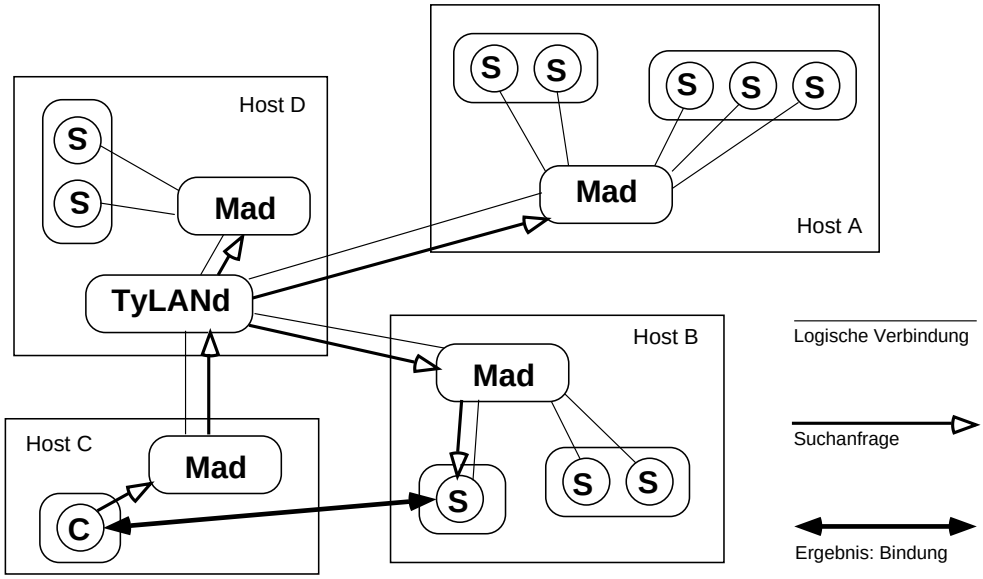


Abbildung 5.3: Auffinden eines Dienstes im LAN

Der Mapping Daemon

Die Hauptaufgabe des Mapping Daemons besteht darin, eine Liste aller auf dem betreffenden Rechner verfügbaren Dienste zu führen. Diesen Diensten stellt er

konkrete Netzwerkadressen zur Verfügung. Dazu muß ein Pool dieser Adressen, also Ports, ebenfalls verwaltet werden.

Einem neuen Dienst ordnet der Daemon bei der ersten Registrierung einen weltweit eindeutigen Service-Identifikator zu, der im folgenden dann in eine konkrete Adresse übersetzt werden kann und mit dem sich der Diensterbringer bei Neustarts und Verlagerungen beim Daemon (eventuell also auf einem anderen Rechner) wieder einträgt.

Wegen der erwarteten Dynamik des Wechsels der konkreten Adressen ist der Daemon gegen nicht mehr existente Dienste robust und paßt seinen Datenbestand sicher an geänderte Situationen an. Funktionalität zum Abmelden nicht mehr vorhandener Dienste ist so nicht mehr notwendig, da sich diese Bereinigung automatisch mit nachfolgenden Anfragen ergibt.

`register`	neuen Dienst mit Dienst-Typ, Attribut-Typ registrieren und neuen Service-Identifikator zurückgeben
`put`	Diensterbringer mit bekanntem Service-Identifikator wieder anmelden
`get`	zu Dienst-Typ, Attribut-Typ und Prädikat Dienst suchen
`translate`	Service-Identifikator in konkrete Adresse übersetzen
`getFreePort`	freie Netzwerkadresse bereitstellen

Abbildung 5.4: Funktionen des Mapping Daemons

Bei allen Auskunftsfunktionen ist eine Suchtiefe anzugeben, die die oben erwähnten Ebenen der Infrastruktur wiedergibt. Dabei ist es durchaus möglich einen Dienst im LAN zu binden, nachfolgende Suchen nach der konkreten Netzwerkadresse jedoch weiter zu fassen oder auf das System zu reduzieren, auf dem der Dienst initial gefunden worden ist. Damit ergeben sich u.a. für Abschnitt 5.4 flexible Möglichkeiten.

