**Diplomarbeit**

# Entwurf und Implementation eines verteilten Laufzeitsystems für ProSet-Linda


Knuth Waltenberg


Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

6. Mai 1996

## Gutachter:

Dr. W. Hasselbring
Prof. Dr. E.-E. Doberkat

Es liegt mir sehr am Herzen mich bei allen zu bedanken, die mir bei der vorliegenden Arbeit geholfen haben, vor allem bei meinem Betreuer Dr. Wilhelm Hasselbring. Weiter seien aber auch Peter Jodeleit, Marcus Kirsch, Wolfgang Emmerich, Helmut Henning, Melanie Everz und Holger Apel genannt, die mich mit großen und kleinen Ratschlägen unterstützten. Zuletzt möchte ich mich bei Britta Meiering für die Durchsicht des Skripts und der Hilfe in nicht-informatischen Dingen bedanken.

# Inhaltsverzeichnis

## Zusammenfassung

ProSet ist eine mengenorientierte Prototypingsprache [Doberkat *et al.*, 1990a]. Linda ist ein Koordinationssprachenkonzept zur Synchronisation und Kommunikation von parallelen Prozessen, basierend auf einem gemeinsamen Speicher, einem s.g. *Tupelraum* [Gelernter, 1985]. ProSet-Linda bildet die Vereinigung beider Ansätze, wobei das Tupelraumkonzept von Linda durch mehrfache (multiple) Tupelräume erweitert wurde [Hasselbring, 1994]. Eine erste quasi-parallele Implementierung eines Laufzeitsystems für ProSet-Linda wurde mit Leichtgewichtsprozessen realisiert.

Diese Arbeit beschreibt den Entwurf und die Implementierung eines neuen verteilten Laufzeitsystems für ProSet-Linda auf einem LAN mit Hilfe des Message-Passing-Systems *Parallel Virtual Machine* [Geist *et al.*, 1994]. Insbesonders wird untersucht, inwieweit sich der Ansatz der multiplen Tupelräume in bezug auf das Laufzeitverhalten auswirkt.

# 1. Einleitung

## 1.1 Motivation

Die Entwicklung von parallelen Programmen stellt Programmierer vor große Probleme, denn die Nebenläufigkeit von Prozessen ist durch den Menschen sehr schwer erfaßbar.

Eine Methode der Software-Entwicklung, die bei der Programmierung von parallelen Programmen helfen will, ist das Algorithmen-Prototyping. Durch diese Methode wird in kurzer Zeit ein Prototyp des zu verwendenden Algorithmus erstellt und dieser evaluiert. Der Ansatz sollte durch eine Programmiersprache unterstützt werden, die ein hohes Ausdrucksvermögen besitzt, um die schnelle Erstellung des Prototyps zu ermöglichen.

Eine Prototyping-Sprache, die diesen Ansatz unterstützen möchte, ist ProSet-Linda. ProSet-Linda ist eine Kombination aus der mengenorientierten Breitbandsprache ProSet [Doberkat *et al.*, 1990a] und dem Koordinationssprachkonzept Linda [Gelernter, 1985], das zur Kommunikation und Synchronisation von nebenläufigen Prozessen dient. Die Kommunikation zwischen den Prozessen findet bei Linda über einen virtuellen gemeinsamen Speicher, dem Tupelraum, statt. Dieses Konzept wurde für ProSet-Linda um multiple Tupelräume erweitert [Hasselbring, 1994].

Am Lehrstuhl X des Fachbereichs Informatik der Universität Dortmund wurde ein erstes quasi-paralleles Laufzeitsystem für ProSet-Linda implementiert, wobei die nebenläufigen Prozesse durch Leichtgewichtsprozesse (LWP) und der Tupelraum durch den gemeinsamen Speicher der LWPs verwirklicht wurde.

In dieser Arbeit wird nun der Entwurf und die Implementierung eines zweiten parallelen Laufzeitsystems für lokale Netze beschrieben. Während der Arbeit mußten folgende Fragestellungen gelöst werden:

- Welche Techniken und Heuristiken sind geeignet, die Tupelräume sinnvoll zu verteilen?

- Wie können die nebenläufigen Kontrollflüsse, die in ProSet durch Prozeduren repräsentiert werden, in eigenständige Prozesse umgewandelt und im Netz gestartet werden?
- Führt das Konzept der multiplen Tupelräume zu einer Performance-Steigerung gegenüber der Verwendung eines einzelnen Tupelraumes?

## 1.2 Übersicht

Die vorliegende Arbeit gliedert sich in fünf Teile. Im ersten Teil wird eine kurze Einführung in Konzepte von ProSet-Linda und dem verwendeten Message-Passing-System PVM gegeben. Danach werden die Vorgaben dieser Arbeit erläutert. Im zweiten Teil werden dann die drei großen Problembereiche „verteiltes Tupelraum-Management", „entfernte Prozeßkreation" und „Loadbalancierung" beschrieben und Lösungen diskutiert. Im dritten Teil folgt die Beschreibung und Spezifikation der Funktionen und Protokolle. Im vierten Teil wird dann auf die Implementierungsphase dieser Arbeit eingegangen, bevor im letzten Teil die Ergebnisse der Laufzeittests vorgestellt werden und ein Fazit gezogen wird.

In den Anhängen befindet sich eine kurze Einführung in die Ethernet-Technologie und der Quelltext zweier Testprogramme. Zuletzt ist der gesamte Quelltext des neuen Laufzeitsystems für ProSet-Linda abgedruckt.

# Teil I

# Grundlagen und Vorgaben

# 2. Einführung

## 2.1 Die Prototyping-Sprache ProSet

Die Prototyping-Sprache, um deren verteiltes Laufzeitsystem es in dieser Arbeit geht, heißt ProSet. ProSet wird momentan am Lehrstuhl X des Fachbereichs Informatik der Universität Dortmund entwickelt.

Der Name „ProSet" weist auf eine wichtige Eigenschaft dieser Prototyping-Sprache hin, denn er ist die Abkürzung für „Prototyping with Sets". ProSet basiert also auf der mathematischen Mengenlehre und bietet u.a. einen Persistenzmechanismus, Ausnahmebehandlung (exception handling) und Module an. Der Sprachkern wurde zuerst in [Doberkat *et al.*, 1990a] vorgestellt, das System in [Doberkat *et al.*, 1990b]. Vorgänger von ProSet war SETL, deshalb wurde ProSet zuerst SETL/E genannt.

Nun soll kurz auf die besonderen Datentypen *set* und *tuple* eingegangen werden, da diese in dieser Arbeit eine wichtige Rolle spielen. Alle anderen oben genannten Konzepte von ProSet werden in [Doberkat *et al.*, 1992] beschrieben.

Der ProSet-Datentyp *set* entspricht der Menge in der mathematischen Mengenlehre. Die Elemente eines *set*s können von den ProSet-Datentypen *atom, integer, real, string, boolean* sein, aber auch wieder *set* oder *tuple*. Wie in der Mengenlehre besitzen die Elemente in *sets* keine festgelegte Reihenfolge. Mengen werden in der ProSet-Syntax mit geschweiften Klammern dargestellt. Ein Beispiel für ein Datum vom Typ *set* ist:

```
{"Hallo",123,45}
```

Der Datentyp *tuple* entspricht dem *set*-Datentyp. Die Unterschiede bestehen darin, daß die Reihenfolge der Elemente bei diesem Typ festliegen und mehrfache Kopien eines Elementes erlaubt sind. Auf die Elemente kann deshalb auch über die Angabe ihrer

Position im Tupel zugegriffen werden. Tupel werden in der ProSet-Syntax mit eckigen Klammern versehen. Ein Beispiel:

```
["Hallo",123,45]
```

## 2.2 Linda

Linda ist ein Koordinationssprachenkonzept zur parallelen Programmierung [Gelernter, 1985]. Es dient zur Synchronisation und Kommunikation von parallelen Prozessen und ist hardware-unabhängig.

Die Synchronisation und Kommunikation von parallelen Prozessen wird in Linda mit Hilfe eines virtuellen gemeinsamen Speichers aller parallelen Prozesse verwirklicht. Dieser gemeinsame Speicher wird *Tupelraum* (TR) genannt, eben weil in diesem „Raum" Tupel abgelegt werden. Die Tupel in Linda entsprechen dem gleichnamigen, in Kapitel 2.1 auf Seite 5 beschriebenen ProSet-Datentyp. Der Zugriff auf diese erfolgt assoziativ, d.h. nicht über Adressen, sondern anhand von Eigenschaften der gesuchten Tupel.

Um die Eigenschaften der gesuchten Tupel zu beschreiben, werden sogenannte *Templates* angegeben. Diese werden im Kapitel 2.3.1 auf Seite 7 genauer beschrieben.

Das Linda-Konzept kann theoretisch mit fast jeder Programmiersprache kombiniert werden. Die erste Programmiersprache, in die Linda integriert wurde, war C [Gelernter, 1985]. Seitdem wurden aber schon viele andere Sprachen mit Linda kombiniert [Wilson, 1991]. In [Hasselbring, 1991] wurde die Kombination ProSet-Linda zum ersten mal vorgestellt, in [Hasselbring, 1994] genau spezifiziert. Mehr dazu im nächsten Kapitel.

## 2.3 ProSet-Linda

Nachdem in den letzten Kapiteln recht kurz auf ProSet und Linda eingegangen worden ist, wird nun ProSet-Linda etwas genauer betrachtet. Auch hier kann nur das Wesentliche beschrieben werden. Für Einzelheiten sei auf [Doberkat *et al.*, 1992] verwiesen.

In ProSet-Linda ist die Prototyping-Sprache ProSet mit den Konzepten von Linda vereinigt worden. Dabei wurde der Ansatz der Tupelräume modifiziert.

### 2.3.1 Der Tupelraum und seine Operationen

Das Konzept des Tupelraumes aus Linda wurde in ProSet-Linda insoweit verändert, daß nicht nur `ein` Tupelraum, sondern mehrere Tupelräume benutzt werden können. Diese *multiplen Tupelräume* werden durch den eindeutigen Wert eines *Atoms* unterschieden. Die Werte des speziellen ProSet-Linda-Datentyps *Atom* werden durch die Funktion *newat()* generiert. Diese Werte sind in dem Sinne eindeutig, daß die Implementierung garantiert, daß ein neugenerierter Wert sich von allen bestehenden Atomwerten (in allen Prozessen) unterscheidet.

Im Tupelraum werden, wie schon erwähnt, Tupel abgelegt. Es werden dabei zwei Arten von Tupeln unterschieden:

1. **Passive Tupel:** Sie unterscheiden sich in keinster Weise von den schon im Abschnitt 2.1 auf Seite 5 beschriebenen.
2. **Aktive Tupel:** In diesen Tupeln ist mindestens ein Element unbestimmt. Zur Bestimmung der unbestimmten Elemente wird ein paralleler Prozeß gestartet. Aktive Tupel sind im TR unsichtbar. Wenn die Werte aller Elemente bestimmt wurden, wird das aktive Tupel zu einem passiven Tupel und somit im TR sichtbar. Die Prozeßkreation wird im Kapitel 2.3.2 ab Seite 8 beschrieben.

*Templates* sind „Schablonen" oder „Masken", mit denen passende Tupel im TR gesucht werden. Templates sehen ähnlich aus wie passive Tupel, können aber sog. *Formals* besitzen. Hier ein Beispiel:

```
("Hallo",?,?)
```

Die Fragezeichen kennzeichnen die *Formals*, d.h. die Elemente des Templates, die unbestimmt sind. Geeignete Tupel stimmen in der Länge mit dem Such-Template überein, und außerdem sind die Werte der festliegenden Elemente (im Beispiel der *String* "Hallo") des Templates und der entsprechenden Elemente der Tupel gleich.

Für den Zugriff auf den Tupelraum werden vier Befehle zur Verfügung gestellt.

- Der Befehl `deposit` dient zum Ablegen von Tupeln im Tupelraum.
- Mit dem Befehl `fetch` wird mit Hilfe der oben beschriebenen Templates ein passendes Tupel im TR gesucht und entfernt. Passen mehrere Tupel eines TR auf

die Such-Templates, so wird ein zufälliges entfernt. `fetch` kann sowohl blockierend, wie auch nicht blockierend ausgeführt werden. Blockierend heißt, daß der ausführende Prozeß auf ein passendes Tupel wartet, wenn keines im TR vorhanden ist (analog z.B. zum Message-Passing-Modell).

- Der Befehl `meet` arbeitet wie `fetch`, nur entfernt er das Tupel nicht aus dem TR. Auch dieser Befehl kann blockierend oder nicht blockierend ausgeführt werden.
- Der letzte Befehl, `meet-into`, kann als atomare Ausführung eines `deposit`s und eines `fetch`s aufgefaßt werden. „Atomar" bedeutet hier, daß zwischen der Ausführung der beiden Befehle kein anderer Prozeß auf den TR zugreifen darf, also keine Prozeßumschaltung in der Zwischenzeit stattfindet. Ein `meet-into` liest demnach ein passendes Tupel aus und verändert es gleichzeitig.

### 2.3.2 Die Prozeßkreation unter ProSet-Linda

Zum Starten von parallelen Prozessen steht unter ProSet-Linda der ||-Operator (sprich: „double-pipe") zur Verfügung. Seiteneffekte und Write-Parameter sind für Prozesse unter ProSet-Linda nicht erlaubt. Dies wird durch einen zusätzlichen *Closure*-Ausdruck im Prozeßaufruf verdeutlicht. Der Closure-Ausdruck friert die Umgebung einer Prozedur, die nur Write-Parameter besitzt und zu einem neuen Kontrollfluß werden soll, ein und kann auch ohne ||-Operator benutzt werden. Durch das „Einfrieren der Umgebung" erhält die Prozedur eine statische Umgebung, d.h. die Werte aller sichtbaren globalen Variablen bleiben beim weiteren Programmablauf aus Sicht der Prozedur unverändert bestehen (siehe Genaueres in [Doberkat *et al.*, 1992]). Ein kurzes Beispiel:

$$|| \; closure \; p();$$

Im Beispiel wird der Prozeß $p$ ohne Parameter gestartet und parallel abgearbeitet. Denkbar ist, daß der gestartete Prozeß $p()$ einen Rückgabeparameter besitzt. Dann sähe das Beispiel wie folgt aus:

$$a := || \; closure \; p();$$

Der Wert von der Variablen $a$ ist in dem Ausdruck noch unbestimmt. Erst wenn $a$ in einem folgenden Programmausdruck enthalten ist, wird der Wert von $a$ ermittelt, d.h. gegebenenfalls auf die Beendigung des Kontrollflusses $p()$ gewartet.

Der ||-Operator kann nicht nur als einzelner ProSet-Ausdruck, sondern auch innerhalb des `deposit`-Befehls als Tupelelement auftreten. Dieses Tupel ist dann ein sogenanntes *aktives Tupel* (wie oben beschrieben). Wird der `deposit`-Befehl ausgeführt, so werden die Prozesse zur Berechnung der Tupelelemente gestartet.

Die durch den ||-Operator abgespalteten Kontrollflüsse werden im folgenden *Threads* genannt.

### 2.3.3 Weitere Befehle zur Behandlung von Tupelräumen

Zur Behandlung von Tupelräumen allgemein stehen weitere Befehle zur Verfügung. Diese Befehle sollten nach [Doberkat *et al.*, 1992] Teil einer Standardbibliothek für ProSet-Linda sein, sind aber in der bisherigen Implementierung Teil des Sprachkerns. Dies wird auch in dem Laufzeitsystem dieser Arbeit nicht anders sein.

**CreateTS(limit)** – erzeugt einen TR. Zur Identifikation des TRs wird ein Atom erzeugt und zurückgegeben. „Limit“ steht für die maximale Anzahl von Tupeln, die im Tupelraum gleichzeitig vorhanden sein dürfen.

**ExistsTS(ts)** – sucht nach einem TR mit dem Atom „ts“.

**ClearTS(ts)** – löscht alle Tupel aus dem TR mit dem Atom „ts“.

**RemoveTS(ts)** – löscht den Tupelraum mit dem Atom „ts“ aus der Liste der Tupelräume.

### 2.3.4 Das lokale Netz am Lehrstuhl X

Am Informatik-Lehrstuhl für Softwaretechnologie der Universität Dortmund werden zur Zeit Workstations der Firma *Sun Microsystems* unter dem UNIX-Betriebssystem Sun-OS 4.1 eingesetzt. In Art und Ausstattung unterscheiden sich die Rechner sehr. Dies ist aber nur für die Performance-Messung von Bedeutung. Deshalb wird im Abschnitt 12.1 auf Seite 93 dieses Problem genauer behandelt.

Verbunden sind die Workstations durch ein Ethernet. Ethernet ist die zur Zeit der Arbeit häufigste Art der Implementierung von lokalen Netzen. Eine kurze Beschreibung befindet sich im Anhang A.

Wichtig für diese Arbeit ist vor allem der Umstand, daß im Ethernet Single-, Multi- und Broadcasting, d.h. das Senden zu einer Station, einer Gruppe und allen Stationen, im Ethernet die gleichen Kommunikationskosten verursachen. Dies folgt aus dem Umstand, daß im Ethernet quasi alle Packete von jeder Station empfangen werden und soll in Kapitel 4.3 auf Seite 21 zur Berechnung der Kommunikationskosten für verschiedene Ansätze zum Tupelraum-Management benutzt werden. Auch zur Implementierung der drei Kommunikationsarten befindet sich im Anhang A eine kurze Beschreibung.

## 2.4 Das verwendete Message-Passing-System PVM

*Parallel Virtual Machine* (PVM) ist ein Message-Passing-System, das es ermöglicht, ein heterogenes Netzwerk als virtuellen parallelen Computer mit verteiltem Speicher anzusehen und es als solchen zu benutzten.

PVM ist das Hauptprojekt des *Heterogeneous Network Computing research project*, einer Kooperation zwischen dem Oak Ridge National Laboratory, der Universität von Tennessee, der Emory Universität und der Carnegie Mellon Universität.

Der Aufbau, die Funktionsweise und die Anwendungsmöglichkeiten von PVM sind ausführlich in [Geist *et al.*, 1994] beschrieben. Außerdem gibt es im Internet ausführliche WWW-Seiten (http://www.netlib.org/pvm3/) und im Usenet eine Newsgroup zu PVM (comp.parallel.pvm).

Nun soll kurz auf die Möglichkeiten, die PVM bietet, eingegangen werden:
Unter PVM können die verschiedensten Rechnertypen miteinander kooperieren. Die unterstützten Architekturen reichen von PCs bis zu Cray-Rechnern. Es werden also sowohl Einprozessormaschinen, als auch echte Parallelrechner unterstützt. Alle diese Maschinen müssen nur eins gemeinsam haben: Sie müssen unter einem UNIX-Betriebssystem laufen.

**Die virtuelle Maschine:** Um nun diese Rechner unter PVM zu nutzen, müssen sie zuerst Teil der *virtuellen Maschine* von PVM sein. Praktisch heißt dies nichts anderes, als daß ein Server für PVM auf diesem Host gestartet werden muß. Beim Start von PVM in einer Shell eines Rechners erfolgt dies automatisch für diesen Rechner und alle Rechner des Netzwerkes, welche in einer Konfigurationsdatei definiert worden sind. Allerdings ist es auch möglich, Hosts dynamisch, also zur Laufzeit eines Programms, zur virtuellen Maschine hinzuzufügen.

**Dynamische Prozeßerzeugung:** Unter PVM können Prozesse dynamisch erzeugt werden. Die Prozessor-Allokation wird standardmäßig durch einen einfachen

Round-Robin-Scheduler übernommen, kann aber auch durch den Vaterprozeß erfolgen (siehe dazu auch Kapitel 6.4 auf Seite 43).

Zur Verwaltung von Prozessen gibt es zahlreiche Funktionen, die hier nicht aufgeführt werden sollen. Näheres dazu ist in den PVM-Manuals zu finden ([Geist *et al.*, 1994]). Auf die Prozeßkreation wird allerdings noch einmal genauer in Kapitel 5.3 ab Seite 37 eingegangen.

**Message-Passing:** Der Nachrichtenaustausch erfolgt durch einfache Befehle zum Einpacken, Versenden und Auspacken von Daten. Als Datentypen werden alle C-Standarddatentypen (aber natürlich keine Zeiger) unterstützt. Für heterogene Netze steht eine Möglichkeit der Datenkonvertierung in eine externe Datenrepräsentation (XDR) zur Verfügung. Der Nachrichtenaustausch kann unter PVM als zuverlässig angesehen werden. Protokolle zum sicheren Nachrichtenaustausch sind daher nicht nötig.

Das Umleiten von Ausgaben (auf `stdout`) zum Rechner des Benutzers übernimmt ebenfalls PVM.

# 3. Vorgaben

Nachdem nun das Umfeld dieser Arbeit beschrieben worden ist, sollen die Vorgaben zu dieser Arbeit (sowohl restriktiver, als auch erleichternder Art) erläutert werden.

## 3.1 Homogenes lokales Netz

Um die Komplexität dieser Arbeit nicht weiter zu erhöhen, sollte zu Anfang die Implementierung nur auf einem homogenen *Local Area Network* (LAN) erfolgen. Dies bedeutet, daß alle Rechner, die über das Netzwerk verbunden sind, vom gleichen Typ (z.B. SUN-Workstations) sind und das gleiche Betriebssystem benutzen. Durch die Verwendung der Message-Passing-Library *Parallel Virtuel Machine* (PVM) ist diese Beschränkung aufgehoben worden. PVM unterstützt heterogene Netze problemlos. Die Programme müssen aber (natürlich) für die verschiedenen Maschinentypen übersetzt werden.

Aus meßtechnischen Gründen wurden die Performance-Messungen auf einem homogenen Netz durchgeführt. Dazu aber erst im Kapitel 12.1 ab Seite 93 mehr.

## 3.2 Keine wesentlichen Veränderungen des Compilers und des Übersetzungsvorganges

Der Compiler, der mit dem bisherigen Laufzeitsystem arbeitete, soll auch für den Übersetzungsvorgang des neuen Systems benutzt werden können. Dies schließt aber kleinere Änderungen, wie z.B. das Einfügen einiger zusätzlicher Funktionsaufrufe (z.B. für das Starten der Tupelraum-Server) nicht aus.

Der Übersetzungsvorgang für den Programmierer soll sich nicht verändern. Allerdings muß der Benutzer seinen Wunsch zur Benutzung des verteilten Laufzeitsystems durch eine Option beim Aufruf des Compilers ausdrücken.

Außerdem kann der Anwender eine Konfigurationsdatei mit den Angaben, die sich auf die zu verwendenden Rechner und anderen Optionen beziehen, spezifizieren, wenn er nicht auf Standardeinstellungen zurückgreifen möchte. Diese Konfigurationsdatei wird in Kapitel 11.1 auf Seite 86 beschrieben.

## 3.3 Grundsätzliche Entwurfsanforderungen an ein verteiltes System

Nach den eher speziellen Restriktionen nun noch einige allgemeinere, die auch in bezug auf andere verteilte Systeme (dann aber meist Betriebssysteme) angewendet werden und in die Designüberlegungen dieses Systems einfließen sollten:

### 3.3.1 Transparenz

Transparenz ist eine der wichtigsten Anforderungen an ein verteiltes System, wird aber meist nicht verwirklicht (vgl. [Tanenbaum, 1992]). Mit Transparenz ist hier die Unsichtbarkeit der Verteiltheit für den Anwender gemeint. Für den Benutzer soll ein verteiltes System also genau so aussehen wie ein zentralistisches Einrechner-System.

Transparenz sollte sowohl für die „normale" Bedienung des Systems gelten, als auch im Falle eines Fehlers. Unter „normaler" Bedienung sind in dieser Arbeit besonders die schon erwähnte Benutzung des Compilers und manuelle Eingaben für ein Programm während der Laufzeit gemeint. Dies ist ohne Aufweichung der Transparenz nicht zu erreichen, wie schon im Kapitel 3.2 erläutert wurde.

### 3.3.2 Skalierbarkeit

Ein weiterer Punkt ist die mögliche Skalierbarkeit des verteilten Systems. Unter Skalierbarkeit wird die problemlose Erweiterung des Netzwerkes ohne Veränderung der angewendeten Techniken verstanden. Wichtig ist in diesem Zusammenhang besonders die Betrachtung der Kommunikationskosten. Diese sollten nur linear mit der Anzahl der Knoten im Netz steigen.

Um Skalierbarkeit zu erreichen, sollten möglichst dezentralistische Algorithmen angewendet werden, da zentralistische bei einer Vergrößerung des Netzes Engpässe (oder Flaschenhälse) entwickeln.

### 3.3.3 Performance

Die Betrachtung der Performance steht in dieser Arbeit im Mittelpunkt. Die Performance eines Systems hängt nach Tanenbaum (vgl. [Tanenbaum, 1992]) erheblich von der Geschwindigkeit der Kommunikation ab. Daher sollte die Anzahl der Nachrichten möglichst klein sein. Allerdings hängt eine Performance-Steigerung des verteilten Systems erheblich von der Intensität der parallelen Verarbeitung ab. Intensive parallele Verarbeitung benötigt aber mehr Kommunikation. Dies sind also konträre Faktoren.

Die Laufzeit eines ProSet-Programms hängt aber nicht nur von dem Laufzeitsystem ab, sondern vor allem vom Algorithmus des Programms selber. Das Laufzeitsystem kann diesen Algorithmus durch einen minimalen Systemaufwand unterstützen. Ziel dieser Arbeit muß es also sein, Protokolle und Techniken zu implementieren, die durch eine flexible Struktur und einen geringen Aufwand verschiedenste Granularitäten[1] von Parallelität unterstützen.

### 3.3.4 Platzbedarf

Auch der Platzbedarf des Systems auf den Rechnern sollte Beachtung finden. Besonders bei der Implementierung der verteilten Tupelräume sollten redundante Informationen (Tupel) vermieden werden.

### 3.3.5 Zuverlässigkeit

Von verteilten Systemen wird zumeist eine höhere Zuverlässigkeit erwartet als von zentralistischen Systemen. Dies wäre zwar in bezug auf die Tupelräume mit bestimmten Algorithmen des Tupelraum-Managements zu erreichen (siehe dazu auch Kapitel 4 auf Seite 19), jedoch nicht ohne gewisse Einbußen bei der Performance des Systems. Da das Hauptaugenmerk in dieser Arbeit auf der Performance liegt, sollte die Zuverlässigkeit des Systems nicht schlechter als die des zentralistischen Laufzeitsystems sein.

[1] Unter Granularität wird hier der Quotient aus Arbeits- und Kommunikationsaufwand pro Prozeß verstanden.

Auf die Implementierung von Fehlertoleranzverfahren und Redundanz von Informationen wird in dieser Arbeit aber bewußt verzichtet (s.o.).

# Teil II

# Einordnung der Arbeit und Beschreibung der eigenen Ansätze

# 4. Ansätze zur verteilten Tupelraumimplementierung

In der bisherigen zentralistischen Implementierung von ProSet-Linda werden die Tupelräume durch den gemeinsamen Speicher der Leichtgewichtsprozesse, die mit dem ||-Operator gestartet werden, verwirklicht. Durch den gemeinsamen Speicher läßt sich das Linda-Konzept auf einfachste Weise umsetzen.

In Netzwerken kann zur Tupelraumimplementierung nicht auf einen gemeinsamen Speicher (shared memory) zurückgegriffen werden. Hier müssen andere Techniken gefunden werden. Welche Anforderungen müssen diese Ansätze erfüllen? Diese Frage wird im folgenden Kapitel diskutiert, bevor dann bestehende Ansätze beschrieben werden.

## 4.1 Grundsätzliche Anforderungen an ein verteiltes Tupelraum-Management

Da auf Netzwerken kein gemeinsamer Speicher vorhanden ist, muß der virtuelle gemeinsame Speicher von Linda in irgendeiner Weise auf die Knoten des lokalen Netzes abgebildet werden. Außerdem muß jeder Ansatz folgende Fragen beantworten (vgl. [Bjornson, 1993]):

- Wohin sendet der `deposit`-Befehl das Tupel?
- Wohin sendet der `fetch`- und der `meet`-Befehl das Template, und wie findet das Template ein passendes Tupel?

Zur genaueren mathematischen Beschreibung dieser Anforderungen werden im folgenden einige Definitionen eingeführt, die in den nachfolgenden Abschnitten verwendet werden:

1. $K$ sei die Menge der Knoten $\{k_1, \ldots, k_n\}$ im Netzwerk, wobei $n > 0$ die Anzahl der Knoten bezeichnet.
2. $\Theta$ sei die Menge aller Tupelräume $\{T_1, \ldots, T_i\}$, die wiederum durch die Prozesse $(t_{1,1}, ..., t_{i,j})$ repräsentiert werden. $i > 0$ bezeichnet die Anzahl der Tupelräume, $j > 0$ die Anzahl der Prozesse pro Tupelraum. Dies bedeutet, daß ein Tupelraum durch mehrere Prozesse implementiert sein kann.
3. $P$ sei die Menge aller Prozesse $\{p_1, \ldots, p_{e+h}\}$. Wobei $P$ aus der Menge der Benutzerprozesse $B = \{b_1, \ldots, b_e\}$ und der Menge der Tupelraumprozesse $T = \{t_1, \ldots, t_h\}$ besteht. $e > 0$ und $h > 0$ bezeichnen hier die Anzahl der Benutzerprozesse und der Tupelraumprozesse.
4. Auf jedem Knoten $k \in K$ kann eine Teilmenge von Prozessen aus $P$ liegen. Diese Teilmenge sei durch $F(k)$ bezeichnet.
5. Ein `deposit` eines Prozesses $p \in P$ wird Tupelkopien für einen Tupelraum $T$ zu einer Menge von Tupelraumprozessen $\{t_1, \ldots, t_s\}$, $0 < s \leq h$, schicken. Diese Menge heiße $D_{p,T}$. (Es sei darauf hingewiesen, daß $D_{p,T} \subset T$ aber nicht $D_{p,T} = T$ gelten muß.)
6. Bei einem `fetch` und einem `meet` empfängt eine Menge von Knoten das Template von dem Prozeß $p$ für den Tupelraum $T$, die Menge $F_{p,T}$.

Dann muß folgende Bedingung erfüllt sein, die Bjornson in seiner Dissertation [Bjornson, 1993] angegeben hat. Allerdings geht er dort nicht auf den Ausnahmefall ein, daß die Mengen leer seien könnten. Hier wird dies nachgeholt:

$$\forall p \in P, \forall T \in \Theta : D_{p,T} \neq \emptyset \wedge F_{p,T} \neq \emptyset \Rightarrow D_{p,T} \cap F_{p,T} \neq \emptyset$$

Dies heißt nichts anderes, als daß ein Template und ein gesuchtes Tupel sich auf mindestens einem Knoten im Netzwerk treffen müssen.

## 4.2 Das Vorgehen bei der Berechnung der Kosten

Zum Vergleich der Ansätze werden die Kosten für die Kommunikation und den Platzbedarf berechnet. Dabei wird zwischen den Kosten für Tupel und Templates unterschieden.

- $O_{len}$ bezeichnet nun die Kosten für ein Tupel der Länge *len* (angelehnt an den Befehl `out` in LINDA).
- $I_{len}$ bezeichnet nun die Kosten für ein Template der Länge *len* (angelehnt an den Befehl `in` in LINDA).

Die errechneten Kommunikationskosten beziehen sich im folgenden immer auf ein `deposit` und ein `fetch` und sind von der Anzahl der Tupelräume unabhängig. Der `meet`-Befehl verursacht die gleichen Kosten wie `fetch` und muß daher nicht zusätzlich betrachtet werden. Da der `meet-into`-Befehl eine Besonderheit PROSETs ist und in den hier betrachteten Implementierungen nicht enthalten ist, kann dieser Befehl auch nicht zu Vergleichen herangezogen werden.

Es sei noch einmal darauf hingewiesen, daß von einem Ethernet als Netzwerk ausgegangen wird und daher ein Broadcasting die gleichen Kosten verursacht wie eine normale Nachricht.

Die Berechnung für den Platzbedarf bezieht sich auf den benötigten Speicher für die Tupelräume im gesamten Netz. Die Kosten werden für jeweils ein Tupel eines Tupelraumes abhängig von der Länge des Tupels angegeben.

## 4.3 Techniken zum Tupelraum-Management

Die folgenden Methoden wurden im Original in verteilten Implementierungen verschiedenster LINDA-Varianten eingesetzt. Alle diese Varianten haben gemeinsam, daß sie keine multiplen Tupelräume unterstützen. Zum leichteren Verständnis und um Probleme bei der Verwendung dieser Techniken in PROSET-LINDA so schnell wie möglich aufzudecken, wurden die Ansätze auf PROSET-LINDA-Befehle und multiple Tupelräume übertragen, sowie die Verwendung des Ethernets vorausgesetzt.

Folgende Ansätze werden behandelt:

- Zentrale Speicherung der Tupelräume mit Server-Prozessen.

- Positives Broadcasting, d.h. Replikation jedes Tupelraumes an jedem Knoten.
- Negatives Broadcasting, d.h. Verteilung der Tupelräume über das Netzwerk mit einer Kopie pro Tupel.
- Verteiltes Hashing, d.h. Aufteilung der Tupelräume in disjunkte Partitionen.

Zur Beschreibung der Ansätze werden die oben eingeführten Definitionen benutzt. Eine einfachere, weil sich nur auf singuläre Tupelräume beziehende Notation, hat auch Bjornson in [Bjornson, 1993] für die Beschreibung der drei ersten Ansätze benutzt. Hier wird versucht, auch den vierten Ansatz – das verteilte Hashing – mit der eingeführten Definition zu beschreiben. Bjornson unterläßt dies in seiner Dissertation. Auch hierfür mußten die Definitionen aus [Bjornson, 1993] zur obigen Notation erweitert werden.

Im folgenden werden die Ansätze zum verteilten Tupelraum-Management genauer erläutert.

### 4.3.1 Die zentrale Speicherung der Tupelräume

Diese Methode ist wohl die einfachste. Pro Tupelraum gibt es genau einen ausgezeichneten Knoten mit einem Tupelraumprozeß, der für die Verwaltung des TR verantwortlich ist.

In der schon in Kapitel 4.1 erläuterten Notation wird dies wie folgt beschrieben:

$$\begin{aligned} D_{p_k,T_i} &= \{t_{i,1}\} \\ F_{p_k,T_i} &= \{t_{i,1}\} \end{aligned}$$

Durch die zentrale Verwaltung jeweils eines Tupelraumes durch einen Server werden alle Tupelraumoperationen, die sich auf einen TR beziehen, serialisiert, d.h. sie müssen nacheinander ausgeführt werden. Durch diesen Umstand geht die echte Parallelität des Netzes verloren. Eine Laufzeitverbesserung gegenüber dem bisherigen Laufzeitsystem kann nur durch die parallele Verwaltung der verschiedenen Tupelräume erreicht werden.

Die Kosten für diesen Ansatz sind leicht zu berechnen:

**Kommunikationskosten:** Ein `deposit` sendet das Tupel an den Server und verursacht daher Kosten von $O_{len}$. Ein `fetch` durch einen Benutzerprozeß sendet eine Anfrage an den Tupelraumserver ($I_{len}$). Der Server schickt das Tupel im Erfolgsfall zurück oder aber eine Nachricht, daß ein passendes Tupel nicht gefunden werden konnte. Dies verursacht maximal Kosten von $1O_{len}$. Insgesamt entstehen also Kommunikationskosten von:

$$2O_{len} + I_{len}$$

**Speicherkosten:** Jeder Tupelraum wird nur einmal abgespeichert. Für ein gespeichertes Tupel der Länge $len$ entstehen also für einen Tupelraum ein Platzbedarf von:

$$O_{len}$$

Durch die Zentralität ist diese Methode sehr schlecht skalierbar, denn die Server werden bei einer Zunahme der Tupelraumanfragen zu Flaschenhälsen. Nur durch eine Replikation der zentralen Server und des dadurch verursachten Mehraufwandes an Kommunikation ist eine Erweiterung des lokalen Netzes möglich.

Eine Implementierung, die diese Methode benutzt, ist in [Robinson und Arthur, 1995] beschrieben.

### 4.3.2 Positives Broadcasting

Bei dieser Technik liegen Replikationen eines jeden Tupelraums auf jedem Knoten im Netz. Deshalb gilt hier, daß auf jedem Knoten $k_i$ aus $K$ $|\Theta|$ Tupelraumprozesse liegen.

$$D_{p_k,T_i} = K$$
$$F_{p_k,T_i} = \{t_{i,k}\}, \text{ wobei } t_{i,k} \in F(k) \text{ und } p_k \in F(k).$$

Das heißt, daß ein `deposit` Kopien des abzulegenden Tupels an jeden Netzwerkknoten schicken muß. Die Suche für `fetch` und `meet` ist hier dagegen sehr billig, denn sie kann lokal ausgeführt werden. Die Anfrage an den Besitzer des Tupels ist dann allerdings wieder mit Netzwerkverkehr verbunden.

**Kommunikationskosten:** Das initiale Broadcasting (`deposit`) zu allen Knoten kostet $O_{len}$. Eine `fetch`-Anfrage findet lokal statt und kostet nichts. Wird ein Tupel gefunden, wird eine Anfrage an den Besitzer des Tupels geschickt ($I_{len}$). Die Antwort des Besitzers kostet $O_{len}$. Das Broadcasting des Löschbefehls des Besitzers an alle Tupelraumreplikationen kostet $I_{len}$.
Insgesamt entstehen also maximal Kommunikationskosten von:

$$2O_{len} + 2I_{len}$$

**Platzbedarf:** Ein weiterer gewichtiger Nachteil dieses Ansatzes ist der hohe Platzbedarf der Tupelräume, bedingt durch die Replikation auf jedem Knoten:

$$|\Theta|O_{len}$$

Da zu jeder Zeit die Konsistenz der Tupelraumreplikationen gewährleistet sein muß, ist sicher zu stellen, daß jedes `deposit` alle Knoten im Netz sofort erreicht. Es ist also ein zuverlässiges Broadcasting notwendig. In [Carriero und Gelernter, 1986] wird dieser Ansatz auf einem S/Net implementiert. Dieses Netz bietet ein zuverlässiges Broadcasting (In [Arango und Berndt, 1989] wird die Einschränkung des zuverlässigen Broadcastings allerdings durch ein spezielles Protokoll behoben).

Weiterhin muß bei erfolgreichem `fetch` ein kompliziertes und teureres Löschprotokoll eingesetzt werden, um alle Kopien des Tupels atomar zu löschen.

Ein Vorteil des Positiven Broadcastings ist sicherlich die gute Skalierbarkeit. Das Hinzufügen eines Knotens zum Netz ist problemlos möglich und belastet die Leistungsfähigkeit dieser Methode nicht. Da zur Kommunikation im Netz nur Broadcasting verwendet wird, bleiben die Kosten der Kommunikation im Ethernet gleich. Diese sind in diesem Netz unabhängig von der Anzahl der Knoten[1]. Allerdings wiegt dieser Vorteil die großen Nachteile nicht auf.

[1] Physikalisch sind der Anzahl der Knoten im Ethernet allerdings Grenzen gesetzt. Näheres siehe in [Metcalfe und Boogs, 1976].

### 4.3.3 Negatives Broadcasting

Negatives Broadcasting wurde von Leichter verwendet, um sein verteiltes LINDA-System zu implementieren [Leichter, 1989]. Es bildet das Gegenstück zum Positiven Broadcasting, d.h. alle Tupel werden lokal abgespeichert, die Templates jedoch zu jedem Knoten gesendet. In der in Kapitel 4.1 eingeführten und bisher verwendeten Notation heißt dies:

$$D_{p_k,T_i} = \{t_{i,k}\}, \text{ wobei } t_{i,k} \in F(k) \text{ und } p_k \in F(k).$$
$$F_{p_k,T_i} = P$$

Dies bedeutet, daß in diesem Ansatz ein `deposit` billig ist, weil diese Operation nur lokal ausgeführt wird. `fetch` und `meet` können dagegen teuer sein, wenn sie global, d.h. auf dem gesamten Netz, suchen müssen. Dies ist der Fall, wenn ein gesuchtes Tupel nicht lokal vorhanden ist (vgl. Kostenberechnung unten).

**Kommunikationskosten:** Ein `deposit` veruracht keine Kosten, da es nur lokal ausgeführt wird. Für `fetch` und `meet` ist ein Broadcasting der Templates notwendig. Dies verusacht im Ethernet Kosten von $I_{len}$. Das Zurücksenden eines passenden Tupels kostet nochmals $O_{len}$. Insgesamt entstehen also Kommunikationskosten von:

$$O_{len} + I_{len}$$

**Platzbedarf:** Da alle Tupel nur einmal im Netz vorhanden sind, ist der Platzbedarf gering:

$$O_{len}$$

In diesem Ansatz ist bei dem Broadcasting der Templates kein zuverlässiges Protokoll notwendig. Dies ist dem Umstand zu verdanken, daß mehrfache Kopien eines Templates im Netz keinen logischen Konflikt verursachen, denn es muß (laut LINDA-Semantik) nur sichergestellt sein, daß ein passendes Tupel gefunden wird. Um sicherzugehen, daß der gesamte Tupelraum durchsucht wird (alle Knoten angesprochen werden), arbeiten `fetch` und `meet` mit einem Timeout[2]: Beide Funktionen warten eine gewisse Zeitspanne

[2]Timeouts werden in der Interprozeßkommunikation oft eingesetzt. Für Beispiele siehe z.B. [Tanenbaum, 1992].

auf Antwort. Haben in dieser Zeitspanne nicht alle Knoten geantwortet, senden sie eine weitere Template-Kopie an die Knoten.

Die Kosten für die Speicherung der Tupelräume sind proportional zu ihrer Größe. Die Replikation der Templates bietet im allgemeinen den weiteren Vorteil, daß sie im Hinblick auf die Speicherkosten besser als die Replikation der Tupel ist, denn die Anzahl der konkurrierenden `meet`- und `fetch`-Operationen im Netz sind durch die Anzahl der Prozesse im Netz, die über Tupelräume miteinander kommunizieren, beschränkt.

Der Umstand, daß jeder Tupelraum über die gesamte Knotenmenge verteilt ist, hat noch einen weiteren Vorteil: Droht auf einem Knoten der Speicher für die Tupel knapp zu werden, so können Tupel auf andere Knoten migriert werden. Dies ist durch ein einfaches Zweistufenprotokoll (siehe [Tanenbaum, 1992]) zu erreichen.

Ein großer Nachteil des Negativen Broadcastings ist jedoch die mögliche globale Suche der `meet`- und `fetch`-Operationen. Die globale Suche ist sogar wahrscheinlich. Es findet keine Kommunikation zwischen verschiedenen Netzwerkknoten statt, wenn nur lokale Suchen durchgeführt werden. Durch das Broadcasting werden die Operationen aller Benutzerprozesse netzwerkweit serialisiert. Es geht also auch bei dieser Methode echte Parallelität verloren.

### 4.3.4 Verteiltes Hashing

In diesem Ansatz werden die Tupelräume in disjunkte Partitionen unterteilt, die durch jeweils einen Prozeß verwirklicht werden. Tupel und Template werden durch jeweils eine Hashfunktionen auf diese Partitionen abgebildet. Die Abbildung der Tupel ist dabei in jedem Falle injektiv bzw. eindeutig, d.h. jedes Tupel wird genau auf eine Partition abgebildet.

$$D_{p_k,T_i} = \{t_{i,g}\}$$
$$F_{p_k,T_i} = \{t_{i,j}, \ldots, t_{i,g}, \ldots, t_{i,h}\}$$
$$\text{Bei eindeutigem Hashing ist } j = g = h.$$

Bei den Templates ist dies durch den Mangel an Informationen, der durch die Formals entsteht, nicht immer der Fall. Hier kann auch ein Template auf eine Menge von Partitionen abgebildet werden. Diese Partitionen müssen dann jeweils befragt werden.

Eine eindeutige Abbildung der Tupel und Templates wird in [Bjornson, 1993] erreicht, dies allerdings mit Hilfe eines Precompilers, der in dieser Arbeit nicht verwendet werden soll. Die Autoren von [Douglas *et al.*, 1994] erreichen kein eindeutiges Hashing. Sie benutzen statt eines Precompilers ein zweistufiges Hashing, dessen zweite Stufe die Zahl der Knoten, auf denen ein Tupel zu suchen ist, einschränkt.

**Kommunikationskosten:** Ein `deposit` verursacht Kosten von $O_{len}$. Bei eindeutigem Hashing kosten `fetch`- und `meet`-Anfragen $I_{len}$, bei nichteindeutigem Hashing $hI_{len}$, wobei $h$ die Anzahl der gehashten Partitionen sei. Die Antwort der Partitionen verursacht bei eindeutigem Hashing Kosten von $O_{len}$, bei nichteindeutigem Hashing sind die Kosten vom Protokoll abhängig, welches verhindert, daß mehr als ein Tupel aus dem Tupelraum entfernt wird. In [Douglas *et al.*, 1994] wird die Anfragenachricht an die erste gehashte Partition gesendet. Kann sie dort nicht erfüllt werden, schickt die Partition sie an die nächste weiter, u.s.w. Dieses Vorgehen kostet maximal $hI_{len}$. Die Antwort kosten dann noch $O_{len}$.

Insgesamt entstehen also bei eindeutigem Hashing Kommunikationskosten von:

$$2O_{len} + I_{len}$$

Bei nichteindeutigem Hashing:

$$2O_{len} + hI_{len}$$

**Platzbedarf:** Da alle Tupel nur einmal im Netz vorhanden sind, ist der Platzbedarf gering:

$$O_{len}$$

Alle Operationen auf einen Tupelraum können parallel ausgeführt werden, solange sie sich nicht auf die gleiche Partition beziehen. Mit steigender Anzahl der Partitionen wird die Wahrscheinlichkeit einer Serialisierung geringer.

Es ist kein Broadcasting und kein kompliziertes Löschprotokoll notwendig.

Ein spezifisches Problem von ProSet-Linda ist: Schreibende `meet`s können Tupel so verändern, daß diese ihre Merkmale für ihre Partition verlieren und in eine andere Partition migrieren müssen. In [Bjornson, 1993] und [Douglas *et al.*, 1994] tritt dieses Problem nicht auf, da dort nur Linda-Varianten benutzt wurden, die schreibende `meet`s nicht kennen.

### 4.3.5 Zusammenfassung des Vergleiches der Ansätze

In den obigen Kapiteln wurden vier Ansätze zum verteilten Tupelraum-Management erläutert und ihre wichtigsten Vor- und Nachteile genannt, besonders in bezug auf die Kommunikationskosten und den Platzbedarf.

Welcher Ansatz ist nun der für diese Arbeit geeignetste? Jeder vorgestellte Ansatz hat seine Vor- und Nachteile. Allerdings sind im Rahmen dieser Arbeit noch weitere Aspekte, die bisher noch nicht betrachtet wurden, zu berücksichtigen:

- Alle Ansätze müssen auch in Hinblick auf die `meet-into`-Operation betrachtet werden.
- Es soll der Ansatz der multiplen Tupelräume verwirklicht werden. Dies ist besonders bei den Ansätzen von Bedeutung, bei denen auf jedem Knoten im Netz ein Tupelraumprozeß arbeitet.
- Die echte Parallelität des Netzes sollte so weit wie möglich erhalten bleiben, um möglichst viel Rechenzeit gegenüber einem zentralistischem System zu gewinnen.
- Der Ansatz sollte möglichst flexibel sein. Zum einen sollte er einfach an die Komplexität und auf den wahrscheinlichen Rechenaufwand des ProSet-Programms anpaßbar sein, und zum anderen wäre eine mögliche Emulation eines anderen Ansatzes durch den gewählten zu Vergleichszwecken von Vorteil.

Das positive Broadcasting würde für die `meet-into`-Operation, äquivalent zu dem komplizierten Löschprotokoll für das `fetch`, ein Änderungsprotokoll benötigen, welches alle Tupelreplikationen im Netz atomar verändert. Neben den hohen Kommunikationskosten würde dies auch zu Behinderung nicht nur einiger, sondern aller Benutzerprozesse führen. Um die Vorteile dieses Ansatzes zu nutzen, müßte auf `jedem` Knoten im Netz eine Tupelraumreplikation liegen. Da mehrere Tupelräume benutzt werden sollen, wäre die Anzahl der Tupelraumprozesse im Netz immens.

Letzteres gilt auch für das negative Broadcasting. Allerdings wäre die `meet-into`-Implementierung sehr einfach, da keine Replikationen zu beachten wären. Die Parallelität des Netzes würde aber wahrscheinlich bei diesem Ansatz zu sehr eingeschränkt.

Das verteilte Hashing ist durch eine flexible Wahl der Partitionsanzahl pro Tupelraum an jede Netzgröße und Rechnerleistung leicht anpaßbar und gewährleistet einen nur geringen Verlust an Parallelität. Die Kommunikationskosten sind abhängig von der Güte der Hashfunktion bzw. einem geeigneten Protokoll bei nicht-eindeutigem Hashing.

Wird nur eine Partition pro Tupelraum gewählt, so ist dies äquivalent zum Ansatz des *Zentralen Servers*, so daß beide Ansätze in ihrer Performance verglichen werden können. Für die Implementierung des `meet-into`s könnte das eindeutige Hashing der Tupel nicht nur von den Benutzerprozessen, sondern auch von den Partitionen durchgeführt werden. Ein Zusatzprotokoll würde dann nicht benötigt. Der Aufwand stiege dabei um eine Nachricht.

Somit fällt die Wahl also auf das verteilte Hashing.

## 4.4 Eigener Ansatz

Als zu implementierender Ansatz für das verteilte Tupelraum-Management wurde im letzten Kapitel das verteilte Hashing ausgewählt. Hier sollen nun einige wichtige Details des Ansatzes diskutiert werden.

