Kooperation und Kommunikation in heterogenen Rechner- und Sprachumgebungen mit Perl-Linda

Dissertation
zur Erlangung des akademischen Grades eines

Doktors
der Sozial- und Wirtschaftswissenschaften
an der Wirtschaftsuniversität Wien

eingereicht bei

Erstbegutachter: o.Univ.-Prof. Dr. W.H. Janko
Zweitbegutachter: Univ.-Doz. Dr. A. Geyer-Schulz

Fachgebiet: Informationswirtschaft

von

Ing.Mag. Werner J. Schönfeldinger
Matrnr.: 8850004

Wien, am 22. Juni 1996

Danksagung

Ich möchte mich an dieser Stelle all jenen meinen Dank aussprechen, die am Zustandekommen dieser Arbeit ihren Beitrag geleistet haben:

Ich danke o.Prof. Dr. W.H.Janko für die Unterstützung und Förderung der Arbeiten der WU-Linda Group und meinen Kollegen an der Abteilung für das produktive Arbeitsklima und die Duldung der häufigen Reboots.

Meinem *Mentor* Univ.-Doz. Dr. Andreas Geyer-Schulz, der mir sowohl den anfänglichen Impuls für das Linda-Projekt gab und mir auch im Werden der Arbeit fast rund um die Uhr zur Verfügung stand, danke ich für seinen Einsatz und die Motivation beim Schreiben der Papers.

Ich danke meinen Eltern für die wohlwollende Unterstützung während des Studiums und das Drängen auf die Fertigstellung dieser Arbeit

Im weiteren Danke ich:

. . . den Mitarbeitern der WU-Linda Group, Gerold Hietler und Christoph Leithner für die Überlassung des APL- und C^{++}-Linda-Clients und die Beratung bei der Beschreibung der Funktionalität, *"Wie Gewachsen"* und Ernst Hofmann für Präsenz und die Hendln.

. . . Elfriede Pecker BA, für die vielen Frühstücke und für das sehr notwendige Korrigieren der Arbeit.

. . . dem *24er* für die wachhaltende Wirkung.

. . . G.K. für die Motivation in der Endphase.

Inhaltsverzeichnis

Zusammenfassung

Diese Arbeit stellt *Perl-Linda*, einen flexiblen Prototyp für die Implementation des Koordinations- und Kommunikationsmodells *Linda* für ein Netzwerk von Workstations vor. Der Autor zeigt zuerst Trends in der Informationstechnologie auf, insbesondere die globale Vernetzung durch das Internet, die eine Änderung der Anforderungen an Applikationen herbeiführen. Durch die immer weitere Verbreitung des Betriebsystems UNIX stellen Netzwerke von UNIX-Workstations eine preisgünstige und in einer Vielzahl von Organisationen verfügbare Variante zu Super- und Parallelrechnern dar.

Aufbauend auf den bereits zu Linda existierenden Arbeiten und Implementationen, wird Perl-Linda, ein Prototyp geschaffen, der auf Client/Server-Basis eine einheitliche Schnittstelle für die Erweiterung von Programmiersprachen um den Linda-Befehlssatz implementiert. Auf Basis dieser Schnittstelle werden die für einige Programmiersprachen wie *C*, *C++*, Perl und APL, erstellten Client-APIs beschrieben und deren Funktion anhand von Beispielen erläutert. Die Entwicklungsschritte zum Design und zur Entwicklung einer Linda-Applikation werden beschrieben und an Beispielen durchgeführt.

Abschließend wird der Einsatz von Linda für die Erweiterung der Funktionalität des World Wide Web-Systems (WWW) für die Schaffung von WWW-basierten Transaktionssystemen gezeigt. Es werden mit Linda und WWW realisierte Informationssysteme als Beispielapplikationen vorgestellt und erläutert.

1 Motivation

> *"What are you able to build with your blocks?*
> *Castles and palaces, temples and docks."*
> [Stevenson, 1885]

In der römischen Baukunst verfolgten die Baumeister das Ziel, immer weiter spannende Kuppeln unter dem herrschenden technischen Fortschritt zu bauen. Viele dieser Bauwerke, deren gewaltigstes das Panteon in Rom ist, können wir noch heute bewundern. Doch wurden all diese Bauten nicht berechnet, sondern rein aufgrund handwerklicher Erfahrung gebaut.

Beim Konzipieren und Erstellen von Software steht der Designer vor ähnlichen Problemen wie die antiken Baumeister. Zu den wenigsten Problemen gibt es zufriedenstellende Patentlösungen. Die Verwendung von Standardsoftware kann in der Regel auch nur einen allgemeinen Problemkreis abdecken. Somit ist der Designer und Programmierer auf die handwerkliche Erfahrung angewiesen um eine softwaretechnische Lösung des Problems in Anbetracht und Ausnutzung der jeweiligen technischen Möglichkeiten zu erzielen. Er versucht, eine möglichst weite *Kuppel* für die Lösung des jeweiligen Problems zu spannen. Bei diesem Prozeß, sollte er versuchen die durch die Entwicklung zur Verfügung stehenden Bausteine möglichst effizient zu einem Ganzen zusammenzusetzen.

2 Einführung

In der Informationstechnologie haben sich Trends seit jeher schneller entwickelt und sind auch kurzlebiger als in den meisten anderen Sparten. Einen großen Anteil an diesem Prozess hat die Entwicklung von Hardware, Betriebsystemen, Netzwerktechnologie und deren Interaktion. Im folgenden Abschnitt wird ein Überblick über die Entwicklung und Prognose dieser Komponenten gegeben. Im weiteren werden Organisations- und Kooperationsformen in Software und deren Einsatzgebiete erläutert. Der Abschnitt schließt mit einem Überblick über diese Arbeit ab.

2.1 Technische Entwicklung

> *"The software industry is in the midst of a revolution with developer limitations decreasing and user choice becoming increasingly a factor"*
> [Lewis *et al.*, 1995, S. 21].

Die fallenden Restriktionen in der Entwicklung von Software werden durch die technische Weiterentwicklung von Hardware, Betriebssystemen und Vernetzung unterstützt. Viele neue Applikationen sind nur durch die Leistungs- und Funktionalitätssteigerungen in diesen Bereichen möglich. Um zu verstehen, wo sich die Softwaretechnologie im Augenblick befindet, und wohin sie sich bewegen wird, ist es notwendig, sich die Leistungsentwicklung in den einzelnen Bereichen anzusehen.

2.1.1 Hardware: Prozessor, Hauptspeicher, Massenspeicher und Netzwerk

Durch die Einführung und Verbreitung des Intel-Pentiumchip ist die Kluft zwischen Microprozessorrechnern und Mini- bzw. Großrechnern deutlich kleiner geworden. Die rapide Entwicklung in der Halbleitertechnologie und in der Prozessorarchitektur haben zu einer überproportionalen Leistungsteigerung seit den 50er-Jahren geführt. Zu dieser Entwicklung führten zwei maßgebende Faktoren [Hennessy and Jouppi, 1991, vgl. S 19]:

1. Die dramatische Erhöhung der Anzahl der Transistoren auf einem Chip.

2. Der Fortschritt in der Chip-Architektur, insbesondere die Nutzung von RISC (Reduced Instruction-Set Computing), Pipelining und Caching.

Mikroprozessoren haben extrem kurze Lebenszyklen und können rasch auf die Weiterentwicklungen in der VLSI (Very Large Scale Integration) Technologie adaptiert werden. Dies führt zu einer steigenden Performance der Prozessoren bei sinkenden Kosten in der Produktion. In Abbildung 1 wird die CPU-Performanceentwicklung verschiedener Rechnertypen seit 1965 verglichen.

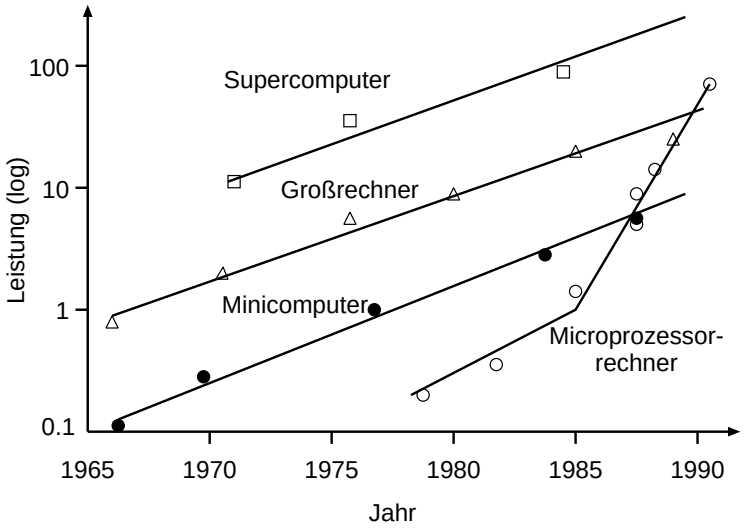

Abbildung 1: Vergleich der CPU-Performanceentwicklung, Quelle: [Hennessy and Jouppi, 1991, S. 19]

Aus der Grafik läßt sich erkennen, daß die CPU-Performance der Rechner insgesamt ansteigt. Die Performance der Mikroprozessorrechner steigt gegenüber allen anderen Sparten überproportional an.

Während bei Prozessoren durch Nutzung von Pipelinig and Caching die Geschwindigkeit überproportional zum Anstieg der rein physischen Zugriffsgeschwindigkeit gesteigert werden kann, stehen diese Möglichkeiten bei Speicherbausteinen nicht zur Verfügung. Daher entwickelt sich mit steigender Performance der Prozessoren zusehends die Performance des Arbeitsspeichers als Engpaß im System. Dieses Engpaßproblem kann durch die Einführung von Caches zwischen Prozessor und Arbeitsspeicher gemildert werden. [Hennessy and Jouppi, 1991, vgl. S. 23].

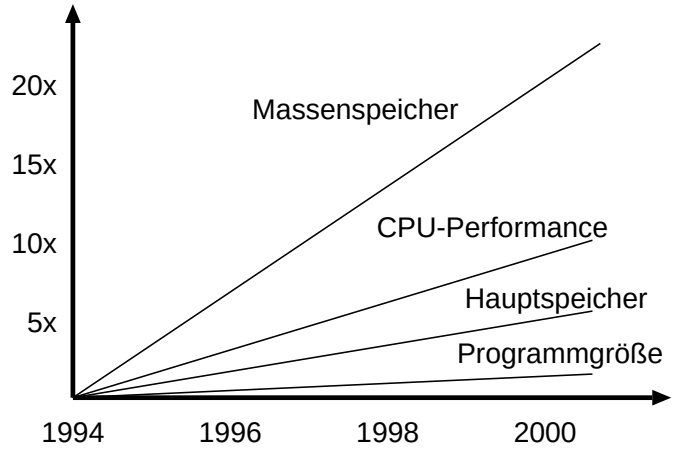

Abbildung 2: Entwicklung der Workstation-Ausstattung, Quelle: [Lewis, 1994, S. 61]

Die zukünftige Entwicklung der Ausstattung einer durchschnittlichen Einprozessor-Workstation, bei gleichbleibenden Kosten, ist in Abbildung 2 dargestellt. Sie prognostiziert, daß die Festplattenausstattung schneller als alles andere auf das 20-fache des Stands von 1994, ca. 340 MB, auf 7 GB steigen wird. Die Hauptspeicherausstattung wird von ca. 8 MB auf 64 MB steigen. Drei Faktoren indizieren diesen Trend:

- Größerer Ressourcenbedarf der Programme, speziell im Bereich Multimedia.
- Die steigende Mehrbenutzerfähigkeit der Betriebsysteme bedingt, daß mehr Daten und Programme in separaten Bereichen im Hauptspeicher gehalten werden.
- Da die Zugriffsgeschwindigkeit der Massenspeicher unterproportional zur Prozessorgeschwindigkeit und zur Zugriffsgeschwindigkeit des Massenspeichers steigt, ist es für die Performancesteigerung des Gesamtsystems notwendig, immer größere Caches zwischen Platte und Hauptspeicher bzw. Prozessor für das Caching der Plattenzugriffe zwischenzuschalten [Hennessy and Jouppi, 1991, vgl. S. 27].

Die Prozessorgeschwindikeit wird sich von ca. 80 SPECmarks auf 800 SPECmarks erhöhen. Die Steigerung des Adressbereichs der Programme wird von 32 auf 42 Bit prognostiziert.

Neben den Einprozessorrechnern werden sich auch Mikroprozessorrechner mit mehreren Prozessoren (4-20) durchsetzen. Diese Maschinen haben folgende Einsatzgebiete [Hennessy and Jouppi, 1991, vgl. S. 24]:

- Sie sind die ideale Alternative zu Einprozessorrechner, die mit *Timesharing* arbeiten – jeder Benutzer kann einen eigenen Prozessor zugewiesen bekommen. Moderne Betriebssysteme wie z.B. Digital UNIX, Windows NT etc. unterstützen Mehrprozessorrechner schon auf Betriebssystemebene.
- Sie eignen sich gut als Transaction Processing Systems, da sie den parallelismus zwischen den Transaktionen nutzen.
- In der Wissenschaft und in der Technik sind diese Rechner für rechenintensive Anwendungen geeignet.

Netzwerke als verbindende Komponente zwischen den einzelnen Rechnern haben in Universitätsrechenzentren und Großbetrieben zum Teil schon den Großrechner ersetzt. Ein entscheidender Faktor für den Einsatz von Netzwerken ist, daß diese eine *"usable" Speed* erreicht haben [Lewis, 1994, vgl. S. 62f.]. Die Geschwindigkeit der Netze bestimmt auch die Typen von Applikationen, die auf diesen Netzen einsetzbar sind. Eine Übersicht über die verwendete Technologie und die Bandbreite ist in Tabelle 1 dargestellt.

Technologie	Bits/Sek.
Telefon/Modem	10 Kbps
ISDN	100 Kbps – 1 Mbps
MBone, MPEG, Ethernet	1 Mbps – 10 Mbps
Asynchonous Transfer Mode (ATM)	10 Mbps – 1 Gbps

Tabelle 1: Übersicht über Netzwerke und Bandbreite, Quelle: [Lewis, 1994, vgl. S. 63]

Ein treibender Faktor für die Dezentralisierung ist die Tatsache, daß das Preis/Leistungsverhältnis für Netzwerke schneller gefallen ist als für Computer [Lewis, 1994, vgl. S. 62]. Im universitären Bereich und in Unternehmen sind heute Ethernet-Netze mit einer Übertragungsrate von 10 Mbps Standard. Diese Übertragungsrate läßt die Bandbreite bei einer größeren Anzahl von Rechnern und dem Einsatz kommunikationsintensiver Software schnell zum Engpaß werden. Mit dem Umstieg auf ATM, der jetzt noch vergleichsweise teuer ist, kann jedoch auch dieser Engpaß entschärft werden.

2.1.2 Betriebssysteme

Direkt auf der Hardware setzen die Betriebssysteme auf und schaffen die Grundlage für die darauf laufende Software. Wir sehen uns in der heutigen Zeit einer Vielzahl von Betriebssystemen gegenüber. Die Anforderungen an die heutigen Betriebssysteme sind Mehrbenutzerfähigkeit, grafische Benutzeroberfläche, Multi-Tasking, Unterstützung von Netzwerk, Speicher- und Dateimanagement und Systemsicherheit. So werden z.B. die Hauptmerkmale des von AT&T neu entwickelten Betriebssystems *Plan 9* beschrieben [Korenzeniowski, 1995, vgl. S. 116]:

- *kleine Betriebsystemgröße*, das Core-System umfaßt einen Kernel der auf einer Boot-Disk platz finden sollte, einen Editor und ein einfaches Netzwerk-Interface.
- *modularer Aufbau*, bestehend aus Client (Benutzerinterface), CPU-Server (Hardware-Interface) und einem Dateisystem-Server.
- *vielseitige Hardwareunterstützung*, Unterstützung der Plattformen Intel x86, Motorola 680x0, Sun SPARC und MIPS.
- *erweiterte Systemsicherheit*, inklusive Authentizierung nach dem Kerberos-Schema.

In [Halfill, 1996] wurden die Marktanteile der verschiedenen Betriebssysteme miteinander verglichen (siehe Tabelle 2). Der Vergleich zeigt, daß die 1992 durchgeführten Prognosen nicht eintrafen und daß sich jene Betriebssysteme durchsetzten, die die oben genannten Anforderungen nur teilweise erfüllen.

Betriebssystem	Prognose92 für 1996	Ist 1995	Prognose95 1999
DOS	40%	6%	0%
Windows 3.1	0%	52%	0%
Windows 95	0%	22%	48%
Windows NT	37%	1%	41%
Unix	7%	2%	2%
Sonstige	16%	17%	9%

Tabelle 2: Vergleich der Betriebssystem-Marktanteile, Quelle: [Halfill, 1996]

Die Prognose für das Jahr 1999 sagt voraus, daß über ein Drittel des Marktes von mehrbenutzerfähigen Betriebsystemen dominiert wird, die als Grundlage für Server-Dienste in einem Netzwerk von Rechnern einsetzbar sind.

Zwischen den Betriebssystemen haben sich auch Erweiterungen und Module etabliert, die einen Datentausch zwischen den Systemen ermöglichen. Als Beispiele sind hier *Remote Procedure Calls* (Windows NT, UNIX), *Network File System* (Windows NT, UNIX, Windows 95) und *Samba* (eine LAN-Manageremulation auf Basis des SMB-Protokolls für Windows NT, UNIX und Windows95) zu nennen. Der Einsatz dieser Module und Erweiterungen ermöglicht den Austausch von Dateien und den gegenseitigen Aufruf von Programmen.

Speziell für PC-basierte Betriebssysteme wie Windows 95 und Windows NT gibt auf der Netzwerkseite einige Erweiterungen TCP/IP-basierter Servicedienste, die früher nur auf UNIX und auf Großrechnersystemen verfügbar waren wie Nameserver (`named`, `bootpd`), Druckerspooler (`lpd`), `ftp`-Server, World Wide Web-Server etc.

Trotz der schlechten Prognosen wird sich UNIX als Betriebssystem gerade in Bereichen, wo belastbare Server benötigt werden, durchsetzen bzw. seine bisherige Position halten. Gerade durch die Verfügbarkeit von UNIX (Linux, NetBSD) für PC-Workstations, stellt UNIX eine kostengünstige Variante eines Serverbetriebssystems für PC-Plattformen dar. Indikatoren dafür sind die geringen Kosten in der Anschaffung – Linux und NetBSD sind frei verfügbar – und die steigende Produktion kommerzieller Software für diese Systeme. So gibt es mit z.B. Star Office, ein komplettes Standardsoftware System, daß in Kürze auf der Plattform Linux verfügbar sein wird. Die Verfügbarkeit von Emulatoren für Betriebssysteme wie `DOS` und `Windows` auf einigen UNIX-Derivaten erweitert die Palette der unter UNIX lauffähigen Programme.

2.1.3 Software

Neben der Hardware und Betriebssystemen ist die Software ein entscheidender Faktor für die Entwicklung der Informationstechnologie. Software kann jedoch nicht unabhängig von Hardware und Betriebssystem gesehen werden, da diese den Rahmen für die nutzbaren Möglichkeiten festlegen.

Objektorientiertes Design, objektorientierte Programmierung und *Client/Server*-Architektur sind ohne Zweifel Gebiete, die auf die Erstellung von Software in der

heutigen Zeit entscheidenden Einfluß haben.

> *"Object-oriented design is the method that leads us to an object-oriented decomposition; object-oriented design defines a notation and process for constructing complex software systems, and offers a rich set of logical and physical models [. . .]."*
> [Booch, 1991, S.23]

Die Verwendung objektorientierter Methoden führt den Software-Entwickler zu einer objektorientierten Sicht des Problems. Objektorientiertes Design stellt eine Notation zur Verfügung, die es ermöglicht die Komplexität des Gesamtproblems zu verringern.

Der Einsatz der Client/Server-Technologie bestimmt ein Kommunikations und Kooperationsschema, durch welches die einzelnen Teile der Software zusammenwirken. Der Server stellt auf der einen Seite zentral die Servicedienste zur Verfügung, der Client arbeitet auf der anderen Seite dezentral und nimmt die Dienste des Servers in Anspruch.

Mit dem Aufkommen von Client/Server Computing verbanden viele Unternehmen die Bestrebungen, eine Veränderung in der Struktur ihrer EDV-Ausstattung vorzunehmen. Zu den meist praktizierten Techniken zählen [Orfali *et al.*, 1994, S.8]:

Downsizing:
Beim Prozess des Downsizing wird versucht, die im Betrieb verwendeten Informationssysteme und die Geschäftssoftware von einer Grorechner-basierten in eine PC-basierte Lösung überzuführen.

Die Bearbeitung der Daten erfolgt auf lokalen Desktop-Maschinen, die Daten werden mehr oder weniger zentral gehalten. Durch den Einsatz der Client/Server Technologie läßt sich auf diesem Weg die steigende Rechenleistung der PC-Clients mit den Vorteilen zentraler Datenhaltung mit Mehrbenutzerzugriff verbinden.

Upsizing:
Im Gegensatz zum Downsizing ist die Ausgangssituation beim Upsizing eine Ansammlung von Einzel-PCs, die eventuell noch durch ein Netzwerk verbunden sind. Upsizing verfolgt hier das Ziel, die PCs mit ihrer Vernetzung zu kooperativer Arbeit zu nutzen. Dies erfolgt einerseits durch das

Ressourcensharing teurer Peripheriegeräte wie Scanner, Laserdrucker etc. auf dem Netz, andererseits durch das Verteilen verschiedener kooperativer Dienste auf die vernetzten Computer.

Rightsizing:
Beim Rightsizing wird nicht, wie bei den beiden anderen Methoden, die grundsätzliche Hardwareinfrastruktur geändert, sondern ein sich bereits in einer bestimmten Konfiguration befindliches System in seiner Funktionalität abgeändert. Dienste und Aufgaben werden auf den am besten passenden Rechner verlegt, Betriebssysteme umgeschichtet, um die Ablaufeffizienz und die Verfügbarkeit des Gesamtsystems zu erhöhen.

Der Einsatz der Client/Server-Technologie und des objektorientierten Designs beeinflußen sich bei der Erstellung der Software nicht, im Gegenteil, es können dadurch in das durch Client/Server bestimmte Kommunikationsschema die Vorteile objektorientierten Designs einfließen. Der Einsatz objektorientierter Methoden bringt dem Entwickler folgende Vorteile [Lewis, 1994, vgl. S. 62]:

- Durch die iterative Vorgangsweise beim Erstellen und Testen von Objekten, kann er die Software schrittweise in mehreren Phasen testen. Dies führt zu kürzeren Entwicklungszeiten. Durch diese kann er mit den immer geringer werdenden Produktlebenszyklen für Software mithalten.
- Durch die Wiederverwendung von bereits erprobten Objekten und Modulen erübrigt sich das Testen bereits vorhandener Komponenten. Fertige Modulbibliotheken können entwickelt bzw. zugekauft werden.
- Durch den modularen Aufbau der Software verkürzen sich die Wartungsarbeiten nach der Inbetriebnahme der Software, da nicht die ganze Anwendung monolithisch sondern immer nur kleine Module gewartet werden müssen.

Neben der Wiederverwendbarkeit nennt [Meyer, 1996] auch noch die Möglichkeit zur Bildung abstrakter Objekte als weiteren Vorteil für den Entwickler. Durch solche *"Programs with Holes"* lassen sich einerseits Schablonen für Standardobjekte erstellen, die als Checklisten für Objekte dienen können, andererseits bleiben die abstrakt erstellten Objekte in jede Richtung erweiterbar und adaptierbar.

Java^{tm}, eine von Sun Microsystems entwickelte objektorientierte Sprache, stellt ein gutes Beispiel für das oben Beschriebene dar. Das in Java geschriebene Programm wird in einen Zwischencode (Bytecode) übersetzt, in dem es systemunabhängig von einem Java-Interpreter ausgeführt werden kann. So ein Interpreter ist in verschiedenen Anwendungen wie z.B. im Netscape World Wide Web-Browser integriert. Es existieren für Java eine Reihe von standardisierten Klassenbibliotheken für die Implementation von Graphical User Interfaces, Netzwerkfunktionalität, Multi-Media etc. Der ausführbare Client-Code wird dabei über das Netz geholt und mit den Standardbibliotheken für das lokale Betriebssystem verknüpft. Diese Standard-Klassen stellen eine Basisfunktionalität bezüglich GUI, Ton und Bildverarbeitung etc. zur Verfügung, die durch den Benutzer/Entwickler erweitert werden kann.

Gerade für Bereiche wie Multi-Media, Computer Supported Cooperative Work, Mobile Computing, Software Agents ist eine modulare Struktur der Software, wie sie durch Client/Server und objektorientierte Designtechniken und Sprachen erreicht werden kann, für die Anpassung an die zur Verfügung stehende Hardware und die Erweiterbarkeit der Applikation von entscheidender Bedeutung für die zukünftige Entwicklung.

2.1.4 Vernetzung

In [Coulouris and Dollimore, 1988, vgl. S. 8] wird die Verfügbarkeit von leistungsfähiger Vernetzung zu moderaten Kosten als Faktor für die Verbreitung von verteilten Systemen genannt. In der Tat sind die Verfügbarkeit schneller Netzwerke und die verschiedenen Dienste der Betriebssyteme ein Treibfaktor für die Entwicklung von Client/Server-Applikationen und verteilten Systemen. Die Realisierung und der Einsatz dieser Arten von Anwendungen sind entscheidend von der zur Verfügung stehenden Bandbreite des verbindenden Netzwerkes abhängig. Abbildung 3 zeigt die Bandbreitenbereiche, welche die verschiedenen Anwendungen benötigen.

Die Grafik zeigt die Übertragungsleistungen der drei Übertragungstechnologien *Modem, Ethernet* und *ATM* und die Applikationen, welche im Bereich dieser Technologie eingesetzt werden können. Dienste die in Client/Server-Applikationen eingesetzt werden, wie Graphical User Interfaces, Peripheral Sharing, Remote File Systems, können vom heutigen Netzstandard (1 Mbps) rein technisch abgedeckt werden. Problematisch ist jedoch die gleichzeitige Nutzung der Dienste durch mehrere Benutzer, da sich diese die zur Verfügung stehende

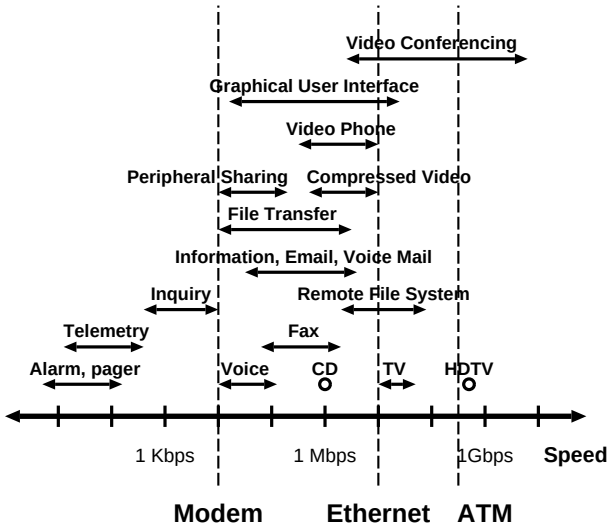

Abbildung 3: Bandbreitenverbrauch diverser Anwendungen, Quelle: [Forman and Zahorjan, 1994, S. 40]

Bandbreite teilen müssen.

Die heute meist verbreitete Technologie ist Ethernet. Die basiert auf dem für Busnetzwerke in ISO 8802-3 genormten Protokoll CSMA/CD. Dieses entspricht den beiden untersten Schichten des ISO/OSI Referenzmodells. Das auf diesem Protokoll aufsetzende *Internet Protocol* (IP) bietet die Basis für die in den meisten Client/Server-Applikationen verwendeten Übertragungsprotokolle `tcp` (Transmission Control Protocol) und `udp` (User Datagram Protocol). IP stellt weiters noch die Möglichkeit des Anschlußes und der Teilnahme am *weltweiten Internet* zur Verfügung.

Das Internet, welches sich aus dem ARPANET entwickelte, setzte TCP/IP seit 1983 als offizielles Protokoll ein. Durch die Einbindung mehrerer Netzwerke, u.a. dem NSFnet der National Science Foundation, das alle Supercomputer Amerikas verbindet, wurde die Netzwerkinfrastruktur schrittweise verbessert. Abbildung 4 zeigt die Zunahme der am Internet angeschlossenen System über die Jahre 1989 bis 1994. Der stark steigendende Trend setzte sich in den Jahren 1995 und 1996 weiter fort. Im August 1995 wurden beim *Internet Host Count* bereits $3,902,980$ Rechner am Netz gezählt. Dies entspricht einer Steigerung von 24% seit der letzten Zählung.

Die Mehrzahl der am Internet angeschlossenen Rechner liefen in den Jahren vor

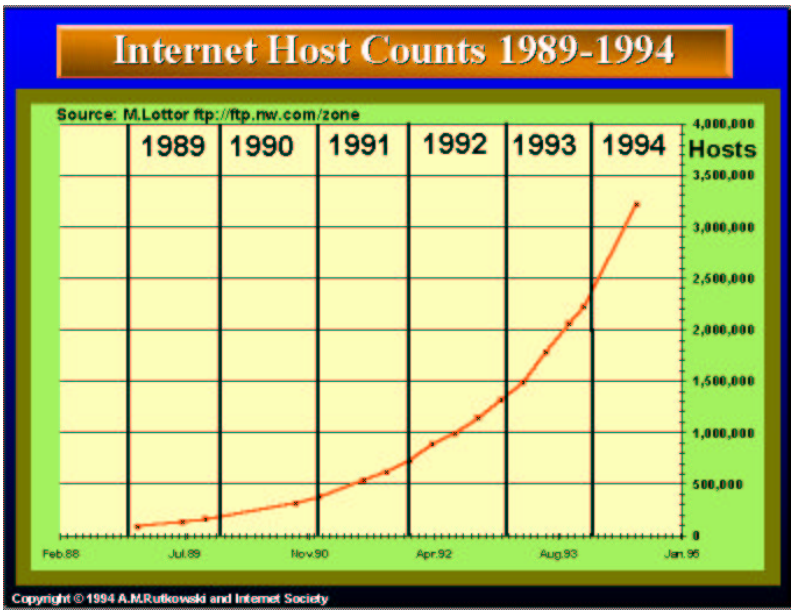

Abbildung 4: Zunahme der am Internet angeschlossenen Systeme, Quelle: *Internet Society*

1992 unter dem Betriebsystem UNIX, welches außer der Netzwerkfunktionalität auch die Möglichkeit zur Einrichtung von Diensten an `sockets` des Rechner erlaubte. Ein Socket ist ein Kommunikationsendpunkt, der durch die *Portnummer* bestimmt ist. An diesen Socket können Dienste gebunden werden, welche von anderen Rechnern aus genutzt werden können. Einer dieser Dienste, das World Wide Web [Berners-Lee *et al.*, 1995], hatte großen Anteil an der Popularisierung und Verbeitung der Nutzung des Internet. Man kann den überproportionalen Anstieg der Internet-Hosts ab 1993 in Abbildung 4 erkennen. Da andere Betriebssysteme heute auch schon TCP-Erweiterungen, wie z.B. `WinSock` für Windows-Systeme, unterstützen, stehen Alternativen zu UNIX-Systemen als Internet-Server zur Verfügung.

Durch die steigende Anzahl von Internet-Benutzern kommt es auch zu einer Erhöhung des Netzverkehrs. Überproportional nehmen damit auch die Benutzer am Internet zu, da man davon ausgehen muß, daß auf jedem Hostrechner mindestens ein Benutzer arbeitet. Da bei der Mehrzahl der am Internet angeschlossenen Rechner mehr als ein Benutzer arbeiten, kann eine stark überproportionale Benutzersteigerung angenommen werden. Als Beispiel aus dem universitären Bereich kann hier das an der Wirtschaftsuniversität Wien befindliche *PowerNet* genannt werden, welches mit 6 UNIX-Servern die Infrastruktur des Internets für ca. 12000

Benutzer zugänglich macht.

Die Erhöhung des Netzverkehrs wird vor allem durch die vermehrte Nutzung von nachrichtenintensiven interaktiven Diensten verstärkt. Abbildung 5 zeigt den gemessenen Netzwerkverkehr auf der NFSnet Hauptleitung.

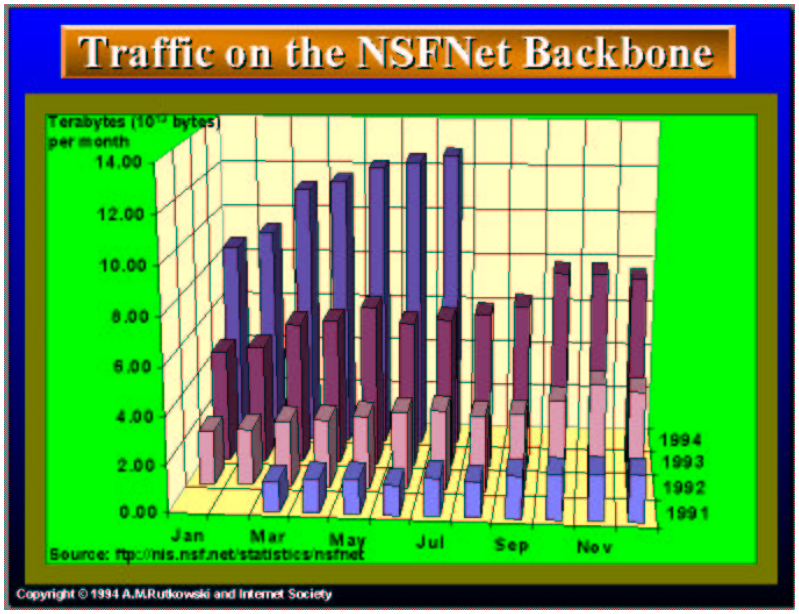

Abbildung 5: Verkehrstatistik am Internet Hauptknoten, Quelle: *Internet Society*

Die Grafik zeigt u.a. sehr deutlich die überproportionale Steigerung in den Jahren 1993 und 1994 durch die Verbreitung des World Wide Web. Gerade diese Technologie trug durch die Möglichkeit der Übertragung von Bildern im großen Maße zur Erhöhung des Netzwerkverkehrs bei. Nicht berücksichtigt sind in dieser Statistik die Steigerungen im lokalen Netzwerkverkehr, der durch den Einsatz der Internetdienste im lokalen Bereich entsteht.

Durch den Umstieg von textbasierten auf multimediale Dienste kam es auch zu einer Verlagerung der Belastung zwischen den Diensten (siehe Abbildung 6). War in den Jahren 90–91 der Dateitransfer (`ftp`) noch der dominierende Dienst am Internet so verlagert sich seine Bedeutung hin zu den anderen Informationsdiensten. Das langsamere Absinken des Anteils von Dateitransfers ist durch den Umstand zu erklären, daß Informationsdienste wie z.B. WWW auch ein Front End für den Dateitransfer bieten und dieser Dienst durch die im WWW anders darstellbare Information einfacher zu bedienen und daher attraktiver ist. Es läßt sich der Trend in

der Zunahme der `tcp/udp`-basierten Dienste erkennen, von denen die meisten auf Client/Server-Basis arbeiten.

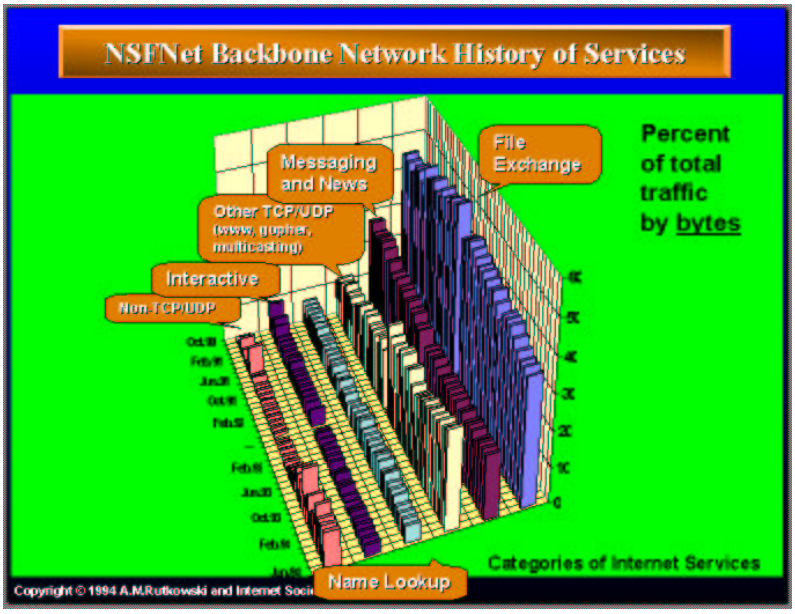

Abbildung 6: Anteil der verschiedenen Internetdienste am Netzverkehr, Quelle: *Internet Society*

Die Verfügbarkeit einer Vielzahl von teilweise interaktiven Diensten auf dem Netzwerk stellt an die Softwareentwickler andere Anforderungen. Zu diesen zählen z.B. der gleichzeitige Zugriff von vielen Benutzern auf eine Anwendung und die Verwendung eines *unsicheren* Netzwerkes zwischen Client und Server. Der steigende Anteil kommerzieller Anbieter am Internet hat für die Softwarefirmen einen neuen Teilmarkt für die Softewareentwicklung geschaffen. Forderungen auf diesem Teilmarkt sind globale Erreichbarkeit, Sicherheit und Exklusivität in der Benutzung der Anwendungen. Von Benutzerseite wird zusätzlich noch die Forderung nach moderaten Antwortzeiten und einfacher Bedienung gestellt.

Diese Forderungen sind für eine eingeschränkte Zahl von Benutzern an sich zu erfüllen. Bei den potentiell am Internet vorhandenen Benutzern, die den Dienst bzw. die Anwendung benutzen könnten, müssen dialogbasierte Anwendungen in einem anderen Licht gesehen werden. Ein deutliches Beispiel ist die sofortige Überlastung von diversen Bulletin Board Systemem und `ftp`-Servern bei Bekanntwerden der Lagerung von *interessanten* Daten auf den jeweiligen Rechnern. Dies führt meistens zum vollkommenen Zusammenbruch der Rechner und

zur Aussetzung der Dienste. Ein österreichisches Beispiel für diesen Mechanismus ist die Überlastung des universitätsweiten Bibliothekssystems BIBOS durch die globale zur Verfügungstellung der Dienste über ein WWW-Gateway [Barta and Hauswirth, 1995].

2.2 Organisation und Kooperation

In den vorigen Abschnitten wurde eine Übersicht über Trends in Hard-, Software und Netzwerktechnologie gegeben. Als Haupttrends ergaben sich:

- Schnellere CPUs, mehr Hauptspeicher und mehr Massenspeicher in den Rechnern.
- Mehrprozessorrechner mit 4-20 Prozessoren.
- Leistungsfähigere Netzwerke, die neben den normalen Netzwerkdiensten auch die Übertragung von Multi-Media und interaktive Video-Übertragung ermöglichen.
- Modulare Anwendungen mit objektorientiertem Design auf Client/Server-Basis.
- Steigender Bedarf an Bandbreite im internationalen Netzverkehr durch steigende Benutzerzahl und kommunikationsintensivere Anwendungen.

Diese Trends ziehen natürlich auch eine Umschichtung der internen Kommunikations- und Kooperationsstrukturen mit sich, mit der die zu Verfügung stehenden Ressourcen können genutzt werden. Das verbindende Netzwerk ist schnell genug, um die Bearbeitung von Teilprozessen auslagern und an andere Systeme vergeben zu können. In der EDV von großen Organisationen gibt es speziell in der Nacht eine große Anzahl von unausgelasteten Rechnern, die für die Verteilung rechenintensiver Applikationen genutzt werden können. Um dies jedoch zu realisieren, muß in der Anwendung die Möglichkeit zur Verteilung bzw. parallelen Abarbeitung vorgesehen sein.

Im folgenden wird ein Überblick über Kooperations- und Kommunikationsparadigmen bei paralleler Verarbeitung gegeben. Diese Paradigmen lassen sich sowohl auf Mehrprozessor bzw. Parallelrechnern, als auch auf *loosely coupled Systems*,

d.h. Zusammenarbeit mehrerer durch ein Netzwerk verbundener Rechner, realisieren.

2.2.1 Paradigmen für parallele Verarbeitung

> *"Producing quality software depends in part upon the approach to the design. To design a sequential program, the first step is to define a very high-level algorithm that outlines how the problem will be solved, and then define the data structures. This drives the design of the program in detail.*
> *In contrast, in designing a parallel program, the description of the high-level algorithm must include the method you intend to use to break the application into processes and distribute data to different nodes – the decomposition method. [. . .]*
> *The decomposition method you choose drives the rest of the program development."*
> [Ragsdale, 1991, S. 33]

Um ein Problem parallel bearbeiten zu können, muß der Programmierer neben der üblichen Definition von Algorithmen und Datenstrukturen in beiden Bereichen die Möglichkeit zur parallelen Verarbeitung vorsehen. Das *"Divide and Conquer"*-Paradigma [Metaxas, 1995, vgl. S. 284], das als Grundsatz über allen Verteilungsparadigmen steht, legt die Entwicklungsrichtung für den Programmierer grundsätzlich fest. Die Problemstellung ist in voneinander unabhängige Subprobleme aufzusplitten, sodaß diese dann parallel bearbeitet werden können.

In [Carriero and Gelernter, 1989a, vgl. S. 325f] werden drei verschiedene konzeptionelle Klassen der Aufgliederung in Subprobleme, *Result Parallelism*, *Agenda Parallelism* und *Specialist Parallelism* unterschieden:

Result Parallelism:

Beim Result-Parallelism kann das Endresultat des Problems in viele unabhängige Teilprobleme zerlegt werden. Beispielsweise kann das Endresultat *"Auto"* aus Rädern, Motor, Karosserie, Sitzen etc. bestehen. Diese Teile können unabhängig voneinander simultan gefertigt und am Ende dann zu einem Auto zusammengesetzt werden. Die zur Verfügung stehenden Ressourcen können gleichmäßig auf alle zu erledigenden Arbeiten verteilt werden.

Specialist Parallelism:

Der Specialist-Parallelism zerlegt das Gesamtproblem in Subprobleme zu deren Bewältigung spezielle Fähigkeiten erforderlich sind. Im Falle unseres Autos wäre diese Arbeitsaufteilung mit der eines Montagebandes vergeleichbar. Zuerst wird die Karosserie geschweißt, danach wird sie lackiert und der Motor eingebaut. In der nächsten Arbeitsstufe wird das Fahrgestell eingebaut, usw. Für jeden einzelnen Arbeitsschritt werden spezielle Ressourcen benötigt, die im Extremfall auch nur für diesen Schritt eingesetzt werden.

Der Specialist-Parallelism eignet sich daher für die Serienbearbeitung von Problemen, d.h. mehrere Probleme mit gleichen Arbeitsschritten können *in Serie* bearbeitet werden, sodaß sich die Bearbeitungsschritte der einzelnen Probleme überlappen. Dies ändert zwar an der Durchlaufzeit des Einzelproblems nichts, verringert jedoch die Durchlaufzeit der Serie.

Agenda Parallelism:

Beim Agenda-Parallelism wird von der Existenz eines Arbeitsplanes, einer Agenda, ausgegangen. Für jeden auf dieser Agenda aufgezeichneten Arbeitsschritt werden alle zur Verfügung stehenden Ressourcen eingesetzt. Voraussetzung ist, daß die Arbeitsschritte genügend teilbar sind. Für das Auto-Beispiel sind bei dieser Klasse z.B. alle verfügbaren Arbeiter mit der Erstellung der Karosserie, dann mit der Fertigstellung des Motors, darauf mit dem Zusammenbau des Fahrgestells usw. beschäftigt.

Diese Klasse stellt ein Mittelding zwischen den beiden vorgestellten Klassen dar. Sind die Arbeitsschritte nicht genügend teilbar und voneinander unabhängig, geht der Agenda-Parallelism in den Result-Parallelism über. Sind die Arbeitsschritte voneinander abhängig, geht der Agenda-Parallelism in den Specialist-Parallelism über.

Die Wahl der konzeptionellen Klasse ist stark von der Art des Problems abhängig. Es können auch mehrere Klassen für ein Problem passend sein. Die Auswahl hängt dann von den Präferenzen des Designers ab. Durch die Wahl der konzeptionellen Klasse für eine Problemstellung wird die grobe Bearbeitungsstruktur des Problems und die Organisation der einzelnen Lösungsteile festgelegt. Wie effizient das Gesamtproblem auf den zur Verfügung stehenden Ressourcen gelöst werden kann, hängt auch im großem Maße von der Implementierung der konzeptionellen Klassen ab.

2.2.2 Voraussetzung für den Einsatz verteilter Verarbeitung

Ziel paralleler Bearbeitung ist die Verkürzung der Bearbeitungszeit des bearbeiteten Problems gegenüber sequentieller Bearbeitung. In der Praxis kann sich dies in z.B. schnellerer Abarbeitung großer Simulationen, kürzeren Antwortzeiten bei Datenbankabfragen, größerer Verfügbarkeit einer Anwendung, auf die viele Benutzer gleichzeitig zugreifen, etc. auswirken.

Eine Maßzahl für die Effizienz paralleler Bearbeitung ist der *Speedup*. Speedup gibt an, um wieviel schneller ein paralleler Algorithmus unter Verwendung von p Prozessoren im Vergleich zu seiner sequentiellen Variante mit nur einem Prozessor ist. Ist $T(p)$ die Laufzeit eines Algorithmus bei Verwendung von p Prozessoren, so ist der Speedup definiert als $S(p) = \frac{T(1)}{T(p)}$ [Cap *et al.*, 1993, vgl. S. 21].

Voraussetzung für die parallele Bearbeitung von Problemen ist entsprechende Unterstützung von Hard-, Software und Netzwerkseite. Diese kann in mehreren Arten erfolgen, die an dieser Stelle nur kurz vorgestellt werden sollen (eine ausführlichere Darstellung erfolgt in Abschnitt 3):

Einsatz von Parallelrechnern:

Parallelrechner sind Computer mit mehreren Prozessoren, die miteinander verbunden sind. Zu den heute meist verbreiteten Kategorien von Parallelrechnern zählen *SIMD*- (single instruction stream, multiple data stream) und *MIMD* (multiple inststruction stream, multiple data stream)-Maschinen [Fountain, 1994, vgl. S. 31]:

- *SIMD:* In der SIMD-Architektur führen alle Prozessoren die gleiche Instruktion zur selben Zeit aus. Daher ergibt sich die grundsätzliche Parallelität in paralleler Verarbeitung der Daten.
- *MIMD:* Bei der MIMD-Architektur können die einzelnen Prozessoren unabhängig voneinander jeden Prozeß entweder allein oder auch in Zusammenarbeit ausführen. Es gibt mehrere Möglichkeiten für die hardwaremäßigen Verbindungen zwischen den einzelnen Prozessoren, die die Struktur des Rechners bestimmen. Das Problem kann auf MIMD-Maschinen entweder nach Funktionen oder nach Daten parallelisiert sein.

Parallelrechner sind, durch Anschaffungskosten und Leistungsbild bestimmt, eher für den Einsatz in der Wissenschaft und Technik geeignet. Sie

benötigen eigene Betriebssysteme, die die zur Verfügung stehende Hardware ausreichend unterstützt.

Einsatz von verteilten Betriebssystemen:
Ein verteiltes Betriebssystem ist ein homogenes Betriebssystem, daß identisch als eine Menge kooperierender Betriebssysteme auf den verschiedenen Knoten eines verteilten Systems implementiert ist. Das verteilte Betriebssystem bietet dem Benutzer Transparenz, d.h. er arbeitet auf logischen Ressourcen ohne die physischen Ressourcen kennen zu müssen. Für den Benutzer erscheinen die im verteilten System eingebundenen Computer wie ein großer Computer.

Diese Systeme werden für Parallelrechner und für Netzwerke von Workstationen eingesetzt. Während auf Parallelrechnern die Hardware einen Teil der Verteilung erledigt (d.h. das Betriebssystem kann auf bestimmte Hardwareroutinen zurückgreifen), muß bei einem Netzwerk aus Workstations die ganze Verteilung vom Betriebssystem allein erledigt werden, da die Rechner meist auf keinen gemeinsamen Speicher zugreifen können.

Parallele Programmiersprachen:
Bei parallelen Programmiersprachen übernimmt der Compiler in Verbindung mit einem Runtime-System die Verteilung der Programmes. Die Verteilung erfolgt dabei entweder nach Besonderheiten der Sprache (z.B. AND/OR-Parallelismus bei logischen Programmiersprachen wie bei Concurrent Prolog) oder wenn sich durch den Algorithmus unabhängige Programmteile parallel verarbeiten lassen. Der Compiler übernimmt hierbei auch fallweise die Optimierung der Verteilung.

Interprozeßkommunikation:
Bei Verwendung von Interprozeßkommunikation muß die Verteilung direkt in der Applikation ausprogrammiert werden. Diese Technik kann bei Unterstützung vom Betriebssystem in jedem Falle realisiert werden. Der Programmierer kann sich dabei einiger Kommunikationsparadigmen bedienen, wie z.B. Message Passing, Generative Communication oder Remote Procedure Calls.

Verschiedene Programmiersprachen wie C, C^{++}, Perl etc. unterstützen Interprozeßkommunikation durch Systemfunktionen. Da bei expliziter Programmierung sowohl die Kommunikation als auch die Koordination zwischen den einzelnen Programmteilen implementiert werden muß, ist die Nutzung eines bereits fertigen Systems zu bevorzugen.

Der Einsatz der oben angeführten Methoden hängt stark von der zur Verfügung

stehenden Hardware an. Diese Arbeit beschäftigt sich im weiteren mit der verteilten Verarbeitung auf einem Netzwerk von heterogenen Workstations. Dieses Gebiet wurde gewählt, weil ein Netzwerk von Workstations in Form von Rechnernetzen wesentlich häufiger in Großbetrieben bzw. Forschungsstätten zur Verfügung steht als ein Parallel- oder Superrechner. Dieses Netzwerk von Workstations kann mit entsprechender Softwareunterstützung die Funktion eines Parallelrechners zumindest teilweise übernehmen ([Kolarik, 1993], [Cap, 1993] und [Gelernter and Philbin, 1990]).

2.3 Problemstellung

Die im vorigen Abschnitt beschriebenen Trends weisen darauf hin, daß in Zukunft eine leistungsfähigere Infrastuktur zur Verfügung stehen wird. Durch die Infrastruktur und die Dienste des Internets besteht die Möglichkeit zur Implementierung von global (internetweit) verfügbaren Applikationen. Diese erfordern aber auch neue Strukturen in der Verteilung der Applikationen und der Unterstützung seitens der Betriebssysteme. Es sollte dabei möglich sein, die zur Verfügung stehende Hardware ohne teure und aufwendige Anpassungen zu verwenden. Bei den existierenden verteilten Betriebssystemen bzw. Applikationstools existieren folgende Problemfelder, die ihre Nutzung für verteilte Verarbeitung erschweren:

- Existierende verteilte Programmierumgebungen (z.B. DCE, CORBA) sind aufwendig in der Handhabung und meistens nur in kommerzieller Form verfügbar. Dies schließt die Nichtverfügbarkeit des Sourcecodes ein, was große Nachteile hinsichtlich der Anpassung des Systems auf den jeweiligen Verwendungszweck nach sich zieht.
- Gemischtsprachige Gesamtanwendungen bzw. die Zusammenarbeit von gemischtsprachigen Einzelanwendungen ist in diesen Systemen problematisch, da die Verteilung und die Optimierung zumeist beim Kompilieren festgelegt wird. Die Auswahl der zu verwendenden Programmiersprachen ist auf eine kleine Gruppe wie z.B. C, C^{++} beschränkt. Der Datenaustausch zwischen den Anwendungsteilen muß explizit programmiert werden.
- Es besteht zumeist keine Möglichkeit der Einbindung bereits bestehender Applikationen.
- Die Systeme laufen nur auf speziellen Rechnertypen bzw. unter speziellen Betriebssystemen. Die Verwendung von Low-Cost-Maschinen ist nicht oh-

ne weitere Maßnahmen möglich. Dies setzt der Skalierbarkeit und Erweiterbarkeit ökonomische Grenzen.

In dieser Arbeit wird, ausgehend von David Gelernters Ansatz von *Linda*, eine portable Implementation von Linda für die Kooperation und Kommunikation von gemischsprachigen Applikationen entwickelt. Das auf Client/Server-Basis implementierte *Perl-Linda* ermöglicht in einem Netzwerk von heterogen Workstations den Zugriff von Clients in verschiedenen Programmiersprachen auf ein *shared Memory* in Form eines Linda-Tuplespace unter Verwendung von Linda als Schnittstelle. Perl-Linda ermöglicht somit, Cluster von heterogenen Rechnern für verteiltes Rechnen zu nutzen.

Es wird anhand von Beispielen der Einsatz dieses Prototyps gezeigt. Die Erweiterung des World Wide Web-Systems (WWW) mit Perl-Linda erlaubt die Erstellung von dialogbasierten Applikationen mit einem WWW-Front End, die Verwendung gemischsprachiger Applikationsteile in einer Gesamtappplikation und Verteilung der Bearbeitung einer WWW-basierten Applikation über mehrere Rechner.

2.4 Aufbau der Arbeit

Kapitel 3 behandelt Methoden und Systeme für verteilte Bearbeitung. Nach einer Auflistung der grundsätzlichen Bausteine für verteilte Verarbeitung werden Implementationsmethoden für die Kommunikation und Kooperation zwischen Prozessen beschrieben. Das Kapitel endet mit einem Überblick über Systeme für verteilte Verarbeitung.