Den eindeutigen Service-Identifikator eines Dienstes, der der in Abschnitt 5.1 formulierten Bedingung gehorcht und im lokalen Netzwerk liegt, erhält man, wenn er existiert, durch folgenden Code:

```
let md = mad.bind()
let sid = md.get(:ServiceType  :Attributes  condition  1)
```

Der Tycoon LAN Daemon

Auf der Ebene des lokalen Netzwerkes wird genau ein Dienst auf einem festgelegten Host-Rechner installiert, bei dem alle laufenden lokalen Komponenten registriert sind. Ist eine dieser lokalen Instanzen nicht in der Lage eine Anfrage zu beantworten, kann sie sich an den LAN-Daemon binden und die Übersetzung eines Service-Identifikators oder Suche nach einem Dienst über alle *Mapping Daemons* im Netz ablaufen lassen.

Der Tycoon LAN Daemon selbst verwaltet keine Listen von Diensten sondern nur die bekannten lokalen Komponenten. Seine Schnittstelle ist der der lokalen Dienste sehr ähnlich mit dem Unterschied, daß sich *Mapping Daemons* und keine Dienste registrieren und daß nur Dienstsuche und Adreß-Übersetzung im Netz sinnvoll sind. Der LAN-Daemon verfügt auch nicht über die Unterscheidung zwischen Netzebenen, da er ausschließlich für die Verteilung der Anfragen über die Komponenten in seinem Bereich zuständig ist und Anfragen mit größerer Reichweite von den Dienstnehmern initiiert werden müßten, die sich gemäß der Ebenen-Hierarchie zunächst immer an lokale Komponenten wenden.

`register`	neue lokale Komponenten registrieren
`get`	zu Dienst-Typ, Attribut-Typ und Prädikat Dienst suchen
`translate`	Service-Identifikator in konkrete Adresse übersetzen

Abbildung 5.5: Funktionen des Tycoon LAN Daemons

5.4 Generische Client/Server-Bindungsformen

Mit den Netzdiensten ist es möglich, Client-Stubs bei Bindung eines Dienstes zu erzeugen, die in Lage sind, bei Nichtauffindung eines Diensterbringer an der bisherigen konkreten Netzwerkadresse die neue Adresse des Dienstes anhand des eindeutigen Service-Identifikators aufzulösen.

Die in Abschnitt 5.3 angedeutete Benutzung des Mapping Daemons zeigt, wie für die Bindung eines Dienstes

```
let svc = client.bind(:ServiceType :Attributes condition 1)
```

dessen Identifikator erhalten werden kann. Die nachfolgende Abbildung auf eine konkrete Netzwerkadresse scheint an dieser Stelle trivial, da die lokale Komponente und damit der Rechner, auf dem der Dienst angeboten wird, bekannt ist.

Dabei wird aber von einer zeitlichen Nähe dieser beiden Anfragen ausgegangen, die für langlebige Prozesse wie in Tycoon nicht gilt [Matthes, Schmidt 94b]. Bei der Bindung wird hier aber mit der gleichen Suchtiefe auch die Auflösung der Adresse angefordert.

```
let p :port.T = mad.translate(sid 1)
```

Danach kann bis zu einem Fehlerfall der entfernte Dienst direkt mit der konkreten Netzwerkadresse, also wie in Kapitel 4, benutzt werden. Im Fehlerfall muß im Dienstnehmer eine neue Abbildung auf die konkrete Adresse erreicht werden.

In der Tycoon-Client/Server-Bibliothek ist eine Standardfunktion enthalten, die im Sinne voller Persistenz der Bindung zu einem Service-Identifikator einen Standort im Netz solange sucht, bis er aufgefunden wird. Diese Funktion ist jedoch nicht Bestandteil des Moduls, das Dienstnehmer generiert.