### 4.4.1 Die Hashfunktion

Wichtig für die Effizienz des verteilten Hashings ist die Eindeutigkeit der Hashfunktion. In den oben diskutierten Implementierungen handelte es sich ausnahmslos um C-Linda-Systeme. C-Linda ist eine Kombination aus der stark typisierten Sprache C mit Linda. Durch die starke Typisierung von C ist der Datentyp der Formals in den Templates bekannt und kann zum Hashing benutzt werden. Da ProSet schwach getypt ist, kann dieser Ansatz in dieser Arbeit nicht übernommen werden.

Die starke Typisierung scheint aber im allgemeinen nicht auszureichen, um ein eindeutiges Hashing zu erreichen. In [Douglas *et al.*, 1994] wird eindeutiges Hashing nur für Templates erreicht, deren erstes Element kein Formal ist (wobei der Typ der Formals und die Länge des Templates bekannt sind). In [Bjornson, 1993] wird die Eindeutigkeit erst durch eine Analyse während der Kompilierung erreicht. Wie schon erwähnt soll hier kein Precompiler benutzt werden (siehe Kapitel 3 auf Seite 12).

Theoretisch wäre ein eindeutiges Hashing bereits durch die Berücksichtigung der Länge zu erreichen, nur würde dies in den meisten Programmen nicht zu einer gleichmäßigen Verteilung der Tupel auf die Menge der Partitionen führen. Im schlimmsten Fall käme es zu einer Saturation einer Partition und somit zu einem Flaschenhals wie im Ansatz des Zentralen Servers (siehe Abschnitt 4.3.1 auf Seite 22).

Das spätere System wird dem ProSet-Programmierer mehrere Hashfunktionen anbieten, aus denen er dann wählen kann.

Mögliche Ansätze sind:

- Die Länge und die Typen der Elemente der Tupel und der Actuals der Templates werden benutzt. Dies könnte bei Templates mit wenigen Formals schon zu guten Ergebnissen führen.
- Es wird nur die Länge der Tupel und Templates benutzt. Dies führt zu eindeutigem Hashing. Es sollten allerdings die Längen der Tupel und Templates stark variieren, da sonst, wie oben beschrieben, keine optimale Verteilung stattfindet.
- Es wird über die Länge und das erste Element gehasht. Dann darf das erste Element der Templates nie ein Formal sein. Auch hier kann eindeutig gehasht werden. Die Gleichverteilung hängt aber wiederum vom Programmierer ab.

Eine endgültige Lösung des Saturationsproblems ist wohl nur durch eine Precompiler-Analyse möglich, denn bei den übrigen Lösungen kann immer der Programmierer die Verteilung der Tupel und Templates beeinflussen. Dies gestehen auch Douglas u.a. in [Rowstron *et al.*, 1995] ein.

Gleichverteiltes und eindeutiges Hashing ist also für ProSet ohne Precompiler nicht zu erreichen. Daher wird ein besonderes Protokoll notwendig sein, um eine korrekte Suche auf der Menge der gehashten Partitionen durchzuführen. Trotzdem sollte eine schlecht gewählte Hashfunktion nicht zu übergroßen Laufzeitverschlechterungen führen. Ein Protokoll mit dem versucht wird, diese Probleme zu lösen, wird im nächsten Abschnitt zunächst informal beschrieben.

### 4.4.2 Die Protokolle

In Abschnitt 4.3.4 auf Seite 26 wurde schon kurz das Protokoll aus [Douglas *et al.*, 1994] beschrieben. Dieses Protokoll befragt im Falle von nicht-eindeutigem Hashing zuerst den ersten Prozeß in der Liste der gehashten Prozesse nach einem passenden Tupel. Ist dort die Suche erfolglos, fragte die erste Partition die nächste in der Liste usw.

Dieses Vorgehen verursacht minimale Kommunikationskosten. Allerdings wird dafür die Suche auf der Partitionsmenge serialisiert. Die Zeit für diese Suche ist somit abhängig von der Kardinalität der Menge der Partitionen und somit wiederum von der Güte der Hashfunktion.

In dieser Arbeit wird ein anderes Protokoll benutzt, das versucht die Arbeit auf den Partitionen zu parallelisieren und somit die benötigte Zeit unabhängig von der Güte der Hashfunktion zu machen. Dies kann allerdings nur durch einen höheren Kommunikationsaufwand erreicht werden.

Wie schon erklärt, wird in diesem Laufzeitsystem ein Tupelraum aus einer oder mehreren Partitionen bestehen. Jede dieser Partitionen wird durch genau einen Prozeß repräsentiert. Die Verwaltung der aktiven Tupel (bzw. deren Prozesse) übernimmt ebenfalls ein Prozeß. Dieser Prozeß wird *Eval-Server* genannt[3]. Es gibt nur einen einzigen Eval-Server für alle Tupelräume.

Kommunikation findet fast ausschließlich zwischen dem Benutzerprozeß und den Partitionen, dem Benutzerprozeß und dem Eval-Server oder den Partitionen und dem Eval-Server statt.

Die beiden Ausnahmen dieser Regel bilden die beiden folgenden Situationen:

1. Beim Prozeßstart wird eine Initialisierungsnachricht mit allen notwendigen Informationen vom Vater- zum Sohnprozeß gesendet.
2. Bei Thread-Prozessen wird der „Returnwert" zurück zum Vaterprozeß geschickt, wenn kein Eval-Server gestartet wurde.

Warum wird überhaupt ein Thread-Prozeß sein Ergebnis zuerst zum Eval-Server senden? Dies liegt daran, daß im generierten C-Code des zu verwendenden ProSet-Compilers aktive Tupel und Prozesse, die durch den ||-Operator gestartet werden, nicht unterschieden werden. Solange also aktive Tupel benutzt werden und somit ein Eval-Server vorhanden ist, muß dieser die Unterscheidung vornehmen.

Jedes Tupel auf dem ein `deposit` durchgeführt werden soll wird durch eine Hashfunktion eindeutig einer Partition zugewiesen. Die Templates der `fetch`- und `meet`-Operationen können durch ihre Hashfunktion meistens nicht eindeutig zugeordnet werden. Die Protokolle dieser Befehle arbeiten dann auf einer Menge von $n$ Partitionsprozessen.

Durch die Verwendung von PVM wird davon ausgegangen, daß keine Nachricht verloren geht. Quittierungen sind also nicht notwendig. PVM bietet zwei Arten von Routing an: Einerseits können die Nachrichten über die PVM-Server (pvmd) geroutet werden, andererseits können die Nachrichten mit Hilfe von TCP [Stevens, 1990] direkt geroutet werden. In dieser Spezifikation wird zur Kostenberechnung von der zweiten Art ausgegangen. Diese wurde auch später in der Implementierung verwandt.

[3] Im originalen Linda ist „Eval" der Befehl zur Erschaffung von aktiven Tupeln (vgl. [Gelernter, 1985])

Nun zu den Protokollen der einzelnen Tupelraumbefehle. Zuerst werden alle Tupelraumbefehle ohne aktive Tupel betrachtet.

Es sei darauf hingewiesen, daß eine vollständige, formale Spezifikation der im folgenden kurz beschrieben Protokolle im Kapitel 8.2 ab Seite 61 zu finden ist.

### 4.4.3 Informale Beschreibung der Protokolle und Kosten für die Tupelraumbefehle mit passiven Tupeln

#### Das Protokoll für `deposit`

Das `deposit`-Protokoll ist nur eine einzelne Nachricht: An die gehashte Partition wird die `deposit`-Nachricht gesand, die aus einer Identifikation der Art der Nachricht und dem Tupel besteht. (Wie schon erwähnt erfolgt keine Quittierung.) Es entstehen also bei dieser Operation konstante Kosten.

#### Das Protokoll der nicht-blockierenden `fetch`s und `meet`s

Nacheinander wird jeder Partition aus der Hashmenge eine `fetch`- bzw. `meet`-Nachricht gesendet, die aus einer Operations-ID und der Liste der Templates besteht, und dann auf eine Antwort gewartet. Diese Antwortnachricht ist entweder leer (bei erfolgloser Suche) oder besteht aus der Nummer des Templates, für das ein Tupel gefunden wurde und den fehlenden Elementen, deren Platzhalter die *Formals* waren. War die Suche erfolglos, wird der nächsten Partition in der Hashmenge eine `fetch`- bzw. `meet`-Nachricht gesendet. Ist die Menge leer oder die Suche erfolgreich, wird die Suche beendet. Bei erfolgloser Suche wird dann der *Else*-Zweig des ProSet-Statements durchlaufen.

Im Worst-Case (der erfolglosen Suche) werden Kommunikationskosten von $2n$ Nachrichten verursacht, wobei $n$ der Anzahl der gehashten Partitionen entspricht.

Das Abfragen der Partitionen nacheinander erhöht den Zeit- und Rechenaufwand des Benutzerprozesses. Allerdings besteht die Möglichkeit, daß nicht alle Partitionen befragt werden müssen und so bis zu 2(n-1) Nachrichten einzusparen sind. Weiterhin müßte bei der gleichzeitigen Anfrage an alle Partitionen ein Protokoll entwickelt werden, daß Mehrfachlöschungen von gefundenen Tupeln verhindert. Dies würde zusätzlichen Kommunikationsaufwand verursachen.

### Das Protokoll der blockierenden `fetch`s und `meet`s

Da das Protokoll für `meet` äquivalent ist zum `fetch`-Protokoll, wird zunächst nur das `fetch` behandelt.

Bei der blockierenden Version des `fetch`s wird anders verfahren als bei der nicht-blockierenden Variante: Zuerst werden allen Partitionen aus der gehashten Menge die Liste der Templates gesendet und somit angefragt, ob die jeweilige Partition ein gesuchtes Tupel besitzt. Diese Anfrage ist mit einer `meet`-Operation zu vergleichen. (Es sei darauf hingewiesen, daß sofort allen Partitionen diese Nachricht gesendet wird. Es wird nicht wie bei der nicht-blockierenden Variante zuerst auf eine Antwort gewartet.)

Findet eine Partition ein gesuchtes Tupel, so sendet es eine positive Nachricht an den anfragenden Prozeß. Das Tupel wird allerdings nicht aus dem Speicher der Partition entfernt. Findet sie kein passendes Tupel, so wird der Prozeß von der Partition in ihre Liste der wartenden Prozesse eingefügt. Findet ein `deposit` mit einem passenden Tupel statt, erhält der Prozeß eine positive Nachricht und wird wieder aus der Liste gelöscht. In beiden Fällen hat er noch **kein** gesuchtes Tupel erhalten, sondern nur einen „Hinweis“ wo sich eines befindet.

Der anfragende Prozeß wartet so lange, bis er eine positive Nachricht erhält. Dann startet er für die Partition, die ihm diese Nachricht gesendet hat, das oben beschriebene Protokoll des nicht-blockierenden `fetch`s. Im Falle eines blockierenden `meet`s wird hier das Protokoll des nicht-blockierenden `meet`s gestartet. Dies ist der einzige Unterschied der beiden Protokolle.

Nun können zwei Fälle auftreten:

1. Ist das nicht-blockierende `fetch` erfolgreich, läßt sich der anfragende Prozeß durch eine Nachricht an alle übrigen Partitionen aus den Listen der wartenden Prozesse entfernen und beendet damit das `fetch`-Protokoll.

2. Ist das nicht-blockierende `fetch` erfolglos, weil ein anderer Prozeß schneller war, dann wird für diese Partition das Protokoll von vorn begonnen, d.h. die Liste der Templates verschickt und auf die nächste positive Nachricht einer Partition gewartet.

Das gleichzeitige Befragen aller Partitionen der Menge parallelisiert die Suchoperation. Reagiert wird auf die Antwort der Partition, die zuerst antwortet. Dies bedeutet, daß eine Partition im folgenden weiter belastet wird, die die momentan geringste Auslastung besitzt.

Der Kommunikationsaufwand ist allerdings sehr viel höher als bei der Lösung von [Douglas *et al.*, 1994]. Sei $n$ die Anzahl der Partitionen, die zu durchsuchen sind, und $z$ die Anzahl der erfolglosen Versuche das Tupel nicht-blockiernd zu `fetch`en. Dann liegt der Kommunikationsaufwand zwischen $2n + 2$, wenn nur eine Partition ein passendes Tupel findet und dieses sofort gefetcht wird, und $3n + 4z + 1$, wenn alle Partitionen ein passendes Tupel finden, aber $z$ vergebliche Versuche unternommen werden müssen, eines dieser Tupel zu bekommen. Der Kommunikationsaufwand für ein blockierendes `meet` entspricht dem des blockierenden `fetch`s.

### 4.4.4 Das Sonderprotokoll für `meet-into`

Das Protokoll für `meet-into` ist äquivalent zu den oben erklärten Protokollen für `fetch` und `meet`. Es gibt nur einen Unterschied: Wird in dem nicht-blockierenden `meet-into`-Protokoll – welches auch in dem blockierenden enthalten ist – ein Tupel in der Partition gefunden, dann wird dieses mit Hilfe der `into`-Funktion verändert. Dann wird die Hashfunktion auf dieses Tupel angewendet und falls die Ergebnis-ID nicht mit der ID der Partition übereinstimmt, ein `deposit` ausgeführt oder, wenn das Tupel nach der Veränderung aktiv ist, zum Eval-Server gesendet. D.h. in diesem Protokoll kann auch eine Partition ein deposit ausführen. Erst dann wird die Antwortnachricht an den eigentlichen Benutzerprozeß geschickt.

Während des `deposit`s ist der Benutzerprozeß blockiert. Führt er gleich nach dem `meet-into` ein `meet` oder `fetch` auf das veränderte Tupel aus, dann müssen die Anfragen hinter der `deposit`-Nachricht der `meet-into`-Partition in der Warteschlange der neuen Partition liegen. Ist dies der Fall, so ist die LINDA-Semantik gesichert.

### 4.4.5 Die Behandlung von Threads

Threads sind mit Hilfe des ||-Operators abgespaltete Kontrollflüsse. Diese werden in dem Laufzeitsystem dieser Arbeit durch eigenständige Prozesse repräsentiert. Im generierten C-Code des ProSet-Compilers werden beim Starten Threads und aktive Tupel gleichbehandelt und sind erst durch ihre spätere unterschiedliche Behandlung zu unterscheiden. Daher können Threads ihr Ergebnis nur dann direkt zum Vaterprozeß senden, wenn im ProSet-Programm keine aktiven Tupel verwendet werden und der Start eines Eval-Servers daher in der Konfigurationsdatei ausgeschaltet ist (vgl.Abschnitt 11.1 auf Seite 86). Ansonst müssen die Ergebnisse erst zum Eval-Server gelangen und von dort durch den Vaterprozeß abgeholt werden.

### 4.4.6 Das Vorgehen bei aktiven Tupeln

Aktiv ist ein Tupel dann, wenn mindestens eins seiner Elemente ein ||-Operator-Ausdruck ist. Allerdings können auch mehrere Ausdrücke dieser Art in einem aktiven Tupel vorkommen. Somit entsteht das Problem, daß eine Instanz existieren muß, welche auf die einzelnen Ergebnisse der Threads wartet, diese an den richtigen Stellen in das Tupel einsetzt und sie dann in dem richtigen Tupelraum ablegt.

Diese Instanz wird durch einen Eval-Server übernommen. Es gibt nur einen Eval-Server im System, der die Arbeit für alle vorhandenen Tupelräume übernimmt. Zu ihm werden alle aktiven Tupel gesendet. Da die zugehörigen Prozesse schon vorher gestartet wurden, enthalten diese nur die IDs der Threads. Der Eval-Server verwaltet nun eine Liste aller unvollständigen Tupel, wartet auf Ergebnisse, die die Prozesse vor ihrer Beendigung automatisch zu ihm senden und führt das `deposit`-Protokoll für die vervollständigten Tupel auf dem richtigen Tupelraum aus. So werden nur passive Tupel in den Tupelräumen abgelegt.

# 5. Prozeßkreation

Dieses Kapitel beschäftigt sich mit der Erzeugung von Prozessen auf dem lokalen Rechner und auf entfernten Hosts im Netz. Zuerst wird noch einmal kurz auf die Möglichkeiten, die ProSet-Linda bietet, eingegangen. Dann werden die Alternativen unter UNIX beschrieben, bevor zu der verwendeten Message-Passing-Bibliothek PVM übergegangen wird. Zuletzt werden die Probleme, die in dieser Arbeit liegen, verdeutlicht und eine Lösung vorgestellt.

## 5.1 ProSet-Linda

ProSet-Linda bietet, wie schon in Kapitel 2.3 beschrieben, zwei Möglichkeiten zur Prozeßkreation. Zuerst einmal hat der Programmierer die Möglichkeit, Prozesse mit Hilfe des ||-Operators zu starten. Dies sähe z.B. wie folgt aus:

```
x:=||p();
```

Hierbei wird in dem bisherigen Laufzeitsystem ein *Leichtgewichtsprozeß* (LWP) oder *Thread*[1] gestartet, der die Prozedur `p()` quasiparallel bzw. parallel ausführt. (In dem in dieser Arbeit behandelten verteilten Laufzeitsystem werden dann parallele (echte) Prozesse auf einem anderen Rechner gestartet).

Die zweite Möglichkeit besteht darin, ein aktives Tupel im Tupelraum (siehe Kapitel 2.3) zu erzeugen. Die geschieht ebenfalls mit dem ||-Operator. Ein Beispiel:

```
deposit ["tuple1",||p()] at TS end deposit;
```

[1] Hier sei auf die Überladung des Wortes „Thread" hingewiesen. Während an dieser Stelle Leichtgewichtsprozesse unter SOLARIS als Threads bezeichnet werden, waren in den vorhergehenden Abschnitten parallele Kontrollflüsse gemeint.

Hier wird also mit einem `deposit` ein Tupel in einen Tupelraum abgelegt, welches ein Element enthält, dessen Wert erst noch mit einem parallelen Aufruf der Prozedur `p()` ermittelt werden muß. Auch in diesem Fall wird in dem bisherigen System ein LWP bzw. Thread gestartet.

## 5.2 UNIX

Unter dem Betriebssystem UNIX gibt es i.allg. zwei Befehle, die Prozesse erzeugen.

Einer dieser Befehle ist `fork`. `Fork` erzeugt vom aufrufenden Prozeß einen **identischen** Sohnprozeß. Dieser unterscheidet sich vom Vaterprozeß nur durch seine Prozeß-ID.

Alle Prozesse können Programme durch den `exec`-Befehl aufrufen. Dieser Befehl erzeugt aber keinen neuen Prozeß, sondern führt das Programm unter der ID des aufrufenden Prozesses aus. Nach Ausführung des Befehls wird ganz normal in die nächste Programmzeile gesprungen.

Ein mit `fork` erzeugter Sohnprozeß hat also nur die Möglichkeit, nach Abfrage seiner Prozeß-ID seinen Kontrollfluß umzuleiten oder den Befehl `exec` aufzurufen, um damit ein Kommando bzw. ein Programm aufzurufen.

Der zweite Befehl, mit dem Prozesse erzeugt werden können, ist `rsh`. `Rsh` führt Kommandos auf entfernten Rechnern aus. Der Prozeß, der hierbei erzeugt wird, ist ein Sohn der *Login-shell* des entfernten Hosts. Er ist also nicht wie bei `fork` identisch mit dem aufrufenden Prozeß. Dies wird im folgenden noch wichtig sein.

Zur genaueren Beschreibung der genannten UNIX-Befehle siehe [Stevens, 1990] oder die elektronischen Manual-Seiten.

## 5.3 PVM

Unter PVM können dynamisch Prozesse auf der virtuellen Maschine erzeugt werden. Hierzu dient der Befehl `pvm_spawn`. Intern benutzt diese Funktion den `rsh`-Befehl von UNIX, so daß auch unter PVM nur neue Prozesse erzeugt werden können. Somit gibt es unter PVM zum entfernten Starten von Prozessen ebenfalls keinen `fork`-Mechanismus.

Natürlich ist die Funktionalität von `pvm_spawn` nicht nur auf einen bloßen `rsh`-Aufruf beschränkt. Gleichzeitig wird mit dem Befehl auch festgelegt, auf welcher Maschine der Prozeß gestartet werden soll. Zum einen kann dies PVM überlassen werden. Dann wird der Round-Robin-Scheduler angestoßen, der schon im Kapitel 2.4 erwähnt wurde. Zum anderen kann aber auch die Maschine vom Programmierer bestimmt oder die Menge der Maschinen eingeschränkt werden.

Zusätzlich bietet `pvm_spawn` noch gewisse Debug-Optionen, auf die aber hier nicht eingegangen wird.

## 5.4 Der eigene Ansatz zur Prozeßkreation

In dieser Arbeit muß der oben erläuterte Mechanismus von ProSet nachgebildet werden. Dazu wäre es nützlich, einen Befehl wie *fork* netzwerkweit anwenden zu können. Dies ist aber nicht möglich. Wie kann mit den oben genannten Befehlen ein „remote-fork" nachgebildet werden? Dieses Problem soll in dieser Arbeit gelöst werden.

In [Robinson und Arthur, 1995] wird eine C-Linda-Implementierung beschrieben, in der dieses Problem durch einen sogenannten „case-main"-Ansatz gelöst wurde.

In C-Lindagibt es nur die Möglichkeit, Prozesse als aktive Tupel zu starten. Der ||-Operator ist dort unbekannt. Deshalb haben sich Robinson u.a. in ihrer Implementierung auf den Tupelraum-Mechanismus abgestützt.

Wenn ein aktives Tupel in den Tupelraum abgelegt werden soll, wird ein neuer Prozeß mit `rsh` gestartet und eine eindeutige ID als passives Tupel in den einzigen Tupelraum abgelegt. Diese ID bestimmt den neuen Einstiegspunkt des Prozesses, d.h. die Funktion, die den neuen Thread des Linda-Programms repräsentiert.

Das Tupel mit der ID wird zu Anfang von jedem Sohnprozeß im Tupelraum gesucht. Danach wird anstatt des normalen Hauptprogramms `main()` eine Funktion `case_main()` durchlaufen, die aus einer einfachen `case`- bzw. `switch`-Anweisung besteht, in der anhand der ID die richtige Funktion angesprungen wird.

Die Idee dieser Technik wird auch in dieser Arbeit verwendet. Allerdings wird hier nicht auf einen Tupelraum zurückgegriffen. Außerdem wird die Technik nicht nur zum Start von aktiven Tupeln benutzt, sondern zum Start fast aller Prozesse des Laufzeitsystems (Sohnprozesse, Partitionen und Eval-Server). Die einzige Ausnahme bilden die Loadagenten, die eigene Programme darstellen.

Nun soll der eigene Ansatz genauer beschrieben werden:
Mit `pvm_spawn` wird ein neuer Prozeß gestartet. Dieser Prozeß durchläuft (wie auch der Hauptprozeß) am Anfang eine Initialisierungsfunktion für Linda. Diese Funktion gibt es auch schon in der zentralistischen Implementierung, sie heißt `plp_init_linda`.

Die einzelnen Prozeßtypen sind im neuen Laufzeitsystem durch eigenständige Funktionen implementiert, die den Kontrollfluß dieser Prozesse steuern.

Zur Unterscheidung der einzelnen Prozeßtypen werden den Aufrufen einfache Parameter übergeben, die somit der ProSet-Programmierer nicht benutzen darf. Dies ist aber nur eine kleine Einschränkung, da in ProSet ein Zugriff auf Programmparameter nicht vorgesehen ist und dies nur über den Umweg von C-Funktionen erfolgen kann. Auch in der oben genannten Implementierung von Robinson u.a. kann es zu Problemen kommen, da C-Linda nur singuläre Tupelräume kennt. Dieser eine Tupelraum wird sowohl von dem System für die IDs, die als normale Tupel abgelegt werden, als auch vom Benutzer verwendet.

Anhand der übergebenen Parameter wird nun in `plp_init_linda` entschieden, welche Funktion zur Steuerung des Kontrollflusses aufgerufen wird. Welche Parameter genau verwendet werden, wird erst zur Implementierung entschieden. Eine Beschreibung der Parameter findet sich unter C.3.3.

Innerhalb der Funktion für den Sohnprozeß (`plp_remote`) wird dann auf eine Initialisierungsnachricht vom Vater gewartet. Diese Nachricht enthält dann alle notwendigen Informationen, wie den Namen der auszuführenden Prozedur, die Parameter der Prozedur, die Liste der Tupelräume usw.

# 6. Die statische Loadbalancierung

Neben den Protokollen steigert vor allem eine gute Verteilung der Rechenlast im Netz die Performance. Deshalb sollte ein verteiltes Laufzeitsystem eine Loadbalancierung anbieten. Obwohl es zuerst den Anschein hatte, stellte sich im Verlauf dieser Arbeit heraus, daß PVM in der verwendeten Version keine lastabhängige Prozessor-Allokation anbietet. Die Bestimmung eines Hosts zum Start eines neuen Prozesses wird durch einen einfachen Round-Robin-Algorithmus gewährleistet. Erst in einer späteren Version soll PVM eine Loadbalancierung erhalten.

Dieser Umstand führte dazu, daß eine Loadbalancierung im Verlauf dieser Arbeit implementiert werden mußte. Die Entscheidung fiel auf eine statische Loadbalancierung.

## 6.1 Warum eine statische Loadbalancierung?

Bevor diese Frage beantwortet wird, sollen zunächst die Alternativen vorgestellt werden.

Grundsätzlich werden zwei Strategien der Loadbalancierung unterschieden: Erstens die *nicht migrierende Prozessor-Allokation oder statische Loadbalancierung* und zweitens die *migrierende Prozessor-Allokation oder dynamische Loadbalancierung*.

Bei der erstgenannten Methode wird zum Start eines jeden neuen Prozesses entschieden, auf welchem Rechner im Netz er laufen soll. Bei der zweiten Methode kann dem Prozeß während seiner Laufzeit zusätzlich ein neuer Host zugewiesen werden. Dieser Wechsel von Prozessen von einem Rechner auf einen anderen wird *Migration* genannt.

Während die statische Loadbalancierung einfacher zu implementieren ist als die dynamische, bietet diese aber den Vorteil, die Systemauslastung auch während der Laufzeit beeinflussen zu können. In die Entscheidung, ob eine Migration stattfinden soll oder nicht fließen vielfältige Faktoren ein, wie z.B. die Last des bisherigen Rechners, die Last der möglichen neuen Rechner oder die Kosten einer Migration, die nicht unerheblich sind. Dies erhöht die Komplexität des Entscheidungsalgorithmus sehr und verteuert die Loadbalancierung.

Neben diesem Problem tritt hier noch das technische Problem der Migration selbst auf. PVM unterstützt Migration von Prozessen in keinster Weise. Dies heißt, daß eine Verlagerung von Prozessen nur durch einen großen Aufwand und erhebliche Eingriffe in den Quelltext von PVM möglich ist. Dies würde den Rahmen dieser Arbeit sprengen. Somit wurde nur eine statische Loadbalancierung implementiert.

## 6.2 Das Ziel der Loadbalancierung

Die Loadbalancierung allgemein gehört zu den Optimierungsproblemen. Die Optimierungsziele können höchst unterschiedlich sein:

- Ein mögliches Ziel könnte eine maximale Ausnutzung der CPUs sein, d.h. Brachzeiten sollten vermieden werden.
- Ein weiteres Ziel könnte die Minimierung der Antwortzeit der Prozesse sein.

Weitere Ziele sind leicht vorstellbar. Da in dieser Arbeit die schnelle Abarbeitung der ProSet-Programme im Vordergrund steht, ist das Ziel der Loadbalancierung eine Minimierung der Antwortzeit. Um dieses Ziel zu erreichen, können vielfältige Wege beschritten werden. Im den folgenden beiden Kapiteln werden die Entscheidungen, die zum implementierten Ansatz führten, erläutert.

Nach der Festlegung der Art der Loadbalancierung müssen weitere Entscheidungen bezüglich des verwendeten Algorithmus gefällt werden, die den Entwurf (Design) und die Implementierung betreffen. Diese Entscheidungen werden in den nächsten zwei Kapiteln ebenfalls diskutiert.

## 6.3 Der Entwurf des Loadbalancierungs-Algorithmus

In [Tanenbaum, 1992] werden vier große Entscheidungen genannt, die in der Entwurfsphase eines Prozessor-Allokations-Algorithmus gefällt werden müssen:

### 6.3.1 Deterministisch oder heuristisch?

Die erste Frage ist, ob ein deterministischer oder ein heuristischer Algorithmus verwendet werden soll. Deterministische Algorithmen werden eingesetzt, wenn alles über das Prozeßverhalten bekannt ist. Dies ist in einem Laufzeitsystem nicht der Fall. So kann der verwendete Algorithmus also nur heuristisch sein.

### 6.3.2 Zentralisiert oder verteilt?

Die zweite Frage ist, ob alle Informationen erst an einer Stelle gesammelt werden sollen oder nicht. Die Sammlung aller Informationen vereinfacht die Entscheidung, erhöht aber auch den Kommunikationsaufwand. Es gibt aber in dieser Arbeit ein paar Besonderheiten: Einmal wird kein zentraler Prozeß zur Loadbalancierung benötigt, sondern die Prozessor-Allokation kann von dem Prozeß durchgeführt werden, der einen neuen Prozeß starten möchte (also ein aktives Tupel in einem Tupelraum ablegt oder der ||-Operator verwendet wird).

Zweitens wurde bei der Planung des Algorithmus davon ausgegangen, daß ein Broadcasting oder Multicasting unter PVM möglich ist und so nicht $2n$ Nachrichten zur Loadbestimmung benötigt werden, sondern nur $n+1$ (eine Broadcasting-Nachricht an die n Hosts und n Antworten)[1].

Durch die Einfachheit und die nur mäßigen Kosten fiel die Entscheidung so auf den zentralistischen Ansatz.

### 6.3.3 Suboptimale oder optimale Lösung?

Optimale Lösungen sind nach [Tanenbaum, 1992] mit viel höheren Kosten verbunden als suboptimale Lösungen. Da hier aber nur eine statische Loadbalancierung eingesetzt werden soll, wird Wert auf eine möglichst gute (optimale) Lösung gelegt. Die höheren Kommunikationskosten werden, wie schon im vorherigen Kapitel erwähnt, als nicht erheblich angesehen. Fallen sie doch höher als erwartet aus, muß eine suboptimale Lösung gewählt werden.

[1] Leider stellte sich in der Testphase heraus, daß das Multicasting von PVM nicht fehlerfrei ist und deshalb nur bedingt eingesetzt werden konnte. Eine genaue Fehlerbeschreibung findet sich in Abschnitt 10.4 auf Seite 85.

### 6.3.4 Senderinitiiert oder empfängerinitiiert?

Die Frage, ob die Initiierung der Loadbalancierung von einem belasteten Host oder unbelasteten Host ausgeht, stellt sich bei der statischen Loadbalancierung nicht, da hier die Entscheidung nur an den bestimmten Zeitpunkt der Prozeßkreation gebunden ist und dann keine Berichtigung bzw. Anpassung an eine neue Belastungssituation durchgeführt werden kann. Wie schon erwähnt, wird die Initiierung durch den Prozeß durchgeführt, der einen neuen Prozeß starten möchte. Dieser Prozeß kann sowohl auf einer belasteten oder unbelasteten Maschine liegen.

### 6.3.5 Zusammenfassung der gewählten Entwurfsmerkmale

Nach der oben beschriebenen Diskussion fiel die Entscheidung auf einen heuristischen, zentralistischen Algorithmus, der eine optimale Lösung liefern soll. Er wird durch die Prozesse initiiert und durchgeführt, die neue Sohn-Prozesse als aktive Tupel oder neue Threads starten möchten.

## 6.4 Der Ansatz zur Implementierung des Loadbalancierungs-Algorithmus

In diesem Kapitel soll das Augenmerk auf Implementierungsaspekte gelegt werden.

PVM bietet dem Programmierer an, den Round-Robin-Scheduler durch einen eigenen zu ersetzen. Dies ist aber nur durch Eingriffe in das PVM-System und die Benutzung von PVM-internen Protokollen möglich. Diese Lösung war deshalb für diese Arbeit zu aufwendig.

Die Entscheidung fiel daher auf Benutzerprozesse, die auf jedem Rechner der virtuellen Maschine gestartet werden und den aktuellen Load berechnen. Diese Prozesse werden im folgenden Load-Agenten oder kurz Agenten genannt. Das eigentliche Scheduling wird jeweils von dem Prozeß übernommen, der einen Sohn-Prozeß starten möchte.

Wie errechnen die Agenten die Belastung einer Maschine? Zur Lösung dieses Problems wurde auf einen Mechanismus zurückgegriffen, den auch ein anderes Programm, das den Load einer Maschine anzeigt, benutzt. Dieses Programm ist *xload*, welches den Load einer Maschine in einem Fenster für den Benutzer anzeigt. Xload benutzt intern eine C-Funktion `get_load.c`. Diese Funktion greift auf UNIX-Maschinen auf den

Kernel selber zu und zwar auf `/dev/kmem`. `kmem` enthält neben einem Abbild (Image) des virtuellen Speichers auch einen Loadwert. Der Zugriff durch `get_load` ist maschinenspezifisch, d.h. zu jeder Hardwareplattform eines UNIX-Systems muß die Funktion portiert werden.

In der Implementierung wird `get_load` selber benutzt. Die Loadagenten bestehen aus einem Hauptprogramm, welches die Kommunikation mit den übrigen Programmen übernimmt und einem Aufruf von `get_load`. Nach ihrem Start warten die Agenten passiv auf eine Aufforderung zur Berechnung des Loads. Dann rufen sie `get_load` auf, übermitteln den Loadwert und warten wieder passiv auf eine neue Anfrage.

Beim Scheduling im Vaterprozeß können nun entweder alle Ergebnisse der Loadagenten benutzt werden oder auch nur einige. Welche Heuristiken genau eingesetzt werden, wird experimentell entschieden. Genauer wird in dem Kapitel 12 ab Seite 91 zur Performance-Messung auf die verwendeten Heuristiken eingegangen.

# Teil III

# Die Spezifikation

# 7. Die Spezifikation der Funktionen

## 7.1 Die gemeinsam genutzten Funktionen

### plp_init_linda

**Eingabe:** Parameter des ProSet-Programms (`argc, argv`).
**Ausgabe:** Keine.
**Beschreibung:** Diese Funktion wird in jedem Prozeß als erstes aufgerufen und verwirklicht den `case-main`-Ansatz. Anhand der übergebenen Parameter wird entschieden, ob es sich bei dem Prozeß um den Hauptprozeß, um einem Thread, eine Partition eines Tupelraumes oder um einen Eval-Server handelt. Entsprechend werden weitere Aufrufe getätigt.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 244.

### plp_error

**Eingabe:** Fehlerbeschreibung (Ort und Art).
**Ausgabe:** Keine.
**Beschreibung:** Diese Funktion wird im Fehlerfalle aufgerufen. Sie schreibt eine Fehlermeldung auf den Bildschirm oder in die Protokoll-Datei und beendet danach den Prozeß.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 251.

### plp_pack_obj

**Eingabe:** Zeiger auf ProSet-Objekt.
**Ausgabe:** Keine.
**Beschreibung:** Diese Fuktion packt ProSet-Objekte in eine PVM-Puffer. Dieser Puffer kann dann versendet werden. Bei Funktionen wird auch die Umgebung eingepackt.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 291.

### plp_unpack_func

**Eingabe:** Keine.
**Ausgabe:** Zeiger auf ProSet-Objekt vom Typ `Process`.
**Beschreibung:** Diese Funktion entpackt aus einem PVM-Puffer ProSet-Objekte vom Typ `Process`.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 304.

## 7.2 Die Funktionen der kommunizierenden Prozesse

### 7.2.1 Die externen Funktionen

### plo_deposit

**Eingabe:** Zeiger auf das Tupel, Atom des Tupelraumes.
**Ausgabe:** Standardwert für Handler.
**Beschreibung:** Diese Funktion überprüft, ob zu dem übergebenden Atom ein Tupelraum im Netz vorhanden ist. Ist das Tupel passiv, so wird eine Hashing-Funktion aufgerufen, die die *Tid* der richtigen Partition bestimmt. Dann wird eine `deposit`-Nachricht erzeugt und an die Partition gesendet. Ist das Tupel aktiv, so wird es an den Eval-Server geschickt. Zur Spezifikation der Protokolle siehe Kapitel 8.2.
**Referenz zum Code:** Kapitel C.2.3 auf Seite 126.

### plo_fetch

**Eingabe:** Liste der Templates, Atom des Tupelraumes, Typ des Aufrufs (blockierend oder nicht).
**Ausgabe:** Standartwert des Handlers, Nummer des ausgewählten Templates.
**Beschreibung:** Diese Funktion überprüft, ob zu dem übergebenden Atom ein Tupelraum im Netz vorhanden ist. Die `fetch`-Nachricht wird eingepackt. Dann wird entsprechend dem spezifiziertem Protokoll verfahren (siehe 8.2).
**Referenz zum Code:** Abschnitt C.2.3 auf Seite 131.

### plo_meet

**Eingabe:** Liste der Templates, Atom des Tupelraumes, Typ des Aufrufs (blockierend oder nicht).
**Ausgabe:** Standartwert des Handlers, Nummer des ausgewählten Templates.
**Beschreibung:** Diese Funktion überprüft, ob zu dem übergebenden Atom ein Tupelraum im Netz vorhanden ist. Die `meet`-Nachricht wird eingepackt. Dann wird entsprechend dem spezifiziertem Protokoll verfahren (siehe 8.2).
**Referenz zum Code:** Kapitel C.2.3 auf Seite 149.

### plp_finish_linda

**Eingabe:** Ein Exit-Wert.
**Ausgabe:** Keine.
**Beschreibung:** Diese Funktion wird in jedem Hauptprozeß als letztes aufgerufen. Es werden alle Tupelräume, alle bekannten Threads, alle Partitionen und gegebenfalls der Eval-Server und die Load-Agenten beendet. Danach beendet sich der Prozeß selbst.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 248.

### 7.2.2 Die internen Funktionen

#### plp_remote

**Eingabe:** Keine.
**Ausgabe:** Keine.
**Beschreibung:** Die Funktion wird von `plp_init_linda` aufgerufen und steuert den Kontrollfluß innerhalb eines Threads. Zuerst wird vom Vaterprozeß eine Nachricht mit allen notwendigen Informationen empfangen, wie z.B. die Liste der Tupelräume, die Tid des Eval-Servers usw. Danach wird die Funktion des C-Programms aufgerufen, welche die Prozedur des ProSet-Programms entspricht. Ein von dieser Funktion zurückgegebener Returnwert wird an den Eval-Server oder – wenn dieser nicht gestartet wurde – an den Vaterprozeß gesendet.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 251.

#### pli_process_in_tup

**Eingabe:** Zeiger auf Tuple.
**Ausgabe:** Wahrheitswert.
**Beschreibung:** Überprüft, ob im Tupel ein Objekt vom Typ `PROCESS` vorhanden ist.
**Referenz zum Code:** Kapitel C.2.3 auf Seite 143.

#### pli_pack_tl

**Eingabe:** Liste der Templates, Typ der Operation.
**Ausgabe:** Nummer des PVM-Puffers.
**Beschreibung:** Packt die Liste der Templates mit Hilfe von PVM-Funktionen ein und gibt die Nummer des PVM-Puffers zurück.
**Referenz zum Code:** Kapitel C.2.3 auf Seite 143.

### pli_hash_tup_general

**Eingabe:** Zeiger auf Tupel, Atom des Tupelraumes.
**Ausgabe:** Tid der richtigen Partition.
**Beschreibung:** Ruft die mit der externen Variable `plp_hash_mode` festgelegte Hashfunktion auf und gibt die Tid der gehashten Partition zurück. Die genauen Hashfunktionen werden erst während der Implementierungsphase bestimmt.
**Referenz zum Code:** Kapitel C.2.4 auf Seite 171.

### pli_hash_temp_general

**Eingabe:** Liste der Templates, Atom des Tupelraumes.
**Ausgabe:** Array der Tids der richtigen Partition.
**Beschreibung:** Ruft die mit der externen Variable `plp_hash_mode` festgelegte Hashfunktion auf und gibt ein Array der Tid der gehashten Partition zurück. Die genauen Hashfunktionen werden erst während der Implementierungsphase bestimmt.
**Referenz zum Code:** Kapitel C.2.4 auf Seite 173.

### pli_read_file

**Eingabe:** Name der Konfigurationsdatei.
**Ausgabe:** Keine.
**Beschreibung:** Liest die Konfigurationsdatei aus und bestimmt somit, welcher Hashmodus gewählt wurde, wieviel Partitionen pro Tupelraum gestartet werden sollen, ob eine Loadbalancierung stattfinden und ob ein Eval-Server gestartet werden soll. Während der Performance-Tests ist diese Funktion inaktiviert.
**Referenz zum Code:** Kapitel C.2.4 auf Seite 186.

### pli_scan (von Heiner Pohland)

**Eingabe:** Zeiger auf String.
**Ausgabe:** ProSet-Objekt.
**Beschreibung:** Konvertiert einen kodierten String in ein Proset-Objekt durch Aufruf von Spezialfunktionen.
**Referenz zum Code:** Kapitel C.2.4 auf Seite 189.

### pli_scipBlancs (von Heiner Pohland)

**Eingabe:** Zeiger auf String.
**Ausgabe:** Keine.
**Beschreibung:** Entfernt überflüßige Leerzeichen aus einem String.
**Referenz zum Code:** Kapitel C.2.5 auf Seite 199.

### pli_scanOm (von Heiner Pohland)

**Eingabe:** Zeiger auf String.
**Ausgabe:** ProSet-Objekt vom Typ `om`.
**Beschreibung:** Konvertiert einen kodierten String in ein Proset-Objekt vom Typ `om`.
**Referenz zum Code:** Kapitel C.2.5 auf Seite 200.

### pli_scanBoolean (von Heiner Pohland)

**Eingabe:** Zeiger auf String.
**Ausgabe:** ProSet-Objekt vom Typ `boolean`.
**Beschreibung:** Konvertiert einen kodierten String in ein Proset-Objekt vom Typ `boolean`.
**Referenz zum Code:** Kapitel C.2.5 auf Seite 201.

### pli_scanAtom

**Eingabe:** Zeiger auf String.
**Ausgabe:** ProSet-Objekt vom Typ `atom`.
**Beschreibung:** Konvertiert einen kodierten String in ein Proset-Objekt vom Typ `atom`.
**Referenz zum Code:** Kapitel C.2.5 auf Seite 202.

### pli_scanNumber (von Heiner Pohland)

**Eingabe:** Zeiger auf String.
**Ausgabe:** ProSet-Objekt vom Typ `integer` oder `real`.
**Beschreibung:** Konvertiert einen kodierten String in ein Proset-Objekt vom Typ `integer` oder `real`.
**Referenz zum Code:** Kapitel C.2.5 auf Seite 204.

### pli_scanString (von Heiner Pohland)

**Eingabe:** Zeiger auf String.
**Ausgabe:** ProSet-Objekt vom Typ `string`.
**Beschreibung:** Konvertiert einen kodierten String in ein Proset-Objekt vom Typ `string`.
**Referenz zum Code:** Kapitel C.2.5 auf Seite 207.

### pli_scanTuple (von Heiner Pohland)

**Eingabe:** Zeiger auf String.
**Ausgabe:** ProSet-Objekt vom Typ `tuple`.
**Beschreibung:** Konvertiert einen kodierten String in ein Proset-Objekt vom Typ `tuple`.
**Referenz zum Code:** Kapitel C.2.5 auf Seite 209.

### pli_scanSet (von Heiner Pohland)

**Eingabe:** Zeiger auf String.
**Ausgabe:** ProSet-Objekt vom Typ `set`.
**Beschreibung:** Konvertiert einen kodierten String in ein Proset-Objekt vom Typ `set`.
**Referenz zum Code:** Kapitel C.2.5 auf Seite 211.

### pli_scanProcess

**Eingabe:** Zeiger auf String.
**Ausgabe:** ProSet-Objekt vom Typ `Process`.
**Beschreibung:** Konvertiert einen kodierten String in ein Proset-Objekt vom Typ `process`.
**Referenz zum Code:** Kapitel C.2.5 auf Seite 213.

### pli_ots (ursprünglich von Heiner Pohland)

**Eingabe:** ProSet-Objekt.
**Ausgabe:** Zeiger auf String.
**Beschreibung:** Konvertiert ein Proset-Objekt in einen String.
**Referenz zum Code:** Kapitel C.2.4 auf Seite 192.

### plp_spawn

**Eingabe:** Zeiger auf ProSet-Objekt vom Typ `Function`, auf das vorher der `closure`-Operator angewendet worden war.
**Ausgabe:** Zeiger auf ProSet-Objekt vom Typ `Process`.
**Beschreibung:** Diese Funktion verwirklicht den ||-Operator. Sie startet einen neuen Prozeß mit Hilfe der PVM-Funktion `pvm_spawn()`. Dieser neue Prozeß repräsentiert einen Thread des ProSet-Programms. Es wird ein Objekt vom Typ `Process` zurückgegeben, in dem die PVM-Tid des neuen Prozesses gespeichert ist.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 235.

**plp_wait_task**

**Eingabe:** Zeiger auf ProSet-Objekt vom Typ `Process`.
**Ausgabe:** Ergebnisobjekt des zum Eingabeobjekt gehörigen Prozesses.
**Beschreibung:** Wenn ein Eval-Server im Netz vorhanden ist, ruft die Funktion das Ergebnis des Prozesses von ihm ab. Hat der Prozeß sich noch nicht beendet, wird gewartet. Ist kein Eval-Server im Netz, dann wird auf das Ergbnis direkt gewartet.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 240.

### 7.2.3 Die Standartbibliotheksfunktionen

**pll_create_ts**

**Eingabe:** Maximale Größe des Tupelraumes.
**Ausgabe:** Atom des Tupelraumes.
**Beschreibung:** Die Funktion erzeugt ein neues Atom für einen Tupelraum. Dann werden die Partitionsprozesse im Netz gestartet und die Liste der Tupelräume erweitert. Wenn ein Eval-Server vorhanden ist, wird dieser über den neuen Tupelraums informiert.
**Referenz zum Code:** Kapitel C.2.4 auf Seite 159.

**pll_exists_ts**

**Eingabe:** Atom.
**Ausgabe:** Wahrheitswert.
**Beschreibung:** Sucht mit Hilfe der Funktion `plo_find_ts` nach dem Tupelraum mit dem entsprechendem Atom.
**Referenz zum Code:** Kapitel C.2.4 auf Seite 165.

### pll_clear_ts

**Eingabe:** Atom.
**Ausgabe:** Wahrheitswert.
**Beschreibung:** Sucht mit Hilfe der Funktion `plo_find_ts` nach dem Tupelraum mit dem entsprechendem Atom und löscht alle passiven und aktiven Tupel des TR.
**Referenz zum Code:** Kapitel C.2.4 auf Seite 167.

### pll_remove_ts

**Eingabe:** Atom.
**Ausgabe:** Wahrheitswert.
**Beschreibung:** Sucht mit Hilfe der Funktion `plo_find_ts` nach dem Tupelraum mit dem entsprechendem Atom und beendet alle Partitionen.
**Referenz zum Code:** Kapitel C.2.4 auf Seite 169.

## 7.3 Die Funktionen der Partitionen

### plp_partition

**Eingabe:** Keine.
**Ausgabe:** Keine.
**Beschreibung:** Steuert den Kontrollfluß einer Partition. Wartet auf Tupelraumbefehle und arbeitet diese ab. Die passiven Tupel werden dabei in einer Hashtabelle abgespeichert. Zur Beschreibung der Nachrichten an Partitionen siehe Kapitel 8.2. und
**Referenz zum Code:** Kapitel C.3.3 auf Seite 279.

### plp_unpack_tl

**Eingabe:** Operationstyp.
**Ausgabe:** Liste von Templates.
**Beschreibung:** Packt eine PVM-Nachricht aus und konvertiert sie in eine Template-Liste. Da zu verschiedenen Nachrichttypen unterschiedliche Templatekomponenten übertragen werden, wird die Eingabe eines Operationstyps benötigt. Die Länge der übertragenen Templateliste kann variable sein. Das genaue Aussehen der Nachrichten wurde erst in der Implementierungsphase entschieden. Zur genauen Beschreibung ist der unten stehenden Referenz zu folgen.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 296.

### plp_unpack_comp

**Eingabe:** Zeiger auf Komponente eines Templates, Operationstyp.
**Ausgabe:** Keine.
**Beschreibung:** Packt aus einem PVM-Puffer die Komponenten eines Templates aus. Die Zusammensetzung des Puffers kann je nach Nachrichtentyp unterschiedlich sein. Siehe auch hier näheres unter der Referenz.
**Referenz zum Code:** Kapitel C.3.3 auf Seite 300.

## 7.4 Die Funktionen des Eval-Servers

### plp_eval_server

**Eingabe:** Keine.
**Ausgabe:** Keine.

**Beschreibung:** Diese Funktion wird von `plp_init_linda` aufgerufen, wenn der Prozeß ein Eval-Server ist. Die Liste aller Tupelraum-Partitionen, der Ergebnisse der Threads und der aktiven Tupel wird hier verwaltet. Der Eval-Server reagiert dabei nur auf Nachrichten und aktualisiert dann gegebenfalls seine Listen. Für mögliche Nachrichten an den Eval-Server siehe Kapitel 8.2.