In Kapitel 4 wird ein Überblick über Linda gegeben. Nach einer Beschreibung der Elemente und Funktionsweisen von Linda, werden die einzelnen Linda-Befehle exemplarisch erklärt. Es folgt ein Vergleich von Linda mit den anderen zur Interprozeßkommunikation eingesetzten Methoden. Danach wird ein Überblick über bereits existierende Linda-Implementationen gegeben. Perl-Linda wird in Kapitel 5 beschrieben. Es wird das Perl-Linda-Interface, insbesondere die Darstellung von Tuple und Pattern in Perl-Linda, der Perl-Linda-Befehlssatz und seine Funktionsweise erläutert. Danach werden die beiden Hauptbestandteile von Perl-Linda, der Perl-Linda-Server und die Schnittstelle zum Perl-Linda-Client behandelt.

Die Beschreibung des Perl-Linda-Servers in Kapitel 6 gliedert sich in die Auflistung der Server-Funktionen, die Darstellung der Grammatik der Client/Server-

Kommunikation, einer Beschreibung des Matching-Prozesses und der Restriktionen in Perl-Linda und in eine Beschreibung der Client-Schnittstelle zum Perl-Linda-Server. Aufbauend auf der beschriebenen Schnittstelle wird in Kapitel 7 die Implementation der Schnittstelle in der Programmiersprache Perl beschrieben. Die Implementation der Perl-Linda-Schnittstelle für die Programmiersprachen C^{++} und APL befindet sich in Kapitel 8.

Kapitel 9 enhält eine Übersicht über Verteilungsmethoden und Kompositionsparadigmen für verteilte Verarbeitung. Es werden die Entwicklungsschritte zur Entwicklung eines parallelen Programmes in Perl-Linda beschrieben und an einem Entwicklungsbeispiel demonstriert. Das Kapitel schließt mit Beispielen zu den grundsätzlichen Entwicklungsparadigmen ab.

Der Einsatz von Perl-Linda mit dem World Wide Web zur Erstellung global verfügbarer Transaktionssystem ist in Kapitel 10 beschrieben. Nach einer Beschreibung der Komponenten des WWW-Systems und ihrer Funktionsweise werden Probleme und Beschränkungen des Systems aufgezeigt. Der Einsatz von Perl-Linda mit WWW und die sich daraus ergebenden Möglichkeiten werden beschrieben und es wird anhand von Applikationen das Zusammenwirken der beiden Komponenten erläutert.

Im Anhang der Arbeit sind der vollständige Source-Code von Perl-Linda und die in Kapitel 9 vorgestellten Source-Beispielen enthalten.

3 Methoden und Systeme für verteilte Verarbeitung

In diesem Kapitel soll ein Überblick über die Bausteine und Implementationsmethoden für verteilte Systeme gegeben werden. Nach einer Beschreibung der grundsätzlichen Bausteine, die dem Programmierer zur Verfügung stehen, werden verschiedene gängige Implementationsmethoden, ihre Grenzen und Einsatzgebiete beschrieben. Zum Abschluß werden einige verteilte Systeme in ihrer Funktionsweise erklärt.

3.1 Einführung

> *"One way to solve a difficult problem fast is to break the problem into separate pieces and work on all the pieces at the same time. This idea is the whole basis for parallel computing."*
> [Gelernter and Philbin, 1990, S. 213]

Die Problemzerlegung in mehrere Subprobleme ist die Ausgangsbasis für die parallele Abarbeitung dieser Problemteile. Dies beschleunigt die Gesamtbearbeitungszeit und macht so rechenaufwendige Probleme auch ohne den Einsatz eines Superrechners berechenbar. Dieses parallele Bearbeiten eines Problems kann auf einem einzelnen Rechner mit mehreren Prozessoren oder durch mehrere Rechner, die durch ein Netzwerk verbunden sind, erfolgen. In jedem Fall sind an der Bearbeitung mehrere Prozessoren beteiligt. Um ein Problem mit mehreren Prozessoren zu lösen, muß der Programmierer neben der eigentlichen Problemlösung zusätzlich auch den Datenaustausch und die Koordination zwischen den beteiligten Komponenten implementieren.

Der Programmierer muß bei der Implementierung nicht das Rad neu erfinden. Aufbauend auf dem jeweiligen Betriebssystem kann er sich einiger grundsätzlicher Programmbausteine bedienen, die er zur Gestaltung und zur Implementierung der Kommunikation und Koordination zwischen den einzelnen Prozessen einsetzen kann. Als Bausteine stehen *Filter*, *Server* und *Monitore*, *Clients* und *Peers* zur Verfügung. Mit diesen Bausteinen kann Interprozeßkommunikation und Koordination nach verschiedenen Paradigmen erfolgen. Als Vertreter seien hier *Message Passing*, *Generative Communication* und *Remote Procedure Calls* genannt. In verteilten Umgebungen sind einige der vorher genannten Bausteine und Methoden bereits in das Betriebssystem integriert. Diese stehen entweder als ver-

teilte Betriebssysteme, wie *Amoeba*, *Mach* und *Chorus* oder als Erweiterungen zu Programmiersprachen zur Verfügung.

3.2 Bausteine verteilter Verarbeitung

Die im folgenden vorgestellten Bausteine sind Prozesse, die sich bei der Erstellung einer verteilten Anwendung einsetzen lassen. [Andrews, 1991, vgl. S. 50] gibt vier Prozesse an, welche in verteilten Programmen verwendet werden: *Filter*, *Monitore / Server*, *Clients* und *Peers*:

Filter:

Ein Filter ist ein Programm zur Datentransformation. Jeder Filter besitzt genau einen Eingabe- und einen Ausgabestrom. Er liest Daten von seinem Eingabestrom und verarbeitet diese in bestimmter Weise. Die Ergebnisse der Verarbeitung werden auf den Ausgabestrom geschrieben. Filter sind speziell im Betriebssystem UNIX verbreitet und werden für viele Kommandozeilen-Befehle und Shell-Programme eingesetzt.

Durch die Eigenschaft, mindestens einen Ein- und genau einen Ausgabestrom zu haben, lassen sich die Filter zu sogenannten *Pipes* zusammensetzen. Bei einer Pipe dient die Ausgabe des einen Filters als Eingabe des anderen Filters. So lassen sich mehrere Filteroperationen aneinanderreihen. Es führt z.B. die Befehlszeile

```
find . -name "*.tex" -print | wc -l
```

dazu, daß alle vom Befehl `find` ausgegebenen Dateien an den Filter `wc` (word count) weitergeleitet werden. Dieser zählt die Anzahl der Zeilen. Da jeweils nur eine Datei pro Zeile ausgegeben wird, ist das Ergebnis dieser Pipe die Anzahl der Dateien mit der Endung `*.tex`.

Monitore / Server:

Monitore und Server sind Programmbausteine, die mit der Kommunikation und Koordination zwischen Programmteilen oder anderen Programmbausteinen befaßt sind.

> *"A monitor is a synchronisation mechanism that is commonly used in concurrent programs that execute on shared memory machines."*
> [Hoare, 1974] in [Andrews, 1991, S. 56]

Ein Server ist ein Programm, welches Services zur Verfügung stellt, die durch Messages von Clients aufgerufen werden. Der Hauptunterschied zwischen Monitore und Server ist, daß der Server aktiv ist und im Gegensatz zum Monitor nicht bei Nichtausführbarkeit einer Anfrage blockieren kann, sondern die anderen wartenden Anfragen weiter bearbeiten muß [Andrews, 1991, vgl. S. 58]. Die Prozeduren werden von den Clients beim Monitor durch das Aufrufen einer Monitorprozedur aktiviert, beim Server geschieht dies durch Senden und Empfangen von Messages.

Welcher der beiden Bausteine zum Einsatz kommt, wird durch die Art des Rechners und die Effizienz der Implementation des Bausteins bestimmt. Während Monitore auf Shared-Memory-Maschinen die effizientere Lösung für das Betriebssystem darstellen, sind Server auf Basis von Message Passing und Remote Procedure Calls (RPC) für Implementation von verteilten Systemen effizienter.

Clients:

Clients werden gemeinsam mit Servern bzw. Monitoren eingesetzt. Sie stellen den aufrufenden Teil des Systems dar und nehmen die angebotenen Dienste bzw. Prozeduren in Anspruch. Die meisten Clients blockieren bei der Inanspruchnahme eines Dienstes vom Server bis die Bearbeitung der Anfrage beendet ist und das Ergebnis vom Server zurückgesandt wird.

Peers:

Peers sind Gruppen von Prozessen, welche gemeinsam eine Ressource verwalten oder ein Service bereitstellen. Jeder Peer hat eine lokale Kopie von der Ressource, die von allen Peers gemeinsam auf dem letzten Stand gehalten wird. Beispiele in der Anwendung finden sich bei `ftp`-Servern, Fileservern, verteilten Dateisystemen und verteilten Datenbanken.

Neben der reinen Verwaltung einer Ressource oder eines Service können Peers auch noch eingesetzt werden, um ein Problem gemeinsam zu lösen, welches Interaktion zwischen den einzelnen Peers benötigt.

Mit diesen Prozeßbausteinen lassen sich verteilte Programme und verteilte Systeme konstruieren. Um diese Bausteine auf der jeweiligen Hardware implementieren zu können, muß eine Methode gewählt werden, die eine Kommunikation zwischen den verschiedenen Objekten bzw. Prozessen ermöglicht. Im folgenden Abschnitt werden einige Kommunikationsmethoden vorgestellt, die dem Software-Designer grundsätzlich zur Verfügung stehen.

3.3 Implementationsmethoden für Interprozeßkommunikation

Es werden im folgenden drei Implementationsmethoden zur Interprozeßkommunikation beschrieben, welche dann im weiteren zur Unterstützung des Design der im nächsten Abschnitt beschriebenen Kommunikationsstrukturen herangezogen werden können. [Andrews, 1991, vgl. S. 52] unterscheidet:

Message Passing:

Bei dieser Methode wird die Kommunikation zwischen den Prozessen durch das Schicken von Nachrichten (Messages) durchgeführt. Nach der Beschaffenheit der Kommunikationskanäle und der Art der Übertragung wird zwischen *synchronous*, *asynchronous* und *buffered* Message Passing unterschieden. Während beim asynchronous Message Passing die Prozesse unabhängig vom Empfang der von ihnen gesendeten Nachrichten agieren können, gibt es bei den beiden anderen Varianten Abhängigkeiten, die fallweise zu einem temporären Blockieren der Prozesse führen.

Die Messages werden zwischen den Prozessen auf eigenen Kanälen übergeben und mittels dem Kommando `send` abgeschickt und mittels `receive` empfangen. Bei der Kommunikation muß der Kommunikationskanal angegeben werden. Weiters müssen die gesendeten Daten beiden Prozessen bekannt sein, d.h., sie müssen die Anzahl und den Typ der gesendeten Daten kennen.

Generative Communication:

Diese in [Gelernter, 1985] erstmals vorgestellte Methode ist eine Weiterentwicklung des Message Passing. Die kommunizierenden Prozesse verwenden nur einen zentralen Kommunikationskanal, dem Tuplespace, über den sie ihre Nachrichten austauschen. Auf diesen Kommunikationskanal greifen die Prozesse über ein Interface zu, daß aus drei Befehlen besteht. Durch diese lassen sich die im Tuplespace befindlichen Daten von den Prozessen manipulieren. Dieses Konzept bildet die Grundlage von *Linda* und wird im Hauptteil dieser Arbeit (siehe Abschnitt 4) noch genauer erläutert.

Remote Procedure Call:

Diese Methoden kombinieren die Funktionsweise von Monitoren und synchronous Message Passing. Die Operationen werden wie bei Monitoren durch Aufruf gestartet, die Übermittlung der Daten bzw. der Ergebnisse erfolgt jedoch über Messages, die in ein standardisiertes Übermittlungsformat gebracht werden. Der aufrufende Prozeß blockiert so lange, bis er

ein Ergebnis erhalten hat. RPC ist sowohl zwischen Prozessen auf dem lokalen Rechner als auch zwischen Prozessen auf verschiedenen Rechnern möglich. Es müssen aber (z.B. beim Betriebssystem UNIX) spezielle Permissions zwischen den Rechnern gesetzt sein, d.h., der Administrator des Rechners, auf dem der RPC durchgeführt wird, muß seine explizite Erlaubnis zu dieser Nutzung geben.

Rendezvous:
Rendezvous ermöglicht Interprozeßkommunikation durch Herstellen eines Rendezvous mit einem anderen Prozess. Diese Prozesse können dann Daten tauschen. Dies ermöglicht die Implementation der Funktion von Monitoren zwischen verschiedenen Rechnern. Es kann eine Synchronisation zwischen verschiedenen Prozessen hergestellt werden. Der Prozeß der früher am Rendezvouspunkt eintrifft, wartet bis der andere Prozeß soweit ist. Während RPC auf Betriebssystemebene implementiert ist, findet sich Rendezvous zumeist auf der Ebene der Programmiersprachen (vgl. dazu [Ben-Ari, 1982, S. 93ff.])

Die vorgestellten Methoden sind grundsätzlich equivalent und Programme, in welchen eine der Methoden verwendet wurde, können in jede der anderen Methoden mit mehr oder weniger Aufwand transformiert werden. [Andrews, 1991, vgl. S. 52] präferiert trotz der Gleichwertigkeit asynchronous Message Passing (AMP) aus den folgenden drei Gründen:

- AMP ist die flexibelste Methode, sie ist der kleinste gemeinsame Teiler der Methoden.
- Unterstützung zur Implementation von AMP findet sich in den meisten Betriebssystemen von Netzwerk- bzw. Multiprozessorrechnern.
- AMP ist der natürlichste Zugang zu verteilter Verarbeitung. Speziell durch den Einsatz objektorientierter Programmiersprachen wird dieses Argument noch erhärtet.

3.4 Kommunikations- und Kooperationsstrukturen

Die in Abschnitt 3.2 dargestellten Bausteine können mit Hilfe der im vorigen Abschnitt beschriebenen Methoden zur Implementation von verschiedenen

Kommunikations- und Koopeationsstrukturen dienen. Hierbei geht es mehr um die Verwendung und Zusammensetzung der Bausteine als um die tatsächliche Wahl der Methode. Die in den folgenden Unterkapiteln beschriebenen Strukturen folgen der in [Andrews, 1991, vgl. S. 54ff.] dargestellten Einteilung.

3.4.1 Netzwerk von Filtern

Ein Netzwerk von Filtern wird durch eine Aneinanderreihung von Filtern erzeugt. Der Datenfluß in diesem Netzwerk kann durch die Funktion der Filter prinzipiell nur in eine Richtung erfolgen. Es lassen sich grundsätzlich zwei Varianten von Filternetzwerken unterscheiden:

Homogene Filternetzwerke:
In homogenen Filternetzwerken werden nur Filter gleicher Funktion aneinandergereiht. Ein Beispiel für dies ist ein Sortiernetzwerk, welches aus einer Aneinanderreihung von Sortierfiltern (Merge-Filter) besteht, die jeweils aus 2 sortierten Inputlisten eine sortierte Liste erzeugen. Die Effizienz des Sortieralgorithmus kann so bei entsprechendem Vorhandensein von Prozessoren gesteigert werden (vgl. dazu [Andrews, 1991, S. 55ff.]).

Heterogene Filternetzwerke:
Im Unterschied zur vorher vorgestellten Gruppe werden in heterogenen Filternetzwerken nicht einzelne Filter, sondern Pipes von heterogenen Filtern zusammengeschlossen. Diese können verschiedene Aufgaben durchführen, die nacheinander ablaufen müssen.

Dies kann beispielsweise in der Bildbearbeitung der Fall sein: *Rendern des Bildes* $\rightarrow$ *Umwandeln* $\rightarrow$ *Zuschneiden* $\rightarrow$ *Katalogisieren* $\rightarrow$ *Drucken*.

Während bei der homogenen Variante die Filter als kleinstes Element auf die zur Verfügung stehenden Prozessoren verteilt werden können, ist bei der heterogenen Variante die Pipe das kleinste Element, nach dem verteilt werden kann. Daher hat diese Variante eine gröbere Granularität als die homogene Variante. Bei einem Aufgabenstrom kommt es bei einer Pipe zu einer überlappenden Verarbeitung ähnlich einem Fließband.

Da der Datenstrom bei Filternetzwerken nur in eine Richtung fließen kann, stellen diese eine sehr inflexible Variante zur Implementation von verteilten Programmen dar. Sie benötigen viele Prozessoren und einen Prozeß, der die Ein- und Ausgabeströme in der vorgesehenen Weise umleitet.

3.4.2 Client/Server-basierte Kommunikationsstrukturen

In diesem Abschnitt werden Client/Server-basierte Strukturen beschrieben. Gegenüber Filternetzwerken weisen Client/Server-Kommunikationsstrukturen größere Flexibilität im Hinblick auf Skalierbarkeit und Routing auf. Ein Nachteil von Client-Server Kommunikationsstrukturen ist das mögliche Auftreten von Deadlocks. Diese treten auf, wenn mehrere Programmteile auf eine Message warten, aber niemand diese sendet, und umgekehrt.

Es können sowohl symmetrische als auch asymmetrische Kommunikationsstrukturen geschaffen werden. Bei symmetrischen Kommunikationsstrukturen kann jeder Client die gesamte Arbeit auch alleine erledigen; es wird im späteren auch von *homogenen Clients* gesprochen. Hingegen arbeiten bei der asymmetrischen Variante Clients mit verschiedenen Aufgabenbereichen zusammen, die die Gesamtaufgabe nur gemeinsam lösen können. Server stellen in diesem Zusammenhang in erster Linie die Infrastruktur für das Zusammenwirken der Clients zur Verfügung. Diese Aufgabe teilt sich in Kommunikations- und Koordinationsaufgabe. Der Server dient als Informationslager für die Clients. Andererseits kann er die Clients auch steuern und koordinieren. Beides muß aber vom Programmierer explizit festgelegt werden.

Bei den nachfolgenden dargestellten Kommunikationsalgorithmen geht es um die Implementation der Kommunikation zwischen den einzelnen Clients bzw. Servern. Ihr Einsatz ist von der zu bewältigenden Aufgabe und von der Netzwerkstruktur abhängig, mit der die an der Problemlösung beteiligten Komponenten vernetzt sind. Die vorgestellten Varianten sind selten gleich geeignet für die Implementation von bestimmten Client/Server-Anwendungen. Die Untergliederung folgt im groben der in [Andrews, 1991, vgl. S. 57ff.] aufgelisteten Gliederung:

Heartbeat-Algorithmen:

Für Netzwerktopologien, in denen nicht jeder Knoten mit jedem anderen Knoten verbunden ist, sind Heartbeat-Algorithmen geeignet. Sie können sowohl als homogene als auch als heterogene Clients implementiert werden. Den Namen bekamen diese Algorithmen von ihrer der des Herzen ähnlichen Funktionsweise: zuerst expandiert der Client, indem er Information nach außen sendet, dann zieht er sich zusammen, indem er Information von außen bekommt.

Als Interaktionspartner stehen, als Folge der Netzwerktopologie, jeweils nur die direkt mit dem Knoten verbundenen Rechner zur Verfügung. Durch die

homogene Clientstruktur versucht jeder Client, die Gesamtaufgabe allein zu erfüllen, d.h., alle Informationen lokal zu sammeln, zu speichern und mit seinen Nachbarclients auszutauschen. Beispiele für die Verwendung von Heartbeat-Algorithmen zur Ermittlung der Netzwerktopologie sind in [Andrews, 1991, vgl. S. 64ff.] beschrieben.

Probe/Echo-Algorithmen:

Ebenfalls für die selbe Gruppe von Netzwerktopologien wie Heartbeat-Algorithmen sind Probe/Echo-Algorithmen geeignet. Im Unterschied zu den Heartbeat Algorithmen, sind Probe/Echo-Algorithmen zur Implementation eines Broadcasts auf so einer Netzwerktopologie eingesetzt. Da ein Broadcast auf dieser Topologie durch die Hardware nicht garantiert werden kann, muß ein Reply/Response-Schema, daß das Erreichen jedes Netzwerknotens sicherstellt, explizit implementiert werden.

Die Anordnung der Rechner zu einem Netzwerk wird in diesem Fall als gerichteter, fallweise zyklischer Graph gesehen, an dessen Kanten sogenannte Probes geschickt werden, die von Knoten anhand der Kante weitergegeben werden. Jeder Knoten gibt die Probe an alle Nachbarknoten weiter, ausgenommen des Knotens, von dem er sie erhalten hatte. Danach wartet er auf Echos von allen Nachbarknoten. Sollten weitere Probes gesendet werden (dies kann durch Zyklen im Graphen auftreten), werden diese sofort mit einem Echo beantwortet. Hat er alle Echos empfangen, so sendet er ein Echo an den Knoten, von dem er die erste Probe empfangen hat. Damit ist sichergestellt, daß der Teilgraph unterhalb des Knotens den Broadcast erhalten hat.

Broadcast-Algorithmen:

Broadcasts sind in der Hardware implementierte Mechanismen, eine Nachricht an alle Prozesse/Knoten zu senden. Sie werden bei einer Netzwerktopologie verwendet, in der alle Knoten logisch mit allen anderen Knoten verbunden sind. Da Broadcasts nicht gleichzeitig bei allen Knoten ankommen, muß bei der Verwendung eines Broadcast Algorithmus ein Timestamp verwendet werden, wenn die Reihenfolge des Eintreffens für die Anwendung ein Kriterium darstellt.

Durch Broadcasts läßt sich Kommunikation zwischen Prozessen auf einfache Weise realisieren, da die Übermittlung der Messages durch das Netzwerk erledigt wird. Es entsteht bei der Verwendung von Broadcasts jedoch ein Overhead gegenüber zielgerichteter *Point-to-Point* Kommunikation wenn die Anzahl der Netzwerkknoten groß ist und nur wenige Knoten die Messages benötigen. Beispiele für den Einsatz von Broadcasts zur Implementation von verteilten Semaphoren bzw. Shared Variables finden sich

in [Andrews, 1991, vgl. S. 74ff.] und [Kolarik, 1993].

Token-Passing-Algorithmen:

Diese Art von Algorithmen wird zur impliziten Steuerung des Zugriffs für von mehreren Prozessen gemeinsam genutzte Ressourcen wie Dateien, gemeinsame Speicherbereiche, etc. verwendet. Der Zugriff auf diese wird durch das Vergeben eines Tokens gesteuert. Nur der Prozeß, der im Besitz des Tokens ist, darf auf die Ressource zugreifen.

Für die Verteilung und Vergabe des Tokens sind mehrere Kooperationsformen zwischen den Prozessen möglich. Das Token kann entweder zwischen den in Frage kommenden Clients zirkulieren oder, durch einen Server verwaltet, nach Bedarf vergeben werden. Diese Implementationsform kann dezentral ablaufen und ist nicht auf eine zentrale Zugriffskontrolle durch z.B. Zugriffskontrollvektoren [Janko, 1981, vgl. S. 336ff.] und [Kolarik, 1993, vgl. S. 29ff.] angewiesen.

Beispiele für die Implementation von kritischen Code-Sektionen und der Feststellung der Beendigung der Verarbeitung einer verteilten Anwendung mittels Token-Passing-Alogrithmen finden sich in [Andrews, 1991, vgl. S. 77ff.].

Für ein Netzwerk von Workstations, das mit Ethernet oder ähnlichem vernetzt ist, kann angenommen werden, daß jeder Prozessor mit jedem verbunden ist. Die Annahme anderer Netzwerktopologien ist nur mehr bei Multiprozessorrechnern und bei Kommunikation über Wide Area Networks (WAN) sinnvoll, da in diesen Fällen die Kommunikationskosten durch die Netzwerktopologie wesentlich bestimmt werden und zwischen zwei beliebigen Knoten nicht mehr als konstant betrachtet werden können.

3.4.3 Replizierte Server

Bei der Verwendung einer Client/Server-Konfiguration für Applikationen erweist sich der Server bei Nutzung des angebotenen Service durch viele Clients als Engpaß. Eine Lösung zur Umgehung dieses Engpasses bietet der Einsatz mehrerer replizierter Server, die das gleiche Service anbieten. Das Problem bei dieser Variante ist der Update und die Konsistenz der von den Servern gemeinsam benötigten oder gehaltenen Information. Der Update kann zwischen den Servern durch mehrere Mechanismen erfolgen:

- *Backups:* Die gemeinsamen Daten werden in bestimmten Intervallen von einem Hauptserver den anderen Servern als Backupkopie zur Verfügung gestellt.
- *Einsatz eines Superservers:* Durch einen Superserver regeln die replizierten Server ihren Zugriff auf die gemeinsamen Daten. Bei Dateien werden z.B. Lesezugriffe durch den Server, Schreibzugriffe aber hingegen durch den Superserver durchgeführt. Hier kann z.B. ein Token-Passing-Algorithmus zum Einsatz gebracht werden. Das Token für den Schreibzugriff wird hierbei vom Superserver vergeben.
- *Automatischer simultaner Update zwischen den Servern:* Diese Technik stellt die größten Anforderungen an die Implementierung, da ein Protokoll entwickelt werden muß, das durch den Einsatz von z.B. Broadcasts den Update der Daten regelt.

Der Einsatz von replizierten Servern dient aber nicht nur der Erhöhung des Durchsatzes eines Service, sondern auch der Erhöhung der Verfügbarkeit durch eine Erhöhung der Ausfallsicherheit bei Ausfall eines oder mehrerer Rechner (vgl. dazu [Bakken, 1993]).

3.5 Systeme

> *"In general, message passing primitives, such as synchronous or asynchronous message passing, RPC, and rendevouz, are low level communication concepts that do not necessarily fit well into the host language and thus do not lead to a clean integrated language concept which meets the requirements of distributed systems.*
> [Forst *et al.*, 1994]

Die in den vorhergehenden Abschnitten dargestellten Bausteine und Methoden stellen also nur die Basis für die Implementation von verteilten Systemen dar. Um verteilte Verarbeitung für Anwendungen nutzbar zu machen, existieren Unterstützungsysteme und -umgebungen auf drei Ebenen, der Betriebsystemebene, auf der Ebene der Anwendungsprogramme und als Erweiterungen zu Programmiersprachen, um diese für verteilte Verarbeitung tauglich zu machen. Im folgenden werden Beispiele aus allen drei Ebenen behandelt und ihre Fähigkeiten und Grenzen aufgezeigt.

3.5.1 Betriebssysteme/Microkernels

Wie jeder Einprozessorrechner brauchen Mehrprozessorrechner auch ein eigenes Betriebssystem, welches sowohl die lokalen Ressourcen jedes einzelnen Prozessors als auch die gemeinsamen, globalen Ressourcen verwaltet. Prinzipiell könnte das Betriebssystem UNIX dafür herangezogen werden. Ein vollständiges UNIX-Betriebssystem auf jedem der Prozessoren bedeutet jedoch einen unverhältnismäßig hohen Overhead, da UNIX zu groß und zu ineffizent für viele auf Multicomputern implementierten Applikationen ist [Tanenbaum, 1994, vgl. S. 2]. Daher ging man dazu über, Betriebssysteme für diese Art von Computern auf Basis von Microkernels zu implementieren. Diese stellen die Basisfunktionalität bezüglich *Prozeßmanagement*, *Speichermanagement* und *Prozeßkommunikation* zur Verfügung. Andere traditionelle Dienste wie Dateisystem sind durch Server abgedeckt, die von den Microkernels gemeinsam genutzt werden können.

In [Tanenbaum, 1994] und [Coulouris *et al.*, 1994] wurden drei verteilte Betriebssysteme auf der Basis von Microkernel, *Amoeba*, *Mach*[1] und *Chorus* bezüglich der drei oben genannten Basisfunktionalitäten gegenübergestellt und miteinander verglichen. Es ergaben sich folgende Gemeinsamkeiten in Fähigkeiten und Funktion:

Prozeßmanagement:
Alle Systeme unterstützen mehrere *Threads* pro Prozess. Diese werden alle vom Kernel kontrolliert. Während die Verteilung von Prozessen über die verfügbaren CPU von allen Systemen unterstützt wird, können in Amoeba Threads eines Prozesses nicht verteilt verarbeitet werden. Als Mittel der Prozeßsynchronisation stehen in den Betriebsystemen Signals (Amoeba), Mutexes[2] und Semaphore zur Verfügung.

Speichermanagement:
Während bei Amoeba sich der gesamte Prozeß im Speicher befinden muß, unterstützen die anderen Systeme *Demand Paging*, bei dem Speichersegmente wie auch Dateien dynamisch aus dem Speicher auf den Massenspeicher ein- und ausgelagert werden können. Alle drei Systeme unterstützen

[1] Hierbei ist nicht der von OSF für den Einsatz als Basis für den UNIX-Dialekt OSF/1 übernommene und adaptierte Microkernel gemeint, sondern CMU-Mach V3.0.

[2] Ein Mutex ist ein binärer Semaphor der nur die Zustände *locked* und *unlocked* annehmen kann. Er wird zur Koordination zwischen Prozessen eingesetzt. Versucht ein Prozeß, ein Mutex im Zustand *unlocked* zu sperren, geht er in den Zustand *locked* über. Der Prozeß arbeitet weiter. Versucht ein anderer Prozeß den gesperrten Mutex noch einmal zu sperren blockiert der Prozeß so lange, bis der Mutex sich wieder im Zustand *unlocked* befindet.

Distributed Shared Memory, wenn auch in unterschiedlicher Weise und Granularität. Während in Amoeba Softwareobjekte in Variablengröße durch den Kernel direkt im gemeinsamen Zugriff kontrolliert werden, verwenden die anderen Systeme Server zur Bereitstellung dieser Funktionalität.

Prozeßkommunikation:

Grundsätzlich werden von den verteilten Betriebsystemen zwei Arten der Kommunikation zwischen Prozessen unterstützt, Messages und Broadcasts. Die Unterscheidung liegt hier in der Verläßlichkeit der Übermittlung. Amoeba unterstützt unverläßliches *One-Way Message Passing*, verläßlicher *Remote Procedure Call* und *Totally-Ordered Group Communication*, ein Subset aus Broadcast, wobei die Reihenfolge in der die Broadcasts ausgesendet werden erhalten bleibt. Da in Amoeba keine *Ports* im UNIX-Sinne existieren, werden die Prozesse durch sogenannte *Service Addresses* angesprochen.

Mach und Chorus besitzen *Ports*, die explizit adressiert werden können. Während die Ports in Mach – ähnlich UNIX – nur zum Empfangen von Messages eingesetzt werden, können sie in Erweiterung bei Chorus auch zum Senden verwendet werden. Server können in Chorus auch auf allen ihren Kommunikationskanälen simultan emfangen[3]. Dies bedeutet, daß ein Serverprozeß gleichzeitig auf allen seinen Serviceports lauschen kann und sie deshalb nicht z.B. einer Schleife abfragen muß.

Es kann auch ein Multicast für Gruppen von Prozessen durchgeführt werden. Weiters gibt es bei Mach die Möglichkeit, die Anzahl der Messages für einen Prozess und ein Port zu limitieren.

Die Funktionalität der Systeme ist hinsichtlich der vorgestellten Aufgaben relativ ähnlich, wenn auch einzelne Features voneinander abweichen. Der effiziente Einsatz von Microkernels bedingt eine spezielle Hardware bzw. spezielle Konfiguration der Hardware und der Netzwerkverbindungen und ist deshalb nicht für ein Netzwerk von Workstations geeignet. Verteilte Betriebssysteme basierend auf Microkernels extistieren dennoch für eine große Zahl von Spezialrechnern wie z.B. Cray T3D, Intel Convex, ICL Goldrush, Archipel Goldrush u.a. Systeme.

[3]Diese Funktionalität entspricht der des UNIX-Systembefehls `select(2)` mit dem gleichzeitig mehrere Eingabeports, die mittels einer Bit-Maske adressiert sind, abgehört werden können.

3.5.2 Verteilte Umgebungen und Spracherweiterung

In diese Gruppe fallen alle Systeme, die nicht direkt in das Betriebssystem integriert sind, sondern als *Runtime*-System mit oder ohne Anbindung an Programmiersprachen zur Verfügung stehen. Die Systeme haben den Vorteil, daß sie, je weiter sie vom Betriebssystem entfernt sind, ohne spezielle Hardware eingesetzt werden können.

Bei verteilten Umgebungen werden die Prozesse und Objekte von Servern oder Objekt Request Brokern (ORB), die als Daemonprozesse auf allen Rechnern laufen, verteilt und in Evidenz gehalten. Bei Spracherweiterungen, speziell bei Compilersprachen kann die Verteilung aber auch schon beim Übersetzen festgelegt werden. Die Umgebungen und Erweiterungen unterstützen den Entwickler in drei Hauptaufgaben [Orfali *et al.*, 1994, vgl. S. 55f]:

- Sie erweitern das lokale Betriebssystem für die Nutzung von gemeinsamen Ressourcen wie Drucker, Dateiserver und Modempools. Diese Funktionalität wird von Netzwerkbetriebssystemen wie NetWare 3.0, LAN Server 3.0, NFS u.ä. übernommen.
- Durch ihren Einsatz läßt sich die Basis für ein *Einzelsystem*, das alle über ein Netzwerk verteilten Ressourcen einschließt, erstellen. Tools wie NetWare 4.0, Banyan Vines oder OSF Distributed Computing Environment (DCE) stellen neben der Basisfunktionalität auch noch Möglichkeiten zur (automatischen) Administration der Ressourcen zur Verfügung.
- Sie unterstützen die Koordination von Applikationen, die auf Client/Server-Basis über die Ressourcen aufgeteilt sind. Man kann hier grundsätzlich zwischen *tightly-coupled* Reply/Response Interaktionen auf Basis von RPC (DCE, Sun Open Network Computing) und einer *loosely-coupled queue-based* Kommunikation (Message Oriented Middleware) unterscheiden.

Auf dieser Grundfunktionalität können nun speziellere Dienste aufsetzen, die einen Teilbereich oder eine Anwendung spezifischer unterstützen.

Erweiterungen von Programmiersprachen hinsichtlich verteilter Verarbeitung dienen dazu, die in Abschnitt 3.2 – 3.4 beschriebenen Komponenten dem Programmierer in einer der Sprache eigenen Art und Weise zur Verfügung zu stellen. Weiters schaffen die Erweiterungen eine Verbindung zu der entsprechenden Imple-

mentation im Basissystem (Betriebssystem). Es ist zwischen drei Erweiterungsansätzen zu unterscheiden:

- *verteilte Bausteine als zusätzliche Bibliothek:*
  Bei dieser Gruppe wird der Programmiersprache eine Funktionenbibliothek zur Verfügung gestellt. In Verbindung mit einem Runtime-System ermöglicht diese, verteilte Programme zu schreiben.

  Parallel Virtual Machine (PVM) zählt zu den Vertretern dieser Gruppe. Durch den Einsatz von PVM kann eine benutzerdefinierte Auswahl von Rechnern wie ein großer, verteilter Rechner verwendet werden. PVM setzt auf das Betriebsystem UNIX auf und stellt dem Programmierer die Werkzeuge für Interprozesskommunikation zur Verfügung. Die Prozesse können innerhalb der virtuellen Maschine miteinander über die PVM Interface-Routinen kommunizieren. Bei der Kommunikationsart können sowohl Signals als auch Messages verwendet werden. PVM garantiert, daß die Reihenfolge der Messages beibehalten wird (vgl. dazu [Geist *et al.*, 1993]).

- *Die Verteilung wird durch den Compiler durchgeführt:*
  Bei diesem Erweiterungsansatz sind die Verteilungswerkzeuge direkt in die Sprachumgebung integriert. Der Compiler analysiert beim Kompilieren das Programm und verteilt diejenigen Programmabschnitte, die voneinander unabhängig bearbeitet werden können. Als Vertreter dieser Gruppe lassen sich `Fortran90`, `ParLog` und AUC-C^{++}-Linda [Thomas, 1991] nennen. Zumeist wird die Analyse und Optimierung des Programms beim kompilieren in einer eigenen Phase durchgeführt. Das Verteilungspotential hängt dabei sehr von der Programmstruktur ab.

- *Die Verteilung wird durch Aussnutzen von Sprachmechanismen erreicht:*
  Bei den funktionalen Programmiersprachen wird der inhärente Parallelismus nichtprozeduraler Programmiersprachen [Taudes, 1991, vgl. S. 106] bzw. eine entsprechende Dekompositionsmethode (näheres siehe Abschnitt 9.1.1) als Ausgangspunkt für die Verteilung von Applikationen genommen (siehe dazu die Implementation des verteilten Each-Operators in APL [Mitlöhner, 1993]). Es wird durch Kapselung möglichst ohne globale Daten, versucht, die Lösung des Problems möglichst verteilbar zu machen.

  Bei logischen verteilten Programmiersprachen (Parlog, Concurrent Prolog) wird der inhärente AND/OR Parallelismus logischer Programme ausgenutzt [Taudes, 1991, vgl. S. 107] (vgl. dazu auch [Carriero and Gelernter, 1989b]). Es lassen sich z.B. bei Prolog die Klausen parallelisiert nach Termen (AND), nach Klausen (OR) oder nach

beiden Gesichtspunkten parallelisieren. Treten term- oder klausenübergreifende Variable auf, müssen diese entweder wie Shared Variables behandelt werden, oder die parallele Verarbeitung muß für diese Untergruppe in sequentieller Form durchgeführt werden.

Die meisten Erweiterungen von Programmiersprachen hinsichtlich verteilter Verarbeitung sind ohne spezielle Runtime-Systeme bzw. spezielle Hardware nicht einzusetzen. Es ist mit ihnen ohne größeren Aufwand auch keine Kommunikation und Koordination zwischen den einzelnen Programmiersprachen möglich. Einzelne Implementationen wie [Kolarik, 1993] bieten die Möglichkeit der Verwendung von Shared Variables von verschiedenen Programmiersprachen, doch können wiederum für die Koordination von Prozessen nur über explizit programmierte Mechanismen (Flags, etc.) herangezogen werden.

4 Linda

In diesem Abschnitt wird *Linda*, ein Werkzeug zur Erweiterung von Programmiersprachen für die Kooperation und Kommunikation zwischen Prozessen, vorgestellt. Nach einer kurzen Einführung und Erklärung der grundsätzlichen Funktionsweise von Linda erfolgt ein Überblick über die Entwicklung. Dieser Überblick umfaßt auch die bisherigen Einsatzgebiete von Linda und Linda-ähnlichen Umgebungen.

Nach dieser allgemeinen Einführung wird *Perl-Linda* beschrieben, eine sprach- und hardwareunabhängige Implementierung von Linda die auf Basis eines Client/Server basierten Runtime-Systems eine einheitliche Schnittstelle für die Erweiterung von Programmiersprachen mittels einer Linda-Client-Schnittstelle bietet. In den folgenden Unterabschnitten erfolgt eine Erklärung der Funktionsweise von Perl-Linda und eine Abgrenzung gegenüber bisher existierenden Linda-Implementierungen. Es werden die einzelnen Teile von Perl-Linda und deren Zusammenwirken erklärt.

4.1 Was ist Linda?

Linda ist ein kleiner Satz von Befehlen, welcher zu einer beliebigen Programmierprache hinzugefügt wird und deren Fähigkeiten zur Kommunikation und Kooperation mit anderen Prozessen erweitert. D. Gelernter entwickelte Linda als Konzept für die Kooperation und den Datentausch zwischen Prozessen. In seiner Grundstruktur besteht Linda aus einem von allen Prozessen gemeinsam nutzbaren Speicherbereich, dem *Tuplespace*. Im Tuplespace werden Datenelemente, sogenannte *Tuple*, gespeichert. Prozesse führen auf dem Tuplespace Schreib- und Leseoperationen durch und können so den Inhalt des Tuplespace modifizieren. Ein Teil der Leseoperationen blockieren solange, bis im Tuplespace die benötigten Tuple zur Verfügung stehen. Diese Eigenschaft wird zur Koordination und Synchronisation von Prozessen eingesetzt.

Der Befehlssatz von Linda besteht aus drei Grundbefehlen [Gelernter, 1985], in denen die Synchronisation noch nicht gelöst ist:

- `out()`: stellt ein Tuple in den Tuplespace.

- `rd()`: liest Tuple vom Tuplespace. Der Befehl blockiert, wenn kein entsprechendes Tuple gefunden ist.
- `in()`: entfernt Tuple aus dem Tuplespace. Der Befehl blockiert, wenn kein entsprechendes Tuple gefunden ist.

In [Carriero and Gelernter, 1989a] wurde dieser Befehlssatz um drei weitere Befehle erweitert:

- `inp()`: funktioniert wie `in()`. Anders als bei `in()` blockiert der Befehl nicht, wenn kein passendes Tuple gefunden wird.
- `rdp()`: funktioniert analog zu `inp()`.
- `eval()`: stellt ein aktives Tuple in den Tuplespace. Die Elemente dieses Tuple können Programme darstellen, die evaluiert werden.

In der Interaktion in einem Linda-Programm verhält sich der Tuplespace passiv, d.h. er reagiert nur auf die von den Prozessen abgesetzten Linda-Befehle und sendet den Prozessen die Ergebnisse der Befehle zurück. Ein Schema der Interaktion zwischen Prozessen und Tuplespace ist in Abbildung 7 dargestellt.

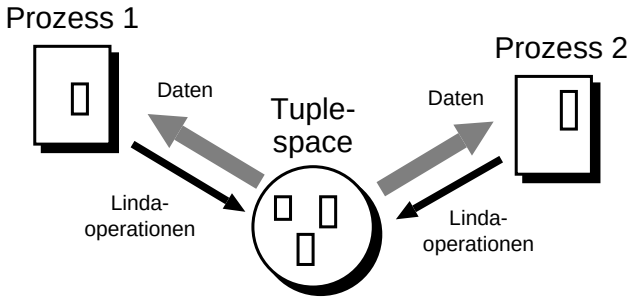

Abbildung 7: Grundprinzip von Linda

Die dargestellten Prozesse greifen mit den durch Linda zur Verfügung stehenden Operationen auf den gemeinsamen Tuplespace zu. Da die Befehle `rd()` und `in()` zu einem Blockieren der zugreifenden Prozesse führen, wenn keine passenden Daten im Tuplespace vorhanden sind, können nicht nur Daten ausgetauscht werden, sondern es kann auch eine Koordination der Prozesse erfolgen.

Von den herkömmlich verwendeten Mechanismen wie *Shared Variables*, *Message Passing* und *Remote Procedure Calls* unterscheidet sich Linda durch mehrere Punkte:

- Linda ermöglicht verteilte Programmierung sowohl in räumlicher als auch in zeitlicher Hinsicht.
- In vielen Fällen produziert der Einsatz von Linda durch die einfache Struktur eine leichter verständliche Lösung als der Einsatz von herkömmlichen Techniken. Der resultierende Algorithmus ist intuitiver verständlich, da die programmtechnische Seite der Kommunikation wegfällt.
- Durch die Verwendung von Linda kann selbst Programmieranfängern verteilte Verarbeitung nähergebracht werden. Durch die geringe Anzahl von Befehlen kann der Befehlssatz schnell erlernt werden. Es ist daher bereits in Einführungskursen möglich, verteilte Anwendung mit Linda zu demonstrieren.

Durch seine modulare Struktur und seine Flexibilität in der Anwendungsprogrammierung unterscheidet sich Linda von herkömmlichen Ansätzen, Programme zu verteilen. Da Linda orthogonal zur verwendeten Programmiersprache ist, kann der Befehlssatz von Linda leicht zu einer Programmiersprache hinzugefügt werden, ohne die normale Funktion der Sprache zu beeinflußen.

Gelernter [Gelernter, 1985, vgl. S.85ff.] unterstreicht folgende Eigenschaften von Linda. Diese Eigenschaften ermöglichen es, Linda flexibel zur Koordination mehrerer Prozesse und zum Datentausch zwischen diesen einzusetzen.

Space uncoupling (distributed naming):
Linda ermöglicht eine Kommunikation zwischen Prozessen. Tuple können als Eingabe für mehrere Prozesse verwendet werden. Genauso können die Ergebnisse mehrerer Prozesse für den Inhalt eines Tuple verantwortlich sein. Der *Name* eines Tuple ergibt sich, abgesehen von *Singletons* (einelementige Tuple), als Summe aller Elemente des Tuple.

Time uncoupling:
Im Linda-Konzept ist der Tuplespace als eine von den Prozessen getrennte Einheit zu sehen. Das bringt den Vorteil, daß von den Prozessen im Tuplespace abgelegte Tuple nicht an die sendenden Prozesse gebunden sind.

Ein mittels eines `out()`-Statement abgelegtes Tuple ist nicht an die Lebensdauer des sendenden Prozesses gebunden. Es steht daher den anderen Prozessen auch nach der Beendigung des erzeugenden Prozesses zur Verfügung. Diese Eigenschaft wird mit *Persistenz* bezeichnet.

Distributed Sharing:
Die Struktur von Linda erlaubt die Erzeugung von Shared Variables, welche von mehreren Prozessen geteilt verwendet werden. Eine Variable wird in diesem Fall als ein 2-Tuple mit der Struktur *(Name,Wert)* dargestellt. Im Unterschied zu anderen Implementationen [Fountain, 1994, Kolarik, 1993] muß der Programmierer der Applikation keine Schutz- oder Lockingmaßnahmen vorsehen. Diese werden durch die Verwendung der Linda-Operationen automatisch zur Vergfügung gestellt. Der Befehlssatz von Linda ermöglicht eine automatische Zugriffskoordination auf die Variablen.

Support for Continuation Passing:
Bei der Koordination von mehreren Prozessen kann mitunter die Notwendigkeit auftreten, daß die Steuerung eines blockierenden Prozesses von einem Kontrollprozess an einen anderen weitergereicht wird. Linda unterstützt dies durch eine flexible Tuple-Struktur, in welcher neben eigentlichen Daten auch Steuerelemente gespeichert werden können.

Es wird dadurch die Möglichkeit der Steuerung eines Prozesses durch andere Prozesse geschaffen. Zu diesen Steuerelementen zählen das Stoppen, das Wiederaufnehmen der Verarbeitung und in weiterer Form auch die Übermittlung von Programmen. Diese Steuerungsmöglichkeit muß aber vom Programmierer explizit vorgesehen sein.

4.2 Tuple und Tuplespace

Die kleinsten Elemente von Linda sind Tuple. Ein Tuple besteht aus einem oder mehreren Elementen, welche die eigentlichen Daten enthalten. Tuple müssen Information beinhalten, da in leeren Tuple keine Information gespeichert werden kann und auch als Platzhalter für sie keine sinnvolle Verwendung in Linda ist. Bei den Elementen unterscheidet Gelernter [Carriero and Gelernter, 1989a] zwischen *aktualen* und *formalen* Elementen.

aktuales Element:
Bei dieser Kategorie ist der Inhalt des Elements bestimmt. Es kann z.B. ein String, eine Zahl, ein Objekt u.ä. sein.

formales Element:

Formale Elemente stehen als Platzhalter für ihre akutalen Gegenstücke. Ein formales Element kann durch jedes aktuale Element ersetzt werden. Verwendung finden formale Elemente in sogenannten *Pattern*, in denen sie als Templates für das Finden von Tuple eingesetzt werden. Ein formales Element wird in Linda durch das Zeichen `'?'` gekennzeichnet.

Im Tuplespace dürfen nur Tuple mit aktualen Elementen enthalten sein. Tuple mit formalen Elementen werden nur bei Linda-Operationen zum Abfragen der Inhalte des Tuplespace eingesetzt.

Für die Darstellung der Datentypen von Elementen wie z.B. `float`, `int` etc. wird in den meisten Linda-Implementationen der Weg über die ASCII-Form gegangen, d.h., alle Daten werden als Zeichenketten dargestellt. Die eigentliche Typdefinition erfolgt durch Zusätze zu den Elementen mit von Implementation zu Implementation wechselnder Syntax. Da in den verschiedenen Linda-Implementationen unterschiedliche Wege der Prädetermination der Datentypen gegangen wurden, sind die folgenden Beispiele kompatibel zu Perl-Linda gehalten. Sie könnten jedoch auch in den meisten anderen Linda-Implementationen ohne größere Änderungen dargestellt werden. In Abbildung 8 werden einige Beispiele für Tuple dargestellt.

```
(This is a Singleton)
(This,is,a,5,-,Tuple)
(Data,int,5,string,test,float,5.4)
```

Abbildung 8: Beispiele für Tuple

Die kleinste Form eines Tuple ist ein *Singleton*, welches im besonderen für Koordinationszwecke für Prozesse eine besondere Aufgabe hat. Im weiteren sehen wir Tuple mit Elementen verschiedenen Typs, was jedoch keinen Einfluß auf die Bearbeitung hat, da die gesamten Daten als ASCII-Zeichenkette dargestellt werden. Das dritte Beispiel gibt ein Beispiel für eine zusätzliche Typbestimmung, wie sie z.B. bei [Gelernter, 1985] [Callsen *et al.*, 1991] vorgeschlagen und implementiert wird.

Im Tuplespace können mehrere gleiche Tuple existieren. Die enthaltenen Tuple unterliegen keiner Reihenfolge, die Anordung im Tuplespace ist zufällig. Prinzipiell ist die Länge der Elemente und die Anzahl der Elemente im Tuple unbegrenzt. Abhängig von der der Implementation zugrundeliegenden Programmiersprache

oder des zugrundeliegenden Systems gibt es jedoch Einschränkungen. Für Perl-Linda finden sich diese Einschränkungen im Abschnitt 6.6.

Pattern werden aus Tuple gebildet, in denen formale Elemente vorkommen können. Jedes vorkommende formale Element kann durch jedes beliebige Element ersetzt werden. Ein kleines Beispiel:

> Es kann die Variable *myvar* mit dem Wert `15` durch das Tuple `(myvar,15)` im Tuplespace dargestellt werden. Will ein Prozeß auf diese Variable zugreifen, braucht er einen entsprechenden Pattern dazu. Wenn dieser Pattern in einer `rd()`- oder `in()`-Operation angewendet wird, soll er die gespeicherte Variable als Ergebnis liefern. Dieser Pattern läßt sich aus einem Tuple bilden, dessen erstes Element das aktuale Element `"myvar"` und dessen zweites Element ein formales Element ist. Das formale Element wird dann durch den Wert `15` ersetzt. Das benötigte Pattern sieht folgendermaßen aus:
>
> `("myvar", ?)`

Die Syntax und die Funktionsweise der Ersetzung differieren stark in den verschiedenen Versionen von Linda. Die Verwendung von Pattern in Perl-Linda und der genaue Ersetzungsmechanismus wird in Abschnitt 6.4 ausführlicher beschrieben.

4.3 Der Linda-Befehlssatz

Der Linda-Befehlssatz umfaßt sechs Befehle zum Manipulieren der Daten im Tuplespace. Die Befehle werden von den Prozessen aufgerufen. Der Tuplespace führt die Befehle aus und sendet die entsprechenden Daten an den Prozeß zurück. Die Befehle gliedern sich in Ausgabe-Befehle zum Senden von Daten an den Tuplespace und in Eingabe-Befehle, mit denen sich Daten aus dem Tuplespace entnehmen lassen. In der folgenden Gliederung sind die Befehle mit ihrer grundsätzlichen Funktion dargestellt (vgl. dazu [Carriero and Gelernter, 1989a]):

Ausgabe-Befehle:

Diese Gruppe umfaßt zwei Befehle. Durch diese kann der Prozess Daten in Form von Tuple an den Tuplespace senden. Der Befehl `out(`*tuple*`)` stellt

das Tuple *tuple* in den Tuplespace. Dieses Tuple darf nur aus aktualen Elementen bestehen und ist somit nicht mehr veränderbar.

Im Gegensatz zu `out()` stellt der Befehl `eval(`*tuple*`)` ein aktives Tuple in den Tuplespace. Bei einem aktiven Tuple müssen die Elemente nicht aktual sein, sondern können auch Operationen (Funktionen etc.) enthalten, deren Wert erst berechnet werden muß. Diese Ausdrücke werden dann evaluiert. In welcher Weise `eval()` ausgeführt wird, hängt von der jeweiligen Implementation ab.

Eingabe-Befehle:

Diese gliedern sich in zwei Gruppen: Befehle, die Tuple im Tuplespace nur lesen können und Befehle, die diese auch entnehmen können. Es wird im Tuplespace nach auf einen als Parameter mitgegebenen Pattern passende Tuple gesucht. Die Befehle `rd(`*pattern*`)` und `rdp(`*pattern*`)` lesen alle auf den Pattern *pattern* passenden Tuple und geben sie an den aufrufenden Prozeß zurück. Während `rd()` solange blockiert, bis mindestens ein passendes Tuple im Tuplespace vorhanden ist, blockiert `rdp()` nicht und gibt gegebenenfalls auch kein Tuple zurück.

Die Befehle `in(`*pattern*`)` und `inp(`*pattern*`)` funktioniern analog zu `rd()` und `rdp()` mit dem Unterschied, daß sie die Tuple auch aus dem Tuplespace entfernen.

4.4 Vergleich zu anderen Kommunikationsansätzen

Im Vergleich zu anderen Kommunikationslösungen ergibt sich daher folgendes Schema, welches in Tabelle 3 abgebildet ist.

Die vorgestellten Ansätze implementieren die angeführten Punkte jedoch nicht ohne Probleme. Es sind der Aufstellung daher noch folgende Punkte hinzuzufügen:

Dateien:

Die Kommunikation zwischen den Prozessen erfolgt bei diesem Ansatz über eine oder mehrere Dateien. Die Daten und Nachrichten werden auf diese gespeichert und von diesen gelesen.

Bei verschiedenen verteilten Betriebssystemen kann eine Datei vollständig in einen verteilten Speicherbereich geladen werden. Das Betriebsystem er-

Methode	Zeit		Host		Koordination	Persistenz der Daten
	gleichzeitig	verschieden	gleich	verschieden		
Kommunikation mittels Dateien	•	•	•	(•)		(•)
Remote Procedure Call	•		•	•		
Berkeley Sockets	•		•	•		
Shared Variables	•		•	•	(•)	(•)
Linda	•	•	•	•	•	•

Tabelle 3: Kommunikationsschema zwischen Prozessen

ledigt dann den Zugriff und das Konsistenthalten der enhaltenen Daten (siehe dazu Abschnitt 3.5.1). Bei Verwendung eines *normalen* Betriebssystem muß man aber andere Wege gehen.