Genau wie die Attribute ist auch das Verlagerungs-Verhalten dienstabhängig. Das Szenario, daß ein Dienst an einem Ort abbricht und wieder neu gestartet wird beziehungsweise im gleichen lokalen Netz auf einem anderen Rechner wieder anläuft, mag typisch sein, es kann aber nicht davon ausgegangen werden, daß dies für alle Arten von Diensten sinnvoll ist. Es sind auch Dienste denkbar, die an eine Person gebunden sind, die tagsüber im lokalen Netz einer Abteilung arbeitet und nachts in einer Netz-Domain zum externen Einwählen erreichbar ist. Dabei ist dann ein Suchen im lokalen Netz nie sinnvoll, sondern bei der Suche sind nur zwei konkrete Standorte zu berücksichtigen, wenn keine unnötige Netzbelastung bewirkt werden soll.

Zu einem Dienst kann eine Suchfunktion definiert werden, die auf die Art des Dienstes zugeschnitten ist, oder die oben beschriebene Standard-Suchfunktion eingesetzt werden (Abbildung 5.6). Damit wird in Tycoon eine persistente Bindung von Netzdiensten angeboten, aber darüber hinaus ergibt sich ein generisches Bindungsverhalten, das vom jeweiligen Dienst abhängig sein kann. Die Suchfunktion wird bei Bindung des Dienstes vom Diensterbringer in den Dienstnehmer kopiert und steht dort lokal zur Verfügung. Sie ist damit auch ein weiteres Beispiel für Code-Mobilität zwischen Tycoon-Systemen.

Die dienstabhängigen Suchfunktionen behandeln auf dieser Ebene (Module `client` und `server`) auch einen Teil der Frage der Aufrufsemantik. Abgesehen von Netzwerkfehlern nachdem ein Dienst (wieder) aufgefunden worden ist, kann je nach Verlagerung oder Ausfall jede der in Abschnitt 4.1 dargestellten Möglichkeiten abgebildet werden. Aus Effizienzgründen ist aber festgelegt, daß zunächst an der letzten erfolgreichen konkreten Netzwerkadresse ein Aufruf-Versuch unternommen wird.

Zum Abschluß soll das Suchszenario mit den dynamischen Attributen zusammengeführt werden. Die Dienstbindung stellt nur einen Spezialfall der Suche dar, bei

```
let searchFun(S <:Service  s :sid.T(S)  var at :port.T) :Ok
begin
  try
    let madOfOldSite :port.T = tuple
                                 madService.portNumber
                                 oldLocation.host
                               end
    (* trying to find service at old host *)
    let m = portClient.bind(madOfOldSite  :madService.T
                            madService.tRep  madService.name)
    at := m.translate(s 0)
  end
  while at.portNumber == ~1 do
    (* searching service in LAN *)
    try
      let m = madService.bind()
      at := m.translate(s 1)
    end (* try *)
  end (* while *)
end (* searchFun() *)
```

Abbildung 5.6: Standard-Suchfunktion des Server-Moduls

der der konkrete Dienst noch nicht bekannt ist. Statt des Identifikators wird daher eine Typbeschreibung und Anforderungen an weitere Attribute des Dienstes übergeben. Genau wie die Suchfunktion bei der Bindung auf die Dienstnehmer-Seite übertragen werden muß, migriert der Code der Attribut-Bedingung in den lokalen Mapping Daemon und eventuell, wenn die Anfrage lokal nicht befriedigt werden kann, über den LAN-Daemon in weitere Mapping Daemons im Netz. Dort kann die Funktion nur angewendet werden, wenn die vorhandenen Diensterbringer die aktuellen Attributwerte der Dienste angeben (Abbildung 5.3). Die dazu notwendige Kontaktierung aller Dienste, die Typkompatibel mit der Anfrage sind, stellt jedoch keinen großen zusätzlichen Aufwand dar, da die Verfügbarkeit eines Dienstes ohnehin überprüft werden muß, bevor seine Existenz mit Herausgabe eines Service-Identifikators bestätigt wird. Die Nachfrage der Attribute oder der reinen Existenz sind vom Standpunkt der Netzkommunikation nahezu gleichwertig, es sein denn, die Bewertung der Attribute ist für den Dienst eine aufwendige Funktion, was eine *dynamische* Bewertung dann aber wohl ohnehin erfordern würde.