**Referenz zum Code:** Kapitel C.3.3 auf Seite 258.

# 8. Die Spezifikation des Kommunikationsprotokolls

In diesem Kapitel soll nun genauer auf die Kommunikationsprotokolle eingegangen werden. Diese werden mit der ISO-Spezifikationssprache Estelle formal beschrieben. Die Sprache wurde zur Spezifikation ausgewählt, da sie für diesen Zweck geeignet und einfach zu verstehen ist. Außerdem waren Werkzeuge verfügbar, die die Arbeit mit Estelle unterstützten.

## 8.1 Die Spezifikationssprache Estelle

*Estelle* ist eine formale Beschreibungssprache für die Spezifikation von verteilten nebenläufigen Prozessen, insbesondere von Kommunikationsprotokollen und Diensten. Sie wurde von der *International Organization for Standardization* (ISO) initiiert. Heute ist Estelle ein ISO-Standard (vgl. [International Standard ISO/IS 9074, 1987]). Estelle basiert auf dem Modell der *erweiterten endlichen Automaten*. Im Unterschied zu endlichen Automaten kann ein erweiterter endlicher Automat neben seinen Zuständen lokale Daten halten und verändern.

Diese erweiterten endlichen Automaten werden in Estelle *Moduln* genannt. Sie werden in Estelle in einer Pascal-ähnlichen Form spezifiziert und sind daher leicht verständlich[1].

Moduln können über sogenannte *Kanäle* miteinander asynchron kommunizieren. Die Nachrichten werden an den Enden der Kanäle, den *Interaktionspunkten* (IP), gepuffert. Moduln sind in Estelle hierarchisch über Vater-Sohn-Beziehungen strukturiert. Neben Aktionen der Sohnmoduln kann ein Vatermodul selber ebenfalls Aktionen durchführen. Sohnmoduln können sequentiell oder parallel existieren.

[1] Estelle wird von machen sogar als Erweiterung von ISO-Pascal (ISO International Standard 7185) aufgefaßt.

Im wesentlichen bestehen die Aktionen aus Empfangen und Senden von Nachrichten. Interne Funktionen oder Prozeduren werden meist gar nicht oder nur rudimentär beschrieben. Die Sende- und Empfangsaktionen haben, da die Module wie oben beschrieben als Automaten angesehen werden, Zustandsänderungen zur Folge. Diese werden in Estelle mit sog. *Transitionen* beschrieben.

Eine ausführliche Beschreibung Estelles stellt natürlich ihre ISO-Norm dar [International Standard ISO/IS 9074, 1987]. Kurze Einführungen bieten aber auch die Artikel [Budkowski und Dembinski, 1987] und [Sijelmassi und Linn, 1992]. Das Buch [Hogrefe, 1989] beschäftigt sich neben Estelle auch mit zwei weiteren Dienst- und Protokollspezifikationssprachen *LOTOS* und *SDL*.

Zur Unterstützung der Spezifikation wurden bis jetzt eine Reihe von Werkzeugen entwickelt. Hervorgehoben seien nur das *Estelle Development Toolset* (SDL) und die in dieser Arbeit in Teilen benutzten Werkzeuge PET und DINGO (siehe [Budkowski, 1992] und [Sijelmassi und Strausser, 1993]).

## 8.2 Die Spezifikation der Protokolle

Nun folgt die Spezifikation der verwendeten Protokolle. Es sei noch einmal erwähnt, daß der Ansatz des Zentralen Tupelraumservers ein Sonderfall des hier spezifizierten Ansatzes des Verteilten Hashings ist und daher nicht gesondert formal beschrieben werden muß.

Diese Estelle-Spezifikation wurde durch das Tool PET des Entwicklungswerkzeugs PET and DINGO (vgl. [Sijelmassi und Strausser, 1993]) auf Syntaxfehler hin untersucht.

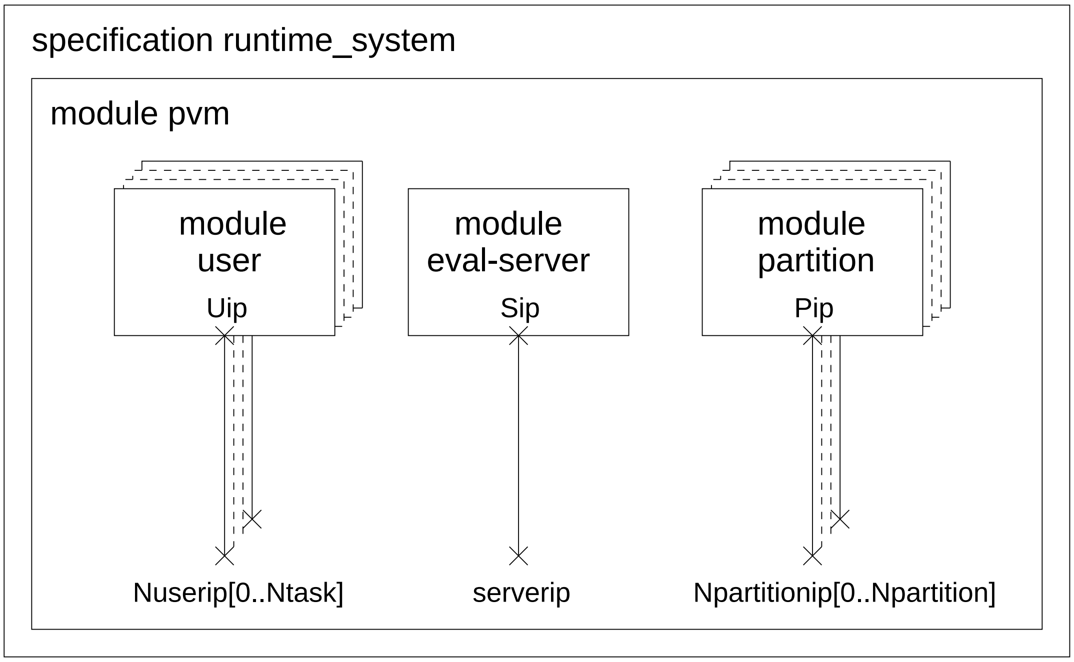


Abbildung 8.1: Die graphische Darstellung der Estelle-Spezifikation mit allen Modultypen und ihren IPs.

⟨*⟩≡
  ⟨*Das Protokoll für das Verteilte Hashing*⟩

⟨*Das Protokoll für das Verteilte Hashing*⟩≡

```
specification runtime_system;
default individual queue;          { Es gibt nur individuelle Eingabequeues }
⟨Die Konstanten⟩
⟨Die Typdefinitionen⟩
⟨Die globalen Varibalen⟩
⟨Das Modul PVM⟩
```

Die folgenden Konstanten legen die maximale Anzahl der startbaren Tupelräume, Partitionen und Benutzerprozesse fest. Dies geschieht, da in *Estelle* die Länge von Arrays fest angegeben werden muß.

⟨*Die Konstanten*⟩≡

```
const Npartition = 20;       { Die Anzahl der Partitonen pro TS. }
      Nts        = 5;        { Die Anzahl der Tupelräume. }
      Ntask      = 15;       { Die Anzahl der Benutzerprozesse. }
```

Im bestehenden Laufzeitsystem sind die Datenstrukturen für Tupel und Templates komplexe Zeigerstrukturen. In Estelle können keine Zeigerstrukturen zwischen Moduln versendet werden. Um diese dynamischen Strukturen zu modellieren, müßten zumindest File-Operationen (im Sinne von ISO-Pascal), d.h. Sequenzen und dazugehörige Operationen zur Verfügung stehen. Diese sind in der Estelle-Norm aber nicht mehr enthalten. Deshalb kann hier keine Umsetzung der Zeigerstrukturen stattfinden. Um die eigentlichen Protokolle zu spezifizieren, sind die zu verschickenden Datenstrukturen aber auch sekundär. Deshalb wurden hier ein Dummy-Datentypen (Char) für den `tuple_type` und `template_type` gewählt.

⟨*Die Typdefinitionen*⟩≡

```
type partid_type    = 0..Npartition;       { Die IDs der Partitionen. }
     tsid_type      = 0..Nts;              { Die IDs der Tupelräume. }
     taskid_type      = 0..Ntask;          { Die IDs der User-Prozesse. }
     tuple_type     = Char;                { Der Dummy-Typ der Tuple. }
     templates_type  = Char;                                 { Der Dummy-
  Typ der Templates. }
     object_type     = Char;                                 { Der Dummy-
  Typ der Rückgabeobjekt. }
     string22 = packed array [1..22] of Char;      { Stringdefinitionen. }
     string = packed array [1..80] of Char;
```

⟨*Die globalen Variablen*⟩≡

```
var into_flag : Boolean;                 { Flag für meet_into }
    block_flag : Boolean;                { Flag für blockierenden oder nicht
blockierenden TS-Zugriff. }
```

In *Estelle* können nur Module dynamisch andere Module erzeugen. Deshalb gibt es in dieser Spezifikation ein Hauptmodul PVM, welches die Prozeßkreation übernimmt. Außerdem gibt es Nachrichten an den Adressaten weiter. Das Modul übernimmt damit also die Rolle von PVM im späteren System. Alle anderen Moduln sind Sohnmoduln dieses Moduls.

⟨*Das Modul PVM*⟩≡

```
module pvm systemprocess;
end;
body pvm_body for pvm;
⟨Die Konstanten von PVM⟩
⟨Die Variablen von PVM⟩
⟨Die Prozeduren⟩
⟨Die Kanaldefinitionen⟩
⟨Das Modul user⟩
⟨Das Modul partition⟩
⟨Das Modul server⟩
⟨Der eigentliche Body⟩
```

⟨*Die Konstanten von PVM*⟩≡

```
const empty_tuple = '';                  { Ein leeres Tupel }
```

Die globalen Variablen des Moduls PVM dienen zur Verwaltung der IDs von Partitionen und laufenden Prozessen.

⟨*Die Variablen von PVM*⟩≡

```
var x : taskid_type;
    running_partition : Array[1..Npartition] of Boolean;
    running_task      : Array[1..Ntask] of Boolean;
    number_of_tasks   : Integer;
    number_of_ts      : Integer;
    number_of_partition : Integer;
    bottom, top    : Integer;
```

Die Prozedur `kill_all` beendet alle Module im Fehlerfall. Eine solche Funktionalität gibt es in PVM ebenfalls, weswegen diese hier nicht genauer beschrieben wird und als `primitive` ausgewiesen wird.

⟨*Die Prozeduren*⟩≡

```
procedure kill_all(message : string22);
primitive;
```

Das Modul PVM kommuniziert mit seinen Söhnen (User-Prozesse, Partitionen) über Kanäle, die von seinen internen IPs zu den externen IPs seiner Sohnmoduln führen. Diese sowie die Nachrichten, die über die Kanäle versendet werden, werden im folgenden definiert:

⟨*Die Kanaldefinitionen*⟩≡

⟨*Der Kanal zwischen den Benutzerprozessen und PVM*⟩
⟨*Der Kanal zwischen den Partitionen und PVM*⟩
⟨*Der Kanal zwischen dem Server fuer aktive Tuple und PVM*⟩

In Estelle sind in den Kanaldefinitionen alle Nachrichtentypen aufgeführt, die von den einzelnen kommunizierenden Moduln benutzt werden können.

Zuerst wird der Kanal zwischen den Benutzerprozessen und PVM definiert. Hier finden sich alle Tupelraumoperationen, der ||-Operator, die Befehle der Standardlibrary und der Ergebniswert wieder.

⟨*Der Kanal zwischen den Benutzerprozessen und PVM*⟩≡

```
channel U_PVMchn(pvm, user);
  by user :                { Die möglichen Nachrichten der Benutzerprozesse }
    deposit(p_id : partid_type; tuple: tuple_type);
    fetch(p_id : partid_type; templates : templates_type;
          block_flag : Boolean);
    meet(p_id : partid_type; templates : templates_type;
              into_flag: Boolean; block_flag : Boolean);
    create_task;
    create_ts;                          { Die Befehle der Standardlibrary }
    clear_ts(ts_id : tsid_type);
    exists_ts(ts_id : tsid_type);
    remove_ts(ts_id : tsid_type);
    deposit_active_tuple(tuple : tuple_type; ts_id : tsid_type);
    return_value(object : object_type);                 { Ergebniswert }

  by pvm :                          { Die möglichen Nachrichten von PVM }
    fetch_conf(tuple : tuple_type);
    meet_conf(tuple : tuple_type);
    create_task_conf(task_id : taskid_type);
    create_ts_conf(ts_id : tsid_type);
    exists_ts_conf(flag : Boolean);
```

Der folgende Kanaltyp verbindet die Partitionen mit PVM. Hier finden sich vor allem die Nachrichten wieder, die die Tupelraumoperationen verwirklichen. Aber auch der Befehl zum Löschen der Tupel der Partitionen findet sich mit `clear_partition_ind` wieder.

⟨*Der Kanal zwischen den Partitionen und PVM*⟩≡

```
channel P_PVMchn(pvm, partition);
  by partition :            { Die möglichen Nachrichten von den Partitionen }
    fetch_resp(task_id : taskid_type; tuple : tuple_type);
    meet_resp(task_id : taskid_type; tuple : tuple_type);
    deposit(p_id : partid_type; tuple : tuple_type);
    deposit_active_tuple(tuple : tuple_type; ts_id : tsid_type);
  by pvm :                              { Die möglichen Nachrichten von PVM }
    deposit_ind(tuple : tuple_type);
    fetch_ind(task_id : taskid_type; templates : templates_type;
              block_flag : Boolean);
    meet_ind(task_id : taskid_type; templates : templates_type;
             into_flag : Boolean; block_flag : Boolean);
    clear_partition_ind;
```

Der letzte Kanaltyp verbindet den Eval-Server mit PVM. Neben den Nachrichten zur Behandlung von aktiven Tupeln und der Verwaltung der globalen Liste der Tupelräume sind auch die Nachrichten für die Standardlibrary-Funktionen definiert.

⟨*Der Kanal zwischen dem Server fuer aktive Tuple und PVM*⟩≡

```
channel S_PVMchn(pvm, server);
  by server :
    deposit(p_id : partid_type; tuple: tuple_type);
    kill_partition(p_id : partid_type);
    clear_partition(p_id : partid_type);
    exists_ts_resp(task_id : taskid_type; flag : Boolean);
  by pvm :
    deposit_active_tuple_ind(tuple : tuple_type; ts_id : tsid_type);
    clear_ts_ind(ts_id : tsid_type; task_id : taskid_type);
    exists_ts_ind(ts_id : tsid_type; task_id : taskid_type);
    remove_ts_ind(ts_id : tsid_type; task_id : taskid_type);
    new_ts(ts_id : tsid_type);
    return_value_ind(task_id : taskid_type; object : object_type);
```

Nachdem die Definition der Kanäle abgeschlossen ist, folgt die Definition der Moduln. Zuerst das Modul der Benutzerprozesse:

Der Body dieses Modultyps ist leer, d.h. sein Verhalten bezüglich der Reihenfolge seiner Nachrichten ist unbestimmt. Dies kann auch nicht anders sein, denn die Kontrollflüsse der Benutzermoduln werden durch das vom Programmierer erstellte PRO-SET-Programm bestimmt.

⟨*Das Modul user*⟩≡

```
module user process (tsparent : tsid_type; userparent : taskid_type;
            tuple : tuple_type);
ip Uip : U_PVMchn(user);
end;                                                { module definition }
body user_body for user;
end;                                                { user_body }
```

Nun folgt die Definition des Modultyps für die Partitionen:

⟨*Das Modul partition*⟩≡

```
module partition process (ts : tsid_type);
ip Pip : P_PVMchn(partition);
end;
body partition_body for partition;
⟨Die Typdefinitionen der Partitionen⟩
⟨Die Variablen der Partitionen⟩
⟨Die Prozeduren der Partitionen⟩
⟨Die Funktionen der Partitionen⟩
⟨Die Transitionen der Partitionen⟩
end;                                                { partition_body }
```

Die Variablen `my_ts` und `my_pid` enthalten die ID des Tupelraums und die ID der Partition. Das Modul ist zustandlos bzw. besitzt nur einen Zustand.

⟨*Die Variablen der Partitionen*⟩≡

```
var my_ts     : tsid_type;
    my_pid    : partid_type;

state STATELESS;
```

⟨*Die Prozeduren der Partitionen*⟩≡

```
procedure assign_p_list(operation : char; task_id : taskid_type;
                        templates : templates_type);
                      { Hängt den Prozeß in die Liste der wartenden Prozesse }
  primitive;

procedure kill_p_list;
  primitive;
```

⟨*Die Funktionen der Partitionen*⟩≡

```
procedure op_deposit(tuple : tuple_type);
            { Verwirklicht den deposit-Zugriff auf den Tupelraumspeicher. Wartet
ein Prozeß auf tuple, so wird er aus der Liste der wartenden
Prozesse gelöscht }
  primitive;

function op_fetch(templates : templates_type): tuple_type;
                  { Verwirklicht den fetch-Zugriff auf den Tupelraumspeicher }
  primitive;
function op_meet(templates : templates_type;
      into_flag : Boolean): tuple_type;
                  { Verwirklicht den meet-Zugriff auf den Tupelraumspeicher }
  primitive;
function hashing_in(tuple : tuple_type): partid_type;
                                    { Verwirklicht das Hashing der Tupel }
  primitive;
function active(tuple : tuple_type): Boolean;
  primitive;
```

⟨*Die Transitionen der Partitionen*⟩≡

```
initialize to STATELESS begin end;  { Das Modul besitzt nur einen Zustand }

trans
from STATELESS to same
  when Pip.deposit_ind(tuple)
    begin
      op_deposit(tuple);
    end;

  when Pip.fetch_ind(task_id, templates, block_flag)
    var tuple: tuple_type;
    begin
      tuple := op_fetch(templates);          { Zugriff auf den TS-Speicher }
      if (tuple = '') ∧ (block_flag = True) then
          assign_p_list('f', task_id, templates)
      else
        output Pip.fetch_resp(task_id, tuple);
    end;

  when Pip.meet_ind(task_id, templates, into_flag, block_flag)
    var tuple: tuple_type;
        pid  : partid_type;
    begin                                    { Zugriff auf den TS-Speicher }
      tuple:= op_meet(templates, into_flag);
      if (tuple = '') ∧ (block_flag = True) then
        assign_p_list('m', task_id, templates)
      else
        begin
          if active(tuple) then
          output Pip.deposit_active_tuple(tuple, my_ts);
          if into_flag = True then
          begin
            pid := hashing_in(tuple);
            if pid <> my_pid then
               output Pip.deposit(pid, tuple)
          end
        end;
```

```
        output Pip.meet_resp(task_id, tuple)
      end;

    when Pip.clear_partition_ind
      begin
        kill_p_list;
      end;
```

Das Modul `server` beschreibt die Funktionalitäten des späteren Eval-Servers.

⟨*Das Modul server*⟩≡

```
module server process;
ip Sip : S_PVMchn(server);
end;
body server_body for server;
state STATELESS;
⟨Die Prozeduren des Eval-Servers⟩
⟨Die Funktionen des Eval-Servers⟩
⟨Die Transitionen des Eval-Servers⟩
end;
```

Die Prozeduren und Funktionen des Eval-Servers dienen vor allen Dingen zur Verwaltung der Liste der aktiven Tupel und der Tupelräume.

⟨*Die Prozeduren des Eval-Servers*⟩≡

```
procedure assign_tuple_list(tuple : tuple_type; ts_id : tsid_type);
                          { Hängt ein Element an die Liste der aktiven Tupel }
primitive;

procedure kill_from_tuple_list(ts_id : tsid_type);
                          { Löscht ein Element aus der Liste der aktiven Tupel }
primitive;

procedure assign_ts_list(ts_id : tsid_type);
                          { Hängt ein Element an die Liste der Tupelräume }
primitive;

procedure remove_tuples_list(p_id : partid_type);
                          { Löscht ein Element aus der Liste der TRs }
primitive;
```

⟨*Die Funktionen des Eval-Servers*⟩≡

```
function search_ts_list(ts_id : tsid_type): boolean;
                                            { Durchsucht die Liste der TRs }
primitive;

function make_passive(taskid : taskid_type;
                                  object : object_type) : tuple_type;
                              { Hängt ein Ergebnisobjekt in ein aktives Tupel }
primitive;

function hashing(tuple : tuple_type) : partid_type;
                                          { Hasht ein fertiges passives Tupel }
primitive;
```

Die Transitionen beschreiben das Verhalten des Servers. Auch er ist zustandslos, wartet also auf eine Nachricht und arbeitet diese sofort ab. Die Arbeit besteht zumeist aus Listenverwaltung.

⟨*Die Transitionen des Eval-Servers*⟩≡

```
initialize to STATELESS
  begin end;

trans
  when Sip.deposit_active_tuple_ind(tuple, ts_id)
    var p_id : partid_type;
    begin
      assign_tuple_list(tuple, ts_id);
    end;

  when Sip.clear_ts_ind(ts_id, task_id)
    var first_partition : partid_type;
                  i : integer;
    begin
      first_partition := ((ts_id * (Npartition-1)) +1);
      for i := first_partition to (first_partition + Npartition) do
      begin
             remove_tuples_list(i);
        output Sip.clear_partition(i);
      end;
    end;

  when Sip.new_ts(ts_id)
    begin
      assign_ts_list(ts_id);
    end;

  when Sip.exists_ts_ind(ts_id, task_id)
    var flag : Boolean;
    begin
      flag := search_ts_list(ts_id);
      output Sip.exists_ts_resp(task_id, flag);
    end;
```

```
when Sip.remove_ts_ind(ts_id, task_id)
  var first_partition : partid_type;
                i : integer;
  begin
    first_partition := ((ts_id * (Npartition-1)) +1);
    for i := first_partition to (first_partition + Npartition) do
    begin
           remove_tuples_list(i);
      output Sip.kill_partition(i);
    end;
  end;

when Sip.return_value_ind(task_id, object)
   var p_id : partid_type;
       tuple : tuple_type;
   begin
     tuple := make_passive(task_id, object);
     if tuple <> empty_tuple then
     begin
       p_id := hashing(tuple);
       output Sip.deposit(p_id, tuple);
     end
   end;
```

Nun folgt der eigentliche Body des Moduls PVM, der die Umsetzung und Weiterleitung der Nachrichten vornimmt. Das Modul besitzt zu jedem Tochtermodultyp einen internen Kanal.

⟨*Der eigentliche Body*⟩≡

```
ip                                { Die Kommunikationsknoten von PVM }
  Nuserip      : array [0..Ntask] of U_PVMchn(pvm);
  Npartitionip : array [0..Npartition] of P_PVMchn(pvm);
  serverip     : S_PVMchn(pvm);

modvar                            { Die IDs für die Kommunikationsknoten }
   userid : array[1..Ntask] of user;
   pid    : array[1..Npartition] of partition;
   mother : user;
   serverid : server;
initialize
  begin                           { Nur der Benutzerprozess wird gestartet }
      init mother with user_body(0, 0, empty_tuple);
      init serverid with server_body;
      connect Nuserip[1] to userid[1].Uip;
  end;

trans
  when Nuserip[x].deposit(p_id, tuple)
    begin
      if (running_partition[p_id] = True) then
        output Npartitionip[p_id].deposit_ind(tuple)
      else
        kill_all('cannot find partition ');
    end;

  when Nuserip[x].fetch(p_id, templates, block_flag)
    begin
      if (running_partition[p_id] = True) then
        output Npartitionip[p_id].fetch_ind(x, templates, block_flag)
      else
        kill_all('cannot find partition ');
    end;
```

```
when Nuserip[x].meet(p_id, templates, into_flag, block_flag)
  begin
    if (running_partition[p_id] = True) then
      output Npartitionip[p_id].meet_ind(x, templates, into_flag,
                                         block_flag);
  end;

when Nuserip[x].create_task
  begin
    if number_of_tasks ≤ Ntask then
      begin
        number_of_tasks := number_of_tasks + 1;
        init userid[number_of_tasks]
                      with user_body(0, x, empty_tuple);
        connect Nuserip[number_of_tasks] to
          userid[number_of_tasks].Uip;
        running_task[number_of_tasks] := True;
        output Nuserip[x].create_task_conf(number_of_tasks);
      end
    else kill_all('too many tasks        ');
  end;

when Nuserip[x].create_ts
  var i : Integer;
  begin
    if number_of_ts ≤ Nts then
      begin
        number_of_ts := number_of_ts + 1;
        top := number_of_ts * Npartition;
        bottom := (top - Npartition) + 1;
        for i := bottom to top do
           begin
             init pid[i] with partition_body(number_of_ts);
             connect Npartitionip[i] to pid[i].Pip;
             running_partition[i] := True;
             output Nuserip[x].create_ts_conf(number_of_ts);
           end;
```

```
        output serverip.new_ts(number_of_ts);
      end
    else kill_all('cannot create TS      ');
  end;

when Nuserip[x].clear_ts(ts_id)
  begin
    output serverip.clear_ts_ind(ts_id, x);
  end;

when Nuserip[x].exists_ts(ts_id)
  begin
    output serverip.exists_ts_ind(ts_id, x);
  end;

when Nuserip[x].remove_ts(ts_id)
  begin
    output serverip.remove_ts_ind(ts_id, x);
  end;

when Nuserip[x].deposit_active_tuple(tuple, ts_id)
  begin
    output serverip.deposit_active_tuple_ind(tuple, ts_id);
  end;

when Nuserip[x].return_value(object)
  begin
    output serverip.return_value_ind(x, object);
  end;


when Npartitionip[x].fetch_resp(task_id, tuple)
  begin
    if running_task[task_id] = True then
      output Nuserip[task_id].fetch_conf(tuple)
    else kill_all('cannot find task      ');
  end;
```

```
when Npartitionip[x].meet_resp(task_id, tuple)
  begin
    if running_task[task_id] = True then
      output Nuserip[task_id].meet_conf(tuple)
    else kill_all('cannot find task      ');
  end;

when Npartitionip[x].deposit_active_tuple(tuple, ts_id)
  begin
    output serverip.deposit_active_tuple_ind(tuple, ts_id);
  end;

when serverip.deposit(p_id, tuple)
  begin
    if (running_partition[p_id] = True) then
      output Npartitionip[p_id].deposit_ind(tuple)
    else
      kill_all('cannot find partition ');
  end;

when serverip.clear_partition(p_id)
  begin
    if (running_partition[p_id] = True) then
      output Npartitionip[p_id].clear_partition_ind
    else
      kill_all('cannot find partition ');
  end;

when serverip.exists_ts_resp(task_id, flag)
  begin
    if running_task[task_id] = True then
       output Nuserip[task_id].exists_ts_conf(flag)
    else kill_all('cannot find task      ');
  end;

when serverip.kill_partition(p_id)
  begin
    disconnect Npartitionip[p_id];
```

```
    end;

end;

modvar
   pvmid : pvm;

initialize
  begin                                                { PVM wird gestartet }
    init pvmid with pvm_body;
  end;
end.
```

# Teil IV

# Die Implementierung

# 9. Die Erfüllung der Vorgaben

Nun soll kurz genannt werden, welche Vorgaben durch das implementierte Laufzeitsystem erfüllt wurden:

- Das erstellte Laufzeitsystem ist durch die Verwendung der Externen Datenrepräsentation (XDR) von PVM voll für heterogene Netze geeignet. Die Übersetzung in die XDR wird auf homogenen Netzen vom System automatisch ausgeschaltet.
- Der bisherige ProSet-Compiler muß für das neue Laufzeitsystem nur an einer einzigen Stelle geändert werden: Der Aufruf der Funktion `plp_init_linda` besitzt jetzt zwei Parameter (siehe übernächsten Abschnitt). Alle anderen Änderungen des generierten C-Codes sind durch Fehler im Compiler begründet. Die Änderungen werden im Abschnitt 10.3 beschrieben.
- Alle Protokolle, die die Tupelraumbefehle betreffen, wurden implementiert. Zusätzlich wurden auch die in Abschnitt 2.3.3 auf Seite 9 beschriebenen Befehle der Standardbibliothek verwirklicht.
- Der entfernte Start von Threads wurde ebenfalls mit dem `case-main`-Ansatz verwirklicht.
- Zur selbstgesteckten Aufgabe, der Erstellung einer Loadbalancierung, wurden zwei Strategien implementiert. Die erste, welche immer den unbelastetsten Host für den Prozeßstart bestimmte, wurde wieder verworfen, da Fälle auftraten, bei denen alle Prozesse auf dem gleichen Rechner gestartet wurden. Da dann keine echte Parallelität genutzt werden kann, wurde dieser Ansatz verworfen. Die zweite Strategie gibt anhand einer oberen Grenze relativ unbelastete Hosts zurück. Die Effektivität dieser Methode wurde allerdings nicht mehr getestet.

Alle Einschränkungen des Systems werden im nächsten Abschnitt genannt.

# 10. Probleme, die während der Implementierung auftraten

Während der Implementierungsphase traten sehr viele Probleme auf, die zumeist aus der Weiterverwendung des alten Kernsystems resultierten. Der größte Teil von ihnen konnte gelöst werden. In diesem Abschnitt werden aber einige Probleme beschrieben, die Einschränkungen des Systems bzw. Workarounds nötig machten.

## 10.1 Das Versenden von Umgebungen

Die *Closure*-Bildung einer Prozedur p() in ProSet speichert die gesamte Umgebung dieser Prozedur. Auf diese Umgebung greift die eingefrorene Prozedur bei ihrer Ausführung zu. Diese Umgebung beinhaltet alle für die Prozedur sichtbaren globalen Variablen und weiter Prozeduren, die wiederum komplette Umgebungen besitzen.

Durch die komplizierte C-Datenstruktur der Variablen und Funktionen war eine Versendung der gesamten Umgebungsstruktur sehr schwierig. Schon die Realisierung der Versendung der globalen Variablen war sehr komplex und zeitaufwendig.

Da es möglich war, interne Prozeduren der Closure-Objekte aufzurufen, wurde die Versendung der Umgebung auf globale Variablen beschränkt. Dies bedeutet, daß entfernt gestartete Prozeduren (Threads) derzeit nur lokale Prozeduren aufrufen können. In den meisten Fällen reicht dies aber aus, da der Code der Worker meist gekapselt ist. Gegebenenfalls müssen globale Prozeduren in lokale umgewandelt werden.

## 10.2 Die Anzahl der Parameter der Threads

Im alten Laufzeitsystem werden zur Argumentenübergabe bei Threadaufrufen Parameterlisten variabler Länge benutzt, die mit Hilfe der `va_arg`-Makros gehandhabt werden. Diese Parameterliste wird für den Thread-Aufruf mehrfach übergeben.

Dieser Mechanismus funktioniert nur in der sequentiellen Version des alten Laufzeitsystems. In der Version, die mit Threads arbeitet, kommt es bereits zu Fehlern. Auch in der parallelen Version dieser Arbeit konnte trotz extremster Fehlersuche keine fehlerfreie Parameterübergabe erstellt werden. Daher darf in dieser Version ein Thread nur einen Parameter besitzen[1]

## 10.3 Probleme mit dem ProSet-Compiler

Der ProSet-Compiler befand sich während dieser Arbeit noch in der Entwicklung. Daher war er nicht fehlerfrei. Für die Arbeit wurde die Version 0.5 benutzt. Eine Umstellung auf die Version 0.6 wurde durch eine zu große Fehleranfälligkeit der neuen Version verhindert.

Im generierten C-Code mußten einige Änderungen vorgenommen werden, die im einzelnen nun an Beispielen erklärt werden. Werden diese Änderungen nicht alle durchgeführt, kommt es entweder zu Fehlern während des C-Compilerlaufs oder das Programm arbeitet nicht parallel.

Wird im ProSet-Programm der ||-Operator mit Zuweisung benutzt, dann muß im C-Code der betreffende Aufruf der Funktion `plp_spawn` geändert werden.

Falscher Code:

```
gpt_0 = plp_spawn(pbp_act_par,
pbu_closure(pbu_env_obj(gps_env[5])/*worker*/,
 pbp_act_par), gps_env, gps_return_status, gps_return_address);
pbh_abort_test( gpt_0 );
```

[1]Kurz vor Beendigung der Arbeit konnte der wahrscheinliche Fehler gefunden werden: In den Manual-Seiten der SUN-Version der Macro-Library wird eine Weitergabe der Parameterliste ausdrücklich erlaubt. Allerdings muß dann auf die einzelnen Argumente nicht mit einer Variable vom Typ `va_list`, sondern mit einem Zeiger auf eine Variable vom Typ `va_list` zugegriffen werden. Dies wird im Code des verwendeten Kernsystems nicht getan. Allerdings konnte dieser Vermutung aus Zeitgründen nicht mehr nachgegangen werden.

```
pbo_assign(&gph_z1,gpt_0 );
```

Berichtigung: Ersetze `gpt_0` durch die verwendete Zuweisungsvariable (in diesem Falle `gph_z1` und lösche den `pbo_assign`-Aufruf.

```
gph_z1 = plp_spawn(pbp_act_par,
pbu_closure(pbu_env_obj(gps_env[5])/*worker*/,
 pbp_act_par), gps_env, gps_return_status, gps_return_address);
pbh_abort_test( gph_z1 );
```

Obwohl die Version 0.5 des Compilers zu Beginn dieser Diplomarbeit freigegeben war, wurden noch Änderungen vorgenommen. Durch diese Änderungen war der Code von `main()` betroffen. Außerdem mußte zur Verwirklichung der `case-main`-Technik die Funktion `plp_init_linda()` um zwei Parameter erweitert werden.

Falscher Code:

```
int main(int argc, char *argv[])
{
/*pbp_init_library(argc,argv,&gui_linda);*/
 plp_init_linda();
pbp_init_library(argc,argv,&gui_linda);
 /*ppo_init(argc,argv);*/
Name_des_Hauptprogrammes();
plp_finish_linda(0);
}
```

Berichtigung: Kommentiere den ersten `pbp_init_library`-Aufruf aus, den zweiten dafür ein. Füge `argc` und `argv` als Argumente von plp_init_linda ein.

```
int main(int argc, char *argv[])
{
pbp_init_library(argc,argv,&gui_linda);
 plp_init_linda(argc,argv);
 /*pbp_init_library(argc,argv,&gui_linda);*/
 /*ppo_init(argc,argv);*/
Name_des_Hauptprogrammes();
plp_finish_linda(0);
}
```

## 10.4 Probleme mit PVM

Zuerst einmal soll erwähnt werden, daß PVM ein sehr stabiles System ist, mit dem sich gut arbeiten läßt. Auch das Visualisierungswerkzeug `xpvm` ist sehr hilfreich.

Es konnte nur ein wirklicher Fehler gefunden werden, der aber für die Arbeit nicht folgenlos war. Leider funktioniert das Multicasting nicht richtig. In einigen Situationen, wenn zwei Prozesse, die auf dem gleichen Host laufen, an eine Gruppe Nachrichten versenden, kommt es zum Verlust entweder einer oder beider Nachrichten. Dieser Fehler ist im Visualisierungswerkzeug `xpvm` nicht zu erkennen, da dort keine Multicasting-Nachrichten dargestellt werden. Deshalb konnte nicht in allen Fällen Gruppenkommunikation eingesetzt werden, wodurch sich die Performance des Systems verringerte.

# 11. Die Benutzung des Systems

## 11.1 Die Konfigurationsdatei

Die Konfigurationsdatei dient zur Voreinstellung einiger Optionen des Laufzeitsystems. Zusätzlich gibt es noch eine Konfigurationsdatei für PVM, in der im wesentlichen nur die virtuelle Maschine durch die Rechnernamen definiert wird. Diese wird hier nicht beschrieben. Eine genaue Beschreibung findet sich in [Geist *et al.*, 1994].

Die Konfigurationsdatei des Laufzeitsystems hat den festen Namen `.netproset` und muß sich im Homeverzeichnis des Benutzers befinden. Mit Hilfe der Konfigurationsdatei können folgende Einstellungen vorgenommen werden:

- Die Loadbalancierung kann ein- oder abgeschaltet werden. Standardmäßig ist sie abgeschaltet. Zum Einschalten muß das Wort `loadbalancing` am Anfang einer Zeile stehen.
- Die Anzahl der Partitionen pro Tupelraum kann durch das Wort `partitions` und einer ganzen Zahl angegeben werden. Standardmäßig sind zwei Partitionen vorgegeben. Beispiel: `partitions 5`
- Der Hashalgorithmus kann mit dem Wort `hashmode` und einer ganzen Zahl ausgewählt werden. Ein Standardhashalgorithmus, der sowohl die Länge als auch die Datentypen der Elemente berücksichtigt, ist auf 0 voreingestellt. Beispiel: `hashmode 3`. Eine genauerer Beschreibung der entgültig verwendeten Hashalgorithmen findet sich im Quelltext ab Abschnitt C.2.4 auf Seite 171.

Alle Optionen müssen jeweils einzeln in einer Zeile stehen. Die Reihenfolge der Optionen ist beliebig. Fehlen Optionen, erscheint eine Warnung auf dem Bildschirm und die Standardeinstellung wird benutzt.

## 11.2 Der Übersetzungsvorgang und der Start eines ProSet-Programms

Der Übersetzungsvorgang ist im Vergleich zum alten System gleichgeblieben. Allerdings müssen die beiden PVM-Bibliotheken `pvm3` und `gpvm3` zusätzlich gebunden werden. Außerdem muß natürlich das neue Laufzeitsystem anstatt des alten eingebunden werden. Aus diesen Gründen kann das ProSet-Werkzeug *xpst* nur beschränkt benutzt werden.

Der vom ProSet-C-Compiler generierte C-Code muß teilweise verändert werden. Alle diese Veränderungen sind aber **nicht** durch das neue Laufzeitsystem bedingt, sondern resultieren aus Fehlern bei der Code-Generierung. Diese notwendigen Berichtigungen werden im nächsten Abschnitt an Beispielen erklärt.

Zum Start des übersetzten Programms muß zunächst PVM mit dem Befehl `pvm` gestartet werden. Nützlich wäre hier die Angabe der PVM-Konfigurationsdatei, um den Start der virtuellen Maschine zu erleichtern.

Um ein Programm aufzurufen, sollte PVM wieder durch den Befehl `quit` verlassen werden. Dann können ein oder mehrere Programme aus der normalen Shell gestartet werden. Nach dem Lauf aller Programme muß wieder mit `pvm` zur PVM-Befehlszeile zurückgekehrt werden. Der Befehl `halt` beendet PVM.

Ein Starten und Beenden von PVM durch das Laufzeitsystem selber während der Laufzeit ist möglich, aber noch nicht implementiert. Der Start der virtuellen Maschine benötigt aber auch einige Zeit, die dann der Laufzeit des ProSet-Programms zugerechnet werden müßte.

# Teil V

# Bewertung des Systems und Fazit

# 12. Die Performance-Tests

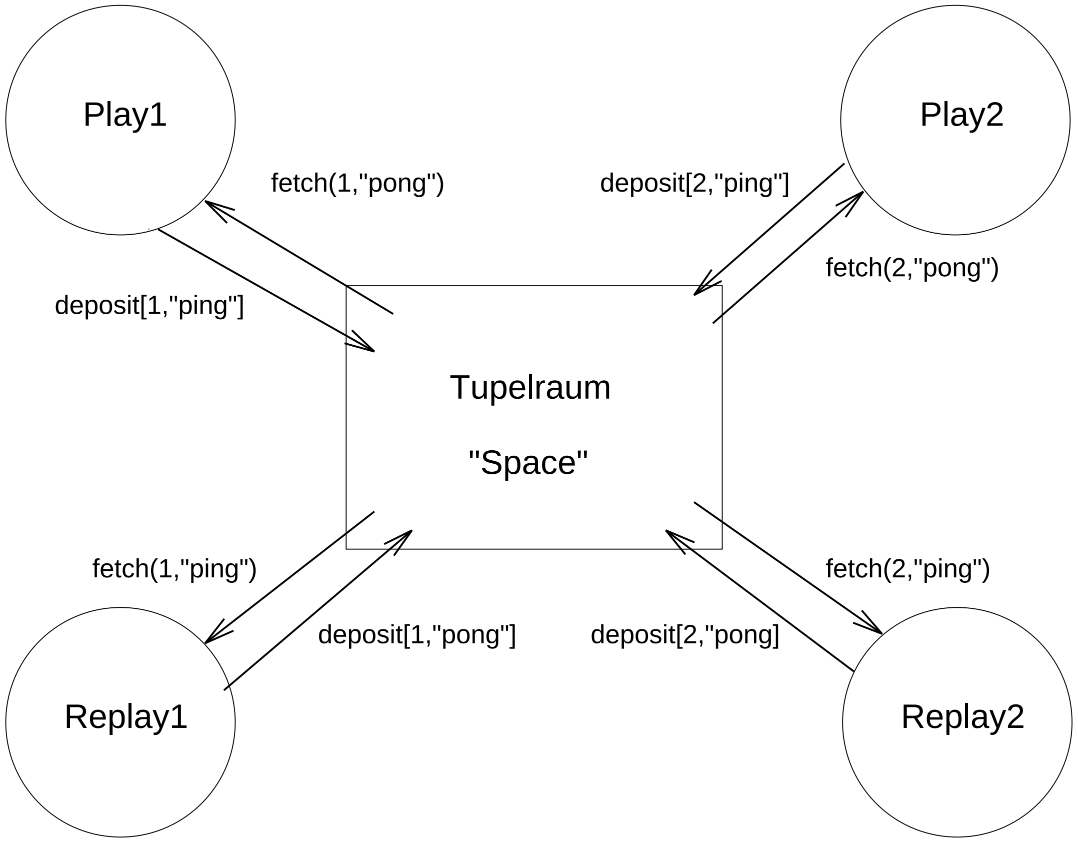


Abbildung 12.1: Die Kommunikationsstruktur des Programms „Ping_Pong“.

Nun werden die Laufzeittests dieser Arbeit diskutiert. Entsprechend den folgenden Fragestellungen wurden die Tests durchgeführt:

- Tritt beim neuen parallelen Laufzeitsystem eine Laufzeitverbesserung gegenüber dem alten quasi-parallen System auf?
- Kann mit dem Ansatz des Verteilten Hashings Performance-Verbesserungen gegenüber dem Zentralem-Server-Ansatz erzielt werden?
- Kann auf dem verteilten System durch den Ansatz der multiplen Tupelräume eine Laufzeitverbesserung gegenüber dem Ansatz der singulären Tupelräume erreicht werden?

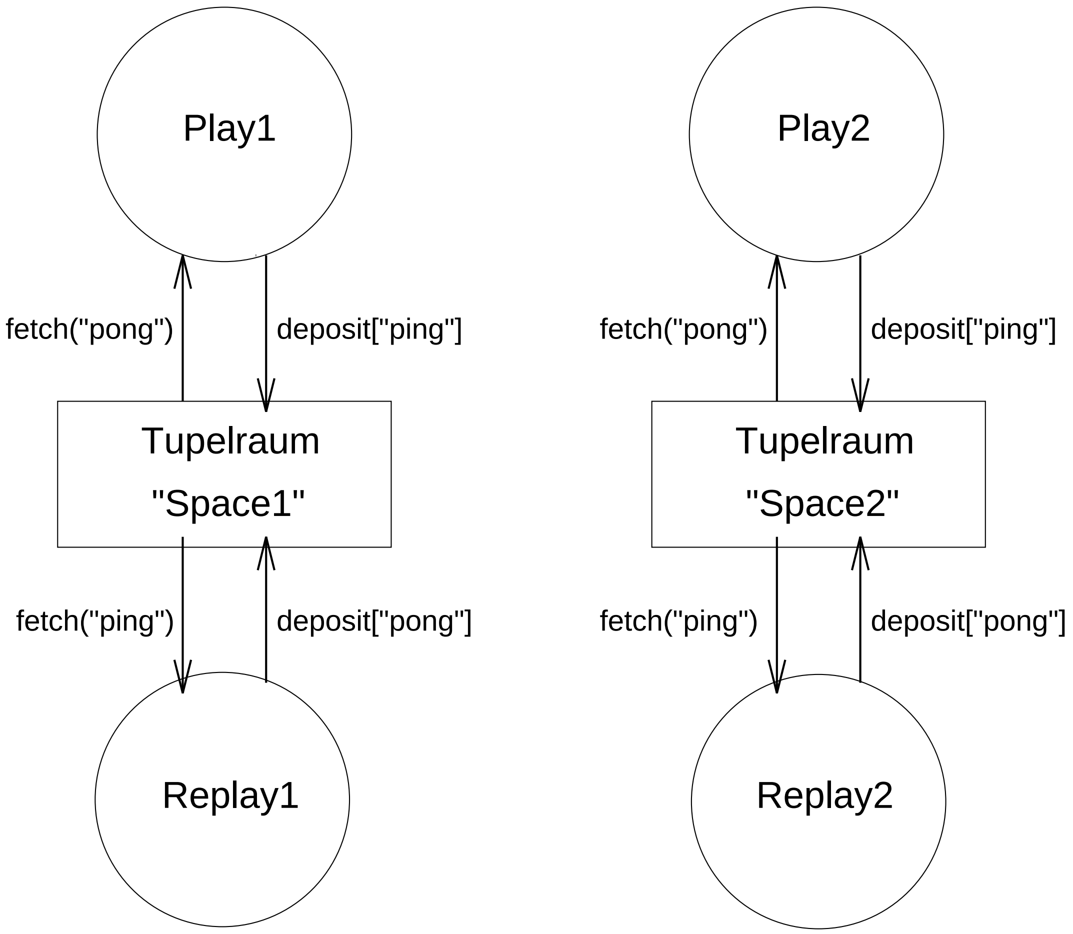


Abbildung 12.2: Die Kommunikationsstruktur des Programms „Ext_Ping_Pong“.

Außerdem ergab sich die Gelegenheit, das verteilte ProSet-System mit dem komerziellen Produkt C-Linda vergleichen zu können. Dabei stand natürlich kein direkter Vergleich der Laufzeiten im Vordergrund, sondern das Verhalten der Laufzeitkurven bei Veränderung der Anzahl der Arbeitsprozesse (*Worker*).

## 12.1 Wie wurde gemessen?

Nach [Crowl, 1994] und [Schumann und Lobinger, 1994] sind die Messungen der Realzeit bei parallelen Programmen am aussagekräftigsten. Die Realzeit ist die Zeit, welche das Programm vom Programmstart bis zur Beendigung benötigt. Diese Zeit wurde deshalb in den Laufzeittests dieser Arbeit verwendet.

Die Messungen wurden mit dem System V Befehl `time` durchgeführt, der die Realzeit mit einer Genauigkeit von einer Zehntel-Sekunde liefert.

Durchgeführt wurden die Messungen auf folgender Hardware:

- Die Laufzeittests des alten quasi-parallelen Systems wurde auf einer 2-Prozessor-Workstation von SUN (SPARCstation 10-512) gefahren.
- Die Messungen des verteilten Laufzeitsystems auf einem Host wurden auf einer 1-Prozessor-Workstation von SUN (SPARCstation10) durchgeführt.
- Alle Netzwerkmessungen wurden auf einem 10-MBit-Ethernet und 5 modellgleichen 1-Prozessor-Workstations von SUN (SPARCstation10) gefahren.

Für jeden Meßpunkt wurden mindestens drei Messungen vorgenommen. Während der Messungen für einen Punkt zeigten sich teilweise Laufzeitverbesserungen. Dieser Effekt hängt mit der verwendeten Caching-Technik zusammen, die auf den Hosts verwendet wird. Nach [Crowl, 1994] soll das Caching entweder ausgeschaltet oder die ersten Messungen ignoriert und weiter Test gefahren werden. Letztere Methode wurde in den Tests dieser Arbeit angewendet.

Während der Tests war im neuen System die Loadbalancierung immer ausgeschaltet.

Die Auswahl geeigneter ProSet- und C-Linda-Programme für die Messungen warf einige Probleme auf. Seit Beginn der Arbeit war geplant, ProSet-Implementierungen der Salishan- und Cowichan-Probleme zu benutzen. Diese zehn Probleme dienen dem Vergleich von parallelen Programmiersprachen und wurden größtenteils von Peter Jodeleit in seiner Diplomarbeit [Jodeleit, 1996] implementiert.

Leider wurde bei Laufzeittests deutlich, daß die Probleme sehr feingranular sind und daher einen hohen Kommunikationsaufwand verursachen. Der Kommunikationsaufwand war so hoch, daß Netzwerkprobleme während der Tests auftraten. So blieb nur die Erstellung eines ProSet-Programms übrig, das es ermöglicht, die Problemgröße und somit die Granularität sinnvoll zu wählen.

Dieses Programm errechnet mit Hilfe des Master-Worker-Modells aus einem Tupel von Primzahlen alle möglichen Produkte von Potenzen dieser Zahlen in aufsteigender Reihenfolge bis zu einer gegebenen Obergrenze $n$. Bei allen Tests mit dem Hamming-Zahlen-Programm bestand das Eingabetupel aus den 25 Primzahlen, die kleiner 100 sind. $n$ wurde auf 1000000 festgelegt [1].

Allerdings stützt sich die Implementierung des Hamming-Zahlen-Problems nur auf singuläre Tupelräume ab. Für den Vergleich des singulären gegenüber dem multiplen Tupelraumansatz mußten zusätzliche ProSet-Programme geschrieben werden. Dabei fiel die Wahl auf einfache, aber bewährte „Ping-Pong“-Programme (Der Source-Code ist im Anhang im Kapitel B.1 auf Seite 113 einsehbar). Ähnliche „Ping-Pong“-Programme wurden zur Performance-Messung z.B. in [Bjornson, 1993], [Thomas, 1991] oder [Rowstron *et al.*, 1995] eingesetzt.

Die Programme dieser Arbeit enthalten einen Master und vier Worker (Threads). Die Worker kommunizieren **paarweise** miteinander. Der Master wartet nur auf die Beendigung der Worker. Im Programm `Ping_Pong` erfolgt die Kommunikation über einen gemeinsamen Tupelraum, im Programm `Ext_Ping_Pong` über jeweils einen Tupelraum (also zwei). Die Abbildungen 12.1 und 12.2 sollen dies verdeutlichen.