Bei der Kommunikation über Dateien wird bei Prozessen auf unterschiedlichen Hosts unterstellt, daß diese über ein Netzwerk-Dateisystem wie z.B. NFS verbunden sind. Eine solche Verbindung bringt jedoch eine Lücke in der Systemsicherheit mit sich, da alle Prozesse, die Daten aus den Dateien benötigen, zumindest Lesezugriff auf das Filesystem haben müssen. Wenn die Prozesse die Daten auch noch verändern können, muß ihnen zusätzlich zum Lesezugriff auch noch ein Schreibzugriff eingeräumt werden. Aus der Sicht der Systemsicherheit ist dies eine äußerst bedenkliche Lösung.

Neben der Systemsicherheit ergeben sich bei häufigen Zugriffen auch Performance-Probleme, da ein Dateizugriff an sich langsam ist und relativ große Systemressourcen verbraucht.

Remote Procedure Calls (RPC):

RPCs [Comer and Stevens, 1993, S. 233ff.] bieten eine relativ elegante Lösung für das Prozeßmanagement auf anderen UNIX-Systemen an. Der Prozeß wird durch den Aufruf einer Library-Routine (Stub) gestartet, welche das gleiche Interface wie die äquivalente Routine auf dem Ziel-Computer hat. Die Parameter und im Gegenzug das Ergebnis werden in eine Nachricht gepackt und über das Netzwerk versandt.

Für die Anwendung von RPC sind aber spezielle Zugangsberechtigungen zwischen den Rechnern erforderlich. Dies liegt nicht immer im Sinne des Ausführenden. Weiters lassen sich RPC Prozesse nur auf Reply/Response-Basis starten, eine andere Kommunikationsform ist über RPC nicht

möglich. Der aufrufende Prozeß blockiert daher so lange, bis der aufgerufene Prozeß beendet ist.

Sockets:

Berkeley Sockets [Comer and Stevens, 1993, S. 43ff.] stellen den auf UNIX-Systemen üblichen Weg zur Implementation von Kommunikation zwischen Client- und Serverprozessen auf Basis von Message Passing dar. Sie liegen auch der Implementation von Perl-Linda (siehe 5) zugrunde. Neben der reinen Kommunikationsfunktion wird aber keine Möglichkeit geboten, Prozesse zu koordinieren. Diese muß in den Prozessen explizit implementiert werden.

Es entsteht dadurch die Notwendigkeit, der Kommunikation zwischen den Prozessen ein geeignetes Protokoll zugrundezulegen, welches die Möglichkeit einer Koordination und Synchronisation ermöglicht. Weiters gilt die Beschränkung auf die Gleichzeitigkeit bei der Kommunikation, d.h. beide kommunizierenden Prozesse müssen zum Kommunikationszeitpunkt aktiv sein. Es besteht keine Möglichkeit einer globalen Speicherung von Daten. Diese müssen in einem der Prozesse lokal gehalten werden.

Shared Variables:

Shared Variables implementieren eine Möglichkeit, Daten zwischen Prozessen in heterogenen Sprachumgebungen durch einen zwei Prozessen zugänglichen Speicherbereich auszutauschen. Besonderes Augenmerk erfordert die interne Darstellung der Variablen (Datenobjekte), da diese Darstellung die Basis für die Implementation der Zugriffs- und Änderungsmöglichkeit in allen Sprachen bildet.

In [Janko, 1981, vgl. S. 317ff.] wird vorgeschlagen, diese Aufgabe von einem Supervisor ausführen zu lassen, der die Zugriffsrechte auf den Speicherbereich und die Variablen verwaltet.

Netzwerklösungen mit Shared Variables basieren auf Sockets, wobei hier ein Manager (Daemonprozeß) gestartet wird, der die auf dem Netzwerk befindlichen Clients servisiert. Zugriffschutz auf die nutzbaren Daten ist in der Applikation mittels Zugriffskontrollvektoren implementiert die ihrerseits intern Semaphore verwenden. Der Managerprozeß ist in diesem Falle auch für den Zugriffsschutz verantwortlich (vgl. dazu [Kolarik, 1993] [Janko, 1981]).

4.5 Linda-Implementationen und Linda-ähnliche Systeme

Das Tuplespace-Konzept wurde in einer Reihe von Sprachen auf unterschiedlichen Computertypen bereits implementiert. Im folgenden soll eine Aufstellung über bereits implementierte Linda-Systeme und deren Anwendungsgebiet gegeben werden.

4.5.1 Literatur- und Forschungsüberblick

Ausgehend von den ersten Publikationen über Linda an der Yale University [Gelernter, 1985] [Carriero and Gelernter, 1985] [Carriero *et al.*, 1986] [Carriero and Gelernter, 1989b] hat Linda seinen Weg in Forschungsstätten über den ganzen Erdball geschafft. Gelernter beschreibt in seinem populärwissenschaftlichen Buch *Mirror Worlds* [Gelernter, 1992] die Entwicklung von Linda derart:

> *"After the knock-down battles of the last decade, Linda is one of the few contenders left on her feet. There is perhaps half a dozen systems on the contenders list, as of today. Linda has been taken up by a number of large companies, gets used at labs and universities all over the world and has inspired computer science research projects in North and South America, Europe, Asia, Africa and Australia. In short, most continents."*

Wenn man die Literatur zu Linda nach der Entstehung betrachtet, stellt man fest, daß gut die Hälfte der existierenden Literatur von der *Yale Linda Group*, einer Forschergruppe um Gelernter, Carriero, Leichter, Minsky et al. an der Yale University entstanden ist. Von dieser Gruppe geht auch die Entwicklung eines Standards von grundsätzlich zu implementierenden Linda-Befehlen aus. Eine weitere größere Gruppe befindet sich am Edinburgh Parallel Computing Centre. In Italien arbeitet P. Caincarini mit einer internationalen Projektgruppe an der Weiterentwicklung von Linda. Auf den anderen Kontinenten existieren Publikationen, doch läßt sich daraus keine zusammenhängende Forschung erkennen. Eine der jüngsten Forschungsgruppen ist die *WU-Linda Group* an der Wirtschaftsuniversität Wien.

4.5.2 Implementierungen und Forschungsschwerpunkte

Wenn auch die Forschung über Linda keine weltweite Koordination und auch kein Konsortium hat, welches sich mit der Erhaltung und Definition der Standards beschäftigt, hat die Forschung doch gewisse Schwerpunkte. Im folgenden erfolgt ein Überblick nach der folgenden Gliederung:

1. nach Hard-/Software-Plattformen.
2. nach der Verteilung der Daten auf die Tuplespaces.
3. nach den Schnittstellen zu Programmiersprachen.

4.5.3 Gliederung nach Hard-/Software-Plattformen

Workstation:

Die ersten Implementationen von Linda dienten dazu, durch den Einsatz von Linda Datentausch und Koordination zwischen Prozessen auf **einem** Rechner zu ermöglichen. Die Implementationen für Einprozessorrechner halten sich aufgrund der fehlenden Möglichkeiten zur Verteilung in einem sehr eingeschränkten Rahmen. Vertreter dieser Gruppe sind z.B. [Sutcliffe and Pinakis, 1992] und [Gelernter, 1985].

Parallelrechner und Netzwerkrechner:

In diese Gruppe fallen sehr systemspezifische Implementationen von Linda. Die meisten von ihnen setzen auf einer sehr tiefen Systemebene auf und sind daher schlecht zu portieren. Es sind in dieser Gruppe Systeme mit geteiltem und verteiltem Speicherbereich zu unterscheiden. Manche Systeme wie [Butler *et al.*, 1993] setzen ihrerseits auf speziellen Umgebungen auf, welche eine Schnittstelle zu der jeweiligen Rechnerarchitektur darstellen.

Diese Systeme sind in der Anwendung zumeist auf die Verwendung der gleichen Programmiersprache für alle Applikationsteile und auf ein Netzwerk von homogenen Rechnern (Prozessoren) beschränkt. Systeme sind in [Carriero and Gelernter, 1989b], [Callsen *et al.*, 1991], [Bakken, 1993], [Hasselbring, 1993], [Ciancarini, 1993b], [Ciancarini, 1993a] und [Siegel and Cooper, 1991] beschrieben.

UNIX-System mit LAN:

Die meisten Linda-Systeme sind für Netzwerkrechner konzipiert. Die

Verwendung von UNIX-Workstation und eines normalen LANs ermöglichen hingegen nur wenige Linda-Derivate. Verschiedene Implementationen auf Kernel-Ebene existieren. Die meist in *C* geschriebenen Anwendungen haben aber einen eingeschränkten Funktionsumfang. Beispiele für die Beschränkungen sind Befehlsumfang bzw. Beschränkung der formalen Elemente im Tuple und die fehlende Erweiterbarkeit bezüglich anderer Programmiersprachen. Zu ihren Vertretern zählen [Narem, 1990], [Leler, 1990], [Thomas, 1991], [Ciancarini and Guerrini, 1993] und [Mattson *et al.*, 1992].

Kommerzielles Linda:
Eine kommerzielle Version von Linda wurde für die Sprachen *C* und FORTRAN von der *Scientific Research Group* entwickelt. Diese Version basiert auf den *C*-Linda Versionen von Gelernter und Carriero und implementiert einen Single-Tuplespace auf einem Parallelrechner oder einem Netzwerk von UNIX-Workstations. Die Software umfaßt den normalen Befehlsumfang von Linda und eine Implementation des *eval()*-Befehls. Dieser wird mittels RPC durchgeführt.

Linda-Programme werden mit einem Linda-Precompiler in *C*-Code übersetzt und danach mit einem normalen *C*-Compiler kompiliert. Dies erfordert eine Neuübersetzung der Gesamtanwendung bei jeder Änderung. Zusammen mit dem Linda-Runtimesystem kann das Programm dann ausgeführt werden. Es werden noch Tools für Debugging und Monitoring mitgeliefert. Das System erfordert eine spezielle Konfiguration auf den beteiligten Rechnern hinsichtlich des Zugriffs über RPC. Eine Ausweitung des Sprachumfanges und eine Anbindung an weitere Sprachen, insbesondere Skriptsprachen ist weder möglich noch vorgesehen. Die Anzahl der Elemente im Tuple ist mit 16 Elementen limitiert. Als weitere Problemfelder werden in heterogenen Rechnerumgebungen *Byte-Order* der Maschine, die interne Repräsentation von Gleitkommazahlen und die interne Darstellung von Arrays bzw. Strukturen angegeben. Für weiterführende Information siehe [Sci, 1995].

4.5.4 Gliederung nach der Verteilung der Daten auf Tuplespaces

Die Einteilung folgt der in [Kahn and Miller, 1989, S. 1255] getroffenen Einteilung nach der Anzahl der Tuplespace:

S-Linda (1 Tuplespace):
Die Lösung mit einem zentralen Tuplespace wird in allen Im-

plementationen von Linda als erste Ausbaustufe implementiert. Mit einem einzelnen Tuplespace läßt sich neben Laufzeitüberlegungen das Kriterium der Fault-Tolerance gegenüber Abstürzen des Rechners, auf dem der Tuplespace läuft, nicht erfüllen. Der einzelne Tuplespace stellt sich bei sehr feinkörniger Aufteilung des Problems auch zumeist als Engpaß heraus, da die gesamte Kommunikation und Koordination zwischen den einzelnen Programmteilen über diesen Tuplespace erfolgt.

Als weiteres Manko eines zentralen Tuplespace wird auch angesehen, daß die über den Tuplespace kommunizierenden Applikationen sich gegenseitig beeinflussen. So kann es vorkommen, daß eine Applikation einer anderen wichtige Tuple aus dem Tuplespace entfernt, was bei der anderen Applikation zu Deadlocks führen kann. Es kann daher für eine Linda-Applikation nicht gezeigt werden, daß sie zu einem Ergebnis kommt, ohne daß man das Verhalten aller anderen auf demselben Tuplespace laufenden Applikationen kennt. Zu den Implementierungen von Linda mit einem Standalone-Tuplespace zählen [Gelernter, 1985], [Carriero and Gelernter, 1989a] und [Leler, 1990].

M-Linda (mehrere Tuplespace):

Der Einsatz mehrerer Tuplespaces bietet zwei grundsätzliche Möglichkeiten die über das Linda-Konzept hinausgehen:

- Die Verfügbarkeit der Tuplespaces für die zugreifenden Prozesse kann erhöht werden. Es liegt hierbei die Annahme zugrunde, daß ein Tuplespace nur eine gewisse Anzahl von Prozessen servisieren kann.
- Die Datensicherheit kann erhöht werden, in dem der Tuplespace-Inhalt auf mehrere Tuplespace repliziert werden kann.

Dies sind auch die zwei Grundrichtungen, in welche die Forschung und Entwicklung in Verbindung mit mehreren Tuplespaces geht. Bei der Replikation von Tuplespaces steht das Streben nach Ausfallsicherheit im Vordergrund. Ein Nebeneffekt ist auch, daß über replizierte Tuplespaces mehr Clients servisiert werden können, da die Anzahl der Clients bei einem einzelnen Tuplespace einen Engpaß darstellt. Bei der verteilten Datenhaltung wird das Ziel verfolgt, das Suchen passender Tuple im Tuplespace zu parallelisieren. Daher werden die Daten auf mehrere Tuplespace aufgeteilt. Die Requests der Clients müssen daher

aber auch verteilt behandelt werden.

Die Verwendung mehrerer Tuplespaces mit der Möglichkeit zur Replikation der Inhalte ist in [Bakken, 1993] und [Bakken and Schlichting, 1995] beschrieben. Verteilte Datenhaltung in mehreren Tuplespace ist in [Gelernter, 1989], [Wilson, 1991a], [Pinakis, 1991], [Ciancarini, 1993b] und [Ciancarini, 1993a] ausgeführt.

4.5.5 Gliederung nach den Schnittstellen zu Programmiersprachen

C, C^{++}:

Die ersten Implementierungen von Linda basierten auf der Programmiersprache C ([Gelernter, 1985] und [Carriero and Gelernter, 1989a]). C verbindet die Möglichkeit der systemnahen Programmierung mit der Erweiterbarkeit der Sprache. Die meisten auf C und C^{++} basierenden Versionen von Linda wurden durch den Einsatz eines Linda-Precompilers implementiert. Dieser analysiert, optimiert und übersetzt das Linda-Programm in reinen C- bzw. C^{++}-Sourcecode. Dieser wird dann in einem zweiten Schritt mit einem regulären Compiler in das ausführbare Programm übersetzt. Die Optimierung erfolgt in bezug auf die Abarbeitung von `eval()`-Befehlen [Carriero and Gelernter, 1991] und auf das Matching von Tuple [Callsen *et al.*, 1991]. Diese Implementationen bedingen eine homogene Programmiersprache in den Clients und eine Neuübersetzung des Gesamtsystems bei Änderungen an einem Client, erlauben aber eine automatische Optimierung der Performance durch Analyse der durchzuführenden Linda-Operationen.

Abweichend von dieser Methode wird Linda auch mit Library-Funktionen implementiert [Thomas, 1991]. Weitere auf C basierende Implementationen finden sich in [Narem, 1990], [Leler, 1990], [Thomas, 1991], [Butler *et al.*, 1993] und [Bakken, 1993].

Prolog:

Der Vergleich von Linda mit *Concurrent Logic Programming* in [Carriero and Gelernter, 1989b] lenkte die Aufmerksamkeit der Prolog-Welt auf Linda. Die in Prolog inherente Unifikation bietet Synergieeffekte beim Matching von Tuple und bei der Aus-

wertung von `eval()`-Befehlen. Implementierungen von Linda in Prolog finden sich in [Ciancarini, 1993b], [Ciancarini, 1993a] und [Sutcliffe and Pinakis, 1992].

Sonstige:

Die sonstigen Implementierungen sind Kombinationen von Linda mit anderen als den oben explizit ausgewiesenen Programmiersprachen und Prototyping-Sprachen. Siehe dazu z.B. [Butcher, 1991], [Siegel and Cooper, 1991], [Hasselbring, 1991] und [Hasselbring, 1993].

5 Perl-Linda

Dieser Abschnitt befaßt sich mit der Implementation von Linda in der Programmiersprache Perl. Nach einer Übersicht über die Implementation wird das Interface zu Perl-Linda beschrieben. Es wird die Darstellung von *Tuple*, *Pattern* wie auch *Tuplelist* im Server, sowie die Plattform-unabhängige Vercodierung der Daten beschrieben. Danach wird der Befehlssatz des Perl-Linda-Servers erklärt. Die Funktionsweise der Befehle wird an kurzen Beispielen demonstriert.

5.1 Das Perl-Linda-Interface

Perl-Linda ist eine Implementation von Linda beruhend auf der Sprache Perl V4.036 [Wall and Schwartz, 1990]. Von den in Abschnitt 4.5 vorgestellten Implementationen unterscheidet sich der Ansatz durch drei Punkte:

- Während die meisten Linda-Implementationen für Parallelrechner oder Closely Coupled Systems entwickelt wurden, ist Perl-Linda für ein Netzwerk von Einprozessor-Computern implementiert worden. Daher ist es auf jedem Unix-System lauffähig. Die Voraussetzungen für die Installation sind das Vorhandensein von Berkeley Sockets und Perl4.036.
- Perl-Linda wurde für die Applikationsebene konzipiert und ist nicht Bestandteil des Betriebssystems. Das auf Client-Serverbasis laufende Perl-Linda erlaubt gemischtsprachigen Clients die Eigenschaften von Linda bezüglich Koordination und Datentausch gemeinsam zu nutzen. Perl-Linda kommt ohne jeden Precompiler aus und stellt eine Schnittstelle für Clients als Unterprogrammbibliotheken in den jeweiligen Programmiersprachen zur Verfügung. Daher ist es einfach auf andere Systeme und Programmiersprachen portierbar.
- Durch die Verwendung einer standardisierten Darstellung der Daten im Server ist die Implementation einer Schnittstelle für verschiedene Programmiersprachen einfach durchzuführen und ist für einige Sprachen bereits erfolgt. Daher ist in dieser Implementierung die Nutzung von Linda auch bei heterogen-sprachigen Clients möglich, d.h., Clients in unterschiedlichen Programmiersprachen können sich über den Perl-Linda-Server koordinieren und Tuple gemeinsam bearbeiten. Dies ermöglicht gemischtsprachige

Applikationen, wobei auf die Stärken der jeweiligen Programmiersprache fokussiert werden kann.

In seiner Grundstruktur besteht Perl-Linda aus zwei Komponenten, Server und Client, welche die Struktur von Linda modellieren (vgl. Abbildung 7). Der Server repräsentiert den Tuplespace. Seine Aufgabe ist das Verwalten der Client-Verbindungen und der im Tuplespace gehaltenen Daten. Weiters bearbeitet er die von den Clients ankommenden Linda-Operationen und hält diese in Evidenz.

Die Clients stellen die Schnittstelle zur jeweiligen Programmiersprache dar. Die Aufgabe der Clients ist die Umwandlung der Tuple für die Linda-Operationen, welche in den jeweiligen Sprachen unterschiedlich repräsentiert werden, in die Darstellung des Servers. Er stellt somit eine Schnittstelle zu Linda für die jeweilige Sprache dar. Neben der Transformation der Daten übernimmt der Client auch noch das Blockieren des Linda-Programmes bei den entsprechenden Kommandos.

Der Befehlsatz von Perl-Linda stellt eine Erweiterung des in Abschnitt 4.3 beschriebenen Original-Linda-Befehlssatzes dar und gliedert sich in zwei Gruppen von Kommandos:

Linda Funktionen:

In dieser Gruppe sind alle Funktionen enthalten, welche Transaktionen mit Tuple durchführen. Durch den Aufruf einer solchen Funktion vom Client werden die Daten im Tuplespace bearbeitet und fallweise verändert. Diese Funktionen stellen den minimalen Befehlssatz dar, der beim Bau eines Client-API in einer Programmiersprache implementiert werden sollte.

Neben den von Gelernter in [Gelernter, 1985] und [Carriero and Gelernter, 1989a] beschriebenen Grundbefehlen sind auch noch einige weitere Funktionen, welche sich bei der Erstellung von Applikationen als sinvoll und brauchbar herausgestellt haben, enthalten. Diese erweiterten Funktionen lassen sich auch aus dem Original-Linda-Befehlssatz konstruieren, doch bringt ihre explizite Implementierung Vorteile hinsichtlich der Laufzeit und des Datentransfers zwischen Client und Server.

Server Funktionen:

Durch die Arbeit mit der Client-Server Architektur von Perl-Linda ergaben sich in der Entwicklungsphase öfters Situationen und Probleme, welche mit Hilfe des Linda-Befehlssatzes nicht zufriedenstellend und ein-

fach genug zu lösen waren. Bei diesen Problemen handelte es sich meistens um Schwierigkeiten bei der Administration der Server-Client Umgebung und um Debugging-Probleme beim Entwickeln und Testen von Linda-Applikationen.

Um eine Hilfestellung für den Administrator/Entwickler zu bieten, wurden Perl-Linda noch eine Anzahl von weiteren Befehlen hinzugefügt, welche zur Administration, zum Testen und zum Monitoring durch einen *menschlichen* Client dienen. Diese Server-Funktionen müssen nicht obligatorisch in das Client-API implementiert werden.

Die Namen der Funktionen folgen grundsätzlich den Original-Funktionsnamen von Gelernter folgend, wenn auch andere Autoren in der Namensgebung abweichen (vgl. [Leler, 1990]).

Jeder Befehl den der Client an den Server sendet, wird atomar ausgeführt, d.h., er wird entweder vollständig oder gar nicht ausgeführt. Der Programmierer muß dies in seinen Programmen berücksichtigen. In anderen Linda-Implementationen [Bakken, 1993] wird auch eine atomare Ausführung von Befehlsgruppen unterstützt. Wie dies in Perl-Linda erreicht werden kann, wird in Abschnitt 6.5.2 noch genauer erläutert.

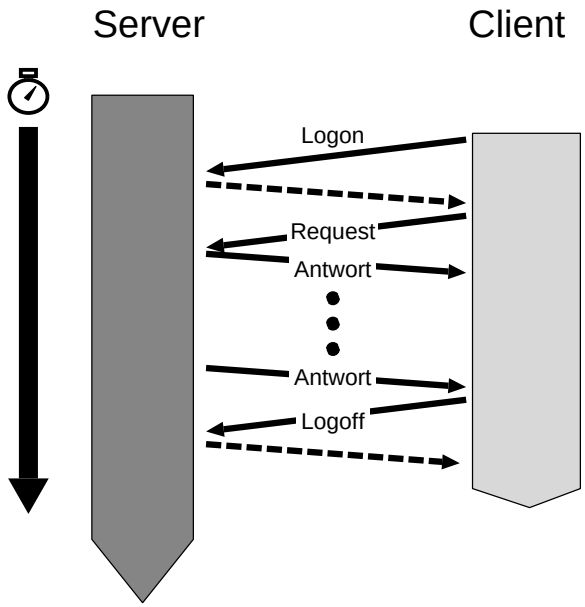

Abbildung 9: Schema einer Linda-Perl Session

Der Dialog zwischen Client und Server ist verbindungsorientiert. Der Ablauf einer typischen Session ist in Abbildung 9 dargestellt. Der Client baut eine Verbindung zum entsprechenden Port des Rechners auf, auf dem der Server läuft. Ist die Verbindung zum Server aufgebaut und akzeptiert der Server den Client, wird dieser Verbindung ein exklusiver Port zugewiesen, über den die weitere Kommunikation stattfindet. Danach kann der Client mittels der Linda-Funktionen auf die Daten im Tuplespace zugreifen. Nach Ausführung aller Kommandos wird die Verbindung zum Server beendet. Da ein schließen der Verbindung seitens des Client auf die Stabilität des Perl-Linda-Servers negative Auswirkungen haben kann, teilt der Client dem Server mit, daß er seine Interaktion beendet hat. Der Server schließt dann die Verbindung zum Client.

5.2 Tuple und Pattern

In diesem Abschnitt wird die interne Darstellung der in Linda verwendeten Datenstrukturen *Tuple*, *Pattern* und *Tuplelist* – eine Liste von Tuple – beschrieben. Die Datenstrukturen werden in dieser Form im Perl-Linda-Server gespeichert und auch bei der Kommunikation mit dem Server verwendet. Da die Darstellung unabhängig von der jeweiligen Plattform und Programmiersprache sein soll, muß die interne Darstellung der Daten in der jeweiligen Programmiersprache in dieses Format umgewandelt werden. Um ein vollständiges 256-Zeichen Alphabet zur Verfügung zu haben, werden die Steuerzeichen mittels eines Escape-Zeichens speziell vercodiert. Die folgende Darstellung der Datenstrukturen erfolgt in der Backus-Naur-Form (BNF).

Tuple und Pattern dienen als Argumente für die Linda-Befehle des Clients an den Server. Während in der Datenstruktur Tuple ein Datenelement an den Server gesandt wird, wird ein Pattern dazu verwendet, Daten vom Server abzurufen. Tabelle 4 stellt die BNF für Tuple und Tuplelist dar.

Es stehen die Zeichen des ASCII-Zeichensatzes ohne die Zeichen `{.}` `{,}` `{:}` `{?}` `{\n}` `{\0}` zur Verfügung (die geschwungenen Klammern dienen nur zur Hervorhebung des darzustellenden Zeichens), die in der internen Darstellung Steuerzeichen sind. Grundsätzlich werden alle Datenstrukturen als Zeichenketten gespeichert. Die Elemente im Tuple sowie im Pattern werden durch das Trennzeichen `{,}` getrennt. Zur Kennzeichnung formaler Elemente wird das Zeichen `{?}` verwendet. Tuple können somit folgende Gestalt haben (die Klammern `( )` dienen zur Kennzeichnung der äußeren Begrenzung der Datenstrukturen):

```
(Das,ist,ein,Tuple)
(x,2)
(Ein Tuple kann auch Leerzeichen haben)
```

```
          <char>  :=  ASCII Zeichen ohne {.,:?\n\0};
     <delimiter>  :=  ",";
      <query_op>  :=  "?";
      <wildcard>  :=  "*";
  <encoded_char>  :=  ":e" | ":n" | ":d"
                  |   ":q" | ":p" | ":0";
       <element>  :=  <element> <char>
                  |   <char>
                  |   <encoded_char>;
      <var_name>  :=  <var_name> <char>
                  |   <char>
                  |   <encoded_char>;
        <formal>  :=  <query_op>
                  |   <element> <query_op>
                  |   <element> <query_op> <var_name>
                  |   <query_op> <var_name>
                  |   <element> <query_op> <wildcard>
                  |   <query_op> <wildcard>;
         <tuple>  :=  <tuple> <delimiter> <tuple>
                  |   <element>
       <pattern>  :=  <pattern> <delimiter> <pattern>
                  |   <element>
                  |   <formal>;
```

Tabelle 4: BNF für Tuple und Pattern

Pattern können im Gegensatz zu Tuple auch formale Elemente enthalten. Diese werden in Linda-Operationen dann durch aktuale Element der im Tuplespace gespeicherten Tuple ersetzt. Bei Pattern wird als Zeichen {?} zur Darstellung verwendet. Pattern werden daher folgendermaßen in Perl-Linda dargestellt:

```
(myvar,?)
(x,?,y,?)
(Ein konstanter Pattern)
```

Das letzte Beispiel zeigt, daß ein Pattern zwar formale Elemente haben kann, aber nicht unbedingt haben muß. Auf einen konstanten Pattern paßt jedoch nur ein Tuple der gleichen Form. Kommt eines der Sonderzeichen in einem aktualen Element des Tuple oder Pattern vor, so wird es mittels des Escape-Zeichens `{:}` vercodiert. Dies verhindert, daß es bei der Bearbeitung der Tuple als Steuerzeichen interpretiert wird. Tabelle 5 zeigt alle verwendeten Steuerzeichen, ihre Bedeutung und ihre Vercodierung mittels des Escape-Zeichen.

Zeichen im Client	**Zeichen im Server**	**Aufgabe im Protokoll**
`","`	`":d"`	Trennzeichen der Elemente im Tuple.
`"\n"`	`":n"`	`End of Line` markiert das Ende des Kommandos bei der Kommunikationsrichtung Client → Server. In der Kommunikationsrichtung Server → Client steht `'\n'` als Trennzeichen für die einzelnen Tuple der Tuplelist.
`"."`	`":p"`	`End of Server Output` markiert das Ende der Antwort des Servers an den Client.
`":"`	`":e"`	`Escape` wird als Escape-Zeichen für alle anderen Vercodierungen verwendet.
`"??"`	`":q"`	Da das Zeichen `'?'` als Markierung für ein formales Element im Pattern einer Linda-Operation dient, muß eine Möglichkeit geschaffen werden, das Zeichen `'?'` als Textbestandteil eines Elements zu senden. Es wird daher vorgeschlagen die Zeichenfolge `'??'` zu senden, wenn ein einzelnes Zeichen `'?'` gewünscht wird.
`NULL-Pointer`	`":0"`	In Programmiersprachen wie C und C^{++} kann es notwendig sein, einen Null-Pointer, welcher ein leeres Element kennzeichnen soll, als Element eines Tuple darzustellen. Da ein Tuple aber definitionsgemäß keine leeren Elemente haben darf, wird diese Möglichkeit der internen Darstellung vorgesehen.

Tabelle 5: Vercodierung der Steuerzeichen

Das Escape-Zeichen `{:}` wird seinerseits in einem ersten Arbeitsschritt zur Zeichenfolge `{:e}` vercodiert. Nach der Vercodierung des Escape-Zeichens werden alle anderen Zeichen ebenfalls vercodiert. Beim Decodieren erfolgt alles in der umgekehrten Reihenfolge. Eine Abweichung von diesem System ergibt sich bei der Verwendung des Zeichens `{?}` als Bestandteil eines Elements. Da ein einmaliges Auftreten dieses Zeichens das Element als formales Element kennzeichnen würde, muß, um ein Fragezeichen im Text vorkommen zu lassen, dies bereits bei der Eingabe doppelt vorhanden sein (also `{??}`). Dieses wird dann zur Folge `{:q}` vercodiert. Tabelle 6 zeigt einige Beispiele von Tuple und Pattern vor und nach der Vercodierung.

Vor der Vercodierung	Nach der Vercodierung
("a","b","c")	(a,b,c)
("pi","3.1459")	(pi,3:p1359)
("Name:","Werner,Schoenf")	(Name:e,Werner:dSchoenf)
("Name:","?")	(Name:e,?)
("Wer??","?")	(Wer:q,?)
("Zeilen-\numbruch")	(Zeilen-:nUmbruch)

Tabelle 6: Beispiele für die Vercodierung

Die Datenstruktur Tuplelist wird verwendet, um die Antworten des Servers an den Client zu senden. Je nach gegebenem Befehl kann die Tuplelist mindestens 0 oder 1 Tuple enthalten (siehe Tabelle 10). Da die Tuplelist nur in den Antworten des Servers verwendet wird, enthält sie nur Tuple und niemals Pattern. Das serverseitige Speicherformat des Tuplespace entspricht auch dem Tuplelist-Format. In Tabelle 4 befindet sich die formale Beschreibung von Tuple und Pattern in BNF.

```
<tuple_del>  :=  "\n";
<tuplelist>  :=  <tuplelist> <tuple_del> <tuple>
              |  <tuple>
              |  ϵ;
```

Tabelle 7: BNF für Tuplelist

In der Tuplelist sind die einzelnen Tuple durch das Zeilentrennzeichen {\n} getrennt. Bei der Rückgabe der Antwort auf den Linda-Befehl wird somit jeweils ein Tuple pro Zeile ausgegeben.

Das vorgestellte Format unterstützt die Darstellung der Datenstrukturen mit den in Perl vorhandenen Standarddatentypen. Es wird sowohl für die interne Darstellung im Server als auch für die Argumente und Rückgabewerte der Linda-Operationen verwendet. Die Vercodierung der Rohdaten erfolgt durch den Client vor der Übermittlung an den Server. Der Server überprüft die syntaktische Korrektheit der übermittelten Daten.

5.3 Perl-Linda-Befehle

Der Befehlssatz von Perl-Linda basiert auf dem Original-Linda-Befehlssatz. Die Befehle und deren Argumente werden vom Server auf syntaktische Richtigkeit geprüft und dann ausgeführt. Die Ergebnisse der Operationen werden an den Client zurückgeschickt. Die Befehle von Perl-Linda gliedern sich in zwei Gruppen:

Linda-Befehle:
Diese implementieren die Linda-Funktionen in Perl-Linda. Neben den von Gelernter vorgestellten Befehlen `out()`, `in()`, `rd()`, `inp()` und `rdp()` gibt es noch eine Anzahl von Befehlen, die die Funktionalität von Linda erweitern.

Administrative Befehle:
Diese Befehle dienen zur Administration des Perl-Linda-Servers durch den Systemadministrator und gliedern sich in Befehle zum Beenden des Servers, zum Auflisten des Status des Servers und zum Speichern und Löschen der Daten.

Tablle 8 zeigt die in Perl-Linda implementierten Original-Linda-Befehle. Die Namen der Original-Funktionen `inp()` und `rdp()` wurde in `nbin()` (non-blocking in) und `nbrd()` (non-blocking rd) geändert.

Befehl	Beschreibung
`out(`*tuple*`)`	`out` stellt das als Argument mitgegebene Tuple in den Tuplespace.
`in(`*pattern*`)`	`in` entfernt alle auf das Pattern passenden Tuple aus dem Tuplespace. Dieser Befehl blockiert so lange, bis ein passendes Tuple im Tuplespace vorhanden ist. Die gefundenen Tuple werden als *Tuplelist* zurückgegeben.
`rd(`*tuple*`)`	`read` liest alle auf das Pattern passenden Tuple aus dem Tuplespace, läßt sie jedoch dort. Wie `in` blockiert `read` so lange, bis zumindest ein passendes Tuple im Tuplespace gefunden wurde.
`nbin(`*pattern*`)`	`non-blocking in` entfernt alle auf das Pattern passenden Tuple aus dem Tuplespace. Werden keine passenden Tuple gefunden, gibt der Befehl eine leere Tuplelist zurück.
`nbrd(`*pattern*`)`	`non-blocking read` liest alle auf das Pattern passenden Tuple aus dem Tuplespace. Werden keine passenden Tuple gefunden, gibt der Befehl wie bei `nbin` eine leere Tuplelist zurück.

Tabelle 8: Original Linda-Kommandos

Die folgenden Beispiele in Pseudocode sollen die Funktionalität der Perl-Linda-Befehle erläutern. In Abschnitt 5.3 werden diese noch genauer erklärt.

- Abfrage, ob ein bestimmtes Tuple `(x,y,z)` im Tuplespace existent ist:
  `rd(x,y,z)`
  Der Prozeß blockiert, wenn `(x,y,z)` nicht existiert.
  `nbrd(x,y,z)`
  Der Prozeß blockiert **nicht**, wenn `(x,y,z)` nicht existiert.
- Entfernen des Tuple `(x,y,z)` aus dem Tuplespace: `in(x,y,z)`
  Der Prozeß blockiert, wenn `(x,y,z)` nicht existiert.
  `nbin(x,y,z)`
  Der Prozeß blockiert **nicht**, wenn `(x,y,z)` nicht existiert.
- Austtauschen des Tuple `(x,a)` durch das Tuple `(x,b)`:
  `in(x,a)`
  `out(x,b)`
  Der Prozeß blockiert so lange, bis das Tuple `(x,a)` im Tuplespace enthalten ist, dann erst wird das Tuple `(x,b)` in den Tuplespace gestellt. Es ist damit sichergestellt, daß sich `(x,a)` und `(x,b)` nicht gleichzeitig im Tuplespace befinden.
- Simulation eines Mutex (binärer Semaphor, vgl. dazu S. 33):
  `in(mutex1,unlocked)`
  `out(mutex1,locked)`
  ...
  Verwendung der durch den Mutex gesicherten Ressource.
  ...
  `in(mutex1,locked)`
  `out(mutex1,unlocked)`
  Diese Form der Implementation stellt sicher, daß jeder Client, der auf die gesicherte Ressource zugreifen will, wartet bis der Mutex sich wieder im Zustand `unlocked` befindet.

Die Funktionalität des Perl-Linda-Befehlssatzes wurde um die in Tabelle 9 erklärten Befehle `ain()`, `ard()`, `uin()` und `uout()` erweitert. Der Bedarf für die Erweiterung um die Befehle `ain()` und `ard()` ergab sich aus der Eigenschaft der Befehle `in()` und `rd()`, alle auf ein Pattern passenden Tuple aus dem Tuplespace zu entfernen. Zumeist ist es für eine Anwendung nur notwendig, **ein** passendes Tuple als Ergebnis des Kommandos zu erhalten. Die Befehle `uin()` und `uout()` wurden zur Implementation einer atomaren

`in()--out()`-Kombination hinzugefügt. Es ist dadurch einem Client möglich, **ein** Tuple ausfallsicher zu bearbeiten.

Befehl	Beschreibung
`ain(`*pattern*`)`	`atomic in` hat grundsätzlich die gleiche Funktionsweise wie `in`. Es gibt jedoch genau ein Tuple statt der von `in` zurückgegebenen Tuplelist zurück.
`ard(`*pattern*`)`	`atomic read` liest genau ein auf *pattern* passendes Tuple aus dem Tuplespace. Dieser Befehl ist nur bedingt einsetzbar, da bei wiederholter Anwendung immer dasselbe Tuple zurückgegeben wird, da es nie aus dem Tuplespace entfernt wird.
`uin(`*pattern*`)`	`update in` entfernt genau ein auf *pattern* passendes Tuple aus dem Tuplespace. Das entfernte Tuple wird jedoch im Server gespeichert, sodaß es im Falle eines Datenverlusts im Client wiederhergestellt werden kann. Dieser Befehl ist eine Komponente zur Implementation einer fault-toleranten Update-Funktion im Client-API. Diesem Befehl muß unmittelbar ein `uout` folgen, um die fault-tolerante Zwischenspeicherung des entfernten Tuple aufzuheben. Folgt ein anderer Befehl oder bricht der Client die Verbindung ab, wird das entfernte Tuple wieder in den Tuplespace gestellt.
`uout(`*tuple*`)`	`update out` funktioniert wie ein normaler `out`-Befehl und stellt die zweite Komponente zur Implementation einer update-Funktion dar. Durch die Verwendung von `uout` wird die Speicherung des entfernten Tuple im Server gelöscht.

Tabelle 9: Erweiterte Linda-Kommandos

Die Funktionalität der erweiterten Linda-Befehle ist zu den Original-Linda-Befehlen kompatibel. Die Auswirkungen der verschiedenen Befehle beeinflußen sich gegenseitig nicht. Die erweiterten Linda-Befehle sind in Tabelle 9 beschrieben. Der Einsatz der Befehle `uin()` und `uout()` wird in Abschnitt 5.4 beschrieben.

Das Ergebnis der Linda-Befehle wird in Form der Datenstruktur *Tuplelist* dargestellt. Eine Tuplelist ist eine Liste von Tuple. Die Grammatik für die Datenstruktur Tuplelist ist in Abbildung 7 dargestellt. Die Anzahl der Tuple kann im kleinsten Fall null Tuple betragen. In diesem Fall wird eine leere Tuplelist zurückgegeben. Tabelle 10 listet die Rückgabewerte der einzelnen Linda-Befehle auf.

Neben den Linda-Befehlen, welche direkt in der Client-Schnittstelle implementiert werden sollten, kennt der Server noch eine Anzahl von Befehlen, welche hier als administrative Befehle bezeichnet werden. Diese Befehle dienen als Hilfsbefehle für das Debuggen und Testen von Linda-Programmen und haben sich als nützlich bei der Administration des Servers herausgestellt. Mit den Befehlen

Befehl	Rückgabewert
`out`	leere Tuplelist
`uout`	leere Tuplelist
`in`, `rd`	Tuplelist, 1 oder mehrere Tuple
`nbin`, `nbrd`	Tuplelist, 0 oder mehrere Tuple
`ain`, `ard`	Tuplelist, 1 Tuple
`uin`	Tuplelist, 1 Tuple
`cleanup`	Tuplelist, 1 Tuple: `(reset)`
`bye`	`Connection closed by foreign host`
`lst, save, load, merge, kill`	Da die Befehle nicht für die Verwendung im Client-API bestimmt sind, folgen ihre Ausgaben keiner besonderen Grammatik.

Tabelle 10: Rückgabewerte der Linda-Befehle

`bye()` und `kill()` kann ein Schließen einer Client-Verbindung bzw. für einen gesamten Shutdown des Servers durchgeführt werden. Information über den Inhalt des Tuplespace listet der Befehl `lst()` auf. Mit `cleanup()` läßt sich der Inhalt des Tuplespace löschen. Die Befehle `save()`, `load()` und `merge()` erlauben ein Abspeichern und späteres Laden des Tuplespace-Inhalts.

Diese administrativen Befehle sollten von Clients jedoch nur mit Bedacht aufgerufen werden, da manche Befehle das Verhalten der eigenen Anwendung und eventuell anderer auf dem Server laufender Anwendungen drastisch beeinflussen können. Einzelne Befehle können vom Programmierer durch Auskommentieren dieser in der Serverhauptfunktion deaktiviert werden. Die administrativen Befehle des Perl-Linda-Servers sind in Tabelle 11 dargestellt.

5.4 Atomare Bearbeitung von Tuple

Es kann kein fault-toleranter Austausch eines Tuples im Tuplespace mittels einer `in()--out()`-Kombination erfolgen. Liest der Client das Tuple mit `in()` aus dem Tuplespace ein und versagt, so ist das Tuple verloren. Um dieses Versagen abzufangen, wurden die Befehle `uin` (update in) und `uout` (update out) dem Linda-Befehlssatz hinzugefügt. Die Funktion von `uin` ist in Abbildung 10 ersichtlich.

Befehl	Beschreibung
`bye()`	`bye()` ist das einzige der administrativen Kommandos, welches im Client-API implementiert werden sollte. Sendet der Client `bye()`, so zeigt er dem Server an, daß er die Verbindung zu beenden wünscht. Der Server bricht die Verbindung dann von sich aus ab. Alle Statusdaten im Zusammenhang mit dem Client werden gelöscht.
`kill()`	`kill` beendet den Server. Der Server beendet die Verbindung zu allen Clients und führt einen Shutdown durch. Die im Tuplespace enthaltenen Daten gehen dabei verloren.
`lst()`	`list` listet den augenblicklichen Inhalt des Tuplespace auf. Die Tuple werden in vercodierter Form gezeigt. Dies kann unter Umständen zu Problemen bei der Lesbarkeit führen.
`cleanup()`	`cleanup` führt einen Reset des Servers durch. Der Inhalt des Tuplespace wird gelöscht. Alle blockierenden Clients erhalten vom Server das Tuple `(reset)` gesandt, um die Blockierung zu beenden.
`save(`*filename*`)`	`save` speichert den Inhalt des Tuplespace in die angegebene Datei. Aus Sicherheitsgründen wird die Datei in das Verzeichnis `/usr/tmp` gespeichert. Wird kein Dateiname angegeben, wird `tuplespc.sav` als Dateiname verwendet. Die im Tuplespace enhaltenen Tuple werden vercodiert, ein Tuple pro Zeile gespeichert. Diese Art der Speicherung erleichtert die weitere Verarbeitung des Tuplespace-Inhalts, da dieses Format leicht in andere Formate wie z.B. MS-EXCEL konvertiert werden kann.
`load(`*filename*`)`	`load` lädt den Inhalt der angegebenen Datei in den Tuplespace. Der alte Inhalt des Tuplespace geht dabei verloren. Es wird wie bei `save` das Verzeichnis `/usr/tmp` zum Lesen verwendet. Obwohl eine syntaktische Überprüfung beim Laden stattfindet, sollten nur durch `save` gespeicherte Inhalte geladen werden.
`merge(`*filename*`)`	`merge` hat grundsätzlich die gleiche Funktionalität wie `load`. Im Unterschied zu `load` wird der Tuplespace vor dem Laden nicht gelöscht. Die in der Datei gespeicherten Tuple werden somit dem Inhalt des Tuplespace hinzugefügt.

Tabelle 11: Administrative Befehle des Perl-Linda-Server

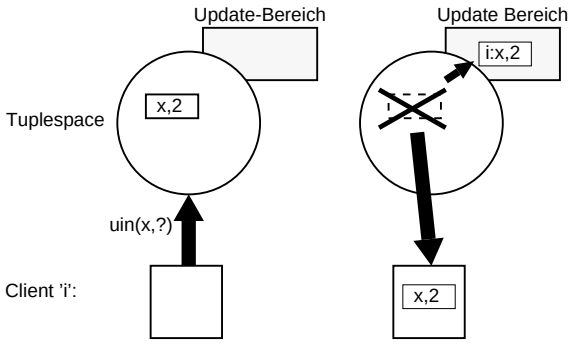

Abbildung 10: Funktion von `uin`

Durch den `uin()`-Befehl wird jeweils nur ein passendes Tuple aus dem Tuplespace entnommen. Sendet ein Client diesen Befehl, entfernt der Server das Tuple aus dem Tuplespace, speichert es aber zusammen mit der Identifzierung des aufrufenden Client. Der Client muß nun als nächsten Befehl `uout()` senden. Die Funktion von `uout()` ist in Abbildung 11 dargestellt.

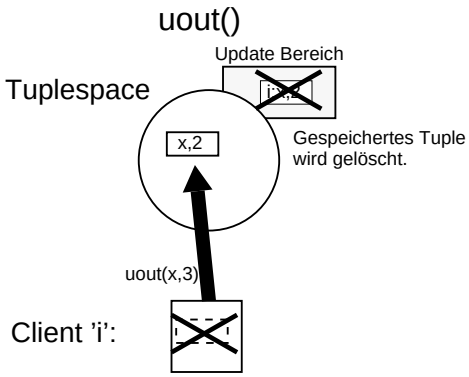

Abbildung 11: Funktion von `uout()`

Wenn der Client nach `uin()` den Befehl `uout()` mit dem ersetzenden Tuple sendet, wird das Tuple in den Tuplespace gestellt und die gespeicherte Version des entnommenen Tuple gelöscht. Diese Funktionalität stellt den normalen Einsatz der `uin()`–`uout()` Befehlskombination dar.

Sendet der Client einen anderen Befehl, oder bricht die Verbindung zum Client ab, stellt der Server das gespeicherte Tuple wieder in den Tuplespace zurück (siehe Abbildung 12). Dies geschieht auch, wenn der Client die Verbindung durch

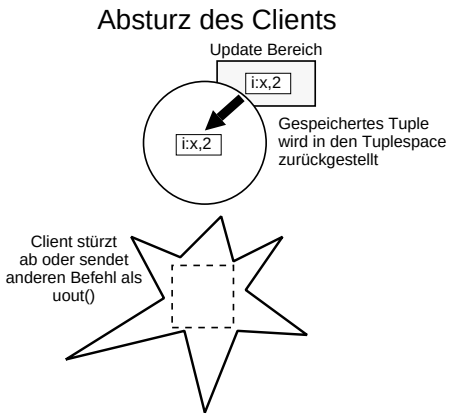

Abbildung 12: Verhalten bei Absturz des Clients

`bye()` beendet. Durch dieses Verhalten kann das Tuple der Applikation erhalten bleiben, obwohl es durch ein Fehlverhalten oder durch den Absturz des Clients verloren geht.

6 Der Perl-Linda-Server

Dieser Abschnitt beschreibt die Implementation des Perl-Linda-Servers. Es werden Grundkenntnisse in der Programmiersprache Perl4.036 vorausgesetzt. Der Sourcecode zum Server befindet sich in Anhang A. Nach einem Überblick über die Funktionsweise und die Hauptkomponenten des Perl-Linda-Servers wird die Implementation der Linda-Funktionen und das Zusammenwirken mit den Hauptkomponenten anhand von Pseudocode erklärt. Im Anschluß daran wird die Implementation des Matching von Pattern in Perl-Linda dargestellt. Mögliche Konzepte zur Erweiterung der atomaren Operationen, die mehrere Befehle umfassen, werden danach erläutert. Abschließend werden Restriktionen beim Einsatz des Server erläutert und Installation und Betrieb beschrieben.

6.1 Funktionsweise

Der Perl-Linda-Server basiert auf der Serverimplementierung `easy-ipc.pl`[4], das die grundsätzliche Funktionalität bezüglich der Sockets zur Verfügung stellt. Das Programmpaket unterstützt in seiner Originalversion die Implementation eines Servicedienstes an einem benutzerdefinierten Port eines Unix-Rechners. Da das ursprüngliche Kommunikationsprotokoll zwischen dem Server und dem ebenfalls enthaltenen Client nicht für die Zwecke von Linda geeignet war, blieben vom urspünglichen Paket nur mehr die Teile bezüglich der Socket-Verwaltung erhalten.

Die Hauptteile des Perl-Linda-Servers sind in Abbildung 13 ersichtlich. Der Server besteht aus drei Komponenten, der Server-Funktion, den Linda-Funktionen inklusive der administrativen Funktionen und den Datenstrukturen Tuplespace und *Waiting Queue*.

Abbildung 14 zeigt das Zustandsdiagramm des Perl-Linda-Servers. Zur Vereinfachung ist das Zustandsdiagramm nur für die Funktionen `in()`, `rd()`, `out()` und `kill()` dargestellt. Die Server-Funktion wartet am benutzerdefinierten Port auf neu ankommende Verbindungen und nimmt diese an. Sie überprüft bereits bestehende Verbindungen und nimmt den Input der Clients entgegen. Die Requests der Clients werden einer syntaktischen Überprüfung unterzogen. Danach wird an-

[4] EASY-IPC.PL wurde 1990 von Hiroshi Sakoh (sakoh@sra.co.jp) (© by Software Research Associates) programmiert. Das Programmpaket wurde 1995 vom Autor dieser Arbeit modifiziert und teilweise neu geschrieben.

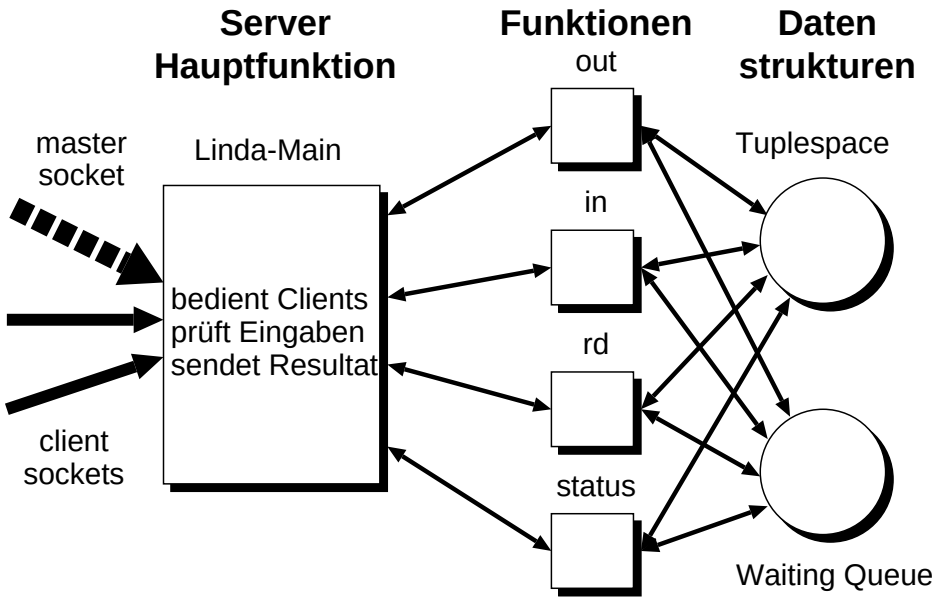

Abbildung 13: Hauptkomponenten des Perl-Linda-Servers

hand des Requests entschieden, was zu tun ist:

in():

Beim Aufruf der Funktion `in()` wird anhand des als Parameter mitgegebenen Pattern der Tuplespace nach passenden Tuple durchsucht. Gefundene Tuple werden aus dem Tuplespace extrahiert und an den Client zurückgesandt. Werden keine passenden Tuple gefunden wird der Request mit den Daten des Client in der Waiting Queue gespeichert.

rd():

Die Funktion `rd()` funktioniert analog zur Funktion `in()` mit dem Unterschied, daß die gefunden Tuple beim Suchvorgang nicht aus dem Tuplespace entfernt werden.

out():

Bei einem `out()`-Request muß überprüft werden, ob das mitgegebene Tuple einen der in der Waiting Queue wartenden Requests erfüllt, bevor es in den Tuplespace gestellt werden kann. Paßt das Tuple auf einen der wartenden Requests, wird das Tuple an den jeweiligen Client gesandt und der Request aus der Waiting Queue entfernt. Solange der wartende Request ein `rd()`-Request war, kann die Waiting Queue weiter geprüft werden. War

es ein `in()`-Request ist das Tuple nicht mehr verfügbar und der `out()`-Request ist beendet. Ist das Ende der Waiting Queue erreicht und das Tuple noch immer verfügbar, dann wird es in den Tuplespace gestellt.

kill():

Die Funktion `kill()` beendet den Server. Die Verbindungen zu allen Clients werden abgebrochen und der Serverprozeß beendet.