Es ist in Tycoon nicht vorgesehen, einen Dienst durch eine neuere Version so zu ersetzen, daß nachfolgende Aufrufe automatische an den neuen Dienst geleitet

werden, wie dies für benannte Dienste wie den einfachen RPCs in Kapitel 4 oder auch für Napier/RPC gilt, da dies bei einer lokalen Bindung auch nicht der Fall ist. Neben dem Widerspruch zur Analogie mit der lokalen Bindung wären dann die statt der Benennung eingeführten eindeutigen Service-Identifikatoren eben keine weltweiten *Identifikatoren*, da sie irgendeinen Dienst nach belieben des Diensterbringers bereitstellen könnten und nicht mehr den Dienst, den der Dienstnehmer bereit war zu binden.

Kapitel 6

Zusammenfassung und Ausblick

Durch die Anbindung der Socket-Schnittstelle und die Verwendung der vorhandenen linearen Wertrepräsentationen ist eine schmale, portable Basis zur polymorphen, verbindungslosen Wertübertragung entstanden. Mit der Verbindungslosigkeit und durch flexible Programmierung sind Programmteile in der Rolle eines Dienstnehmers beim Senden von Werten, selbst wenn eine Rückantwort erwartet wird, voll migrierbar. Da diese wichtige Eigenschaft von anderen externen Kommunikations-Diensten nicht erreicht wird, stellt sich ihr über die Wertübertragung hinausgehender Funktionsumfang als nicht mehr verwendbar dar.

Die Typvollständigkeit der Wertübertragung wird in den Modulen `portServer` und `portClient` mit einer Typüberprüfung zur einem Tycoon-zu-Tycoon-RPC zusammengeführt. Dieser RPC gewährt mittels dynamischer Typrepräsentationen, die zum Bindungszeitpunkt zur Verfügung stehen, statische Typsicherheit nach der Bindung. Typsicherheit wird damit nicht durch Geschwindigkeits-Einbußen des RPC selbst erkauft.

Die diensterbringende Instanz kann multi-threaded genutzt werden (siehe Abschnitt 4.3).

Da für einen Diensterbringer eine konkrete Adresse zur Kommunikation notwendig ist, stellt sich hier, wenn aus Sicherheits- und Flexibilitätsgründen eine Verlagerung erlaubt werden muß, die Adressierungs-Problematik neu. Um persistente Bindungen herzustellen ist mit eindeutigen Service-Identifikatoren in einem hierarchischen Schema eine persistente Adressierungs-Ebene geschaffen worden. Durch geeignete Netzdienste kann eine solche Adresse wieder auf die konkrete Netzwerkadresse, nach gegenwärtigem Stand der Implementation mindestens für das lokale Netz, des Diensterbringers abgebildet werden. Zur Erhöhung der Flexibilität wird diese Abbildung für den Fall, daß der Diensterbringer nicht mehr am bisherigen Ort auffindbar ist, durch eine dienstabhängige Suchfunktion durchgeführt. Damit sind persistente Bindungen, bei denen die Suche fortgesetzt werden muß, bis der Diensterbringer wieder bereit und auffindbar ist, nur ein

Spezialfall dieser generischen Bindungsform, die Eigenschaften beim Verlagern des Diensterbringers nicht immer vor dem Benutzer verbergen muß.

Da für die Adressierung der Dienste die konkreten Netzwerkadressen nicht mehr zur Verfügung stehen, kann eine Auswahl der Dienste nicht mehr durch deren Typangabe und durch Namen, die nur in einem Diensterbringer Gültigkeit haben, erfolgen. Mit den Netzdiensten zur Adreßabbildung steht bereits eine geeignete Infrastruktur auch für die Auswahl eines Dienstes bei der Bindung zur Verfügung. Die Frage nach einer geeigneten Attributierung der Dienste über die Typangabe hinaus wird nicht weiter beantwortet: Durch dynamische Attribute beliebigen Tycoon-Typs und die Formulierung der Bedingungen des Dienstnehmers an den zu bindenden Dienst mit einer booleschen Funktion sind hier keinerlei Einschränkungen gegenüber der allgemeinen Ausdrucksmächtigkeit in Tycoon gemacht worden.