Nach dem `deposit` durchläuft jeder Thread eine Warteschleife, deren Iterationen durch eine globale Variable $d$ bestimmt wird. Außerdem kann die Anzahl der Kommunikationszyklen mit Hilfe der Variable $n$ variiert werden. So ist die Granularität des Programms frei einstellbar. Diese Technik wurde auch in [Bjornson, 1993] eingesetzt.

## 12.2 Die Ergebnisse

### 12.2.1 Vergleich zwischen zentralistischem und verteiltem System

Zuerst soll der Performance-Gewinn des neuen parallelen Laufzeitsystems gegenüber dem alten quasi-parallelen System untersucht werden. Dazu wurden beide Systeme mit

[1] Das Ergebnis ist in diesem Fall übrigens 72271.

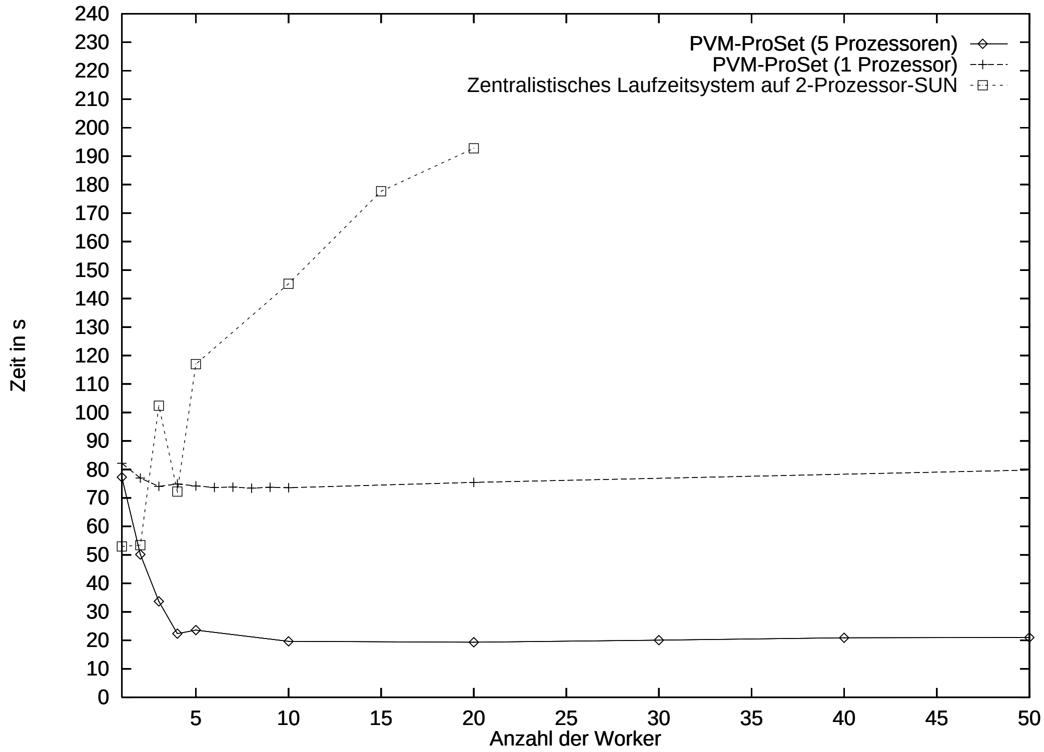


Abbildung 12.3: Vergleich zwischen dem zentralistischem Laufzeitsystem und PVM-ProSet auf einem und fünf Hosts mit Hilfe des Hamming-Zahlen-Programms.

Hilfe des oben beschriebenen Hamming-Zahlen-Programms getestet. Gemessen wurde die jeweilige Realzeit abhängig von der Anzahl der Worker. Das neue System wurde auch lokal auf einem Host getestet. Das Ergebnis ist in der Abbildung 12.3 zu sehen. Sehr deutlich ist die rapide Verschlechterung der Laufzeit des alten Systems bei steigender Worker-Anzahl zu erkennen. Schon die lokale Testreihe der neuen Version ist mit fünf Workern sehr viel besser als das alte System. Dies mag daran liegen, daß PVM bei Einrechnerbetrieb, genauso wie Threads, mit Hilfe von gemeinsamen Speicher kommunizieren, aber die Prozesse selber eigenständig sind. Gegenüber Threads bekommen sie mehr Rechenzeit vom Scheduler des Betriebssystems, während Threads sich die Rechenzeit eines Prozesses teilen müssen.

Auffällig ist weiterhin, daß die Zeiten des lokal getesteten neuen Systems mit steigender Workeranzahl nur leicht zunehmen. Anscheinend ist der Prozeßstart in PVM sehr effizient implementiert. Von den kurzen Prozeßstartzeiten werden auch die Ergebnisse der Tests der nachfolgenden Abschnitte profitieren.

Der eigentliche Sinn dieses Tests ist der Vergleich des alten mit dem neuen System. Die Ergebnisse des neuen Systems sind in der untersten Kurve zu sehen. Deutlich ist der Performance-Gewinn abzulesen. Ab 20 Workern steigt die Kurve wieder leicht an, aber nur im gleichen Maße wie beim lokalen Test. (In einem weiteren hier nicht abgebildeteten Test mit 100 Workern stieg die Laufzeit nur auf 23.77 Sekunden an.)

Beim Minimum der Kurve bei 20 Workern benötigt das parallele System nur noch 10% der Laufzeit des quasi-parallelen Systems und nur rund 25% der Laufzeit der lokal-getesteten neuen Version.

Dieser Test zeigt: Erst wirkliche Parallelität bedingt Performance-Gewinn.

### 12.2.2 Der Test des Verteilten Hashings mit schlechter Hashfunktion

In diesem Test ging es um die Frage: Inwieweit beeinflußt eine schlechte Hashfunktion die Laufzeit?

Hierzu wurde das Hamming-Zahlen-Programm mit dem neuen System, zusätzlich zum oben beschriebenen Test mit einer Partition, mit zwei und vier Partitionen getestet. Die Hashfunktion war hierbei die Standardhashfunktion, die Länge und Datentypen der Elemente berücksichtigt. Die Funktion ist in keinster Weise auf das Hamming-Zahlen-Programm hin optimiert.

In Abbildung 12.4 sind die Ergebnisse zu sehen. Die Kurven aller drei Tests zeigen fast identische Verläufe. Eine eindeutige Verschlechterung auf Grund der schlechten Hashfunktion ist nicht festzustellen.

Dies mag zum einen wieder am schnellen Prozeßstart von PVM liegen, der dadurch keine Erhöhung der Laufzeit bei mehreren Partitionen verursacht, zum anderen an dem verwendeten Kommunikationsprotokoll, das die Wartezeit auf nicht belegte Partitionen bei der Suche nicht erhöht.

Eine Erhöhung der Laufzeit bei schlechter Hashfunktion ist nicht festgestellt worden. Aber kommt es zu einer Laufzeitverbesserung bei optimierter Hashfunktion? Dieser Frage wird im nächsten Abschnitt nachgegangen.

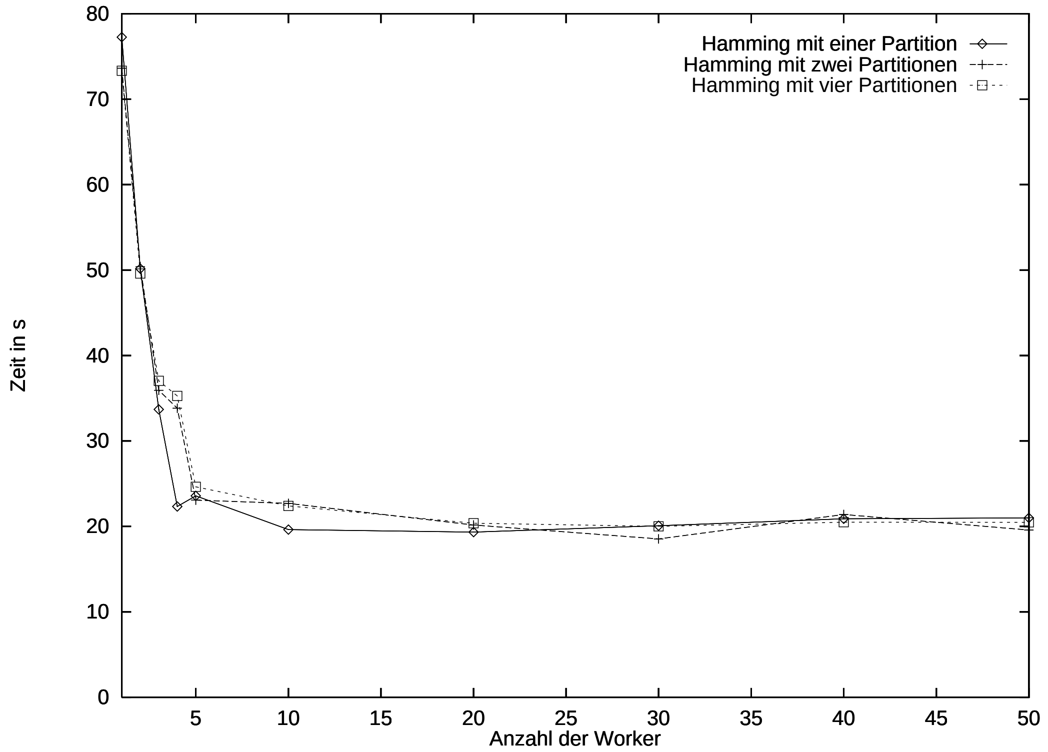


Abbildung 12.4: Das Hamming-Zahlen-Programm mit einer, zwei und vier Partitionen im Vergleich.

### 12.2.3 Vergleich zwischen dem Verteiltem Hashing und dem „Server"-Ansatz

Der Test zwischen den beiden implementierten Ansätzen zum verteilten Tupelraum-Management wurde mit Hilfe des beschriebenen Programms `Ping_Pong` durchgeführt. Der Tupelraum des Programms wurde im ersten Test mit einer Partition, d.h. als Zentraler Server, umgesetzt. Im zweiten Test wurde das Verteilte Hashing mit zwei Partitionen und optimierter Hashfunktion angewendet. Die Funktion hasht nach dem ersten Element in den Tupeln und Templates (siehe Source-Code im Anhang auf Seite 113).

In diesem Test wurde die Workeranzahl konstant gehalten. Dagegen wurde die Zahl der Kommunikationszyklen (Ping-Pongs) bei festem Arbeitsaufwand (1000 Warteschleifendurchläufe) und der Arbeitsaufwand pro Worker bei fester Anzahl der Kommunikationszyklen (400) variiert. In beiden Fällen variiert also die Granularität des Programms.

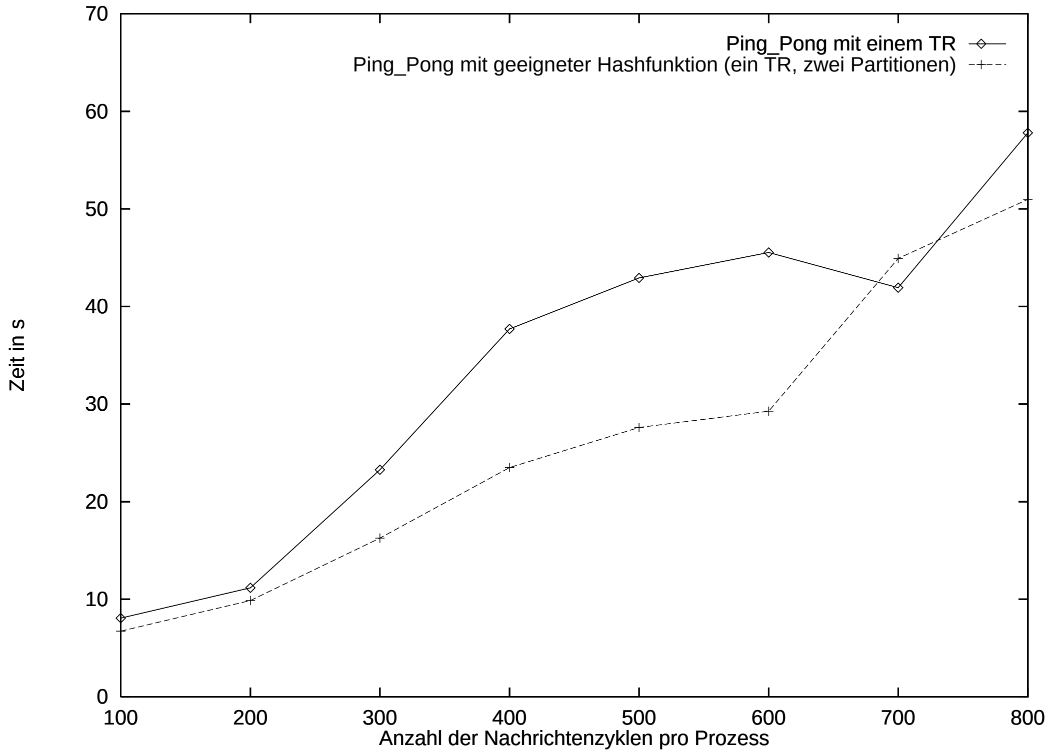


Abbildung 12.5: Vergleich zwischen dem Verteiltem Hashing und dem „Server"-Ansatz durch die Variation des Kommunikationsaufwandes.

Zuerst werden die Ergebnisse des Tests mit variablen Kommunikationszyklen betrachtet:
In Abbildung 12.5 sind die Ergebnisse dargestellt. Deutlich ist eine Laufzeitverbesserung durch das Verteilte Hashing zu erkennen. Weiterhin ist zu sehen, daß die Verbesserung der Laufzeit mit der Erhöhung des Kommunikationsaufwandes zuerst zunimmt. Die Erhöhung der Kommunikationszyklen bedeutet ja eine Steigerung der Belastung des Tupelraums bzw. der Partitionen, so greift also die Arbeitsteilung zwischen den Partitionen voll.

Die Anomalien über 600 Zyklen kann hier nicht erklärt werden. Vielleicht hängen sie mit den physischen Grenzen des Ethernets zusammen.

Nun zu den Ergebnissen des Tests mit variablem Arbeitsaufwand: Wie schon beschrieben, wurde die Simulation des Arbeitsaufwands pro Prozeß durch eine Warteschleife realisiert. Die Ergebnisse des Tests sind in Abbildung 12.6 zu sehen. Wichtig ist zur

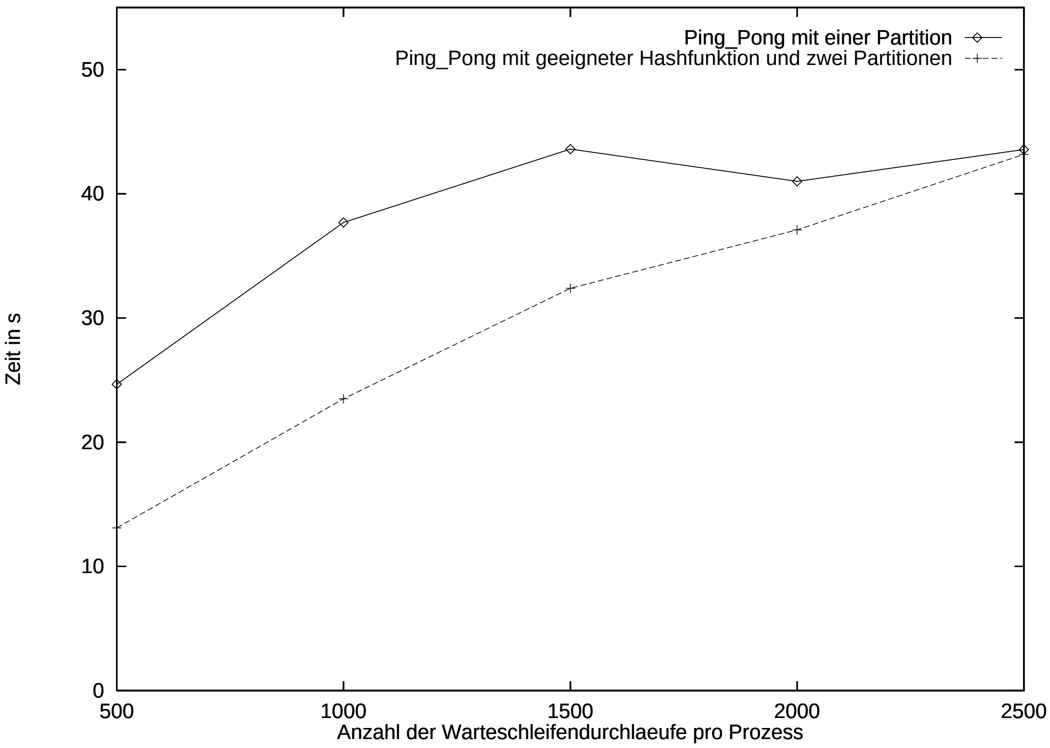


Abbildung 12.6: Vergleich zwischen dem Verteiltem Hashing und dem „Server“-Ansatz durch variierenden Arbeitaufwand pro Worker.

Interpretation der Kurve, daß die Feinheit der Granularität von links nach rechts abnimmt.

Auch hier ist ein Performance-Gewinn deutlich, der wie bei der Variation der Nachrichtenzyklen stark von der Granularität des Programms abhängt. Bei etwa 2500 Wartezyklen ist die Granularität so grob, daß der Zentrale Server die Arbeit genauso effizient lösen kann wie die beiden Partitionen, da weniger Nachrichten pro Zeiteinheit verarbeitet werden müssen.

### 12.2.4 Vergleich zwischen dem Ansatz der singulären und der multiplen Tupelräume

Zuletzt zur eigentlichen Aufgabenstellung dieser Arbeit: Dem Vergleich zwischen dem Ansatz der singulären Tupelräume und der multiplen Tupelräume.

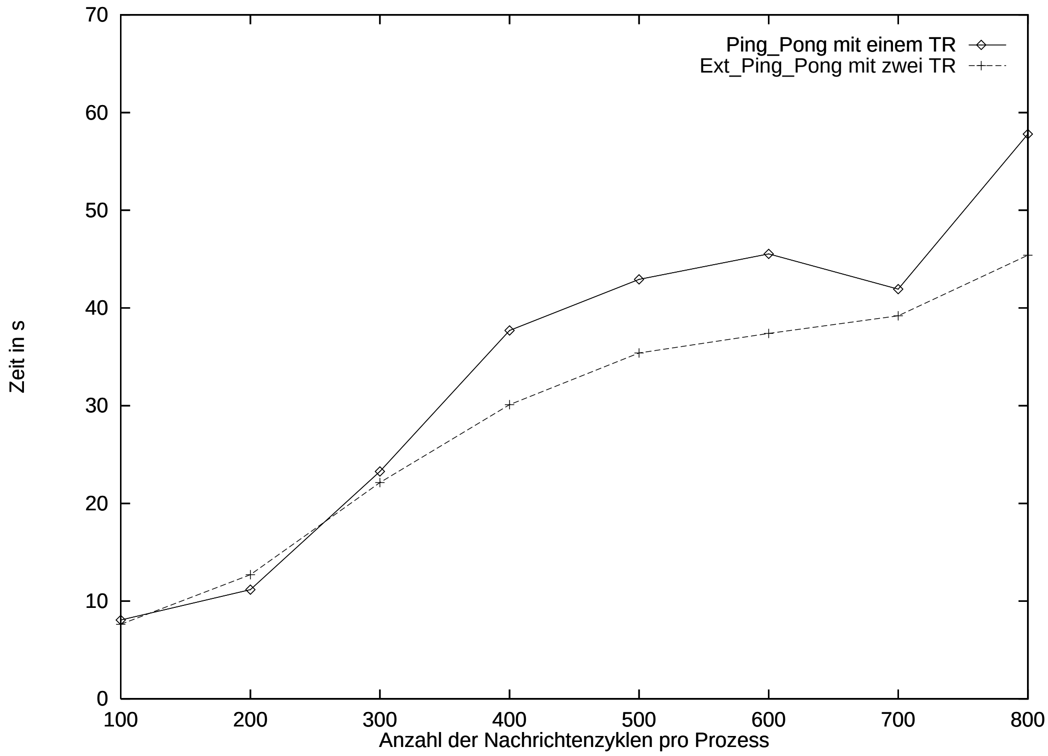


Abbildung 12.7: Vergleich zwischen den ProgrammenPing_Pong (ein Tupelraum) und Ext_Ping_Pong (zwei Tupelräume) durch die Variation der Nachrichtenzyklen.

Hier ist das Vorgehen das gleiche, wie bei den Tests im vorhergehenden Abschnitt. Auch hier wird zum einen die Zahl der Nachrichtenzyklen, zum anderen die Zahl der Warteschleifendurchläufe variiert. Allerdings wird jetzt das Programm `Ping_Pong` mit dem Programm `Ext_Ping_Pong` verglichen, welches den Ansatz der multiplen Tupelräume realisiert (Source-Code siehe im Anhang ab Seite 115).

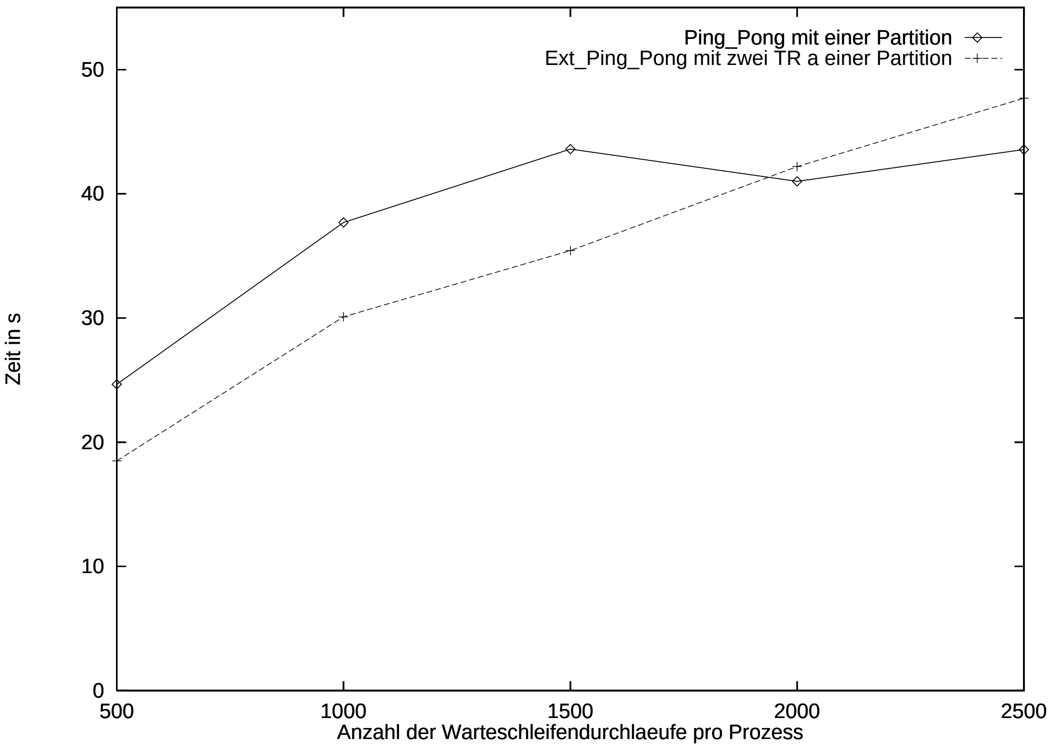


Abbildung 12.8: Vergleich zwischen den Programmen `Ping_Pong` (ein Tupelraum) und `Ext_Ping_Pong` (zwei Tupelräume) durch die Variation des Arbeitsaufwandes pro Worker.

Abbildung 12.7 zeigt die Ergebnisse bei der Variation des Kommunikationsaufwandes. Wieder ist ein wachsender Performance-Gewinn bei steigender Nachrichtenanzahl zu erkennen. Die Kommunikation über mehrere Tupelräume kann also bei hohem Kommunikationsaufwand eine Performance-Steigerung bewirken.

Die Ergebnisse der Variation des Arbeitsaufwandes finden sich in Abbildung 12.8. Auch hier sei darauf hingewiesen, daß die Granularität der Worker von links nach rechts abnimmt, der Arbeitsaufwand pro Zeiteinheit für die Tupelraum-Server ebenfalls sinkt.

Eindeutig ist ein Performance-Gewinn durch multiple Tupelräume gegenüber singulären zu verzeichnen. Allerdings läßt sich keine Steigerung durch eine Vergröberung der Granularität der Worker erkennen.

Bei ca. 2000 Schleifendurchläufen tritt wie beim Ansatz des Verteilten Hashings wahrscheinlich wieder das Problem der „Unterforderung“ des singulären Tupelraum-Servers auf. Ab diesem Wert können die beiden Tupelräume zusammen nicht mehr Arbeit vollbringen, als der einfache Tupelraum, weil nicht mehr Nachrichten ankommen. Dann macht sich der größere Aufwand für zwei TR bemerkbar, der durch die Parallelverarbeitung nicht mehr aufgefangen werden kann.

# 13. Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Untersuchung, inwieweit der Ansatz der multiplen Tupelräume in ProSet-Linda im parallelen Einsatz zu Laufzeitverbesserungen gegenüber dem Ansatz des singulären Tupelraumes führt.

Dazu wurde in dieser Arbeit ein neues paralleles Laufzeitsystem für die Prototyping-Sprache entwickelt, welches bis auf wenige Teilbereiche die gesamte Linda-Funktionalität zur Verfügung stellt. Spezifische ProSet-Eigenschaften wie Persistenz und Ausnahmebehandlung wurden bewußt ausgeklammert, da sie keinen Beitrag zur Fragestellung der Arbeit leisteten.

Zum verteilten Tupelraum-Management wurde der flexible Ansatz des Verteilten Hashings benutzt und ein eigenes Protokoll zur Verwirklichung der Linda-Semantik entwickelt und implementiert. Das entfernte Starten von nebenläufigen Kontrollflüssen (Threads) wurde durch den `case-main`-Ansatz verwirklicht, aber auch auf Tupelraumpartitionen und Eval-Server erweitert.

Eine Loadbalancierung wurde entwickelt und implementiert. Eine Aussage über die Nützlichkeit kann aber wegen fehlender Tests nicht getroffen werden. Allerdings ist die Performance ohne Loadbalancierung schon sehr zufriedenstellend und der Spielraum für eine Verbesserung der Laufzeit durch eine Balancierung sehr klein. Die nächste PVM-Version wird eine integrierte Loadbalancierung besitzen, die aller Voraussicht nach effizienter als jede Benutzer-programmierte sein wird.

Aus Sicht der Aufgabenstellung waren die Laufzeittests ein voller Erfolg. Die verwendeten Ansätze wurden bestätigt. Es konnte folgendes gezeigt werden:

- Eine parallele Verarbeitung kann die Laufzeit eines Master-Worker-Programms schon mit dem Zentralem-Server-Ansatz und singulärem Tupelraum unter ProSet-Linda um den Faktor 10 verbessern.

- Bei Programmen, die multiple Tupelräume sinnvoll einsetzen, kann dieser Ansatz zu Laufzeitverbesserungen führen. Bei schlechter Auslastung der Tupelräume tritt eine Laufzeitverschlechterung ein. Je höher die Auslastung, desto größer ist der Zeitgewinn.

- Durch den Einsatz von geeigneten Hashfunktionen kommt es ebenfalls zu Laufzeitverbesserungen. Schlechte Hashfunktionen beeinträchtigen die Laufzeit im Vergleich zum Zentralem-Server-Ansatz nicht. Die Höhe der Laufzeitverbesserung hängt auch hier vom Kommunikationsaufwand ab.

Die Verwendung von mehrfachen Tupelräumen sollte sinnvoll geschehen, d.h. es sollte der notwendige, zusätzliche Aufwand gegenüber dem Nutzen abgewogen werden. Mehrere Tupelräume können einen Performance-Gewinn nur dann bewirken, wenn sie genügend stark ausgelastet sind. Wenn ein Tupelraum aber während der Laufzeit unbenutzt bleibt und z.B. nur zur Übermittlung eines Endergebnisses dient, sollte auf ihn verzichtet werden.

Auch wenn die Performance im Algorithmen-Prototyping nicht im Vordergrund steht, ist ein echt-paralleles Laufzeitsystem doch von Vorteil. Gerade bei dem Prototyping von parallelen Algorithmen kann auf ein verteiltes Laufzeitsystem nur schwer verzichtet werden, da nur dann Laufzeitvergleiche zwischen den einzelnen Kontrollflüssen durchgeführt werden können und außerdem viele Fehler im Algorithmus in quasi-parallelen Umgebungen nicht auftreten und somit nicht entdeckt werden können.

Deshalb sollte das erstellte Laufzeitsystem durch die Implementierung der hier vernachlässigten ProSet-Linda-Funktionalitäten (Persistenz und Ausnahmebehandlung) komplettiert werden. Der Autor dieser Arbeit würde sich darüber freuen.

# Literaturverzeichnis

[Arango und Berndt, 1989] M. Arango und D. Berndt. *TSnet: A Linda implementation for networks of Unix-based computers.* Technischer Bericht 739, Universität Yale, New Haven, CT, August 1989.

[Bjornson, 1993] Robert Bjornson. *Linda on Distributed Memory Multiprocessors.* Dissertation, Department of Computer Science, Universität Yale, Mai 1993.

[Budkowski und Dembinski, 1987] S. Budkowski und P. Dembinski. *An Introduction to Estelle: A Specification Language for Distributed Systems. Computer Networks and ISDN Systems*, 14:3–23, 1987.

[Budkowski, 1992] S. Budkowski. *Estelle development toolset (EDT). Computer Networks and ISDN Systems*, 25:63–82, 1992.

[Carriero und Gelernter, 1986] Nicholas John Carriero und David Gelernter. *The S/Net's Linda Kernel. ACM Transaction on Computer Systems*, 4(2):110–129, Mai 1986.

[Crowl, 1994] Lawrence A. Crowl. *How to Measure, Present, and Compare Parallel Performance. IEEE Parallel & Distributed Technology*, Seiten 9–25, Frühjahr 1994.

[Doberkat *et al.*, 1990a] E.-E. Doberkat, U. Gutenbeil und W. Hasselbring. *SETL/E Sprachbeschreibung Version 0.1.* Informatik-Bericht 01-90, Universität Essen, März 1990.

[Doberkat *et al.*, 1990b] E.-E. Doberkat, U. Gutenbeil und W.Hasselbring. *SETL/E – A Prototyping System based on Sets.* In W. Zorn, Editor, *Proc. TOOL'90*, Seiten 109–118. Universtät Karlsruhe, November 1990.

[Doberkat *et al.*, 1992] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers und C. Pahl. *ProSet – Prototyping with Sets: Language Definition.* Technischer Bericht 02-92, Universität Essen, April 1992.

[Douglas *et al.*, 1994] Andrew Douglas, Alan Wood und Antony Rowstron. *Linda Implementation Revisited.* Technischer Bericht, Universität York, Heslington, York YO1 5DD, UK, November 1994.

[Geist *et al.*, 1994] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek und V. Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[Gelernter, 1985] D. Gelernter. *Generative Communication in Linda. ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Hasselbring, 1991] Wilhelm Hasselbring. *Combining SETL/E with Linda.* In Wilson [1991], Seiten 84–97.

[Hasselbring, 1994] Wilhelm Hasselbring. *Prototyping Parallel Algorithms in a Set-Oriented Language.* Dissertation, Universität Dortmund, 1994.

[Hogrefe, 1989] D. Hogrefe. *Estelle, LOTOS und SDL — Standard-Spezifikationssprachen für verteilte Systeme.* Springer Verlag, 1989.

[International Standard ISO/IS 9074, 1987] *ISO: ESTELLE: A formal description technique based on an extended state transitionmodel, International Standard ISO/IS 9074.* International Organization for Standardization, 1987.

[Jodeleit, 1996] P. Jodeleit. *Implementation der Salishan und Cowichan Probleme durch Prototyp-Evaluation und Transformation.* Diplomarbeit, Universität Dortmund, Mai 1996.

[Leichter, 1989] J.S. Leichter. *Shared Tuple Memories, Buses and LAN's – Linda Implementations across the Spectrum of Connectivity.* Dissertation, Yale Universität, New Haven, CT, Juli 1989.

[Metcalfe und Boogs, 1976] Robert M. Metcalfe und David R. Boogs. *Ethernet: Distributed Packet Switching for Local Computer Networks. Communications of the ACM*, 19(7):395–404, Juli 1976.

[Robinson und Arthur, 1995] Patrick G. Robinson und James D. Arthur. *Distributed Process Creation Within a Shared Data Space Framework. Software–Practice and Experience*, 25(2):175–191, Februar 1995.

[Rowstron *et al.*, 1995] Antony Rowstron, Andrew Douglas und Alan Wood. *A Distributed Linda-like Kernel for PVM.* Technischer Bericht, Universität York, Heslington, York, YO1 5DD, UK, 1995.

[Schumann und Lobinger, 1994] Matthias Schumann und Andreas Lobinger. *Zeitspiel – Messung von Programmen am Beispiel von SunOS und Solaris. iX*, 11:168–173, November 1994.

[Sijelmassi und Linn, 1992] Rachid Sijelmassi und Richard J. Linn. *Guidelines for using Estelle to specify OSI services and protocols. Computer Networks and ISDN Systems*, 23:343–362, 1992.

[Sijelmassi und Strausser, 1993] Rachid Sijelmassi und Brett Strausser. *The PET and DINGO tools for deriving distributed implementations from Estelle. Computer Networks and ISDN Systems*, 25:841–851, 1993.

[Stevens, 1990] W. R. Stevens. *UNIX network programming.* Prentice Hall software series. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Tanenbaum, 1992] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[Thomas, 1991] Owen Thomas. *A Linda Kernel for Unix Networks.* In Wilson [1991], Kapitel 10, Seiten 124–128.

[Wilson, 1991] Greg V. Wilson, Editor. *Proc. Workshop on Linda-Like Systems and Their Implementation.* Edinburgh Parallel Computing Centre TR91-13, Juni 1991.

# Teil VI

# Anhänge

# A. Einführung in das Ethernet

Ethernet-Systeme sind, obwohl sich auch leistungsstärkere Systeme auf dem Markt befinden, die wohl am weitesten verbreitete Hardware für lokale Netze. Dies mag einerseits daran liegen, daß dieser Ansatz schon in den 70er Jahren entwickelt wurde, andererseits aber auch an seiner Einfachheit und den dadurch bedingten geringen Kosten.

Es soll hier nicht in allen Details auf das Ethernet eingegangen werden (Der Interessierte kann z.B. auf [Metcalfe und Boogs, 1976] zurückgreifen), doch sollen im folgenden wenigstens die rudimentären Ansätze, die hinter diesem lokalen Kommunikationsmittel stehen.

## A.1 Die Hardware

Ethernet ist ein Kommunikationssystem für lokale Netze. Die physikalische Leitung besteht aus einem einfachen gedrillten oder Koaxialkabel mit einer maximalen Länge von zehn Kilometern. An den Enden wird das Kabel durch sogenannte Terminatoren abgeschlossen. Diese verhindern ein Reflektieren der elektrischen Signale am Kabelende[1].

Die Stationen sind bit-seriell mit einem Transceiver und dieser wiederum mit dem Koaxialkabel verbunden. Zusätzlich zu den Stationen können sogenannte *Repeater* (Wiederholer) auf die gleiche Weise am Kabel angeschlossen werden. Diese sind aber nicht nur mit einem, sondern mit zwei Kabeln verbunden. Sie wiederholen alle Pakete, die sie auf der einen Leitung empfangen auf der anderen und umgekehrt. So können Ethernetsysteme gekoppelt werden. Es entstehen beliebige baumartige Topologien ohne ausgezeichneten Wurzelknoten.

[1](Ehemaligen) Informatikstudenten der Universität Dortmund, die das elektrotechnische Praktikum absolviert haben, ist vielleicht diese Reflexion noch in Erinnerung, mit deren Hilfe in einem Versuch die Länge des Koaxialkabels gemessen werden mußte.

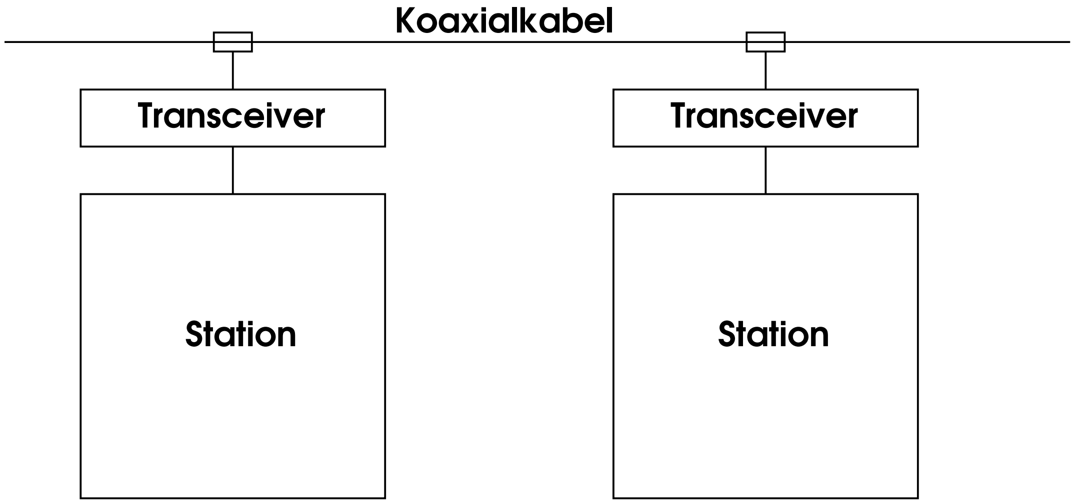


Abbildung A.1: Der schematische Aufbau des Ethernets

## A.2 Das Kommunikationsprotokoll

Ethernet-Systeme arbeiten mit einem sehr einfachen Kommunikationsprotokoll. Alle Stationen (auch ein Repeater sei im folgenden eine Station) lauschen immer am Koaxialkabel. Jedes Paket ist mit der Adresse des Empfängers gekennzeichnet. Sollen alle Stationen das Paket empfangen, so ist die Adresse nach Konvention die Zahl 0, da dies im Ethernet keine gültige Adresse ist. Registriert eine Station ein Paket mit seiner Adresse, so speichert es dieses ab. Alle anderen Pakete werden ignoriert. Will eine Station nun senden, so wartet sie bis keine andere Station sendet, d.h. sie nichts mehr empfängt. Dann fängt sie an, zu senden. Gleichzeitig lauscht die Station aber am Netz und gleicht das Gesendete mit dem Empfangenen ab. Sind diese Bitströme nicht gleich, so muß eine Kollision vorliegen. Das bedeutet, daß zwei Stationen zur gleichen Zeit angefangen haben zu senden. Bemerkt eine Station dies, so gibt sie das Senden auf, wartet eine zufällig gewählte Zeitspanne und beginnt von vorn, wenn das Kabel frei erscheint.

# B. Der Quelltext der Testprogramme

## B.1 Der Quelltext von Ping_Pong

```
program ping_pong;
visible n := 200;
visible d:=1000;
visible Space:= CreateTS(om);
begin

|| closure play1();
|| closure play2();
|| closure replay1();
|| closure replay2();
for i in [1..2] do
   fetch (i,"the end")
      at Space
   end fetch;
end for;

procedure play1();
begin

for i in [1..n] do
   deposit [1, "ping"]
      at Space
   end deposit;
   for z in [1..d] do
      pass;
```

```
      end for;
      fetch (1, "pong")
         at Space
      end fetch;
   end for;
   deposit [1,"the end"]
      at Space
   end deposit;

   end play1;

   procedure replay1();
   begin

   for i in [1..n] do
      fetch (1, "ping")
         at Space
      end fetch;
      for z in [1..d] do
         pass;
      end for;
      deposit [1, "pong"]
         at Space
      end deposit;
   end for;

   end replay1;

   procedure play2();
   begin

   for i in [1..n] do
      deposit [2, "ping"]
         at Space
      end deposit;
      for z in [1..d] do
         pass;
      end for;
```

```
   fetch (2, "pong")
      at Space
   end fetch;
end for;
deposit [2, "the end"]
   at Space
end deposit;

end play2;

procedure replay2();
begin

for i in [1..n] do
   fetch (2, "ping")
      at Space
   end fetch;
   for z in [1..d] do
      pass;
   end for;
   deposit [2, "pong"]
      at Space
   end deposit;
end for;

end replay2;

end ping_pong;
```

## B.2 Der Quelltext von Ext_Ping_Pong

```
program ext_ping_pong;
visible n:=200;
visible d:=1000;
visible Space1:= CreateTS(om);
```

```
visible Space2:= CreateTS(om);
begin

|| closure play1();
|| closure play2();
|| closure replay1();
|| closure replay2();
fetch ("end")
   at Space1
end fetch;
fetch ("end")
   at Space2
end fetch;

procedure play1();
begin

for i in [1..n] do
   deposit ["ping"]
      at Space1
   end deposit;
   for z in [1..d] do
      pass;
   end for;
   fetch ("pong")
      at Space1
   end fetch;
end for;
deposit ["end"]
   at Space1
end deposit;

end play1;

procedure replay1();
begin

for i in [1..n] do
```

```
      fetch ("ping")
         at Space1
      end fetch;
      for z in [1..d] do
         pass;
      end for;
      deposit ["pong"]
         at Space1
      end deposit;
   end for;

   end replay1;

   procedure play2();
   begin

   for i in [1..n] do
      deposit ["ping"]
         at Space2
      end deposit;
      for z in [1..d] do
         pass;
      end for;
      fetch ("pong")
         at Space2
      end fetch;
   end for;
   deposit ["end"]
      at Space2
   end deposit;

   end play2;

   procedure replay2();
   begin

   for i in [1..n] do
      fetch ("ping")
```

```
        at Space2
    end fetch;
    for z in [1..d] do
        pass;
    end for;
    deposit ["pong"]
        at Space2
    end deposit;
end for;

end replay2;

end ext_ping_pong;
```

# C. Der Quelltext des Laufzeitsystems

# C.1 Die Linda–Typen

## C.1.1 Einleitung

Diese Programmcode ist Teil der Diplomarbeit von Knuth Waltenberg mit dem Titel „Entwurf und Implementation eines verteilten Laufzeitsystems für ProSet-Linda“. Er stellt die Linda-Implementierung des ProSet-Linda-Laufzeitsystems für lokale Netze dar.

In dieser Implementierung sind die ProSet-Threads als eigenständige Prozesse verwirklicht. Die Kommunikation zwischen Prozessen wurde durch das Message-Passing-System PVM realisiert.

## C.1.2 Abhängigkeiten

⟨*Importe von proc.h*⟩≡

⟨*Importe von linda.h*⟩≡

```
#include <pvm3.h>
```

⟨*Importe von intLinda.h*⟩≡

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdarg.h>
#include <assert.h>
#include <string.h>
```

### C.1.3 Exportierte Typen

⟨*Typ des Templates*⟩≡

```
typedef struct template
{
  struct template *next;                /* the next template */
  unsigned int arity;         /* the number of components */
  Pbp_Object *condition;          /* the template condition */
  Pbp_Env dL;        /* belongs to the template condition */
  Pbp_Exit *RetStat;    /* belongs to the template condition */
  Pbp_Env *RetAd;      /* belongs to the template condition */
  Ple_Temp_Comp *comps;           /* the list of components */
} Ple_Template;
```

⟨*Typ der Template-Komponente*⟩≡

```
typedef struct tempcomp
{
  struct tempcomp *next;              /* the next component */
  Ple_Temp_Comp_Type type;      /* flags wether the component
                       is an actual or a formal */
  union      /* the component is an actual or a formal: */
  {
    Pbp_Object *actual;
    struct
    {
      Pbp_Object **lvalue;                    /* the l-value */
      Pbp_Object *into;                 /* into expression */
      Pbp_Env dL;         /* belongs to the into expression */
      Pbp_Exit *RetStat;  /* belongs to the into expression */
      Pbp_Env *RetAd;      /* belongs to the into expression */
    } formal;
  } comp;
} Ple_Temp_Comp;
```

⟨*Exportierte Typen*⟩≡

```
typedef enum { PLE_ACTUAL, PLE_FORMAL } Ple_Temp_Comp_Type;
```

⟨*Exportierte Typen*⟩+≡

```
typedef enum { PLE_ELSE, PLE_WAIT } Ple_Else_Type;
```

⟨*Exportierte Typen*⟩+≡
```
typedef int Ple_Process;                  /* typ of a process */
```

⟨*Exportierte Typen*⟩+≡
```
typedef struct hostlist
{
  struct hostlist *next;
  char *hostname;
  double load;
} Ple_Hostname_List;
```

### C.1.4 Interne Typen

Die Menge aller vorhandenen Tupleräume wird als verkettete Liste dargestellt. Ein einzelner Tupelraum besteht aus einem Zeiger auf den nächsten Tupelraum, einer eindeutigen Identifizierung, der maximalen und der tatsächlichen Anzahl von Tupeln, der Multimenge der enthaltenen Tupel und der Liste der wartenden Prozesse. Ist der Wert von `max` negativ, so ist vom Benutzer keine Beschränkung vorgegeben worden.

⟨*Typ des Tupelraumes*⟩≡
```
typedef struct tuple_space
{
  struct tuple_space *next;           /* the next tuple space */
  Pbp_Object *atom;               /* the tuple space identity */
  char *atom_str;             /* the atom converted to string */
  int tids[20];                     /* array of partition ids */
  int hash_mode;               /* Index of the hash mode */
  int max_partitions;    /* number of partition of this ts */
  int max;                     /* the highest number of tuples*/
  int card;                           /* the number of tuples */
  Pli_Hash_List *hashtable[PLI_PRIME]; /* contains all tuples */
  Pli_Pend_Proc *pendprocs;  /* contains the list of the pending
                              processes */
} Pli_Tuple_Space;
```

Die Multimenge ist als Hashtabelle verketteter Listen von Objekten realisiert. Es wurde sich weitestgehend an die interne Typdefinition der Mengen gehalten. Da es sich hier jedoch um eine Multimenge handelt, ist es durchaus möglich, daß ein Tupel mehrfach vorhanden ist. Aus diesem Grund konnten nicht alle Konzepte, Funktionen und Makros der Mengen unverändert übernommen werden.

⟨*Typ der Hashtabelle*⟩≡

```
typedef struct liste
{
  Pbp_Object *tuple;           /* element of the tuple space */
  struct liste *next;      /* next element of the hashbucket */
} Pli_Hash_List;
```

⟨*Typ der Hashliste*⟩≡

```
typedef struct processlist
{
  struct processlist *next;
  int parent_tid;
  int number;                /* Nummer des Eintrags (positiv,
                   wenn Tupel, 0, wenn Ergebnisobjekt
                      negativ, wenn Vater wartet)*/
  Pbp_Object *obj;       /* aktives Tupel oder Ergebnisobjekt */
   int partition_tid;
} Pli_ProcHash_List;
```

⟨*Typ des wartenden Prozesses*⟩≡

```
typedef struct pendlist
{
 struct pendlist *next;             /* the next pending entry */
 Ple_Template *temps;        /* the list of templates of one
                                  statement */
 Pli_Op_Type optype;          /* either PLI_FETCH or PLI_MEET */
 int *ptempnum;    /* place to store the selected template */
               /* number when a matching tuple has been */
                                            /* deposited */
 int proc;                           /* the pending process */
} Pli_Pend_Proc;
```

⟨*Typ der gestarteten Prozesse*⟩≡
```
typedef struct spawnedlist
{
 struct spawnedlist *next;                  /* the next entry */
 int tid;                         /* the process tid */
} Pli_Spawned_Proc;
```

⟨*Typ der Art der Linda-Operation*⟩≡
```
typedef enum { PLI_NOOP, PLI_FETCH, PLI_MEET, PLI_MEETINTO } Pli_Op_Type;
```

⟨*Typ der Art der plo_find_ts-Option*⟩≡
```
typedef enum { PLI_REMOVE, PLI_FIND, PLI_CLEAR} Pli_FindTs_Type;
```

⟨*Typ der Parameter-Liste*⟩≡
```
typedef struct
{
    Pbp_Object *** pars;  /* Liste der Argumente eines Threads */
} Pli_ParList;
```

### C.1.5 Interne Konstanten

⟨*Anzahl der Partitionen*⟩≡
```
#define PLI_MAX_PARTITIONS 10   /* Maximal m"ogliche Anzahl
                           von Partitionen */
```

Die Anzahl der Hashbuckets in einem Tupelraum wird durch die Konstante `PLI_PRIME` bestimmt.

⟨*Anzahl der Hashbuckets*⟩≡
```
#define PLI_PRIME 5                /* size of the hashtable */
```

### C.1.6 C Dateien

⟨*linda.h*⟩≡

```
#ifndef PLEXT_H
#define PLEXT_H
⟨Name⟩
⟨Importe von linda.h⟩
⟨Exportierte Typen⟩
⟨Typ der Art der Linda-Operation⟩
⟨Typ der Template-Komponente⟩
⟨Typ des Templates⟩
⟨Typ der Hashtabelle⟩
⟨Typ der Hashliste⟩
#endif
```

⟨*intLinda.h*⟩≡

```
#ifndef PLINT_H
#define PLINT_H
⟨Typ der Art der plo_find_ts-Option⟩
⟨Importe von intLinda.h⟩
⟨Anzahl der Partitionen⟩
⟨Anzahl der Hashbuckets⟩
⟨Typ des wartenden Prozesses⟩
⟨Typ der gestarteten Prozesse⟩
⟨Typ des Tupelraumes⟩
⟨Typ der Parameter-Liste⟩
#endif
```

## C.2 Die Linda–Operationen

### C.2.1 Einleitung

Hier befinden sich die Routinen, die für die Codeerzeugung der drei Lindaoperationen `deposit`, `fetch` und `meet` erforderlich sind. Des weiteren sind hier auch (vorläufig) die Laufzeitbibliotheksfunktionen für Linda zu finden.

### C.2.2 Abhängigkeiten

⟨*Importe der Quelltextdatei*⟩≡

```
#include "proset/proset.h"
#include "proset/unit.h"
#include "proset/overloaded.h"
#include "proset/function.h"
#include "proset/handler.h"
#include "proset/module.h"
#include "proset/loop.h"
#include "proset/atom.h"
#include "proset/string.h"
#include "proset/set.h"
#include "proset/tuple.h"
#include "proset/io.h"

#include "linda/linda.h"
#include "../intLinda.h"
```

### C.2.3 Exportierte Operationen

#### plo_deposit

⟨*Exportierte O-Operationen*⟩≡

```
extern Pbp_Object *plo_deposit(Pbp_Object *tup, Pbp_Object *tsid,
                     pbp_form_par);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist im Ausnahmefall der Wert des Handlers. |
| `tup` | ist das Tupel, das in den Tupelraum eingefügt werden soll. |
| `tsid` | ist die Tupelraumidentität. |
| `pbp_form_par` | ist das Makro für die Ausnahmeparameter. |

**Beschreibung:** `plo_deposit` fügt `tup` in eine gehashte Partition eines Tupelraumes mit der Identität `tsid` ein. Im Ausnahmefall wird der Wert des Handlers zurückgegeben. Sonst om.