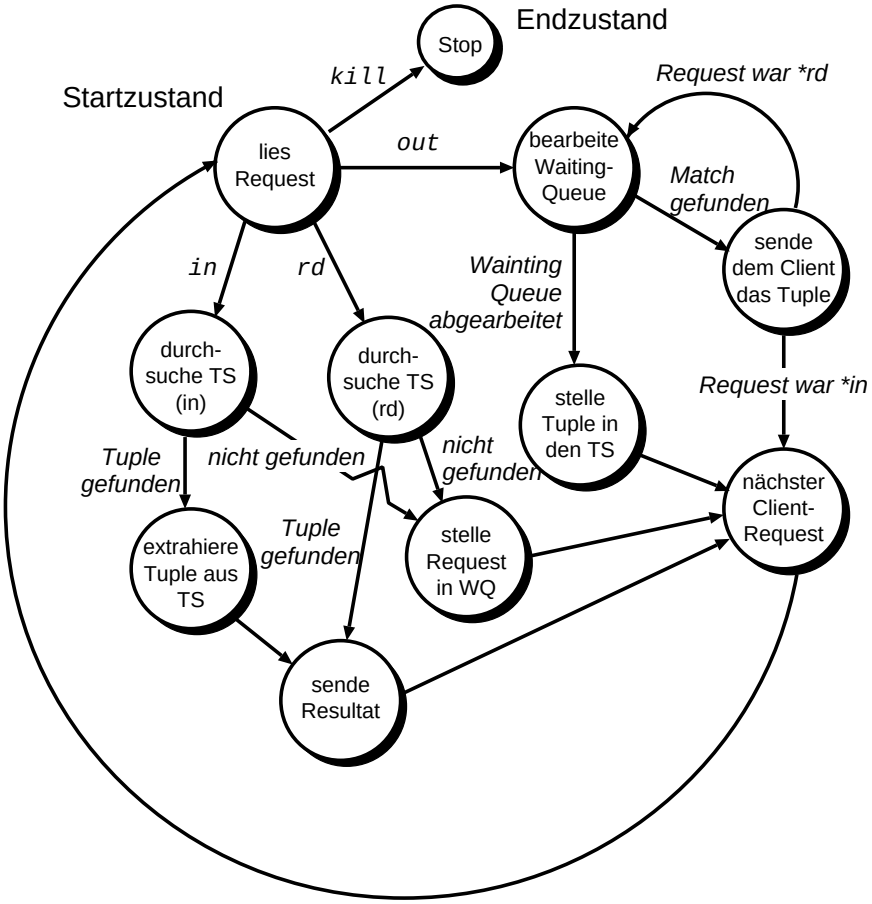

Abbildung 14: State-Diagramm bezüglich der Linda-Befehle des Perl-Linda-Server

Wird eine Verbindung vom Client abgebrochen, übernimmt die Server-Funktion auch die Löschung aller mit dem Client in Verbindung stehenden Daten. Während der Verarbeitung des Requests in der entsprechenden Linda-Funktion wartet die Server-Funktion auf das Resultat der aufgerufenen Funktion und schickt es an den Client zurück.

Die Datenstrukturen haben die Aufgabe, die Inhalte des Tuplespace und die damit verbundenen Daten zu speichern. Es wurden sowohl für Tuplespace und Waiting Queue Perl-Listenstrukturen verwendet, da diese einfach und schnell zu verarbeiten sind. Jedes Tuple wird als Element der Liste dargestellt. Neu hinzukommende Tuple werden an das Ende der Liste gehängt. Somit sind in dieser Version mehrere auf ein Pattern passende Tuple nach der Reihenfolge ihres Eintreffens sortiert. Dies ist in Linda nicht explizit vorgesehen. Applikationsprogramme sollten daher diese Eigenschaft nicht für ihr Funktionieren voraussetzen.

In der Waiting Queue werden alle blockierenden Client-Requests gespeichert. Ein Request wird als ein Listenelement der Waiting Queue gespeichert. Neben der Identifikation des Clients werden auch noch der gegebene Befehl und das Pattern im Regular Expression Format (siehe Tabelle 15) gespeichert. Da der Server bei jedem eintreffenden `out`-Befehl eines Clients prüfen muß, ob das in den Tuplespace zu stellende Tuple auf ein Element der Waiting Queue paßt, muß das Pattern nicht noch einmal in das Regular Expression Format übersetzt werden, sondern kann gleich geprüft werden.

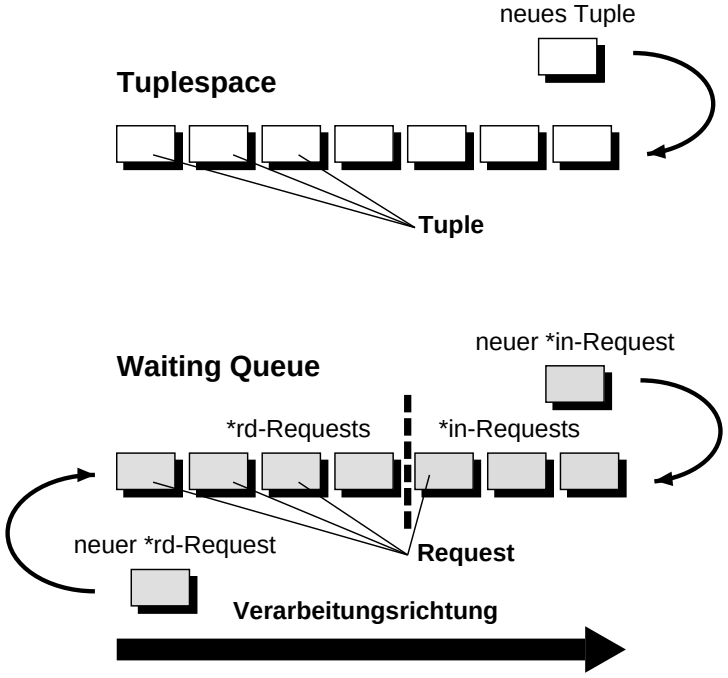

Abbildung 15: Tuplespace und Waiting Queue

Während im Tuplespace die Elemente unsortiert nach der Reihenfolge des Eintreffens gespeichert werden, sind die Elemente in der Waiting Queue nach den zu-

grundeliegenden Befehlen sortiert (siehe Abbildung 15). Zuerst werden alle von den Clients aufgerufenen und blockierenden `rd()`- und `ard()`-Befehle gespeichert. Darauf folgen alle blockierenden `in()`-, `ain()`- und `uin()`-Befehle. Die letzere Gruppe ist zeitlich nach dem Eintreffen der Requests sortiert. Die erstere Gruppe ist nicht sortiert. Diese Struktur der Waiting Queue ist notwendig, da alle in der Waiting Queue enthaltenen `rd()`- und `ard()`-Befehle, die den Inhalt des Tuplespace ja nicht verändern, vor den anderen Befehlen bearbeitet werden müssen. Würde es umgekehrt geschehen, wäre das Tuple vielleicht schon durch einen `in()`-, `ain()`- oder `uin()`-Befehl aus dem Tuplespace extrahiert, bevor die blockierenden `rd()`- oder `ard()`-Befehle geprüft werden konnten. Um die Struktur konsistent zu halten, werden alle neu ankommenden und blockierenden `in()`- `ain()`- und `uin()`-Befehle an das Ende der Liste, die `rd()`- und `ard()`-Befehle an den Anfang der Liste angehängt.

Die oben dargestellte Ordnung der blockienden Requests im Linda-Konzept bringt einen Nebeneffekt mit sich. Angenommen, *Client1* setzt den Befehl `in(x,y)` zum Entfernen des Tuple `(x,y)` aus dem Tuplespace ab. Da sich das Tuple `(x,y)` im Zeitpunkt des Absetzens des Befehls nicht im Tuplespace befindet, blockiert der Befehl. Kurz danach setzt *Client2* den Befehl `rd(x,y)` ab, der ebenfalls blockiert. Wenn das Tuple `(x,y)` von einem anderen Prozeß in den Tuplespace gesandt wird, so erhalten durch die spezielle Abarbeitung in der Waiting Queue beide Clients das Tuple. Durch den `in()`-Request kommt es aber nicht in den Tuplespace. Bei sequentieller Abarbeitung der Requests in der Reihenfolge ihres Eintreffens, wie im normalen Linda-Konzept vorgesehen, würde der `in()`-Request zuerst ausgeführt und das Tuple sofort aus der Waiting Queue entfernt. Der `rd()`-Request würde weiter blockieren.

Im normalen Linda-Konzept ist diese Funktionalität nicht vorgesehen. Bei den bisher erstellten Applikationen entstanden dadurch keine Probleme. Es ist aber doch anzuraten, diese Funktionalität nicht bei der Entwicklung von Linda-Applikationen zu verwenden.

6.2 Linda-Funktionen

Die Funktionen stehen als Bindeglied zwischen der Server-Funktion und den oben beschrieben Datenstrukturen. Sie teilen sich in Linda-Funktionen und administrative Funktionen. Die Linda-Funktionen teilen sich wiederum in Original-Linda-Funktionen und erweiterte Linda-Funktionen. Die Linda-Funktionen nehmen Tu-

ple oder Pattern als Eingabe (Argument) und verarbeiten diese. Dabei greifen sie auf Tuplespace und Waiting Queue zu. Wenn die Operation aus den Daten nicht erfüllt werden kann, d.h., wenn die Operation blockiert, wird der gesamte Request in die Waiting Queue eingefügt und es wird ein entsprechender Rückgabewert zurückgegeben, um der Server-Funktion dies anzuzeigen. Ist dies nicht der Fall, wird das Ergebnis (eine Tuplelist) an die Server-Funktion zurückgegeben. Die Server-Funktion schickt das Ergebnis dann an den Client zurück.

Die einzelnen Linda-Funktionen sind im folgenden in Pseudocode dargestellt, um die Funktionsweise und die Zusammenarbeit mit den anderen Bestandteilen des Servers zu erläutern. Der Pseudocode ist nicht direkt äquivalent mit der tatsächlichen Funktion in Perl, da aus Performanceüberlegungen einige Dinge weniger verkapselt programmiert wurden. Die genaue Übertragung des Sourcecode in Pseudocode würde sich jedoch negativ auf das Verständnis und die Lesbarkeit auswirken.

rd(*ClientConnection c, Pattern p*):

```
TupleList found;
found=searchTupleSpace(p);

if(found is empty)
{
   addToWaitingQueue(c,p);
   return 0;
}
else
   return found;
```

Die Funktionen `rd()` und `nbrd()` sind die einfachsten Funktionen, da in ihnen nur eine Suche nach passenden Tuple durchgeführt wird, ohne den Inhalt des Tuplespace zu verändern. Wird kein passendes Tuple gefunden, wird der aufrufende Client und der Suchpattern der Waiting Queue hinzugefügt. Es werden in der Waiting Queue mit dem Request der aufrufende Client und der gegebene Befehl gespeichert. Zusätzlich wird der bereits in das Perl Regular Expression Format umgewandelte Pattern gespeichert.

Wurde zumindest ein passendes Tuple gefunden, wird dieses an die Server-Funktion zurückgegeben. Wird an das Hauptprogramm der Wert `0` zurückgegeben, weiß die Server-Funktion, daß die Suche ergebnislos verlaufen ist.

nbrd(*ClientConnection c, Pattern p*):

```
TupleList found;
```

```
found=searchTupleSpace(p);

if(found is empty)
   found=TupleList("");

return found;
```

Anders als bei `rd()` wird bei dieser Funktion eine leere Tuplelist an die Hauptfunktion zurückgegeben, wenn kein passendes Tuple gefunden wurde. Eine leere Tuplelist unterscheidet sich vom Return-Wert `0`.

ard(*ClientConnection c, Pattern p*):

```
TupleList found;

found=searchTupleSpace(p);

if(found is empty)
{
   addToWaitingQueue(c,p);
   return 0;
}
else
   return found[1];
```

Bei `ard()` wird nur das erste gefundene Tuple zurückgegeben. Da die im Tuplespace enthaltenen Tuple nach der Reihenfolge des Eintreffens sortiert sind, bekommt man das älteste passende Tuple. Auf dieses Feature sollte man sich bei der Programmierung des Clients allerdings nicht verlassen. Wenn kein passendes Tuple gefunden wird, wird der Client und der Pattern ebenfalls der Waiting Queue hinzugefügt.

in(*ClientConnection c, Pattern p*):

```
TupleList found;

found=searchTupleSpace(p);
if(found is empty)
{
   addToWaitingQueue(c,p);
   return 0;
}
else
{
```

```
   extractTuples(found);
   return found;
}
```

`in()` hat grundsätzlich die gleiche Funktionalität wie `rd()`, mit dem Unterschied, daß die passenden Tuple aus dem Tuplespace entfernt werden. Da `in()` bei Fehlen passender Tuple blockiert, wird auch hier der Client und das Pattern an die Waiting Queue – allerdings an das Ende – hinzugefügt. Das Unterprogramm `extractTuples()` extrahiert die gefundenen Tuple anhand ihrer Indizes aus dem Tuplespace.

nbin(*ClientConnection c, Pattern p*):

```
TupleList found;

found=searchTupleSpace(p);

if(found is empty)
   found=TupleList("");
else
   extractTuples(found);

return found;
```

`nbin()` ist die nicht blockierende Version des Befehls in `in()`. Hier wird wie bei `nbrd()` eine leere Tuplelist zurückgegeben, wenn kein passendes Tuple gefunden wird.

ain(*ClientConnection c, Pattern p*):

```
TupleList found;

found=searchTupleSpace(p);

if(found is empty)
{
   addToWaitingQueue(c,p);
   return 0;
}
else
{
   extractTuples(found[1]);
   return found[1];
}
```

Im Unterschied zur Funktion `in()` gibt `ain()` jeweils nur das erste gefundene Tuple zurück. Es gelten die gleichen Rahmenbedingungen über die Anordung der Tuple im Tuplespace wie bei `ard()`.

out(*ClientConnection c,tuple t*):

```
int avail=1;
ClientConnection r;
Pattern p;

for((r,p) in WaitingQueue)
{
  if(matches(t,p))
  {
    if(r.function='in'
       || r.function='uin'
       || r.function='ain')
       avail=0;
    sendClient(r,t);
    deleteFormWQ(r);
  }
  if(avail == 0) return;
}
addToTuplespace(t);
```

Die Funktion `out()` stellt ein Tuple in den Tuplespace. Bevor dies jedoch geschehen kann, muß überprüft werden, ob das Tuple auf einen der in der Waiting Queue wartenden, blockierenden Requests paßt. Aus der Waiting Queue werden der gespeicherte Request und das dazugehörige Pattern extrahiert. Es wird überprüft, ob das Tuple auf das Pattern paßt. Paßt das Tuple, wird es direkt dem entsprechenden Client zugesandt. Der Request und das dazugehörige Pattern wird aus der Waiting Queue gelöscht. Die Variable `avail` zeigt in diesem Zusammenhang an, ob das Tuple noch für weitere Überprüfungen vorhanden ist. Ist das Tuple nach Abarbeitung der Waiting Queue noch immer vorhanden, kann es dem Tuplespace hinzugefügt werden. Der Rückgabe-Wert der `out()`-Funktion wird weder abgefragt noch verwendet.

uin(*ClientConnection c, Pattern p*):

```
TupleList found;

found=searchTupleSpace(p);

if(found is empty)
```

```
{
   addToWaitingQueue(c);
   return found;
}
else
{
   extractTuples(found[1]);
   preserveTuple(c,found[1]);
   return found[1];
}
```

`uin()` hat die gleichen Aufbau wie `ain()` mit dem Unterschied, daß das gefundene Tuple zusammen mit dem Client in einer speziellen Liste gespeichert wird. Folgt dem `uin()`-Befehl ein anderer Befehl als `uout()`, dann wird das gespeicherte Tuple in den Tuplespace zurückgestellt.

uout(*ClientConnection c, tuple t*):

```
int avail=1;
ClientConnection r;
Pattern p;

if(c.lastCmd== 'uin')
   clearPreservedTuple(c);

for((r,p) in WaitingQueue)
{
  if(matches(t,p))
  {
    if(r.function='in'
       || r.function='uin'
       || r.function='ain')
       avail=0;
    sendClient(r,t);
  }
  if(!avail) return;
}
addToTuplespace(t);
```

`uout()` hat die gleiche Funktionalität wie `out()`. Wurde als letztes Kommando `uin()` aufgerufen, so wird das mit der Funktion `preserveTuple()` gespeicherte Tuple wieder gelöscht.

serverLoop():

```
ClientConnection c;
```

```
Pattern          pat;
TupleList        return;
int              shutdown=0;

while(!shutdown)
{
  removeDeadClients();
  handleNewClients();
  for(c in clientConnections)
  {
     if(c and c.newRequest())
     {
        result=0;
        if(c.lastCmd == 'uin'
           and (not (c.function == 'uout')))
        {
           out(c, getPreservedTuple(c));
           clearPreservedTuple(c);
        }

        if(c.function=='kill')
           shutdown=1;
        else
           if(c.function=='uout' or c.function=='out')
              process(c, c.function, c.argument);
           else
           {
              pat=convertRegExp(c.argument);
              result = process(c, c.function, pat);
           }

        sendClient(c,result);
     }
  }
}

closeAllClients();
```

Die Server-Funktion ist verantwortlich für das Überprüfen der Client-Verbindungen und das Aufrufen der entsprechenden Server-Funktionen. Die eigentliche Server-Funktion ist eine Endlosschleife, die nur durch das Kommando `kill()` beendet werden kann.

In der Schleife werden die nicht mehr aufrechten Clientverbindungen gelöst

und alle Informationen bezüglich der Clients gelöscht. Danach wird festgestellt, ob neue Clients inzwischen eine Verbindung aufgebaut haben.

In der inneren Schleife wird für jeden Client überprüft, ob er einen neuen Request abgesetzt hat. Der Request wird auch syntaktisch überprüft. Wurde als letztes Kommando `uin()` abgesetzt, wird gleich an dieser Stelle überprüft, ob ein anderes Kommando als `uout()` gesendet wurde. Ist dies der Fall, wird das durch die Funktion `preserveTuple()` zwischengespeicherte Tuple mittels eines normalen `out()`-Befehls in den Tuplespace zurückgestellt. Das `out()`-Kommando wird verwendet, um sicherzustellen, daß auch die Waiting Queue hinsichtlich des zwischengespeicherten Tuple bearbeitet wird.

Ruft der Request die Funktionen `out()` und `uout()` auf, kann das in `c.argument` gespeicherte Tuple direkt an die Funktion übergeben werden. Bei allen anderen Funktionen wird als Argument ein Pattern übergeben, der erst in das Regular-Expression-Format von Perl umgewandelt werden muß. Dies geschieht an dieser Stelle, da auch in der Waiting Queue die Pattern zur Zeitersparnis nur mehr in diesem Format gespeichert werden.

Das von den Funktionen zurückgegebene Ergebnis wird in der Variablen `result` gespeichert. Diese wird am Ende der Verarbeitung des Requests an den Client zurückgesandt.

6.3 Grammatik der Client/Server Kommunikation

Die folgenden Beschreibungen der Grammatik von Perl-Linda treffen sowohl auf das Protokoll zwischen Client und Server als auch auf die interne Darstellung im Server und in den Sicherungsfiles des Servers zu.

In Tabelle 12 ist die Syntax der Linda-Befehle dargestellt. Dieses Format wird vom Server überprüft. Stellt der Server fest, daß das Format syntaktisch nicht korrekt ist, so wird der Befehl nicht ausgeführt. Die Befehle an den Server können entweder durch einen Client oder durch den Benutzer mittels einer `telnet`-Verbindung eingegeben werden.

In der Grammatik sind nur die Linda-Befehle dargestellt. Die administrativen Befehle werden im Format <*cmd_name*>`()`<*tuple_del*> an den Server geschickt. Der Client beendet die Verbindung mit dem Befehl `bye()`.

Beim Senden eines Befehls vom Client an den Server ist die zeilenmäßige Länge

```
<ocmd>  :=  "out" | "uout";
<icmd>  :=  "in" | "ain" | "nbin"
         |  "rd" | "ard" | "nbrd";
<lcmd>  :=  <ocmd> "(" <tuple> ")" <tuple_del>
         |  <icmd> "(" <pattern> ")" <tuple_del>;
```

Tabelle 12: BNF der Perl-Linda Befehle

des Befehls mit genau einer Zeile begrenzt. Bei der Antwort des Servers kann dies nicht bestimmt werden, da der Client keine Information dazu hat, wieviele Tuple die Tuplelist, mit welcher der Server antwortet, enthält. Es wird daher ein ausgezeichnetes Symbol benötigt, um das Ende der Übertragung anzuzeigen. In Analogie zum SMTP-Protokoll [Postel, 1982] wurde daher die Zeichenfolge `".\n"` am Anfang einer Zeile als Markierung des Endes der Übertragung gewählt. Tabelle 13 stellt die Grammatik der Server-Anwort dar.

```
<eo_rep>  :=  ".\n";
 <reply>  :=  <tuplelist> <eo_rep>;
```

Tabelle 13: BNF der Server-Antwort

6.4 Matching

In diesem Abschnitt wird das Matching von Tuple auf ein Pattern erklärt. Nach Erläuterung des normalen Matching für formale Elemente werden die erweiterten Möglichkeiten, die sich durch die Verwendung von Regular-Expressions für das Matching ergeben, beschrieben.

6.4.1 Implementation des normalen Tuple-Matching

Bei der Ausführung der Funktionen `in()`, `nbin()`, `ain()`, `uin()`, `rd()`, `nbrd()` und `ard()` wird versucht, für die im Pattern enthaltenen aktualen und formalen Elemente, Tuple mit einer passenden Struktur zu finden. Da ein formales Element durch jedes beliebige Element ersetzt werden kann, wird versucht, ein Tuple zu finden, welches auf die aktualen Elemente und die Gesamtzahl der Elemente im Pattern paßt. Tabelle 14 illustriert diesen Vorgang.

Inhalt des Tuplespace
(x,b,3,z)
(a,b,c)
(x,b,2)
(anfang)
(a,c,4)
(x,b,3,y)

Pattern	Passende Tuple
`(?)`	(anfang)
`(a,b,?)`	(a,b,c)
`(?,b,?)`	(a,b,c), (x,b,2)
`(a,c,4)`	(a,c,4)
`(x,?,3,?)`	(x,b,3,z), (x,b,3,y)
`(?,?)`	Funktion blockiert den Client bis 2-Tuple in Tuplespace gestellt wird.

Tabelle 14: Matching von Tuple durch Pattern

Das Beispiel zeigt die Möglichkeiten, welche Pattern zur Extraktion oder zum Lesen von Tuple aus dem Tuplespace bieten. Insbesondere ist darauf hinzuweisen, daß auch ein vollkommen aus aktualen Elementen bestehendes Pattern verwendet werden kann. Das Pattern liefert jedoch nur die genau passenden Tuple als Ergebnis zurück.

Das Matching wird in Perl-Linda mit Regular Expressions durchgeführt. Das mitgegebene Pattern wird in eine korrespondierende Regular Expression umgewandelt und diese wird auf den Inhalt des Tuplespace angewandt. Grundsätzlich wird bei der Umwandlung jedes formale Element, das durch das Zeichen `{?}` makiert ist, durch die Regular-Expression `/[^,]+/` ersetzt. Diese Regular-Expression bedeutet soviel wie: *Es passen ein oder mehrere Zeichen außer dem Zeichen* `{,}`*, da dieses Trennzeichen für Tupleelemente darstellt.* Die aktualen Element des Tuple werden in Klartext in die Regular-Expression übernommen. Die Umwandlung von Pattern in Regular Expressions ist in Tabelle 15 ersichtlich. Die umgewandelten Pattern entsprechen den Pattern in Tabelle 14.

Mit der erzeugten Regular-Expression wird der Tuplespace elementweise durchsucht. Passende Tuple werden an den Client zurückgesandt und je nach aufgerufener Funktion aus dem Tuplespace gelöscht.

Pattern	Regular Expression
(?)	/[^,]+/
(a,b,?)	/a,b,[^,]+/
(?,b,?)	/[^,]+,b,[^,]+/
(a,c,4)	/a,c,4/
(x,?,3,?)	/x,[^,]+,3,[^,]+/
(?,?)	/[^,]+,[^,]+/

Tabelle 15: Umwandlung von Pattern in Regular Expressions

6.4.2 Erweitertes Matching in Perl-Linda

Mit Regular Expressions wurde in Perl-Linda, die Funktionalität des Matching von Tuple erweitert. Dadurch stehen in Perl-Linda außerdem noch folgende Möglichkeiten zur Verfügung:

Matching mit Wildcards:

Wenn längere Tuple oder eine größere Anzahl von Tuple unterschiedlicher Länge aus dem Tuplespace entfernt werden sollen, so ist dies mit der normalen Matching-Funktionalität nur durch mindestens einen `in()`- oder `nbin()`-Befehl pro Tuplelänge (Anzahl der Elemente) möglich. Es existiert in der Grundkonzeption kein Mechanismus, um alle Tuple mit dem gleichen Prefix auf einmal aus dem Tuplespace zu entfernen. Mit dem Pattern `(ABC,?*)` werden z.B. alle Tuple, die als erstes Element `ABC` haben, gematcht.

Durch den Einsatz des Wildcard-Operators `'?*'` lassen sich mehrere aktuale Elemente durch ein formales Element in einer Operation matchen (siehe Tabelle 16).

Verwendung von temporären Server-Variablen:

Bei manchen Linda-Anwendungen kann die Notwendigkeit bestehen, gewisse Abhängigkeiten zwischen den Tuple-Elementen zu prüfen. Die Verwendung von Regular Expressions für das Matching bietet die Möglichkeit, die Werte von Elementen auf lokale Variable zuzuweisen und diese an einer anderen Stelle der Regular Expression wieder zu verwenden. Dies kann sehr einfach zur Feststellung der Gleichheit zwischen Elementen verwendet werden.

Durch das Pattern `'?myvar'` wird der Wert des Elements der Variable

`myvar` zugewiesen und kann an späterer Stelle im Pattern wieder verwendet werden, z.B.:

`(ABC,?myvar,xy,?myvar)`

Durch dieses Pattern werden alle Tuple mit gleichem 2. und 4. Element gematcht.

Durch die Beschränkungen der Regular Expressions in Perl können maximal 9 verschiedene temporäre Variable im Pattern verwendet werden.

`pattern`	Beschreibung
`?`	Wird durch ein aktuales Element ersetzt werden. Dies entspricht dem normalen Linda-Matching.
`abc?`	Wird durch ein aktuales Element ersetzt, das mit der Zeichenfolge `"abc"` beginnt, z.B. `abcd` oder `abc123`, nicht jedoch `abc`.
`?x`	Wird durch ein aktuales Element ersetzt. Der Wert des ersetzten Elements wird der temporären Servervariable *x* zugewiesen. Wird *x* in einem späteren Element wieder verwendet, so muß der Inhalt der Elemente gleich sein, damit der Match erfolgreich ist (z.B. `(?x,?x)` trifft auf alle 2-Tuple zu, welche gleiche Elemente haben).
`abc?x`	Eine Kombination der oben beschriebenen Funktionalität.
`?*`	Joker, wird durch ein oder mehrere Elemente ersetzt.
`abc?*`	Kombination der oben beschriebenen Funktionalität. Das Pattern wird durch ein oder mehrere Elemente ersetzt. Das erste der ersetzten Elemente muß mit der Zeichenfolge `"abc"` beginnen.

Tabelle 16: Erweiterte Matching-Funktionalität in Perl-Linda

6.5 Stabilität und Recovery von Linda-Applikationen

In diesem Abschnitt werden Probleme, die die Stabilität von Linda-Anwendungen gefährden, vorgestellt. Es werden mögliche Lösungen für die Probleme in Perl-Linda aufgezeigt. Diese sind jedoch nicht implementiert.

6.5.1 Probleme

Eines der Hauptprobleme für die Stabilität einer Linda-Anwendung ist neben dem Absturz des Servers der Absturz eines an der Anwendung beteiligten Clients. Die Folgen des Absturzes beeinflussen die Anwendung dann, wenn der Client zum Zeitpunkt des Absturzes für die Anwendung wichtige Tuple aus dem Tuplespace

entfernt hat. Dies kann zur Nicht-Erledigung eines Teils der durch die Anwendung zu erledigenden Arbeit oder im schlimmsten Falle zum Steckenbleiben der gesamten Anwendung führen.

Bakken nennt zwei Hauptprobleme mit *Failures in Linda* [Bakken, 1993, vgl. S. 32f.]:

Lack of Tuple Stability:

Das Linda-Konzept kennt keine Mechanismen, die garantieren, daß ein Tuple nach Absturz des Server-Rechners im Tuplespace verbleibt. Dieses Problem kann nicht durch einen einzelnen Tuplespace gelöst werden, da die Client-Verbindungen nicht aufrechterhalten werden können. Auch die Speicherung und Wiederherstellung des Tuplespace-Inhalts kann nur für einen angekündigten Shutdown (z.B. durch das Signal `TERM`) implementiert werden.

Bakken löst dieses Problem durch Verwendung des *Replicated State Machine Approach* zur Replikation von Tuplespaces auf verschiedenen Rechnern. Er greift dann auf die Tuplespaces mittels *atomic Multicasts* zu. Fällt ein Tuplespace durch Maschinenversagen aus, kann die Operation über einen der replizierten Tuplespaces abgewickelt werden. Die Verwendung von atomic Multicasts stellt sicher, daß alle Tuplespaces die gleiche Information erhalten.

Lack of Sufficient Atomicity:

Linda besitzt *Single-Operation-Atomicity*. Dies bedeutet, daß jedes Kommando atomar ausgeführt wird. Bei der Entnahme von Tuple aus dem Tuplespace ist es für den korrekten Ablauf des Programmes wichtig, daß das entnommene Tuple – oder ein Äquivalent – nach der Bearbeitung durch den Client wieder in den Tuplespace gestellt wird.

Nehmen wir die Implementation einer *Shared Variable* als Beispiel: Eine Variable kann im Tuplespace durch das Tuple `(x,2)` dargestellt werden, wobei das erste Element den Variablennamen und das zweite Element den Wert darstellt. Um den exklusiven Zugriff eines Clients zur Veränderung der Variable zu gewährleisten, muß der jeweils agierende Client die Variable mit dem Befehl `in(x,?)` aus dem Tuplespace entfernen. Nach der Bearbeitung stellt der Client die Variable mit `out(x,`*wert*`)` wieder in den Tuplespace. Um ein Verlorengehen der Variable während der `in()-out()` Sequenz zu verhindern, muß diese als atomare Operation durchgeführt werden. Daraus ergibt sich, daß die gesamte `in()-out()`-Sequenz entweder vollständig oder gar nicht durchgeführt werden muß.

Dies gilt nicht nur für das oben genannte Beispiel, sondern auch für Sequenzen von `out`-Befehlen. Dieser Fall kommt z.B. bei Ausgabe der Ergebnisse von Clients vor, die nicht in einem Tuple untergebracht werden können bzw. sollen. So muß ein Client bei der Applikation *DiscTool* (siehe Abschnitte 9.2.2 und 10.8.2), die die Plattenauslastung in einem Netzwerk von Workstations anzeigt, die Daten über alle auf der Maschine befindlichen Plattenpartitionen in den Tuplespace stellen. Dies tut er mit einer Sequenz von `out()`-Statements, die atomar ausgeführt werden sollten.

Bakken bietet zur Erweiterung der Single-Operator-Atomicity *atomic guarded statements*. Diese werden bezüglich ihrer Auswirkungen auf den Tuplespace als eine Operation betrachtet, welche entweder vollständig oder gar nicht ausgeführt wird.

Da Perl-Linda in der jetzigen Version nur auf einem einzelnen Tuplespace basiert, kann der erstere Problemkreis nur durch *Checkpointing* gelöst werden. Der Inhalt des Tuplespace wird in vom Client bestimmten Abständen gesichert und kann nach einem Absturz wieder geladen werden. Alle Änderungen seit der letzten Sicherung gehen allerdings verloren.

Bezüglich des zweiten Problems bietet Perl-Linda eine serverseitige Lösung durch das Zur-Verfügung-Stellen der Kommandos `uin` und `uout`. Die Funktionsweise und die Implementation von Perl-Linda wird im nächsten Abschnitt vorgestellt.

6.5.2 Lösungen durch Erweiterung der Atomizität

Neben der durch `uin()` und `uout()` implementierten Erweiterung atomarer Operationen in Linda wird von Bakken [Bakken, 1993, S. 35] zur Lösung des *duplicated Tuple*-Problems auch eine atomare Operation für eine Folge von `out`-Befehlen vorgeschlagen. Diese Erweiterung bietet Client-Programmen die Möglichkeit, Ergebnisse von Operationen, die nicht in einem Tuple dargestellt werden können, atomar in den Tuplespace zu stellen. Im folgenden wird ein Ansatz zur Erweiterung des Perl-Linda-Servers beschrieben.

Eine Implementation von atomaren multi-`out()` Befehlen ist in Perl-Linda sowohl client- als auch serverseitig möglich. Beide Methoden eignen sich für bestimmte Zugriffstrukturen und sind mit Veränderungen an Client und Server verbunden. Das Funktionsprinzip soll im Folgenden nur überblicksmäßig beschrieben werden.

`mout` zur Übertragung von Tuplelisten:

Clientseitig können multiple atomare `out`-Befehle durch die Erweiterung des Linda-Befehlssatzes um den Befehl `mout` (multi out) erreicht werden. Der Befehl `mout` funktioniert im Prinzip wie `out`, mit dem Unterschied, daß er als Argument eine Tuplelist statt einem Tuple nimmt. Dies erfordert, daß im Client die zu übermittelnden Tuple vor der Versendung an den Tuplespace in eine Tuplelist umgewandelt werden. Diese wird in einer Operation an den Tuplespace geschickt. Nach dem Erhalt der Tupelist sendet der Perl-Linda-Server eine Bestätigung an den Client, welche diesem die erfolgreiche Ausführung des Befehls anzeigt.

Obwohl die Verwendung von `mout` eine elegante Möglichkeit der Erweiterung darstellt, ist sie nicht ohne Probleme. Der Client muß die Tuplelist in einer Operation an den Tuplespace schicken. Um die Integrität der gesendeten Daten zu überprüfen, muß der Tuplespace ihre Struktur auch in einer Operation mit Hilfe von Regular Expressions prüfen. Diese Vorgehensweise limitiert die Gesamtlänge der gesendeten Tuplelist auf ca. 30000 Zeichen. Dies entspricht der Begrenzung, welche Perl für Regular Expressions vorgibt.

Implementierung von expliziten Begrenzungen:

Diese Methode orientiert sich grundsätzlich an Bakkens Implementierung von *guarded statements* in FT-Linda. Die Idee ist ein Begrenzen der atomaren Übermittlung durch explizite Beginn- und Endmarken. Abbildung 16 stellt diese Methode dar.

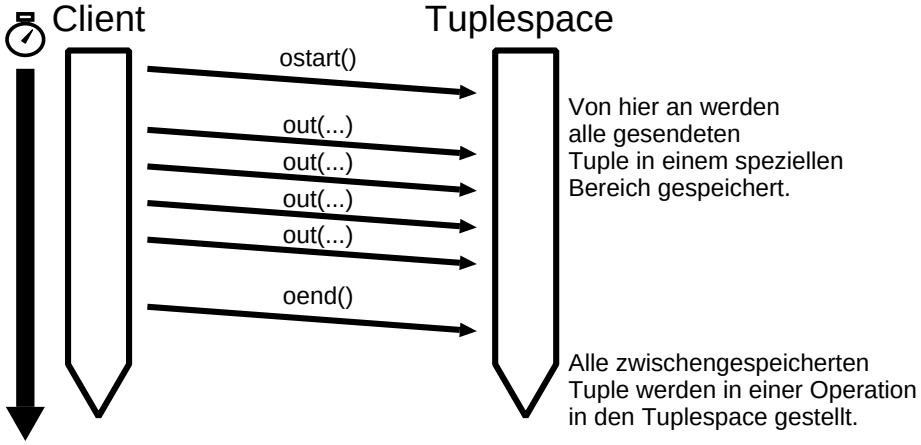

Abbildung 16: Multiple out-Statements durch explizite Markierung

Ähnlich der Behandlung des fault-toleranten Austausches von Tuple durch

`uin` und `uout` sollen bei der Behandlung von mehreren atomaren `out`-Befehlen, nach Übermittlung der Beginnmarke, die in den Tuplespace zu stellenden Tuple so lange im Server zwischengespeichert werden, bis der Client durch die Endmarke das Ende der Übertragung anzeigt.

Wird die Endmarke übermittelt, werden alle bis jetzt gespeicherten Tuple in den Tuplespace gestellt und die Transaktion so abgeschlossen. Bricht die Verbindung vor Übermittlung der Endmarke ab, werden alle bis jetzt gespeicherten Tuple aus der Zwischenspeicherung gelöscht und die Transaktion abgebrochen.

Diese Art der Implementation entlastet den Server, da er nicht eine ganze Tuplelist auf einmal verarbeiten muß, sondern nur jeweils ein Tuple zwischenspeichert. Weiters wird die Gefahr des Datenverlusts durch einen Verbindungsabbruchs während der Übertragung ausgeschaltet.

Die vorgestellten Möglichkeiten der Erweiterung der Atomizität vermindern den Datenverlust und die Fehlfunktion von Applikationen durch Ausfall eines Clients. Durch geeignete Wahl der Applikationsstruktur kann die Laufsicherheit einer Applikation durch den Einsatz von `uin()` und `uout()` auch ohne zusätzliche Erweiterung des Linda-Befehlsumfanges erhöht werden.

6.6 Restriktionen des Perl-Linda-Servers

Die Verwendung der Programmiersprache Perl ermöglicht eine portable Implementation des Perl-Linda-Servers. Durch die Implementation auf der Applikationsebene ist der Server einfach erweiterbar und für spezielle Bedürfnisse anzupassen.

Die Verwendung von Perl bringt jedoch nicht nur Vorteile mit sich, sondern auch Nachteile und Restriktionen für den Betrieb. Im folgenden sind die Probleme aufgelistet, die wir beim Server im Betrieb feststellen konnten.

Geschwindigkeit:
Perl ist eine Interpretersprache und daher bezüglich der Laufzeit grundsätzlich um einiges langsamer als Compiler-Sprachen wie z.B. C oder C^{++}. Die Verwendung von Listen und Regular Expressions kommt der Les- und Erweiterbarkeit des Programmes zwar sehr entgegen, doch benötigen die

Operationen, insbesondere das Matching von Pattern mittels Regular Expressions im Tuplespace und in der Waiting Queue Zeit. Daher treten bei folgenden Situationen oder deren gemeinsamen Auftreten überproportional lange Antwortzeiten beim Server auf:

- Viele Tuple im Tuplespace.
- Viel-elementige Tuple im Tuplespace.
- Viele blockierende Requests in der Waiting Queue. Hier kommt es besonders bei `out()`-Befehlen zu langen Antwortzeiten, da für jeden Request in der Waiting Queue geprüft werden muß, ob das in den Tuplespace zu stellende Tuple den Request erfüllt.
- Viele Clients und hohe Befehlsdichte. Im Server werden alle Client-Verbindungen auf eventuelle Befehle überprüft. Daher verlangsamt sich mit steigender Client-Anzahl die Gesamtperformance des Servers. Da die Befehle sequentiell abgearbeitet werden, stauen sich bei hoher Befehlsdichte die Befehle im Eingabepuffer, bis der Server sie bearbeiten kann.

Länge der Tuple:
Die Länge der Tuple ist mit ca. 30000 Zeichen begrenzt. Diese Restriktion resultiert aus der Verwendung von Regular Expressions. Das von Perl verwendete Regular Expression Paket kann diese nur bis zu dieser Länge verarbeiten, wenngleich die reine Listenstruktur auch längere Listen zuließe.

Freigabe des Serversocket:
Nachdem der Perl-Linda-Server mittels `kill()` oder durch einen Absturz beendet wurde, bleibt der Server-Socket bis zur Freigabe durch das Betriebssystem blockiert, sodaß der Server nicht sofort wieder hochgestartet werden kann. Diese Zeitspanne beträgt ca. 60 Sekunden. Nach dieser Zeit wird der Socket wieder freigegeben und der Server kann ohne Probleme gestartet werden. Dieser Umstand ist besonders beim automatischen Hochstarten des Servers zu beachten.

Diese Restriktionen hindern jedoch nach unserer praktischen Erfahrung nicht den Einsatz des Perl-Linda-Servers in Applikationen. Bei geschickter Wahl der Applikationsstruktur, des Verteilungsparadigmas und bei genügend grobkörniger Verteilung kann der beim Server auftretende Engpaß vermieden, bzw. vermindert werden.

6.7 Installation und Betrieb

Die folgenden Installations- und Betriebshinweise sind auf den Betrieb des Perl-Linda-Servers auf UNIX-Systemen abgestimmt. Um den Perl-Linda-Server zu verwenden, ist es notwendig, die Programmdateien mit den Funktionsbibliotheken (siehe Tabelle 17) in das Standard-Include Verzeichnis von Perl4.036 zu kopieren. Im Normalfall ist es eines der System-Verzeichnisse `/usr/lib/perl4`, `/usr/local/lib/perl` oder `/usr/local/lib/perl4`. Danach ist das Hauptprogramm in ein Verzeichnis entlang des Programmpfades zu kopieren. Damit ist der Server einsatzbereit und kann gestartet werden. Standardmäßig startet der Server auf Port `7999`. Wird ein anderes Port gewünscht, so kann man dem Server das Port als erstes Argument beim Aufruf mitgeben.

Programmname	Inhalt
`lindaserver`	Hauptprogramm, enthält die Server-Schleife mit der Verzweigung zu den Befehlsfunktionen.
`linda-ipc.pl`	Diese Datei enthält alle Dateien zur Interprozeßkommunikation.
`linda-func.pl`	Die serverseitige Implementation der Linda-Befehle befindet sich in dieser Datei.
`linda-serv.pl`	In dieser Datei befinden sich die Funktionen, welche mit dem Handling der Clients und dem Hoch- und Herunterfahren des Servers zu tun haben.
`linda-cli.pl`	Diese Datei ist nur der Vollständigkeit halber in der Server-Distribution enthalten und enthält die client-seitige Implementation der Linda-Funktionen für die Sprache Perl.

Tabelle 17: Programmübersicht der Perl-Linda-Distribution

Soll der Server ständig verfügbar sein, kann der Serveraufruf durch einen Eintrag in der Datei `/etc/inittab` (siehe Abbildung 17) vom Betriebssystem beim Hochstarten des Systems durchgeführt werden. Der Server wird dann auch automatisch wiedergestartet, sollte er während des Betriebs abstürzen oder heruntergefahren werden (kill). Der Befehl `sleep 60` erzeugt eine Wartezeit von 60 Sekunden vor dem neuerlichen Start des Servers. Diese Zeit braucht das Betriebssystem in etwa, um den Socket (in unserem Beispiel Port 8000) nach dem Herunterfahren oder Absturz des Servers wieder für die Nutzung freizugeben.

```
...
l1:5:respawn:/usr/local/bin/lindaserver 8000; sleep 60}
...
```

Abbildung 17: Eintrag in `/etc/inittab` für den automatischen Start des Servers

Der Perl-Linda-Server ermöglicht dem Programmierer mittels einer `telnet`-Session eine Client-Verbindung zum Server aufzubauen. Damit kann der Programmier alle Linda-Befehle manuell eingeben und so in der Testphase mit seinen Client-Programmen Verbindung aufnehmen und diese testen. Abbildung 18 gibt ein Beispiel für das Hochstarten des Servers und die Eingabe von einfachen Linda-Befehlen.

6.8 Die Client-Schnittstelle zun Perl-Linda-Server

Die Client-Schnittstelle stellt ein Interface zwischen dem Perl-Linda-Server und der jeweiligen Programmiersprache dar, für die es implementiert ist. Mit der Client-Schnittstelle wird der Funktionsumfang der Programmiersprache um jene Funktionalität erweitert, die es ermöglicht, die Services des Perl-Linda Servers zu nutzen. Die Befehle gliedern sich in folgende Gruppen:

Connect:
: Der Connect-Befehl ermöglicht dem Client einen Verbindungsaufbau zum Server. Es wird eine verbindungsorientierte Verbindung über einen `tcp-socket` zum Host und Port, auf dem der Server läuft, aufgebaut. Der Client bekommt vom Server einen Namen zugewiesen, welchen er für die Dauer der Verbindung behält.

Linda-Befehle:
: Diese Gruppe von Befehlen entspricht den Linda-Befehlen im Perl-Linda-Server und stellt den richtigen Aufruf der Befehle sicher. Die Linda-Befehle schicken das jeweilige Kommando zusammen mit den Daten an den Server und erwarten die Antwort bzw. Bestätigung des Servers. In dieser Gruppe findet die eigentliche Kommunikation und das eventuelle Blockieren des Clients statt.

```
wallace:~/sci/linda> lindaserver &
  [1] 183
  starting on port 7999.
wallace:~/sci/linda> telnet wallace 7999
  Trying 137.208.51.193...
  Connected to wallace.
  Escape character is '^]'.
lst()
  tuplespace:
  -------------
  ()
  -------------
  .
status()
  STATUS Information
  --------------------------
  Tuplespace=     0       Tuples
  WaitingQueue=   0       Tuples
  UpdateQueue=    0       Tuples
  .
out(This,is,my,first,tuple)
  .
out(x,2)
  .
lst()
  tuplespace:
  -------------
  (This,is,my,first,tuple)
  (x,2)
  -------------
  .
in(?,2)
  x,2
  .
lst()
  tuplespace:
  -------------
  (This,is,my,first,tuple)
  -------------
  .
bye()
  Connection closed by foreign host.
wallace:~/sci/linda>
```

Abbildung 18: Beispielsitzung mittels `telnet`

De- und Vercodierungsbefehle:
Um ein vollständiges 256-Zeichen Alphabet für das Abspeichern von Daten in den Elementen der Tuple zu erhalten, dürfen einige Zeichen aus dem ASCII-Zeichensatz nicht als Datenzeichen verwendet werden. Diese Zeichen werden im Tuplespace in vercodierter Form dargestellt. Die De- und Vercodierungsfunktion ermöglicht den Transfer von Tuple von der vercodierten Server-Darstellung in die der jeweiligen Programmiersprache zugrundeliegende Datenstruktur.

Close:
Mit dem Close-Befehl wird dem Server durch das Schicken des Kommandos `bye()` angezeigt, daß der Client die Verbindung zu beenden wünscht. Der Server löscht dann alle mit dem Client in Zusammenhang stehenden Daten, beendet die Verbindung und gibt den zugewiesenen Namen wieder frei.

Diese Funktionen stellen das Minimum dar, um das die Funktionalität der Programmiersprache erweitert werden muß. Die Funktionen sollten aber nicht fix in die Sprache eingebaut werden, sondern als Funktionen in einer Linda-Bibliothek zur Verfügung stehen. Dieses Konzept läßt sich sowohl bei kompilierten als auch bei interpretierten Sprachen verwirklichen. Die Vorteile gegenüber dem fixen Einbau in eine Sprache wie z.B. bei [Carriero and Gelernter, 1989a] [Carriero and Gelernter, 1991] sind die Portierbarkeit auf andere Systeme und der fehlende Bedarf für weitere Tools (Precompiler, Runtime-Systeme, etc.). Beispiele für die Implementierung der Client-Schnittstelle für Perl-Linda finden sich in den Abschnitten 7 und 8.

Neben der Implementation der Funktionen sind auch noch die entsprechenden Datenstrukturen für die Darstellung von `Tuple` und `Tuplelist` in die jeweilige Sprache zu implementieren. Diese Datenstrukturen sollten drei Erfordernissen genügen, welche sich bei Linda-Programmen immer wieder als notwendig herausgestellt haben:

- Es sollte leicht möglich sein – eventuell durch Operatoren – die in der Sprache standardmäßig vorhandenen Datenstrukturen in ein Tuple umzuwandeln. Die Verwendung von einer leicht umzuwandelnden Tuplestruktur vereinfacht in erster Hinsicht den Datentausch zwischen Clients der eigenen Programmiersprache. In weiterer Hinsicht erleichtert es auch dem unerfahrenen oder neuen Programmierer von Linda-Applikationen, sich unter dem

Tuple-Konzept etwas vorzustellen und dieses für seine Programme zu verwenden.

- Das Entnehmen eines einzelnen Elements aus einer Tuplelist durch Angabe des Index des Tuples in der Liste und des Index des Elements im Tuple. Fast alle Linda-Befehle liefern eine Struktur vom Typ `Tuplelist` zurück (siehe Tabelle 10). Diese Struktur kann null bis mehrere Tuple enthalten. Es erweist sich bei Linda-Programmen oft als Notwendigkeit, ein einzelnes Element aus dieser Tuplelist-Struktur zu entnehmen (z.B. aktueller Zählwert beim Hochzählen einer verteilten Variable).

- Die Datenstrukturen sollten ohne größeren Aufwand in einer Datei abzuspeichern sein. Weiters sollten bereits abgespeicherte Datenstrukturen von dieser Datei wiederum lesbar sein. Dies bietet neben der Speicherung von Tuple durch den Tuplespace die Möglichkeit, Tuple im Client selbst abzuspeichern. Diese Methode hat sich bei Erstellung von Save- und Backup-Clients für bestimmte Applikationen bewährt.

Diese Erfordernisse lassen sich nicht in allen Sprachen in ihrer elegantesten Form implementieren, doch erleichtert eine bessere Implementation die Integration in das Gesamtkonzept der Sprache. Ausserdem wird das Verständnis und die Lesbarkeit der Programme erhöht.

7 Perl-Linda-Client

Der Perl-Linda-Client ist eine Schnittstelle zum Perl-Linda-Server in der Programmiersprache Perl Version 4.036 [Wall and Schwartz, 1990]. Der Client erweitert die Programmiersprache, sodaß die Möglichkeit geschaffen wird, die Funktionalität des Perl-Linda-Servers von einem normalen Perl-Programm zu nutzen.

Der Perl-Linda Client stellt im Vergleich mit den in Abschnitt 8 vorgestellten Clients die einfachste und mit der wenigsten Funktionalität ausgerüstete Schnittstelle dar. Dennoch ist Perl eine flexible Sprache für Anwendungen kleineren Umfanges. Die Einsatzgebiete sind aber doch limitiert. Schwerpunktmäßig sollte der Perl-Linda-Client bei folgenden Problemkreisen eingesetzt werden:

Prototypen und Programme kleineren Umfangs:
Perl ist eine Script-Sprache, die das schnelle Erstellen von Prototypen zuläßt. Aufgrund der kompakten Stuktur sind die Scripts auch zumeist umfangmäßig kurz.

Das Ziel bei der Erstellung von Prototypen für Linda-Applikationen ist in den meisten Fällen die Feststellung der Tauglichkeit des Einsatzes von Linda für die entsprechende Aufgabe und die Demonstration der Funktion. Es ist hier weniger die Geschwindigkeit des Protoyps entscheidend, als das Präsentieren eines geeigneten Lösungsansatzes. Der Perl-Linda-Client bietet hier auch die Möglichkeit, die Scripts ohne großen Übersetzungsaufwand zu verändern und auf andere Systeme zu portieren.

Legacy-Applikationen und Linda:
Die Einbindung von Legacy-Applikationen – zumeist Großrechnerbasierte Anwendungen, für die es keine Schnittstellen gibt – in ein Informationssystem bereitet zumeist Probleme mit den Schnittstellen. Die Anbindung einer solchen an Linda bietet die Möglichkeit, mit der Applikation über ein standardisiertes, erweiterbares Interface zu kommunizieren. Durch die mächtigen Funktionen für die Behandlung von Zeichenketten und Pipes bietet Perl gegenüber anderen Programmiersprachen den Vorteil, den Programmieraufwand für das Interface klein zu halten.

Input/Output Processing für Linda-Applikationen:
Interfaces haben bei Linda-Applikationen in den meisten Fällen die Aufgabe, die im Tuplespace enthaltenen Daten darzustellen. Da bei dieser Auf-

gabe die Geschwindigkeit eine untergeordnete Rolle spielt, kann der Geschwindigkeitsnachteil von Perl durch die Flexibiliät in der Verarbeitung und Aufbereitung von Daten aufgewogen werden.

Im folgenden wird der Perl-Linda-Client beschrieben. Nach einer allgemeinen Übersicht einer Demonstration der Funktionalität anhand eines Beispiels erfolgt die Beschreibung der Funktionen und deren Einsatzmöglichkeiten.

7.1 Allgemeines

Tuple werden im Perl-Client als Listen dargestellt. Da Perl keine enkapsulierten Listenstrukturen zuläßt, können keine *Listen von Listen* dargestellt werden. Zur Darstellung von Tuplelisten müssen die Tuple in das in Tabelle 5 beschriebene Format übergeführt werden. Erst dann können sie als Liste gespeichert werden. Dieses Prozedere sieht auf den ersten Anblick recht umständlich aus, ist in der Anwendung jedoch einfach. Durch den Umstand, daß in allen Linda-Befehlen jeweils nur ein Tuple oder Pattern mitgeschickt wird, kann dieses in der normalen Listenform von Perl dargestellt werden.

Tuplelists treten jeweils nur in den Antworten des Servers auf. Da die Antworten des Servers ohnehin zuerst vom vercodierten Server-Format in das decodierte Perl-Format umgewandelt werden müssen, geschieht die Unterteilung in Tuple auch in diesem Prozeß.

Der Perl-Linda-Client besteht im groben aus zwei Programmpaketen, die auf dem Rechner installiert sein müssen. Die Linda-Routinen werden in der Datei `linda-cli.pl` zur Verfügung gestellt und sind als normale Funktionen aufrufbar. Die Datei muß mittels des Befehls `require` in das Programm inkludiert werden. Neben der oben genannten Datei muß weiters noch das Paket `chat2.pl` installiert sein, welches in der normalen Perl-Distribution enthalten ist. `chat2.pl` ist für das Erstellen und das Handling der Verbindung zum Server zuständig.

7.2 Funktionen des Perl-Linda-Client

In diesem Abschnitt werden die Funktionen des Perl-Linda-Client und deren Anwendung in kurzen Beispielen beschrieben. Neben den Linda-Befehlen `in()`,

`nbin()`, `ain()`, `uin()`, `rd()`, `nbrd()`, `ard()`, `out()` und `uout()`, welche schon in Abschnitt 5.3 beschrieben wurden, sind in der Client-Schnittstelle noch einige weitere Funktionen implementiert, die sich mit der Behandlung von Tuple befassen. Diese gliedern sich in folgende Gruppen (eine genaue Erklärung findet sich in Tabelle 18):

Auf- und Abbau der Verbindung:
`register_client()`, `close_client()`

Linda-Funktionen:
`in()`, `nbin()`, `ain()`, `uin()`, `rd()`, `nbrd()`, `ard()`, `out()`, `uout()`, `update()`

Ver- und Entcodierungsfunktionen:
`entrans()`, `detrans()`

Ausgabe:
`printout()`

Zugriffsfunktion:
`element()`

Da die Datenstrukturen von Perl keine implizite Verkapselung zulassen, muß dies jedesmal explizit durchgeführt werden. Dies wird notwendig, wenn Tuple zu einer Tuplelist zusammengesetzt werden bzw. aus dieser wieder extrahiert werden sollen. Während Ersteres zumeist nur vom Server gemacht wird, so muß Letzteres bei jeder Antwort des Servers durchgeführt werden.