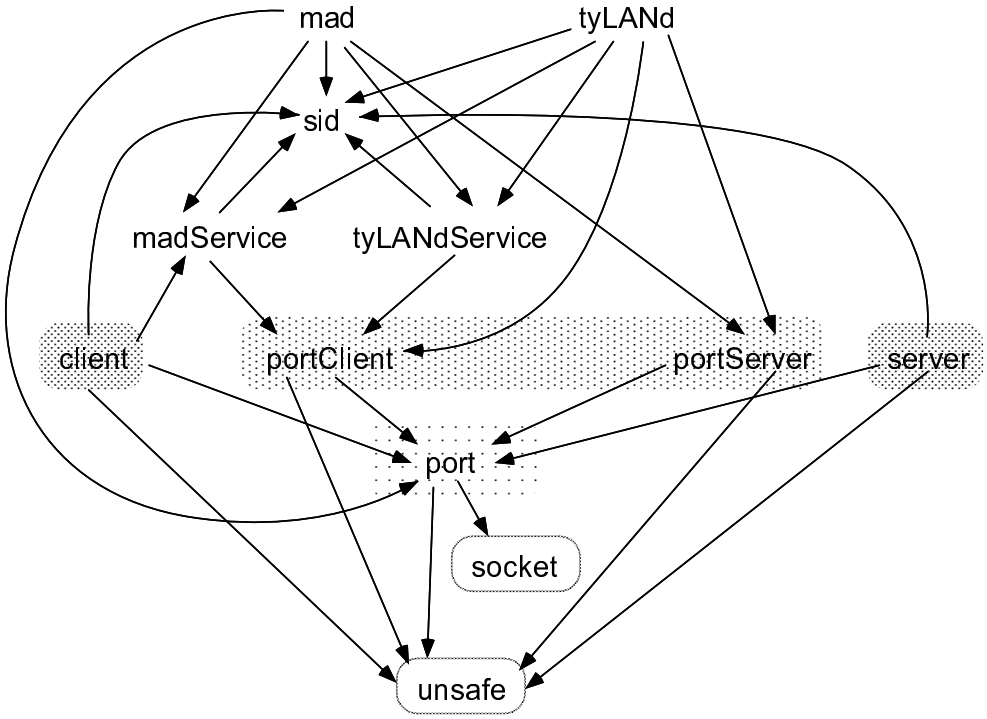


Abbildung 6.1: Modulstruktur der Client/Server-Bibliothek

In der Modulstruktur in Abbildung 6.1 ist ersichtlich, daß die Bibliothek so gegliedert ist, daß nicht immer der volle Funktionsumfang bis hin zu persistenten Bindungen genutzt werden muß. Wenn zum Beispiel weitere Netzdienste analog den in Abschnitt 5.3 entwickelten erstellt werden sollen, die sich durch ihre Ortsfestigkeit beziehungsweise explizite Ausnutzung der Netzwerkumgebung kennzeichnen, kann eine Andere Ebene gewählt werden.

Die Tycoon-Client/Server-Bibliothek stellt damit aus ihrer Architektur heraus für verschiedene Szenarien geeignete Mittel zur Verfügung und wird *nicht* durch *ergänzende* Schichten für den Programmierer wieder beherrschbar. In mit ihr erstellten Applikationen ist dadurch nur der benötigte Umfang an Funktionalität enthalten.

6.1 Erfahrungen

Die meisten Probleme der Implementation haben sich trotz des geringen Umfangs bei der portablen Anbindung der Socket-Schnittstelle ergeben, wie es bereits in Kapitel 3 angeklungen ist. An dieser Stelle ist auch die einzige Änderung an der virtuellen Tycoon-Maschine notwendig gewesen.

Oberhalb dieser relativ niedrigen Ebene stehen TL und das Tycoon-System zur Verfügung, womit volle Integration heterogener Netzumgebungen, Polymorphie, ein ausdrucksstarkes Typsystem und Persistenz bis hin zu persistenten Sicherungspunkten [Kornacker 95] auch für die Implementation vorhanden sind.