**Ausnahmen:** `type_mismatch`, `ts_invalid_id`

**Abhängigkeiten:** `pbp_type`, `plo_find_ts`, `plo_add_tuple`, `pvm_initsend`, `pvm_pkint`, `pvm_pkstr`, `pvm_send`, `pli_process_in_tup`, `plp_pack_obj`, `pli_hash_tup_general`, `pvm_perror`

⟨*Implementierung der Operationen*⟩≡

```
Pbp_Object *plo_deposit(Pbp_Object *tup, Pbp_Object *tsid,
                 pbp_form_par)
{
 extern int plp_encoding;
 Pbp_Object *retObj = pbp_omega;/* the return value of an
                             exception handler */
 Pli_Tuple_Space *ts;          /* the tuplespace with the
                                identity tsid */
 pbh_decl_var;                 /* the exception handling
                                 variables */
 char *buffer;                 /* the string buffer */
 char *string;                /* tup converted to string */
 int tids[PLI_MAX_PARTITIONS];  /* array of partition ids */
 int info;                     /* error variable */
 char *tmpstring;
 int partition_id;
 Pbp_Bool_Type active = PBP_FALSE; /* flag for aktive or passive
                                  tuples */
 int size;

  assert (tup != NULL);         /* compiler error? */
  assert (tsid != NULL);        /* compiler error? */
```

```
  #ifdef NETDEBUG
      pvm_perror("plo_deposit: Here I am!");
  #endif

    if(pbp_type(tup) != PBP_TUPLE) /* type_mismatch */
    {
    pvm_perror("plo_deposit: type mismatch in tuple");
    pbh_type_mismatch(retObj, "in plo_deposit");
    return(retObj);
    }

    if(pbp_type(tsid) != PBP_ATOM) /* type_mismatch in atom*/
    {
    #ifdef NETDEBUG
      tmpstring = pli_ots(tsid, NULL);
      pvm_perror(tmpstring);
    #endif

    pvm_perror("plo_deposit: type mismatch in atom");
    pbh_type_mismatch(retObj, "in plo_deposit");
    return(retObj);
    }

#ifdef TRACE
/* trace informations */
    if(gps_linda_trace)
    {
      fprintf(gps_trace_file, "o d b %s %i %s %d %i %s\n",
            pli_ots(tup, gps_return_address), tup->pbp_comp_number,
            pli_ots(tsid, gps_return_address),
            plp_thread_id(), pbh_get_line(), pbh_get_file());
    }
#endif

  /* finding the tuple space with the identity tsid */
    ts = plo_find_ts(tsid, PLI_FIND);
    if (ts == NULL)          /* ts_invalid_id */
```

```
  {
    pvm_perror("plo_deposit: Can't find tuplesspace!");
    pbh_ts_invalid_id(retObj, "in plo_deposit");
    return(retObj);
 }

 /* send deposit message to partition */
 info = pvm_initsend(plp_encoding);
#ifdef NETDEBUG
 if (info < 0)
  plp_error("plo_deposit (pvm_initsend)",
                 "Can't allocate memory!");
#endif

 info = pvm_pkint(&plp_main_id, 1, 1);
#ifdef NETDEBUG
 if (info < 0)
  plp_error("plo_deposit (pvm_pkint)",
                  "Can't allocate memory!");
#endif

 info = pvm_pkstr("d");
#ifdef NETDEBUG
 if (info < 0)
  plp_error("plo_deposit (pvm_pkstr)",
                  "Can't allocate memory!");
#endif

/* Is tup active? */
active = pli_process_in_tup(tup);

/* If the eval-server running? */
if (active == PBP_TRUE && plp_eval != PBP_TRUE)
  plp_error("plo_deposit",
          "Deposit of active tuple, but no eval server!");

 plp_pack_obj(tup); /* pack the tuple */
```

```
  /* hashing */
   partition_id = pli_hash_tup_general(tup, ts);

  if (active == PBP_FALSE)
  {
    /* send to right partition */
    info = pvm_send(partition_id, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_deposit (pvm_send)",
                   "Pvmd is not responding!");
    #endif

  }else{
    /* pack partition_id */
    info = pvm_pkint(&partition_id, 1, 1);
    #ifdef NETDEBUG
     if (info < 0)
      plp_error("plo_deposit (pvm_pkint)",
                   "Can't allocate memory!");
    #endif

    /* send to eval-server */
    info = pvm_send(plp_eval_tid, 5);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_deposit (pvm_send)", "Error in pvm_send!");
    #endif
  }
  #ifdef NETDEBUG
    pvm_perror("Leave plo_deposit");
  #endif

    return(retObj);
}
```

### plo_fetch

⟨*Exportierte O-Operationen*⟩+≡

```
extern Pbp_Object *plo_fetch(Ple_Template *temps,
                         Pbp_Object *tsid,
                         Pbp_Object *tempnum,
                    Ple_Else_Type elsetype, pbp_form_par);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist im Ausnahmefall der Wert des Handlers. |
| `tempnum` | ist die Nummer des ausgewählten Templates. (Rückgabeparameter) |
| `temps` | ist die Liste der Templates. |
| `tsid` | ist die Identität des Tupelraumes, in dem sich ein passendes Tupel befinden soll. |
| `elsetype` | gibt an, ob gewartet werden muß, bis ein passendes Tupel vorhanden ist. |
| `pbp_form_par` | ist das Makro für die Ausnahmeparameter. |

**Beschreibung:** `plo_fetch` versucht, für eins der gegebenen Templates in `temps` ein passendes Tupel in einer gehashten Partition eines Tupelraumes mit der Identität `tsid` zu entfernen. Den Formals des Templates werden die entsprechenden Werte zugewiesen. Ist dort kein passendes Tupel vorhanden, und `elsetyp` ist gleich `PLE_WAIT`, so wird gewartet, bis ein passendes Tupel in diesen Tupelraum eingefügt wird. Der Rückgabeparameter `tempnum` entspricht der Nummer des ausgewählten Templates. Falls kein passendes Tupel vorhanden ist, und auch nicht gewartet wird, ist der Wert 0. Im Ausnahmefall wird der Wert des Handlers zurückgegeben. Sonst om.

**Ausnahmen:** `type_mismatch`, `ts_invalid_id`

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:** `plp_encoding`, `pbp_type`, `plo_find_ts`, `pvm_perror`, `pvm_initsend`, `pvm_pkint`, `pvm_pkstr`, `pvm_send`, `pvm_recv`, `pvm_freebuf`, `pvm_bufinfo`, `pvm_setsbuf`, `pvm_mkbuf`, `pli_pack_tl`, `pvm_probe`, `plt_kill_temp`, `pvm_upkint`, `pvm_upkstr`, `pbp_set_use`, `pbo_assign`

⟨*Implementierung der Operationen*⟩+≡

```
Pbp_Object *plo_fetch(Ple_Template *temps, Pbp_Object *tsid,
     Pbp_Object *tempnum, Ple_Else_Type elsetype, pbp_form_par)
{
  extern int plp_encoding;
  Pli_Op_Type optype = PLI_FETCH; /* parameter to plo_match_in_ts */
  Pbp_Object *newtup = pbp_omega; /* dummy parameter to
                                     plp-match-in-ts */
  Pbp_Object *retObj = pbp_omega;   /* the return value of an
```

```
                             exception handler */
Pbp_Object *tempnum2 = pbp_omega; /* for internal fetch-call */
Pbp_Object *rvalue;            /* result value for condition
                                    function */
Ple_Temp_Comp *comp;        /* help pointer for the components */
Pli_Tuple_Space *ts;  /* the tuplespace with the identity tsid */
pbh_decl_var;               /* parameter for handler call */
Ple_Process myid;             /* process id of me */
int info;                  /* error variable for pvm functions */
int tempnumber;        /* number of template which fetched */
int i;                 /* index */
char *objstr;              /* string of converted object */
char *tmp;                 /* for free(objstr) after pli_scan */
int length;                     /* length of message string */
int buffer,buffer2,buffer3,rbuffer; /* four send and one reseive
                             buffer */
int bytes, type;           /* only for pvm_bufinfo call */
int who;                        /* who send me this message? */
char operation;                 /* operation id */
char *string;                   /* string of converted object
                              (for debug only)*/
int partition_ids[PLI_MAX_PARTITIONS]; /* hashed partitions */

 assert (temps != NULL);         /* compiler error? */
   assert (tsid != NULL);          /* compiler error? */

 #ifdef NETDEBUG
   pvm_perror("plo_fetch: BEGIN");
   string = pli_ots(tsid, NULL);
   pvm_perror(string);
 #endif

   if(pbp_type(tsid) != PBP_ATOM)  /* type_mismatch */
   {
   pvm_perror("plo_fetch : type_mismatch (atom)");
   pbh_type_mismatch(retObj, "in plo_fetch");
   return(retObj);
   }
```

```
#ifdef TRACE
/* trace informations */
    if(gps_linda_trace)
    {
    fprintf(gps_trace_file, "o f b %s %d %i %s\n",
            pli_ots(tsid, gps_return_address),
            plp_thread_id(), pbh_get_line(), pbh_get_file());
    }
#endif
/* finding the tuple space with the identity tsid */
    ts = plo_find_ts(tsid, PLI_FIND);
  #ifdef NETDEBUG
    pvm_perror("nach find_ts");
  #endif

    if (ts == NULL)          /* ts_invalid_id */
    {

#ifdef TRACE
/* trace informations */

    if(gps_linda_trace)
      {
        fprintf(gps_trace_file, "o f f %s %d %i %s\n",
              pli_ots(tsid, gps_return_address),
              plp_thread_id(), pbh_get_line(), pbh_get_file());
      }
#endif

      pvm_perror("plo_fetch: Can't find tuplespace!");
      pbh_ts_invalid_id(retObj, "in plo_fetch");
      return(retObj);
    }
  #ifdef NETDEBUG
    pvm_perror("after invalid loop");
  #endif
```

```
/* hashing ueber die Partitions-IDs */
  pli_hash_temp_general(temps, ts, partition_ids);
  #ifdef NETDEBUG
    pvm_perror("after hashing");
  #endif

/* Templates einpacken */
  if (elsetype == PLE_WAIT)
  {
    operation = 'F';
  }else{
    operation = 'f';
  }

  buffer = pli_pack_tl(temps, operation);
  #ifdef NETDEBUG
    pvm_perror("fetch: after packing");
    printf("max_part: %d \n",ts->max_partitions);
  #endif

  for (i=0; partition_ids[i] != 0; i++)
  {
    #ifdef NETDEBUG
      pvm_perror(" fetch (n.b.): sending fetch");
    #endif

    info = pvm_send(partition_ids[i], 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_fetch (pvm_send)",
            "Pvmd is not responding!");
    #endif

  /* receive message if non blocking operation */
    if (elsetype == PLE_ELSE)
    {
      #ifdef NETDEBUG
        pvm_perror("fetch: (n.b.) receive message.");
```

```
    #endif

    /* receive message from partitions*/
      rbuffer = pvm_recv(partition_ids[i], 1);
      switch (rbuffer){
      case PvmBadParam :
      plp_error("plo_fetch", "Wrong parameters in pvm_recv");
      break;
      case PvmSysErr :
      plp_error("plo_fetch", "Pvmd is not responding");
      break;
      }

      info = pvm_upkint(&tempnumber, 1, 1);
    #ifdef NETDEBUG
      if (info < 0)
      plp_error("plo_fetch (pvm_upkint)",
                     "Can't allocate memory!");
    #endif

      if (tempnumber != 0)
    {
      pvm_freebuf(rbuffer);
       break;
    }
  }
}
/* blocking operation */
if (elsetype == PLE_WAIT)
{
  do{
     /* I'll get the message from anybody */
      rbuffer = pvm_recv(-1, 1);
      switch (rbuffer){
      case PvmBadParam :
      plp_error("plo_fetch", "Wrong parameters in pvm_recv");
      break;
      case PvmSysErr :
```

```
    plp_error("plo_fetch", "Pvmd is not responding");
    break;
    }

  info = pvm_upkint(&tempnumber, 1, 1);
  #ifdef NETDEBUG
    if (info < 0)
    plp_error("plo_fetch (pvm_upkint)",
                     "Can't allocate memory!");
  #endif

  if (tempnumber != 0) /* partition has tuple, get it */
  {
    info = pvm_bufinfo(rbuffer, &bytes, &type, &who);
    #ifdef NETDEBUG
      if (info < 0)
      plp_error("plo_fetch (pvm_bufinfo)",
                           "Error occurs!");
    #endif

    /* non blocking, packing is needed*/
    buffer2 = pli_pack_tl(temps, 'f');

    info = pvm_setsbuf(buffer2);
    #ifdef NETDEBUG
      if (info < 0)
      plp_error("plo_fetch (pvm_setsbuf)",
                           "Error occurs!");
    #endif

    info = pvm_send(who, 1);
    #ifdef NETDEBUG
      if (info < 0)
      plp_error("plo_fetch (pvm_send)",
                     "Pvmd is not responding!");
    #endif

    info = pvm_freebuf(buffer2);
```

```
#ifdef NETDEBUG
  if (info < 0)
  plp_error("plo_fetch (pvm_freebuf)",
                        "Error occurs!");
#endif
/* I'll get the message from who */
rbuffer = pvm_recv(who, 1);
  switch (rbuffer){
  case PvmBadParam :
  plp_error("plo_fetch",
             "Wrong parameters in pvm_recv");
  break;
  case PvmSysErr :
  plp_error("plo_fetch (pvm_recv)",
                  "Pvmd is not responding");
  break;
  }

info = pvm_upkint(&tempnumber, 1, 1);
#ifdef NETDEBUG
  if (info < 0)
  plp_error("plo_fetch (pvm_upkint)",
                   "Can't allocate memory!");
#endif

if (tempnumber == 0)
/* I'm too late, tuple is gone, do it again */
{
  /* free old receive buffer for
                memory optimization*/
  pvm_freebuf(rbuffer);

  info = pvm_setsbuf(buffer);
  #ifdef NETDEBUG
    if (info < 0)
    plp_error("plo_fetch (pvm_setsbuf)",
                       "Error occurs!");
  #endif
```

```
  info = pvm_send(who, 1);
  #ifdef NETDEBUG
    if (info < 0)
    plp_error("plo_fetch (pvm_send)",
                  "Pvmd is not responding!");
  #endif
}else{
  buffer3 = pvm_mkbuf(plp_encoding);
  #ifdef NETDEBUG
  if (buffer3 < 0)
    plp_error("plo_fetch (pvm_mkbuf)",
                    "Can't allocate memory!");
  #endif

  info = pvm_setsbuf(buffer3);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plo_fetch (pvm_setsbuf)",
                          "Error occurs");
  #endif

   info = pvm_pkint(&plp_main_id, 1, 1);
  #ifdef NETDEBUG
   if (info < 0)
    plp_error("plo_fetch (pvm_pkint)",
                    "Can't allocate memory!");
  #endif

/* I find the tuple gefunden!
   Order partitions to delete from list */
  info = pvm_pkstr("t");
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plo_fetch (pvm_pkstr)",
                    "Can't allocate memory!");
  #endif
```

```
    for (i=0; partition_ids[i] != 0; i++)
    {
      if (partition_ids[i] != who)
      {
        info = pvm_send(partition_ids[i], 1);
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plo_fetch (pvm_send)",
                 "Pvmd is not responding!");
        #endif
      }
    }
    info = pvm_freebuf(buffer3);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_fetch (pvm_freebuf)",
                         "Error occurs");
    #endif

  /* there are some messages, kill it*/
    while ((info = pvm_probe(-1, 1)) > 0)
    {
       /* I'll get the message from anybody */
      info = pvm_recv(-1, 1);
        switch (info){
        case PvmBadParam :
        plp_error("plo_fetch",
              "Wrong parameters in pvm_recv");
        break;
        case PvmSysErr :
        plp_error("plo_fetch",
                   "Pvmd is not responding");
        break;
        }
        pvm_freebuf(info);
    }

  }
```

```
      }
    }while (tempnumber == 0);
  }
    tempnum->pbp_integer = tempnumber;    /* manipulate tempnum */

  if (tempnumber == 0)
    {
      plt_kill_temps(temps);           /* free memory of temps */
    #ifdef NETDEBUG
      pvm_perror("Leave plo_fetch");
    #endif

      return(retObj);
    }

  for (i=1; i < tempnumber; i++)
  {
    temps = temps->next;
  }
  comp = temps->comps;
  for (i=1; i <= temps->arity; i++)
  {
    if (comp->type == PLE_FORMAL)
    {
      #ifdef NETDEBUG
        pvm_perror("fetch: in if FORMALL");
      #endif

        info = pvm_upkint(&length, 1,1);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plo_fetch (pvm_upkint)",
                     "Can't allocate memory!");
      #endif

      if ((objstr =
          (char *)malloc(sizeof (char)*(length+1))) == NULL)
        plp_error("plo_fetch (malloc)",
```

```
                        "Cannot allocate memory!");

        info = pvm_upkstr(objstr);  /* unpack the lvalue */
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plo_fetch (pvm_upkstr)",
                          "Can't allocate memory!");
        #endif

        if (strcmp(objstr,"n") != 0)
        {
          #ifdef NETDEBUG
            pvm_perror("fetch: vor assign");
            pvm_perror(objstr);
          #endif

          tmp = objstr;
          rvalue = pli_scan(&objstr);
          pbp_set_use(rvalue);
             pbo_assign(comp->comp.formal.lvalue, rvalue);
        }
        free(tmp);
      }
      comp = comp->next;
    }
     info = pvm_freebuf(rbuffer);    /* memory optimization */
    info = pvm_freebuf(buffer);     /* free buffer */
      plt_kill_temps(temps);           /* free memory of temps */
    #ifdef NETDEBUG
      pvm_perror("Leave plo_fetch");
    #endif

      return(retObj);
  }
```

### pli_process_in_tup

⟨*Interne O-Operationen*⟩≡
```
extern Pbp_Bool_Type pli_process_in_tup(Pbp_Object *tup);
```

**Profil:** `Rückgabewert`

**Beschreibung:** Überprüft, ob das Tuple aktiv ist, d.h. ob es mindestens ein Element vom Typ `PBP_PROCESS` besitzt.

**Ausnahmen:** Keine.

**Abhängigkeiten:** Keine.

⟨*Implementierung der Operationen*⟩+≡
```
Pbp_Bool_Type pli_process_in_tup(Pbp_Object *Tuple)
{
 Pbp_Bool_Type active = PBP_FALSE;
 PBP_TUPLE_ELEM *element ;
 unsigned int i;

  element =
  ((PBP_TUPLE_HEADER *)Tuple->pbp_comp_header)->First_Element ;
  while (element != NULL)
  {
    if (pbp_type(element->ObjFromElem) == PBP_PROCESS)
    {
      active = PBP_TRUE;
      return(active);
    }
    element = element->nextTupleElem ;
  } /* while */
   return active;
}
```

### pli_pack_tl

⟨*Interne O-Operationen*⟩+≡
```
extern int pli_pack_tl(Ple_Template *temps, char operation);
```

**Profil:** `Rückgabewert`

**Beschreibung:** Kopiert die Liste der Templates `temps` in einen PVM-Puffer. Da aus Effizienzgründen nur die jeweils relevanten Teile kopiert werden, müssen mit Hilfe von `operation` die Tupelraumbefehle unterschieden werden.

**Ausnahmen:** Keine.

**Abhängigkeiten:** `plp_encoding`, `pvm_mkbuf`, `pvm_setsbuf`, `pvm_pkint`, `pvm_pkstr`, `plp_error`, `pvm_error`, `plp_pack_obj`

⟨*Implementierung der Operationen*⟩+≡

```
int pli_pack_tl(Ple_Template *temps, char operation)
{
 extern int plp_encoding;
 Ple_Template *temp;         /* help pointer for the templates */
 Ple_Temp_Comp *comp;        /* help pointer for the components */
 int arity;
 int length;
 int info;
 int buffer;
 int i;

  /* make buffer */
  buffer = pvm_mkbuf(plp_encoding);
  #ifdef NETDEBUG
   if (buffer < 0)
    plp_error("pli_pack_tl (pvm_mkbuf)",
                        "Can't allocate memory!");
  #endif

  /* make it to the actual buffer */
  info = pvm_setsbuf(buffer);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("pli_pack_tl (pvm_setsbuf)", "Error occurs");

    pvm_perror("nach mkbuf");
  #endif
```

```
 info = pvm_pkint(&plp_main_id, 1, 1);
#ifdef NETDEBUG
 if (info < 0)
  plp_error("pli_pack_tl (pvm_pkint)",
                     "Can't allocate memory!");

  pvm_perror("nach pkint");
#endif

info = pvm_pkstr(&operation);
#ifdef NETDEBUG
 if (info < 0)
  plp_error("pli_pack_tl (pvm_pkstr)",
                     "Can't allocate memory!");

  pvm_perror("nach pkstr");
 #endif

 temp = temps;
 comp = temp->comps;

 while(temp != NULL)
 {
  #ifdef NETDEBUG
    pvm_perror("in while");
  #endif

  arity = temp->arity;
  info = pvm_pkint(&arity,1,1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("pli_pack_tl (pvm_pkint)",
                     "Can't allocate memory!");

    pvm_perror("nach pkint");
  #endif
```

```
if (temp->condition != NULL)
  {
      plp_pack_obj(temp->condition);
}else{
  length = 1;
  info= pvm_pkint(&length,1,1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("pli_pack_tl (pvm_pkint)",
                     "Can't allocate memory!");
  #endif

  info= pvm_pkstr("c");  /* no condition */
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("pli_pack_tl (pvm_pkstr)",
                     "Can't allocate memory!");

    pvm_perror("no condition");
  #endif
}

  for (i=1; i <= arity; i++)
{
  #ifdef NETDEBUG
    pvm_perror("in comp loop");
  #endif

  if (comp->type == PLE_ACTUAL)
  {
    #ifdef NETDEBUG
      pvm_perror("in if actual");
    #endif
        info = pvm_pkstr("a");
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("pli_pack_tl (pvm_pkstr)",
                     "Can't allocate memory!");
```

```
  #endif

  plp_pack_obj(comp->comp.actual);
}else{
  #ifdef NETDEBUG
    pvm_perror("in if formal");
  #endif

  if (comp->comp.formal.lvalue == NULL)
  {
    info = pvm_pkstr("o");
    #ifdef NETDEBUG
     if (info < 0)
      plp_error("pli_pack_tl (pvm_pkstr)",
                   "Can't allocate memory!");
    #endif
  }else{
    info = pvm_pkstr("f");
    #ifdef NETDEBUG
      if (info < 0)
      plp_error("pli_pack_tl (pvm_pkstr)",
                   "Can't allocate memory!");
    #endif
   /* packing of lvalue is not needed */
  }
  if (operation == 'm')
  {
      /* there is no into-function */
    if (comp->comp.formal.into == NULL)
    {
      length = 1;
      info= pvm_pkint(&length, 1, 1);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("pli_pack_tl (pvm_pkint)",
                   "Can't allocate memory!");
      #endif
```

```
          info = pvm_pkstr("c");
          #ifdef NETDEBUG
          if (info < 0)
            plp_error("pli_pack_tl (pvm_pkstr)",
                       "Can't allocate memory!");
          #endif
        }else{
          #ifdef NETDEBUG
            pvm_perror("PACKE INTO-FUNKTON EIN");
          #endif

          plp_pack_obj(comp->comp.formal.into);
                    /* into-function packen*/
        }
      }
      if (operation == 'M' || operation == 'F')
      {
        length = 1;
        info= pvm_pkint(&length, 1, 1);
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("pli_pack_tl (pvm_pkint)",
                       "Can't allocate memory!");
        #endif

        info = pvm_pkstr("c");
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("pli_pack_tl (pvm_pkstr)",
                       "Can't allocate memory!");
        #endif
      }
    }
    comp = comp->next;
  }
    temp = temp->next;
    if (temp == NULL)
  {
```

```
        info = pvm_pkstr("q");
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("pli_pack_tl (pvm_pkstr)",
                        "Can't allocate memory!");
      #endif
    }else{
      info = pvm_pkstr("n");
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("pli_pack_tl (pvm_pkstr)",
                         "Can't allocate memory!");
      #endif
    }
  }
  return buffer;
}
```

**plo_meet**

⟨*Exportierte O-Operationen*⟩+≡

```
extern Pbp_Object *plo_meet(Ple_Template *temps, Pbp_Object *tsid,
          Pbp_Object *tempnum, Ple_Else_Type elsetype,
          pbp_form_par);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die Nummer des ausgewählten Tempates. |
| `temps` | ist die Liste der Templates für das gesuchte Tupel. |
| `tsid` | ist Tupelraumidentität. |
| `elsetype` | gibt an, ob auf ein passendes Tupel gewartet werden muß. |
| `pbp_form_par` | ist das Makro für die Ausnahmebehandlungsparameter. |

**Beschreibung:** `plo_meet` versucht, für eins der gegebenen Templates in `temps` ein passendes Tupel in der gehasten Partition des Tupelraumes mit der Identität `tsid` zu besuchen. Den Formals des Templates werden die entsprechenden Werte zugewiesen. Falls ein `into` statement vorhanden ist. wird das Tupel entsprechend verändert. Anschließend wird überprüft, ob es jetzt zu einem wartenden Prozeß paßt. Ist jedoch kein passendes Tupel vorhanden, und `elsetyp` ist gleich `PLE_WAIT`, so wird gewartet, bis ein passendes Tupel in diesen Tupelraum eingefügt wird. Der Rückgabewert ist die Nummer des ausgewählten Templates. Falls kein passendes Tupel vorhanden ist, und auch nicht gewartet wird, ist der Rückgabewert 0.

**Ausnahmen:** `type_mismatch`, `ts_invalid_id`

**Abhängigkeiten:** `plp_encoding`, `pbp_type`, `plo_find_ts`, `pvm_perror`, `pvm_initsend`, `pvm_pkint`, `pvm_pkstr`, `pvm_send`, `pvm_recv`, `pvm_freebuf`, `pvm_bufinfo`, `pvm_setsbuf`, `pvm_mkbuf`, `pli_pack_tl`, `pvm_probe`, `plt_kill_temp`, `pvm_upkint`, `pvm_upkstr`, `pbp_set_use`, `pbo_assign`

⟨*Implementierung der Operationen*⟩+≡

```
Pbp_Object *plo_meet(Ple_Template *temps, Pbp_Object *tsid,
                     Pbp_Object *tempnum, Ple_Else_Type elsetype,
                                    pbp_form_par)
{
 extern int plp_encoding;
 Pli_Op_Type optype = PLI_MEET;  /* parameter to plo_match_in_ts */
 Pbp_Object *newtup = pbp_omega; /* the changed tuple if there is
                           a meet-into statement */
 Pbp_Object *retObj = pbp_omega;/* the return value of an
                             exception handler */
 Pbp_Object *rvalue;    /* result value of condition function */
 Ple_Template *temp;        /* help pointer for the templates */
```

```
 Ple_Temp_Comp *comp;      /* help pointer for the components */
 Pli_Tuple_Space *ts;  /* the tuplespace with the identity tsid */
 pbh_decl_var;
 Ple_Process myid;        /* id of process */
 int info;              /* error variable for pvm functions */
 int arity;
 int tempnumber;      /* ... */
 int i;
 char *objstr;
 char *tmp;
 int length;
 int buffer, buffer2, buffer3, rbuffer;
 int bytes, type;
 int who;
 char operation;
 char *string;
 int partition_ids[PLI_MAX_PARTITIONS];

  assert(temps != NULL);          /* compiler error? */
  assert(tsid != NULL);           /* compiler error? */

  #ifdef NETDEBUG
    pvm_perror("plo_meet: Here I am!");
    string = pli_ots(tsid, NULL);
    pvm_perror(string);
  #endif

    if(pbp_type(tsid) != PBP_ATOM)  /* type_mismatch */
    {
    pvm_perror("plo_meet : type_mismatch (atom)");
    pbh_type_mismatch(retObj, "in plo_meet");
    return(retObj);
    }
  #ifdef NETDEBUG
    pvm_perror("meet: nach test");
  #endif

#ifdef TRACE
```

```
/* trace informations */
  if(gps_linda_trace)
    {
    fprintf(gps_trace_file, "o m b %s %d %i %s\n",
            pli_ots(tsid, gps_return_address),
            plp_thread_id(), pbh_get_line(), pbh_get_file());
    }
#endif
  /* finding the tuple space with the identity tsid */
    ts = plo_find_ts(tsid, PLI_FIND);
    if (ts == NULL)          /* ts_invalid_id */
    {

#ifdef TRACE
/* trace informations */
   if(gps_linda_trace)
   {
    fprintf(gps_trace_file, "o m f %s %d %i %s\n",
              pli_ots(tsid, gps_return_address),
              plp_thread_id(), pbh_get_line(), pbh_get_file());
   }
#endif

    pbh_ts_invalid_id(retObj, "in  plo_meet");
    return(retObj);
    }

  /* hashing ueber die Partitions-IDs */
  pli_hash_temp_general(temps, ts, partition_ids);
  #ifdef NETDEBUG
    pvm_perror("nach dem hashing");
  #endif

  /* Templates einpacken */
  if (elsetype == PLE_WAIT)
  {
    operation = 'M';
  }else{
```

```
    operation = 'm';
  }

  buffer = pli_pack_tl(temps, operation);

/* send deposit message to partition */

  for (i=0; partition_ids[i] != 0; i++)
  {
    info = pvm_send(partition_ids[i], 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_meet (pvm_send)",
                      "Pvmd is not responding!");
    #endif
  /*Antwort empfangen, wenn nicht blockiernd */
    if (elsetype == PLE_ELSE)
    {
       info = pvm_recv(partition_ids[i], 1);
      /* I get the message from the partition */
       switch (info){
      case PvmBadParam :
        plp_error("plo_meet", "Wrong parameters in pvm_recv");
        break;
       case PvmSysErr :
        plp_error("plp_meet", "Pvmd is not responding");
        break;
       }

       info = pvm_upkint(&tempnumber, 1, 1);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plo_meet (pvm_pkstr)",
          "Can't allocate memory!");
        #endif

      if (tempnumber != 0) break;
    }
```

```
}
/* --------- Blockierend, fuehre Protokoll aus----------------*/
if (elsetype == PLE_WAIT)
{
  do{
    /* I'll get the message from anybody */
      rbuffer = pvm_recv(-1, 1);
      switch (rbuffer){
    case PvmBadParam :
      plp_error("plo_meet", "Wrong parameters in pvm_recv");
      break;
    case PvmSysErr :
      plp_error("plo_meet", "Pvmd is not responding");
      break;
      }

    info = pvm_upkint(&tempnumber, 1, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_meet (pvm_upkint)",
                       "Can't allocate memory!");
    #endif

    if (tempnumber != 0) /* Tuple ist da, hole es ab */
    {
      info = pvm_bufinfo(rbuffer, &bytes, &type, &who);
      #ifdef NETDEBUG
        if (info < 0)
        plp_error("plo_meet (pvm_bufinfo)",
                          "Error occurs!");
      #endif

      /* nicht blockierend , Einpacken leider noetig */
      buffer2 = pli_pack_tl(temps, 'm');

      info = pvm_setsbuf(buffer2);
      #ifdef NETDEBUG
        if (info < 0)
```

```
  plp_error("plo_meet (pvm_setsbuf)",
                   "Error occurs!");
#endif

info = pvm_send(who, 1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plo_meet (pvm_send)",
              "Pvmd is not responding!");
#endif

info = pvm_freebuf(buffer2);
#ifdef NETDEBUG
  if (info < 0)
  plp_error("plo_meet (pvm_freebuf)",
                     "Error occurs!");
#endif
/* I'll get the message from who */
rbuffer = pvm_recv(who, 1);
switch (rbuffer){
case PvmBadParam :
  plp_error("plo_meet",
           "Wrong parameters in pvm_recv");
  break;
case PvmSysErr :
  plp_error("plo_meet (pvm_recv)",
               "Pvmd is not responding");
  break;
}

info = pvm_upkint(&tempnumber, 1, 1);
#ifdef NETDEBUG
  if (info < 0)
  plp_error("plo_meet (pvm_upkint)",
                 "Can't allocate memory!");
#endif

if (tempnumber == 0)
```

```
   /* Tuple verpasst, noch mal anfragen */
{
  info = pvm_setsbuf(buffer);
  #ifdef NETDEBUG
    if (info < 0)
    plp_error("plo_meet (pvm_setsbuf)",
                     "Error occurs!");
  #endif

  info = pvm_send(who, 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plo_meet (pvm_send)",
             "Pvmd is not responding!");
  #endif
}else{
  buffer3 = pvm_mkbuf(plp_encoding);
  #ifdef NETDEBUG
    if (buffer3 < 0)
    plp_error("plo_meet (pvm_mkbuf)",
               "Can't allocate memory!");
  #endif

  info = pvm_setsbuf(buffer3);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plo_meet (pvm_setsbuf)",
                      "Error occurs");
  #endif

   info = pvm_pkint(&plp_main_id, 1, 1);
  #ifdef NETDEBUG
   if (info < 0)
    plp_error("plo_meet (pvm_pkint)",
               "Can't allocate memory!");
  #endif

  /* Tuple gefunden! Alle Partitionen zum loeschen
```

```
     aus Liste auffordern */
  info = pvm_pkstr("t");
  #ifdef NETDEBUG
    if (info < 0)
    plp_error("plo_meet (pvm_pkstr)",
               "Can't allocate memory!");
  #endif

  for (i=0; partition_ids[i] != 0; i++)
  {
    if (partition_ids[i] != who)
    {
      info = pvm_send(partition_ids[i], 1);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plo_meet (pvm_send)",
               "Pvmd is not responding!");
      #endif
    }
  }
  info = pvm_freebuf(buffer3);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plo_meet (pvm_freebuf)",
                    "Error occurs");
  #endif

/* Alle schon angekommenen Nachrichten aus
       Pvm-Puffer loeschen */
  while ((info = pvm_probe(-1, 1)) > 0)
  {
     /* I'll get the message from anybody */
    info = pvm_recv(-1, 1);
      switch (info){
      case PvmBadParam :
      plp_error("plo_meet",
             "Wrong parameters in pvm_recv");
      break;
```

```
            case PvmSysErr :
            plp_error("plo_meet",
                    "Pvmd is not responding");
            break;
            }
        }
      }
    }
  }while (tempnumber == 0);
}
tempnum->pbp_integer = tempnumber;

if (tempnumber == 0)
  {
    plt_kill_temps(temps);          /* free memory of temps */
  #ifdef NETDEBUG
    pvm_perror("Leave plo_meet");
  #endif
    return(retObj);
  }

  for (i=1; i < tempnumber; i++)
  {
  temps = temps->next;
  }
  comp = temps->comps;
for(i=1; i <= temps->arity; i++)
  {
  if (comp->type == PLE_FORMAL)
  {
    info = pvm_upkint(&length, 1, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_meet (pvm_upkint)",
                  "Can't allocate memory!");
    #endif

    if ((objstr =
```

```
          (char *)malloc(sizeof(char)*(length+1))) == NULL)
          plp_error("plo_meet (objstr)",
                          "Can't allocate memory!");

        info = pvm_upkstr(objstr);  /* unpack the lvalue */
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plo_meet (pvm_upkstr)",
                           "Can't allocate memory!");
        #endif

        if (strcmp(objstr,"n") != 0)
        {
          tmp = objstr;
          rvalue = pli_scan(&objstr);
          pbp_set_use(rvalue);
          pbo_assign(comp->comp.formal.lvalue, rvalue);
        }
        free(tmp);
      }
        comp = comp->next;
    }
    plt_kill_temps(temps);           /* free memory of temps*/
    #ifdef NETDEBUG
      pvm_perror("Leave plo_meet");
    #endif
    return(retObj);
  }
```

## C.2.4 Standartbibliotheksfunktionen

### pll_create_ts

⟨*Exportierte O-Operationen*⟩+≡

```
  extern Pbp_Object *pll_create_ts(Pbp_Object *limit, pbp_form_par);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die Identität des erzeugten Tupleraums. |
| `limit` | sollte die maximale Anzahl von Tupeln angeben; sollte vom Typ `integer` oder `om` sein. |
| `pbp_form_par` | beinhalten die Ausnahmeparameter. |

**Beschreibung:** `pll_create_ts` erzeugt `plp_max_partitions` Partitionsprozesse eines neuen Tupelraumes und hängt ihn an die Liste aller Tupelräume an. Wenn ein Eval-Server im Netz vorhanden ist, wird der Tupelraum dort angemeldet.

**Ausnahmen:** `type_mismatch`

**Abhängigkeiten:** `plp_hash_mode`, `plp_max_partitions`, `plp_my_name`, `plp_filename`, `plp_loadbalancing` `pvm_catchout`, `pba_newat`, `pli_ots`, `pbp_type`, `pvm_perror`, `pvm_exit`, `pvm_spawn`, `plp_min_load`, `plp_error`, `allTupleSpaces`, `plp_eval`, `pvm_initsend`, `pvm_pkint`, `pvm_pkstr`, `plp_pack_obj`, `pvm_send`, `pbo_copy`

⟨*Implementierung der Operationen*⟩+≡

```
Pbp_Object *pll_create_ts(Pbp_Object *limit, pbp_form_par)
{
 extern int plp_hash_mode;
 extern int plp_max_partitions;
 extern char *plp_my_name;
 extern char plp_filename[12];
 register unsigned i;
 Pbp_Object *retObj = pbp_omega;/* the return value of an
                           exception handler */
 Pli_Tuple_Space *newTs;      /* the new created tuple space */
 pbh_decl_var;               /* the exception handling variables */
 char *max;                    /* limit converted to string */
 char *buffer;
 char **tmp_argv;
 Ple_Hostname_List *where;
 int spawned_processes = 0;
 int partitiontids[PLI_MAX_PARTITIONS];
 int spawned_proc;
 int proc;
 int info;
```

```
/* output redirected to master task */
pvm_catchout(stdout);

/* creating a new tuple space */
newTs = (Pli_Tuple_Space *)malloc(sizeof(Pli_Tuple_Space));
if(newTs == NULL)
  plp_error("pll_create_ts (malloc)", "not enough memory");

newTs->atom = pba_newat();
pbp_set_use(newTs->atom);
newTs->atom_str = pli_ots(newTs->atom, gps_return_address);

 /* Standard Konfiguration setzen */
newTs->max_partitions = plp_max_partitions;
newTs->hash_mode = plp_hash_mode;

if(pbp_type(limit) == PBP_OM)
{
  max = (char *)calloc(3, sizeof(char));
  if(max == NULL)
    plp_error("pll_create_ts (calloc)", "not enough memory");
  max = "-1";             /* no limit */
}else if(pbp_type(limit) == PBP_INTEGER){
    if(limit->pbp_integer <= 0)
  {
      max = (char *)calloc(2, sizeof(char));
    if(max == NULL)
      plp_error("pll_create_ts (malloc)",
                       "not enough memory");
      max = "0";             /* ever full tuple space */
    }else{
      buffer = (char *)calloc(80, sizeof(char));
    if(buffer == NULL)
      plp_error("pll_create_ts (malloc)",
                         "not enough memory");
      sprintf(buffer, "%d", limit->pbp_integer);
      max = (char *)calloc(strlen(buffer)+1,sizeof(char));
    if(max == NULL)
```

```
      plp_error("pll_create_ts (malloc)",
                         "not enough memory");
      sprintf(max, "%d", limit->pbp_integer);
      free(buffer);
    }
  }else{                              /* type_mismatch */
  pbh_type_mismatch(retObj,"in pll_create_ts");
    pvm_perror("type mismatch in pll_create_ts: limit");
    pvm_exit();
    return(retObj);
}

/* argument list for partitions */
tmp_argv = (char **) malloc (sizeof (char *)*4);
if(tmp_argv == NULL)
  plp_error("pll_create_ts (malloc)", "not enough memory");

tmp_argv[0] = (char *)calloc(2, sizeof(char));
if(tmp_argv[0] == NULL)
  plp_error("pll_create_ts (malloc)", "not enough memory");

sprintf(tmp_argv[0],"p");
tmp_argv[1] = 0;          /* newTs->atom_str;*/
tmp_argv[2] = max;

#ifdef NETDEBUG
  pvm_perror(plp_my_name);
#endif

if (plp_loadbalancing == PBP_FALSE) /* Keine Loadbalancierung */
{
    spawned_processes = pvm_spawn(plp_my_name, tmp_argv,
          (PvmTaskDefault),"", newTs->max_partitions,
                            partitiontids);
}else{                              /* Loadbalancierung */
  for (i=0; i < newTs->max_partitions; i++)
  {
    where = plp_min_load(1);
```

```
      proc = pvm_spawn(plp_my_name, tmp_argv, (PvmTaskHost),
         where->hostname, 1, &spawned_proc);
      spawned_processes = spawned_processes + proc;
      partitiontids[i] = spawned_proc;
      free(where->hostname);
    }
    free(where);
  }

  if (spawned_processes < newTs->max_partitions)
    plp_error("pll_create_ts" ,"Can't spawn all partitions!");

  /* save the partition ids */
  for (i=0; i < newTs->max_partitions; i++)
  newTs->tids[i]  = partitiontids[i];
  for (i=newTs->max_partitions; i < 20; i++)
  newTs->tids[i] = 0;

  /* putting newTs in the list of all tuple spaces */
  newTs->next = allTupleSpaces;
  allTupleSpaces = newTs;

  /* send all partitions the partitiontids */
  info = pvm_initsend(plp_encoding);
  #ifdef NETDEBUG
   if (info < 0)
    plp_error("pll_create_ts (pvm_initsend)",
                       "Can't allocate memory!");
  #endif

  info = pvm_pkint(newTs->tids, 20, 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("pll_create_ts (pvm_pkint)",
                     "Can't allocate memory!");
  #endif

  info =  pvm_pkint(&(newTs->hash_mode), 1, 1);
```

```
#ifdef NETDEBUG
if (info < 0)
  plp_error("pll_create_ts (pvm_pkint)",
                   "Can't allocate memory!");
#endif

for (i=0; i < plp_max_partitions; i++)
{
  info = pvm_send(newTs->tids[i], 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("pll_create_ts (pvm_send)",
                     "Pvmd is not responding!");
  #endif
}


if (plp_eval == PBP_TRUE)
{
  /* beim eval-server anmelden */
  info = pvm_initsend(PvmDataDefault);
  #ifdef NETDEBUG
   if (info < 0)
    plp_error("pll_create_ts (pvm_initsend)",
                     "Can't allocate memory!");
  #endif

  info = pvm_pkint(&plp_main_id, 1, 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("pll_create_ts (pvm_pkint)",
                   "Can't allocate memory!");
  #endif

  info = pvm_pkstr("i");
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("pll_create_ts (pvm_upkstr)",
```

```
                    "Can't allocate memory!");
    #endif

    plp_pack_obj(newTs->atom);

    info = pvm_pkint(newTs->tids, 20, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("pll_create_ts (pvm_pkint)",
                      "Can't allocate memory!");
    #endif

    info = pvm_send(plp_eval_tid, 5);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("pll_create_ts (pvm_send)",
                      "Pvmd is not responding!");
    #endif
  }
    /* returning the tuple space identity */
  return(pbo_copy(newTs->atom));
}
```

### pll_exists_ts

⟨*Exportierte O-Operationen*⟩+≡

```
extern Pbp_Object *pll_exists_ts(Pbp_Object *tsid, pbp_form_par);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist `true`, falls ein Tupelraum mit der Identität `tsid` existiert. |
| `tsid` | ist die Tupelraumidentität. |
| `pbp_form_par` | beinhaltet die Ausnahmeparameter. |

**Beschreibung:** `pll_exists_ts` gibt an, ob ein Tupelraum mit der Identität `tsid` existiert. Die Funktionalität ist vollkommen in `plo_find_ts` verborgen.