In Tabelle 18 sind die Funktionen des Perl-Linda Clients aufgelistet. In den nachfolgenden Beispielen werden oft gebrauchte, aber nicht in Funktionen gepackte Source-Beispiele mit der Verwendung der Funktionen gezeigt. Die Argumente und Rückgabewerte der Funktionen sind normale Perl-Datenstrukturen und können mit allen anderen Perl-Operationen verarbeitet werden. Im folgenden Abschnitt wird die Syntax von Perl-Linda am Beispiel des Count-Clients [Schönfeldinger, 1995] erklärt.

Funktion	Beschreibung
`register_client(`*host, port*`)`	öffnet die Verbindung zum Tuplespace. Als Argumente nimmt sie den *Hostnamen* als Zeichenkette und das *Port* in Zahlenform. `register_client` retourniert den Returncode der `open()`-Funktion, die auf Systemebene den Socket öffnet. Dieser Returnwert ist *TRUE* für die aufgebaute Verbindung.
`close_client()`	ist das Pendant zu `register_client`. Die Funktion schließt die offene Verbindung durch Senden des Kommandos `bye()`. Die Funktion wartet das explizite Beenden der Verbindung durch den Server ab.
`entrans(`*tuple*`)`	wandelt ein Tuple in Perl-Listenform in ein für den Server vercodiertes Tuple um und gibt es als Zeichenkette zurück.
`detrans(`*encoded_tuple*`)`	ist die Umkehrfunktion zu `entrans()`. Das Tuple wird als (vercodierte) Zeichenkette als Argument der Funktion mitgegeben. `detrans` gibt es decodiert in Perl-Listenform zurück.
`printout(`*tuplelist*`)`	dient als Ausgabefunktion für Tuple und Tuplelist. Die Funktion erwartet als Argument ein Tuple oder eine Tuplelist in Listenform und gibt es aus. Diese Funktion ist weniger für die Anwendung als für das Austesten von Linda-Programmen gedacht.
`element(`*tuple,elem,tuplist*`)`	automatisiert ein Problem, das oft im Zusammenhang mit Tuplelisten entsteht. Oft ist es nur notwendig, ein Element eines Tuple der Tupleliste zu extrahieren. Mittels `element` kann man bei Angabe des Tuple und der Position des Elements das Element in einem Schritt extrahieren.
`update(`*tuple,pattern*`)`	ist eine Implementation der `uin/uout`-Kombination. Das als erstes Argument angegebene Tuple ersetzt das durch das Pattern bestimmte Tuple.

Tabelle 18: Perl-Linda-Client-Funktionen

7.3 Count.pl, ein Beispiel für einen Perl-Linda-Client

Das Programm `count.pl` ist ein Programm, das kooperatives Zählen mehrerer unabhängiger Clients ermöglicht. Es wird eine im Tuplespace gespeicherte Variable von allen Clients gemeinsam hinaufgezählt. Wenn das Programm für sich gesehen nicht sehr sinnvoll ist, so stellt es doch ein gutes Beispiel für die Koordinationsfähigkeit von Linda dar. Der Zählprozess ist durch die Anzahl der Clients beliebig verteilbar. Die Clients werden durch Linda automatisch koordiniert.

Die Variable wird von 1 bis 10000 hinaufgezählt. Sobald der Wert 10000 erreicht ist, stoppen alle Clients und beenden die Verbindung. Der nun folgende Source-Code demonstriert die Darstellung von Tuple im Perl-Linda-Client und die Verarbeitung der Antworten des Server.

```
#!/usr/local/bin/perl
require "linda-cli.pl";
```

Der Perl Interpreter wird aufgerufen. Die Client-Bibliothek wird mit `require` geladen. In dieser Bibliothek befinden sich alle für den Client wichtigen Funktionen.

Es ist zu beachten, daß `linda-cli.pl`, um erfolgreich in das Client-Programm integriert werden zu können, im Standardverzeichnis für Include-Dateien liegen muß. Weiters muß die ebenfalls benötigte Datei `chat2.pl` sich auch dort befinden.

```
&register_client("aig", 7999)
 || die "Cannot open Server !!\n";
```

Mit diesem Befehl meldet sich der Client beim Perl-Linda-Server an. Es müssen der *Hostname* oder die *IP-Adresse* zusammen mit der *Portnummer* des Servers angegeben werden. Die Funktion gibt zurück, ob der Verbindungsaufbau erfolgreich war. Die Perl-Funktion `die` beendet das Programm mit der angegeben Fehlermeldung, wenn der Verbindungsaufbau nicht erfolgreich war.

Es kann pro Client nur eine Server-Verbindung aufgemacht werden. Diese Re-

striktion entspringt der Implementation auf Basis von `chat2.pl`. Ebenfalls kann neben der Server-Verbindung keine weitere Socket-Verbindung aufrechterhalten werden. Wird dies doch benötigt, so müssen die Verbindungen abwechselnd geschlossen werden.

Im Paradigma des *Agenda-Parallelism* [Carriero and Gelernter, 1989a, vgl. S. 326] sind mehrere gleiche *Worker-Clients* an der Abarbeitung einer Agenda beteiligt. Im Beispiel des Count-Client ist dies die wiederkehrende Aufgabe des Hinaufzählens einer Variable. Diese Agenda muß jedoch beim Start der Bearbeitung definiert und initialisiert werden. Dieser Prozeß kann entweder durch einen explizit dafür vorgesehenen *Init-Client* oder durch den ersten der Worker-Clients durchgeführt werden. Bei der letzteren Methode muß jeder Client vor Beginn seiner Beteiligung am Gesamtprozeß feststellen, ob er der erste Client ist oder ob die Aufgabe schon durch einen anderen Client initialisiert wurde.

In unserem Beispiel wird das Singleton `(COUNTBEGIN)` als Zeichen der bereits erfolgten Definition der Aufgabe in den Tuplespace gestellt. Dies ist ein Beispiel für die Verwendung von Singletons. Da ein Singleton in der Regel keinen Wert sondern nur einen Namen repräsentieren kann, wird es zumeist als Marke für den Eintritt eines Ereignisses verwendet. Jeder Client versucht, bevor er mit seiner Aufgabe beginnt, das Singleton `(COUNTBEGIN)` zu lesen. Ist es bereits im Tuplespace enthalten, kann er zur Bearbeitung der Aufgabe übergehen. Wenn nicht, stellt er das Tuple in den Tuplespace und initialisiert das Problem durch das Definieren der Shared Variable `COUNT`.

```
@tl=&nbrd("COUNTBEGIN");
&out("COUNTBEGIN"), &out("COUNT",1)
        if @tl==();
```

Das Resultat (eine Tuplelist) der Operation `nbin` wird der Liste `@tl` zugewiesen. Es wird im weiteren die Länge dieser Liste überprüft. Ist `@tl` eine leere Liste, so war das Starttuple nicht im Tuplespace enthalten. Der Client stellt dann das Starttuple und die Variable in den Tuplespace.

Der eigentliche Hauptteil des Programms besteht aus einem `in()`-Befehl, welcher die Variable einliest und einem `out()`-Befehl, der sie wieder in den Tuplespace zurückstellt. Der Wert der Variable wird aus dem Tuple entnommen und erhöht. Mit diesem Wert wird dann das Tuple neu gebildet und in den Tuplespace gestellt.

```
while($var<10000)
{
        @tl=&in("COUNT", "?");
        &printtuple(@tl);
        $var=&element("last",2,@tl);
        $var++;
        &out("COUNT",$var);
}
```

Der augenblickliche Wert der Variable wird in `$var` gespeichert. Mittels `in()` werden die die Variable repräsentierenden Tuple aus dem Tuplespace entfernt und in der Liste `@tl` gespeichert. Die Funktion `element()` wird verwendet, um auf ein Element eines Tuple dieser Liste zugreifen zu können. Es wird auf das 2. Element des letzten Tuple der Tuplelist zugegriffen. Das Ergebnis dieser Operation wird in `$var` gespeichert, `$var` wird erhöht und mit `out` an den Tuplespace gestellt.

Jedes Element der von `in()` zurückgelieferten Liste repräsentiert ein Tuple. Es wird zur Kontrolle die zurückgelieferte Tuplelist mittels der Funktion `printtuple()` ausgegeben. So läßt sich kontrollieren, welchen Wert der Shared Variable jeder Client erhält.

Statt des zum Einlesen des Tuple verwendeten Befehl `in()` könnte bei diesem Beispiel auch der Befehl `ain()` verwendet werden, der nur ein Tuple einliest. Die Verwendung von `in()` hat einen positiven Nebeneffekt: werden durch einen Koordinationsfehler mehrere Variablen-Tuple in den Tuplespace gestellt, so werden durch `in()` alle eingelesen, jedoch nur mehr ein Tuple in den Tuplespace zurückgestellt.

Die Beendigung einer Anwendung mit mehreren Clients ist nicht mehr so einfach wie bei der Verwendung nur eines aktiven Clients. Es müssen alle beteiligten Clients mitbekommen, daß die Aufgabe erledigt ist. Darum muß jeder Client, der anhand des Werts der Variable das Ende seiner Aufgabe feststellt das Tuple vor dem Beenden wieder in den Tuplespace zurückstellen.

```
&out("COUNT", 10000);
&close_client();
```

Der Client stellt nach Beendigung der `while`-Schleife das Tuple wieder in den Tuplespace zurück und beendet die Verbindung durch die Funktion `close_client()`. Diese sendet im wesentlichen den Befehl `bye()` an den Server und wartet bis der Server die Verbindung unterbricht.

Das Beispiel veranschaulicht, daß der Perl-Linda-Client eine in die Sprache eingebundene Schnittstelle zum Perl-Linda-Server darstellt. Es werden die in Perl4.036 vorhandenen Strukturen und das bereits vorhandene Programmpaket (`chat2.pl`) verwendet. Dadurch läßt sich die Flexibilität der Sprache Perl zur schnellen Erstellung von Clients kleineren Umfanges und deren Nähe zum UNIX-Betriebssystem nutzen.

7.4 Anwendung der Perl-Linda-Funktionen

Im folgenden Abschnitt wird die Anwendung der Funktionen von Perl-Linda anhand von kleinen Beispielen gezeigt. Das erste Beispiel bezieht sich auf das sichere Öffnen der Verbindung zwischen Server und Client. Es muß hier das Fehlschlagen des Verbindungsaufbaus abgefangen werden.

```
$server="aig.wu-wien.ac.at";
$port=7999;
&register_client($server,$port) ||
     die "Cannot open Server\n";
```

Dies ist die in Perl übliche Form, das Fehlschlagen eines Kommandos abzufangen. Das Kommando `die` bricht das Programm mit der entsprechenden Fehlermeldung ab.

```
$server="aig.wu-wien.ac.at";
$port=7999;
if(!&register_client($server,$port))
{
    # Client side Output
    exit;
}
```

Bei Verwendung von Perl-Linda mit Anwendungen, die eine Ausgabe mittels die nicht zulassen (z.B. World Wide Web), ist diese Möglichkeit des Öffnens der Verbindung vorzuziehen. Hier kann im Block des `if`-Statements eine für den Client angepaßte Fehlerbehandlung durchgeführt werden.

Die Funktionen `entrans` und `detrans` dienen zum Umwandeln der Tuple von Perl-Datenstrukturen in das vom Server geforderte Format (siehe Abschnitt 6.3) und umgekehrt. Es wird dabei ein Perl-Listentyp in einen Perl-Skalartyp umgewandelt, der mit anderen wieder zu einer Liste zusammengesetzt werden kann. Die Umwandlung von Tuple und Pattern erfolgt automatisch in den Linda-Funktionen.

```
@tuple=("Is","this a","tuple ????");
$enc=&entrans(@tuple);
```

Die in `tuple` gespeicherte Liste wird durch `entrans` in eine skalare Form gebracht. Das Ergebnis wird von `entrans` zurückgeliefert. Es ist zu beachten, daß die 4 Fragezeichen am Ende in 2 Fragezeichen umgewandelt werden.

Dieser Vorgang wird nur beim Abspeichern von Tuple im Server-Format benötigt. Es haben sich in den Applikationen noch keine anderen Verwendungszwecke expliziter Umwandlung von Listen- in Skalarform ergeben.

Öfters hingegen wird die umgekehrte Form der Umwandlung gebraucht. Jede Antwort des Servers erhält der Client in Form einer Tuplelist, die für die weitere Verarbeitung umgewandelt werden muß. Dies erfolgt in zwei Schritten: zuerst muß die Tuplelist in die einzelnen Elemente aufgespaltet werden, zweitens müssen diese dann mittels `detrans` in die Perl-Listenform gebracht werden. Eine Ausnahme dazu stellt das Testen der Rückgabewerte von `nbrd()` und `nbin()`-Befehlen dar. Wie schon am Beispiel des Count-Client gezeigt, wird hierbei nur geprüft, ob Tuple zurückgegeben wurden.

```
@tl=&nbrd("SomeTuple");
print "Tuple exists"
   if @tl!=();
```

Es erfolgt hierbei keine Umwandlung der Tuplelist, sondern lediglich eine Überprüfung, ob die Liste Elemente enthält.

```
@tl=&in(@pattern);
for(@tl)
{
     @curtuple=&detrans($_);
     # now you can do something
     # with the tuple in listform
     # ...
}
```

Die von `in` retournierte Tuplelist wird in einer Schleife elementweise bearbeitet. Jedes Tuple wird in der Schleife in das Listenformat übersetzt und kann dann weiterbearbeitet werden.

Der Einsatz dieser Methode ist dann empfehlenswert, wenn viele der zurückgelieferten Tuple bearbeitet werden sollen. Weiters empfiehlt sich der Einsatz, wenn die Tuple als Ganzes bearbeitet werden. Wenn jedoch nur wenige Tuple der Tuplelist oder nur einzelne Elemente bearbeitet werden, kann mittels des Indexoperators `[ ]` auf diese explizit zugegriffen werden.

```
@tl=&in(@pattern);
@_2ndTup=&detrans($tl[1])
     if $#tl>0;
```

In diesem Beispiel wird nur das zweite Tuple der zurückgelieferten Tuplelist zur Bearbeitung übersetzt. Es muß in diesem Zusammenhang jedoch überprüft werden, ob überhaupt zwei Tuple zurückgeliefert wurden.

Für die Bearbeitung einzelner Elemente in der Tuplelist steht die Funktion `element()` zur Verfügung. Sie extrahiert ein Element eines Tuple aus einer Tuplelist und liefert es zurück. Die Decodierung erfolgt automatisch.

```
@tl=&in(@pattern);
$elem=&element(2,4,@tl);
```

Aus der Tuplelist `@tl` wird das vierte Element des zweiten Tuple zurückgeliefert. Statt des Index des Tuple in der Tuplelist, können auch die symbolischen Indizes `"first"` und `"last"` verwendet werden, die sich auf das erste bzw. letzte Tuple in der Tuplelist beziehen.

Die Funktion `update()` ist als atomarer Befehl zum Auswechseln eines Tuple im Tuplespace vorgesehen. Der Einsatzbereich dieser Funktion ist jedoch durch ihre Starrheit eingeschränkt. Diese ergibt sich aus der Tatsache, daß sowohl das Pattern als auch das Tuple vor dem Aufruf feststehen müssen.

```
$tuple=&entrans("x",2,"max",4);
@pattern=("x","?","max","?");
&update($tuple,@pattern);
```

Zuerst werden die Argumente der Funktion erzeugt. Das Tuple muß bereits in vercodierter Form sein. Das Pattern hingegen muß sich in der Listenform befinden. Danach wird die Funktion `update` als atomarer Befehl ausgeführt.

Dieser Befehl sieht sehr umständlich aus, ist aber durch die fehlende Möglichkeit zur Enkapsulierung in Perl4.036 nicht anders lösbar. Viel öfters als die `update`-Funktion wird man die mit `uin` und `uout` ausprogrammierte Variante verwenden. Dies ist dort unumgänglich wo das ersetzende Tuple erst nach der Auswertung des Ergebnisses der `in()`- bzw. `rd()`-Operation gebildet werden kann.

```
@tl=&uin("x","?","max","?");
($x,$max)=(&element("first",2,@tl),
           &element("first",4,@tl));
&uout("x",$x+1,"max",$max+$x);
```

Nach der Extraktion der Elemente aus der Antwort des Servers wird das neue Tuple berechnet und mit `uout` an den Server gesandt.

Die in diesem Abschnitt vorgestellten Funktionen und deren Anwendung des Perl-Linda-Client werden in Abschnitt 10 an Beispielen demonstriert. Im nächsten Abschnitt behandeln wir Client-Schnittstellen zum Perl-Linda-Server in anderen Sprachen, insbesondere in C^{++} und APL.

8 Perl-Linda-Schnittstellen zu anderen Sprachen

Während die Client-Schnittstelle in Perl das schnelle Bauen von Prototypen und flexiblen Anwendungen ermöglicht, bieten die Client-Schittstellen anderer Programmiersprachen die Möglichkeit zur Nutzung der Stärken dieser Sprachen in Linda-Applikationen. Zur Zeit existieren Client-Schnittstellen zu den Programmiersprachen C^{++} und APL.

Die Schnittstellen sind vom Aufbau grundsätzlich wie die Perl-Schnittstelle beschaffen, doch bieten die Implementierungen darüber hinausgehend Vorteile, die in Perl durch die Beschränkungen der Sprache nicht zu implementieren waren.

Die folgenden Abschnitte geben einen Überblick über die Implementation von C^{++}-Linda und APL-Linda [Hietler, 1995]. Als Illustration der Syntax wird der schon bei Perl-Linda verwendete Count-Client in den jeweiligen Sprache erklärt. Es werden die Besonderheiten der jeweiligen Schnittstelle erklärt und gegenüber der Perl-Schnittstelle abgegrenzt.

8.1 Die C^{++}-Linda Schnittstelle

Die Implementierung von Linda in C^{++} zeichnet sich durch die Nutzung der Möglichkeiten der Sprache aus. Es wird über die reine Kommunikation mit dem Server hinaus ein Rahmen zur Repräsentation von Linda-Objekten geschaffen. Die C^{++}-Client-Schnittstelle bietet Geschwindigkeit und Erweiterbarkeit und wurde schon zufriedenstellend für einige größere Linda-Applikationen eingesetzt.

Der C^{++}-Client kann für ein breites Spektrum von Applikationen eingesetzt werden. Besonders eignet sich der Einsatz für Client-Programme mittleren und größeren Umfangs. Unter diese Gruppe fällt auch die Ein- und Anbindung von Linda für bereits bestehende, im Source-Code vorhandene Applikationen. Ein weiteres Kriterium stellt die im Vergleich zu Perl schnellere Laufgeschwindigkeit der Programme dar.

Entwickelt wurde die C^{++}-Linda-Schnittstelle von Gerold Hietler und Christoph Leithner im Frühjahr 1995 im Rahmen einer Projektarbeit für das Anwendungsprojekt aus *"Programmieren in C^{++}"*. Es wurde schon in mehreren Applikationen erfolgreich eingesetzt und getestet. Der Source-Code der ersten Version ist

public domain[5] über `ftp` verfügbar.

8.1.1 Grundstruktur

Die Client-Schnittstelle bietet die Möglichkeit zur Nutzung von Linda aus C^{++}-Programmen. Es implementiert die Darstellung von Linda-Objekten in der Sprache C^{++} und bietet weiters Möglichkeiten an, Tuple und Tuplelisten aus den Standard-Datenstrukturen zu erzeugen. Die Schnittstelle wurde als Erweiterung von C^{++} auf der Applikationsebene implementiert. Es sind zur Verwendung der Erweiterung keine zusätzlichen Compiler oder Bibliotheken notwendig.

Die Implementation besteht grundsätzlich aus drei Teilen: der Darstellung der Linda-Datenstrukturen, den Linda-Funktionen und dem Programm-Interface, das die Schnittstelle zum Benutzer für den leichteren Umgang mit Linda-Programmen und für das Testen darstellt.

Für das Erstellen einer Applikation muß die Linda-Header-Datei in das Client-Programm inkludiert werden. Beim Übersetzen des Programmes müssen die entsprechenden Programmdateien mitübersetzt werden. Sind diese bereits in einer Programmbibliothek vorhanden, so wird diese lediglich beim Linken des Programmes dazugelinkt.

8.1.2 Datenstrukturen

Die Datenstruktur besteht aus den drei Klassen: TupleList, Tuple und Element.

TupleList: einfach gelinkte Liste von Tuples

Tuple: einfach gelinkte Liste von Elementen

Element: ist das eigentliche Datenelement, es enthält die Daten in Form von Zeichenketten. Diese werden wiederum in einem Buffer gespeichert, der dynamisch an die jeweils geforderte Elementlänge angepaßt wird.

[5] Die C^{++}-Linda Schnittstelle ist auf dem ftp-Server `ftpai.wu-wien.ac.at` im Verzeichnis */pub/Linda/clients/cpp/* verfügbar.

Die Erfassung und Verarbeitung der Tuple-Daten erfolgt in Form von gelinkten Listen. Das Grundelement der Datenstruktur ist das Element. Es kann einen Text, eine Integerzahl, eine Floatzahl oder auch andere Datenstrukturen repräsentieren. Der Inhalt ist jedoch immer als Zeichenkette gespeichert. Elemente werden zu einem Tuple zusammengefaßt, das in Form einer gelinkten Liste von Element-Pointern gespeichert wird. Mehrere Tuple können auch als Tuplelist gespeichert werden, wobei sich sowohl das einzelne Tuple als auch einzelne Elemente des Tuple indiziert ansprechen lassen. Durch die Verkapselung der Strukturen Element und Tuple können die vom Tuplespace erhaltenen Daten besser und einfacher als im Perl-Linda-Client bearbeitet werden. In Abbildung 19 sind die Datenstrukturen von C^{++}-Linda dargestellt.

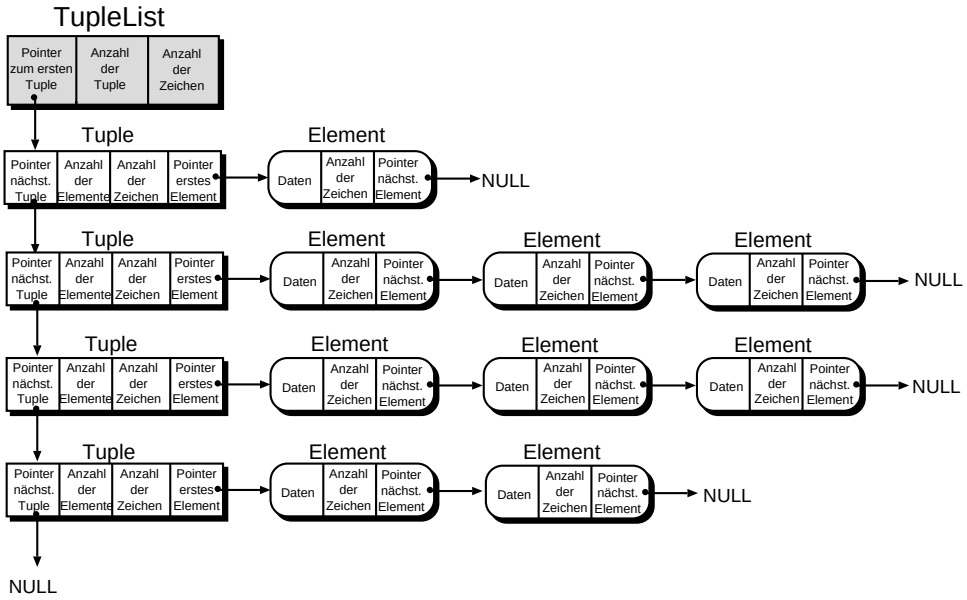

Abbildung 19: Datenstrukturen des C^{++}-Client

Mit diesen Datenstrukturen arbeiten alle Funktionen und Operatoren der C^{++}-Schnittstelle. Durch die Benutzung einer dynamischen Bufferstruktur, ist die Länge der Elemente nicht begrenzt. Dies trägt, wenngleich diese Lösung laufzeitmäßige Nachteile bringt, entscheidend zur Erweiterbarkeit der C^{++}-Schnittstelle bei. In der Datenstruktur `Tuple` werden sowohl Tuple als auch Pattern gespeichert.

Ein Tuple oder Pattern kann im C^{++}-Client entweder durch den expliziten Aufruf eines Konstruktors oder durch Verwendung des Stream-Operators << (Put To)

gebildet werden (siehe Abbildung 20). Die Elemente des Tuple-Objektes können über die Elementfunktionen manipuliert werden.

```
// Erstellung mit Konstruktor:
     Tuple t("Three","Element","Tuple",NULL);
     Tuple t1;
     t1=Tuple("3","Element","Tuple",NULL);

// Erstellung mit Stream-Operatoren:
     Tuple t2;
     t2 << "Three" << "Element" << "Tuple";
```

Abbildung 20: Erstellung eines Tuple im C^{++}-Client

Neben der Implementierung der Linda-Funktionen gibt es auch noch eine Reihe von Funktionen und Operatoren, die eine Transformation von C^{++}-Standarddatentypen in den Element-Datentyp und umgekehrt zulassen.

8.1.3 Funktionen und Operatoren

Die Linda-Funktionen der C^{++}-Schnittstelle haben zwei unterschiedliche Parametersätze, mit denen sie aufgerufen werden können. Selten werden fertig im Programm vorhandene Tuple und Pattern verarbeitet. Diese ergeben sich oft *on the fly* erst beim Aufruf der jeweiligen Funktion. Um diese dynamische Konstruktionsmöglichkeit wahrzunehmen, wurde von der Verkapselung in Objekte abgewichen und die Methode des Aufrufs mit variablen Argumenten (Varargs) implementiert. Bei dieser Methode wird das Tuple nicht als Ganzes, sondern es wird nur eine Folge von gültigen `char*` (Pointer auf `char`) übergeben.

Aufruf mit Tupleobjekt:

```
Connection con("Somehost", 7999);
Tuple t("This is a Tuple","15", NULL);

con.out(t);
```

Aufruf mit variablen Argumenten:

```
Connection con("Somehost", 7999);

con.out("This is a Tuple","15", NULL);
```

Aufgrund der Implementation von Varargs in C^{++} muß diese Folge von einem ausgezeichneten Element terminiert sein (vgl. Abbildung 20). Als dieses Element wurde ein Null-Pointer gewählt. Das Tuple wird erst in der entsprechenden Funktion erzeugt. Wird ein Tuple oder Pattern in der Datenstruktur `Tuple` übergeben, so kann dieses gleich weiterverarbeitet werden.

Aus dem Linda-Befehlssatz stehen folgende Funktionen zur Verfügung:

- `int out(char*, ..., NULL)`
- `int out(Tuple)`
- `int out(TupleList)`
- `TupleList* in(char*, ..., NULL)`
- `TupleList* in(Tuple)`
- `TupleList* nbin(char*, ..., NULL)`
- `TupleList* nbin(Tuple)`
- `TupleList* rd(char*, ..., NULL)`
- `TupleList* rd(Tuple)`
- `TupleList* nbrd(char*, ..., NULL)`
- `TupleList* nbrd(Tuple)`
- `TupleList* ain(char*, ..., NULL)`
- `TupleList* ain(Tuple)`
- `TupleList* uin(char*, ..., NULL)`
- `int uout(Tuple)`
- `int uout(char*, ..., NULL)`

Durch die Verkapselung und die damit verbundene Trennung der Elemente können die Antworten des Servers gleich in dieser Form weiterverarbeitet werden. Sie bedürfen keiner weiteren Umwandlung.

Die Linda-Funktionen sind als Elementfunktionen der Klasse `Connection` implementiert. Daher ist es mit dem C^{++}-Linda API möglich, mehr als eine Verbindung zu Perl-Linda-Servern aufrecht zu erhalten. Jede Verbindung wird als eigenes Objekt repräsentiert, auf das alle oben beschriebenen Funktionen und Operatoren anwendbar sind. Damit können durch C^{++}-Clients Verbindungen zu verschiedenen Servern hergestellt werden und die Daten der Applikation somit auch über mehrere Tuplespace verteilt werden.

8.1.4 Der Count-Client in C^{++} Linda

Anhand des **Count-Client** soll nun die Anwendung der Tuplelist-Datenstruktur und der C^{++}-Linda-Funktionen erklärt werden.

```
// File: COUNT.cc
#include "lindac.h"

int main(){
  Connection c;
  // *** OPEN LINDA-CONNECTION ***
  if(!c.register_client("aig",
    7999)) {
  cerr << "NO CONNECTION" << endl;
  return 0;
  }
```

Der Aufbau der Verbindung zum Perl-Linda-Server ist funktionsgleich zum Perl-Linda-Client und bereits dort ausführlich behandelt worden. An dieser Stelle soll näher auf die Tuplelist-Datenstruktur und die C^{++}-Linda Funktionen eingegangen werden.

Anders als beim Perl-Client ist die Verbindung zum Server nicht als globales Element im Programm sondern als eigenständiges Objekt vorhanden. Alle Linda-Funktion sind somit Elementfunktionen dieses Objektes, in unserem Beispiel die Connection `c`.

```
// *** COUNT - PROGRAM ***
Tuplelist *cnt_lst=NULL;
Tuple cnt_qu("COUNT","?",NULL);
int cnt;

Tuplelist *begin_lst=NULL;
Tuple begin_qu;
begin_qu << "COUNTBEGIN";
```

Zunächst werden die notwendigen Tuples und Tuplelist-Pointer erzeugt.

Tuple können entweder explizit vor dem Aufruf erzeugt werden oder den Funktionen als Argumente mitgegeben werden. Bei diesem Programm wurde aus Demonstrationsgründen die explizite Form gewählt.

```
if(!(begin_lst=c.nbrd(begin_qu))){
  c.out("COUNT","1",NULL);
  c.out(begin_qu);
}
```

Einfacher als beim Perl-Client läßt sich beim C^{++}-Client durch eine einfache logische Operation feststellen, ob ein `nbrd()` oder `nbin()`-Befehl erfolgreich war.

Beim Testen des Ergebnisses der `nbrd()`-Funktion gibt es zwei Ausgänge:

begin_lst==NULL:
Wenn das Tuple nicht im Tuple-Space ist, wandelt der C^{++}-Client die Antwort in einen NULL-Pointer um. Für den Count-Client bedeutet das, daß noch keiner der Clients zu zählen begonnen hat. Es werden daher die Tuple `begin_qu` und `(COUNT,1)` in den Tuplespace gestellt.

begin_lst!=NULL:
Wenn das Tuple im Tuple-Space ist, erzeugt der C^{++}-Client eine Tuplelist aus der Antwort und gibt den Pointer auf diese Tuplelist zurück. Für den Count-Client bedeutet das, daß bereits mit dem Zählen begonnen wurde.

```
do {
  cnt_lst=c.in(cnt_qu);
  cout << *cnt_lst;
  cnt=cnt_lst->getint(1,2);
  cnt++;
  cnt_lst->update(1,2,cnt);
  c.out(*cnt_lst);
  delete cnt_lst;
  cnt_lst=NULL;
} while(cnt < 10000);
```

Die do-Schleife bewirkt, daß der Client zählt, solange die `while`-Bedingung erfüllt ist.

Die einzelnen Schritte eines Zählvorganges sind:

cnt_lst=c.in(cnt_qu); Der Zähler (das Tuple `(COUNT,<ZAHL>)` wird als Tuplelist eingelesen. Die `in()`-Funktion läßt den Server nach dem Argument-Tuple im Tuplespace suchen. Wenn das Tuple nicht im Tuplespace ist, wartet der Server so lange mit der Antwort auf die Abfrage, bis er von einem anderen Client ein Tuple erhält, das der Abfrage entspricht. Der C^{++}-Client erzeugt aus dieser Antwort eine Tuplelist und gibt den Pointer auf diese Tuplelist zurück. Beim Count-Client besteht diese Tuplelist in jedem Fall aus einem Tuple mit 2 Elementen `(COUNT,<ZAHL>)`.

cnt=cnt_lst->getint(1,2); Der Inhalt des 2. Elementes im 1. Tuple der Tuplelist, die auf der Adresse `cnt_lst` liegt, wird in eine Integer-Zahl umgewandelt und der Integer-Variable `cnt` zugewiesen.

cnt++; Die Integer-Variable `cnt` wird um 1 erhöht.

cnt_lst->update(1,2,cnt); Der Wert der Integer-Variable `cnt` wird in einen String umgewandelt. Mit diesem String wird der Inhalt des 2. Elementes im 1. Tuple der Tuplelist, auf die der Tuplelist-Pointer `cnt_lst` zeigt, ersetzt.

cout << *cnt_lst; Die Tuplelist wird am Bildschirm ausgegeben.

out(*cnt_lst); Die modifizierte Tuplelist `cnt_lst` wird in den Tuplespace zurückgeschrieben.

delete cnt_lst;cnt_lst=NULL; Die dynamisch allozierte Tuplelist, auf die der Tuplelist-Pointer `cnt_lst` zeigt, wird gelöscht und der Speicherplatz wieder freigegeben. `cnt_lst` wird auf `NULL` gesetzt.

```
  // *** CLOSE LINDA-CONNECTION ***
  c.out("COUNT","10000",NULL);
  return c.close_client();
}
```

Die Verbindung zum LINDA-Server wird abgebrochen und das Programm beendet.

Durch die Einbettung der Funktionalität in die Objekte und die leicht bearbeitbaren Datenstrukturen haben die C^{++}-Programme im Sourcecode in etwa die gleiche Länge als die Perl-Programme. Die Datenstrukturen bieten auch die Möglichkeit, Tuplelists auf Dateien abzuspeichern und von diesen wieder zu lesen.

8.2 APL-Linda

Die APL-Linda Schnittstelle ist eine Erweiterung der Sprache APL für die Verwendung des Perl-Linda-Servers. Wie auch schon die in den letzten Abschnitten vorgestellten Schnittstellen, wurde APL um die Kommunikation mit dem Perl-Linda-Server erweitert. Es wurden die Linda-Funktionen und Koversionsfunktionen zur Um- und Rückwandlung der APL-Datenstrukturen in das Perl-Linda-Format. Da die verschiedenen APL-Versionen einen unterschiedlichen Zugang zum Betriebssystem besitzen, konnte kein einheitliches API implementiert werden. Es existieren Client-Schnittstellen für Dyalog-APL und APL/2. Die folgenden Ausführungen beziehen sich auf Dyalog-APL. Eine genauere Beschreibung der APL-Client Schnittstelle findet sich in [Hietler, 1995].

Das Einsatzgebiet von APL findet sich in der Stärke der Sprache im mathematischen Bereich. Durch Einsatz bzw. Einbindung eines APL-Client in eine Applikation können Auswertungen und Berechnungen der Gesamtapplikation nach dem Paradigma des *Specialist Parallelism* [Carriero and Gelernter, 1989a, vgl S. 326] durchgeführt werden.

8.2.1 Grundstruktur

Die APL-Linda Schnittstelle ermöglicht sowohl die Übermittlung von Einzeldaten als auch von verschachtelten Daten und Programmen. Der Client besteht aus der Implementation der Linda-Befehle in APL und einem *Auxilliary Processor* in C,

der die Netzwerkanbindung ermöglicht.

Im Unterschied zur Implementation in Perl, ermöglicht die APL-Schnittstelle gleichzeitige Verbindungen zu mehreren Tuplespace. Jeder Verbindung wird eine Verbindungsnummer zugewiesen, die bei allen Befehlen angegeben werden muß. APL-Linda Funktionen haben jeweils als linkes Argument die Verbindungsnummer und als rechtes Argument die eigentlichen Funktionsparameter.

8.2.2 Umwandlung der APL-Datenstrukturen

Der APL-Client ermöglicht das Senden von geschachtelten APL-Strukturen als Elemente eines Tuple. Zuerst werden die Strukturen in eine Standard APL-Transferform gebracht. In dieser Form können sie als APL-Text Zeichenkette dargestellt werden. Eine genaue Beschreibung der Transferform und des Verfahrens findet sich bei [Mitlöhner, 1992]. Hat man die Textform erhalten, müssen noch die Linda-Sonderzeichen ersetzt werden. Alle diese Vorgänge sind in APL implementiert. Hat man das Tuple in der passenden Form, wird es noch in APL-Text an den Auxilliary Processor übergeben, der bei der Ausführung des Kommandos die Daten von APL-Text auf `ASCII`-Text umwandelt. Der umgewandelte Text wird dann an den Perl-Linda-Server verschickt. Die Umwandlung der vom Server erhaltenen Ergebnisse erfolgt vice versa.

Sofern keine APL-spezifischen Konstrukte wie APL-Zeichen oder geschachtelte Strukturen verwendet werden, kann der entstehende Text von jedem anderen Client gelesen und verarbeitet werden. Bei der Verwendung APL spezifischer Ausdrücke ist ein sinnvolles Lesen nur durch andere APL-Clients möglich. Dies ermöglicht auch einen Daten- und Programmaustausch zwischen verschiedenen APL-Implementationen. Somit stellt für APL die Anbindung an Linda eine eingeschränkte Alternative zu Mitlöhners Workspace-Transfer Verfahren dar [Mitlöhner, 1992].

8.2.3 APL-Linda-Befehle

Die implementierten Befehle sind in Tabelle 19 dargestellt. `CON_NO` stellt dabei die Verbindungsnummer dar, auf die sich das entsprechende Kommando bezieht. Als Tuple kann jede beliebige APL Struktur verwendet werden. Die Vektorelemente auf der obersten Ebene werden als Elemente des mitzugebenden Tuple oder

Pattern genommen.

Linda-Befehl	APL-LINDA command
out(<*tuple*>)	<*CON_NO*> OUT <*TUPLE*>
in(<*tuple*>)	<*CON_NO*> IN <*TUPLE*>
rd(<*tuple*>)	<*CON_NO*> RD <*TUPLE*>
nbin(<*tuple*>)	<*CON_NO*> NBIN <*TUPLE*>
nbrd(<*tuple*>)	<*CON_NO*> NBRD <*TUPLE*>
ain(<*tuple*>)	<*CON_NO*> AIN <*TUPLE*>
uin(<*tuple*>)	<*CON_NO*> UIN <*TUPLE*>
uout(<*tuple*>)	<*CON_NO*> UOUT <*TUPLE*>

Tabelle 19: Implementierte Linda-Befehle

Es wurden in den APL-Client keine zusätzlichen Funktionen implementiert, da es wenig tunlich erscheint, den Server über eine APL-Schnittstelle zu administrieren.

8.2.4 Count-Client in APL

Um die Anwendung von Linda anhand eines Programmes zu zeigen, wird im folgenden der Count-Client in APL-Linda präsentiert. Dieser ist, wie schon der Count-Client in C^{++}, mit allen anderen Clients kompatibel, d.h., die Clients können ein verteiltes Zählen bewerkstelligen, unabhängig davon, in welcher Sprache sie ablaufen.

```
     ∇ Z←CON COUNT MAX;
       COUNTER
[1]    CON_NO←INIT_CLIENT CON
[2]
[3]    →(⎕AV[47 4]≠(CON_NO
       NBRD 'COUNTBEGIN'))/
       LOOP
[4]    ⎕←'INITIERE DAS
       ZAEHLEN'
[5]    CON_NO OUT 'COUNTBEGIN'
[6]    CON_NO OUT 'COUNT' 1
```

Wie bei den anderen Clients wird die Verbindung zum Server geöffnet (Zeile 1). Danach wird festgestellt, ob der Client schon der erste Teilnehmer ist (Zeile 3). Ist er der erste Client, werden die Initialisierungstuple an den Tuplespace gesandt (Zeilen 5-6).

Die bei der Eröffnung der Verbindung mitgegebene `CON_NO` identifiziert die Verbindung. Sie wird im folgenden bei allen Linda-Kommandos mitgegeben.

```
[7]    LOOP:
[8]    Z←CON_NO OUT 'COUNT'(
       COUNTER←1+(2⊃1⊃(CON_
       NO IN 'COUNT' '?')))
[9]    ⎕←'COUNT' COUNTER
[10]   →(COUNTER<MAX)/LOOP
```

Hier sehen wir die eigentliche Arbeitsschleife des Client. Das Tuple wird mittels `IN` aus dem Tuplespace entfernt, der Wert erhöht und mittels `OUT` wieder in den Tuplespace gestellt. Dies geschieht in APL in nur einer Zeile (Zeile 8). Ist der Maximalwert, der der Funktion als Argument mitgegeben wird, erreicht, wird die durch `LOOP` markierte Schleife beendet.

Der kompakte Syntax von APL bietet prinzipiell die Möglichkeit, komplexe Programme in wenigen Zeilen zu schreiben. Speziell die Zusammenfassung der `IN` und `OUT`-Befehle in Zeile 8 bietet ein gutes Beispiel dafür. Ähnlich wie schon beim C^{++}-Client, muß bei jedem Linda-Befehl die `CON_NO` angegeben werden, welche die Verbindung zu einem Server identifiziert.

```
[11]    CLOSE_CLIENT CON_NO
[12]    Z← 'ENDE'
    ▽
```

Nach Erfüllung seiner Aufgabe bricht der Client die Verbindung zum Server ab.

Neben den bereits implementierten Schnittstellen zu Perl, C^{++} und APL sind in Zukunft noch weitere Anbindungen geplant. Besonders geeignet sind dafür Scriptsprachen wie z.B. Python oder Sprachen, welche dominierende Eigenschaften auf möglichst vielen Plattformen implementieren, wie z.B. *'J'*.

9 Applikationen mit Perl-Linda

In diesem Abschnitt wird der Entwurf und die Implementation einer Problemstellung mit Perl-Linda behandelt. Nach einer Darstellung der Entwicklungsschritte für ein paralleles Programm in Perl-Linda werden die vorgestellten Entwicklungsschritte am Beispiel des *Ping-Pong* Client durchgeführt. Als Abschluß wird je ein Beispiel für die einzelnen Entwicklungsparadigmen *Result-*, *Specialist-* und *Agenda-Parallelism* beschrieben.

9.1 Entwurf eines Linda-Programmes

> *"To write a parallel program, (1) choose the conceptual class that is most natural for the problem; (2) write a program using the method that is most natural for that conceptional class; and (3) if the resulting program is not acceptably efficient, transform it into a more efficient version by switching from a more-natural method to a more-efficient one."*
> [Carriero and Gelernter, 1989a, S. 326]

Carriero und Gelernter sehen die Entwicklung paralleler Software als dreistufigen Prozeß:

1. Als erstes wird für das zu lösende Problem eine *konzeptionelle Klasse* ausgewählt, die für die Art des Problems und die zu Verfügung stehenden Ressourcen passend ist. Es sollte die Klasse gewählt werden, die sich intuitiv für die Problemlösung am besten eignet.
2. Im weiteren wird dann unter Berücksichtigung der gewählten Klasse für das Problem ein passender Algorithmus und eine Implementationsmethode erarbeitet. Genauso wie bei der Wahl der Klasse sollte hier auch eine intuitiv passende Methode gewählt werden.
3. Ist dieser Algorithmus nicht effizient genug, so kann die Problemlösung von einem intuitiveren Algorithmus auf einen effizienten Algorithmus übergeleitet werden.

Linda unterstützt diesen Prozeß insofern, daß es Bausteine für den Entwurf und die Programmierung eines parallelen Programmes bietet. Mit den in Linda zur

Verfügung stehenden Operationen können parallele Clients entworfen und programmiert werden, ohne daß der Programmierer die technische Seite der Kommunikation und Koordination zwischen den Programmteilen explizit ausprogrammieren muß. Anlehnend an die oben dargestellten Schritte stellen sich die Entwicklungsschritte für ein Linda-Programm folgendermaßen dar:

1. *Wahl einer geeigneten konzeptionellen Klasse für das Problem:* Dieser Schritt überdeckt sich mit der von Carriero und Gelernter aufgezeigten Vorgangsweise.

2. *Festlegen der Struktur und Aufgaben der an der Gesamtanwendung beteiligten Clients:* Dies ist der wichtigste Schritt im Entwicklungsprozeß, da bei ihm die eigentliche Verteilung der zu erledigenden Aufgaben auf die einzelnen Programmteile erfolgt. Hat man im ersten Schritt eine ungünstige Auswahl bei der konzeptionellen Klasse getroffen, so sollte es spätestens bei diesem Schritt auffallen.

3. *Festlegen des Protokolls zwischen den einzelnen Applikationsteilen:* In diesem Schritt wird das Protokoll zwischen den einzelnen Applikationsteilen definiert. Es legt fest, welche Daten von welchen Clients in welcher Form in den Tuplespace gestellt werden. Es muß dabei auch überprüft werden, ob es in der Applikation zu Deadlocks kommen kann. In den meisten Fällen ist eine Handsimulation für die Überprüfung auf Deadlocks ausreichend.

4. *Festlegen von Start und Beendigung der Applikation:* Bei Applikationen mit mehreren Clients muß festgelegt werden, wie die Applikation gestartet wird und wie alle Clients von der Beendigung der Applikation beim Erreichen der Endbedingung (soweit es sich nicht um eine Serverapplikation handelt) in Kenntnis gesetzt werden.

5. *Implementation der Applikationsclients und Testen der Gesamtapplikation:* Dieser Prozeß kann stufenweise erfolgen. Die Applikationsclients sollten einzeln getestet werden. Hierbei kann der Entwickler die erforderliche Testsituation im Tuplespace manuell mittels einer `telnet`-Verbindung herstellen. Die Ausgaben des Clients kann er, sofern sie in den Tuplespace zurückgegeben werden auch mittels des administrativen Kommandos `lst()` überprüfen.

   Sind alle Clients getestet, so kann die Gesamtapplikation und somit die Zusammenarbeit der Clients getestet werden. Der Entwickler kann sich hier der gleichen Möglichkeiten wie beim Testen eines einzelnen Clients bedienen.

Im folgenden werden die einzelnen Phasen des Entwicklungsprozesses genauer ausgeführt. Es werden die in jeder Phase zur Verfügung stehenden Möglichkeiten aufgezeigt. Am Beispiel des Ping-Pong-Client wird der ganze Entwicklungsprozeß demonstriert.

9.1.1 Konzeptionelle Klassen für parallele Verarbeitung

Als konzeptionelle Klassen unterscheiden Carriero und Gelernter, ausgehend von der Sichtweise, in der das Gesamtproblem gesehen wird, *Result Parallelism*, *Agenda Parallelism* und *Specialist Parallelism*. Diese drei Formen sind nicht direkt an die Implementation gebunden, sondern stellen abstrakte Sichtweisen dar, die sich auf der obersten Ebene auf die nachfolgenden Schritte auswirken. Im folgenden sind diese drei Klassen genauer ausgeführt [Carriero and Gelernter, 1989a, vgl. S. 325f]:

Result Parallelism:

Hier wird das Ergebnis als Hauptziel bei der Lösung des Problems gesehen. Dieses wird, soweit möglich, in gleiche Teile gespalten. Diese werden von identischen Clients simultan erarbeitet. Die Aufsplittung in parallel zu bearbeitende Teilprobleme setzt voraus, daß diese unabhängig voneinander lösbar sind. Das Gesamtergebnis wird dann aus allen Teilergebnissen zusammengesetzt werden.

Specialist Parallelism:

Der Specialist Parallelism liegt im Vergleich zum Result Parallelism am anderen Ende des Spektrums. Bei dieser konzeptionellen Klasse wird von zwei Grundannahmen ausgegangen, die den Specialist Parallelism nur für eine spezielle Klasse von Problemen implementierbar machen:

- Es sind spezielle Fähigkeiten zum Erreichen des Gesamtziels notwendig. Das Gesamtproblem wird nicht wie beim Result in gleiche Teile gespalten, sondern nach dem Kriterium der für die Bearbeitung der Teile benötigten Fähigkeiten.
- Die bearbeitenden Clients sind nicht homogen, sondern jeweils mit speziellen Fähigkeiten ausgestattet. Diese machen sie für das Bearbeiten eines Aspekts der Gesamtaufgabe besonders geeignet.

Es wird davon ausgegangen, daß die spezialisierten Teile der Gesamtlösung auch parallel ausgeführt werden können.

Agenda Parallelism:

Beim Agenda-Parallelism wird von der Existenz eines Arbeitsplanes, einer Agenda, ausgegangen. Die an der Erarbeitung des Gesamtresultats beteiligten Clients wählen von dieser Agenda einen noch nicht bearbeiteten Auftrag aus und bearbeiten diesen. Sowohl die gerade bearbeiteten, als auch die schon fertiggestellten Aufträge müssen in irgendeiner Form gekennzeichnet sein.

Diese Klasse stellt ein Mittelding zwischen den beiden vorgestellten Klassen dar. Es werden keine homogenen Teilprobleme für die Clients zur Verfügung gestellt, die Teilprobleme sind aber auch nicht nach den speziellen Voraussetzungen zur Lösung dieser untergliedert. Es können sowohl gleichartige wie auch verschiedenartige Clients verwendet werden.

Mittels Linda kann zwar der konzeptionelle Aufwand nicht verringert werden, jedoch stehen durch die Linda-Operationen sowohl ein funktioneller Rahmen für den konzeptionellen Entwurf als auch in der jeweiligen Programmiersprache bereits implementierte Kommunikations- und Koordinationsprimitive zur Verfügung.

Die Auswahl der konzeptionellen Klasse als oberste Entscheidung im Entwicklungsprozeß hat Auswirkungen auf die Komplexität der Teilprozesse und die Struktur der Applikation. Diese sind in Tabelle 20 übersichtsmäßig dargestellt.

Die dargestellten Ausprägungen sind generell für die jeweilige Klasse gültig, es gibt in jeder Klasse natürlich Beispiele, auf die die dargestellte Charakteristik nicht zutrifft. Man kann aus Tabelle 20 entnehmen, daß die Wahl des Result Parallelism zwar weniger komplexe Clients aber eine aufwendigere Struktur der Gesamtapplikation bedingt, während es bei der Wahl des Agenda Parallelism gerade umgekehrt ist.

9.1.2 Aufgabenteilung zwischen Client und Server

Bei der Verteilung der Aufgaben muß man zwischen zwei grundsätzlichen Arten der Kooperation der Teilprozesse in einer Applikation unterscheiden:

Zusammenarbeit von homogenen Clients:

Bei dieser Art der Kooperation hat jeder der Clients die gleichen Fähig-

Komplexität der Teilprozesse

konzeptionelle Klasse	benötigte Fähigkeiten	Prozesse	Implementation
Result Parallelism	Eine	Ein	Einfach
Specialist Parallelism	Eine	Mehrere	...
Agenda Parallelism	Mehrere	Mehrere	Schwer

Applikationsstruktur

konzeptionelle Klasse	Anzahl der Client-Prozesse	Koordinationsbedarf bei der Verteilung
Result Parallelism	Hoch	Groß
Specialist Parallelism	Mittel	Mittel
Agenda Parallelism	Adjustierbar	Gering

Tabelle 20: Charakterisierung der konzeptionellen Klassen, Quelle:[Carriero and Gelernter, 1989a, vgl. S. 335]

keiten und Aufgaben. Die Gesamtheit der zu bearbeitenden Task wird zwischen den beteiligten Clients aufgeteilt. Diese Form kommt bei reinen Rechenapplikationen, die den Zweck der Verteilung der Rechenlast haben, vor.

Zusammenarbeit nach dem Client/Server-Prinzip:
Hier hat der Entwickler der Applikation die Wahl, Aufgaben zwischen Client und Server aufzuteilen. Die Form der Aufteilung bestimmt die Struktur der Applikation und die Kooperation zwischen Client und Server. Diese Form kommt bei der praktischen Anwendung speziell bei Applikationen mit Benutzerinteraktion vor.

Bei Standalone-Applikationen ist die ganze Funktionalität, Ein- und Ausgabe in einem Programm implementiert, welches auf einem Rechner läuft. Für die Erstellung von Client-Server Programmen muß über den Rahmen der grundsätzlichen Aufgaben hinaus auch noch über die Verteilung zwischen Client und Server entschieden werden. Für den Rahmen der Betrachtung genügt es, hierbei bei der Art der Ausführung der Applikation zwischen Ausführung am Server (serverside) und Ausführung am Client (clientside) zu unterscheiden:

Serverside Computing:
Bei dieser Applikationsstruktur ist das Hauptgewicht der Funktionalität zen-

tral auf einem Server etabliert. Die Aufgaben der eventuell vorhanden Clients ist es, die Darstellung und Behandlung der Ein- und Ausgaben zu übernehmen.

Die Vorteile liegen bei dieser Struktur in der einfachen Handhabung und Wartung, da ein Großteil der Funktionalität zentral änderbar ist. Neben dem eigentlichen Datentausch mit dem Client wird durch abstraktere Dienste und Datenstrukturen versucht, neben den Daten auch Objekte und Prozeduren zu exportieren.

Clientside Computing:

Im Gegensatz zur vorher vorgestellten Applikationsstruktur werden die Hauptteile der Applikation vom Client ausgeführt. Der Server übernimmt in diesem Falle nur eine Rolle als koordinierendes Element.

Durch die Kenntnis der Speicherungsstruktur der Daten am Server ist es dem Client möglich, selbständig auf diese zuzugreifen und diese zu verarbeiten. Dadurch wird eine größere Flexibilität des Front End ermöglicht. Es kann auch eine Einbindung von benutzerdefinitierten Erweiterungen implementiert werden, durch welche, wie sich am Standardsoftwaremarkt an diversen *Visual*-Makrosprachen erkennen läßt, die Akzeptanz der Applikationen deutlich gesteigert wird.

Die Entscheidung beim Entwurf der Applikationsstruktur liegt in der Verteilung der Aufgaben auf Client und Server. Abbildung 21 zeigt die Problematik dieser Entscheidung. Werden in der Applikation zu viele Funktionen auf der Server-Seite erledigt, dann ist der Server zwar für einen großen Teil der Funktionalität verantwortlich, benötigt aber zumeist eine relativ große Rechenleistung. Dies gilt vor allem bei Multi-User-Applikationen. Auf der anderen Seite bedingt eine schwerpunktmäßige Funktionalität im Client einen größeren Aufwand bei der Kommunikation und bei den Protokollen. Weiters ergibt sich ein Mehraufwand bei der Koordination zwischen den Clients.