Die für das Tycoon-System häufig postulierte Nutzung bestehender Dienste und der generischen Bibliothek konnten hier erfolgreich angewendet werden. Es standen dynamische Typen und Linearisierungsfunktionalität bereit, sodaß, zusammen mit einem externen Netzwerk-Dienst, nur die eigentlichen Fragen der Netzwerkumgebung diskutiert werden mußten. Neuartige Typbeschreibungen und dazugehörige Beschreibungssprachen konnten vermieden werden.

6.2 Anschlußarbeiten

Wie in [Johannisson 95] sind auch für den hier vorgestellten RPC keine Referenz- sondern nur Wert-Parameter vorgesehen. Die dynamischen Typen in Tycoon bieten neben bereits genutzten Eigenschaften auch die Möglichkeit, Typen nach ihrer Struktur zu inspizieren, wie es in [Köhler 96] zur Visualisierung beliebiger Datenstrukturen verwendet wird. Dadurch ließen sich auch Referenz-Parameter im Dienstnehmer und Diensterbringer erkennen und in der Rücksendephase berücksichtigen. Der gegenwärtige Implementierungs-Stand enthält diese Erweiterung vor allem deshalb nicht, weil [Köhler 96] teilweise parallel zu dieser Arbeit erst entstand und dynamische Typen nicht über den gesamten Entwicklungszeitraum zur Verfügung standen. Die in [Goos 94] vorgeschlagene einfache Lösung, auch bei der Rücksendung des Funktionswertes alle Argumente wieder zurückzusenden, kann nur für die dort gegebene Hauptspeicher-Implementation erfolgreich sein, sie widerspricht aber dem Konzept des hier zugrundeliegenden persistenten Objektspeichers [Matthes et al. 95], in den die empfangenen Objekte *neu* eingefügt werden. Außerdem läuft es datenintensiven Anwendungen entgegen, wenn unnötig Werte übertragen werden.

Ist die Inspektion der Dienst-Typen in der Generierung der Bindungen einmal enthalten, kann der Dienstbegriff von Funktions-Tupeln erweitert werden. Es wäre somit möglich, einfache Werte entfernt anzubieten. Sollen diese Werte dann bereits im Diensterbringer veränderlich sein, muß zusätzlich noch an dieser Stelle der Zugriff auf Variable durch den Dienstnehmer eingeführt werden.

Für das Tycoon-System entstehen zur Zeit Bibliotheken, die generisch Komponenten mit Sicherheits-Funktionalität versehen können ([Rudloff et al. 95], [Kaß 95]). Die Client/Server-Bibliothek läßt sich damit um geeignete Authentisierungs- und Autorisierungs-Mechanismen ergänzen. Es erscheint daher nicht als notwendig, Sicherheit in die Bibliothek selbst zu integrieren, aber eine verbindende Programmier-Schicht könnte die Sichtweise der Programmierer auf verteilte Anwendungen mit Sicherheits-Mechanismen vereinfachen.

Literaturverzeichnis

Breilmann 95: Breilmann, Markus. „Portierung des Tycoon-Systems auf das Macintosh Betriebssystem“. Studienarbeit, Fachbereich Informatik, Universität Hamburg, November 1995.

Corbin 91: Corbin, J.R. *The Art of Distributed Applications.* Sun Technical Reference Library. Springer-Verlag, 1991.

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 95: 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 (to appear), 1995.

Goos 94: Goos, Ralph. „Transfer dynamischer RPC-Parameter bei datenintensiven Client/Server-Anwendungen“. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Juli 1994.

Halstead 85: Halstead, R. „Multilisp: A Language for Concurrent Symbolic Computation“. *ACM Transactions on Programming Languages and Systems*, Oktober 1985.

Hutchinson 87: Hutchinson, Norman C. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, University of Washington, September 1987.

Intel Architekture Labs 95: Intel Architekture Labs. „WinSock 2 Specifications“. http://web.jf.intel.com:80/IAL/winsock2/index.htm, 1995.