**Ausnahmen:** `type_mismatch`

**Abhängigkeiten:** `pbp_type`, `plo_find_ts`

⟨*Implementierung der Operationen*⟩+≡

```
Pbp_Object *pll_exists_ts(Pbp_Object *tsid, pbp_form_par)
{
 Pbp_Object *retObj = pbp_omega;/* the return value of an
                               exception handler */
 pbh_decl_var;                 /* the exception handling
                                variables */

  assert(tsid != NULL);            /* compiler error? */

  if(pbp_type(tsid) != PBP_ATOM) /* type_mismatch */
  {
      pvm_perror(
    "pll_exists_ts : Type mismatch! Type of tsid != PBP_ATOM");
    pbh_type_mismatch(retObj,"in pll_exists_ts");
    return(retObj);
  }

  if (plo_find_ts(tsid, PLI_FIND) == NULL)
  {
    return(pbp_false);             /* returning false */
  }else{
    return(pbp_true);              /* returning true */
  }
}
```

### pll_clear_ts

⟨*Exportierte O-Operationen*⟩+≡

```
extern Pbp_Object *pll_clear_ts(Pbp_Object *tsid, pbp_form_par);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist om. Nur im Ausnahmefall wird der Rückgabewert des Handlers hochgereicht. |
| `tsid` | ist die Identität des Tupelraums, der geleert werden soll. |
| `pbp_form_par` | beinhaltet die Ausnahmeparameter. |

**Beschreibung:** `pll_clear_ts` entfernt alle Tupel (auch die aktiven) aus dem Tupelraum mit der Identität `tsid`. Die Funktionalität ist vollkommen in `plo_find_ts` gekapselt.

**Ausnahmen:** `type_mismatch`, `ts_invalid_id`

**Abähngigkeiten:** `pbp_type`, `plo_find_ts`

⟨*Implementierung der Operationen*⟩+≡

```
Pbp_Object *pll_clear_ts(Pbp_Object *tsid, pbp_form_par)
{
 register unsigned hashNum;    /* number of the actual hash list */
 Pbp_Object *retObj = pbp_omega;/* the return value of an
                             exception handler */
 Pli_Tuple_Space *ts;  /* the tuple space with the identity tsid */
 pbh_decl_var;              /* the exception handling variables */
 int i;                 /* Index*/
 int info;                /* PVM Fehlervariable */

  assert(tsid != NULL);            /* compiler error? */

  if(pbp_type(tsid) != PBP_ATOM) /* type_mismatch */
  {
    pvm_perror(
    "pll_clear_ts : Type_mismatch! Type of tsid != PBP_ATOM");
    pbh_type_mismatch(retObj, "in pll_clear_ts");
    return(retObj);
  }

#ifdef TRACE
/* trace informations */
  if(gps_linda_trace)
  {
    fprintf(gps_trace_file, "m c b %s %d %i %s\n",
```

```
        pli_ots(tsid, gps_return_address),
        plp_thread_id(), pbh_get_line(), pbh_get_file());
  }
#endif

  ts = plo_find_ts(tsid, PLI_CLEAR);  /* finding the tuplespace
                              with the identity tsid */
  if (ts == NULL)             /* ts_invalid_id */
  {
#ifdef TRACE
/* trace informations */
    if(gps_linda_trace)
    {
      fprintf(gps_trace_file, "m c f %s %d %i %s\n",
          pli_ots(tsid, gps_return_address),
          plp_thread_id(), pbh_get_line(), pbh_get_file());
    }
#endif

    pbh_ts_invalid_id(retObj, "in pll_clear_ts");
    return(retObj);
  }
  return(retObj);
}
```

### pll_remove_ts

⟨*Exportierte O-Operationen*⟩+≡

```
extern Pbp_Object *pll_remove_ts(Pbp_Object *tsid, pbp_form_par);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist om. Nur im Ausnahmefall wird der Rückgabewert des Handlers hochgereicht. |
| `tsid` | ist die Identität des Tupelraums, der gelöscht werden soll. |
| `pbp_form_par` | beinhaltet die Ausnahmeparameter. |

**Beschreibung:** löscht den Tupelraum mit der Identität `tsid` und alle aktiven Tupel des Tupelraumes.

**Ausnahmen:** `type_mismatch`, `ts_invalid_id`

**Abhängigkeiten:**

⟨*Implementierung der Operationen*⟩+≡

```
Pbp_Object *pll_remove_ts(Pbp_Object *tsid, pbp_form_par)
{
 register unsigned listNum;    /* number of the actual hash list */
 Pbp_Object *retObj = pbp_omega; /* the return value of an
                           exception handler */
 Pli_Hash_List *kill, *next;      /* pointers to members of a hash list */
 Pli_Tuple_Space *prev;           /* the tuple space with the identity tsid */
 Pli_Tuple_Space  *ts = NULL;      /* the previous tuple space */
 pbh_decl_var;              /* the exception handling variables */
 int info;
 int i;

  assert(tsid != NULL);            /* compiler error? */

  /* remove it from my list */
  ts = plo_find_ts(tsid, PLI_REMOVE);

  if(ts == NULL)
  {
    /* Kann ich hier pvm_exit aufrufen, oder nicht? */
    pbh_ts_invalid_id(retObj, "in pll_clear_ts");
    return(retObj);
  }

  free(ts);
```

```
  return(retObj);
}
```

### pli_hash_tup_general

⟨*Interne O-Operationen*⟩+≡

```
extern int pli_hash_tup_general(Pbp_Object *tup,
                        Pli_Tuple_Space *ts);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die Tid der gehashten Partition. |
| `tup` | ist das Tupel, welches gehasht werden soll. |
| `ts` | ist der Tupelraum, aus dem die Partitionen stammen. |

**Beschreibung:** Ruft die durch `plp_hashmode` gewählte Hashfunktion auf.

**Ausnahmen:** Wenn ein falscher Hashmodus angegeben wurde, wird die Standardhashfunktion aufgerufen..

**Abhängigkeiten:** `pli_hash_tup_standard`, `pli_hash_tup_one`, `pli_hash_tup_two`, `pvm_perror`

⟨*Implementierung der Operationen*⟩+≡

```
int pli_hash_tup_general(Pbp_Object *tup, Pli_Tuple_Space *ts)
{
 int partition;
  switch(ts->hash_mode)
  {
    case 0:
      partition = pli_hash_tup_standard(tup, ts);
      break;
    case 1:
      partition = pli_hash_tup_one(tup,ts);
      break;
    case 2:
      partition = pli_hash_tup_two(tup,ts);
      break;
    default:
pvm_perror("Hash_mode not available! Use standard function");
      partition = pli_hash_tup_standard(tup, ts);
      break;
  }
  return partition;
}
```

## pli_hash_temp_general

⟨*Interne O-Operationen*⟩+≡

```
extern void pli_hash_temp_general(Ple_Template *temp,
                                  Pli_Tuple_Space *ts,
                                int *partition_ids);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist `partition_ids`, ein Feld mit allen gehashten Partitonstids. Nach der letzten Partitonstid wird eine 0 in das Feld eingefügt, um das Ende zu markieren. |
| `temp` | ist die Liste der Templates. |
| `ts` | ist der Tupelraum aus dem die zu hashenden Partitonen stammen. |

**Beschreibung:** Ruft die gewählte Hashfunktion auf.

**Ausnahmen:** Bei falschem Hashmodus wird die Standardhashfunktion aufgerufen.

**Abhängigkeiten:** `pli_hash_temp_standard`, `pli_hash_temp_one`, `pli_hash_temp_two`, `pvm_perror`

⟨*Implementierung der Operationen*⟩+≡

```
void pli_hash_temp_general(Ple_Template *temp,
               Pli_Tuple_Space *ts, int *partition_ids)
{
  switch(ts->hash_mode)
  {
    case 0:
      pli_hash_temp_standard(temp, ts, partition_ids);
      break;
    case 1:
      pli_hash_temp_one(temp, ts, partition_ids);
      break;
    case 2:
      pli_hash_temp_two(temp, ts, partition_ids);
      break;
    default:
pvm_perror("hash_mode not available! Use standard function");
      pli_hash_temp_standard(temp, ts, partition_ids);
      break;
  }
}
```

**pli_hash_tup_standard**

⟨*Interne O-Operationen*⟩+≡

```
extern int pli_hash_tup_standard(Pbp_Object *tup,
                                 Pli_Tuple_Space *ts);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die gehashte Partitionstid. |
| `tup` | ist das zu hashende Tupel. |
| `ts` | ist der Tupelraum, dessen Partitionen gehasht werden sollen. |

**Beschreibung:** Die Summe der Typwerte aller Tupelelemente wird durch die Anzahl der Elemente des Tupels geteilt. Dieser Wert wird nun mit Hilfe der Modulo-Funktion in das Intervall *[0, Anzahl der Partitionen im Tupelraum-1]*. Ist das Tupel leer wird immer auf 0 abgebildet. Dieser Wert wird als Index auf dem Feld der Partitionstids interpretiert. Die Abbildung ist immer eindeutig.

**Ausnahmen:** Keine.

**Abhängigkeiten:** `pbp_type`

⟨*Implementierung der Operationen*⟩+≡

```
int pli_hash_tup_standard(Pbp_Object *tup, Pli_Tuple_Space *ts)
{
 PBP_TUPLE_ELEM *element;
 int sum = 0;
 int partition;

  if (tup->pbp_comp_number == 0)
  {
    return (ts->tids[0]);
  }
  element =
  ((PBP_TUPLE_HEADER*)tup->pbp_comp_header)->First_Element ;
  while (element != NULL)
  {
    sum = sum + pbp_type(element->ObjFromElem);
    element = element->nextTupleElem;
  }
  partition = (sum/tup->pbp_comp_number)%ts->max_partitions;
  return (ts->tids[partition]);
}
```

### pli_hash_tup_one

⟨*Interne O-Operationen*⟩+≡
```
extern int pli_hash_tup_one(Pbp_Object *tup, Pli_Tuple_Space *ts);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die gehashte Partitionstid. |
| `tup` | ist das zu hashende Tupel. |
| `ts` | it der Tupelraum, dessen Partitionen gehasht werden sollen. |

**Beschreibung:** Wie Standardhashing, nur wird hier der Hashwert nur aus der Länge des Tupels errechnet.

**Ausnahmen:** Keine.

**Abhängigkeiten:** Keine.

⟨*Implementierung der Operationen*⟩+≡
```
int pli_hash_tup_one(Pbp_Object *tup, Pli_Tuple_Space *ts)
{
  if (tup->pbp_comp_number == 0)
  {
    return (ts->tids[0]);
  }else{
    return (ts->tids[tup->pbp_comp_number%ts->max_partitions]);
  }
}
```

### pli_hash_tup_two

⟨*Interne O-Operationen*⟩+≡
```
extern int pli_hash_tup_two(Pbp_Object *tup, Pli_Tuple_Space *ts);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die gehashte Partitionstid. |
| `tup` | ist das zu hashende Tupel. |
| `ts` | it der Tupelraum, dessen Partitionen gehasht werden sollen. |

**Beschreibung:** Wie Standardhashing, nur wird hier angenommen, daß der erste Wert des Tupels ein Integer ist. Der Hashwert wir nun aus diesem Integer-Wert errechnet. Wenn das Tupel leer ist, wird auf 0 abgebildet.

**Ausnahmen:** Das erste Element des Tuples ist nicht vom Typ `PBP_INTEGER`.

**Abhängigkeiten:** `pbp_type`, `plp_error`

⟨*Implementierung der Operationen*⟩+≡

```
int pli_hash_tup_two(Pbp_Object *tup, Pli_Tuple_Space *ts)
{
 PBP_TUPLE_ELEM *element;
 Pbp_Object *object;
 int sum = 0;
 int partition;

  if (tup->pbp_comp_number == 0)
  {
    return (ts->tids[0]);
  }
  element =
  ((PBP_TUPLE_HEADER*)tup->pbp_comp_header)->First_Element ;
  object = element->ObjFromElem;
  if (pbp_type(object) != PBP_INTEGER)
    plp_error("pli_hash_tup_two",
"Wrong hash function! First element of tuple is not of type PBP_INTEGER!");

  partition = (object->pbp_integer)%ts->max_partitions;
  return (ts->tids[partition]);
}
```

### pli_hash_temp_standard

⟨*Interne O-Operationen*⟩+≡

```
extern void pli_hash_temp_standard(Ple_Template *temp,
                                   Pli_Tuple_Space *ts,
                                    int *partition_ids);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist `partition_ids`, das Feld der Partitiontids. |
| `temp` | ist die Liste der Templates. |
| `ts` | ist der Tupelraum in dem die zu hashenden Partitionen liegen. |

**Beschreibung:** Die Berechnung erfolgt für jedes Template einzeln. Die Typwerte aller Actuals werden aufsummiert . Dabei sind Werte von (0=om bis 8 möglich). Die Anzahl der Formals und die Summe der Typwerte werden addiert. Hieraus ergibt sich ein Intervall, für das mit Hilfe der Modulo-Funktion die Indizes der Partitiontids berechnet wird. Dies wird für jedes Template in `temp` durchgeführt und doppelte Tids eleminiert.

**Ausnahmen:** Keine.

**Abhängigkeiten:** `pbp_type`

⟨*Implementierung der Operationen*⟩+≡

```
void pli_hash_temp_standard(Ple_Template *temp,
              Pli_Tuple_Space *ts, int *partition_ids)
{
 int sum;
 int index = 0;
 int minbound = -1;
 int maxbound = -1;
 int tmpmin,tmpmax;
 int formal_counter = 0;
 int i,z;            /* Indizes */
 Ple_Template *template;
 Ple_Temp_Comp *comp;

  template = temp;

   while(template != NULL)
   {
    sum = 0;
    tmpmin = tmpmax = 0;
    comp = template->comps;
    formal_counter = 0;
    if (template->arity != 0)
```

```
    {
      for (i=1; i <= template->arity; i++)
      {
        if (comp->type == PLE_ACTUAL)
        {
         /*Summe der Typen berechnen*/
          sum = sum + pbp_type(comp->comp.actual);
        }else{
          formal_counter++;
        }
        comp = comp->next;
      }
      /* arithmetisches Mittel bilden */
      tmpmin = (sum/template->arity);
      tmpmax = (sum + (formal_counter * 8))/template->arity;
    }
     /* untere Grenze setzen */
    if ((minbound == -1) || (minbound > tmpmin))
    {
      minbound = tmpmin;
    }
     /* obere Grenze setzen */
    if ((maxbound == -1) || (maxbound < tmpmax))
    {
      maxbound = tmpmax;
    }
    template = template->next;
  }
  z = 0;
  for (i=minbound;(i<=maxbound && z < ts->max_partitions); i++)
  {
     /* gefundenen Tids uebergeben */
    partition_ids[z] = ts->tids[(i%ts->max_partitions)];
    z++;
  }
  partition_ids[z] = 0;    /* nur sicherheitshalber */
}
```

### pli_hash_temp_one

⟨*Interne O-Operationen*⟩+≡

```
extern void pli_hash_temp_one(Ple_Template *temp,
                              Pli_Tuple_Space *ts,
                              int *partition_ids);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist `partition_ids`, das Feld der Partitiontids. |
| `temp` | ist die Liste der Templates. |
| `ts` | ist der Tupelraum in dem die zu hashenden Partitionen liegen. |

**Beschreibung:** Hier wird nur über die Länge der Templates gehasht. Doppelt gehashte Tids werden eliminiert.

**Ausnahmen:** Keine.

**Abhängigkeiten:** Keine.

⟨*Implementierung der Operationen*⟩+≡

```
void pli_hash_temp_one(Ple_Template *temp, Pli_Tuple_Space *ts,
                         int *partition_ids)
{
 Pbp_Bool_Type found;
 int newtid;
 int i,z;              /* Indizes */
 Ple_Template *template;

  template = temp;
  z = 0;
  found = PBP_FALSE;

   while(template != NULL)
   {
    if (template->arity == 0)
    {
      newtid = ts->tids[0];
    }else{
       newtid = ts->tids[(template->arity%ts->max_partitions)];
    }
    for(i=0; i<z; i++)
    {
      if (newtid == partition_ids[i])
      {
        found = PBP_TRUE;
        break;
```

```
      }
    }
    if (found == PBP_FALSE)
    {
      partition_ids[z] = newtid;
      z++;
    }else{
      found = PBP_FALSE;
    }
    template = template->next;
  }
  partition_ids[z] = 0;    /* nur sicherheitshalber */
}
```

**pli_hash_temp_two**

⟨*Interne O-Operationen*⟩+≡

```
extern void pli_hash_temp_two(Ple_Template *temp,
                              Pli_Tuple_Space *ts,
                               int *partition_ids);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist `partition_ids`, das Feld der Partitiontids. |
| `temp` | ist die Liste der Templates. |
| `ts` | ist der Tupelraum in dem die zu hashenden Partitionen liegen. |

**Beschreibung:** Es wird angenommen, daß das erste Element eines jeden Templates in `temp` vom Typ Integer ist. Dieser Wert ist gleich dem Hashwert.

**Ausnahmen:** Erstes Element ist kein Actual und nicht vom Typ `PBP_INTEGER`.

**Abhängigkeiten:** `pbp_type`

⟨*Implementierung der Operationen*⟩+≡

```
void pli_hash_temp_two(Ple_Template *temp, Pli_Tuple_Space *ts,
                          int *partition_ids)
{
 Pbp_Bool_Type found;
 int newtid;
 int i,z;              /* Indizes */
 Ple_Template *template;
 Ple_Temp_Comp *comp;
 Pbp_Object *object;

  template = temp;
  z = 0;
  found = PBP_FALSE;

   while(template != NULL)
   {
    if (template->arity == 0)
      newtid = ts->tids[0];
    comp = template->comps;

    if (comp->type == PLE_FORMAL)
      plp_error("pli_hash_temp_two",
"Wrong Hashfunction! First element of template is a formal.");

    if (pbp_type(comp->comp.actual) != PBP_INTEGER)
      plp_error("pli_hash_temp_two",
```

```
"Wrong Hashfunction! First element of template is not of type PBP_INTEGER.");

     object = comp->comp.actual;
     newtid = ts->tids[(object->pbp_integer)%ts->max_partitions];

     for(i=0; i<z; i++)
     {
       if (newtid == partition_ids[i])
       {
         found = PBP_TRUE;
         break;
       }
     }
     if (found == PBP_FALSE)
     {
       partition_ids[z] = newtid;
       z++;
     }else{
       found = PBP_FALSE;
     }
     template = template->next;
   }
   partition_ids[z] = 0;    /* nur sicherheitshalber */
 }
```

### pli_read_file

⟨*Interne O-Operationen*⟩+≡

```
extern void pli_read_file(char *filename);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | Keinen. |
| `filename` | Name der Konfigurationsdatei. |

**Beschreibung:** Ließt das Konfigurationsfile aus. Die externen Variablen `plp_loadbalancing`, `plp_eval`, `plp_max_partitions` werden gesetzt. Während der Performacemessungen war die Funktion auskommentiert.

**Ausnahmen:** Wird eine Option nicht gesetzt, so wird eine Warnung auf stdout ausgegeben.

**Abhängigkeiten:** `pvm_perror`, `lp_loadbalancing`, `plp_eval`, `plp_max_partitions`

⟨*Implementierung der Operationen*⟩+≡

```
void pli_read_file(char *filename)
{
 extern Pbp_Bool_Type plp_loadbalancing;
 extern Pbp_Bool_Type plp_eval;
 extern int plp_max_partitions;
 char *string;
 FILE *f;
 int info;
 int getit[4];                  /* Felder fuer Flags */

plp_loadbalancing = PBP_TRUE;
plp_max_partitions = 4;
plp_eval = PBP_FALSE;
plp_hash_mode = 0;

/*
  if ((string = (char *)malloc(sizeof (char)*30)) == NULL)
    plp_error("plp_read_file (malloc)",
               "Cannot allocate memory!");

    f = fopen(filename, "r");
  if (f == NULL)
  {
    pvm_perror("plp_read_file (fopen)
     Cannot find config-file! Use standard configuration.");
    return;
```

```
  }
  while( (info=fscanf(f, "%s ", string)) != 0  && info != EOF )
  {
    if (strcmp(string,"--") == 0)
    {
      fscanf(f, "\n");
    }else{
      if (strcmp(string,"partition") == 0)
      {
        fscanf(f, " %d\n", &(plp_max_partitions));
        getit[0] = 1;
      }else{
        if (getit[0] != 1)
  pvm_perror("plp_read_file:No parameter _partition_!");

        if (strcmp(string,"hash_mode") == 0)
        {
          fscanf(f, " %d\n", &(plp_hash_mode));
          getit[1] = 1;
        }else{
          if (getit[1] != 1)
  pvm_perror("plp_read_file: No parameter _hash_mode_!");

          if (strcmp(string,"loadbalancing") == 0)
          {
            fscanf(f, "\n");
            plp_loadbalancing = PBP_TRUE;
            getit[2] = 1;
          }else{
            if (getit[2] != 1)
 pvm_perror("plp_read_file: No parameter _loadbalancing_!");

            if (strcmp(string,"evalserver") == 0)
            {
              fscanf(f, "\n");
              plp_eval = PBP_TRUE;
              getit[3] = 1;
            }else{
```

```
pvm_perror("plp_read_file:Wrong parameter in config-file!!");
            }
          }
        }
      }
    }
  }
  fclose(f);
  free(string);
*/
}
```

### pli_scan

⟨*Exportierte O-Operationen*⟩+≡

```
extern Pbp_Object *pli_scan(char **strp);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist ein Objekt, der den String repräsentiert. |
| `strp` | ist der String, der in ein Objekt umgewandelt wird. |
| `gps_return_status` | |
| `gps_return_address` | enthält die dynamischen Vorgänger. |

**Beschreibung:** `pli_scan` erzeugt ein Objekt aus einem String, der das Objekt repräsentiert. (Die Funktion wurde von Heiner Pohland geschrieben.)

**Abhängigkeiten:** `plo_scanOm`, `plo_scanBoolean`, `plo_scanNumber`, `plo_scanString`, `plo_scanTuple`, `plo_scanSet`, `plo_scanAtom`, `plo_scanProcess`, `pvm_perror`, `plp_error`

⟨*Implementierung der Operationen*⟩+≡

```
Pbp_Object *pli_scan(char **strp)
{
  Pbp_Object *retVal;
  pbh_decl_var;
  Pbp_Exit *gps_return_status = NULL;
  Pbp_Env *gps_return_address = NULL;

  #ifdef NETDEBUG
    pvm_perror("pli_scan: Here I am");
    pvm_perror(*strp);
  #endif

  plo_scipBlancs(strp);
  switch (**strp)
  {
    case '#':
      (*strp)++;
      switch(**strp)
      {
        case 'o':
          retVal = plo_scanOm(strp);
          break;

        case 't':
        case 'f':
```

```
      retVal = plo_scanBoolean(strp);
      break;

    default:
      plp_error("pli_scan",
       "No object in string!(Om and Boolean)");
  }
  break;

case '+':
case '-':
case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
#ifdef NETDEBUG
  pvm_perror("scan: INTEGER");
#endif
  retVal = plo_scanNumber(strp);
  break;

case '\"':
#ifdef NETDEBUG
  pvm_perror("scan: STRING");
#endif
  retVal = plo_scanString(strp);
  break;

case '[':   /* ] (mein Editor spielt sonst verrueckt) */
#ifdef NETDEBUG
  pvm_perror("scan: TUPLE");
#endif
```

```
      retVal = plo_scanTuple(strp, pbp_act_par);
    break;

    case '{':   /* } (mein Editor spielt sonst verrueckt) */
    #ifdef NETDEBUG
      pvm_perror("scan: SET");
    #endif
      retVal = plo_scanSet(strp, pbp_act_par);
      break;

    case 'a':   /* Atome */
    #ifdef NETDEBUG
      pvm_perror("scan: ATOM");
    #endif
      retVal = plo_scanAtom(strp);
      break;
   case '!':  /* Processe */
    #ifdef NETDEBUG
      pvm_perror("scan: PROCESS");
    #endif
    retVal = plo_scanProcess(strp);
    break;
    default:
    pvm_perror(*strp);
      plp_error("pli_scan" ,"No object in string!");
  }
  return (retVal);
}
```

**pli_ots**

⟨*Exportierte O-Operationen*⟩+≡

```
extern char *pli_ots (Pbp_Object *object,
                Pbp_Env *gps_return_address);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist ein string, der das Objekt repräsentiert. |
| `obj` | ist das Objekt, das in einen string umgewandelt wird. |
| `gps_return_address` | enthält die dynamischen Vorgänger. |

**Beschreibung:** `pli_ots` erzeugt einen C-string, der das Objekt repräsentiert. (Die Funktion wurde von Heiner Pohland geschrieben.)

**Abhängigkeiten:** `plp_error`, `pvm_perror`

⟨*Implementierung der Operationen*⟩+≡

```
char *pli_ots(Pbp_Object *object, Pbp_Env *gps_return_address)
{
 char *objstr, *retstr, *buffer;
 int size, i;
 PBP_TUPLE_ELEM *element;
 BUCKET *setTable;
 LIST *setList;


  buffer = (char *)calloc(80, sizeof(char));
  if (buffer == NULL)
      plp_error("pli_ots (calloc)", "Not enough memory!");

  switch(pbp_type(object)){
  case PBP_OM:
    retstr = (char *)calloc(4, sizeof(char));
      if (retstr == NULL)
      plp_error("pli_ots", "Not enough memory!");

      sprintf(retstr, "%s", "#om" );
      break;
  case PBP_BOOLEAN:
    #ifdef NETDEBUG
      pvm_perror("ots: BOOLEAN");
    #endif
      if(object->pbp_boolean == PBP_TRUE)
      {
      retstr = (char *)calloc(6, sizeof(char));
```

```
    if (retstr == NULL)
      plp_error("pli_ots (calloc)", "Not enough memory!");

    sprintf(retstr, "%s", "#true");
    }else{
    retstr = (char *)calloc(7, sizeof(char));
    if (retstr == NULL)
      plp_error("pli_ots (calloc)", "Not enough memory!");
    sprintf(retstr, "%s", "#false");
    }
    break;
case PBP_INTEGER:
  #ifdef NETDEBUG
    pvm_perror("ots: INTEGER");
  #endif
    sprintf(buffer, "%d", object->pbp_integer);
    retstr = (char *)calloc(strlen(buffer) + 1, sizeof(char));
    if (retstr == NULL)
    plp_error("pli_ots (calloc)", "Not enough memory!");

    sprintf(retstr, "%s", buffer);
    break;

case PBP_REAL:
  #ifdef NETDEBUG
    pvm_perror("ots: REAL");
  #endif
    sprintf(buffer, "%g", object->pbp_real);
    retstr = (char *)calloc(strlen(buffer) + 1, sizeof(char));
    if (retstr == NULL)
    plp_error("pli_ots (calloc)", "Not enough memory!");

    sprintf(retstr, "%s", buffer);
    break;

case PBP_STRING:
  #ifdef NETDEBUG
    pvm_perror("ots: STRING");
```

```
  #endif
    retstr = (char *)calloc(strlen(object->pbp_string_string)+3,
                            sizeof(char));
    if (retstr == NULL)
    plp_error("pli_ots", "Not enough memory!");

    sprintf(retstr, "\"%s\"", object->pbp_string_string);
    break;

case PBP_TUPLE:
  #ifdef NETDEBUG
    pvm_perror("ots: TUPLE");
  #endif
    size = object->pbp_comp_number;
    if(size == 0) return("[]");

    retstr = (char *)calloc(2, sizeof(char));
    if (retstr == NULL)
    plp_error("pli_ots", "Not enough memory!");

    sprintf(retstr, "[");
    element =
  ((PBP_TUPLE_HEADER *)object->pbp_comp_header)->firstTElement;

    for(i = 1; i < size; i++)
    {
    objstr=pli_ots(element->TupleObject, gps_return_address);
    retstr = (char *)realloc(retstr,
            strlen(retstr) + strlen(objstr) + 2);
    if (retstr == NULL)
    {
      plp_error("pli_ots(realloc)", "Not enough memory!");
    }
    retstr = strcat(retstr, objstr);
    retstr = strcat(retstr, ",");
    element = element->nextElement;
    }
    objstr = pli_ots(element->TupleObject, gps_return_address);
```

```
      retstr = (char *)realloc(retstr,
                strlen(retstr) + strlen(objstr) + 2);
      if (retstr == NULL)
      plp_error("pli_ots (realloc)", "Not enough memory!");

      retstr = strcat(retstr, objstr);
      retstr = strcat(retstr, "]");
      break;

  case PBP_SET:
    #ifdef NETDEBUG
      pvm_perror("ots: SET");
    #endif
      if(object->pbp_comp_number == 0) return("{}");

      setTable = ((SHEAD *)(object->pbp_comp_header))->hashtable;
      retstr = (char *)calloc(2, sizeof(char));
      if (retstr == NULL)
      plp_error("pli_ots (calloc)", "Not enough memory!");

      sprintf(retstr, "{");
      for (i = 0; i < PRIME; i++)
      {
      if (setTable[i].number != 0)
      {
        setList = setTable[i].begin;
        while (setList != NULL)
        {
          objstr = pli_ots(setList->element,
                    gps_return_address);
          retstr = (char *)realloc(retstr,
                        strlen(retstr) + strlen(objstr) + 2);
          if (retstr == NULL)
            plp_error("pli_ots (realloc)",
                    "Not enough memory!");

          retstr = strcat(retstr, objstr);
          retstr = strcat(retstr, ",");
```

```
          setList = setList->next;
        }
      }
    }
    #ifdef NETDEBUG
      printf("retstr: %s\n", retstr);
    #endif
      retstr[strlen(retstr) - 1] = '}';
/*      retstr[strlen(retstr)] = '\0';*/
      break;

  case PBP_ATOM:
    #ifdef NETDEBUG
      pvm_perror("ots: ATOM");
    #endif
      sprintf(buffer, "a%d %d %d %d", object->pbp_atom_hostid,
              object->pbp_atom_pid, object->pbp_atom_counter,
              object->pbp_atom_time);
      retstr = (char *)calloc(strlen(buffer) + 1, sizeof(char));
      if (retstr == NULL)
      plp_error("pli_ots (calloc)", "Not enough memory!");

      sprintf(retstr, "%s", buffer);
    break;

  case PBP_FUNCTION:
      retstr = (char *)calloc(10, sizeof(char));
      if (retstr == NULL)
      plp_error("pli_ots (calloc)", "Not enough memory!");

      sprintf(retstr, "%s", "function" );
    break;

  case PBP_MODTYPE:
      retstr = (char *)calloc(7, sizeof(char));
      if (retstr == NULL)
      plp_error("pli_ots (calloc)", "Not enough memory!");
```

```
        sprintf(retstr, "%s", "module" );
      break;

    case PBP_INSTANCE:
        retstr = (char *)calloc(9, sizeof(char));
        if (retstr == NULL)
        plp_error("pli_ots (calloc)", "Not enough memory!");

        sprintf(retstr, "%s", "instance" );
      break;

    case PBP_PROCESS:
        sprintf(buffer, "!%d ", object->pbp_proc_id);
        retstr = (char *)calloc(strlen(buffer) + 1, sizeof(char));
        if (retstr == NULL)
        plp_error("pli_ots (calloc)", "Not enough memory!");

        sprintf(retstr, "%s", buffer);
        break;

    case PBP_PROCEDURE:
    case PBP_HANDLER:
    case PBP_MODULE:
    default:
      plp_error("pli_ots", "Illegal object!");
    }
    free(buffer);
    return (retstr);
  }
```

### C.2.5 Lokale Operationen

**plo_scipBlancs**

**Profil:**
`Rückgabewert`
`string`

**Beschreibung:** Es werden die nächsten Blancs aus dem String entfernt.

**Abhängigkeiten:** Keine.

**Benutzung in:** `pli_scan`

⟨*Lokale Operationen*⟩≡

```
static void plo_scipBlancs(char **string)
{
  for(; **string == ' '; (*string)++);
}
```

### plo_scanOm

**Profil:**
```
Rückgabewert
string
```

**Beschreibung:** Es wird der String für Om in ein Objekt umgewandelt.

**Abhängigkeiten:** `pbo_create`, `plp_error`

**Benutzung in:** `pli_scan`

⟨*Lokale Operationen*⟩+≡
```
static Pbp_Object *plo_scanOm(char **strp)
{
 Pbp_Object *retObj;
 pbh_decl_var;


   (*strp)++;
   if (**strp == 'm')
   {
     (*strp)++;
     return (pbo_create(PBP_OM));
   }else{
     plp_error("plo_scanOm", "No object in string!");
   }
}
```

**plo_scanBoolean**

**Profil:**
`Rückgabewert`
`string`

**Beschreibung:** Es wird der String für ein Boolean in ein Objekt umgewandelt.

**Abhängigkeiten:** `plp_error`

**Benutzung in:** `pli_scan`

⟨*Lokale Operationen*⟩+≡

```
static Pbp_Object *plo_scanBoolean(char **strp)
{
 Pbp_Object *retObj;
 pbh_decl_var;


  if(((*strp)[0] == 't') &&
    ((*strp)[1] == 'r') &&
    ((*strp)[2] == 'u') &&
    ((*strp)[3] == 'e'))
  {
    *strp = *strp + 4;
    retObj = pbp_true;
  }else if (((*strp)[0] == 'f') &&
        ((*strp)[1] == 'a') &&
        ((*strp)[2] == 'l') &&
        ((*strp)[3] == 's') &&
        ((*strp)[4] == 'e'))
    {
    *strp = *strp + 5;
    retObj = pbp_false;
  }else{
    plp_error("plo_scanBoolean", "No object in string!");
  }
  return (retObj);
}
```

### plo_scanAtom

**Profil:**
`Rückgabewert`
`string`

**Beschreibung:** Es wird der String in ein Atom umgewandelt. (Diese Funktion wurde `nicht` von Heiner Pohland geschrieben.:-))

**Abhängigkeiten:** `pbo_create`, `plo_scipBlancs`, `pbp_set_use`

**Benutzung in:** `pli_scan`

⟨*Lokale Operationen*⟩+≡

```
static Pbp_Object *plo_scanAtom(char **strp)
{
 int z;
 Pbp_Object *retobj;
 int intNum,negSign;

  retobj = pbo_create(PBP_ATOM);


  if ((*strp)[0] != 'a')
    plp_error("plo_scanAtom", "Wrong character");

  (*strp)++;

   for (z = 0; z < 4; z++)
  {
    negSign = 0;
    intNum = 0;
    if ((*strp)[0] == '+')
    {
        (*strp)++;
      }
      else if ((*strp)[0] == '-')
      {
        negSign = 1;
        (*strp)++;
```

```
      }

    while(isdigit((*strp)[0]))
    {
      intNum = 10*intNum + (*strp)[0] - '0';
      (*strp)++;
    }
    if (negSign == 1)
    {
        intNum = -intNum;
      }

    switch (z){
    case 0 :
      retobj->pbp_atom_hostid = intNum;
      plo_scipBlancs(strp);       /* Blank ueberspringen */
      break;
    case 1 :
      retobj->pbp_atom_pid =  intNum;
      plo_scipBlancs(strp);       /* Blank ueberspringen */
       break;
    case 2 :
      retobj->pbp_atom_counter = intNum;
      plo_scipBlancs(strp);       /* Blank ueberspringen */
      break;
    case 3 :
      retobj->pbp_atom_time = intNum;
      break;
    }
  }
  pbp_set_use(retobj);
  return(retobj);
}
```

### plo_scanNumber

**Profil:**
`Rückgabewert`
`string`

**Beschreibung:** Es wird der String für eine Zahl in ein Integer- oder Real-Objekt umgewandelt. (Diese Funktion wurde von Heiner Pohland geschrieben.)

**Abhängigkeiten:** `pbp_new_integer`, `pbp_new_real`

**Benutzung in:** `pli_scan`

⟨*Lokale Operationen*⟩+≡
```
static Pbp_Object *plo_scanNumber(char **strp)
{
 char *buffer;
 register i = 0, j;
 int intNum = 0, negSign = 0;
 long int longNum;
 double realNum;
 Pbp_Object *retObj;
 pbh_decl_var;

  if ((*strp)[0] == '+')
  {
    (*strp)++;
  }else if ((*strp)[0] == '-')
  {
    negSign = 1;
    (*strp)++;
  }

  if (!(isdigit(**strp)))
    plp_error("plo_scanNumber", "No object in string!");

  for (i = 0; isdigit((*strp)[i]); i++);

  if ((*strp)[i] != '.')
  {
```

```
    buffer = (char *)pbn_calloc(i + 1, sizeof(char));
    if (buffer == NULL)
      plp_error("plo_scanNumber (calloc)",
                "Can't allocate memory!");

    buffer = strncpy(buffer, (*strp), i);

    for (j = 0; j < i; j++)
    {
      intNum = 10*intNum + buffer[j] - '0';
    }

    if (negSign == 1) intNum = -intNum;

    *strp = (*strp) + i;
    free(buffer);
    return (pbp_new_integer(intNum));
  }
  i++;
  if(!(isdigit((*strp)[i])))
    plp_error("plo_scanNumber (isdigit)",
              "No object in string!");

  for (i++; isdigit((*strp)[i]); i++);

  if(((*strp)[i] != 'e') && ((*strp)[i] != 'E'))
  {
    buffer = (char *)calloc(i + 1, sizeof(char));
    if (buffer == NULL)
      plp_error("plo_scanNumber (calloc)",
                  "Not enough memory!");

    buffer = strncpy(buffer, (*strp), i + i);
    realNum = atof(buffer);

    if (negSign) realNum = -realNum;

    *strp = (*strp) + i;
```

```
    free(buffer);
    return (pbp_new_real(realNum));
  }
  i++;
  if (((*strp)[i] == '+') || ((*strp)[i] == '-')) i++;

  i++;
  if(!(isdigit((*strp)[i])))
    plp_error("plo_scanNumber (isdigit)",
                         "No object in string!");

  for (i++; isdigit((*strp)[i]); i++);

  buffer = (char *)calloc(i + 1, sizeof(char));
  if (buffer == NULL)
    plp_error("plo_scanNumber (calloc)", "Not enough memory!");

  buffer = strncpy(buffer, (*strp), i);
  realNum = atof(buffer);

  if (negSign) realNum = -realNum;

  *strp = (*strp) + i;
  free(buffer);
  return (pbp_new_real(realNum));
}
```

## plo_scanString

**Profil:**
```
Rückgabewert
string
```

**Beschreibung:** Es wird der String für den Typ String in ein Objekt umgewandelt. (Diese Funktion wurde von Heiner Pohland geschrieben.)

**Abhängigkeiten:** `plp_error`, `pbp_new_string`

**Benutzung in:** `pli_scan`

⟨*Lokale Operationen*⟩+≡
```
static Pbp_Object *plo_scanString(char **strp)
{
 register i;
 char *buffer;
 Pbp_Object *retObj;
 pbh_decl_var;


  (*strp)++;
  if (**strp =='\0')
    plp_error("plo_scanString", "No object in string!");

  for (i = 0; (*strp)[i] != '\"'; i++)
  {
    if ((*strp)[i] == '\0')
      plp_error("plo_scanString", "No object in string!");
  }
  buffer = (char*)calloc(i + 1, sizeof(char));
  if (buffer == NULL)
      plp_error("plo_scanString (calloc)", "Not enough memory!");

  buffer = strncpy(buffer, (*strp), i);
  *strp = (*strp) + i +1;
  retObj = pbp_new_string(buffer);
  free(buffer);
  return (retObj);
```

```
}
```

### plo_scanTuple

**Profil:**
    Rückgabewert
    string

**Beschreibung:** Es wird der String für den Typ Tuple in ein Objekt umgewandelt.

**Probleme:** noch keine

**Abhängigkeiten:** pbo_create, pbp_set_use, plo_scipBlancs, pli_scan, pbo_dest_with, pbp_set_unuse, pvm_perror, plp_error

**Benutzung in:** pli_scan

⟨*Lokale Operationen*⟩+≡
```
static Pbp_Object *plo_scanTuple(char **strp, pbp_form_par)
{
 Pbp_Object *retObj, *tupElem, *excObj;
 pbh_decl_var;


  retObj = pbo_create(PBP_TUPLE);
  pbp_set_use(retObj);

  (*strp)++;
  if (**strp ==']')
  {
    (*strp)++;
    return (retObj);
  }
  while (1)
  {
    plo_scipBlancs(strp);
    tupElem = pli_scan(strp);
    excObj = pbo_dest_with(retObj, tupElem, pbp_act_par);
    plo_scipBlancs(strp);

    if (**strp == ',')
    {
```

```
      (*strp)++;
    }else if (**strp == ']')
    {
      (*strp)++;
      pbp_set_unuse(retObj);
      return (retObj);
    }else{
      pvm_perror(*strp);
      plp_error("plo_scanTuple", "No object in string!");
    }
  }
}
```

## plo_scanSet

**Profil:**
`Rückgabewert`
`string`

**Beschreibung:** Es wird der String für den Typ Set in ein Objekt umgewandelt. (Diese Funktion wurde von Heiner Pohland geschrieben.)

**Abhängigkeiten:** `pbo_create`, `pbp_set_use`, `plo_scipBlancs`, `pli_scan`, `pbo_dest_with`, `pbp_set_unuse`, `plp_error`

**Benutzung in:** `pli_scan`

⟨*Lokale Operationen*⟩+≡

```
static Pbp_Object *plo_scanSet(char **strp, pbp_form_par)
{
 Pbp_Object *retObj, *setElem, *excObj;
 pbh_decl_var;


  retObj = pbo_create(PBP_SET);
  pbp_set_use(retObj);

  (*strp)++;
  if (**strp ==’}’)
  {
    (*strp)++;
    return (retObj);
  }
  while (1)
  {
    plo_scipBlancs(strp);
    setElem = pli_scan(strp);
    excObj = pbo_dest_with(retObj, setElem, pbp_act_par);
    plo_scipBlancs(strp);
    if (**strp == ’,’)
    {
      (*strp)++;
```

```
    }else if (**strp == '}')
    {
      (*strp)++;
      pbp_set_unuse(retObj);
      return (retObj);
    }else{
      plp_error("plo_scanSet", "No object in string!");
    }
  }
}
```

**plo_scanProcess**

**Profil:**
    Rückgabewert
    string

**Beschreibung:** string wird in ein Objekt vom Typ PLI_PROCESS umgewandelt. (Diese Funktion wurde nicht von Heiner Pohland geschrieben. :-))

**Probleme:** noch keine

**Abhängigkeiten:** pvm_perror, pbo_create

**Benutzung in:** pli_scan

⟨*Lokale Operationen*⟩+≡

```
static Pbp_Object *plo_scanProcess(char **strp)
{
 Pbp_Object *retObj = pbo_create(PBP_PROCESS);
 char *buffer;
 register i = 0, j;
 int intNum = 0;
 long int longNum;
 pbh_decl_var;

  #ifdef NETDEBUG
    pvm_perror(" plo_scanProcess: begin");
    pvm_perror(*strp);
  #endif

  if ((*strp)[0] != '!')
    {
      plp_error("plo_scanProcess", "wrong char");
  }

  (*strp)++;

  if (!(isdigit(**strp)))
    plp_error("plo_scanProcess", "No object in string!");
```

```
    /* Wieviel Stellen hat die Zahl? */
  for (i = 0; isdigit((*strp)[i]); i++);


  buffer = (char*)calloc(i + 1, sizeof(char));
  if (buffer == NULL)
      plp_error("plo_scanString", "Not enough memory!");

  buffer = strncpy(buffer, (*strp), i);

  for (j = 0; j < i; j++)
  {
    intNum = 10*intNum + buffer[j] - '0';
  }

  retObj->pbp_proc_id = intNum;

  *strp = (*strp) + i;
  free(buffer);
  #ifdef NETDEBUG
    pvm_perror("scanProcess: leave");
  #endif
  return (retObj);
}
```

## C.2.6 Interne Operationen

### plo_match_list

⟨*Interne O-Operationen*⟩+≡

```
extern int plo_match_list(Pbp_Object **tup, Ple_Template *temps,
                          Pli_Op_Type *optype, Pbp_Object *tsid,
                   int threadid);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die Nummer des ausgewählten Templates. |
| `tup` | ist das zu vergleichende Tupel. Bei `PLI_MEETINTO` wird hier das neue Tupel zurückgegeben. |
| `temps` | ist die Liste der Templates, mit der `tup` verglichen wird. |
| `optype` | gibt an, ob es sich um ein `meet` oder ein `fetch` statement handelt. Falls `tup` verändert wurde, wird `PLI_MEETINTO` zurückgegeben. |
| `tsid` | ist die Tupelraumidentität. Sie wird für die Trace-Informationen benoetigt. |
| `threadid` | ist die Identität des Threads, dem das Template gehoert. |

**Beschreibung:** `plo_match_list` prüft, ob das Tupel `tup` mit einem Template der Liste `temps` matcht. Trifft dies zu, wird den Formals des Templates die entsprechenden Werte zugewiesen, bei einer meet-Anweisung mit into statement wird `tup` verändert, und die Nummer des ausgewählten Templates zurückgegeben. Wurde kein Template ausgewählt, wird 0 zurückgegeben. (Ist schon Teil des alten Systems gewesen.)

**Abhängigkeiten:** `plo_matching`

⟨*Implementierung der Operationen*⟩+≡

```
int plo_match_list(Pbp_Object **tup, Ple_Template *temps,
       Pli_Op_Type *optype, Pbp_Object *tsid, int threadid)
{
  int tempNum = 0;  /* number of the actual template */

  /* going through the list */
  while(temps != NULL)
  {
    tempNum++;
    if(plo_matching(tup, temps, optype, tsid, tempNum))
                        /* checking the actual template */
    {
      return(tempNum);   /* returning the number of the
                        matching template */
    }
    temps = temps->next;
  }
```

```
  return(0);              /* no matching template found */
}
```

### plo_hash

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist Nummer des gesuchten Hashbuckets. |
| `tupleNumber` | ist die Länge des zu verhashenden Tupels. |

**Beschreibung:** Es wird nur über die Länge des Tupels verhasht, da dies die einzige Information eines Templates ist, die immer vorhanden ist. `Rückgabewert` ist die Nummer des zugehörigen Hashbuckets.

**Abhängigkeiten:** `PLI_PRIME`, die Größe der Hashtabelle muß bekannt sein.

**Benutzung in:** `plo_add_tuple`, `plo_match_in_ts`

⟨*Lokale Operationen*⟩+≡

```
static unsigned plo_hash(unsigned tupleNumber)
{
  return(tupleNumber % PLI_PRIME);  /* very simple */
}
```

### plo_add_tuple

⟨*Interne O-Operationen*⟩+≡

```
extern void plo_add_tuple(Pbp_Object *tup, Pli_Tuple_Space *ts);
```

**Profil:**

| | |
|---|---|
| `tup` | ist das Tupel, das in `ts` eingefügt werden soll. |
| `ts` | ist der Tupelraum, in den `tup` eingefügt werden soll. |

**Beschreibung:** `plo_add_tuple` fügt `tup` in `ts` ein. Dabei wird überprüft, ob ein Prozeß bereits auf ein solches Tupel wartet.

**Ausnahmen:**

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:** `plo_hash`, `pbo_copy`, `plq_check_meets`, `plq_check_fetch`

⟨*Implementierung der Operationen*⟩+≡

```
void plo_add_tuple(Pbp_Object *tup, Pli_Tuple_Space *ts)
{
 unsigned hashNum;           /* number of the actual hash list */
 Pli_Hash_List *newList;   /* the actual member of a hash list */

  /* Check for pending, matching meet templates: */
  while(plq_check_meets (&tup, ts) == PLI_MEETINTO)
  {
    /* now we must check the changed tuple */

#ifdef Linda_Debug
    printf ("deposit: found pending meet-into template\n");
#endif
  }
  #ifdef NETDEBUG
    pvm_perror("add_tuple: nach check_meets");
  #endif
    /* no matching template of a fetch statement was found */
    /* putting tup into ts */
      /* finding the hash list */
    hashNum = plo_hash(tup->pbp_comp_number);

    /* creating a new list element */
    newList = (Pli_Hash_List*)malloc(sizeof(Pli_Hash_List));
    if(newList == NULL)
      plp_error("plo_add_tuple (malloc)", "Not enough memory");
```

```
    #ifdef NETDEBUG
      pvm_perror("add_tuple: vor set_use");
    #endif
    pbp_set_use(tup);
    newList->tuple = tup;

    /* putting the new element into the hash list */
    #ifdef NETDEBUG
      pvm_perror("add_tuple: putting in hash_list");
    #endif
    newList->next = ts->hashtable[hashNum];
    ts->hashtable[hashNum] = newList;

    ts->card++;                  /* increasing the number of ts */
  #ifdef NETDEBUG
    pvm_perror("leaving plo_add_tuple");
  #endif
}       /* end plo_add_tuple */
```

**plo_find_ts**

⟨*Interne O-Operationen*⟩+≡

```
extern Pli_Tuple_Space *plo_find_ts(Pbp_Object *tsid,
                          Pli_FindTs_Type opt);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist der Tupelraum mit der Identität `tsid`. |
| `tsid` | ist die Identität des gesuchten Tupelraumes. |

**Beschreibung:** `plo_find_ts` sucht den zu `tsid` gehörigen Tupelraum. Falls dieser nicht vorhanden sein sollte, wird `NULL` zurückgegeben. Andernfalls wird der Tupelraum zurückgegeben.

**Ausnahmen:** keine

**Probleme:** noch nicht

**Abhängigkeiten:** `pbo_equal`

⟨*Implementierung der Operationen*⟩+≡

```
Pli_Tuple_Space *plo_find_ts(Pbp_Object *tsid, Pli_FindTs_Type opt)
{
 char *string;
 Pli_Tuple_Space *search = NULL;       /* the actual tuple space */
 Pli_Tuple_Space *prev;         /* der Vorgaenger in der Liste */
 Pli_Tuple_Space *newTs;        /* neues Listenelement */
 int counter = 0;
 int length;
 int tids[20];
 int size;
 int i;                         /* Index */
 int info;
 char flag;
 float quot;
 Pbp_Object *atom;

  #ifdef NETDEBUG
    pvm_perror("find_ts: begin");
  #endif
  search = allTupleSpaces;       /* searching from the beginning
                                of the list */
  prev = NULL;
  while(search != NULL)
  {
    counter++;
```

```
    if(pbp_true == pbo_equal(search->atom, tsid))
                 /* checking the identity */
    {
      if (opt == PLI_FIND) return(search);
                         /* we have found it */
      if (opt == PLI_CLEAR) break;
      if (prev == NULL)
      {
        allTupleSpaces = search->next;
      }else{
        prev->next = search->next;
      }
      break;
    }
    prev = search;
    search = search->next;       /* shifting search */
  }
  if (plp_eval == PBP_TRUE)
  {
    size = pvm_gsize("ts");
    if (size < 1) return(NULL);
    quot = (float)size/(float)plp_max_partitions;
  }
  if (quot == counter || plp_eval == PBP_FALSE)
  {
     /* mein Wissen ist korrekt,
               brauche eval-server nicht fragen*/
    if (opt == PLI_FIND) return(search);
  }else{
    if (opt == PLI_FIND && plp_eval == PBP_TRUE)
    {
      /* eval-server fragen */
      info = pvm_initsend(plp_encoding);
      #ifdef NETDEBUG
        if (info < 0)
        plp_error("plo_find_ts (pvm_initsend)",
                     "Can't allocate memory!");
      #endif
```

```
info = pvm_pkint(&plp_main_id , 1, 1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plo_find_ts (pvm_pkint)",
              "Can't allocate memory!");
#endif

  info = pvm_pkstr("s");
#ifdef NETDEBUG
  if (info < 0)
  plp_error("plo_find_ts (pvm_pkstr)",
              "Can't allocate memory!");
#endif

plp_pack_obj(tsid);

info = pvm_send(plp_eval_tid, 5);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plo_find_ts (pvm_send)",
                 "Error in pvm_send!");
#endif

info = pvm_recv(plp_eval_tid, 5);
     /* Receive message from eval-server  */
switch (info){
case PvmBadParam :
  plp_error("plo_find_ts",
          "Wrong parameters in pvm_recv");
  break;
case PvmSysErr :
  plp_error("plo_find_ts", "Pvmd is not responding");
  break;
}
info = pvm_upkint(tids, 20, 1);
#ifdef NETDEBUG
if (info < 0)
```

```
          plp_error("plp_find_ts (pvm_upkint)",
                    "Can't allocate memory!");
        #endif

        if (tids[0] == 0) return(NULL);
             /* Tupelraum nicht gefunden */

        /* creating a new tuple space */
        newTs =
        (Pli_Tuple_Space *)malloc(sizeof(Pli_Tuple_Space));
        if(newTs == NULL)
          plp_error("plo_find_ts (malloc)", "not enough memory");

        newTs->atom = tsid;
        pbp_set_use(newTs->atom);
        newTs->atom_str = NULL;

        newTs->max_partitions = plp_max_partitions;
        newTs->hash_mode = plp_hash_mode;

        /* save the partition ids */
        for (i=0; i < newTs->max_partitions; i++)
          newTs->tids[i]  = tids[i];

          /* putting newTs in the list of all tuple spaces */
        newTs->next = allTupleSpaces;
        allTupleSpaces = newTs;
        return(newTs);
      }
    }

    if ((opt == PLI_REMOVE || opt == PLI_CLEAR) &&
                         plp_eval == PBP_TRUE)
    /* aus meiner Liste ist er geloescht, loesche auf eval-server */
    {
      info = pvm_initsend(plp_encoding);
      #ifdef NETDEBUG
        if (info < 0)
```

```
  plp_error("plo_find_ts (pvm_initsend)",
               "Can't allocate memory!");
#endif

info = pvm_pkint(&plp_main_id , 1, 1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plo_find_ts (pvm_pkint)",
                 "Can't allocate memory!");
#endif

if (opt == PLI_REMOVE)     info = pvm_pkstr("o");
if (opt == PLI_CLEAR)  info = pvm_pkstr("c");
#ifdef NETDEBUG
if (info < 0)
  plp_error("plo_find_ts (pvm_pkstr)",
                 "Can't allocate memory!");
#endif

plp_pack_obj(tsid);

info = pvm_send(plp_eval_tid, 5);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plo_find_ts (pvm_send)", "Error in pvm_send!");
#endif

info = pvm_recv(plp_eval_tid, 5);
          /* Receive message from eval-server  */
switch (info){
case PvmBadParam :
  plp_error("plo_find_ts", "Wrong parameters in pvm_recv");
  break;
case PvmSysErr :
  plp_error("plo_find_ts", "Pvmd is not responding");
  break;
}
```

```
    info = pvm_upkstr(&flag);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_find_ts (pvm_upkstr)",
                       "Can't allocate memory!");
    #endif

    if (flag == 'z')
    {
      return(NULL);
    }else{
      newTs =
        (Pli_Tuple_Space *)malloc(sizeof(Pli_Tuple_Space));
      if(newTs == NULL)
        plp_error("plo_find_ts (malloc)",
                             "not enough memory");
      return(newTs);
    }
  }

  if (search == NULL) return(NULL);

  if (plp_eval == PBP_FALSE)
  {
    info = pvm_initsend(plp_encoding);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_find_ts (pvm_initsend)",
                         "Can't allocate memory!");
    #endif

    info = pvm_pkint(&plp_main_id , 1, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_find_ts (pvm_pkint)",
                           "Can't allocate memory!");
    #endif
```

```
    if (opt == PLI_REMOVE) info = pvm_pkstr("k");
    if (opt == PLI_CLEAR)  info = pvm_pkstr("c");
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_find_ts (pvm_pkstr)",
                          "Can't allocate memory!");
    #endif

    for (i = 0; i < 20; i++)
    {
      if (search->tids[i] == 0) break;
      info = pvm_send(search->tids[i], 1);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plo_find_ts (pvm_send)",
                            "Error in pvm_send!");
      #endif
    }
  }
  return(search);       /* tsid is not a tuple space identity */
}
```

### plo_matching

⟨*Interne O-Operationen*⟩+≡

```
extern int plo_matching(Pbp_Object **tup,
                        Ple_Template *temp,
                        Pli_Op_Type *optype,
                        Pbp_Object *tsid,
                        int tempNum);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist `TRUE`, wenn `tup` mit `temp` matcht, sonst `FALSE`. |
| `tup` | ist die Adresse des zu überprüfenden Tupels. Im Fall von `PLI_MEETINTO` wird hier auch der neue Wert zurückgegeben. |
| `temp` | ist das zu überprüfende Template. |
| `optype` | gibt zurück, ob `tup` verändert wurde. |
| `tsid` | enthält die Id des Tupelraums. |
| `tempNum` | enthält die Nummer des zu untersuchenden Templates. |
| `threadid` | enthält die Id des Threads, dem das Template gehoert. |

**Beschreibung:** `plo_matching` prüft, ob `tup` zu `temp` paßt. Ist das der Fall, werden die formals ausgewertet, `tup` gemäß der `into`–Angabe verändert, `optype` entsprechend gesetzt und `TRUE` zurückgegeben.