Während sich der auf der Seite des Clients entstehende Rechenbedarf durch die Anzahl der Clients skalieren läßt, stehen auf der Server-Seite solche Möglichkeiten nicht ohne zusätzliche Verteilungsmaßnahmen zur Verfügung. Abbildung 22 stellt diese Problematik dar.

Einen Ansatz für die Lösung der Skalierungsproblematik auf der Serverseite bietet die serverseitige Aufteilung der Arbeit auf mehrere Rechner. Dies kann durch den Einsatz von Linda erzielt werden. An der Server-Seite wird dadurch eine Multi-Client-Struktur aufgebaut, wie sie in Abbildung 23 zu sehen ist. Das Back

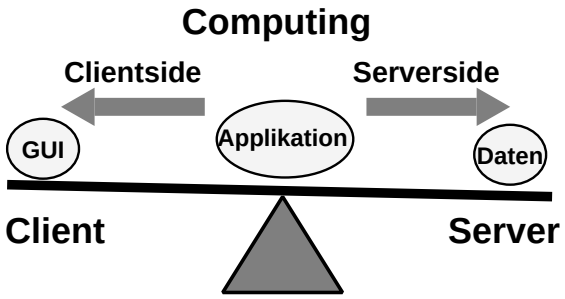

Abbildung 21: Clientside/Serverside Computing

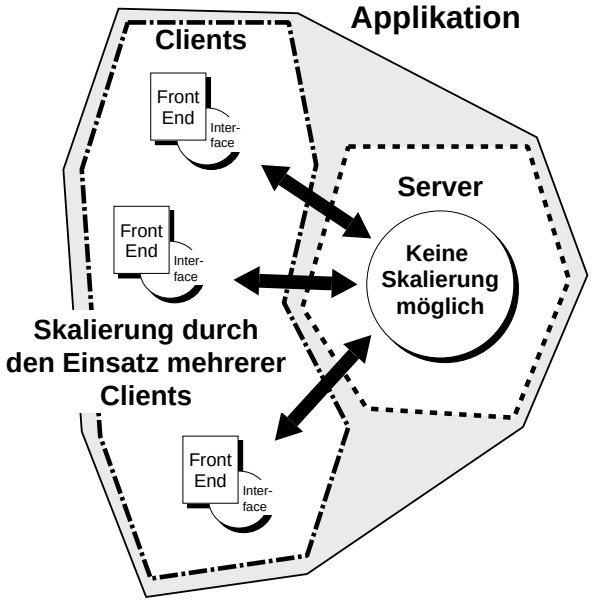

Abbildung 22: Skalierungsmöglichkeiten bei Client/Server

End, das die Funktionalität des Server darstellt, besteht nicht nur mehr aus einem monolithischen Programm, sondern wird durch eine Anzahl von selbstständigen Programmteilen repräsentiert, die die Bearbeitung des Programmes in paralleler Verarbeitung durchführen. In dieser Struktur können die Programmteile nach jeder der vorgestellten konzeptionellen Klassen zusammenarbeiten. Der Tuplespace stellt das Bindeglied zwischen Front End und Back End der Applikation dar. Über ihn kommunizieren die Clients des Front Ends mit den Programmteilen des Back Ends.

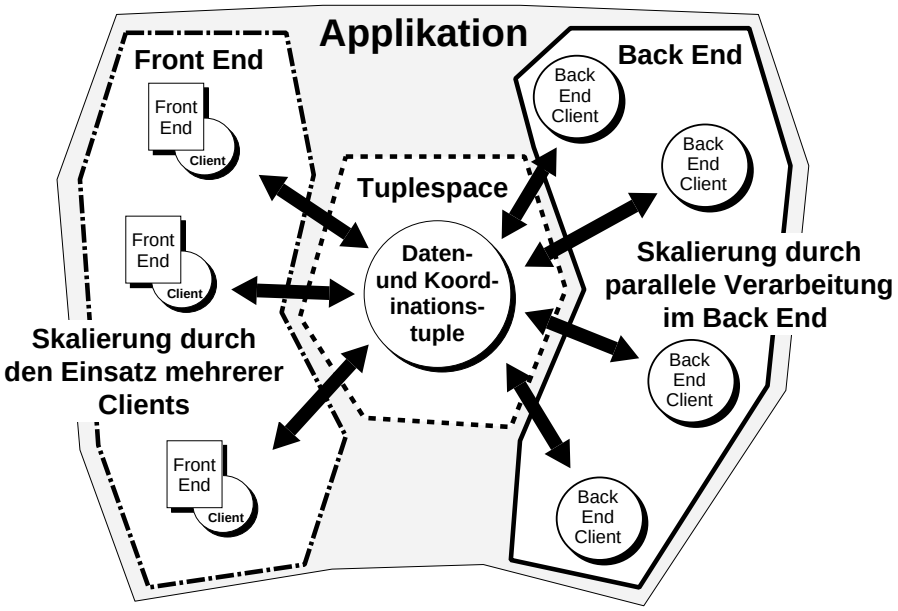

Abbildung 23: Multi-Client-Schema zur serverseitigen Skalierung

Die in Abbildung 23 vorgeschlagene Konfiguration bietet folgende Vorteile:

Möglichkeit zur Skalierbarkeit in zwei Richtungen:

- Es können die Aufgaben zwischen Front End und Back End aufgeteilt werden. Diese Möglichkeit ist bei einer normalen Client-Server Applikation auch gegeben und stellt keine Veränderung dar.
- Im Back End können die dem Server zufallenden Aufgaben nach einem oder mehreren der in Abschnitt 9.1.1 vorgestellten Paradigmen auf verschiedenen Rechnern in verschiedenen Programmiersprachen skaliert werden.

Diese zwei Möglichkeiten ermöglichen eine feine Granularität der Verteilung in der Gesamtapplikation, da die Aufgaben entweder nach der auszuführenden Aufgabe zwischen mehreren Clients oder nach dem Ort der Ausführung zwischen Front End und Back End parallelisiert werden kann.

Erhöhung der Verfügbarkeit:
Durch den Einsatz mehrerer Clients im Back End kann die Verfügbarkeit der Applikation erhöht werden. Bei Ausfall eines Clients kann bei entsprechender Organisation der Arbeitsteilung ein anderer Client die Arbeit übernehmen.

Austauschbarkeit von Komponenten (Plug In):
Einzelne Komponenten einer Anwendung können durch die Koppelung über den Tuplespace flexibel und auch bei laufendem Programm ausgetauscht werden. Es kann ein kompatibler Programmteil ohne Neuübersetzung dem Programm hinzugefügt werden. Dies ermöglicht z.B. die Hinzufügung von Worker-Clients zu einer bereits laufendenden Applikation.

Erweiterbarkeit durch Prototypen:
Einzelne Bereiche der Anwendung können einfach durch das Einbinden eines Protoyps erweitert und getestet werden. Da diese Erweiterungen nicht unbedingt in der gleichen Programmiersprache wie die Gesamtapplikation geschrieben sein müssen, kann ein Abweichen auf Sprachen, die eine schnelle Programmentwicklung unterstützen, zielführend sein.

Im folgenden werden Beispielprogramme für die Arbeitsaufteilung im Back End vorgestellt werden. Diese Programme sind, um den Umfang der Arbeit nicht zu sprengen, kleinerer Natur, doch sind sie für die Verfolgung des jeweilig verwendeten Aufteilungsparadigmas exemplarisch. Als Grundannahme für die Implementation ist davon auszugehen, daß die Dauer für die Bearbeitung der Teilprobleme im Verhältnis zur Kommunikations- und Koordinationsdauer groß ist. Es werden in den Beispielen drei grundsätzliche Wege verfolgt, die in Abbildung 24 dargestellt sind:

Implementation mit homogenen Clients:
Homogene Clients lassen sich überall dort einsetzen, wo die Aufgabe sinnvoll in gleich zu bearbeitende Teile geteilt werden kann. Dies ist bei Verwendung der Paradigmen Result- bzw. eingeschränkt bei Agenda-Parallelism und bei der Perfectly Parallel Decomposition möglich. Hierbei kann die Dauer der Bearbeitung der Gesamtaufgabe durch das Hinzufügen mehrerer Clients proportional verringert werden. Weiters wird die

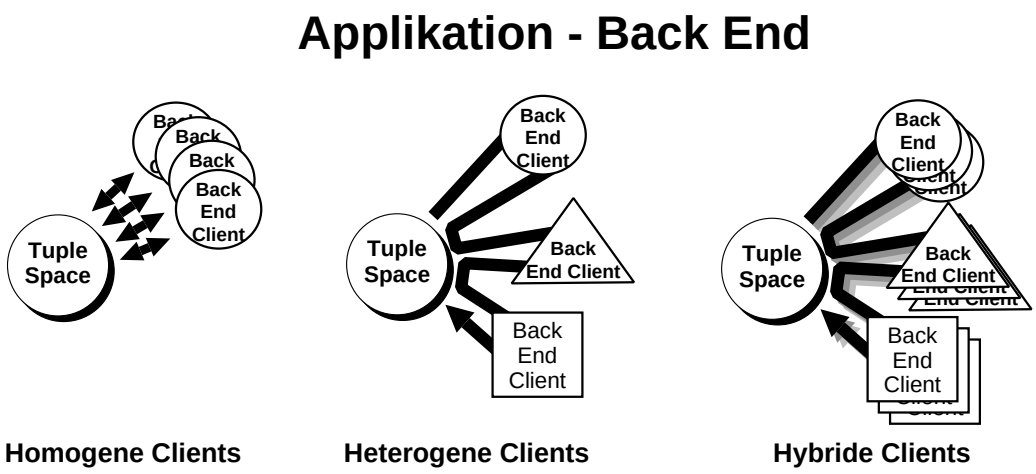

Abbildung 24: Parallelisierungsstrategien im Back End

Verfügbarkeit der Anwendung bezüglich des Ausfalls von einzelnen Clients erhöht.

Implementation mit heterogenen Clients:

Die Implementation mit heterogenen Clients verfolgt das Ziel der Aufteilung der Aufgabe nach funktionalen Kriterien und der Spezialisierung auf Teilaufgaben. Dieser Weg läßt sich bei Verwendung von Specialist-, Master/Worker-Parallelism und Dataflow bzw. Pipelining einsetzen.

Die Verschiedenheit der Clients kann sich nicht nur in der Form der Spezialisierung ausdrücken, sondern auch in den Datenbeständen, auf die sie zugreifen können. Hierdurch läßt sich eine Trennung von sensitiven Datenbeständen erreichen. Der Zugriff auf diese ist dann nur über getrennte Clients möglich.

Implementation mit hybriden Clients:

Dieser Weg stellt eine Kombination der vorher vorgestellten Punkte dar. In der einen Richtung wird das Problem für die Bearbeitung verschiedener Clients gegliedert. Von diesen verschiedenen Clients werden jedoch mehrere zugleich für die Bearbeitung eingesetzt.

Welche Methode für die Problemstellung geeignet ist, hängt von der Art des Problems und der Zielsetzung der Applikation ab. Jede der drei Methoden hat ihre positiven und negativen Auswirkungen auf Protokoll und Koordinationsmechanismus zwischen den einzelnen Clients und innerhalb der Gesamtanwendung.

Bei praktischen Anwendungen kann sich der Entwickler verschiedener objektorientierter Designtechniken bedienen. Eine ohne großen Aufwand schnell durchzuführende Technik ist die Verwendung von CRC-Cards [Beck and Cunningham, 1989]. Bei dieser Technik werden Papierkärtchen im Format A5 verwendet (siehe Abbildung 25).

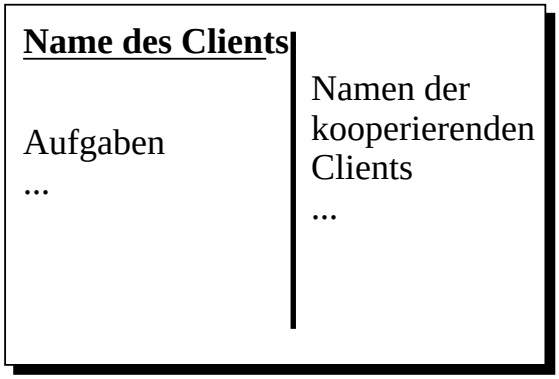

Abbildung 25: CRC-Cards

Jede Karte repräsentiert ein Objekt, in unserem Fall einen der Applikationsclients. Der in [Beck and Cunningham, 1989, vgl. S. 2] ursprünglich vorgesehene Inhalt *ClassName*, *Responsibilities* und *Collaborators* wird zur Verwendung der CRC-Cards für unsere Zecke abgeändert. In der linken oberen Ecke ist der Name des entsprechenden Applikations-Clients angeführt. Auf der linken Seite sind alle Aufgaben vermerkt, die vom Client durchgeführt werden. In diesem Entwicklungsstadium genügt eine verbale Beschreibung der Aufgaben. Auf der rechten Seite werden alle Applikations-Clients vermerkt, mit denen der Client Daten tauscht und koordinieren muß.

CRC-Cards stellen in diesem Stadium eine sehr effiziente Methode dar, für kleinere Projekte zu einem brauchbaren Ergebnis zu kommen. Die Darstellung durch CRC-Cards ist ausreichend, um als Diskussionsgrundlage zu dienen und um die nächsten Schritte darauf aufzubauen.

9.1.3 Entwurf des Protokolls zwischen den Applikationsteilen

In diesem Entwicklungsschritt wird das Protokoll zwischen den Applikationsteilen entworfen und in Form von Linda-Operationen dargestellt. Aus der Darstel-

lung der Aufgabenverteilung und der Kooperation auf den CRC-Cards können wir für diesen Schritt folgende Dinge ableiten:

- Benötigte Daten im Applikationsclient.
- Welche Daten mit den anderen Clients ausgetauscht werden.
- Welche Ein- und Ausgaben der Client mit dem Benutzer oder Betriebssystem durchführt.

Nicht klar ersichtlich sind die Abhängigkeiten in den Kommunikationsbeziehungen zwischen den einzelnen Applikationsclients. Diese werden in diesem Schritt analysiert. Die Linda-Operationen stellen hierbei das Werkzeug zur Darstellung dar. Die Entwicklung des Protokolls kann in mehreren Schritten erfolgen:

1. *Festlegen der generellen Repräsentation der Applikation im Tuplespace:* Da der Tuplespace von mehreren Applikationen gleichzeitig genutzt werden kann, sollten sich die Tuple der verschiedenen Applikationen voneinander unterscheiden. Eine einfache Methode dies zu erreichen, ist die Kennzeichnung der Applikation im ersten Element des Tuple z.B. `(APP1,...,...)` für alle Tuple der Applikation1.
2. *Bilden von Tuple aus den, zwischen den Clients auszutauschenden Daten:* In diesem Schritt werden aus den von den Clients auszutauschenden Daten, unter Berücksichtigung der im vorigen Schritt festgelegten Repräsentation, Tuple geformt. Es sollte versucht werden, möglichst wenig verschiedenartige Tuple zu bilden.
3. *Definieren von Linda-Operationen zur Abfrage der gebildeten Tuple:* Damit die verschiedenen Clients auf die im Tuplespace enthaltenen Tuple zugreifen können, sind die für die Abfragen benötigten Pattern, in diesem Schritt festzulegen. Dabei ist zu beachten, daß durch die Pattern möglichst nur die benötigten Daten zurückgeliefert bzw. aus dem Tuplespace entfernt werden sollten. In dieser Phase fällt auch die Entscheidung darüber, wann ein Client bei der Abfrage an den Tuplespace blockiert. Als Hilfsmittel in diesem Arbeitsschritt können Zustandsdiagramme und Übergangstabellen verwendet werden. Diese Erleichtern auch die Handsimulation und das Prüfen auf Deadlock-Freiheit.

4. *Prüfen des Protokolls auf Deadlocks:* In diesem Arbeitsschritt wird das Protokoll auf das Auftreten von Deadlocks überprüft. Ein Deadlock tritt dann auf, wenn ein Client auf ein bestimmtes Tuple wartet, das aber durch den Zustand der Gesamtapplikation nicht in den Tuplespace gelangen kann. Eine Handsimulation wird in den meisten Fällen ein geeignetes Mittel zur Überprüfung sein. Bei Auftreten eines Deadlocks muß zum vorigen Schritt zurückgegangen werden und das Protokoll noch einmal überarbeitet werden.

9.1.4 Festlegen des Starts und des Programmendes

Zwei besondere Zustände in einer Linda-Applikation sind der Programmstart und das Programmende. Bei seinem Start muß der Client feststellen können, ob die Gesamtapplikation bereits läuft oder ob er der erste Client ist. Dieses Problem wurde in den meisten Applikationen entweder durch einen speziellen Initialisierungs-Client oder durch ein Starttuple gelöst.

Bei Verwendung eines Initialisierungs-Clients initialisiert ein spezieller Client die ganze Applikation. Wenn vor der Initialisierung ein anderer Client bereits aktiv ist, blockiert er so lange, bis die Initialisierung der Applikation beendet ist und nimmt dann seine Arbeit auf. Bei der Verwendung eines Starttuple kann jeder Client die Applikation initialisieren. Er prüft bei seinem Start die Existenz eines speziellen Tuple im Tuplespace und stellt daran fest, ob die Gesamtapplikation bereits läuft. Diese Vorgangsweise wurde bereits beim `Count`-Client in Abschnitt 7.3 beschrieben.

Das Feststellen des Programmendes ist ein zweistufiger Vorgang. Bei Serverprogrammen, die durchgehend verfügbar sein müssen, kann die Feststellung des Programmendes entfallen. Bei allen anderen Anwendungen müssen alle an der Gesamtanwendung beteiligten Clients zunächst einmal feststellen, ob ihre Aufgabe bereits erfüllt ist. Ist die Aufgabe des Clients beendet so kann der entsprechende Client seine Verbindung zum Tuplespace abbrechen und beenden. Da aber die verbleibenden Datentuple, zumindest aber das Starttuple der Anwendung aus dem Tuplespace entfernt werden sollen, muß einer der Clients diese Aufgabe übernehmen. Bei Verwendung eines Initialisierungsclients kann diesem die Aufgabe übertragen werden. Wird dieser nicht verwendet so muß jeder Client, bevor er seine Arbeit beendet feststellen, ob er der letzte Client ist. Ist er der letzte Client, so kann er alle verbleibenden Tuple der Applikation aus dem Tuplespace entfernen.

Da diese Probleme nicht für alle Applikationen generell lösbar sind, kann an dieser Stelle keine Lösung angeboten werden. Das Start- und Endproblem muß daher für jede Applikation individuell gelöst werden.

9.2 Beispiele

In diesem Abschnitt wird der im vorigen Abschnitt dargestellte Entwicklungsprozeß am Beispiel des *Ping-Pong*-Clients erläutert. Danach wird je eine Linda-Beispielapplikation für die konzeptionellen Klassen beschrieben.

9.2.1 Der Ping-Pong Client

Der Ping-Pong-Client ist Beispiel für die Koordination zwischen mehreren homogenen Clients. Die Clients spielen miteinander Tischtennis, wobei ein Client *Ping* spielt und ein anderer Client mit *Pong* antwortet. Im folgenden werden die Schritte der Programmentwicklung bis hin zur Implementierung kurz dargestellt um die Entwicklung eines Linda-Programmes zu veranschaulichen.

Wahl der konzeptionellen Klasse:
Da das Beispiel einen Grenzfall darstellt, kann es keiner der Klassen wirklich zugeordnet werden. Dieser Punkt entfällt hier.

Festlegen der Aufgabenstruktur:
Bei der Verteilung der Aufgaben fällt die Entscheidung auf einen homogenen Client, der sowohl die *Ping* $\rightarrow$ *Pong*-Umwandlung als auch die *Pong* $\rightarrow$ *Ping*-Umwandlung durchführen kann. Der Client sollte das Spiel auch mit sich selbst spielen können. Die CRC-Card für den Client ist in Abbildung 26 dargestellt.

Definieren und Testen des Protokolls:
Die Tuple der Applikation werden durch das erste Element identifiziert. Es ist bei allen zur Applikation gehörenden Tuple das Element BALL. Aus den in Abbildung 26 dargestellten Aufgaben ergeben sich folgende Daten:

- Der Client muß wissen, ob er zuletzt *Ping* oder *Pong* empfangen hat.
- Ein von einem anderen Client empfangenes *Ping* muß in ein *Pong*, ein empfangenes *Pong* muß in ein *Ping* umgewandelt werden.

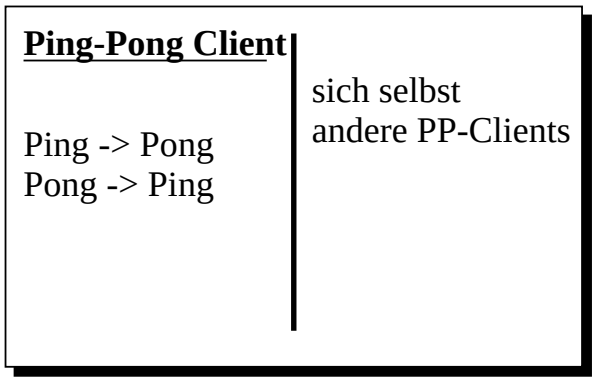

Abbildung 26: CRC-Card für den Ping-Pong Client

Die verwendeten Datentuple setzen sich daher aus dem Übermittelten Zustand und aus `BALL` als erstes Element zusammen. Es können zwischen den Clients daher nur die Tuple `(BALL,ping)` und `(BALL,pong)` übermittelt werden. Die Hauptfunktion des Clients ist in Abbildung 27 als Struktogramm dargestellt.

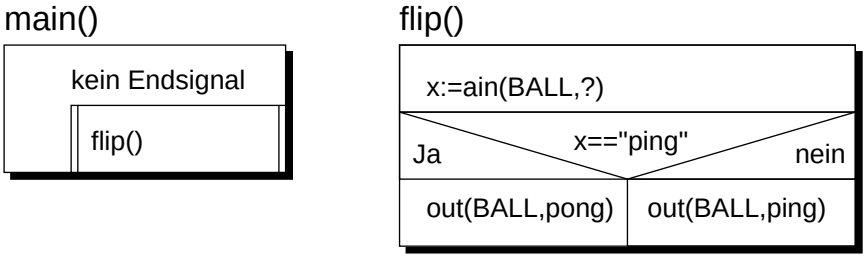

Abbildung 27: Struktogramm für die Hauptfunktion des Ping-Pong Clients

Der Algorithmus ermöglicht die Interaktion von einem oder mehreren Clients. Ist nur ein Client vorhanden, so kooperiert der Client mit sich selbst. Es können in das Spiel der Clients auch mehrere Bälle gebracht werden. Bei einer geraden Anzahl von Clients bearbeiten die Clients entweder nur `Ping` oder nur `Pong`. Nimmt eine ungerade Anzahl von Clients am Spiel teil, so werden von jedem Client `Ping` und `Pong` abwechselnd bearbeitet. Nimmt man alle Bälle, z.B. mittels `ain(Ball,?)` aus dem Spiel, blockieren alle Clients. Sobald wieder ein Ball im Spiel ist, beginnen alle wieder zu spielen.

Feststellen der Deadlock-Freiheit:

Die Deadlock-Freiheit ist grundsätzlich dann gewährleistet, wenn zu einem bestimmten Zeitpunkt, jeder Zustand in den die Applikation kommen kann, von zumindest einem Client weitergeführt werden kann. Tabelle 21 zeigt die Zuständsübergänge in der Applikation mittels deren Repräsentation im Tuplespace.

Status im Tuplespace	Client führt `flip()` aus
leer	leer
`(BALL,ping)`	`(BALL,pong)`
`(BALL,pong)`	`(BALL,ping)`

Tabelle 21: Zustandsübergangstabelle für die Ping-Pong Applikation

Im Tuplespace können drei verschiedene Zustände, `leer` (kein `BALL`-Tuple vorhanden), `(BALL,ping` und `BALL,pong`, auftreten. Beim Auftreten des Zustandes `leer` blockieren alle Clients, da sie keine Tuple aus dem Tuplespace entnehmen können. Durch das Ausführen der Funktion `flip()` (siehe Abbildung 27) durch einen Client wird der gerade im Tuplespace vorhandene Zustand in den jeweils anderen Zustand umgewandelt.

Wenn ein `BALL`-Tuple im Tuplespace vorhanden ist und die Clients absturzsicher arbeiten kann bei der Applikation kein Deadlock auftreten, da jeder Client bei jedem der zwei Zustände, *Ping* und *Pong*, durch die Funktion `flip()` die Umwandlung in den jeweils anderen Zustand durchführen kann.

Implementation:

Für die Implementation des Clients in Perl muß noch das Startproblem gelöst werden. Ähnlich wie beim Count-Client wird das Startproblem durch das Starttuple `(BALL,PINGSTART)` gelöst. Abbildung 28 zeigt den Perl-Sourcecode des Ping-Pong Clients. Wird das Austauschen des Balltuple mittels eines `uin()`/`uout()`-Konstruktes durchgeführt, ist die Anwendung auch gegen den Ausfall eines oder mehrerer Clients gesichert. Abbildung 28 zeigt den Perl-Linda-Source des Ping-Pong Client.

Die Beendigung aller Clients wird durch das Tuple `(BALL,PINGEND)` herbeigeführt. Jeder Client überprüft bei jedem Durchlauf der `while`-Schleife, ob das Tuple im Tuplespace enthalten ist. Wird es durch die Funktion `nbrd()` gelesen, so beendet er seine Ausführung.

```
#!/usr/local/bin/perl
# Ping-Pong Client (c) 1996 W.Schoenfeldinger

require "linda-cli.pl";

&register_client("wallace.wu-wien.ac.at",7999);

@tl=&nbrd("BALL","PINGSTART"); # Are we the first one ?

# Start des Client
&out("PINGSTART"), &out("BALL","ping") if @tl==();
        # We are the first

while(!&nbrd("BALL","PINGEND"))
{
        @tl=&ain("BALL","?");
        $what=&element(1,2,@tl);
        print "Got: $what\n";
        if($what eq "ping")
        { &out("BALL","pong"); }
        else
        { &out("BALL","ping"); }
}
```

Abbildung 28: Source-Code des Ping-Pong Client

9.2.2 Result Parallelism: DiscTool

Als Beispiel für eine Applikation, die nach dem Paradigma des Result Parallelism implementiert ist, wird im folgenden *DiscTool* [Schönfeldinger, 1995] vorgestellt, ein Programm, das die Erfassung und Visualisierung der Festplattenauslastung in einem Cluster von Workstations ermöglicht.

Das zu erreichende Ergebnis, die Darstellung der Cluster-weiten Plattenauslastung kann nur erreicht werden, wenn jeder Client die Plattenauslastung auf *seinem* Rechner in den Tuplespace stellt. Die Verteilungsstruktur von *DiscTool* ist in Abbildung 29 dargestellt.

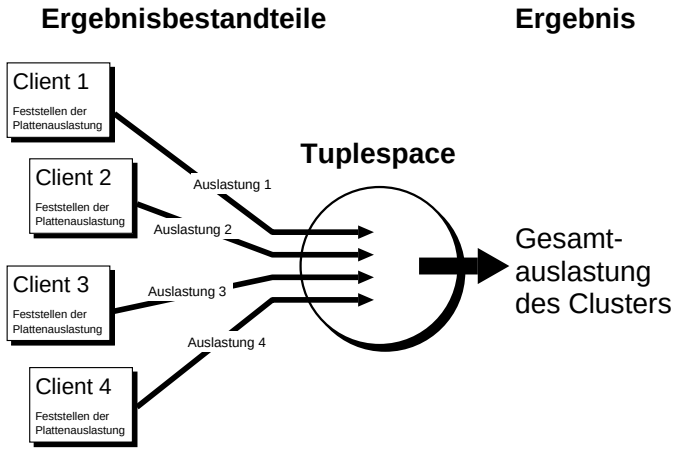

Abbildung 29: Verteilungsstruktur von DiscTool

Die einzelnen Clients stellen Informationen über die Partitionen, die Auslastung und den verfügbaren Speicherplatz in regelmäßigen Intervallen in den Tuplespace. Die Clients sind homogen und führen nur diese Aufgabe durch. Die Gesamtheit der Ergebnisse kann dann mittels eines weiteren Clients über ein World Wide Web-Interface abgefragt werden. Die Applikation wird durch einen Initialisierungsclient gestartet, der anhand der im Tuplespace bereits vorhandenen Daten feststellt, auf welchen Rechnern ein Client läuft und auf den fehlenden Rechnern den Client mittels RPC startet. Die Protokolle und die Implementierung dieser Applikation sind in Abschnitt 10.8.2 ausführlicher beschrieben.

9.2.3 Specialist Parallelism: Bankomat-Simulation

Die Bankomat-Simulation ist ein Beispiel für die Zusammenarbeit von mehreren Clients zur Erfüllung einer Gesamtaufgabe. Es soll uns hier als Beispiel für eine stark vereinfachte Applikation dienen, die nach dem Paradigma des Specialist Parallelism aufgeliedert und implementiert wurde. Die Aufgabe der Anwendung ist die Simulation eines Bankomat-Systems mit verteilter Information. Vier Clients, die jeweils nur einen Teil der Gesamtinformation besitzen, arbeiten hierbei in der Applikation zusammen.

Da die Implementierung nur als Demonstration für die Zusammenarbeit von mehreren Clients nach dem Paradigma des Specialist Parallelism dient, ist nur ein kleines Subset der Funktionalität eines richtigen Bankomatsystems implementiert. Es erfolgt keine Verwaltung des Kontostandes, kein Logging der Zugriffe und keine Ausgabe der Kontobewegungen. Die CRC-Cards für diese Anwendung sind in Abbildung 30 dargestellt.

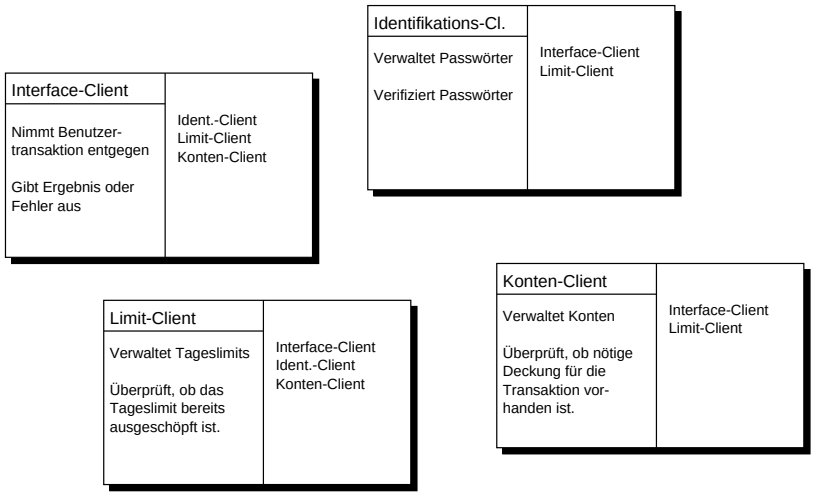

Abbildung 30: CRC-Cards für die Bankomat-Simulation

Die Anwendung teilt sich in vier Clients, von denen jeder verschiedene Funktionen ausübt. Die Clients haben folgende Aufgaben:

Interface-Client:

Dieser Client stellt das textbasierte Front End zum Benutzer dar. Über diesen Client kann der Benutzer mit der Applikation interagieren.

Identifikations-Client:

Die Benutzernamen und die dazugehörigen Passwörter werden vom Identifikations-Client verwaltet. Er verifiziert das vom Benutzer eingegebene Passwort.

Limit-Client:

Dieser Client speichert die täglichen Behebungslimits für die Benutzer. Er stellt fest, ob der vom Benutzer gewünschte Betrag noch innerhalb des zulässigen täglichen Limit liegt. Nach der Abbuchung durch den Konten-Client, bringt der Limit-Client das Limit auf den neuesten Stand.

Konten-Client:

Der Konten-Client speichert und verwaltet den aktuellen Kontostand der Benutzer.

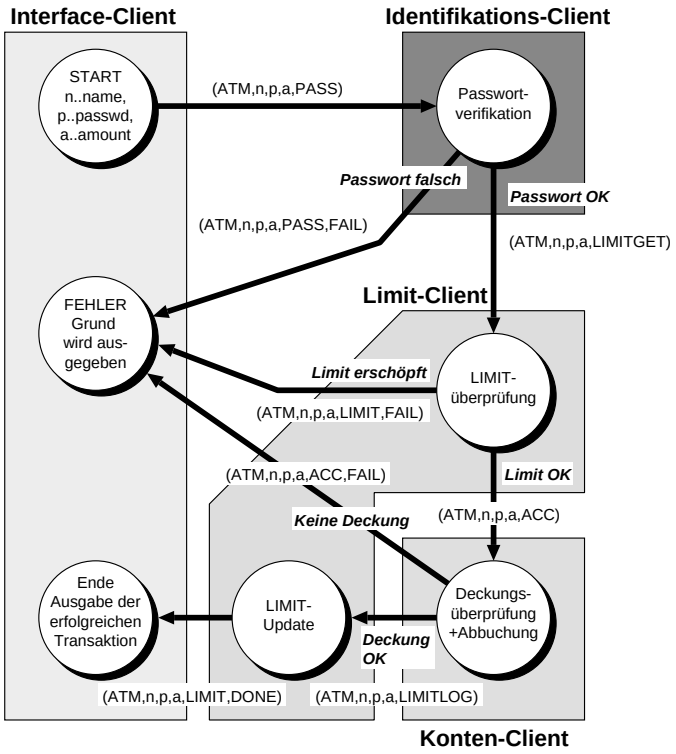

Abbildung 31: Zustandsdiagramm für die Bankomat-Simulation

Abbildung 31 zeigt das Zustandsdiagramm für die Applikation. Es sind in das Zustandsdiagramm zusätzlich die im Protokoll verwendeten Tuple und die von

den Clients ausgeführen Aktionen eingezeichnet. Die Tuple, die zwischen den Clients ausgetauscht werden, haben folgende Form:

(`ATM`,*Name*,*Passwort*,*Betrag*,*Operation*)

Die Operation gibt den nächsten Client in der Bearbeitungsreihe an und kann die Werte `PASS` (Identifikations-Client), `LIMITGET, LIMITLOG` (Limit-Client) und `ACC` (Konten-Client) annehmen. Um zu verhindern, daß der Interface-Client ein Tuple vor Ende der Bearbeitung aus dem Tuplespace entfernt, haben die zu einem Endzustand führenden Tuple ein zusätzliches Element, das die Werte `DONE` (erfolgreiche Beendigung) und `FAIL` (Fehler) annehmen kann. Durch die unterschiedliche Elementzahl, können die den Interface-Client betreffenden Tuple von allen anderen Tuple unterschieden werden.

Die Applikation kann mehrere Anfragen simultan verarbeiten, da die Tuple der einzelnen Benutzer durch den *Namen* unterschieden sind. Es kann jeder Benutzer jeweils nur eine Transaktion zur gleichen Zeit durchführen. Dies entspricht der Funktionalität eines Bankomat in der Realität. Durch die Trennung der Funktionen `Identifikation`, `Limitüberprüfung` und `Deckungsüberprüfung` stellt die Simulation ein anschauliches Beispiel für dezentrale Bearbeitung von Transaktionen nach dem Paradigma des Specialist Parallelism dar. Der vollständige Source-Code für alle Clients in Perl ist in Anhang C enthalten.

9.2.4 Agenda Parallelism: Parallel Povray (ParPov)

Povray ist ein Raytracing-Programm, das frei auf dem Internet verfügbar ist. Das Programm akzeptiert Bildbeschreibungen in einer eigenen Definitionssprache und stellt diese Beschreibungen als Bilder in einer gewählten Auflösung dar. Der Prozeß der Bilderzeugung dauert selbst auf schnellen Workstations geraume Zeit, so dauert die Erzeugung eines Bildes mit einer Auflösung von 1000x750 Punkten auf einer DEC 4000/300 Alpha-Workstation ca. 5 Minuten. Bei Serien von Bildern summiert sich die Bearbeitungzeit bei sequentieller Bearbeitung auf. Verteilung der Bildbearbeitung auf alle im Netzwerk verfügbaren Rechner bietet sich zur Verkürzung der Gesamtdurchlaufzeit an.

Die Problemstellung der Verteilung der Bildbearbeitung entspricht dem Agenda Parallelismus, die einzelnen zu bearbeitenden Beschreibungsdateien für die Bilder sind in Form einer *Agenda* im Tuplespace vorhanden. Als Einschränkung enthält die Agenda aber nur einen Typ von Auftrag. Die beteiligten Clients holen sich den

jeweils nächsten Job aus dem Tuplespace und bearbeiten diesen. Die über den Job im Tuplespace vorhandenen Daten sind die `JobID`, der Ort der Beschreibungsdatei und der Ort, an dem das Ergebnis in Form der fertigen Bilddatei liegt. Weiters bekommt der Job noch eine Statusangabe, die seinen augenblicklichen Zustand (`todo`, `processed`, `done`) angibt.

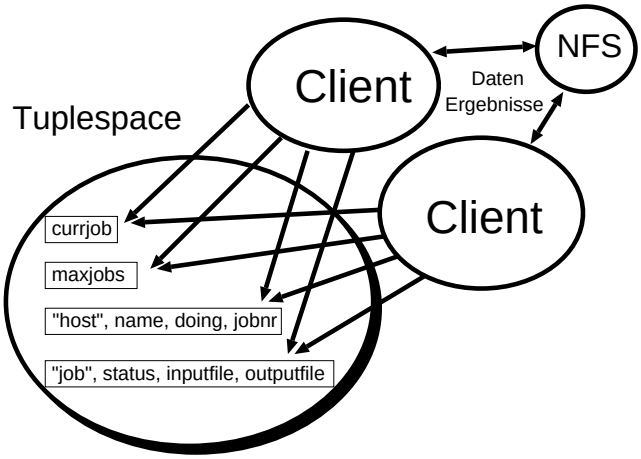

Abbildung 32: Applikationsschema von ParPov

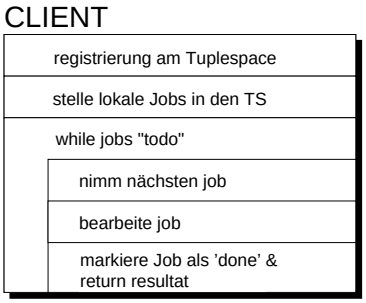

Abbildung 33: Algorithmus des Clients in ParPov

Abbildung 32 zeigt das Verteilungsschema von ParPov. Die Clients greifen auf den Tuplespace zu und stellen anhand der enthaltenen Information die `JobID` des nächsten zu bearbeitenden Job fest. Dieser wird dann aus dem Tuplespace entnommen und bearbeitet. Die Gesamtzahl der im Tuplespace enhaltenen Jobs ist ebenfalls in einem Tuple dargestellt. Erreicht die `ID` des nächsten zu bearbeitenden Jobs die Gesamtzahl ist dies das Signal für die Clients, daß die Gesamtaufgabe beendet ist. Der Algorithmus für den einzelnen Client ist in Abbildung

33 ersichtlich. Zuerst stellt der Client alle lokal vorhandenen Jobs in den Tuplespace. Dann erst widmet er sich der Bearbeitung der allgemeinen Jobs. Der Perl Source-Code zu ParPov ist im Anhang D enthalten.

Die Granularität hängt von der Art der zu bearbeiteten Jobs ab. Da die einzelnen Jobs die kleinste zu bearbeitbare Einheit darstellen, können diese in dieser Implementation nicht weiter aufgeteilt werden. Es wird angenommen, daß auf jedem Rechner nur ein Client läuft. Dies ist deshalb sinnvoll, da ein laufender PovRay-Prozeß den Rechner bereits auslastet. Zwei oder mehrere Prozesse auf einem Rechner würden sich die vorhandene Rechenzeit teilen müssen. Je nach Rechnergeschwindigkeit können in dieser Applikation die Rechner einen entsprechenden Teil der Jobs bearbeiten und so zur Verkürzung der Gesamtdurchlaufzeit beitragen.

10 Fallstudie: Linda meets WWW

In der folgenden Fallstudie wird gezeigt, wie durch die Kombination von World Wide Web (WWW) und Linda neue Nutzungsmöglichkeiten wie globale, transaktionsorientierte Systeme geschaffen werden können. Nach einer Einführung in die Problematik und die Grenzen des Systems erfolgt eine Darstellung der Änderung der Applikationsstruktur durch Linda und die sich daraus ergebenden Erweiterungen. Abgeschlossen wird dieser Teil mit Beispielapplikationen, die Anwendungen von Perl-Linda mit WWW darstellen und in dieser Form mit den sonst zu Verfügung stehenden Techniken nicht bzw. oder nur sehr umständlich zu realisieren gewesen wären. Das vorgestellte System wurde als Beitrag [Schönfeldinger, 1995] auf der *International WWW-Conference* im Dezember 1995 in Boston präsentiert.

10.1 Einleitung

Das World Wide Web (WWW) hat sich in den letzten Jahren zum dominierenden Dienst auf dem Internet entwickelt. Die Möglichkeit, Hypertext, Grafiken, Filme und Ton über das Internet darzustellen hat es bis dahin nur in einigen high-level *Computer Supported Cooperative Work*-Systemen gegeben. Diese waren zumeist an eine spezielle Hard- und Softwareausstattung gebunden und für den *normalen* Benutzer nicht zugänglich. Auch boten diverse Online-Dienste diese Möglichkeiten, jedoch nur mit der für den Dienst speziell geschaffenen Software.

Durch das WWW wurde ein Quasi-Standard geschaffen, der es heute jedem Teilnehmer am Internet ermöglicht, oben genannte Datentypen über das Netzwerk zu übertragen und zu verarbeiten. Die dafür benötigte Software ist *public domain*, d.h., frei über das Internet herunterzuladen und auf fast allen Plattformen installierbar und lauffähig. Die Einfachheit in der Bedienung und die einheitliche, systemunabhängige Oberfläche trugen entscheidend zur Verbreitung dieses Dienstes auf dem Internet bei. Es beschränkten sich die Fähigkeiten von WWW jedoch nicht nur auf die reine Darstellung von Information. Es wurde als Erweiterung der Grundfunktionalität eine standardisierte Schnittstelle zum Betriebssystem geschaffen, durch die auch Programme eingebunden und ausgeführt werden können.

Im weiteren wird die Struktur von WWW genauer beschrieben. Es wird insbesondere auf die Verwendung von WWW als Front End für Applikationen eingegangen. Die dabei auftretenden Probleme und Grenzen für Applikationen werden

aufgezeigt, und es werden einige Ansätze zur Lösung vorgestellt. Danach wird die Einbindung von Linda und die sich dadurch verändernde Programmstruktur gezeigt. Die sich dadurch ergebenden Möglichkeiten werden anhand von Beispielapplikationen demonstriert.

10.2 Struktur von WWW

Das WWW basiert auf dem Client-Server Prinzip. Die Information wird auf einem Server, dem Web-Server, zur Verfügung gestellt. Der Client hat die Aufgabe, die Information aufgrund eines *Uniform Resource Locator* (URL) zu finden, vom Server herunterzuladen und sie darzustellen. Der Client kommuniziert mit dem Server über das *HyperText Transfer Protocol* (HTTP) [Berners-Lee *et al.*, 1995]. Als Darstellungssprache wird *HyperText Markup Language* (HTML) [Berners-Lee and Connolly, 1995] verwendet. Neben der Darstellung von Information ermöglicht das *Common Gateway Interface* (CGI) [Robinson, 1996] den Datentausch zwischen Betriebsystem und WWW-Server. Diese Schnittstelle ermöglicht auch die Anbindung von Programmen, die von WWW-Seiten aufgerufen werden können und ihre Ausgaben wiederum im WWW darstellen. Die zum WWW gehörenden Protokolle und Anbindungen sind in Abbildung 34 dargestellt.

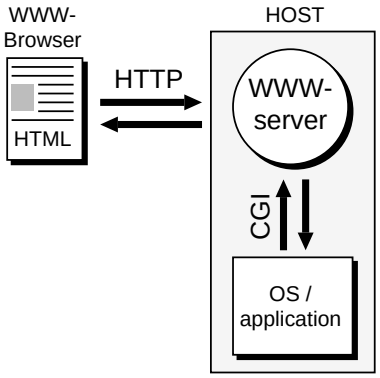

Abbildung 34: Überblick über ein WWW System

Sowohl auf Client als auch auf Serverebene gibt es verschiedene Produkte, welche auf diversen Spezialgebieten, wie z.B. Multi-Media-Anbindung, Caching, Security und Geschwindigkeit besser implementiert sind als andere. Dennoch bieten die Protokolle HTML, HTTP und CGI den größten gemeinsamen Teiler aller Imple-

mentationen. Die im Anschluß vorgestellten Eigenschaften und Lösungsansätze sind mit allen Systemen implementierbar und kompatibel.

Als Front End für Applikationen ist WWW aus folgenden Gründen interessant-[Schönfeldinger, 1995, vgl. S. 1]:

- Ein WWW-basiertes Interface erlaubt internet-weiten, somit globalen Zugriff auf Applikationen.
- Der Einsatz von WWW ermöglicht die Entwicklung eines systemunabhängigen Benutzerinterface für Anwendungen. Dieser Vorteil ergibt sich aus der Verfügbarkeit von WWW-Browsern auf den meisten verbreiteten Betriebssystemen.
- Text, Grafiken, Ton, Filme und Formulare können einfach über ein standardisiertes Front End dargestellt werden.
- Das Common Gateway Interface ermöglicht die Anbindung von Programmen, sogenannten CGI-Skripts, an WWW-Seiten. Durch diese Funktionalität werden dynamisch erstellte WWW-Seiten ermöglicht.
- Durch spezielle Konstrukte (Tags), wie z.B. den `FORM`-Tag, ist eine einfache Datenübertragung aus WWW-Formularen möglich.
- WWW-Seiten können direkt vom WWW-Browser ausgedruckt werden. Weiters können sie lokal als HTML-Source gespeichert werden.
- Zur Erstellung von CGI-Skripts sind oft benutzte Ausgabeelemente und Übertragungsfunktionen in WWW-Erweiterungsbibliotheken verschiedener Programmiersprachen, wie z.B. C, C^{++}, Perl, Python u.a., bereits implementiert.

Alle diese Gründe sprechen für eine Nutzung der WWW-Technologie als Benutzerschnittstelle für Informationsysteme. Es existieren bereits fertige Anbindungen an Datenbanksysteme wie ORACLE, WAIS u.a., jedoch gibt es durch das CGI-Protokoll und die Seitenorientiertheit von WWW gewisse Grenzen für die Implementation von Dialog-Applikationen. Im nächsten Abschnitt werden nach einer genaueren Darstellung des CGI-Protokolls die Grenzen desselben für Dialog und transaktionsbasierte Applikationen aufgezeigt.

10.3 Das Common Gateway Interface

Das CGI-Protokoll ermöglicht die Datenübertragung des WWW-Servers an das Betriebsystem bzw. an ein Programm, das vom WWW-Server gestartet wird. Dieses Programm wird im weiteren als CGI-Skript bezeichnet. Am häufigsten werden CGI-Skripts in Verbindung mit Formularen verwendet. Man kann grundsätzlich bei der Art der Übertragung zwei Verfahren unterscheiden, die im CGI-Protokoll implementiert sind (vgl. dazu [NCSA, 1994]):

GET:
Diese Methode wird mit dem Eintrag `"METHOD=GET"` im `FORM`-Tag gewählt. Bei dieser Datenübermittlungsmethode werden die im Formular eingegebenen bzw. ausgewählten Daten als Zeichenkette in der Umgebungsvariablen `QUERY_STRING` übermittelt. Diese enthält die Daten in der durch das CGI-Protokoll bestimmten Form. Der Nachteil von `GET` ist die Sichtbarkeit der übermittelten Daten in der URL des aufzurufenden Programmes.

POST:
`POST` ist die für die Datenübermittlung an das Programm besser geeignete Methode. Sie kann nur bei Verwendung des `FORM`-Tag benutzt werden und wird durch Angabe von `METHOD="POST"` gewählt. Die Daten werden dem aufgerufenen Programm im Standard-Eingabestrom `STDIN` übermittelt. Die Länge der übermittelten Daten wird in der Umgebungsvariablen `CONTENT_LENGTH` abgespeichert. Dies ermöglicht ein Einlesen und Bearbeiten der Daten durch Standard-Eingabefunktionen in der jeweilig verwendeten Programmiersprache. Der Vorteil in der Verwendung von `POST` liegt im Schutz der Daten beim Aufruf des Programmes. Ein Nachteil von `POST` beim praktischen Einsatz sind Timeout-Probleme mit manchen Server-Betriebssystemkombinationen.

Im CGI-Protokoll sind die Daten in einem speziellen Format codiert, welches die Übermittlung von Sonderzeichen wie die Gesamtübermittlung der Daten in einer Zeichenkette ohne Leerzeichen ermöglicht. Eine ausführliche Beschreibung dieses Formats befindet sich in [Klute, 1994, S. 146].

Das primäre Einsatzgebiet von CGI-Skripts ist das Ausführen einer benutzerdefinierten Aufgabe und die Ausgabe der Ergebnisse in einer WWW-Seite. Die Aufgaben von CGI-Skripts können ein weites Spektrum abdecken, von dem die

einfachste Form die reine Ausgabe von Daten ist. Grundsätzlich kann man drei Hauptelemente in jedem CGI-Skript finden:

- Lesen und Extrahieren der mittels CGI übermittelten Daten und Konvertieren dieser in ein für die jeweilige Programmiersprache passendes Format für die weitere Bearbeitung.
- Be- und Verarbeitung der Daten und Erstellen der Resultate. Dies ist der Hauptteil jedes CGI-Skripts.
- Umwandlung der Resultate in HTML-Format und Senden an den Standard-Ausgabekanal `STDOUT`. Dieser wird dann vom WWW-Server weiter bearbeitet.

Das Resultat eines CGI-Skripts ist die Ausgabe, welche es an den Standard-Ausgabekanal schickt. Sie wird durch den WWW-Server bearbeitet und das Ergebnis an den Browser geschickt. Das CGI-Skript kann entweder den Inhalt oder den Ort einer Seite zurückliefern:

Inhalt (Content):

In diesem Fall liefert das Skript die darzustellenden Ergebnisse als *Content*, d.h. in HTML-Format zurück. Der HTML-Source wird direkt an den Browser gesendet und von diesem interpretiert und dargestellt. Liefert ein Programm Content zurück, so sollte die Ausgabe mit der Zeile

```
Content-type: text/html
```

begonnen werden. Dies zeigt dem Server an, worum es sich bei den folgenden Daten handelt. Das Content-type Statement gliedert sich in Typ und Subtyp. In unserem Beispiel ist der Typ `text` und der Subtyp `html`.

Ort (Location):

Bei dieser Form wird ein `URL` als Ausgabe des Scripts zurückgegeben. Dieser wird dann vom Server interpretiert, wenn notwendig geladen und an den Client gesandt. Eine Location wird durch die Zeile

```
Location: http://someserver/somepage.html
```

angegeben. Die durch die `URL` benannte Seite wird dargestellt. Man findet diese Form der Rückgabe z.B. bei *Clickable Maps*.

Das CGI-Skript wird als Subprozeß des WWW-Server ausgeführt. Es hat daher die gleichen Zugriffsrechte wie der Benutzer, der den Server ausführt. Im Normalfall ist dies auf UNIX-Systemen der Benutzer `nobody` oder ein speziell für den Server eingerichteter Benutzer, z.B. `www`. Die CGI-Programme müssen, damit sie der Server ausführen kann, in einem speziellen Bereich des Dateisystems liegen, auf das der Server zugreifen darf. Der Bereich kann bei der Einrichtung konfiguriert werden.

10.4 Probleme und Beschränkungen des CGI

Wie schon in [Perrochon and Fischer, 1995] erwähnt, sind WWW-Server sogenannte *Stateless Server* und daher grundsätzlich nicht geeignet, Information über Zustände von Applikationen darzustellen. Es können mit dem HTTP-Protokoll in Verbindung mit CGI ohne spezielle Maßnahmen daher nur *Reply-Response* Applikationen implementiert werden.

Bei den meisten WWW-basierten Applikationen, die State-Information benötigen, wird daher einer der beiden *Workarounds* gewählt:

- Speicherung der State-Information in `HIDDEN`-Tags der WWW-Seiten. Diese Methode ist in Abbildung 35 dargestellt.
- Speicherung der State-Information in Dateien auf dem Dateisystem des Server-Hosts. Abbildung 36 zeigt das Schema dieser Methode.

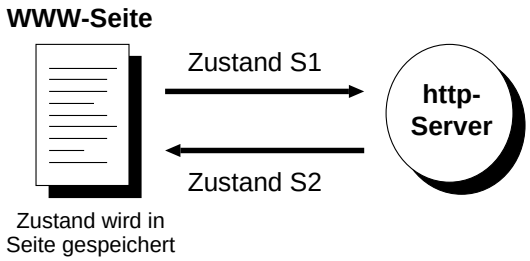

Abbildung 35: Speicherung der State-Information in `HIDDEN`-Tags

Beide Methoden, so gut sie auch funktionieren, haben einige Nachteile. Bei der Benutzung von `HIDDEN`-Tags steht die Information in Klartext im HTML-Source.

Sie wird zwar nicht direkt auf der Seite dargestellt, kann aber jederzeit mit der Funktion zum Anzeigen des HTML-Source angesehen werden. Weiters kann die Information auch gespeichert werden. Es kann daher ein bestimmter Zustand durch Aufruf der gespeicherten Seite immer wieder hergestellt werden. Werden z.B. die Spielparameter in einem WWW-basierten Netzwerkspiel direkt in der Seite gespeichert, so kann einer der Teilnehmer eine Seite mit einem günstigen Spielstand abspeichern. In einem anderem Spiel kann er nun die gespeicherte Version aufrufen um sich gegenüber seinen Mitspielern Vorteile zu verschaffen. Dies liegt selten in der Absicht des Programmierers.