ISO/IEC JTC1/SC21 95: ISO/IEC JTC1/SC21. „ODP Trading Function, Draft International Standard 13235“, Juni 1995.

Johannisson 95: Johannisson, Nico. „Generische Programmierung typsicherer Kommunikationsdienste“. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Mai 1995.

Kaß 95: Kaß, Thomas. „Anwendungsspezifische Autorisierungsstrategien in polymorphen, persistenten Programmierumgebungen: Ein Bibliothekansatz“. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Dezember 1995.

Kerner 89: Kerner, H. *Rechnernetze nach ISO-OSI, CCITT.* Kerner, H., Wolfsgraben (Östereich), 1989.

Kiradjiev 94: Kiradjiev, P. „Dynamic optimization in CPS-based intermediate languages“. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Dezember 1994.

Köhler 96: Köhler, Hubertus. „Generische Daten- und Funktionsvisualisierung in einer persistenten Programmierumgebung“. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Februar 1996.

Kornacker 95: Kornacker, Marcel. „Persistente Sicherungspunkte für langlebige Aktivitäten in offenen Umgebungen“. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, August 1995.

Liskov et al. 84: Liskov, B., Herlihy, M., and Gilbert, L. „Limitations of Remote Procedure Call and Static Process Structure for Distributed Computing“. *Programming Methodology Group Memo 41*, September 1984. revised October 1985.

Liu, Albitz 92: Liu, Cricket and Albitz, Paul. *DNS and BIND.* O'Reilly and Associates, Inc., Oktober 1992.

Mathiske et al. 95: 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. (Also appeared as TR FIDE/95/136).

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. 95: 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 (to appear), 1995.

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 94a: Matthes, F. and Schmidt, J.W. „Persistent Threads". FIDE Technical Report FIDE/94/88, Fachbereich Informatik, Universität Hamburg, August 1994. (A shorter version of this text appeared as [MaSc94].).

Matthes, Schmidt 94b: 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, Schmidt 95: Matthes, F. and Schmidt, J.W. „Bulk Types: Built-In or Add-On?". In: Atkinson, M.P., Hrsg., *Fully Integrated Data Environments.* Springer-Verlag (to appear), 1995.

Matthes 93: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993.

Microsoft 95: Microsoft. *Readings on Microsoft Windows & WOSA.* Microsoft Press, November 1995.

Mira da Silva 95: Silva, M. Mira da. „Programmer's Manual to Napier88/RPC 2.2". Technical report, ESPRIT Basic Research Action 6309, FIDE_2, 1995.

Nelson 81: Nelson, B.J. *Remote Procedure Call.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1981.

Pour 96: Pour, Nastaran Vaziri. „Flexible Bindungstechniken für ubiquitäre Ressourcen in verteilten Anwendungen". Studienarbeit, Fachbereich Informatik, Universität Hamburg, Februar 1996.

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. (Also appeared as TR FIDE/95/138).

Santifaller 93: Santifaller, Michael. *TCP/IP und ONC/NFS in Theorie und Praxis.* Addison-Wesley Publishing Company, zweite aktualisierte und erweiterte edition, 1993.

Schill 92: Schill, Alexander. „Remote Procedure Call: Fortgeschrittene Konzepte und Systeme - Ein Überblick". *Informatik Spektrum*, 15, 1992.

Schill 93: Schill, Alexander. *DCE: Das OSF Distributed Computing Environment, Einführung und Grundlagen.* Springer-Verlag, 1993.

Scott 86: Scott, Michael L. „Language Support for Loosely-Coupled Distributed Programs“. Technical report, Department of Computer Science, The University of Rochester, Rochester, New York, 1986. Technical Report TR183.

Sloman, Kramer 88: Sloman, Morris and Kramer, Jeff. *Verteilte Systeme und Rechnernetze.* Coedition Carl Hanser Verlag, München, Wien und Prentice-Hall International Inc., London, 1988. Übersetzung Kurt Ackermann und Inge Haas-Ackermann.

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.

Walker et al. 90: Walker, Edward F., Floyd, Richard, and Neves, Paul. „Asynchronous Remote Operation Execution in Distributed Systems“. In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, Juni 1990.