**Ausnahmen:**

**Probleme:** noch nicht. Aber da davon ausgegangen werden kann, daß alle hier auftretenden Objekte schon überprüft wurden, wird teilweise auf die Komponenten eines Objektes direkt zugegriffen, oder es werden Tupelfunktionen direkt aufgerufen.

**Abhängigkeiten:** `pbo_assign`, `pbf_call`, `pbo_equal`, `pbt_assign_i`

⟨*Implementierung der Operationen*⟩+≡

```
int plo_matching(Pbp_Object **tup,
                        Ple_Template *temp,
                        Pli_Op_Type *optype,
                        Pbp_Object *tsid,
                        int tempNum)
{
 register unsigned compNum; /*the number of the template
                               component */
 Pbp_Object *savetup = pbp_omega; /* a copy of tup */
 Ple_Temp_Comp *tc;          /* the actual template component*/

  #ifdef NETDEBUG
    pvm_perror("plo_matching: Here I am");
  #endif
  /* Check the arities: */
```

```
if ((*tup)->pbp_comp_number != temp->arity) /* -> problems */
{
  return(FALSE);                 /* no matching */
}
#ifdef NETDEBUG
  pvm_perror("plo_matching: arity");
#endif
/* Check the template condition: */
if (temp->condition != NULL)
{
  #ifdef NETDEBUG
    pvm_perror("plo_matching: condition != NULL");
  #endif
  pbo_assign(&savetup, *tup); /* pbf_call kills the
                               arguments */
  #ifdef NETDEBUG
    pvm_perror("plo_matching: call pbf_call");
  #endif
  if (pbf_call(temp->condition, temp->dL,
         temp->RetStat, temp->RetAd,
    &savetup) == pbp_false) /* -> problems */
  {
    #ifdef NETDEBUG
      pvm_perror("plo_matching: return false");
    #endif
    return(FALSE);               /* no matching */
  }
}
#ifdef NETDEBUG
  pvm_perror("plo_matching: RetStat");
#endif
/* comparing the actuals: */
tc = temp->comps;
for (compNum = 1; compNum <= temp->arity; compNum++)
{
  if (tc->type == PLE_ACTUAL)
  {
    #ifdef NETDEBUG
```

```
        pvm_perror("plo_matching: Ueberpruefung ACTUAL");
      #endif
      if (pbp_false == pbo_equal(pbt_extract_i(*tup, compNum),
                         tc->comp.actual))
          /* -> problems */
      {
        #ifdef NETDEBUG
          pvm_perror("plo_matching: escape FASLE");
        #endif
        return(FALSE);             /* no matching */
      }
    } /* formals match unconditionally */
    tc = tc->next;
  }
  #ifdef NETDEBUG
    pvm_perror("plo_matching: actuals");
  #endif
  /* The actuals are equal:
     assign the lvalues and evaluate the intos of the formals: */
  tc = temp->comps;
  for (compNum = 1; compNum <= temp->arity; compNum++)
  {
    if (tc->type == PLE_FORMAL)
    {
        /* assign the lvalues of the formals: */
      if (tc->comp.formal.lvalue != NULL)
      {
        pbo_assign(tc->comp.formal.lvalue,
                   pbt_extract_i(*tup, compNum));
        /* -> problems */
      }
      /* evaluate the intos of the formals: */
      if (tc->comp.formal.into != NULL)
      {
        *optype = PLI_MEETINTO;
        pbo_assign(&savetup, *tup);
          /* pbf_call kills the arguments */
        pbt_assign_i(*tup, compNum, /* -> problems */
```

```
          pbf_call(tc->comp.formal.into, tc->comp.formal.dL,
            tc->comp.formal.RetStat,
            tc->comp.formal.RetAd, &savetup));
#ifdef Linda_Debug
        printf ("plo_matching: into ");
#endif
      }
    }
    tc = tc->next;
  }
  #ifdef NETDEBUG
    pvm_perror("plo_matching: escape TRUE");
  #endif
  return(TRUE);                    /* matching */
}
```

**plo_match_in_ts**

⟨*Externe O-Operationen*⟩≡

```
extern int plo_match_in_ts(Pli_Tuple_Space *ts,
           Ple_Template *temps,
         Pbp_Object **changedtup, Pli_Op_Type *optype);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die Nummer des passenden Templates oder 0, falls keins matcht. |
| `ts` | ist der Tupelraum, in dem sich das gesuchte Tupel befinden soll. |
| `temps` | ist die Liste der Templates. |
| `changedtup` | Im Fall von `PLI_MEETINTO` wird hier das veränderte Tupel zurückgegeben. |
| `optype` | gibt an, ob es sich um eine `meet`– oder eine `fetch`–Operation handelt. Falls das gesuchte Tupel verändert wurde, wird `PLI_MEETINTO` zurückgegeben. |

**Beschreibung:** `plo_match_in_ts` überprüft, ob irgend ein Template in `temps` mit irgend einem Tupel in `ts` matcht. Sollte dies der Fall sein, wird die Nummer des passenden Template zurückgegeben, sonst 0.

**Ausnahmen:**

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:** `plo_matching`

⟨*Implementierung der Operationen*⟩+≡

```
int plo_match_in_ts(Pli_Tuple_Space *ts, Ple_Template *temps,
         Pbp_Object **changedtup, Pli_Op_Type *optype)
{
 int hashNum,                /* number of the actual hash list */
     tempNum = 1;              /* number of the actal template */
 Pli_Hash_List *hashList, *prevList; /* members of a hash list */
 Ple_Template *temp;

  /* checking for each template in temps */
  temp = temps;
  while(temp != NULL)
  {
       /* getting the hash number */
    hashNum = plo_hash(temp->arity);
      /* getting the hash list */
    hashList = (ts->hashtable)[hashNum];
```

```
/* checking for each tuple in hash list */
if(hashList != NULL)
{
  #ifdef NETDEBUG
    pvm_perror("match_in_ts: call plo_matching");
  #endif
  if(plo_matching(&(hashList->tuple), temp, optype,
                      ts->atom, tempNum))
  /* checking the first tuple */
  {
    #ifdef NETDEBUG
      pvm_perror("match_in_ts: nach plo_matching");
    #endif
    /* this tuple matched */
    if(*optype != PLI_MEET)
    {
        /* tuple has been changed or fetched:
           it has to be removed */
        (ts->hashtable)[hashNum] = hashList->next;
    /* only in case of  PLI_MEETINTO relevant */
      *changedtup = hashList->tuple;
      free(hashList);
           /* decreasing the size of the tuple space */
      ts->card--;
    }
    #ifdef NETDEBUG
  pvm_perror("match_in_ts: after plo_matching matched");
    #endif
    return(tempNum);    /* returning the number of
                     the selected template */
    }
  #ifdef NETDEBUG
   pvm_perror("match_in_ts: after plo_matching not matched");
  #endif
  /* checking all other tuple in hash list */
  prevList = hashList;
  hashList = prevList->next;
  while(hashList != NULL)
```

```
      {
        if(plo_matching(&(hashList->tuple), temp, optype,
                           ts->atom, tempNum))
            /* checking the actual tuple */
        {
          /* this tuple matched */
          if(*optype != PLI_MEET)
          {
            /* tuple has been changed or fetched:
              it has to be removed */
            prevList->next = hashList->next;
            /* only in case of  PLI_MEETINTO relevant */
            *changedtup = hashList->tuple;
            free(hashList);
              /* decreasing the size of the tuple space */
            ts->card--;
          }
           /* returning the number of the selected template */
          return(tempNum);
        }
        /* shifting the tuple */
        prevList = hashList;
        hashList = prevList->next;
      }
    }
    temp = temp->next;    /* shifting the template */
    tempNum++;              /* shifting the number */
  }
  return(0);                /* no matching */
}
```

### C.2.7 Interne Variablen

Die Liste aller Tupelräume:

⟨*Interne Variablen*⟩≡
```
/* begin of the list of all tuple spaces */
extern Pli_Tuple_Space *allTupleSpaces;
```

⟨*Interne Variablen*⟩+≡
```
/* begin of the list of all tuple spaces */
extern  Pli_Spawned_Proc *allProcesses;




/* event file for pld */
extern FILE *gps_trace_file;
extern char *gps_trace_file_name;
```

⟨*Implementierung der internen Variablen*⟩≡
```
Pli_Tuple_Space *allTupleSpaces = NULL;
 Pli_Spawned_Proc *allProcesses = NULL;

/* trace file for pld */
FILE *gps_trace_file = NULL;
char *gps_trace_file_name = "tracefile.pld";
```

### C.2.8 C Dateien

⟨*linda.h*⟩≡
  ⟨*Exportierte O-Operationen*⟩

⟨*intLinda.h*⟩≡
  ⟨*Interne Variablen*⟩
  ⟨*Interne O-Operationen*⟩

$\langle$*operation.c*$\rangle\equiv$
  $\langle$*Importe der Quelltextdatei*$\rangle$
  $\langle$*Implementierung der internen Variablen*$\rangle$
  $\langle$*Lokale Operationen*$\rangle$
  $\langle$*Implementierung der Operationen*$\rangle$

# C.3 Die Linda–Prozeßbehandlung

## C.3.1 Einleitung

Dieses Modul enthält die Routinen zur Prozeßbehandlung. Parallelität wird durch eigenständige Prozesse verwirklicht. Interprozeßkommunikation findet mit Hilfe der PVM-Library statt.

## C.3.2 Abhängigkeiten

⟨*Importe der Quelltextdatei*⟩≡
```
#include "proset/proset.h"
#include "proset/unit.h"
#include "proset/overloaded.h"
#include "proset/function.h"
#include "proset/handler.h"
#include "proset/module.h"
#include "proset/loop.h"
#include "proset/atom.h"
#include "proset/string.h"
#include "proset/set.h"
#include "proset/tuple.h"
#include "proset/io.h"

#include "linda/linda.h"
#include "../intLinda.h"
```

## C.3.3 Exportierte Operationen

#### plp_spawn

⟨*Exportierte P-Operationen*⟩≡
```
extern Pbp_Object *plp_spawn(pbp_form_par, Pbp_Object *tospawn,
    Pbp_Env dL, Pbp_Exit *PS_RetStat, Pbp_Env *PS_RetAd, ...);
```

**Profil:**

| | |
|---|---|
| `tospawn` | ist die Prozedur die parallel gestartet wird. |
| `dL` | ist der dynamische Vorgänger von `tospawn`. |
| `PS_RetStat` | gibt an, ob eine Ausnahme ausgelöst wurde (nicht benutzt). |
| `PS_RetAd` | enthält im Ausnahmefall den dynamischen Vorgänger des Handlers (nicht benutzt). |
| ... | sind Parameter von `tospawn`. |

**Beschreibung:** `plp_spawn` startet einen neuen Prozeß. Dies geschieht unabhängig davon, ob dieser Prozeß Teil eines Tuples ist oder nicht.

**Ausnahmen:** keine.

**Probleme:**

**Abhangigkeiten:**

⟨*Implementierung der Operationen*⟩≡

```
Pbp_Object *plp_spawn(pbp_form_par, Pbp_Object *tospawn, Pbp_Env dL,
          Pbp_Exit *PS_RetStat, Pbp_Env *PS_RetAd, ...)
{
  extern  Pli_Spawned_Proc *allProcesses;
  extern int plp_eval_tid;
  extern int plp_encoding;   /* Wird XDR benutzt oder nicht? */
  Pbp_Object *retObj = pbo_create(PBP_PROCESS);
                       /* Der Rueckgabewert */
  int info;                  /* Die Fehlervariable fuer PVM */
  int newtids[1];          /* die ID des neuen Prozesses */
  int length;
  Ple_Hostname_List *hostlist; /* Die nach dem Load sortierte Liste
                                  der Hosts*/
  char **tmp_argv;       /* Die Parameterliste des Prozesses */
  va_list ap;                  /* Liste der Argumente */
  register i;                    /* Index */
  Pli_Tuple_Space *ts;     /* Zeiger zum Verwalten aller TS */
  Pli_Spawned_Proc *newProc;

 /* Speicher allokieren */
   tmp_argv = (char **) malloc (sizeof (char *)*2);
     if (tmp_argv == NULL)
```

```
  plp_error("plp_spawn (malloc)", "Not enough memory");

  tmp_argv[0] = (char *)calloc(2, sizeof(char));
  if (tmp_argv[0] == NULL)
  plp_error("plp_spawn (malloc)", "Not enough memory");

sprintf(tmp_argv[0],"r");
tmp_argv[1] = NULL;

/* output redirected to master task */
pvm_catchout(stdout);

if (plp_loadbalancing == PBP_TRUE)
{
  /* Loadbalancierung (ein Host zurueckgeben)*/
  hostlist = plp_min_load(1);
  /* Prozess starten (mit Loadbalacierung) */
  info = pvm_spawn(plp_my_name, tmp_argv, (PvmTaskHost),
          hostlist->hostname, 1, newtids);
  free(hostlist->hostname);   /* Loadliste freigeben */
  free(hostlist);
}else{        /* Prozess starten (ohne Loadbalancierung) */
  info = pvm_spawn(plp_my_name, tmp_argv, (PvmTaskDefault),
                         "", 1, newtids);
}
if (info < 1) plp_error("plp_spawn (pvm_spawn)",
                         "Can't spawn task!");

 free(tmp_argv[0]);         /* Parameter freigeben */
free(tmp_argv);

newProc = (Pli_Spawned_Proc *)malloc(sizeof(Pli_Spawned_Proc));
if(newProc == NULL)
  plp_error("plp_spawn (malloc)", "not enough memory");

newProc->tid = newtids[0];
newProc->next = allProcesses;
allProcesses = newProc;
```

```
info = pvm_initsend(plp_encoding); /* Buffer vorbereiten */
#ifdef NETDEBUG
 if (info < 0) plp_error("plp_spawn (pvm_initsend)",
                     "Can't allocate memory!");
#endif

if (pbp_type(tospawn) != PBP_FUNCTION)
  plp_error("plp_spawn (pbp_type)",
                 "tospawn is not function!");

#ifdef NETDEBUG
  pvm_perror("plp_spawn: Remote-Funktion einpacken");
#endif

plp_pack_obj(tospawn);        /* Remotefunktion einpacken */

 va_start(ap, PS_RetAd);        /* Initialisierung von  ap */
/* pack the parameter */

for (i=0; i < pbu_param_count(tospawn); i++)
 {
  #ifdef NETDEBUG
    pvm_perror("plp_spawn: Argumente einpacken ");
  #endif
                      /* Argumente einpacken */
  plp_pack_obj(*((Pbp_Object**)va_arg(ap, Pbp_Object**)));
 }
  va_end(ap);

ts = allTupleSpaces;
while (ts != NULL)          /* Bekannte TS einpacken */
{
  #ifdef NETDEBUG
    pvm_perror("Spawn: Tupelraeume einpacken");
  #endif
  plp_pack_obj(ts->atom);
```

```
    info = pvm_pkint(&(ts->max_partitions), 1, 1);
    #ifdef NETDEBUG
    if (info < 0)
        plp_error("plp_spawn (pvm_pkint)",
                       "Can't allocate memory!");
    #endif
    info = pvm_pkint(&(ts->hash_mode), 1, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_spawn (pvm_pkint)",
                       "Can't allocate memory!");
    #endif
    info = pvm_pkint(ts->tids, ts->max_partitions, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_spawn (pvm_pkint)",
                       "Can't allocate memory!");
    #endif

    ts = ts->next;
  }
  length = 0;
  info = pvm_pkint(&length, 1, 1);
  #ifdef NETDEBUG
  if (info < 0)
      plp_error("plp_spawn (pvm_pkint)",
                         "Can't allocate memory!");
  #endif

  info = pvm_pkint(&plp_eval_tid, 1, 1);
  #ifdef NETDEBUG
  if (info < 0)
      plp_error("plp_spawn (pvm_pkint)",
                         "Can't allocate memory!");
  #endif

    /* Nun buffer an Remote-Task senden */
    info = pvm_send(newtids[0], 2);
```

```
  #ifdef NETDEBUG
    if (info < 0)
    plp_error("plp_spawn (pvm_send)",
                  "Spawned task is not responding!");
  #endif

#if TRACE
  if(gps_linda_debug)
  {
    printf("##### I spawned process with thread id %d #####\n",
     PLI_THREAD_ID(newid));
  }
/* trace informations */
  if(gps_linda_trace)
  {
    fprintf(gps_trace_file, "p b %d %d %i %s\n", plp_thread_id(),
            PLI_THREAD_ID(newid), pbh_get_line(), pbh_get_file());
  }
#endif

  retObj->pbp_proc_id = newtids[0];
  #ifdef NETDEBUG
    pvm_perror("plp_spawn: ENDE");
  #endif
  return(retObj);
}
```

### plp_wait_task

⟨*Exportierte P-Operationen*⟩+≡

```
extern Pbp_Object *plp_wait_task(Pbp_Object *task);
```

**Beschreibung:** Wartet auf das Ergebnis eines mit dem ||-Operator gestarteten Prozesses.

**Ausnahmen:** Keine.

**Probleme:**

**Abhängigkeiten:**

⟨*Implementierung der Operationen*⟩+≡

```
Pbp_Object *plp_wait_task(Pbp_Object *task)
{
 extern int plp_eval_tid;
 int length;       /* Laenge des objstr */
 char *objstr;       /* Ergebnisstring */
 char *tmp;
 Pbp_Object *retobj;    /* Rueckgabeobjekt */
 int info;         /* Fehlervariable fuer PVM */
 int rbuffer;
 int size;         /* Groesse der Gruppe "load"*/
 int remote_tid;      /* Tid des Remote-Prozesses */

  #ifdef NETDEBUG
    pvm_perror("wait_task: BEGIN");
  #endif

  remote_tid = task->pbp_proc_id;

  if (plp_eval == PBP_TRUE)
  {
    /* Ergebnis vom Eval-Server abrufen */
    info = pvm_initsend(plp_encoding);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_wait_task (pvm_initsend)",
                        "Can't allocate memory!");
    #endif

    info = pvm_pkint(&plp_main_id, 1, 1);
```

```
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_wait_task (pvm_pkint)",
                         "Can't allocate memory!");
    #endif

    info = pvm_pkstr("g");
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_wait_task (pvm_pkstr)",
                        "Can't allocate memory!");
    #endif

    info = pvm_pkint(&remote_tid, 1, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_wait_task (pvm_pkint)",
                        "Can't allocate memory!");
    #endif

    /* zum eval-server senden */
    info = pvm_send(plp_eval_tid, 5);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_wait_task (pvm_send)",
                           "Error in pvm_send!");
    #endif

    /* Auf Antwort warten */
     /* Receive message from eval-server  */
    rbuffer = pvm_recv(plp_eval_tid, 5);
  }else{
     /* Receive message from remote_process  */
    rbuffer = pvm_recv(remote_tid, 5);
  }
  #ifdef NETDEBUG
    switch (rbuffer){
    case PvmBadParam :
```

```
    plp_error("plp_wait_task", "Wrong parameters in pvm_recv");
    break;
    case PvmSysErr :
    plp_error("plp_wait_task", "Pvmd is not responding");
    break;
    }
  #endif

  info = pvm_upkint(&length, 1,1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_wait_task (pvm_upkint)",
                   "Can't allocate memory!");
  #endif

  if ((objstr = (char *)malloc(sizeof (char)*(length+1))) == NULL)
      plp_error("plp_wait_task (malloc)",
                "Cannot allocate memory!");

  info = pvm_upkstr(objstr);  /* unpack the retobj */
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_wait_task (pvm_upkstr)",
                  "Can't allocate memory!");

    pvm_perror(objstr);
  #endif
  tmp = objstr;
  retobj = pli_scan(&objstr);
  free(tmp);
  pvm_freebuf(rbuffer);
  return(retobj);

}
```

### plp_init_linda

$\langle$*Exportierte P-Operationen*$\rangle+\equiv$

```
extern void plp_init_linda(int argc, char *argv[]);
```

**Beschreibung:** Erzeugt und initialisiert die benötigten Monitorvariablen und die Conditionvariable für die blockierten Prozesse und setzt die POD Priorität.

**Ausnahmen:**

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:** LWP–Routinen

⟨*Implementierung der Operationen*⟩+≡
```
void plp_init_linda(int argc, char *argv[])
{
 extern int plp_eval_tid;
 extern int plp_encoding;
 extern int plp_max_hosts;
 extern struct pvmhostinfo *plp_hostp[15];
 char arg;
 int narch;
 int info;
 int tids[20];
 int size;
 int i;
 char **tmp_argv;
 Ple_Hostname_List *hostlist;
 int newtids[1];            /* die ID des Eval-Servers */

  /* First look for pvmd and get my ID*/
  plp_main_id = pvm_mytid();
  if (plp_main_id == PvmSysErr)
    plp_error("plp_init_linda", "Maybe pvmd is not running!");

  plp_my_name = (char *)calloc(strlen(argv[0])+1,sizeof(char *));
  if (plp_my_name == NULL)
    plp_error("plp_init_linda", "Cannot allocate memory!");

      /* direkt Routen mit TCP */
  pvm_setopt(PvmRoute, PvmRouteDirect);
    /* stdout umleiten zum Master */
  pvm_setopt(PvmOutputTid, 0);
```

```
      /* Zeilenorientiert ausgeben */
  setlinebuf(stdout);

  info = pvm_config( &plp_max_hosts, &narch, plp_hostp );
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_init_linda", "Pvmd not responding!");
  #endif

  if (narch > 1)     /* ist XDR noetig oder nicht? */
  {
    plp_encoding = PvmDataDefault;  /* XDR einschalten */
  }else{
    plp_encoding = PvmDataRaw;  /* Daten direkt uebertragen */
  }

sprintf(plp_my_name,"%s", argv[0]);
  /* Which ID has my parent? */
  plp_parent_id = pvm_parent();

  #ifdef NETDEBUG
    pvm_perror(argv[1]);
  #endif
  if (argc > 1)
  {
    arg = *argv[1];
  }else{
    arg = ' ';
  }

  switch(arg){
  case 'p':                /* I'm a partition */
    plp_partition(argc, argv);
    break;
    case 'r':                /* I'm a remote task */
    /* Konfigurationsdatei auslesen und
              globale Variablen setzten */
    pli_read_file("netproset");
```

```
    plp_remote();
    break;
  case 'e':
    plp_eval_server();
    break;
  default:                /* I'm the main task */
    /* Konfigurationsdatei auslesen und
                  globale Variablen setzten */
    pli_read_file("netproset");
    if (plp_loadbalancing == PBP_TRUE) /* load agenten starten */
    {
    /* Starte auf jedem Host (scheint so zu funktionieren)*/
      info = pvm_spawn("/home/diplom/waltenbe/loader",
         (char **)0, PvmTaskDefault, "", plp_max_hosts, tids);
      if (info < plp_max_hosts)
        plp_error("plp_spawn (pvm_spawn)",
                        "Can't spawn task!");
    }

    if (plp_eval == PBP_TRUE) /* Eval-Server starten */
    {
      #ifdef NETDEBUG
        pvm_perror("plp_init_linda: Eval-server starten");
      #endif
    /* Speicher allokieren */
      tmp_argv = (char **) malloc (sizeof (char *)*2);
        if (tmp_argv == NULL)
        plp_error("plp_init_linda (malloc)",
                         "Not enough memory");

        tmp_argv[0] = (char *)calloc(2, sizeof(char));
        if (tmp_argv[0] == NULL)
        plp_error("plp_init_linda (malloc)",
                       "Not enough memory");

      sprintf(tmp_argv[0],"e");
      tmp_argv[1] = NULL;
```

```
        if (plp_loadbalancing == PBP_TRUE)
        {
          /* Loadbalancierung (ein Host zurueckgeben)*/
          hostlist = plp_min_load(1);
          /* Prozess starten (mit Loadbalacierung) */
          info = pvm_spawn(plp_my_name, tmp_argv, PvmTaskHost,
                 hostlist->hostname, 1, newtids);
          free(hostlist->hostname);   /* Loadliste freigeben */
          free(hostlist);
        }else{
        /* Eval-Server starten (ohne Loadbalancierung) */
          info = pvm_spawn(plp_my_name, tmp_argv,
                    PvmTaskDefault, "", 1, newtids);
        }
        if (info < 1) plp_error("plp_spawn (pvm_spawn)",
                             "Can't spawn task!");
        free(tmp_argv[1]);
        free(tmp_argv[0]);
        free(tmp_argv);
        plp_eval_tid = newtids[0];
      }
      break;
    }
  }
```

**plp_finish_linda**

⟨*Exportierte P-Operationen*⟩+≡

```
  extern void plp_finish_linda(int exval);
```

**Beschreibung:** Der Hauptprozeß beendet jeden noch aktiven Prozeß und verläßt den POD. Falls noch aktive Prozesse vorhanden sind, wird eine Warnung ausgegeben.

**Ausnahmen:**

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:** LWP

⟨*Implementierung der Operationen*⟩+≡

```
void plp_finish_linda(int exval)
{
  extern  Pli_Spawned_Proc *allProcesses;
  Pli_Spawned_Proc *proc;
  Pli_Tuple_Space *ts;
  int info;
  int i;
   int theend = 0;

/*  info = pvm_initsend(plp_encoding);
    if (info < 0)
     plp_error("plp_finish_linda (pvm_initsend)",
                      "Can't allocate memory!");

  info = pvm_pkint(&plp_main_id, 1, 1);
    if (info < 0)
     plp_error("plp_finish_linda (pvm_pkint)",
                       "Can't allocate memory!");

    info = pvm_pkstr("k");
    if (info == PvmNoMem)
     plp_error("plp_finish_linda (pvm_pkstr)",
                       "Can't allocate memory!");
*/
  ts = allTupleSpaces;
  while (ts != NULL)
  {
    for(i=0; i < ts->max_partitions; i++)
    {
```

```
/*      info = pvm_send(ts->tids[i], 1);
      if (info < 0)
        plp_error("plp_finish_linda (pvm_send)",
                       "Pvmd is not responding!");
*/
      pvm_kill(ts->tids[i]);
    }
  ts = ts->next;
  }

  proc = allProcesses;
  while (proc != NULL)
  {
    pvm_kill(proc->tid);
    proc = proc->next;
  }

  if (plp_loadbalancing == PBP_TRUE)
  {
    /* Beende die Load-Agenten */
    info = pvm_initsend(PvmDataDefault);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_finish_linda (pvm_initsend)",
                     "Can't allocate memory!");
    #endif

    info = pvm_pkint(&theend , 1, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_finish_linda (pvm_pkint)",
                      "Can't allocate memory!");
    #endif

    info = pvm_bcast("load",4);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_finish_linda", "Error in pvm_bcast!");
```

```
    #endif
  }
  if (plp_eval == PBP_TRUE)
  {
    /* Eval-Server beenden */
    pvm_kill(plp_eval_tid);
  }
  pvm_exit();
  exit(exval); /* exit the entire process */
}
```

**plp_error**

⟨*Exportierte P-Operationen*⟩+≡

```
extern void plp_error(char *str1, char *str2);
```

**Beschreibung:**

**Ausnahmen:**

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:**

⟨*Implementierung der Operationen*⟩+≡

```
void plp_error(char *str1, char *str2)
{
  pvm_perror(str1);
  pvm_exit;
  pbp_fehler(str1, PBP_FATAL, str2);
  exit(1);
}
```

**plp_remote**

⟨*Exportierte P-Operationen*⟩+≡

```
extern void plp_remote();
```

**Beschreibung:**

**Ausnahmen:**

**Probleme:**

**Abhängigkeiten:**

⟨*Implementierung der Operationen*⟩+≡

```
void plp_remote()
{
  extern int plp_encoding;
  int parent_tid;  /* The tid of my parent */
  Pbp_Object *tospawn; /* the procedure */
  Pbp_Object *retobj;  /* return object of the procedure */
  int paramcount;     /* the number of the parameters */
  int partitiontids[PLI_MAX_PARTITIONS];
  int length;
  int i;
  int info;
  int rbuffer;
  int tid;
  char *objstr;
  Pli_Tuple_Space *newTs;
  Pbp_Env dL;
  Pbp_Exit *RetStat;
  Pbp_Env *RetAd;
  Pbp_Object *pars[10];
 Pbp_Object *tmp;
  int size;
  char *hbuf;

    /* I get the message from a partition */
   rbuffer = pvm_recv(-1, 2);
    switch (rbuffer){
   case PvmBadParam :
     plp_error("plp_remote (recv)",
                  "Wrong parameters in pvm_recv");
     break;
```

```
    case PvmSysErr :
     plp_error("plp_remote (recv)", "Pvmd is not responding");
     break;
    }

   tospawn = plp_unpack_func();  /* tospawn auspacken */

   #ifdef NETDEBUG
     pvm_perror("plp_remote: nach unpack_func");
   #endif

   paramcount = pbu_param_count(tospawn);
   #ifdef NETDEBUG
     printf(" Paramcount: %d\n", paramcount);
   #endif
 /* Get memory for the parameters and initialize parlist: */

 /*    if ((pars =
   (Pbp_Object **)malloc((1+paramcount) * sizeof(char))) == NULL)
       plp_error("plp_remote (malloc)", "Cannot allocate memory");
 */
   for (i=0; i < paramcount; i++)
   {
     #ifdef NETDEBUG
       pvm_perror("plp_remote: Parameter auspacken");
     #endif
     info = pvm_upkint(&length, 1, 1);
     #ifdef NETDEBUG
     if (info < 0)
       plp_error("plo_remote (pvm_upkint)",
                      "Can't allocate memory!");
     #endif

     if ((objstr =
       (char *)malloc(sizeof (char)*(length+1))) == NULL)
         plp_error("plp_remote (malloc)",
                      "Cannot allocate memory!");
```

```
    info = pvm_upkstr(objstr);  /* unpack the parameter*/
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_remote (pvm_upkstr)",
                          "Can't allocate memory!");
    #endif
/*raus oder rein*/
    if (((pars[i]) =
       (Pbp_Object *)malloc(sizeof(Pbp_Object *))) == NULL)
      plp_error("plo_remote (malloc)",
                      "Can't allocate memory!");

    hbuf = objstr;
    tmp =  pli_scan(&objstr);
    free(hbuf);
    pbp_set_use(tmp);
    pars[i]=tmp;
/*    pbo_assign(pars[i], tmp);*/

  }

  for(;;)
  {
    #ifdef NETDEBUG
      pvm_perror("plp_remote: TS auspacken");
    #endif

    info = pvm_upkint(&length, 1, 1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plo_remote (pvm_upkint)",
                  "Can't allocate memory!");
    #endif

    if (length == 0) break;

    /* create a new tuple space */
    if ((newTs =
```

```
(Pli_Tuple_Space *)malloc(sizeof(Pli_Tuple_Space))) == NULL)
    plp_error("plp_remote (malloc)",
                  "Cannot allocate memory!");

if ((newTs->atom_str =
    (char *)malloc(sizeof(char)*(length+1))) == NULL)
    plp_error("plp_remote (malloc)",
                 "Cannot allocate memory!");

 /* unpack the atom string */
info = pvm_upkstr(newTs->atom_str);
if (info < 0)
  plp_error("plo_remote (pvm_upkstr)",
                   "Can't allocate memory!");

  newTs->atom = pli_scan(&(newTs->atom_str));
pbp_set_use(newTs->atom);

info = pvm_upkint(&(newTs->max_partitions), 1, 1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plo_remote (pvm_upkint)",
                  "Can't allocate memory!");
#endif

info = pvm_upkint(&(newTs->hash_mode), 1, 1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plo_remote (pvm_upkint)",
                  "Can't allocate memory!");
#endif

info = pvm_upkint(partitiontids, newTs->max_partitions, 1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plo_remote (pvm_upkint)",
                   "Can't allocate memory!");
#endif
```

```
    /* save the partition ids */
    for (i=0; i < newTs->max_partitions; i++)
      newTs->tids[i]  = partitiontids[i];

    /* putting newTs in the list of all tuple spaces */
      newTs->next = allTupleSpaces;
      allTupleSpaces = newTs;
  }

  /* Tid des Eval-Servers auspacken */
  info = pvm_upkint(&plp_eval_tid, 1, 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plo_remote (pvm_upkint)",
                 "Can't allocate memory!");
  #endif

  pvm_freebuf(rbuffer);

  if ((dL = (Pbp_Env)malloc(sizeof(Pbp_Env))) == NULL)
        plp_error("plp_remote", "Cannot allocate memory!");

   if ((RetStat = (Pbp_Exit*)malloc(sizeof(Pbp_Exit))) == NULL)
    plp_error("plp_remote", "Cannot allocate memory!");

  *RetStat = PBP_RESUME;
  if ((RetAd = (Pbp_Env*)malloc(sizeof(Pbp_Env))) == NULL)
      plp_error("plp_remote", "Cannot allocate memory!");
  RetAd == NULL;

/*   retobj = pbu_entry_point(tospawn)(tospawn, dL, RetStat,
         RetAd, pars);*/


   retobj = pbf_call(tospawn,dL,RetStat,RetAd, &pars);

   parent_tid = pvm_parent();
```

```
info = pvm_initsend(plp_encoding);
#ifdef NETDEBUG
 if (info < 0)
  plp_error("plp_remote (pvm_initsend)",
                   "Can't allocate memory!");
#endif

if (plp_eval == PBP_TRUE)
{
  info = pvm_pkint(&parent_tid, 1, 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_remote (pvm_pkint)",
                   "Can't allocate memory!");
  #endif

  info = pvm_pkstr("r");
  #ifdef NETDEBUG
   if (info < 0)
    plp_error("plp_remote (pvm_pkstr)",
                     "Can't allocate memory!");
  #endif
  tid = plp_eval_tid;
}else{
  tid = parent_tid;
}

plp_pack_obj(retobj);

#ifdef NETDEBUG
  pvm_perror("plp_remote: nach pack_obj");
#endif

info = pvm_send(tid, 5);
#ifdef NETDEBUG
 if (info < 0)
  plp_error("plp_remote (pvm_send)", "Error in pvm_send!");
```

```
    pvm_perror("plp_remote: ENDE");
  #endif
  pvm_exit;      /* Beim pvmd abmelden */
  exit(0);           /* Beende Remote-Task */
}
```

**plp_eval_server**

⟨*Exportierte P-Operationen*⟩+≡

```
extern void plp_eval_server();
```

**Beschreibung:**

**Ausnahmen:**

**Probleme:**

**Abhängigkeiten:**

⟨*Implementierung der Operationen*⟩+≡

```
void plp_eval_server()
{
  extern Pbp_Bool_Type plp_eval;
  Pli_ProcHash_List *proclist[PLI_PRIME];
  Pli_ProcHash_List *newelement = NULL;
  Pli_ProcHash_List *nextelement = NULL;
  Pli_ProcHash_List *prevelement = NULL;
  Pli_ProcHash_List *killelement = NULL;

  Pbp_Bool_Type active;
  Pbp_Bool_Type gefunden;
  Pli_Tuple_Space *newTs;
  Pli_Tuple_Space *ts;
  char operation;
  int hashnum;
  int info;
  int tid;    /* Tid des Absenders */
  int tids[20];
  int proctid;
  int partition_tid;
  int remote_tid;
  int sender_tid;
  int bytes, type;
  int length;
  int empty_tids[20];
  int bufid; /* ID des Receive-Puffers */
  Pbp_Object *object;
  Pbp_Object *tup_elem;
  Pbp_Object *tuple;
  PBP_TUPLE_ELEM *element;
```

```
char *objectstr;
char *buffer;
int i;

 plp_eval = PBP_FALSE;

 for (i = 0; i < PLI_PRIME; i++)
   proclist[i] = NULL;

 for (i = 0; i < 20; i++)
   empty_tids[i] = 0;

 while(1)
 {
   bufid = pvm_recv(-1 , 5); /* I'll get the messages */
   #ifdef NETDEBUG
     printf("%d\n", info);

   if (bufid < 0)
     plp_error("plp_eval_server(pvm_recv)",
                       "Error in pvm_recv");
   #endif

   info = pvm_upkint(&tid, 1, 1);
   #ifdef NETDEBUG
     if (info < 0)
     plp_error("plp_eval_server (pvm_upkint)",
                      "Can't allocate memory!");
   #endif

   info = pvm_upkstr(&operation);
   #ifdef NETDEBUG
   if (info < 0)
     plp_error("plp_eval_server (pvm_upkstr)",
                     "Can't allocate memory!");
   #endif

   switch(operation){
```

```
  case 'd':
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server: d");
    #endif
    /* Auspacken des Tupels */
    info = pvm_upkint(&length, 1,1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_eval_server (pvm_upkint)",
                  "Message can't be decoded!");
    #endif

    if ((objectstr =
   (char *)malloc(sizeof (char)*(length+1))) == NULL)
      plp_error("plp_eval_server (malloc)",
                  "Cannot allocate memory!");

    info = pvm_upkstr(objectstr);
    #ifdef NETDEBUG
      if (info < 0)
      plp_error("plp_eval_server (pvm_upkstr)",
                  "Message can't be decoded!");
    #endif

    info = pvm_upkint(&partition_tid, 1,1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_eval_server (pvm_upkint)",
                 "Message can't be decoded!");
    #endif

    object = pli_scan(&objectstr);
    free(objectstr);

    /* im Tupel nach aktive Prozessen suchen */
    element =
 ((PBP_TUPLE_HEADER *)object->pbp_comp_header)->First_Element ;
```

```
    while (element != NULL)
    {
      #ifdef NETDEBUG
        pvm_perror("plp_eval_server d: in while");
      #endif
      tup_elem = element->ObjFromElem;
      if (pbp_type(tup_elem) == PBP_PROCESS)
      {
      /* fuer jeden Prozess Hashen und in der Liste suchen */
        hashnum = tid % PLI_PRIME;
        nextelement = proclist[hashnum];
        prevelement = NULL;
        proctid = tup_elem->pbp_proc_id;
        while(nextelement != NULL)
        {
          #ifdef NETDEBUG
        pvm_perror("plp_eval_server d:in zweitem while");
          #endif
          if (nextelement->partition_tid == proctid &&
                   nextelement->parent_tid == tid)
          {
            if (nextelement->number == 0)
            {
              #ifdef NETDEBUG
  pvm_perror("plp_eval_server d: gefunden-> in Tupel einfuegen");
              #endif
              /* Gefunden, in Tupel einfuegen */
              element->ObjFromElem = nextelement->obj;
              /* altes Element loeschen */
              pbo_kill(tup_elem);
              /* Listenelement loeschen */
              if (prevelement == NULL)
              {
                proclist[hashnum]->next =
                             nextelement->next;
              }else{
                prevelement->next = nextelement->next;
              }
```

```
          free(nextelement);
          break;
        }
      }
      nextelement = nextelement->next;
    }
  }
  element = element->nextTupleElem ;
}
/* Tupel noch aktiv? */
active = pli_process_in_tup(object);
if (active == PBP_FALSE)
{
  #ifdef NETDEBUG
    pvm_perror("plp_eval_server d: deposit");
  #endif
  /* nein-> Deposit zur Partition */
  info = pvm_initsend(plp_encoding);
  #ifdef NETDEBUG
  if (info < 0)
  plp_error("plp_eval_server (pvm_initsend)",
                  "Can't allocate memory!");
  #endif

  info = pvm_pkint(&plp_main_id, 1, 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_eval_server (pvm_pkint)",
                  "Can't allocate memory!");
  #endif

  info = pvm_pkstr("d");
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_eval_server (pvm_pkstr)",
                  "Can't allocate memory!");
  #endif
```

```
        plp_pack_obj(object);

        /* zur richtigen Partition senden */
        info = pvm_send(partition_tid, 1);
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plp_eval_server (pvm_send)",
                          "Pvmd is not responding!");
        #endif
      }else{
        #ifdef NETDEBUG
        pvm_perror("plp_eval_server d: neues Listenelement");
        #endif
        /* Ja->Allokation eines neuen Listenelementes */
        /* Hashing */
        hashnum = tid % PLI_PRIME;

        if ((newelement =
(Pli_ProcHash_List *)malloc(sizeof (Pli_ProcHash_List))) == NULL)
          plp_error("plp_eval_server (malloc)",
                         "Cannot allocate memory!");

        /* Belegung der Variablen */
        newelement->parent_tid = tid;
        newelement->number = 1;       /* obj ist ein Tupel */
        newelement->obj = object;
        newelement->partition_tid = partition_tid;

        /* Einhaengen des Listenelementes */
        newelement->next = proclist[hashnum];
        proclist[hashnum] = newelement;
      }
        #ifdef NETDEBUG
          pvm_perror("plp_eval_server d: ENDE");
        #endif
      break;
    case 'r':
      #ifdef NETDEBUG
```

```
  pvm_perror("plp_eval_server: r");
#endif
/* Ergebnisobjekt auspacken */
gefunden = PBP_FALSE;
info = pvm_upkint(&length, 1,1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_eval_server (pvm_upkint)",
              "Can't allocate memory!");
#endif

if ((objectstr =
   (char *)malloc(sizeof (char)*(length+1))) == NULL)
    plp_error("plp_eval_server (malloc)",
               "Cannot allocate memory!");

info = pvm_upkstr(objectstr);  /* unpack the retobj */
#ifdef NETDEBUG
if (info < 0)
plp_error("plp_eval_server (pvm_upkstr)",
                "Can't allocate memory!");
#endif

object = pli_scan(&objectstr);
free(objectstr);

/* Tid des Absenders bestimmen */
info = pvm_bufinfo(bufid, &bytes, &type, &sender_tid);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_eval_server (pvm_bufinfo)",
                       "Error occurs!");
#endif

/* Hashing ueber tid */
hashnum = tid % PLI_PRIME;

/* Suche in Hashliste */
```

```
    nextelement = proclist[hashnum];
    prevelement = NULL;
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server r: vor while");
    #endif
    while(nextelement != NULL)
    {
      element = NULL;
      if (nextelement->parent_tid == tid)
      {
        if (nextelement->number == 1)
        {
          /* Gefunden, in Tupel einfuegen */
          gefunden = PBP_TRUE;
          tuple = nextelement->obj;
          element =
((PBP_TUPLE_HEADER *)tuple->pbp_comp_header)->First_Element ;
          #ifdef NETDEBUG
    pvm_perror("plp_eval_server r:vor zweitem while");
          #endif
          while (element != NULL)
          {
            tup_elem = element->ObjFromElem;
            if (pbp_type(tup_elem) == PBP_PROCESS)
            {
              if (tup_elem->pbp_proc_id == sender_tid)
              {
                element->ObjFromElem = object;
                kill(tup_elem);
              /*  Tupel noch aktiv?*/
                active =
                 pli_process_in_tup(nextelement->obj);
                if (active == PBP_TRUE) break;
              /* Nein, dann zur Partition senden */
                info = pvm_initsend(plp_encoding);
                #ifdef NETDEBUG
                if (info < 0)
                plp_error("plp_eval_server(initsend)",
```

```
                "Can't allocate memory!");
          #endif

          info = pvm_pkint(&plp_main_id, 1, 1);
          #ifdef NETDEBUG
          if (info < 0)
          plp_error("plp_eval_server(pvm_pkint)",
                  "Can't allocate memory!");
          #endif

          info = pvm_pkstr("d");
          #ifdef NETDEBUG
          if (info < 0)
          plp_error("plp_eval_server (pvm_pkstr)",
                  "Can't allocate memory!");
          #endif

          plp_pack_obj(tuple);

          /* zur richtigen Partition senden */
          info =
          pvm_send(nextelement->partition_tid, 1);
          #ifdef NETDEBUG
          if (info < 0)
          plp_error("plp_eval_server(pvm_send)",
                  "Pvmd is not responding!");
          #endif
          break;
        }
      }
      element = element->nextTupleElem ;
    }
    #ifdef NETDEBUG
pvm_perror("plp_eval_server r:nach zweitem while");
    #endif
    if (gefunden == PBP_TRUE)
    {
      /* passendes Element gefunden */
```

```
            break;
          }
        }else{
          if (nextelement->number == -1)
          {
            #ifdef NETDEBUG
          pvm_perror("plp_eval_server r:number = -1");
            #endif
            if (nextelement->partition_tid == sender_tid)
            {
              #ifdef NETDEBUG
pvm_perror("plp_eval_server r:partition_id = sender_id GEFUNDEN");
              #endif
      /* Gefunden-> aus Liste loeschen und zuruecksenden */
              gefunden = PBP_TRUE;
              info = pvm_initsend(plp_encoding);
              #ifdef NETDEBUG
              if (info < 0)
                plp_error("plp_eval_server(initsend)",
                        "Can't allocate memory!");
              #endif
              plp_pack_obj(object);

              info = pvm_send(tid, 5);
              #ifdef NETDEBUG
              if (info < 0)
                plp_error("plp_eval_server (pvm_send)",
                        "Pvmd is not responding!");
              #endif
              if (prevelement == NULL)
              {
                #ifdef NETDEBUG
        pvm_perror("plp_eval_server r:prevelement = NULL");
                #endif
                proclist[hashnum] = nextelement->next;
              }else{
                prevelement->next = nextelement->next;
              }
```

```
              free(nextelement);
              #ifdef NETDEBUG
          pvm_perror("plp_eval_server r:vor break");
              #endif
              break;
            }
          }
        }
      }
      prevelement = nextelement;
      nextelement = nextelement->next;
    }
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server r: nach erstem while");
    #endif
    /* Nicht gefunden, einfuegen in Objektliste */
    if (gefunden == PBP_FALSE)
    {
      #ifdef NETDEBUG
        pvm_perror("plp_eval_server r: neues Element");
      #endif
      /* Allokation eines neuen Listenelementes */
      if ((newelement =
(Pli_ProcHash_List *)malloc(sizeof (Pli_ProcHash_List)))== NULL)
        plp_error("plp_eval_server (malloc)",
                  "Cannot allocate memory!");

      /* Belegung der Variablen */
      newelement->parent_tid = tid;
      newelement->number = 0;    /* obj ist ein Ergebnis */
      newelement->obj = object;
      newelement->partition_tid = sender_tid;

      newelement->next = proclist[hashnum];
      proclist[hashnum] = newelement;
    }
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server r: ENDE");
```

```
    #endif
  break;
case 'g':
  #ifdef NETDEBUG
    pvm_perror("plp_eval_server: g");
  #endif
  gefunden = PBP_FALSE;
  /* PartitionsID auspacken */
  info = pvm_upkint(&remote_tid, 1,1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_eval_server (pvm_upkint)",
                "Can't allocate memory!");
  #endif

  /* Hashing des Tid */
  hashnum = tid % PLI_PRIME;

  /* Suchen in Liste */
  nextelement = proclist[hashnum];
  prevelement = NULL;
  #ifdef NETDEBUG
    pvm_perror("plp_eval_server g: vor while");
  #endif
  while(nextelement != NULL)
  {
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server g: in while");
    #endif
    if (nextelement->parent_tid == tid &&
                nextelement->number == 0)
    {
      #ifdef NETDEBUG
        pvm_perror("plp_eval_server g: tid und number");
      #endif
      if (nextelement->partition_tid == remote_tid)
      {
        #ifdef NETDEBUG
```