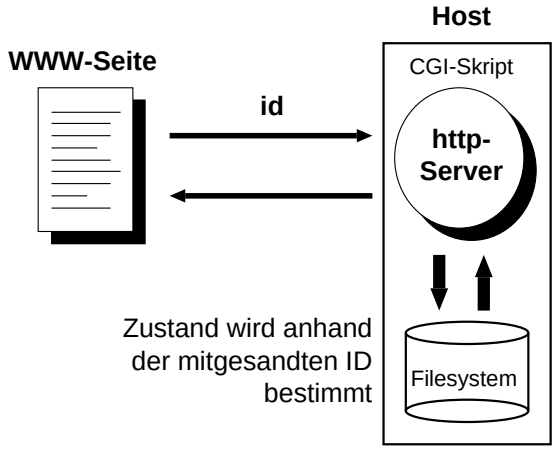

Abbildung 36: Speicherung der State-Information auf dem Dateisystem

Die Speicherung der Daten im Dateisystem setzt voraus, daß der WWW-Server mit Schreib-Zugriffsberechtigung auf das Dateisystem ausgestattet ist. Die Zugriffsberechtigung muß in diesem Falle über das Schreiben der Logging- und Errordateien hinausgehen. Neben dem Sicherheitsrisiko für das System gibt es bei dieser Methode noch zwei weitere Nachteile: Bei Applikationen mit Multi-User-Zugriff müssen die Daten für die verschiedenen Benutzer entweder in indviduellen Dateien gespeichert werden, oder es muß ein Locking-Mechanismus vorgesehen werden, der bei gleichzeitigem Zugriff ein Überschreiben der Daten verhindert. Diese Maßnahmen sind zwar effektiv, aber schwer skalierbar. Der zweite Nachteil ist der Umstand, daß alle Prozesse auf dem gleichen Host ablaufen, bzw. auf die Daten über z.B. NFS zugegriffen werden muß. Da neben der Ausführung des Prozesses, auch zumeist Daten- oder Programmdateien vom Dateisystem geladen werden müssen, kommt es hier auch zu Performance-Problemen, da Datei-Zugriffe dem System viel Leistung kosten. Auch dies ist effektiv aber nicht skalierbar. Das typische Schema einer WWW-Applikation zeigt Abbildung 37.

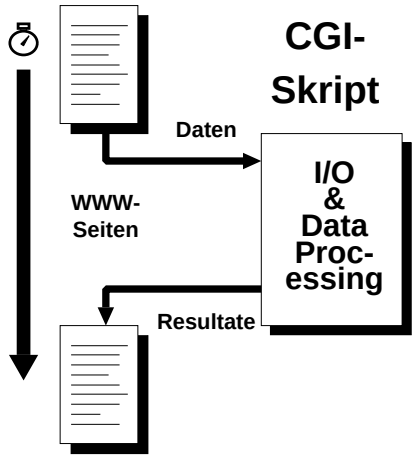

Abbildung 37: Schema einer WWW-Applikation

Das CGI-Skript ist bei dieser Konfiguration sowohl für die Verarbeitung der Daten als auch zum Verarbeiten der Ein- und Ausgaben zuständig. Es bieten sich keine Möglichkeiten, die Berechnung zu verteilen. Diese Probleme verhindern einerseits die Anbindung von rechenzeit-intensiven Applikationen an WWW, da dies bei Zugriff mehrerer Benutzer selbst starke Rechner lahmlegt. Weiters ist der Einsatz datenintensiver Applikationen problematisch, da der ganze Datenbestand bei jedem Aufruf von Neuem geladen und nach der Beendigung gespeichert werden muß. Dies stellt eine große Belastung für das Betriebssystem dar.

Eng verbunden mit dem Problem der fehlenden Möglichkeit zur Skalierung der Verarbeitung ist das Problem, WWW-Front Ends für bereits bestehende Applikationen und sogenannte *Legacy Applikationen* zu schaffen.

Das Hauptproblem hierbei ist, daß die meisten dieser Applikationen dialogorientiert sind. Sie verfügen über keine Schnittstelle zu WWW und sind entweder masken- oder dialogorientiert. Daher läßt sich durch ein normales CGI-Skript keine Anbindung herstellen, bei der die Verbindung zu der im Hintergrund laufenden Applikation gehalten wird.

10.5 Andere Lösungsansätze

Im folgenden werden vier Lösungsansätze präsentiert, welche sich stark voneinander unterscheiden. Sie lösen Teile der oben dargestellten Problemfelder:

Einbindung des Servers in die Applikation:

Verschiedene WWW-Server, die im Sourcecode auf dem Internet vorhanden sind, bieten eine serverseitige Schnittstelle für die Einbindung von Applikationen in den Server. Vertreter dieser Gruppe sind z.B. der *CERN-httpd* und der *Spyglass NCSA-Server*. Mit der Einbindung erreicht man eine direkte Anbindung des HTTP-Protokolls an die Ein- und Ausgaben der Applikation. Es ist daher der Weg über CGI-Skripts unnötig und man erhält eine Applikation aus einem Guß.

Dies bietet den Vorteil der Speicherung der Zustände im Server und macht keine weitere Verbindung zwischen Anwendung und WWW-Server erforderlich. Die oben genannten Probleme werden mit dieser Methode nicht behoben: Da der Server wiederum nur auf einem Host laufen kann, ergibt sich keine Möglichkeit zur Skalierung. Weiters muß für jede Anwendung ein eigener WWW-Server hochgestartet werden, was bei mehreren Anwendungen zu unwirtschaftlich großen Overheads führt.

State-Replication:

Perrochon präsentiert in seinem Paper [Perrochon and Fischer, 1995] IDLE, einen Lösungsansatz für die Einbindung von *Stateful Applications* an WWW. Der State wird bei dieser Methode in speziellen IDLE-Tags gespeichert. Der Weg durch die dialogbasierte Applikation bis zum augenblicklichen Zustand wird serverseitig gespeichert. Auf ihn kann über eine in den Tags vergebene Identifikation zugegriffen werden. Der zuletzt gespeicherte Zustand wird dann im Dialog mit der Applikation von vorne weg noch einmal aufgebaut. Auf diesen Zustand wird die neue Operation angewandt und gespeichert. Die Ergebnisse werden zurückgegeben.

Dieser Ansatz hat den Nachteil, daß er das Problem des *One Shot Gateway* nicht löst, sondern nur auf die Server-Seite verlagert. Der Nachbau des letzten Status in der Applikation erfordert einen neuen Aufruf und benötigt je nach Anzahl der bereits getätigten Operationen eine gewisse Zeit zum Nachvollziehen.

Interface Gateways:

Als Lösung für dialogbasierte Applikationen werden *Interface Gateways* vorgeschlagen [Barta and Hauswirth, 1995]. In diesem Falle simuliert das Gateway als CGI-Programm den Benutzerdialog mit der dialogbasierten Anwendung. Die Daten, die vom Benutzer normalerweise im Dialog mit der Anwendung eingegeben werden, werden in verschiedener Feinheit in einem WWW-Formular verpackt und an das Gateway weitergegeben. Das Gateway führt dann den entsprechenden Dialog durch, speichert die Ergebnisse und gibt diese in gesammelter Form an den Benutzer zurück. Bei der

Durchführung des Dialoges hat das Gateway die Aufgabe, die Rückgaben der Anwendung zu analysieren. Da diese in Maskenform für einen menschlichen Benutzer bestimmt sind, ist dies keine leichte Aufgabe. Als besondere Probleme treten dabei das Erkennen des Seitenendes und die Interpretation der Bildschirmsteuerzeichen.

Diese Methode wurde unter anderem bei der Implementation des WWW-Interfaces für das BIBOS-System der österreichischen Universitäten verwendet. Dort stellt es eine gute Alternative zum Zugang über eine Terminal-Sitzung dar.

Diese Methode bildet eine Möglichkeit der Anbindung von Legacy-Applikationen an WWW. Wenngleich das Gateway intelligent gestaltet und die Dialogfunktion für den Benutzer nachgebildet ist, ist diese Methode weder skalierbar noch ermöglicht sie die Aufgabe eines Auftrags in Batch.

Java:

Java ist eine objektorientierte Sprache, welche als *Plug In* bei verschiedenen WWW-Browsern verwendet wird. Durch Java-Applets können über den Darstellungsumfang von HTML hinausgehende Elemente in WWW Seiten realisiert werden. Das Hauptanwendungsgebiet von Java im Internet wird beschrieben als:

> *"Because there are so many different computers connected to each other, they need a language that is not tied to a particular platform to exchange programs. Java is ideal for this purpose [. . .] It is possible to download a Java program to any computer on the internet and execute it without worrying about the system on which the program was developed."*
> [van Hoff *et al.*, 1995, S. 16]

Java ist eine portable Lösung für das Auslagern von Anwendungsteilen oder ganzen Anwendungen auf die Seite des WWW-Browsers. Es können mit Java zwar interaktive Seiten gebaut werden, aber interaktive Anwendungen sind durch die Beschränkung des Befehlsumfanges und die ausschließliche Ausführung des Applet auf der Client-Seite nicht möglich. Würden diese Beschränkungen auf die Umgebung des WWW-Browsers aufgehoben, stünde dem Raub von Rechenzeit und dem Sammeln von sensitiven Daten durch Java-Programme nichts mehr im Wege.

In der Version 2.1 ist Java auf den Browser beschränkt und daher zum interaktiven Dialog mit dem Benutzer client-seitig fähig. Der Informationstransfer zu einer serverseitigen Anwendung wird dadurch aber nicht gelöst.

10.6 Linda und WWW

Mit Einsatz eines normalen CGI-Skripts kann eine Applikationsstruktur, wie in Abbildung 37 dargestellt, erreicht werden. Für viele Anwendungen ist diese Struktur jedoch aus folgenden Gründen nicht ausreichend:

- Der Benutzer müßte bei langen Ausführungszeiten auf die Ergebnisse warten. Der W3-Browser wäre durch das Programm blockiert.
- Es kann keine Unabhängigkeit zwischen Datenerfassung und Abfrage durch den Benutzer erreicht werden. Jede Informationsabfrage durch den Benutzer resultiert normalerweise in einer Abfrage der Ressourcen.
- Informationsabfrage aus Dateien durch ein CGI-Script stellt ein Sicherheitsrisiko dar.
- Die Applikation ist nicht unabhängig von der im Front End oder auf dem Dateisystem gespeicherten State-Information. Das WWW Front End sollte nur eine Darstellung eines internen Zustandes einer Applikation sein, nicht Medium zur Speicherung und Weitergabe dieser.

Die Integration von Linda in das WWW-System bietet eine Lösung für die aufgezählten Problemfelder durch eine Änderung in der Applikationsstruktur (siehe Abbildung 38).

Durch die Einbindung von Linda ist das Front End mit der Applikation durch den Tuplespace verbunden. Dies führt zu einer Aufspaltung der Aufgaben in den Teilen der Gesamtanwendung in Verarbeitung der Ein- und Ausgabe und in die Verarbeitung der Daten. Das CGI-Skript in der ursprünglichen Konfiguration zuständig für die Abwicklung der ganzen Applikation ist nur mehr für das Verarbeiten der Eingabe- und Ausgabe zuständig. Die Applikation selbst muß nicht notwendigerweise aus einem einzelnen Client bestehen, sondern kann sich auf die Zusammenarbeit verschiedener Clients in verschiedenen Sprachen auf verschiedenen Rechnern erstrecken. Der Tuplespace dient als Kommunikations- und Koordinationsmedium für die Programmteile und für das Front End.

In erster Konsequenz läßt sich damit eine Anwendungsstruktur erzeugen, die vom Schema her der Verwendung von CGI-Skripts ähnlich ist (siehe Abbildung 39). Das CGI-Skript hat die Aufgabe, die Datenübertragung zwischen Server/Browser

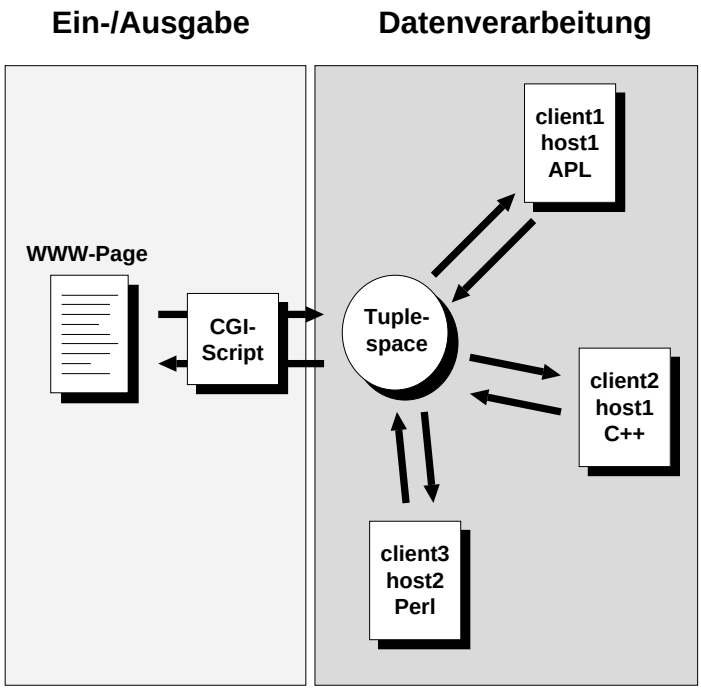

Abbildung 38: Applikationstruktur WWW mit Linda

und dem Tuplespace zu bewerkstelligen. Mittels `out()`-Befehlen werden die Daten vom Front End an den Tuplespace übertragen. Sie werden von der zu dieser Zeit schon aktiven Applikation mittels `in()` eingelesen und verarbeitet. Die Ergebnisse werden mittels `out()` von der Applikation wieder in den Tuplespace gestellt. Diese werden vom CGI-Skript mit `in()` übernommen und in darstellungsfähige Form gebracht. Danach werden sie durch den Browser angezeigt.

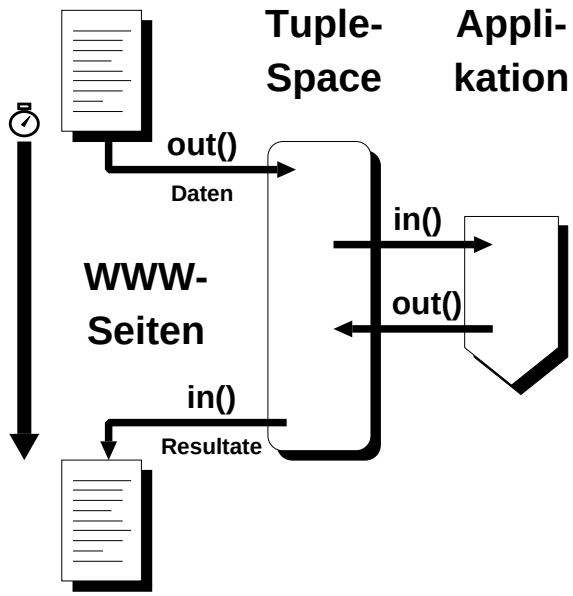

Abbildung 39: WWW-Linda mit CGI-ähnlicher Struktur

Obwohl diese Struktur komplexer als die Verwendung eines einfachen CGI-Skripts ist, birgt sie die Möglichkeit der Skalierbarkeit nach verschiedenen Kriterien. An dem verbindenden Tuplespace können Clients verschiedener Applikationen auf Abfragen ihrer Front Ends warten. Wenn diese Clients gleicher Art sind, so kann eine Skalierung der Anfragen nach Benutzern erfolgen. Bei gleichzeitiger Nutzung der Applikation durch verschiedene Benutzer dient der Tuplespace als Buffer, bis die Anfrage vom nächsten freiwerdenden Client bearbeitet wird. Bei der Verwendung von spezialisierten Applikationsclients können die Verarbeitungsschritte nach ihren Anforderungen an Daten und Ressourcen skaliert werden. Die Aufgabenverteilung zwischen CGI-Skript und Applikation ist in Tabelle 22 dargestellt.

Ein wichtiger Punkt beim Einsatz von Linda ist der Umstand, daß die Applikation durch den Tuplespace als Buffer abgekoppelt vom Front End aktiv sein kann. Sie kann daher als Hintergrundprozeß laufen und muß nicht bei jedem Request neu

CGI-Skript	**Linda-Client**
• Eingabeüberprüfung • Fehlerbehandlung für falsche Eingaben • Warten auf das Ende der Bearbeitung • Visualisierung der Ergebnisse • Verarbeitung der Return- und Fehlercodes	• Verarbeitung der Tuplespace-Daten • Sicherung der Applikationsdaten • Rückgabe der Ergebnisse an den Tuplespace • Rückgabe von Return-Codes • Anzeigen des *Endes der Bearbeitung*

Tabelle 22: Datentransfer, Front End – Tuplespace

hochgestartet und heruntergefahren werden. Durch die Koordinationsfunktion von Linda ist auch kein Locking der Zugriffe notwendig.

In Erweiterung zu der gerade vorgestellten Applikationsstruktur lassen sich auch lang laufende Prozesse mit einem WWW Front End versehen. In diesem Falle wartet der Browser nicht auf das Ergebnis der Operation, sondern erhält nur eine Bestätigung, daß die Anfrage zur Bearbeitung übernommen wurde. Der Benutzer kann danach normal weiterarbeiten und fragt nach einiger Zeit nach, ob die Ergebnisse seiner Anfrage bereits vorliegen. Abbildung 40 stellt diese Applikationsstruktur dar.

Vom Front End werden die Daten, wie schon im vorigen Beispiel, mit `out()` in den Tuplespace übertragen. Die Anwendung liest die Daten entweder mit `in()` oder `nbin()` ein. Die Verwendung von `nbin()` ist in Fällen sinnvoll, wo die Anwendung selbst nicht immer aktiv ist, sondern nur in regelmäßigen Abständen überprüft, ob Aufträge zur Bearbeitung im Tuplespace liegen. Der Benutzer kann mittels `nbin()` überprüfen, ob die Ergebnisse seiner Operation schon vorliegen. Die Verwendung von `nbin()` ist hier obligatorisch, da das Front End bei Nichtauffindung der Ergebnisse sonst blockieren würde. Programmtechnisch wird diese Abfrage natürlich nicht durch Eingabe der Funktion durchgeführt, sondern kann durch einen entsprechenden Button im Browser ausgeführt werden. Nach Beendigung der Bearbeitung stellt die Applikation die Ergebnisse in den Tuplespace. Von da an können sie vom Browser dargestellt werden. Der graue Pfeil zeigt in Abbildung 40 den Informationsfluß in der Applikation an. Eine automatische Benachrichtigung des Browsers ist im Rahmen des `HTTP`-Protokolls nicht

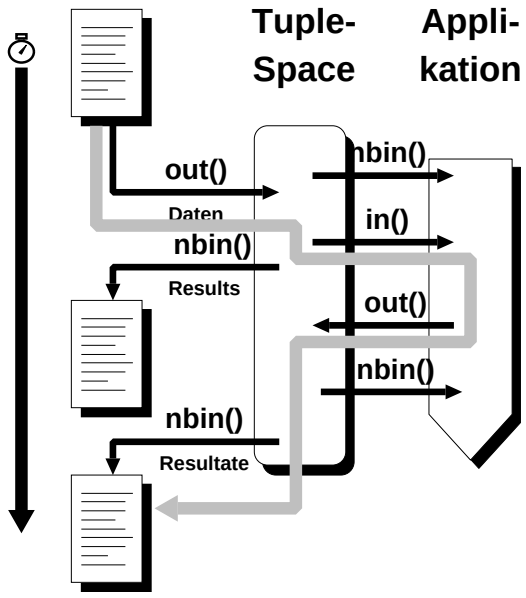

Abbildung 40: WWW und Linda – vom Front End unabhängige Applikation

möglich. Eine automatische Abfrage kann jedoch durch die in Browsern wie z.B. *Netscape* eingebaute *Reload*-Funktionalität erreicht werden. Bei diesem Vorgang wird die Seite in bestimmten Zeitabständen neu geladen. Hierbei wird ein sich eventuell veränderter Inhalt auf den neuesten Stand gebracht.

10.7 Praktische Implementation

In diesem Abschnitt werden einige kleine Beispiele für die Funktion gezeigt. Es wird spezielles Augenmerk auf den Informationstransfer zwischen Browser $\longrightarrow$ Tuplespace $\longrightarrow$ Applikation und umgekehrt gelegt. Abbildung 41 zeigt den Programmaufbau einer WWW-Linda-Anwendung. Sowohl das CGI-Skript als auch die Applikation sind in Linda- bzw. WWW-Schnittstellen eingebettet.

Die einzige Voraussetzung für den Einsatz von Linda mit WWW-Applikationen ist die Verfügbarkeit eines Perl-Linda-Servers. Das erste Beispiel in Abbildung 42 zeigt den Datentransfer zwischen Front End und Tuplespace. Die Daten werden aus einem HTML-`FORM` ausgelesen und mit den im Form zugewiesenen Namen in den Tuplespace gestellt. In diesem Beispiel läßt sich die in Abbildung 41 dargestellte Struktur der Schnittstellen sehr gut erkennen.

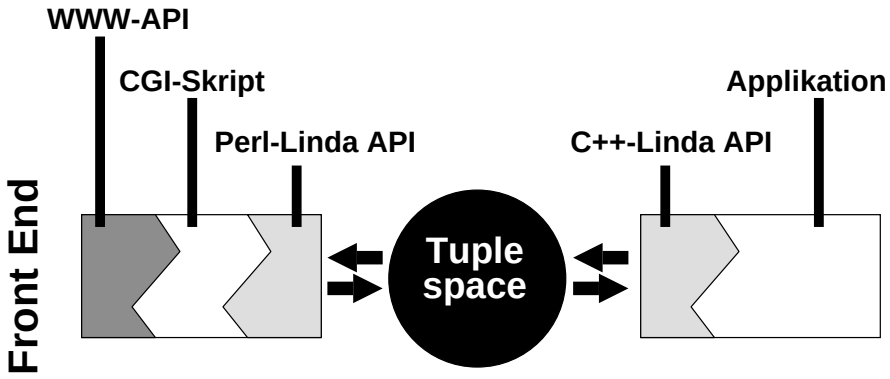

Abbildung 41: WWW-Linda Anwendung

```
#!/usr/local/bin/perl                  > Aufruf von Perl
require "linda-cli.pl";                > Linda-API
require "cgi-lib.pl";                  > CGI-API
&ReadParse;                            > Lesen der FORM-Info
&register_client("aig", 7999);         > Anmeldung beim TS

for( sort keys(%in))                   > Datentransfer zum TS
        { &out($_, $in{$_}); }           (VarName, Wert)
&close_client();                       > Schließen der Verbindung
                                         zum TS
print &PrintHeader;                    > Mime-Header:
print "\n<h1>Data transferred</h1>\n";"Content-type: text/html"
```

Abbildung 42: Datentransfer, Front End – Tuplespace

Dieses Beispiel soll nur das Prinzip des Transfer veranschaulichen. Beim realen Einsatz in einer Applikation sollten Maßnahmen getroffen werden, um die Tuple verschiedener Applikationen gegeneinander abzugrenzen. Dies wird in den Beispielen in Abschnitt 10.8 noch näher erläutert.

Der Transfer der Daten in die andere Richtung funktioniert in ähnlicher Form. Die im Tupleformat im Tuplespace gespeicherten Tuple müssen vom CGI-Skript aus dem Tuplespace gelesen oder entfernt werden. Sie werden dann in darstellungsfähiges HTML-Format umgewandelt und an `STDOUT` gesendet. Die so übermittelten Ergebnisse werden vom Browser dargestellt.

Ein Source-Beispiel dafür ist in Abbildung 43 die Darstellung des Tuplespace-Inhalt auf einer WWW-Seite. Dieses Codebeispiel wird unter anderem im `WWW-Lindatool`[6] verwendet. Dieses dient als WWW Front End, welches die manuelle Eingabe von Linda-Operationen und die Darstellung des Ergebnisses der Operation erlaubt.

```
#!/usr/local/bin/perl                    > Aufruf von Perl
require "linda-cli.pl";                  > Linda-API
require "cgi-lib.pl";                    > CGI-API

&register_client("aig",7999);            > Anmeldung beim TS
@tl=&nbrd("?*");                         > Lesen aller Tuple
&close_client();                         > Schließen der Verbindung

print &PrintHeader;                      > Mime-Header
print "<h2>Content:</h2><hr>\n";          > Überschrift
for( @tl )                               > Schleife über alle
{                                          Tuple
    @tu=&detrans($_);                    > Decodierung in ASCII
    print "@tu<br>\n";                   > Tuple wird ausgegeben
}
print "<hr>\n";
```

Abbildung 43: Datentransfer, Tuplespace – Front End

Der Datentransfer zwischen der eigentlichen Anwendung und Linda wurde schon in den Abschnitten 7 und 8 genau beschrieben und wird an dieser Stelle nicht mehr behandelt. Aufgrund der einfachen Einbindung von Linda in das WWW System können Applikationen geschaffen werden, welche sich mit den normalerweise zur

[6] zu finden unter `http://aia.wu-wien.ac.at/WJS/lindatool.pl`

Verfügung stehenden Möglichkeiten schwer bis gar nicht realisieren lassen. Im folgenden Abschnitt werden zwei praktische an der Abteilung für Angewandte Informatik laufende Anwendungen und ihre Implementation vorgestellt.

10.8 Applikationen mit WWW und Linda

Die folgenden Applikationen wurden einerseits zum Testen von Perl-Linda an realen Anwendungen, andererseits weil Linda eine einfache, praktikable Lösung für die Problemstellung darstellte, entwickelt. Zuerst wird Linda-Tool, eine Applikation zur einfachen Veranschaulichung der Fähigkeiten von Linda, vorgestellt. Diese Applikation dient in der Ausbildung dazu, den Studierenden das Linda-Konzept näherzubringen. Bei der zweiten Applikation handelt es sich um *DiscTool*, eine Anwendung zum Visualisieren der Plattenauslastung in einem Rechnernetz. Die dritte Applikation ist *World-Wide-Web-Date*, eine Beispielanwendung eines Transaktionssystems, das auf dem *Herzblatt*-Prinzip aufbaut. Für jede Anwendung wird zuerst die Problemstellung, die Protokolle, die Implementation und die Erfahrungen im Betrieb, dargestellt.

10.8.1 Linda-Tool

Das Linda-Tool ist ein WWW-basiertes Front End zu Perl-Linda. Es ist komplett in Perl geschrieben und bietet die Möglichkeit, über WWW mit einem Tuplespace zu kommunizieren und Linda-Befehle abzusetzen. Folgende Funktionen sind im Linda-Tool abgebildet:

Connect:
Diese Funktion ist auf der Einstiegsseite abgebildet. Der Benutzer wird aufgefordert, die Netzwerkadresse und den Port des Linda-Tuplespace anzugeben. Danach wird er zum entsprechenden Tuplespace verbunden und die Funktionen-Seite angezeigt. Gibt der Benutzer keinen bestimmten Tuplespace ein, so wird die Verbindung zu einem Default-Tuplespace hergestellt.

Perl-Linda-Funktionen:
Von der Funktionen-Seite (siehe Abbildung 44) kann der Benutzer die Perl-Linda-Funktionen `lst()`, `nbin()`, `nbrd()`, `out()` und `bye()` aufrufen. Um Verzögerungen im Front End zu vermeiden, wurden nur die

nicht blockierenden Abfragekommandos aus dem Linda-Befehlssatz zur Verfügung gestellt.

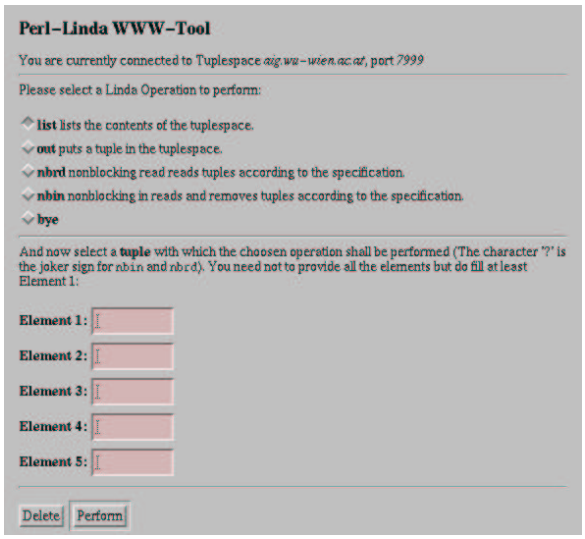

Abbildung 44: Funktionen-Seite von Linda-Tool

Der Benutzer wählt das entsprechende Kommando aus und füllt die Elementfelder aus. Es können bis zu 5 Elemente angegeben werden. Wurde das Kommando abgeschickt, wird dem Benutzer eine Ergebnisseite dargestellt auf der entweder der aktuelle Inhalt des Tuplespace (`lst()`, `out()`) oder das Ergebnis des entsprechenden Befehls (`nbrd()`, `nbin()`) dargestellt ist (siehe Abbildung 45). Von dieser Seite kann der Benutzer wieder zur Funktionen-Seite zurückkehren.

Während dieses Ablaufs wird die Netzwerkadresse und das Port des Servers immer in `HIDDEN`-Tags in den Seiten weitergegeben. So bleibt der Benutzer mit dem gleichen Tuplespace verbunden.

Disconnect:

Die Auswahl des Befehls `bye()` bringt den Benutzer auf eine End-Seite, wo Links zu weiteren Informationsquellen angegeben sind.

Durch das WWW-Front End erlaubt Linda-Tool das Observieren des Inhalts eines Tuplespace mit dem Befehl `lst()` über ein systemunabhängiges Front End. Weiters können durch das Front End die Linda-Befehle `out()`, `nbin()` und `nbrd()` eingegeben werden. Mit ihnen können Tuple dem Tuplespace hinzu-

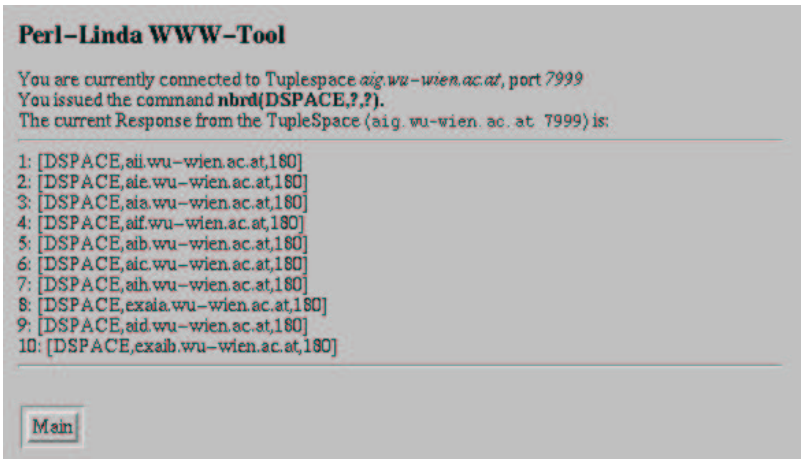

Abbildung 45: Ergebnisseite von Linda-Tool

gefügt und entnommen werden. Die Benutzung von Linda-Tool bietet gegenüber dem Zugriff auf den Tuplespace mittels `telnet` folgenden Vorteile:

- Der Zugriff erfolgt, unabhängig vom System, immer über das gleiche Front End.
- Das Front End ist intuitiv und damit auch für Anfänger leicht erlernbar.
- Mehrere Benutzer können gemeinsam auf den Tuplespace zugreifen und somit beobachten, wie sich ihre abgesetzten Befehle gegenseitig beeinflußen. Durch ein Neu-Laden der Seite mittels `[Reload]` kann das Ergebnis des letzten Befehls aufgefrischt werden. In der `telnet`-Version könnte dies nur durch eine Neu-Eingabe des Befehls erreicht werden.
- Die Ergebnisse der Kommandos werden in übersichtlicher Form dargestellt. Änderungen des Tuplespace-Inhalts, die durch das abgesetzte Kommando hervorgerufen wurden, werden in Fettdruck dargestellt.
- Jede Seite kann direkt vom WWW-Browser ausgedruckt werden. Diese können entweder als Lehrveranstaltungsunterlage oder als Dokumentation verwendet werden.

10.8.2 DiscTool

An der *Abteilung für Angewandte Informatik* werden 20 UNIX-Workstations verschiedenster Bauart in einem Netzwerk betrieben. Die in der Abteilung benötigten Netzwerkdienste sind auf einzelne Workstations verteilt. Ebenso wie die Dienste haben die Benutzer an der Abteilung Präferenz für einen oder zwei der Rechner und arbeiten hauptsächlich nur auf diesen Systemen. Die Festplatten und die darauf befindlichen Dateisysteme sind nicht verteilt und somit werden alle diesen Rechner betreffenden Daten auf diesem lokal abgespeichert.

Durch automatische Prozesse und von verschiedenen Services werden laufend Log-Dateien und andere Daten gepuffert. Wenn eines der Services eine Fehlfunktion hat bzw. die Dienste ungewöhnlich oft in Anspruch genommen werden, werden vermehrt Daten gespeichert. Durch die an der Abteilung angespannte Plattenplatzsituation kommt es daher zu Auslastungen jenseits der 100%-Grenze[7]. Die Aufgabe der Systemadministration ist es, diese Engpäße möglichst früh zu erkennen und entsprechend gegenzusteuern.

Zum Anzeigen des freien Plattenplatzes werden unter UNIX je nach System die Befehle `df` (BSD-UNIX) oder `bdf` (HP-UX) verwendet. Die Befehle liefern die folgende Ausgabe:

```
Filesystem  1024-blocks        Used       Avail Capacity  Mounted on
/dev/rz0a         63167       32166       24684    57%    /
/proc                 0           0           0   100%    /proc
/dev/rz0g       1733317      869721      690264    56%    /usr
/dev/rz1c        991923      293302      599428    33%    /disk2
```

Es werden das Devicefile der entsprechenden Festplattenpartition, die Gesamtkapazität, der bereits durch Dateien belegte Plattenplatz, der freie Speicherplatz, die Auslastung und das Verzeichnis im UNIX-Dateisystem, an dem sich die entsprechende Partition befindet, angezeigt. Der Systemadministrator kann damit auf einen Blick erkennen, wo es am Rechner Probleme gibt und diese beheben.

Was bei einem Rechner mit einem einfachen Befehl getan ist, verursacht bei mehreren Rechnern große Probleme, da man sich zur Kontrolle auf jedem Rechner

[7] UNIX-Dateisysteme können bis zu 110% ausgelastet werden. Dies ist aber nur dem Superuser möglich.

separat einloggen und den Befehl absetzen muß. Es gab somit keine Möglichkeit zur Verdichtung der Information über den abteilungsweiten Zustand der Platten. Probleme wurden jeweils nur von den, auf dem Rechner gerade arbeitenden Benutzern gemeldet, wenn der Zustand der Festplatten am Abteilungsnetz bereits akut war.

Eine Lösung zur Erleichterung des Überblicks über den Zustand ist die Darstellung des Gesamtzustandes in verdichteter Form. So läßt sich der Zustand des Systems auf einen Blick darstellen. Probleme können sofort erkannt und behoben werden. WWW bietet als Darstellungsbasis den Vorteil, daß die Systemadministration den Zustand system- und ortsunabhängig betrachten kann. Weiters kann durch Einbindung von Icons die Informationsdichte vergrößert werden. Das Schema der Applikation ist in Abbildung 46 dargestellt.

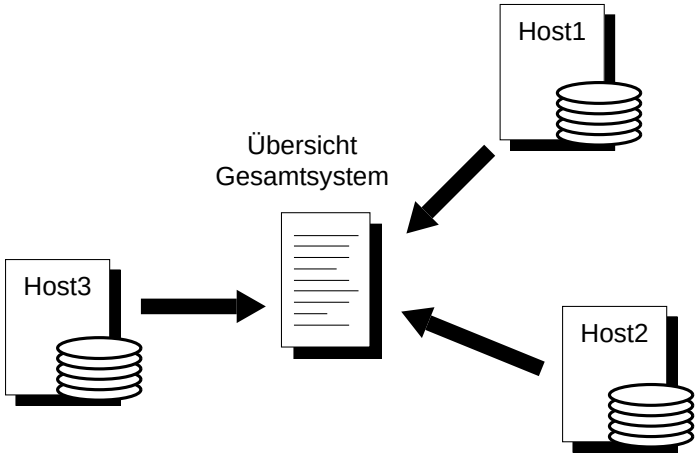

Abbildung 46: Schema von DiscTool

Zur Implementation dieses Systems müssen die Information über den Zustand der Dateisysteme in gesammelter Form vorhanden sein. Es ist jedoch nicht notwendig, die Information bei jeder Anfrage auf den neuesten Stand zu bringen. Für die Anwendung reicht es vollkommen aus, die Plattenplatzdaten in regelmäßigen Zeitabständen auf den neuesten Stand zu bringen. Die Lösung, diesen Update mittels RPC durchzuführen, birgt zwei Probleme:

- Der aufrufende Benutzer muß auf allen Systemen RPC-Berechtigung haben.
- Wird zur Lösung des vorigen Problems das Programm automatisch hochgestartet, so muß der Benutzer, der das Programm hochstartet, Zugriffsberech-

tigung auf eine gemeinsame Datei auf dem Host haben, auf dem die Daten gesammelt werden.

Mit Hilfe von Linda kann dieses Problem relativ einfach und elegant gelöst werden. Die Darstellung der Anwendungsstruktur findet sich in Abbildung 47.

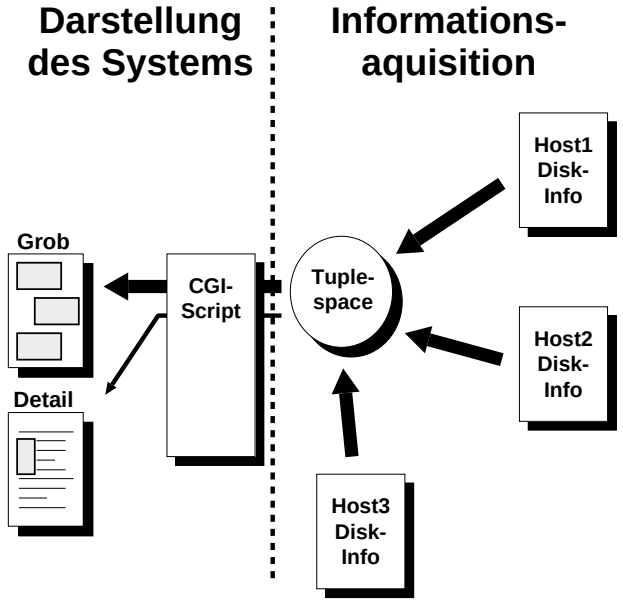

Abbildung 47: Applikationsstruktur von DiscTool

Ein Tuplespace wird als zentrales Speichermedium verwendet. Die auf den Rechnern laufenden Clients werden automatisch beim Start des Systems mitgestartet und legen ihre Information in entsprechenden Intervallen im Tuplespace ab. Will die Systemadministration die Information abrufen, so liest ein CGI-Skript die Information aus dem Tuplespace aus und stellt diese in der gewünschten Form dar. Die Darstellung der Information im Tuple folgt der Ausgabe der Kommandos `bf` und `bdf`. Sie ist in Abbildung 48 auszugsweise dargestellt.

Bei allen Tuple ist das erste Element `DSPACE`. Dies ist erforderlich, um die Tuple der Applikation von anderen, eventuell noch im Tuplespace enthaltenen Tuple abzugrenzen. Zur Applikation gehören zwei Arten von Tuple. Sie sind in Abbildung 48 getrennt dargestellt. In den ersten zwei Tuple sind Informationen über das Update-Intervall der Rechner enthalten. Die restlichen Tuple enthalten die zur Darstellung der Plattenauslastung benötigte Information in der Form:

```
(DSPACE,aie.wu-wien.ac.at,150)
(DSPACE,exaib.wu-wien.ac.at,180)

(DSPACE,aie.wu-wien.ac.at,/dev/rz1c,168408,82,/disk2)
(DSPACE,aie.wu-wien.ac.at,/dev/rz3a,11056,81,/)
(DSPACE,aie.wu-wien.ac.at,/dev/rz3d,37103,20,/var)
(DSPACE,aie.wu-wien.ac.at,/dev/rz3g,60027,91,/usr)
(DSPACE,exaib.wu-wien.ac.at,/dev/dsk/0s0,35623,88,/)
(DSPACE,exaib.wu-wien.ac.at,/dev/dsk/5s0,83568,91,/5s0)
```

Abbildung 48: Darstellung der Platten-Information im Tuplespace

(DPSPACE,*Hostname*,*Device*,*Plattengröße(byte)*,*Auslastung(%)*,*Mountpoint*)

Diese Information wird im Front End in zwei verschiedenen Verdichtungsgraden dargestellt. In der Gesamtübersicht wird die Information nach Rechnern geordnet. Eine Plattenpartition wird in Form eines Icons dargestellt. An diesem kann der Benutzer erkennen, wie die ungefähre Auslastung der Platte ist. Durch die Wahl der Icons in farblicher und gestalterischer Hinsicht läßt sich auf den ersten Blick erkennen, wo Probleme auftreten.

Bei der Überblicksansicht werden die im Tuplespace gespeicherten Daten nicht direkt, sondern in symbolischer Form dargestellt. Anders ist dies bei der zweiten Verdichtungsstufe, der Detailansicht. Wird genauere Information über eine Partition gewünscht, so kann diese durch Klicken auf das entspreche Icon abgefragt werden. Es werden dann die im Tuplespace gespeicherten Daten für den gewählten Rechner in detaillierter Form angezeigt. Die gewählte Partition wird im Schriftstil **fett** angezeigt, um ein schnelleres Auffinden zu ermöglichen.

Durch den Einsatz von Linda konnte die Anwendung so an das WWW angebunden werden, daß eine Kontrolle der Plattenauslastung im Netzwerk und ein Erkennen von auftretenden Engpässen von jeder Betriebssystem-Plattform aus möglich ist. Auch von außerhalb des Abteilungsnetzes können die Daten mit WWW-Browser und mit der Kenntnis der URL der WWW-Einstiegsseite abgefragt werden. Diese Möglichkeit der Fernabfrage wurde auch schon im praktischen Betrieb eingesetzt.

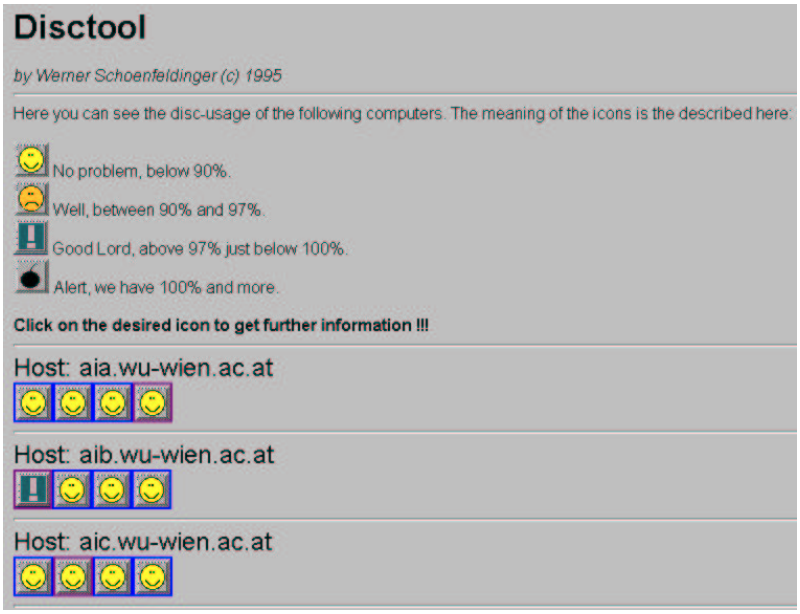

Abbildung 49: Screen-Shot – DiscTool Übersicht

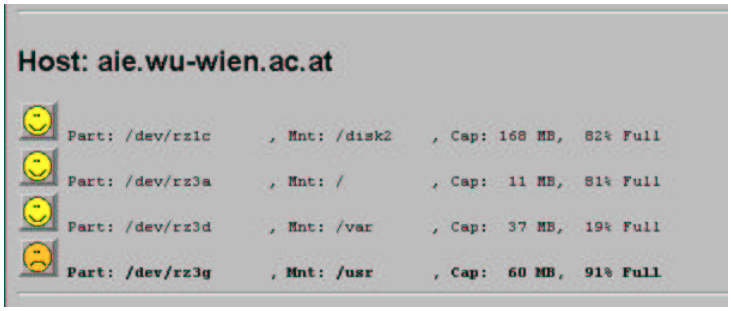

Abbildung 50: Screen-Shot – DiscTool Detailansicht

10.8.3 World-Wide-Web-Date

Die zweite Anwendung mit der Kombination Linda – WWW, die in diesem Rahmen vorgestellt wird, ist *World-Wide-Web-Date* (WWWeD). Diese Anwendung ist ein Prototyp für ein WWW-basiertes Transaktionssystem. Die Anwendung ermöglicht eine *Date*-Vermittlung über das World Wide Web, bei der jeweils ein interaktiver Benutzer mit einem im Datenbestand gespeicherten Partner auf ein Date geschickt wird. Ausgehend von dieser speziellen Form kann das Programm mit entsprechenden Abänderungen für andere Zwecke wie z.B. Lerngruppen-Vermittlung, Sport-Partnervermittlung etc. eingesetzt werden.

Die Anwendung hat folgende grundsätzliche Funktionen:

Anlegen eines Benutzers:
Mit dieser Funktion wird ein neuer Benutzer in die Datenbank aufgenommen und dessen persönliche Daten gespeichert. Es wird in diesem Zuge auch seine Teilnahmeberechtigung überprüft. Vom Benutzer werden folgende Daten erfaßt:

- Spitzname, Matrikelnummer, E-mail-Adresse und Geschlecht des Benutzers.
- Antworten zu 10 Fragen über sich selbst.
- Beschreibung *"Ich über mich in einem Satz"*.

Löschen eines Benutzers:
Mit dieser Operation werden alle benutzerspezifischen Einträge aus dem System entfernt.

Statistik:
Es wird eine Statistik über die Eckdaten des Systems angezeigt.

Regular Date:
Beim *Regular Date* kann der angelegte Benutzer Präferenzen über den zu treffenden Partner eingeben. Danach kann der Benutzer, nach einer Vorsondierung, unter den 5 am besten *passenden* Partnern auswählen. Der ausgesuchte Partner erhält eine E-mail mit der Beschreibung, der Bitte um ein Treffen und der E-mail-Adresse des Benutzers. Der Partner kann dann wählen, ob er den Benutzer kontaktieren will.

Blind Date:
Bei dieser Funktion wird der Benutzer auf die Liste aller *Blinddate-Wütigen*

gesetzt. Sobald ein gleichgesinnter Partner des anderen Geschlechts sich auch zu einem Blind Date bereiterklärt hat, bekommen beide eine Email mit dem Treffpunkt und dem jeweiligen Erkennungszeichen des anderen.

Der Benutzerdialog der Operationen *Anlegen*, *Löschen*, *Statistik* und *Blind Date* folgt dem Reply-Response Schema und ist im Prinzip durch das normale WWW-System ohne Probleme realisierbar. Die Operation *Regular Date* hat hingegen einen zweistufigen Benutzerdialog, welcher in Abbildung 51 dargestellt ist.

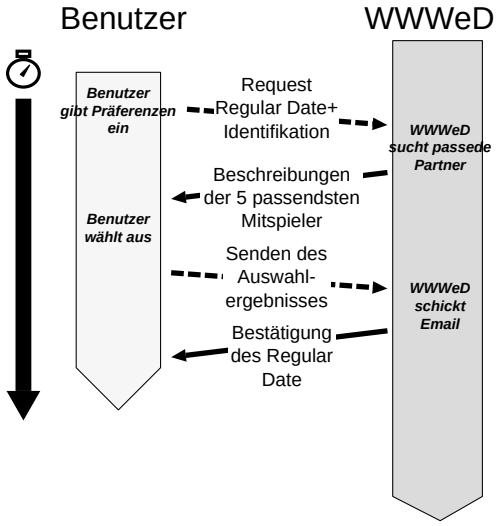

Abbildung 51: Benutzerdialog beim *Regular Date*

Bei der Implementation dieser Funktionalität ergaben sich bei der Anbindung an WWW folgende Problemstellungen, welche in der Natur des WWW-Systems liegen:

- Da beim Regular Date ein über ein Reply-Response-Schema hinausgehender Benutzerdialog notwendig ist, müßte die Applikation den Dialog beim Mehrbenutzerbetrieb abspeichern und jedes Mal neu laden und aufbauen oder einen Locking-Mechanismus einbauen, der dem im Dialog befindlichen Benutzer exklusiven Zugriff sichert. Weiters müßte sich der Benutzer bei jedem Dialogschritt neu identifizieren. Es ergäben sich daraus Redundanzen im Betrieb bzw. lange Antwortzeiten.
- In jedem Falle müßten sowohl die Benutzerstammdaten als auch die offenen Transaktionen bei jedem Aufruf neu geladen werden. Bei einer größeren

Anzahl von Benutzern würde allein schon dies zu einer enormen Belastung des Systems durch Festplattenzugriffe führen.

Der Einsatz von Linda als Plattform zwischen Front End und der im Hintergrund laufenden Applikation bietet eine Lösung für beide Problemkreise. Das Applikationsschema ist in Abbildung 52 dargestellt.

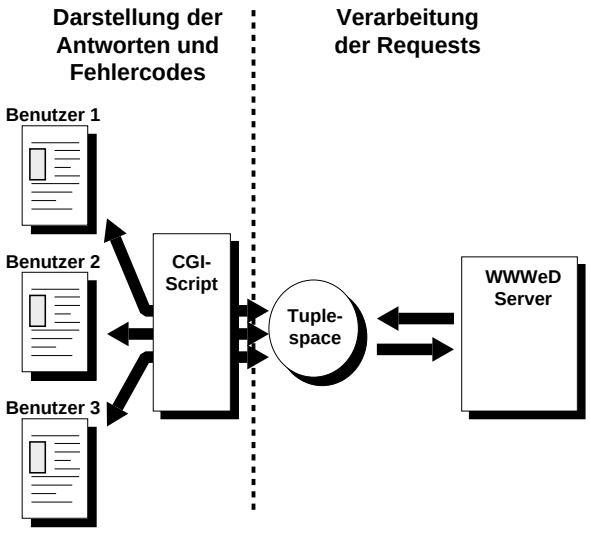

Abbildung 52: Applikationsschema WWWeD

Die von den Benutzern kommenden Requests werden im Tuplespace zwischengespeichert und nacheinander von der WWWeD-Server Applikation bearbeitet. Der Benutzer identifiziert sich beim Request. Seine `id` (Matrikelnummer) wird dann als Identifikation zwischen Server und Front End für die weiteren Abläufe verwendet. Dies schließt mehrere parallele Abfragen des gleichen Benutzers aus.

Im folgenden ist das dem WWWeD zugrundeliegende Protokoll erläutert. Zuerst werden die von der Anwendung benötigten Datentuple dargestellt. Danach wird die Abfolge der Tuple beschrieben und anhand einer Handsimulation eine Beispielsitzung durchgespielt. Für die einzelnen Funktionen sind in der Kommunikationsrichtung Front End $\longrightarrow$ WWWeD folgende Datentuple vonnöten:

Generelles Template:
(`wwwed`,*cmd*,...)

statistics:
(wwwed,s)

Funktion add:
(wwwed,a,*sex,id,data,passwd,name,userid,host,desc*)

remove:
(wwwed,r,*id,passwd*)

Funktion choose:
(wwwed,c,*id,choice*)

blindDate:
(wwwed,b,*id,passwd*)

regularDate:
(wwwed,d,*id,passwd,preferences*)

Generell wird bei den Tuple die Kommunikationsrichtung unterschieden. Dies erleichtert die Abfragen mittels Linda-Operationen. In der Kommunikationsrichtung Front End $\longrightarrow$ Server enthalten alle Tuple als erstes Element wwwed. Das zweite Element legt das Kommando fest. Je nach Funktion braucht der Server noch weitere Elemente zum Ausführen der entsprechenden Funktion. Die Datentuple werden von Server mittels ain() eingelesen. Dies stellt sicher, daß immer nur ein Kommando eingelesen wird. Nach der Bearbeitung dessen stellt der Server seine Resultate wieder in den Tuplespace. Die Tuple der Kommunikationsrichtung WWWeD $\longrightarrow$ Front End sind nachfolgend dargestellt:

choose:
(wwweda,c,*id,number,score,description*)
bei Beendigung der Übermittlung der Datentuples mit den Beschreibungen wird das Tuple (wwweda,c,*id*,ready) als Aufforderung für den Client, die Liste zu holen, in den Tuplespace gestellt.

statistics:
(wwweda,s,*#male,#fem,#bd_male,#bd_fem,#matches*)
Die Statistik beinhaltet die Anzahl der männlichen und weiblichen Teilnehmer. Weiters wird die Anzahl der BlindDate-Teilnehmer beider Geschlechter und die Gesamtzahl der Matches in den Elementen dargestellt. Dies ist die einzige Antwort, die nicht an einen bestimmten Teilnehmer gerichtet ist. Sie wird auf den Request hin nur auf den neuesten Stand gebracht.

error:

(`wwweda,e`,*id*,*errorcode*)
Dies gibt eine Fehlermeldung an das CGI-Skript zurück. Die Fehlercodes bestehen aus einem Wort. Dieses wird dann im CGI-Skript in den entsprechenden Fehlertext umgewandelt und dargestellt. Durch diese Gestaltung kann der Fehlertext unabhängig von einer Änderung des Server-Programmes direkt im CGI-Skript geändert werden kann.

message:

(`wwweda,m`,*id*,*msgcode*)
Dieses Datentuple hat funktionell die gleiche Form wie das Fehlertuple. Es wird jedoch statt einem Fehler eine entsprechende Meldung ausgegeben (z.B. Operation wie gewünscht durchgeführt).

Zur Trennung der Antwort- von den Anfragetuple haben alle Antworttuple `wwweda` als erstes Element. Das CGI-Skript wartet nach dem Absetzen der Anfrage[8] auf ein Tuple der Form (`wwweda`,*id*,`?*`), wobei *id* die bei der Anfrage mitgegebene Matrikelnummer ist. Durch die Bildung des Abfragepattern mit dieser eindeutigen Identifizierung wartet bei Multi-User-Betrieb jede Instanz des CGI-Skripts auf seine ganz individuelle Antwort. Es kommt daher zu keiner Überschneidung.

Zur Illustration des Protokolls wird in Tabelle 23 ein Beispiel für den Benutzerdialog gegeben. Es wird nach der Abfrage der Statistik vom Benutzer ein Regular Date durchgeführt. In der linken Spalte befinden sich jeweils die benutzerseitigen Tuple, während sich in der rechten Spalte die serverseitigen Operationen befinden.

Da der Server für jeden Benutzer weiß, in welchem Zustand er sich gerade befindet, werden doppel- bzw. falsche Dialogschritte erkannt und entsprechende Fehlermeldungen ausgegeben. Der Server liest den nächsten im Tuplespace wartenden Befehl mittels `ain(wwwed,?*)` ein. Er überprüft dann, ob das zweite Element einem zulässigen Kommando entspricht und ob der Benutzer mit der im dritten Element angegebenen Identifikation im System vorhanden ist. Trifft dies zu, dann wird der Befehl weiterbearbeitet. Für die richtige Zusammenstellung des Tuple ist das zwischen Front End und Tuplespace agierende CGI-Skript verantwortlich. Es werden hier grundsätzlich Eingabenüberprüfungen vorgenommen und sichergestellt, daß die dem Server übermittelten Tuple dem Protokoll entsprechen.

[8] Dies gilt nicht bei Abfrage der Statistik.

Front End		Server
Statistik: **out**(wwwed,s)	→	ain(wwwed,?*)
in(wwweda,s,?*) Darstellung der Ergebnisse	←	**out**(wwweda,70,52,5,0,20) ain(wwwed,?*)
Regular Date: **out**(wwwed,d,8850004,9235728336)	→	Wertet Präferenzen aus und sucht passende Partner.
in(wwweda,c,8850004,ready)	 ←	**out**(wwweda,c,8850004,1,93,*Beschr.1*) ... **out**(wwweda,c,8850004,5,139,*Beschr.5*) **out**(wwweda,c,8850004,ready)
in(wwweda,?,8850004,?*) Alle zuvor im Tuplespace gespeicherten Daten werden eingelesen und dargestellt	←	in(wwwed,?*)
out(wwwed,c,8850004,3)	→	Die Auswahl wird verarbeitet. Die Mail an den entsprechenden Partner wird geschickt.
in(wwweda,?,8850004,?)	←	**out**(wwweda,m,8850004,DONE)
`DONE` wird in die äquivalente Nachricht umgewandelt und dargestellt.		ain(wwwed,?)

Tabelle 23: Dialogbeispiel WWWeD

Durch den Einsatz von Linda konnte bei WWWeD eine flexible mehrbenutzerfähige dialogorientierte Anwendung mit WWW-basiertem Front End geschaffen werden. Mit der flexibleren Anwendungsstruktur ist gewisse Mehrarbeit beim Starten und Administrieren der Applikation verbunden, die jedoch mit den auf UNIX-Systemen vorhandenen Tools automatisiert werden kann.

10.9 Zusammenfassung: WWW und Linda

In diesem Abschnitt wurde das Design von Anwendungen mit World Wide Web (WWW) als Front End beschrieben. Nach einer Erklärung der Struktur eines WWW-Systems wurden Wege zur Anbindung von Programmen an den WWW-Server mittels des Common Gateway Interface (CGI) aufgelistet. Zur Erstellung von dialogbasierten Anwendungen stellten sich CGI und das von WWW genutzte Hyper Text Transfer Protokoll (HTTP) als nicht geeignet, da sie keine Möglichkeit zur Speicherung von Zustandsinformation der Anwendung boten. Es wurden vier verschiedene Lösungsansätze gezeigt die entweder nur eine Facette des Problems lösen oder bei der Lösung großen Overhead verursachen.

Durch den Einsatz von Perl-Linda in diesem Rahmen wurde der Tuplespace als Mittler zwischen Applikation und Front End vorgestellt. Der Tuplespace dient als Kommunikationsplattform zwischen Applikation und CGI-Skript. Auf ihn wird über die Perl-Linda-Funktionen zugegriffen. Neben einer Verteilung der Applikationsteile auf verschiedene heterogene Rechner im Netzwerk ermöglicht der Einsatz von Perl-Linda auch eine Haltung der Zustandsinformation in der Applikation und die Zusammenarbeit von gemischtsprachigen Clients mit einem WWW-Front End.

Es wurden drei Beispielanwendungen *Linda-Tool, DiscTool* und *World-Wide-Web-Date* beschrieben, die die Zusammenarbeit von Perl-Linda und WWW illustrieren. Neben den beschriebenen Applikationen existieren an der Abteilung für Angewandte Informatik noch weitere Anwendungen, wie z.B. der WU-weite Political Stockmarket. Das System wurde als Beitrag [Schönfeldinger, 1995] im Rahmen der *International WWW Conference* im Dezember 1995 in Boston präsentiert.

11 Zusammenfassung

In dieser Arbeit wurde Perl-Linda, ein Prototyp für die Implementation von Linda für ein Netzwerk von Rechnern, vorgestellt. Diese Implementation ermöglicht die Kommunikation und Koordination von gemischtsprachigen Anwendungen auf heterogenen Systemen. Gegenüber den bisherigen Implementationen stellt Perl-Linda eine frei verfügbare, auf jedem Unix-System implementierbare Lösung dar, Linda-Programme in verschiedenen Sprachen zu programmieren und auszuführen. Die Kommunikation und Koordination geschieht, zum Unterschied von verschiedenen kommerziellen Verteilungssystemen, auf der Applikationsebene und ist somit für den Entwickler und Anwender leicht anzupassen und zu erweitern.

Neben der Spezifikation des Perl-Linda-Servers wurde für verschiedene Programmiersprachen die Client-Schnittstelle zu Linda beschrieben und anhand von Beispielprogrammen die Nutzung der Koordinations- und Kooperationsfähigkeit von Linda dargestellt. Die verwendete Syntax und die erforderlichen Transformationen der Ein- und Ausgabeströme wurden für alle Perl-Linda-Kommandos in BNF beschrieben. Für den Perl-Linda-Server wurden weiters die durch die Verwendung der Sprache Perl auftretenden Restriktionen und Probleme aufgezeigt.

Es wurden die notwendigen Entwicklungsschritte für die Entwicklung eines parallelen Programmes in Perl-Linda beschrieben und an Beispielen demonstriert. In einer Fallstudie wurde der Einsatz von Perl-Linda zur Erweiterung des World Wide Web (WWW)-Systems beschrieben. Durch die Einbindung von Perl-Linda konnten die Nachteile des bestehenden Interface überwunden werden und es wurde die Möglichkeit für die Implementation global verfügbarer Transaktionssysteme mit WWW-Front End geschaffen.

Verglichen mit kommerziellen verteilten Systemen stellt Perl-Linda eine kleine aber flexibel einzusetzende Erweiterung dar. Perl-Linda ermöglicht das Zusammenwirken von gemischtsprachigen Applikationen in einem Netzwerk von heterogenen Rechnern. Der Perl-Linda-Prototyp stellt ein flexibles System für das Implementieren und Testen von neuen Client-Schnittstellen in anderen Programmiersprachen und ein Modell für die spätere, optimierte Implementation des Linda-Servers in C^{++} dar.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Andrews, 1991] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[Bakken and Schlichting, 1995] D. Bakken and R. Schlichting. Supporting Fault-tolerant Parallel Programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3), March 1995.

[Bakken, 1993] David E. Bakken. Supporting fault-tolerant parallel programming in Linda. Technical Report TR93-18, University of Arizona, Computer Science Department, 1993.

[Barta and Hauswirth, 1995] Robert Barta and Manfred Hauswirth. Parasite Interface Gateways. In *Electronic Proc. 4th Int. World Wide Web Conference "The Web Revolution"*, Boston, MA, December 1995.

[Beck and Cunningham, 1989] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *OOPSLA '89 Proceedings*, pages 1–6, October 1989.

[Ben-Ari, 1982] M. Ben-Ari. *Principles of concurrent programming*. Prentice Hall International Inc., Englewood Cliffs, NJ, 1982.

[Berners-Lee and Connolly, 1995] Tim Berners-Lee and D. Connolly. Hypertext markup language - 2.0. Request for Comments (Standard) RFC 1866, Network Working Group, November 1995.

[Berners-Lee *et al.*, 1995] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext Transfer Protocol – HTTP/1.0. Tr, World Wide Web Consortium, http://www.w3.org/pub/WWW/Protocols/HTTP1.0/draft-ietf-http-spec.html, April 1995.

[Booch, 1991] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[Butcher, 1991] Paul Butcher. Lucinda. In Wilson [1991b], pages 27–37.

[Butler *et al.*, 1993] Ralph M. Butler, Alan L. Leveton, and Ewing L. Lusk. p4-Linda: A portable implementation of Linda. Technical Report MCS-P374-0793, Mathematics and Computer Science Division, Argonne National Laboratory, July 1993.

[Callsen *et al.*, 1991] Christian J. Callsen, Ivan Cheng, and Per L. Hagen. The AUC C++-Linda system. In Wilson [1991b], pages 39–71.

[Cap *et al.*, 1993] Clemens H. Cap, Silvano Maffeis, Lutz Richter, and Volker Strumpen. Ein glossar wichtiger begriffe zu verteilten und parallelen systemen. Technical Report IFI-TR 93.44, Department of Computer Science, University of Zurich, Wintherthurerstraße 190, CH-8057 Zürich, november 1993.

[Cap, 1993] Clemens H. Cap. Massive parallelism with workstation clusters – challenge or nonsense? Technical Report IFI-TR 94.01, Department of Computer Science, University of Zurich, Wintherthurerstraße 190, CH-8057 Zürich, 1993.

[Carriero and Gelernter, 1985] N. Carriero and D. Gelernter. Applications experience with Linda. *ACM Sympos. on Parallel Programming*, July 1985.

[Carriero and Gelernter, 1989a] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[Carriero and Gelernter, 1989b] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[Carriero and Gelernter, 1991] Nicolas Carriero and David Gelernter. New optimization strategies for the Linda precompiler. In Wilson [1991b], pages 39–71.

[Carriero *et al.*, 1986] Nick Carriero, David Gelernter, and Jerry Leichter. Distributed data structures in Linda. *ACM Transactions on Programming Languages and Systems*, 8(1), January 1986.

[Ciancarini and Guerrini, 1993] P. Ciancarini and N. Guerrini. Linda meets Minix. *ACM Operating Systems Review*, 27(4):76–92, October 1993.

[Ciancarini, 1993a] Paolo Ciancarini. Coordinating rule-based software processes with ESP. Technical Report UBLCS-93-8, Laboratory for Computer Science, University of Bologna, Piazza di Porta S. Donato,5, 40127 Bologna, Italy, April 1993. Source: Internet.

[Ciancarini, 1993b] Paolo Ciancarini. Distributed programming with logic tuple spaces. Technical Report UBLCS-93-7, Laboratory for Computer Science, University of Bologna, Piazza di Porta S. Donato,5, 40127 Bologna, Italy, April 1993. Source: Internet.

[Comer and Stevens, 1993] Douglas E. Comer and David L. Stevens. *Internetworking With TCP/IP: Client-Server Programming And Applications*, volume III. Prentice Hall International, Inc., Engelwood Cliffs, NJ, 1993.

[Coulouris and Dollimore, 1988] George F. Coulouris and Jean Dollimore. *Distributed Systems: Concepts and Design*. Addison Wesley Publishers Ltd., Bath, UK, 1988.

[Coulouris *et al.*, 1994] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley Publishers Ltd., Reading, UK, 2 edition, 1994.

[Forman and Zahorjan, 1994] George H. Forman and John Zahorjan. The challenges of mobile computing. *IEEE Computer*, 27(4):38–47, April 1994.

[Forst *et al.*, 1994] Alexander Forst, eva Kuehn, Herbert Pohlai, and Konrad Schwarz. Logic based and imperative coordination languages. In *Proceedings of the PDCS'94, Seventh International Conference on Parallel and Distributed Computing Systems*, October 1994.

[Fountain, 1994] Terry J. Fountain. *Parallel computing: principles and practice*. Cambridge University Press, Cambridge, NY, 1994.

[Geist *et al.*, 1993] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Irdge National Laboratory, U.S. Department of Energy, Oak Ridge, TE, Mai 1993.

[Gelernter and Philbin, 1990] David Gelernter and James Philbin. Spending your free time. *BYTE 5/90*, pages 213–219, May 1990.

[Gelernter, 1985] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Gelernter, 1989] D. Gelernter. Multiple tuple spaces in Linda. *Parle 89*, page 1, March 1989.

[Gelernter, 1992] David Gelernter. *Mirror Worlds – or the Day Software Puts the Universe in a Shoebox... How It Will Happen and What It Will Mean*. Oxford University Press Inc., Oxford New York, 1992.

[Halfill, 1996] Tom R. Halfill. Unix vs windows nt. *Byte*, May 1996.

[Hasselbring, 1991] Willi Hasselbring. Combining SETL/E with Linda. In Wilson [1991b], pages 84–97.

[Hasselbring, 1993] W. Hasselbring. Prototyping Parallel Algorithms with PROSET-Linda. Technical Report TR04/93, University of Essen, Germany, 1993.

[Hennessy and Jouppi, 1991] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, 24(9):18–29, September 1991.

[Hietler, 1995] Gerold Hietler. The APL linda client. TR Nr. 13, Abteilung für Angewandte Informatik, Wirtschaftsuniversität Wien, Augasse 2–6, A-1090 Wien, July 1995.

[Hoare, 1974] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Janko, 1981] Wolfgang H. Janko. *APL II/2 – Eine Weiterführende Darstellung der Sprache und des Systems mit Anwendungen*. Athenäum Verlag, Königstein, DE, 1981.

[Kahn and Miller, 1989] Kenneth M. Kahn and Mark S. Miller. Response to "Linda in context". *Communications of the ACM*, 32(4):444–458, April 1989.

[Klute, 1994] R. Klute. Zweiter Gang. In Verlag Heinz Heisse GmbH & Co KG, editor, *iX Multiuser Multitasking Magazin*, volume 8, Hannover, August 1994. Verlag Heinz Heisse GmbH & Co KG.

[Kolarik, 1993] Thomas Kolarik. Kooperatives Arbeiten und Rechnen in einer gemischtsprachigen heterogenen Rechnerumgebung – Standards und Anwendungen. Dissertation, Wirtschaftsuniversität Wien, Augasse 2–6, A-1090 Wien, 1993.

[Korenzeniowski, 1995] Paul Korenzeniowski. Not that DOS. *Byte*, 20(11):113–118, November 1995.

[Leler, 1990] William Leler. Linda meets UNIX. *IEEE Computer*, 23(2):43–54, February 1990.

[Lewis *et al.*, 1995] Ted G. Lewis, Dave Power, Bertrand Meyer, Jack Grimes, Mike Potel, Ron Vetter, Phil Laplante, Wolfgang Pree, Gustav Pomberger, Mark Hill, James Larus, David Wood, Hesham El-Rewini, and Bruce W. Weide. Where is software headed? a virtual roundtable. *IEEE Computer*, 28(8):20–32, August 1995.

[Lewis, 1994] Ted G. Lewis. Where is computing headed? *IEEE Computer*, 27(8):59–63, August 1994.

[Mattson *et al.*, 1992] Timothy G. Mattson, Rob Bjornson, and David Kaminsky. The C-Linda language for networks of workstations. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University. Proceedings available via anonymous ftp from ftp.scri.fsu.edu in directory pub/parallel-workshop.92.

[Metaxas, 1995] P. Talkis Metaxas. Fundamental ideas for a parallel computing course. *ACM Computing Surveys*, 27(2):284–286, June 1995.

[Meyer, 1996] Bertrand Meyer. The reusability challenge. *IEEE Computer*, 29(2):76–79, Februar 1996.

[Mitlöhner, 1992] Johann Mitlöhner. Portierung von APL-programmen. Dissertation, Wirtschaftsuniversität Wien, A-1090, Augasse 2-6, 1992.

[Mitlöhner, 1993] Johann Mitlöhner. Distributed computing in the workstation environment. *APL Quote Quad (APL93 Conference Proceedings)*, 24(1), August 1993.

[Narem, 1990] James Narem. An informal operational semantics of C-Linda V2.3.5. Technical Report TR-839, Yale University Department of Computer Science, December 1990.

[NCSA, 1994] NCSA. Supporting FORMS with CGI. Technical report, W3-Consortium, http://hoohoo.ncsa.uiuc.edu/cgi/forms.html, 1994.

[Orfali *et al.*, 1994] Robert Orfali, Dan Harkey, and Jeri Edwards. *Essential Client/Server Survival Guide*. Van Nostrand Reinhold, New York, NY, 1994.

[Perrochon and Fischer, 1995] L. Perrochon and R. Fischer. IDLE: Unified w3-access to interactive information servers. In *Electronic Proc. 3rd Int. World Wide Web Conference*, Darmstadt, 1995.

[Pinakis, 1991] James Pinakis. A distributed typeserver and protocol for a Linda tuple space. Tr, Department of Computer Science, University of Western Australia, Nedlands WA 6009, 1991.

[Postel, 1982] J. Postel. Simple mail transfer protocol. Request for Comments (Standard) RFC 821, Internet Engineering Task Force, August 1982. Obsoletes RFC0788.

[Ragsdale, 1991] Susanne Ragsdale, editor. *Camelot and Avalon*. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Robinson, 1996] D. Robinson. The WWW Common Gateway Interface Version 1.1. Internet draft, Internet Engineering Task Force, http://www.ast.cam.ac.uk/ďrtr/cgi-spec.html, Jan 1996.

[Schönfeldinger, 1995] W. J. Schönfeldinger. WWW Meets Linda: Linda for global WWW-based transaction processing systems. In *Electronic Proc. 4th Int. World Wide Web Conference "The Web Revolution"*, Boston, MA, December 1995.

[Sci, 1995] Scientific Cumputing Associates, One Century Tower, 265 Church Street, New Haven, CT, USA. *Linda User's Guide & Reference Manual, V3.0*, January 1995.

[Siegel and Cooper, 1991] Ellen H. Siegel and Eric C. Cooper. Implementing distributed Linda in standard ML. Technical Report 151, Carnegie-Mellon, Department of Computer Science, October 1991.

[Stevenson, 1885] Robert L. Stevenson. *A Child's Garden of Verses*. 1885.

[Sutcliffe and Pinakis, 1992] G. Sutcliffe and J. Pinakis. Prolog-D-Linda. Technical Report TR91/7, Dept. of CS, University of Western Australia, Nedlands, Western Australia, 1992.

[Tanenbaum, 1994] Andrew S. Tanenbaum. A comparism of three microkernels. Technical report, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, Netherlands, 1994.

[Taudes, 1991] Alfred Taudes. *Ressourcenmanagement in verteilten Rechnersystemen*. Wirtschaftsuniversität Wien, Augasse 2–6, A-1090 Wien, 1991.

[Thomas, 1991] Owen Thomas. A Linda kernel for UNIX networks. In Wilson [1991b], pages 124–128.

[van Hoff *et al.*, 1995] Arthur van Hoff, Sami Shaio, Orca Starbuck, and Sun Microsystems Inc. *Hooked on Java*. Addison-Wesley, Reading, MA, December 1995.

[Wall and Schwartz, 1990] Larry Wall and Randal L. Schwartz. *Programming in Perl*. O'Reilly and Associates Inc., Cambridge, MA, 1990.

[Wilson, 1991a] Greg Wilson. Improving the performance of generative communication systems by using application-specific mapping functions. In *Linda-Like Systems and Their Implementation* [1991b], pages 129–142.

[Wilson, 1991b] Greg Wilson, editor. *Linda-Like Systems and Their Implementation*, number 91–13 in TR, Edinburgh, UK, June 1991.

A Source-Code zum Perl-Linda Server

A.1 Server – Hauptprogramm

```
#!/usr/local/bin/perl

# ************************************************************
# Perl Linda Server V.0.74 ***********************************
# ************************************************************
# The Perl Linda Server is a program which manages a Linda Tuple Space
#
# Written by Werner J. Schoenfeldinger, 1994-1995.
#
# Address: Dep. of Applied Computer Science                    10
# Vienna University of Economics
# Augasse 2-6
# A-1090 Vienna-Austria
# Email: schoenf
# Copyright (c) 1994-96. All rights reserved.
#
# See general license below.
#
# ************************************************************
# General License Agreement and Lack of Warranty ***************  20
# ************************************************************
#
# This software is distributed in the hope that it will be useful
# but WITHOUT ANY WARRANTY. The author does not accept res-
# ponsibility to anyone for the consequences of using it or for whether it
# serves any particular purpose or works at all. No warranty is made about
# the software or its performance.
#
# Use and copying of this software and the preparation of derivative
# works based on this software are permitted, so long as the following   30
# conditions are met:
#       o The copyright notice and this entire notice are included intact
#         and prominently carried on all copies and supporting documentation.
#       o No fees or compensation are charged for use, copies, or
#         access to this software. You may charge a nominal
#         distribution fee for the physical act of transferring a
#         copy, but you may not charge for the program itself.
#       o If you modify this software, you must cause the modified
```

```
#       file(s) to carry prominent notices (a Change Log)
#       describing the changes, who made the changes, and the date
#       of those changes.
#      o Any work distributed or published that in whole or in part
#       contains or is a derivative of this software or any part
#       thereof is subject to the terms of this agreement. The
#       aggregation of another unrelated program with this software
#       or its derivative on a volume of storage or distribution
#       medium does not bring the other program under the scope
#       of these terms.
#
# This software is made available AS IS, and is distributed without
# warranty of any kind, either expressed or implied.
#
# In no event will the author(s) or their institutions be liable to you
# for damages, including lost profits, lost monies, or other special,
# incidental or consequential damages arising out of or in connection
# with the use or inability to use (including but not limited to loss of
# data or data being rendered inaccurate or losses sustained by third
# parties or a failure of the program to operate as documented) the
# program, even if you have been advised of the possibility of such
# damages, or for any claim by any other party, whether in an action of
# contract, negligence, or other tortious action.
#
# Please send bug reports, comments, questions and suggestions to
# schoenf
# or improvements you may make.

require "linda-serv.pl";   # loading the library
require "linda-func.pl";   # loading the linda-functions

$port=$ARGV[0];

$_listpattern="[^,]+(,[^,]+)*";
$port= 7999 unless $port;

print "starting on port $port.\n";
chdir "/usr/tmp"; # WJS for security

&register_serv("CLIENT", $port);  # make a server socket on port 7999

&run_serv();      # dispatching (never returns)
```

```
sub serv_body {
    # $_[0] : request data sent from a client
    # $_[1] : socket name corresponding to the current client
    local($req, $sock)=@_;
    local($reply)="Exception State\n";

    $_=$req;
    &log("C: $sock R: $req",0);
                                                                    90
    &out($1,$sock), $reply="" if /^\s*out\s*\(($_listpattern)\)\s*$/;
    &uout($1,$sock), $reply="" if /^\s*uout\s*\(($_listpattern)\)\s*$/;
    $reply=&raw_rd($1,$sock,1,0) if /^\s*rd\s*\(($_listpattern)\)\s*$/;
    $reply=&raw_rd($1,$sock,0,0) if /^\s*nbrd\s*\(($_listpattern)\)\s*$/;
    $reply=&raw_rd($1,$sock,0,0) if /^\s*rdp\s*\(($_listpattern)\)\s*$/;
    $reply=&raw_rd($1,$sock,1,1) if /^\s*ard\s*\(($_listpattern)\)\s*$/;
    $reply=&uin($1,$sock) if /^\s*uin\s*\(($_listpattern)\)\s*$/;
    $reply=&raw_in($1,$sock,1,0) if /^\s*in\s*\(($_listpattern)\)\s*$/;
    $reply=&raw_in($1,$sock,0,0) if /^\s*nbin\s*\(($_listpattern)\)\s*$/;
    $reply=&raw_in($1,$sock,0,0) if /^\s*rdp\s*\(($_listpattern)\)\s*$/;   100
    $reply=&raw_in($1,$sock,1,1) if /^\s*ain\s*\(($_listpattern)\)\s*$/;
    $reply=&status if /^\s*status\s*\(([\w\.]*)\)\s*$/;
    &ts_load($1), $reply="" if /^\s*load\s*\(([\w]*)\)\s*$/;
    &ts_save($1), $reply="" if /^\s*save\s*\(([\w]*)\)\s*$/;
    &ts_merge($1),$reply="" if /^\s*merge\s*\(([\w]*)\)\s*$/;
    &self_close($sock), $reply=$_notfound if /^\s*bye\s*\(([\w\.]*)\)\s*$/;
    $reply=&lst() if /^\s*lst\s*\(?\)?/;
    &shutdown_serv(), $reply="" if /^kill.*/;
    &cleanup_serv(), $reply="" if /^cleanup.*/;
    $reply;                                                         110
}
```

A.2 Server – Linda-Funktionen

```
######
# linda-func.pl
######
# This File is part of the PERL LINDA SERVER distribution
# V 0.74
#
# See complete license in the file 'lindaserver'
#
# Written by Werner J. Schoenfeldinger, 1994-1995.
```

```
#                                                                    10
# Address: Dep. of Applied Computer Science
# Vienna University of Economics
# Augasse 2-6
# A-1090 Vienna-Austria
# Email: schoenf
# Copyright (c) 1994-96. All rights reserved.
#
# Changelog:
# 22.6.95 fixed 2x'uin' bug
# fixed out-uin bug                                                  20


$LOGFILE="server.log";
$LOGLEVEL=13;  # Level of Loggings higher is fewer messages

sub log
{
        local($msg,$level)=@_;
        return if $level < $LOGLEVEL;
                                                                     30
        open(LOG,">>$LOGFILE");
        print LOG "L $level: $msg\n";
        close(LOG);
}


# DEFINITIONS
@ts=();  # tuplespace
@wq=();  # waiting_queue
@upd=();  # update queue                                            40
%updmap=();  # list of pending updates only
$_notfound="___not_found___";

sub get_pattern  # gets the pattern for rd and out
{
        local($src)=@_;
        local($pattern)="^";
        local(%patequ,  $equcnt,$equpat);
        local($more_than_one)=0;
        local(@tuple)=split($_delimit,$src);                         50

        $equcnt=1;
```

```
for (@tuple)
{
# ? is search for pattern
# ?* is more than one pattern

$pattern.=$_delimit if $more_than_one;
$more_than_one++;

if(/\s*(\w*)\?([\*\w]*)\s*/)
{
      if($2)
      {
      if($2 eq "*")
      {
            &log("We got a Joker",1);
            $pattern.=".+";
      }
      else
      {
            if($patequ{$2})
            {
            $equpat=$patequ{$2};
            $pattern.="$1\\".$equpat;
            }
            else
            {
            $patequ{$2}=$equcnt;
            $equpat="";
            $equcnt++;
            $pattern.="$1([^,]+)";
            }
      }
      }
      else
      {
      $pattern.="$1([^,]+)";
      }
}
else
{
      $_=~s/\*/\\*/g;
      $pattern.="$_";
}
```

```
        }
        $pattern.="\$";
        &log("Pattern: $pattern",1);
        $pattern;
}                                                                    100

sub recoverts
{
        local($sock)=$_[0];
        local($i);
        local($notfound)=1;
        local($tuple);
        local($reply);

        &log("Recoverts: Queue, $#upd, [@upd]",2);                   110
        for($i=0; $i<=$#upd && $notfound; $i++)
        {
                ($cli,$tuple)=split(/\n/,$upd[$i]);
                &log("recoverts: Whole-($upd[$i])",2);
                &log("recoverts: $cli, ($tuple)",3);

                $del=$i,
                $notfound=0 if $sock eq $cli;
        }
        &log("Have recovered Client $sock, tuple ($tuple)",5);       120
        splice(@upd,$del,1) if $del >= 0;
        $updmap{$sock}=0;
        $reply=&out($tuple,$sock);
        $reply;
}




sub out                                                              130
{
        # adds a tuple to the LINDA space
        local($tuple, $sock)=@_;
        local($count)=0;
        local(@del)=();
        local($notin)=1;
        local($insert)=1;  # default is to insert the tuple.
```

```
        &recoverts($sock) if $updmap{$sock}==1;
                                                                        140
        for(@wq)
        {
        ($client,$cmd,$req)=split(/\n/,$_);
        &log("out_wq: C:$client,T:$cmd,R:$req.",1);
        if($notin && $tuple=~/$req/)
        {
              @del=($count,@del);
              &log("send_wq: R:$req,T:$tuple.",1);
              print $client "$tuple\n.\n"; # respond
              if($cmd eq "uin")                                         150
              {
              &log("logging upd of wq: C:'$client', T:'$tuple'",3);
              @upd=(@upd,join("\n",($client,$tuple)));
              $updmap{$client}=1;
              }
              $notin=0, $insert=0, last if
              $cmd eq "in"
                    || $cmd eq "ain"
                    || $cmd eq "uin";
        }                                                               160
        $count++;
        }
        for (@del) { splice(@wq,$_,1);} # extract the tuples
        &log("added: T:$tuple",1), @ts=(@ts,$tuple) if $insert;
}


sub client_dead
{
        local($sock)=$_[0];                                             170
        local(@del)=();
        local(@rest);
        local($i)=0;

        for($i=0;$i<=$#wq;$i++)
        {
        ($client,@rest)=split("\n",$wq[$i]);
              @del=($i,@del) if $client eq $sock;
        }
        &recoverts($sock), $updmap{$sock}=0 if $updmap{$sock}==1;       180
        for (@del) { splice(@wq,$_,1);}
```

```
}

sub raw_rd
{
        local($pattern)=&get_pattern($_[0]);
        local($sock)=$_[1];
        local($block)=$_[2];
        local($atomic)=$_[3];
        local($reply)="";                                                    190
        local($notfound)=1;

        &recoverts($sock) if $updmap{$sock}==1;


        for($i=0; $i<=$#ts && ($notfound || ($atomic==0)); $i++)
        {
                $reply.=$ts[$i]."\n",
                $notfound=0
                        if $ts[$i]=~/$pattern/;                              200
        }
        $reply=$_notfound,
        @wq=(join("\n",($sock,($atomic?"ard":"rd"),$pattern)),@wq)
                if $notfound && $block;
        $reply;
}

# extracts a tuple from the tuplespace
sub raw_in
{                                                                            210
        # first parameter Pattern
        # 2nd paramenter Socket
        # 3rd parameter blocking ?
        # 4th parameter Atomic ?
        local($pattern)=&get_pattern($_[0]);
        local($sock)=$_[1];
        local($block)=$_[2]; # 0.. non block, 1..block
        local($atomic)=$_[3]; # 1 is ain
        local($reply)="";
        local(@del)=();                                                      220
        local($notfound)=1;
        local($i);

        &recoverts($sock) if $updmap{$sock}==1;
```

```
        for($i=0; $i<=$#ts && ($notfound || ($atomic==0)); $i++)
        {
        $reply.=$ts[$i]."\n",
        $notfound=0,
        @del=($i,@del) if $ts[$i]=~/$pattern/;                                  230
        }

        for (@del) { splice(@ts,$_,1);}
        $reply=$_notfound,
        @wq=(@wq, join("\n",($sock,($atomic?"ain":"in"),$pattern)))
        if $notfound && $block;
        $reply;
}

                                                                                240

sub uin{
        # first parameter Pattern
        # 2nd paramenter Socket
        # uin is always blocking
        local($pattern)=&get_pattern($_[0]);
        local($sock)=$_[1];
        local($reply)="";
        local(@del)=();
        local($notfound)=1;                                                     250
        local($i);

        &recoverts($sock) if $updmap{$sock}==1;

        for($i=0; $i<=$#ts && $notfound; $i++)
        {
        $reply.=$ts[$i]."\n",
        $notfound=0,
        @del=($i,@del),
        $updmap{$sock}=1,                                                       260
        @upd=(@upd,join("\n",($sock,$ts[$i])))
                if $ts[$i]=~/$pattern/;
        }

        for (@del) { splice(@ts,$_,1);}
        $reply=$_notfound,
        @wq=(@wq, join("\n",($sock,"uin",$pattern)))
```

```
        if $notfound;
        $reply;
}

sub status
{
        local($reply);
        local($nl)="\n";
        $reply.="STATUS Information$nl-------------------------$nl";
        $reply.="Tuplespace=\t".($#ts+1)."\tTuples$nl";
        $reply.="WaitingQueue=\t".($#wq+1)."\tTuples$nl";
        $reply.="UpdateQueue=\t".($#upd+1)."\tTuples$nl";
        $reply;
}

sub uout{
        local($pattern)=$_[0];
        local($sock)=$_[1];
        local($reply)="";
        local($notfound)=1;
        local($del)=-1;
        local($rest);
        local($i);

        for($i=0; $i<=$#upd && $notfound; $i++)
        {
        ($cli,$rest)=split(/\n/,$upd[$i]);
        &log("uout: Processing upd: S:'$sock', C:'$cli'",3);
        $del=$i, $notfound=0 if $sock eq $cli;
        }
        splice(@upd,$del,1) if $del >= 0;
        $updmap{$sock}=0;
        &out($pattern,$sock);
        $reply;
}

# cleans Tuplespace & WQ
sub cleanup_serv
{
        local($client,$cmd,$req);
        @ts=();
        for(@wq)
        {
```

```
        ($client,$cmd,$req)=split(/\n/,$_);
        &log("cleanup: C:$client,T:$cmd,R:$req.",1);
        print $client "reset\n.\n"; # respond (reset)
        }
        @wq=();
        @upd=();
        %updmap=();
}


sub lst # lists the tuplespace
{
        "tuplespace:\n-------------\n(".
        join(")\n(",@ts).
             ")\n-------------\n";
}

1; # to make require happy
```

A.3 Server – Administrative Funktionen

```
######
# This File is part of the PERL LINDA SERVER distribution
# See License in the file 'server'
#
######
#
# EASY-SERV.PL
# Copyright : Software Research Associates, Inc. 1990
#       Written by Hiroshi Sakoh (sakoh
#       Please distribute widely, but leave my name here.
#
# Modified and partly rewritten by
# Werner J. Schoenfeldinger in the year 1994, 95
# V0.74
#
# Changelog

require "sys/fcntl.ph";
require "linda-ipc.pl";

#firstfirst a view definitions
```

```
$_linesep="\n" if !defined($_linesep);
$_delimit="," if !defined($_delimit);


# the failure msg
$failure="__ERROR__";

sub read_string                                                    30
{
        local($sock)=$_[0];
        local($string);

        if(!eof($sock))  # ok, let's do it
        {
        $string=<$sock>;
        return $string;
        }
        else                                                       40
        {
        &log("read_string: EOF detected.\n",1);
        return $failure;
        }
}

#
# &register_serv($name, $port)
#
# Registers a service port.                                        50
#
sub register_serv {
        &defserver($_[0], $_[1]);
        fcntl($_[0], &F_SETFL, &FNDELAY)
        || die "fcntl: $!\n";
        $_eserv_generic{$_[0]} = $_[0];
        $_eserv_sockets{$_[0]} = $_[0];
}

#                                                                  60
# &run_serv()
#
# Wait for a request, dispatch it.
# Sends back a reply to the requester.
```

```
#
sub run_serv
{
    local($bytestoread, $packet,
      $actualread, $request, $reply);
    local($sock,$tmp);                                          70

    for (;;)
    {
    @avails =
        &selectsock(keys(%_eserv_sockets));
      nextavail:
    for(@avails)
    {
        $sock=$_;
        if (defined($_eserv_generic{$sock}))                    80
        {
        &acceptsock($_eserv_newsock,$sock);
        $_eserv_sockets{$_eserv_newsock} =
            $_eserv_newsock;
        &log("New: $_eserv_newsock",1);
        }
        else
        {
        $request=&read_string($sock);
        # returns $failure if it fails to read                  90
        if($request eq $failure)
        {
            &log("Client_dead: $sock",1);
            &client_dead($sock);
            close($sock);
            &delete_sock($sock);
            next nextavail;
        }
        else
        {                                                       100
            $reply="";
            $reply = &serv_body($request, $sock);
            print $sock $reply."\n"
            if $reply ne $_notfound;
        }
        }
    }
```

```
        }
    }

sub delete_sock
{
    local(%es);
    foreach(sort(keys(%_eserv_sockets)))
    {
    $es{$_}=$_eserv_sockets{$_} if $_[0] ne $_;
    }
    undef %_eserv_sockets;
    %_eserv_sockets=%es;
}

sub self_close
{
    local($sock)=@_;
    local(%es);
    &log("Client_self shutdown: $sock",1);

    &client_dead($sock);
    &delete_sock($sock);
    close($sock);
}

######
# &ts_save($fname);
#
$standard_fname="tuplespc.sav";

sub ts_save{
    local($fname)=@_;

    $fname=$standard_fname if $fname=~/.*[\/\.]+.*/;
    # we do not support hackers
    $fname=$standard_fname if $fname eq "";
    open(SAV,">$fname");
    for(@ts)
    {
    print SAV "$_\n";
    }
    close(SAV);
```

```
}

######
# &ts_load($fname);
#
$standard_fname="tuplespc.sav";

sub ts_load{
        local($fname)=@_;
                                                                    160
        &cleanup_serv();

        $fname=$standard_fname if $fname eq "";
        open(SAV,"<$fname");
        while(<SAV>)
        {
        chop;
        @ts=(@ts,$_);
        }
        close(SAV);                                                 170
}

######
# &ts_merge($fname);
#

sub ts_merge{
        local($fname)=@_;

        $fname=$standard_fname if $fname eq "";                     180
        open(SAV,"<$fname");
        while(<SAV>)
        {
        chop;
        @ts=(@ts,$_);
        }
        close(SAV);
}

#                                                                   190
# &shutdown_serv()
#
# Closes down all server sockets and exits.
```

```
#
sub shutdown_serv {
        for(keys(%_eserv_sockets)) {
        close($_);
        }
        for(keys(%_eserv_generic)) {
        close($_);
        }
        exit;
}

1;
```

A.4 Server – Inter Process Communication

```
######
# This File is part of the PERL LINDA SERVER distribution
# See License in the file 'lindaserver'
#
# The LINDA-IPC-PACKAGE is based on EASY-IPC from H. Sakoh
######
#
# EASY-IPC.PL
# Copyright : Software Research Associates, Inc. 1990
#       Written by Hiroshi Sakoh (sakoh
#       Please distribute widely, but leave my name here.
#
# V0.74
# Modified and partly rewritten by
# Werner J. Schoenfeldinger in the year 1994, 95

require "sys/socket.ph";

#
# &defserver(SOCKET, $port);
#
# Returns a server socket ready for acceptsock()
#
sub defserver { # arg0 == socket, arg1 == port
        local($this, $oldfh);
        local($port) = $_[1];
        local($name, $aliases, $proto);
```

```
        local($sockaddr_t) = 'S n a4 x8';

        ($name, $aliases, $port, $proto) =
        getservbyname($port, "tcp")
                unless $port =~ /^\d+$/;
        socket($_[0], &PF_INET, &SOCK_STREAM, 0)
        || die "socket: $!\n";
        $this = pack($sockaddr_t, &PF_INET, $port, "\0\0\0\0");
        bind($_[0], $this) || die "bind: $!\n";
        listen($_[0], 5);
        $oldfh = select($_[0]) ; $| = 1; select($oldfh);
}

$__EIPCSno__ = 'a';
%_eserv_sockets=();
%_eserv_generic=();

#
# &acceptsock($newsock, $serversock)
#
# Returns a bind socket which is derived from a generic server socket
#
sub acceptsock {
        local($addr);
        local($oldfh,$i);
        local($s,$sn);
        # $_[0] Name of socket
        # $_[1] name for clients of server
        $i=$__EIPCSno__;
        do{
        $s=$_[1]."__".$i;
        &log("Testing socket $s\n",2);
        $i++;
        }while(defined($_eserv_sockets{$s}));
        $_[0]=$s;
        $addr = accept($s, $_[1]) || die "accept: $!\n";
        $oldfh = select($s) ; $| = 1; select($oldfh);
        $s;
}

#
# &selectsock(
#
```

```
# Returns available sockets for read (blocks)
#
sub selectsock {
        &selectsock_with_timeout(undef, @_);
}


#
# &selectsock_non_block(
#                                                                    80
# Returns available sockets for read (doesnt block)
#
sub selectsock_non_block {
        &selectsock_with_timeout(0, @_);
}

#
# &selectsock_with_timeout($timeout,
#
# Returns available sockets for read (with timeout)                  90
#
sub selectsock_with_timeout {
        local($nfound, $timeleft);
        local($rin, $rout, $win, $wout, $eout, $ein);
        local($timeout) = shift;

        $rin = &setsockbits(@_);
        ($nfound, $timeleft) =
        select($rout=$rin, undef, undef, $timeout);
                                                                   100
        grep(vec($rout,fileno($_),1),@_);
}

#
# &setsockbits(
#
# Returns a bitvector corresponding to sockets list
#
sub setsockbits{
        local(@fhs) = @_;                                          110
        local($bits);
        for (@fhs) {
                vec($bits, fileno($_), 1) = 1;
```

```
        }
        $bits;
}

#
# &inet_addr($addr)
#
# Converts an internet dot notation (##.##.##.##) to an internet address
#
sub inet_addr {
        pack("C4", split(/\./, $_[0]));
}

1;
```

B Perl Client-Schnittstelle zum Perl-Linda-Server

```
##############################################################
# Utilities for a Linda-Client
######
# This File is part of the PERL LINDA SERVER distribution
# V 0.74
#
# See complete license in the file 'lindaserver'
#
# Written by Werner J. Schoenfeldinger, 1994-1995.
#                                                              10
# Address: Dep. of Applied Computer Science
# Vienna University of Economics
# Augasse 2-6
# A-1090 Vienna-Austria
# Email: schoenf
# Copyright (c) 1994-96. All rights reserved.
#

require "chat2.pl";
$_delimit=",";                                                 20
$_linesep="\n";
$in_timeout=5;
$out_timeout=30;

sub register_client
{
        local($host,$port)=@_;
        return &chat'open_port($host,$port);
}
                                                               30
sub close_client
{
        &chat'print("bye\n");
        &chat'close();
}

#
# tuple=&entrans(tuple)
#
sub entrans                                                    40
{
```

```
        local(@tuple)=@_;
        local($tmp)=0;

        for(@tuple)
        {
                s/:/:e/g;
                s/$_delimit/:d/g;
                s/$_linesep/:n/g;
                s/\./:p/g;
                $tuple[$tmp]=$_;
                $tmp++;
        }
        @tuple;
}


#
# tuple=&detrans_el(tuple)
#
sub  detrans_el
{
        local(@tuple)=@_;
        local($tmp)=0;

        for(@tuple)
        {
                s/:e/:/g;
                s/:d/$_delimit/g;
                s/:n/$_linesep/g;
                s/:p/\./g;
                $tuple[$tmp]=$_;
                $tmp++;
        }
        @tuple;
}

#
#
#
sub  detrans
{
        local(@tuple)=split($_delimit,$_[0]);
        local($tmp)=0;
```

```
        for(@tuple)
        {
                s/:e/:/g;
                s/:d/$_delimit/g;
                s/:n/$_linesep/g;                                    90
                s/:p/\./g;
                $tuple[$tmp]=$_;
                $tmp++;
        }
        @tuple;
}

#
# update($tuple,
#                                                                    100
sub  update
{
        local($tuple,@pattern);

        &uin(@pattern);
        &uout(&detrans($tuple));
}



                                                                    110
#
# $var=&element($row, $column,
#
# $row & $col starts with 1;
sub  element
{
        local($row,$col,@tuple)=@_;
        local(@currow,$elem);

        $row=$#tuple+1  if  $row  eq  "last";                       120
        if($row<1  ||  $col<1)
        {
             print  "Row & Col must be greater than 1.";
        }

        if  ($#tuple  <  ($row-1))
        {
```

```
            print "Tuple has only ".($#tuple - 1).
                " rows, but you requested $row.";
            return "";
        }

        @currow=&detrans($tuple[$row-1]);

        if($#currow < ($row-1))
        {
            print "Tuple has only ".
                ($#currow + 1)." cols, but you requested $col.";
            return "";
        }

        $elem=$currow[$col-1];
        $elem;
}


sub printtuple
{
        local($first)=0;

        print "Tuple:";

        for(@_)
        {
        print " |" if $first;
        print " $_";
        $first=1;
        }
        print ".\n";
}


# Internal CMD
#
# tuple=&_retrieve("rd"|"in"|"ain",tuple);
#

sub _retrieve
{
        local($cmd,@tuple)=@_;
```

```
        local(@result)=();
        local($string);
        local($tmp)=0;
        local($fail);
        local($flag)=1;

        @tuple=&entrans(@tuple);

        $string="$cmd(".join($_delimit,@tuple).")\n";

        &chat'print($string);

        while($flag)
        {
        &chat'expect($in_timeout,
                    TIMEOUT, '$fail="TIMEOUT"',
                    EOF, '$fail="EOF"',
                    '^\.\r?\n', '$fail="", $flag=0',
                    '^(.*)\r?\n', '@result=(@result,$1)');
        }

        # result=&detrans(result);
        # it's better to leave it to the user;

        @result;
}


##############################################
# The LINDA-CMDS
# &out(
#

sub  out{
        &rawout("out",@_);
}

sub  uout{
        &rawout("uout",@_);
}

sub  rawout
{
```

```
        local($cmd,@tuple)=@_;
        local($string);
        local($tmp)=0;
        local($fail);


        @tuple=&entrans(@tuple);                                    220

        $string="$cmd(".join($_delimit,@tuple).")\n";

        &chat'print($string);
        &chat'expect($in_timeout,
                TIMEOUT, '$fail="TIMEOUT"',
                EOF, '$fail="EOF"',
                '^\.\r?\n', '$fail=""');
}
                                                                    230
#
# resultlist=&in(tuple);
#
#
# the tuples are still in 'encoded' format. To decode its advisable
# to split the
#
# This can be accomplished by the following code.
# There are 2 possiblilities
#                                                                   240
# A)
# for(
# {
#   # split the tuple in the parts.
# tuple=&detrans_el(tuple);
# # now you can do something with the tuple
# # ...
# }
#
# B)                                                                250
# for(
# {
#
# # now you can do something with the tuple
# # ...
# }
```

```
sub in
{
        local(@tuple)=&_retrieve("in",@_);
        @tuple;
}

sub uin
{
        local(@tuple)=&_retrieve("uin",@_);
        @tuple;
}


sub ain
{
        local(@tuple)=&_retrieve("ain",@_);
        @tuple;
}

sub nbin
{
        local(@tuple)=&_retrieve("nbin",@_);
        @tuple;
}

sub rd
{
        local(@tuple)=&_retrieve("rd",@_);
        @tuple;
}

sub nbrd
{
        local(@tuple)=&_retrieve("nbrd",@_);
        @tuple;
}

sub bye{
        &chat'print("bye\n");
}

1;  # to make require happy
```

300

C Source-Code zur Bankomat Simulation

C.1 Interface-Client

```
#!/usr/local/bin/perl
######################
# ATM.PL
######################
# Interface - Client for the ATM-Simulation
# (c) by Werner Schoenfeldinger 1996
# schoenf
######################

require "linda-cli.pl";

%errors=(
      "PASS","Wrong username or password",
      "LIMIT","amount exceeds daily limit",
      "ACC","amount exceeds balance");

# usage : if.pl <name> <passwd> <amount>

if  ($#ARGV<2)
{
     print "usage : if.pl <name> <passwd> <amount>\n";
     exit;
}


$name=$ARGV[0];
$pswd=$ARGV[1];
$amount=$ARGV[2];

&register_client("aid.wu-wien.ac.at",7999)
     || die "Cannot connect to TS\n";


&out("ATM",$name,$pswd,$amount,"PASS");
@tl=&in("ATM",$name,"?",$amount,"?","?");
($du,$user,$pswd,$amount,$op,$res)=
      &detrans($tl[0]);
```

```
if($res eq "DONE")
{
    print "You got $amount Coins!!\n";
}
else
{
    print "Sorry, we had this error: ".$errors{$op}."\n";
}

&close_client();
```

C.2 Indentifikations-Client

```
#!/usr/local/bin/perl
######################
# PSWD.PL
######################
# Identification - Client for the ATM-Simulation
# (c) by Werner Schoenfeldinger 1996
# schoenf
######################

require "linda-cli.pl";

%users=("schoenf","ABC",
    "christof","NO");


# Protocol
# (ATM,user,pswd,amount,OPERATION)
# Transfers
# op: PASS to LIMITGET

&register_client("aid.wu-wien.ac.at",7999);
while(1)
{
    @tl=&ain("ATM","?","?","?","PASS");
    ($du,$user,$pswd,$amount,$op)=
    &detrans($tl[0]);


    if(!defined $users{$user} || $users{$user} ne $pswd)
```

```
        {                                                              30
        &out($du,$user,$pswd,$amount,"PASS","FAIL");
        }
        else
        {
        &out($du,$user,"checked",$amount,"LIMITGET");
        }
}
```

C.3 Limit-Client

```
#!/usr/local/bin/perl
######################
# LOG.PL
######################
# Logging & Limit - Client for the ATM-Simulation
# (c) by Werner Schoenfeldinger 1996
# schoenf
######################

require "linda-cli.pl";                                                10

%users=();

$DAYLIMIT=5000;
# Protocol
# (ATM,user,pswd,amount,OPERATION)

&register_client("aid.wu-wien.ac.at",7999);
while(1)
{                                                                      20
        @tl=&ain("ATM","?","?","?","LIMIT?");
        ($du,$user,$pswd,$amount,$op)=
        &detrans($tl[0]);

        $users{$user}=$DAYLIMIT
        if !(defined $users{$user});

        if($op eq "LIMITGET")
        {
        if($amount > $users{$user})                                    30
        {
```

```
            &out($du,$user,$pswd,$amount,"LIMIT","FAIL");
        }
        else
        {
            &out($du,$user,$pswd,$amount,"ACC");
        }
        }
        else
        {
        $users{$user}-=$amount;
        &out($du,$user,$pswd,$amount,"LIMIT","DONE");
        }
}
```

C.4 Konten-Client

```
#!/usr/local/bin/perl
######################
# ACC.PL
######################
# Accounting - Client for the ATM-Simulation
# (c) by Werner Schoenfeldinger 1996
# schoenf
######################

require "linda-cli.pl";

# The accountholders
%users=("schoenf",10000,
    "chris",10000);


# Protocol
# (ATM,user,pswd,amount,OPERATION)
# Transfers
# op: PASS to LIMITGET

&register_client("aid.wu-wien.ac.at",7999);
while(1)
{
    @tl=&ain("ATM","?","?","?","ACC");
    ($du,$user,$pswd,$amount,$op)=
```

```
        &detrans($tl[0]);


        if($amount>$users{$user})
        {
        &out($du,$user,$pswd,$amount,"ACC","FAIL");
        }
        else
        {
        &out($du,$user,$pswd,$amount,"LIMITLOG");
        }
}
```

D Source-Code zu Parallel Povray

```
#!/usr/local/bin/perl
# PARALLEL POVRAY
#
# Written by Werner Schoenfeldinger in the year 1995
# schoenf
#
# USAGE: par_pov.pl host_of_TS Directory_of_POVRAY_files
#

require "linda-cli.pl";

$host=$ENV{"HOST"};
$shost=substr($host,0,3);
$path="/mount/".$shost.$ENV{"PWD"};

print "Starting host $host, shost $shost, path $path\n";

&register_client("aih.wu-wien.ac.at", 7999);

sub start
{
        local(@others);

        # Tell them I'm here
        &out("online", $host);

        @others=&rd("online","?");

        print "Others: $#others\n";

        if($#others == 0)
        {
        # I'm the first one here, so lets do the raw work;
        &out("maxjob",0);  # the last job
        &out("currjob",0);  # the last running job
        }
        # "status" "host" "doing" "jobnr"
        &out("host",$host,"idle",0);
        print "Connected to the server";
}
```

```
sub feed_jobs
{
        local(@jobs)=();
        local($jobnr);
        local($target);

        open(IN, "ls *.pov |");

        while(<IN>)                                                    50
        {
        chop;
        @jobs=();

                # "job" "jobnr" "status" "source" "result"
        # at this state we assume that there
        # are different jobs on every client
        $jobnr=&element(1,2,&in("maxjob","?"));
        print "Old Jobnr: $jobnr\n";
        $jobnr++;                                                      60
        # create a new job
        ($target,$nil)=split(/\./,$_);
        $target.=".tga";
        print "Adding job $jobnr, $path, $target/$_\n";
        &out("job",$jobnr,"todo","$path/$_",$target);
        &out("maxjob",$jobnr);
        }

        print "finished Job-feed\n";
}                                                                      70

sub ex_jobs
{
        local($currjob,  $maxjob);
        local($file,$target);
        local(@job);

        # look what job is to do.
        while(1)
        {                                                              80
        $currjob=&element(1,2,&in("currjob","?"));
        $maxjob=&element(1,2,&rd("maxjob","?"));

        print "Currjob $currjob, Maxjob $maxjob\n";
```

```
        if($currjob>=$maxjob)
        {
            #There There is nothing more to do
            &out("currjob",$currjob);
            &out("host",$host,"?","?");                                    90
            print "Shutting down Client\n";
            return;
        }

        $currjob++;

        @job=&in("job",$currjob,"todo","?","?");

        if($#job>0)
        {                                                                  100
            print "There seem to be 2 jobs with the same JobId ??\n";
            return;
        }

        $file=&element(1,4,@job);
        $target=&element(1,5,@job);

        print "To process: Job $currjob, $file, $target\n";

        &out("currjob",$currjob);                                          110
        &in("host",$host,"?","?");
        &out("host",$host,"processing",$currjob);

        system("/mount/aih/usr/logins/schoenf/bin/mkpov 200 $file\n");


        &in("host",$host,"?","?");
        &out("job",$currjob,"done",$file,$path.$target);
        &in("host",$host,"idle",0);
        }                                                                  120
}


&start();
&feed_jobs();
&ex_jobs();
```