```
  pvm_perror("plp_eval_server g: partition_tid");
      #endif
      /* Gefunden-> aus Liste loeschen und
                  zuruecksenden */
      gefunden = PBP_TRUE;
      info = pvm_initsend(plp_encoding);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plp_eval_server (pvm_initsend)",
                  "Can't allocate memory!");
      #endif

      plp_pack_obj(nextelement->obj);

      info = pvm_send(tid, 5);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plp_eval_server (pvm_send)",
            "Pvmd is not responding!");
      #endif

      if (prevelement == NULL)
      {
        proclist[hashnum] = nextelement->next;
      }else{
        prevelement->next = nextelement->next;
      }
      pbo_kill(nextelement->obj);
      free(nextelement);
      #ifdef NETDEBUG
        pvm_perror("plp_eval_server g: vor break");
      #endif
      break;
    }
  }
  prevelement = nextelement;
  nextelement = nextelement->next;
}
```

```
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server g: nach while");
    #endif
    /* Nicht gefunden -> in Liste der wartenden
                       Prozesse einfuegen */
    if (gefunden == PBP_FALSE)
    {
      #ifdef NETDEBUG
pvm_perror("plp_eval_server g: nicht gefunden neues Element");
      #endif
      /* Allokation eines neuen Listenelementes */
      if ((newelement =
(Pli_ProcHash_List *)malloc(sizeof (Pli_ProcHash_List)))==NULL)
        plp_error("plp_eval_server (malloc)",
                  "Cannot allocate memory!");

      /* Belegung der Variablen */
      newelement->parent_tid = tid;
      newelement->number = -1;          /* obj ist leer */
      newelement->obj = NULL;
      newelement->partition_tid = remote_tid;

      newelement->next = proclist[hashnum];
      proclist[hashnum] = newelement;
    }
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server g: ende");
    #endif
    break;
  case 'i':
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server i");
    #endif
    info = pvm_upkint(&length, 1,1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_eval_server (pvm_upkint)",
                    "Can't allocate memory!");
```

```
#endif

if ((objectstr =
(char *)malloc(sizeof (char)*(length+1))) == NULL)
    plp_error("plp_eval_server (malloc)",
               "Cannot allocate memory!");

info = pvm_upkstr(objectstr);  /* unpack the retobj */
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_eval_server (pvm_upkstr)",
                   "Can't allocate memory!");
#endif

object = pli_scan(&objectstr);
free(objectstr);


/* creating a new tuple space */
newTs =
  (Pli_Tuple_Space *)malloc(sizeof(Pli_Tuple_Space));
if(newTs == NULL)
  plp_error("plo_eval_server (malloc)",
                     "not enough memory");

newTs->atom = object;
pbp_set_use(newTs->atom);

newTs->atom_str = NULL;
newTs->max_partitions = 0;
newTs->hash_mode = 0;

info = pvm_upkint(tids, 20,1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_eval_server (pvm_upkint)",
                 "Can't allocate memory!");
#endif
```

```
    for (i=0; i< 20; i++)
      newTs->tids[i] = tids[i];

      /* putting newTs in the list of all tuple spaces */
    newTs->next = allTupleSpaces;
    allTupleSpaces = newTs;
    break;

  case 'o':
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server o");
    #endif
  case 'c':
    info = pvm_upkint(&length, 1,1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_eval_server (pvm_upkint)",
                          "Can't allocate memory!");
    #endif

    if ((objectstr =
       (char *)malloc(sizeof (char)*(length+1))) == NULL)
        plp_error("plp_eval_server (malloc)",
                   "Cannot allocate memory!");

    info = pvm_upkstr(objectstr);  /* unpack the atom */
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_eval_server (pvm_upkstr)",
                          "Can't allocate memory!");
    #endif

    object = pli_scan(&objectstr);
    free(objectstr);

    if (operation == 'c')
    {
```

```
     /* in Liste finden und Tupel loeschen */
    ts = plo_find_ts(object, PLI_CLEAR);
  }else{
     /* aus Liste loeschen */
    ts = plo_find_ts(object, PLI_REMOVE);
  }

  info = pvm_initsend(plp_encoding);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_eval_server (pvm_initsend)",
                   "Can't allocate memory!");
  #endif

  if (ts == NULL)
  {
    info = pvm_pkstr("z");
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_eval_server (pvm_pkstr)",
                   "Can't allocate memory!");
    #endif
  }else{
    info = pvm_pkstr("x");
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_eval_server (pvm_pkstr)",
                     "Can't allocate memory!");
    #endif
  }
  info = pvm_send(tid, 5);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_eval_server (pvm_send)",
                    "Pvmd is not responding!");
  #endif

  i=0;
```

```
    while(ts->tids[i] != 0 || i == 20)
    {
      /* hashing */
      hashnum = ts->tids[i] % PLI_PRIME;
      /* aktive Tupel in Liste suchen */
      nextelement = proclist[hashnum];
      prevelement = NULL;
      while(nextelement != NULL)
      {
        if (nextelement->number == 1 &&
          nextelement->partition_tid == tids[i])
        {
          if (prevelement == NULL)
          {
            proclist[hashnum] = nextelement->next;
          }else{
            prevelement->next = nextelement->next;
          }
          killelement = nextelement;
          nextelement = nextelement->next;
          free(killelement);
        }else{
        prevelement = nextelement;
        nextelement = nextelement->next;
        }
      }
      i++;
    }
    if (operation == 'o') free(ts);
    break;
  case 's':
    #ifdef NETDEBUG
      pvm_perror("plp_eval_server s");
    #endif
    info = pvm_upkint(&length, 1,1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_eval_server (pvm_upkint)",
```

```
                  "Can't allocate memory!");
#endif

if ((objectstr =
 (char *)malloc(sizeof (char)*(length+1))) == NULL)
    plp_error("plp_eval_server (malloc)",
                "Cannot allocate memory!");

info = pvm_upkstr(objectstr);  /* unpack the retobj */
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_eval_server (pvm_upkstr)",
                 "Can't allocate memory!");
#endif

object = pli_scan(&objectstr);
free(objectstr);
ts = plo_find_ts(object, PLI_FIND);

info = pvm_initsend(plp_encoding);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_eval_server (pvm_initsend)",
               "Can't allocate memory!");
#endif
  if (ts == NULL)          /* ts_invalid_id */
  {
  info = pvm_pkint(empty_tids, 20,1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_eval_server (pvm_pkint)",
                 "Can't allocate memory!");
  #endif
}else{
  info = pvm_pkint(ts->tids, 20,1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_eval_server (pvm_pkint)",
```

```
                        "Can't allocate memory!");
          #endif
        }
        info = pvm_send(tid, 5);
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plp_eval_server (pvm_send)",
                      "Pvmd is not responding!");
        #endif
        break;
      case 'e':
        #ifdef NETDEBUG
          pvm_perror("plp_eval_server e");
        #endif
        pvm_exit();
        exit(0);
        break;
      default:
          plp_error("plp_eval_server (switch)",
                       "Wrong char in message");
      }
    }
  }
```

### plp_partition

⟨*Exportierte P-Operationen*⟩+≡

```
  extern void plp_partition(int argc, char *argv[]);
```

**Beschreibung:** Die Funktion steuert den Kontrollfluß des als Partition gestarteten Prozesses. Zuerst werden die Parameter eingelesen und überprüft. Dann wird der Kreislauf aus Tupelraumoperation empfangen, Operation ausführen, Antwort versenden gestartet, der erst mit der Terminierungsnachricht abgebrochen wird (Event-Steuerung).

**Ausnahmen:**

**Probleme:**

**Abhängigkeiten:**

⟨*Implementierung der Operationen*⟩+≡

```
void plp_partition(int argc, char *argv[])
{
  extern Ple_Process plp_parent_id; /* The tid of my parent */
  extern int plp_encoding;
  char *my_atom;                 /* My tuplespace atom */
  Pli_Tuple_Space *myTs;          /* my tuple space */
  char *buffer;
  int bufid;      /* The id of the new active receive buffer */
  char operation; /* What I have to do.  (fetch, meet, deposit) */
  char *objectstr;  /* Template or Tuple converted to string */
  int info;            /* Error values */
  int sender_tid;     /* the taskid of the sender */
  Pbp_Object *tuple;
  Pli_Op_Type optype;       /* parameter to plo_match_in_ts */
  Pbp_Object *newtup = pbp_omega;/* parameter to plp_match_in_ts */
  int tempnum;        /* number of the matching template */
  Ple_Template *temps;         /* help pointer for the templates */
  Ple_Template *temp;          /* help pointer for the templates */
  Ple_Temp_Comp *comp;       /* help pointer for the components */
  Pbp_Exit *gps_return_status;/* dummy parameter for not used
                                  handler */
  Pbp_Env *gps_return_address;    /* dummy parameter for not
                               used handler */
  int i;       /* index */
  int length;
  Pli_Hash_List *kill, *next;/*pointers to members of a hashlist*/
```

```
char *tmp;
int partition_id;  /* deposit the new tuple to this partition */

pvm_joingroup("ts");

buffer = (char *)calloc(80, sizeof(char));
if (buffer == NULL)
  plp_error("plp_partition (calloc)",
                       "Can't allocate memory!!");

sprintf(buffer, "%s", argv[2]);

my_atom = (char *)calloc(strlen(buffer)+1, sizeof(char));
if (my_atom == NULL)
  plp_error("plp_partition (calloc)",
                     "Can't allocate memory!!");

sprintf(my_atom, "%s", buffer);

/* create the tuplespace */
myTs = (Pli_Tuple_Space *)calloc(1,sizeof(Pli_Tuple_Space));
if(myTs == NULL)
  plp_error("plp_partition (calloc)",
                     "Can't allocate memory!!");

myTs->pendprocs = NULL;

info = pvm_recv(pvm_parent(), 1);
switch (info){
case PvmBadParam :
  plp_error("plp_partition", "Wrong parameters in pvm_recv");
  break;
case PvmSysErr :
  plp_error("plp_partition", "Pvmd is not responding");
  break;
}

info = pvm_upkint(myTs->tids, 20, 1);
```

```
#ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_find_ts (pvm_upkint)",
                    "Can't allocate memory!");
#endif

info = pvm_upkint(&(myTs->hash_mode), 1, 1);
#ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_find_ts (pvm_upkint)",
                    "Can't allocate memory!");
#endif

for(i = 0; i < PLI_PRIME; i++)
  {
  myTs->hashtable[i] = NULL;
}

#ifdef NETDEBUG
  pvm_perror("plp_partition: Go in event circle");
#endif

free(buffer);   /* I don`t need a buffer anymore */

/* parameter are ok, go into event-circle */
while(1)
{
    bufid = pvm_recv(-1, 1); /* I'll receive all messages */
  switch (bufid){
  case PvmBadParam :
    plp_error("plp_partition (recv)",
                  "Wrong parameters in pvm_recv");
    break;
  case PvmSysErr :
    plp_error("plp_partition (recv)",
                       "Pvmd is not responding");
    break;
  }
```

```
info = pvm_upkint(&sender_tid, 1, 1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_partition (pvm_upkint)",
                 "Message can't be decoded!");
#endif

info = pvm_upkstr(&operation);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_partition (pvm_upkint)",
             "Message can't be decoded!");

  pvm_perror("plp_partition: vor switch");
#endif
switch (operation){
case 'd':
  #ifdef NETDEBUG
    pvm_perror("plp_partition: deposit");
  #endif
  info = pvm_upkint(&length, 1,1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_partition (pvm_upkint)",
                 "Message can't be decoded!");
  #endif

  if ((objectstr =
   (char *)malloc(sizeof (char)*(length+1))) == NULL)
    plp_error("plp_partition (malloc)",
                 "Cannot allocate memory!");

  info = pvm_upkstr(objectstr);
  #ifdef NETDEBUG
    if (info < 0)
    plp_error("plp_partition (operation)",
                   "Message can't be decoded!");
```

```
  #endif

  tmp =objectstr;
  tuple = pli_scan(&objectstr);
  pbp_set_use(tuple);
  free(tmp);
   /* Tupel in Tupelraum einfuegen */
  plo_add_tuple(tuple,myTs);
  #ifdef NETDEBUG
    pvm_perror("plp_partition: leave deposit");
  #endif
  break;
 case 'f':
  #ifdef NETDEBUG
    pvm_perror("plp_partition: fetch");
  #endif
  optype = PLI_FETCH;
  temps = plp_unpack_tl(optype);
  #ifdef NETDEBUG
    pvm_perror("plp_partition: nach unpack_tl");
  #endif
  tempnum = plo_match_in_ts (myTs, temps, &newtup, &optype);
  #ifdef NETDEBUG
    pvm_perror("plp_partition: nach match_in_ts");
  #endif
  /* blockierende Anfrage und nichts gefunden */
  if (operation == 'F' && tempnum == 0)
  {
    plp_wait(temps, myTs, optype, sender_tid);
    break;    /* switch verlassen und naechste
                 Anfrage bearbeiten */
  }

  info = pvm_initsend(plp_encoding);

  #ifdef NETDEBUG
    if (info < 0)
    plp_error("plp_partition (pvm_initsend)",
```

```
                     "Error in pvm_initsend!");

    pvm_perror("plp_partition: nach initsend");
  #endif
  info = pvm_pkint(&tempnum, 1, 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_partition (pvm_pkint)",
                      "Can't allocate memory!");

    pvm_perror("plp_partition: nach pkint");
  #endif
  if (tempnum != 0)
  {
    #ifdef NETDEBUG
      pvm_perror("plp_partition: tempnum != 0");
    #endif
    temp = temps;
    for (i=1; i < tempnum; i++)
    {
      #ifdef NETDEBUG
        pvm_perror("plp_partition: in for loop");
      #endif
      temp = temp->next;
    }
    comp = temp->comps;

    #ifdef NETDEBUG
      pvm_perror("plp_partition: vor for arity");
    #endif
    for (i=1; i <= temp->arity; i++)
    {
      #ifdef NETDEBUG
        pvm_perror("plp_partition: in for arity");
      #endif
        if (comp->type == PLE_FORMAL)
      {
        #ifdef NETDEBUG
```

```
          pvm_perror("plp_partition: nach if FORMAL");
        #endif
        if (comp->comp.formal.lvalue == NULL)
        {
          #ifdef NETDEBUG
pvm_perror("plp_partition: comp.formal.lvalue == NULL");
          #endif
          length = 1;
          info = pvm_pkint(&length, 1, 1);
          #ifdef NETDEBUG
          if (info < 0)
            plp_error("plp_partition (pvm_pkint)",
                      "Can't allocate memory!");
          #endif

          info = pvm_pkstr("n");
          #ifdef NETDEBUG
          if (info < 0)
            plp_error("plp_partition (pvm_pkstr)",
                     "Can't allocate memory!");
          #endif
        }else{
          #ifdef NETDEBUG
pvm_perror("plp_partition:comp.formal.lvalue != NULL");
          #endif
           /* pack the formal */
            plp_pack_obj(*comp->comp.formal.lvalue);
        }
      }
      comp = comp->next;
      #ifdef NETDEBUG
        pvm_perror("plp_partition: in for arity II");
      #endif
    }
  }
  #ifdef NETDEBUG
    pvm_perror("plp_partition: vor send");
  #endif
```

```
      info = pvm_send(sender_tid, 1);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plp_partition (pvm_send)",
                        "Pvmd is not responding!");

        pvm_perror("plp_partition: nach send");
      #endif
      plt_kill_temps(temps);
      #ifdef NETDEBUG
        pvm_perror("plp_partition: end of fetch");
      #endif
      break;
    case 'M':
     case 'm':
    case 'F':
      optype = PLI_MEET;
      temps = plp_unpack_tl(optype);
      tempnum = plo_match_in_ts (myTs, temps, &newtup, &optype);
      #ifdef NETDEBUG
        pvm_perror("plp_partition: M: nach match_in_ts");
      #endif
        /* blockierende Anfrage und nichts gefunden */
      if ((operation == 'M' || operation == 'F') &&
               tempnum == 0)
      {
        #ifdef NETDEBUG
          pvm_perror("plp_partition: M: vor wait");
        #endif
        plp_wait(temps, myTs, optype, sender_tid);
        break;      /* switch verlassen und naechste
                        Anfrage bearbeiten */
      }
      #ifdef NETDEBUG
        pvm_perror("plp_partition: M: vor initsend");
      #endif
      info = pvm_initsend(plp_encoding);
      #ifdef NETDEBUG
```

```
if (info < 0)
  plp_error("plo_partition (pvm_initsend)",
                "Can't allocate memory!");
#endif

info = pvm_pkint(&tempnum, 1, 1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_partition (pvm_pkint)",
                "Can't allocate memory!");
#endif

if (operation == 'F' || operation == 'M')
    /* nur tempnum uebertragen */
{
  info = pvm_send(sender_tid, 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_partition (pvm_send)",
             "Pvmd is not responding!");
  #endif
  break;
}
/* ab hier nur noch fuer nicht-blockierenden Aufruf */
if (tempnum != 0)
{
  temp = temps;
  for (i=1; i < tempnum; i++)
  {
    temp = temp->next;
  }
    comp = temp->comps;
  for (i=1; i <= temp->arity; i++)
  {
      if (comp->type == PLE_FORMAL)
    {
      if (comp->comp.formal.lvalue == NULL)
      {
```

```
         length = 1;
        info = pvm_pkint(&length, 1, 1);
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plp_partition (pvm_pkint)",
                    "Can't allocate memory!");
        #endif

        info = pvm_pkstr("n");
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plp_partition (pvm_pkstr)",
                  "Can't allocate memory!");
        #endif
      }else{
        /* pack the formal */
        plp_pack_obj(*comp->comp.formal.lvalue);
      }
    }
    comp = comp->next;
  }
  if(optype == PLI_MEETINTO)
    {
    partition_id = pli_hash_tup_general(newtup,myTs);
    if (partition_id == plp_main_id)
    {
        /* now add the changed tuple */
      plo_add_tuple(newtup, myTs);
    }else{
      /* deposit to other partition */
      info = pvm_initsend(plp_encoding);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plp_partition (pvm_initsend)",
                      "Can't allocate memory!");
      #endif

      info = pvm_pkint(&plp_main_id, 1, 1);
```

```
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plp_partition (pvm_pkint)",
                    "Can't allocate memory!");
        #endif

        info = pvm_pkstr("d");
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plp_partition (pvm_pkstr)",
                    "Can't allocate memory!");
        #endif

        plp_pack_obj(newtup);

        info = pvm_send(partition_id, 1);
        #ifdef NETDEBUG
        if (info < 0)
          plp_error("plp_eval_server (pvm_send)",
                    "Pvmd is not responding!");
        #endif
      }
    }
  }
  info = pvm_send(sender_tid, 1);
  #ifdef NETDEBUG
  if (info == PvmSysErr)
    plp_error("plp_partition (pvm_send)",
              "Pvmd is not responding!");
  #endif
  break;
case 'c':
  #ifdef NETDEBUG
    pvm_perror("plp_partition : ClearTs");
  #endif
  /* alle Tupel loeschen */
  for(i = 0; i < PLI_PRIME; i++)
    {
```

```
          kill = (myTs->hashtable)[i];
          while(kill != NULL)
          {
           next = kill->next;
           pbo_kill(kill->tuple);  /* killing a tuple */
           free(kill);
           kill = next;
         }
       }
       #ifdef NETDEBUG
         pvm_perror("plp_partition : ClearTs Ende");
       #endif
       break;
     case 't':
       #ifdef NETDEBUG
         pvm_perror("plp_partition : pendprocs loeschen ");
       #endif
       plq_kill_task(&(myTs)->pendprocs, sender_tid);
       #ifdef NETDEBUG
         pvm_perror("plp_partition : pendprocs geloescht ");
       #endif
       break;
     case 'k':
       pvm_lvgroup("ts");
       #ifdef NETDEBUG
         pvm_perror("Ende von plp_partition");
       #endif
       pvm_exit();
       exit(0);
       break;
     default:
       plp_error("plp_partition (loop)", "Wrong operation!");
     }
    }
}
```

### plp_pack_obj

⟨*Exportierte P-Operationen*⟩+≡

```
extern void plp_pack_obj(Pbp_Object *object);
```

**Beschreibung:** Pack an object for pvm.

**Ausnahmen:**

**Probleme:**

**Abhängigkeiten:** pvm

⟨*Implementierung der Operationen*⟩+≡

```
void plp_pack_obj(Pbp_Object *object)
{
 char *string; /* string that represents the object */
 Pbp_Env cur_env = (Pbp_Env)0;  /* Zeiger auf die zuletzt
                          kopierte Umgebung */
 int obj_cnt_ind = ENV_NUM_INDEX;
           /* Index von to_freeze fuer die Anzahl der zu */
             /* kopierenden Objekte in einer Umgebung */
 int cur_index = ENV_FIRST_ITEM;
             /* Index von to_freeze fuer den Index des zu  */
             /* kopierenden Objekts in der Umgebung.      */
 int i,j;      /* Indizes zur Iteration ueber to_freeze.    */
 int env_index;   /* Index des zu kopierenden Objekts in der
                  Umgebung.*/
 Pbp_Object *src;      /* Zeiger auf das zu kopierende Objekt.*/
 Pbp_Env source_env;   /* Die Umgebung */
 Pbu_Info *info = pbu_info(object);
                          /* statische Informationen des objects */
 int err;                /* Error varibale for pvm functions */
 char *uid;
 int length;
 char *objstr;

  #ifdef NETDEBUG
    pvm_perror("plp_pack_obj: BEGIN");
  #endif
  switch(pbp_type (object))
  {
    case PBP_OM:
    case PBP_BOOLEAN:
```

```
case PBP_INTEGER:
case PBP_REAL:
case PBP_ATOM:
case PBP_STRING:
case PBP_TUPLE:
case PBP_SET:
  #ifdef NETDEBUG
    pvm_perror("plp_pack_obj: normaler Typ");
  #endif
  string = pli_ots(object, NULL);
  #ifdef NETDEBUG
    pvm_perror(string);
  #endif
  length = strlen(string);
    err = pvm_pkint(&length,1,1);
  #ifdef NETDEBUG
  if (err < 0)
    plp_error("plp_pack_obj (pvm_pkint)",
                  "Can't allocate memory!");
  #endif

  err = pvm_pkstr(string);
  #ifdef NETDEBUG
  if (err < 0)
    plp_error("plp_pack_obj (pvm_pkstr)",
                   "Can't allocate memory!");
  #endif

  free(string);
  break;
case PBP_MODTYPE:
case PBP_INSTANCE:
case PBP_HANDLER:
case PBP_MODULE:
case PBP_PROCEDURE:
case PBP_PROCESS:
  plp_error("plp_pack_obj",
          "Falscher Typ! Nicht geclosured?");
```

```
    break;
  case PBP_FUNCTION:
    #ifdef NETDEBUG
      pvm_perror("plp_pack_obj: Typ:function");
    #endif
      uid = pbu_uid(object);
    length = strlen(uid);

    err = pvm_pkint(&length,1,1);
    #ifdef NETDEBUG
    if (err < 0)
      plp_error("plp_pack_obj (pvm_pkint)",
                           "Can't allocate memory!");
    #endif

    err = pvm_pkstr(uid); /* pack uid (name) */
    #ifdef NETDEBUG
    if (err < 0)
      plp_error("plp_pack_obj (pvm_pkstr)",
                           "Can't allocate memory!");
    #endif

    source_env = pbu_frozen(object);
    #ifdef NETDEBUG
      pvm_perror("plp_pack_obj: nach frozen");
    #endif
    for (i=1; i<=info->to_freeze[ENV_COUNT_INDEX]; i++)
    {
          /* Markiere die Umgebung als besucht */
      pbu_push_env(source_env);

            /* Iteriere ueber das source_env und
        kopiere die Objekte, deren */
            /* Anzahl und Indizes in to_freeze gespeichert sind.*/

      for (j=1; j<=info->to_freeze[obj_cnt_ind]; j++)
      {
    /* Index des zu kopierenden Objekts in der Umgebung. */
```

```
        env_index = info->to_freeze[cur_index];
        src = pbu_env_obj(source_env[env_index]);

        switch(pbp_type(src)){
           case PBP_PROCEDURE:
           case PBP_HANDLER:
           case PBP_MODULE:
           case PBP_MODTYPE:
           case PBP_INSTANCE:
           case PBP_FUNCTION:
        plp_error("plp_pack_obj (pvm_pkstr)",
                "Bad type in env!");
             break;
      default:
        objstr = pli_ots(src, NULL);
        length = strlen(objstr);
        #ifdef NETDEBUG
pvm_perror("plp_pack_obj: Umgebungsobjekt einpacken");
        #endif
        err = pvm_pkint(&length, 1, 1);
        #ifdef NETDEBUG
        if (err < 0)
            plp_error("plp_pack_obj (pvm_pkint)",
                     "Can't allocate memory!");
        #endif

            err = pvm_pkstr(objstr);
        #ifdef NETDEBUG
          if (err < 0)
            plp_error("plp_pack_obj (pvm_pkstr)",
                     "Can't allocate memory!");
        #endif

        free(objstr); /* free the packed objstr */
      }
        cur_index++;

      } /* end for j=1 ... */
```

```
                /* Aktualisiere obj_cnt_ind und cur_index */
          cur_index++;
          obj_cnt_ind = cur_index++;

              /* Gehe zur naechsten Umgebung */
          source_env = pbu_env_sl(source_env);

      } /* end for i=1 ... */
      break;
    }
  }
```

**plp_unpack_tl**

⟨*Interne P-Operationen*⟩≡

```
  extern Ple_Template *plp_unpack_tl(Pli_Op_Type optype);
```

**Beschreibung:** Entpackt eine Liste von Templates.

**Ausnahmen:**

**Probleme:**

**Abhängigkeiten:** pvm

⟨*Implementierung der Operationen*⟩+≡

```
Ple_Template *plp_unpack_tl(Pli_Op_Type optype)
{
 char string; /* which component? */
 int arity;
 int i;
 Ple_Template *temp; /* head of the list of templates */
 Ple_Template *tempfirst = NULL;
 Ple_Template *templast = NULL; /* pointer to temporary last
                               template */
 Ple_Temp_Comp *compfirst = NULL; /* pointer to first component */
 Ple_Temp_Comp *comp; /* pointer to created component */
 Ple_Temp_Comp *complast = NULL; /* pointer to last component */
 int info; /* error values */
 Pbp_Bool_Type next; /* is there another template to unpack? */

  #ifdef NETDEBUG
    pvm_perror("plp_unpack_tl: BEGIN");
  #endif

  /* creating and initializing the new template */
  if ((tempfirst =
    (Ple_Template *)malloc(sizeof(Ple_Template))) == NULL)
    plp_error("plp_unpack_tl", "Cannot allocate memory");

  templast = tempfirst;
  do {
    if ((templast->dL =
      (Pbp_Env)malloc(sizeof(Pbp_Env))) == NULL)
      plp_error("plp_unpack_tl (malloc)",
                    "Cannot allocate memory!");
```

```
if ((templast->RetStat =
  (Pbp_Exit*)malloc(sizeof(Pbp_Exit))) == NULL)
  plp_error("plp_unpack_tl (malloc)",
               "Cannot allocate memory!");

*templast->RetStat = PBP_RESUME;
if ((templast->RetAd =
   (Pbp_Env*)malloc(sizeof(Pbp_Env))) == NULL)
  plp_error("plp_unpack_tl (malloc)",
               "Cannot allocate memory!");

    /* unpack the arity */
  info = pvm_upkint(&arity,1,1);
#ifdef NETDEBUG
  if (info < 0)
  plp_error("plp_unpack_tl (pvm_upkint)", "Error occurs!");
#endif

  templast->arity = arity;
templast->condition = plp_unpack_func();
#ifdef NETDEBUG
  pvm_perror("plp_unpack_tl: nach unpack_func");
#endif

if (arity == 0) return tempfirst;   /* Template is empty */

  if ((compfirst =
 (Ple_Temp_Comp *)malloc(sizeof(Ple_Temp_Comp))) == NULL)
  plp_error("plp_unpack_tl (malloc)",
                 "Cannot allocate memory!");

  complast = compfirst;
/* unpacking the first component */
plp_unpack_comp(complast, optype);

for (i=2; i <= arity ;i++)
  {
```

```
  if ((complast->next =
   (Ple_Temp_Comp *)malloc(sizeof(Ple_Temp_Comp))) == NULL)
        plp_error("plp_unpack_tl (malloc)",
                 "Cannot allocate memory!");

  complast = complast->next;
   /* unpacking the next component */
  plp_unpack_comp(complast, optype);
  }
complast->next = NULL;

templast->comps = compfirst;
info = pvm_upkstr(&string);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_unpack_tl (pvm_upkstr)", "Error occurs!");
#endif

switch(string){
case 'n':
  next = PBP_TRUE;
  if ((templast->next =
   (Ple_Template *)malloc(sizeof(Ple_Template))) == NULL)
    plp_error("plt_cre_temp", "Cannot allocate memory");

  templast = templast->next;
  break;
  case 'q' :
  next = PBP_FALSE;
  templast->next = NULL;
  break;
default :
  #ifdef NETDEBUG
    pvm_perror(&string);
  #endif

   plp_error("plp_unpack_tl (switch)",
           "Wrong character in message!");
```

```
      }
    } while (next == PBP_TRUE);
    #ifdef NETDEBUG
      pvm_perror("plp_unpack_tl: ENDE");
    #endif
    return tempfirst;
  }
```

### plp_unpack_comp

⟨*Exportierte P-Operationen*⟩+≡

```
  extern void plp_unpack_comp(Ple_Temp_Comp *comps,
                              Pli_Op_Type optype);
```

**Beschreibung:** Entpackt Komponenten von Templates

**Ausnahmen:**

**Probleme:**

**Abhängigkeiten:** pvm

⟨*Implementierung der Operationen*⟩+≡

```
void plp_unpack_comp(Ple_Temp_Comp *comps, Pli_Op_Type optype)
{
 int info; /* error variable for pvm functions */
 char string;
 char *string2; /* string to unpack */
 char *tmp;
 Pbp_Object **lvalue;
 int length;

  info = pvm_upkstr(&string);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_unpack_comp (pvm_upkstr)", "Error occurs!");
  #endif

  switch(string){
    case 'a':
    #ifdef NETDEBUG
      pvm_perror("plp_unpack_comp: actual");
    #endif
    info = pvm_upkint(&length, 1,1);
    #ifdef NETDEBUG
    if (info < 0)
      plp_error("plp_unpack_comp (pvm_upkint)",
                   "Can't allocate memory!");
    #endif

    if ((string2 =
       (char *)malloc(sizeof (char)*length+1)) == NULL)
      plp_error("plp_unpack_comp (malloc)",
```

```
                    "Cannot allocate memory!");

    info = pvm_upkstr(string2);
    #ifdef NETDEBUG
      if (info < 0)
      plp_error("plp_unpack_comp (pvm_upkstr)",
                          "Error occurs!");
    #endif

    comps->type = PLE_ACTUAL;
      tmp = string2;
    comps->comp.actual = pli_scan(&string2);
    pbp_set_use(comps->comp.actual);
    free(tmp);
    #ifdef NETDEBUG
      pvm_perror("plp_unpack_comp: actual raus");
    #endif

    break;
  case 'o':
    #ifdef NETDEBUG
      pvm_perror("plp_unpack_comp: formal o");
    #endif
    comps->type = PLE_FORMAL;
    comps->comp.formal.lvalue = NULL;
    if (optype == PLI_MEET)
    {
      comps->comp.formal.into = plp_unpack_func();
      if ((comps->comp.formal.dL =
          (Pbp_Env)malloc(sizeof(Pbp_Env))) == NULL)
            plp_error("plp_unpack_comp",
                      "Cannot allocate memory!");

       if ((comps->comp.formal.RetStat =
          (Pbp_Exit*)malloc(sizeof(Pbp_Exit))) == NULL)
        plp_error("plp_unpack_comp",
                      "Cannot allocate memory!");
```

```
      *comps->comp.formal.RetStat = PBP_RESUME;
      if ((comps->comp.formal.RetAd =
            (Pbp_Env*)malloc(sizeof(Pbp_Env))) == NULL)
          plp_error("plp_unpack_comp",
                     "Cannot allocate memory!");
    }else{
      comps->comp.formal.into = NULL;
    }
    break;
  case 'f':
    #ifdef NETDEBUG
      pvm_perror("plp_unpack_comp: formal f");
    #endif
    comps->type = PLE_FORMAL;
    if ((lvalue = (Pbp_Object**)malloc(sizeof(char))) == NULL)
      plp_error("plp_unpack_comp (malloc)",
                       "Cannot allocate memory!");

    *lvalue = pbp_omega;
    comps->comp.formal.lvalue = lvalue;
    if (optype == PLI_MEET)
    {
      comps->comp.formal.into = plp_unpack_func();
      if ((comps->comp.formal.dL =
           (Pbp_Env)malloc(sizeof(Pbp_Env))) == NULL)
            plp_error("plp_unpack_comp",
                   "Cannot allocate memory!");

       if ((comps->comp.formal.RetStat =
           (Pbp_Exit*)malloc(sizeof(Pbp_Exit))) == NULL)
        plp_error("plp_unpack_comp",
                   "Cannot allocate memory!");

      *comps->comp.formal.RetStat = PBP_RESUME;
      if ((comps->comp.formal.RetAd =
         (Pbp_Env*)malloc(sizeof(Pbp_Env))) == NULL)
          plp_error("plp_unpack_comp",
                   "Cannot allocate memory!");
```

```
      }else{
        comps->comp.formal.into = NULL;
      }
      #ifdef NETDEBUG
        pvm_perror("plp_unpack_comp: formal raus");
      #endif
      break;
    default:
      plp_error("plp_unpack_comp (switch)", "Wrong string!");
      break;
      }
  }
```

### plp_unpack_func

⟨*Exportierte P-Operationen*⟩+≡

```
  extern Pbp_Object *plp_unpack_func();
```

**Beschreibung:** Unpack a function and give back a pointer to this function

**Ausnahmen:**

**Probleme:**

**Abhängigkeiten:** pvm

⟨*Implementierung der Operationen*⟩+≡

```
Pbp_Object *plp_unpack_func()
{
 Pbu_Info *info_part;
 Pbp_Object *object; /* the unpacked function */
 Pbp_Object *src; /* pointer to an object of the unpacking
                   environment */
 char *uid; /* the unique name of the function */
 char *objectstr; /* the string representing an object */
 char *tmp;   /* string zum freigeben des Speicherplatzes */
 Pbp_Env head_env = (Pbp_Env)0; /* Zeiger auf die Kopie der
                         Umgebung von source_env. */
 Pbp_Env cur_env = (Pbp_Env)0;  /* Zeiger auf die zuletzt
                    kopierte Umgebung */
 Pbp_Env source_env; /* the environment of the function */
 Pbu_Shared *shared;
 int obj_cnt_ind = ENV_NUM_INDEX;
               /* Index von to_freeze fuer die Anzahl der zu  */
                /* kopierenden Objekte in einer Umgebung      */
 int cur_index = ENV_FIRST_ITEM;
               /* Index von to_freeze fuer den Index des zu  */
               /* kopierenden Objekts in der Umgebung        */
 int i,j;      /* Indizes zur Iteration ueber to_freeze.     */
 int env_index;
 int info;     /* error variable for pvmfunctions */
 int length;

  #ifdef NETDEBUG
    pvm_perror("plp_unpack_func: vor upkstr");
  #endif
  info = pvm_upkint(&length, 1,1);
```

```
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_unpack_func (pvm_upkint)",
                     "Can't allocate memory!");
  #endif

  if ((uid = (char *)malloc(sizeof (char)*length+1)) == NULL)
    plp_error("plp_unpack_func (malloc)",
                         "Cannot allocate memory!");

  info = pvm_upkstr(uid); /* unpack uid (name) */
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_unpack_func (pvm_upkstr)", "Error occurs!");
  #endif

  if (strcmp(uid,"c") == 0)       /* no function */
  {
    #ifdef NETDEBUG
      pvm_perror("plp_unpack_func : no condition function");
    #endif
    return NULL;
  }
  #ifdef NETDEBUG
    pvm_perror(uid);
  #endif

  info_part = pbu_get_info(uid);
  if (info_part == NULL)
    plp_error("plp_unpack_func (pbu_get_info)",
                     "No function with this uid");

  for (i=1; i<=info_part->to_freeze[ENV_COUNT_INDEX]; i++)
  {
    source_env =
pbu_create_environment(PBU_ENV_OFFSET + info_part->pred->env_size,
                                    sizeof(Pbu_Env_Item * ));
```

```
    /* Iteriere ueber das source_env und kopiere die Objekte,
  deren Anzahl und Indizes in to_freeze gespeichert sind. */

   for (j=1; j<=info_part->to_freeze[obj_cnt_ind]; j++)
   {
 /* Index des zu kopierenden Objekts in der Umgebung. */
     env_index = info_part->to_freeze[cur_index];
     info = pvm_upkint(&length, 1,1);
     #ifdef NETDEBUG
     if (info < 0)
       plp_error("plp_unpack_func (pvm_upkint)",
                 "Can't allocate memory!");
     #endif

     if ((objectstr =
        (char *)malloc(sizeof(char)*length+1)) == NULL)
       plp_error("plp_unpack_func (malloc)",
             "Cannot allocate memory!");

     info = pvm_upkstr(objectstr);
     #ifdef NETDEBUG
     if (info < 0)
       plp_error("plp_unpack_func (pvm_upkstr)",
                           "Error occurs!");
     #endif

     tmp = objectstr;
     src = pli_scan(&objectstr);
     #ifdef NETDEBUG
       pvm_perror("plp_unpack_func: nach scan");
     #endif
     free(tmp);

     switch(pbp_type(src)){
     case PBP_PROCEDURE:
     case PBP_HANDLER:
     case PBP_MODULE:
     case PBP_MODTYPE:
```

```
    case PBP_INSTANCE:
    case PBP_FUNCTION:
      plp_error("plp_unpack_func",
        "env-object is procedure, handler or module");
      break;
    default:
      #ifdef NETDEBUG
        pvm_perror("plp_unpack_func : vor assign");
      #endif
      pbp_set_use(src);
      pbo_assign(&(pbu_env_obj(source_env[env_index])), src);
    }
    cur_index++;

  } /* end for j=1 ... */

    /* Aktualisiere obj_cnt_ind und cur_index */
  cur_index++;
  obj_cnt_ind = cur_index++;

        /* Setze head_env und cur_env */
  if (head_env == (Pbp_Env)0)
  {
       /* Beim ersten Mal */
    head_env = source_env;
    cur_env = source_env;
  }else{
          /* Verkette die kopierten Umgebungen */
    pbu_env_sl(cur_env) = source_env;
    cur_env = source_env;
  }

} /* end for i=1 ... */
object = pbu_new(PBP_FUNCTION, info_part, (Pbp_Env)0);
pbu_shared(object) = pbu_create_shared();
pbu_rc(object) = 1;
pbu_atom(object) = pba_newat();
pbu_frozen(object) = head_env;
```

```
    pbu_exp_list(object) = 0;

    return object;
  }
```

### plp_enable

⟨*Interne P-Operationen*⟩+≡

```
  extern void plp_enable(Pli_Pend_Proc *temp);
```

**Profil:** `proc` ist der zu aktivierende Prozeß.

**Beschreibung:** Sendet dem wartenden Prozeß die Nachricht.

**Ausnahmen:**

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:** LWP

⟨*Implementierung der Operationen*⟩+≡

```
void plp_enable (Pli_Pend_Proc *temps)
{
 extern int plp_encoding;
 Ple_Temp_Comp *comp;
 Ple_Template *temp;
 int length;
 int i,info;

  info = pvm_initsend(plp_encoding);
  #ifdef NETDEBUG
    if (info < 0)
    plp_error("plp_enable (pvm_initsend)",
                  "Can't allocate memory!");

    pvm_perror("plp_enable: nach initsend");
  #endif
  info = pvm_pkint(temps->ptempnum, 1, 1);
  #ifdef NETDEBUG
  if (info < 0)
    plp_error("plp_enable (pvm_pkint)",
                      "Can't allocate memory!");

    pvm_perror("plp_enable: nach pkint");
  #endif
  if (temps->ptempnum != 0)
  {
    #ifdef NETDEBUG
      pvm_perror("plp_enable: ptempnum != 0");
    #endif
```

```
temp = temps->temps;
  for (i=1; i < *temps->ptempnum; i++)
{
  #ifdef NETDEBUG
    pvm_perror("plp_enable: in for-Schleife");
  #endif
  temp = temp->next;
}
comp = temp->comps;
#ifdef NETDEBUG
  pvm_perror("vor for arity");
#endif
for (i=1; i <= temp->arity; i++)
{
  #ifdef NETDEBUG
    pvm_perror("plp_enable: in for arity");
  #endif
  if (comp->type == PLE_FORMAL)
   {
    #ifdef NETDEBUG
      pvm_perror("plp_enable: nach if FORMAL");
    #endif
    if (comp->comp.formal.lvalue == NULL)
    {
      #ifdef NETDEBUG
pvm_perror("plp_enable:comp.formal.lvalue == NULL");
      #endif
       length = 1;
       info = pvm_pkint(&length, 1, 1);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plp_enable (pvm_pkint)",
                     "Can't allocate memory!");
      #endif

      info = pvm_pkstr("n");
      #ifdef NETDEBUG
        if (info < 0)
```

```
            plp_error("plp_enable (pvm_pkstr)",
                      "Can't allocate memory!");
          #endif
        }else{
          #ifdef NETDEBUG
    pvm_perror("plp_enable:comp.formal.lvalue != NULL");
          #endif
           /* pack the formal */
            plp_pack_obj(*comp->comp.formal.lvalue);
        }
       }
       comp = comp->next;
      #ifdef NETDEBUG
        pvm_perror("plp_enable: in for arity II");
      #endif
    }
  }
  #ifdef NETDEBUG
    pvm_perror("plp_enable: vor send");
  #endif
  info = pvm_send(temps->proc, 1);
  #ifdef NETDEBUG
   if (info < 0)
    plp_error("plp_enable (pvm_send)",
                      "Pvmd is not responding!");

    pvm_perror("leaving plp_enable");
  #endif

  plt_kill_temps(temp);
   free(temps->ptempnum);
   free(temps);
}
```

### plp_min_load

$\langle$*Interne P-Operationen*$\rangle{+}{\equiv}$

```
extern Ple_Hostname_List *plp_min_load(int max_hosts);
```

**Profil:**

```
Rückgabewert
```

**Beschreibung:** Fürt das Loadbalancing durch.

**Ausnahmen:**

**Probleme:**

**Abhängigkeiten:**

⟨*Implementierung der Operationen*⟩+≡

```
Ple_Hostname_List *plp_min_load(int max_hosts)
{
 extern int plp_max_hosts;
 extern char *plp_last_host;
 double maximum = 0.5;
 int list_length = 0;     /* Laenge der Liste */
 Ple_Hostname_List *hostlist = NULL; /* Zeiger auf Liste */
 Ple_Hostname_List *newhost = NULL;  /* neues Element der Liste */
 int i;
 int info;
 int size;

  if (plp_last_host == NULL)
  {
    plp_last_host = (char *)calloc(64,sizeof(char));
    if (plp_last_host == NULL)
      plp_error("plp_min_load (calloc)",
       "Cannot allocate memory!");

    gethostname(plp_last_host, 64);
  }

  info = pvm_initsend(PvmDataDefault);
  #ifdef NETDEBUG
   if (info < 0)
    plp_error("plp_min_load (pvm_initsend)",
                       "Can't allocate memory!");
  #endif
```

```
info = pvm_pkint(&plp_main_id, 1, 1);
#ifdef NETDEBUG
if (info < 0)
  plp_error("plp_min_load (pvm_pkint)",
                         "Can't allocate memory!");
#endif

/* leider notwendig, da zum fruehen Zeitpunkt Gruppe nicht vorhanden */
while ((size = pvm_gsize("load")) < plp_max_hosts)
{
  sleep(1);  /* kuerzer gehts nicht? */
}

info = pvm_bcast("load",4);
if (info < 0)
  plp_error("plp_min_load", "Error in pvm_bcast!");

for (i=0; i < plp_max_hosts; i++)
{
  info = pvm_recv(-1, 4); /* I wait for the message from
                           the load agents */
    switch (info){
    case PvmBadParam :
    plp_error("plp_min_load", "Wrong parameters in pvm_recv");
    break;
    case PvmSysErr :
    plp_error("plp_min_load", "Pvmd is not responding");
    break;
    }

  if (list_length < max_hosts)
  {
    if ((newhost = calloc(1, sizeof(Ple_Hostname_List))) == NULL)
      plp_error("plp_min_load (calloc)",
                         "Can't allocate memory!");

    newhost->hostname = (char *)calloc(64,sizeof(char));
```

```
      if (newhost->hostname == NULL)
        plp_error("plp_min_load (calloc)",
                        "Cannot allocate memory!");

      info = pvm_upkstr(newhost->hostname);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plp_min_load (pvm_upkstr)", "Error occurs!");
      #endif

      info = pvm_upkdouble(&newhost->load, 1, 1);
      #ifdef NETDEBUG
      if (info < 0)
        plp_error("plp_min_load (pvm_upkdouble)",
                                 "Error occurs!");
      #endif

      newhost->next = NULL;

      if (hostlist == NULL)
      {   /* wenn Liste leer, einfach einfuegen */
        hostlist = newhost;
        list_length = 1;
      }else{              /* Liste nicht leer, einsortieren */
        if ((strcmp(newhost->hostname,plp_last_host) != 0) ||
           (newhost->load > maximum))
        {
          newhost->next = hostlist;
          hostlist = newhost;
          list_length++;
        }
      }
    }
  }
  strcpy(plp_last_host,hostlist->hostname);
  return hostlist;
}
```

### plp_wait

⟨*Interne P-Operationen*⟩+≡

```
extern void plp_wait(Ple_Template *temps, Pli_Tuple_Space *ts,
        Pli_Op_Type optype, int taskid);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die Nummer des matchenden Templates. |
| `temps` | ist die Liste der Templates. |
| `ts` | ist der Tupelraum, in den ein passendes Tupel eingefügt soll. |
| `optype` | gibt an, ob es sich um eine `meet`– oder `fetch`–Operation handelt. |

**Beschreibung:** `plp_wait` erzeugt einen neuen Eintrag in die Liste der wartenden Prozesse dieses Tupelraums und blockiert den Prozeß solange, bis von einem anderen Prozeß ein passendes Tupel in den Tupelraum eingefügt wurde. Dieser Prozeß weist den Formals ihre Werte zu, setzt auch tempnum und aktiviert `proc` wieder. Daraufhin braucht nur noch `tempnum` zurückgegeben werden.

**Ausnahmen:**

**Probleme:**

**Abhängigkeiten:** `plq_append_temp`

⟨*Implementierung der Operationen*⟩+≡

```
void plp_wait(Ple_Template *temps, Pli_Tuple_Space *ts,
              Pli_Op_Type optype, int taskid)
{
 Pli_Pend_Proc *NewPend = NULL; /* for the new entry in the
                         list of pending processes */
 Pli_Pend_Proc *hpend = NULL;
 int tempnum = 0;    /* for the selected template number */

  /*creating and initializing NewPend */
  if ((NewPend = malloc(sizeof(Pli_Pend_Proc))) == NULL)
    plp_error("plp_wait", "cannot allocate memory");

  NewPend->temps = temps;
  NewPend->optype = optype;
  NewPend->ptempnum = &tempnum;
  NewPend->proc = taskid;
  NewPend->next = NULL;   /* becomes the last entry (FIFO) */

  /* putting NewPend at the end of the list of pending processes */
```

```
  if(ts->pendprocs == NULL)
  {
    #ifdef NETDEBUG
      pvm_perror("plp_wait: pendprocs = NULL");
    #endif
    ts->pendprocs = NewPend;
  }else{
    #ifdef NETDEBUG
      pvm_perror("plp_wait: pendprocs != NULL");
    #endif

      hpend = ts->pendprocs;
    while(hpend->next != NULL)
      {
      #ifdef NETDEBUG
        pvm_perror("plp_wait: in while");
      #endif
      hpend = hpend->next;
      }
      hpend->next = NewPend;
  }
  #ifdef NETDEBUG
    pvm_perror("plp_wait: ende");
  #endif
}
```

### C.3.4 Lokale Variablen

Die Thread-ID für das Hauptprogramm:

⟨*Externe Variablen*⟩≡

```
extern Ple_Process plp_main_id;
            /* for the id of the main process */
```

⟨*Implementierung der externen Variablen*⟩≡

```
Ple_Process plp_main_id;
```

Die Thread-ID für den Vater:

⟨*Externe Variablen*⟩+≡
```
extern Ple_Process plp_parnt_id;
                /* for the id of the parent process */
```

⟨*Implementierung der externen Variablen*⟩+≡
```
Ple_Process plp_parent_id;
```

Der Name des Hauptprogrammes:

⟨*Externe Variablen*⟩+≡
```
extern char *plp_my_name;
```

⟨*Implementierung der externen Variablen*⟩+≡
```
char *plp_my_name;
```

⟨*Externe Variablen*⟩+≡
```
extern int plp_hash_mode;
```

⟨*Implementierung der externen Variablen*⟩+≡
```
int plp_hash_mode = 0;
```

⟨*Externe Variablen*⟩+≡
```
extern int plp_max_partitions;
```

⟨*Implementierung der externen Variablen*⟩+≡
```
int plp_max_partitions = 5;
```

⟨*Externe Variablen*⟩+≡
```
extern Pbp_Bool_Type plp_loadbalancing;
```

⟨*Implementierung der externen Variablen*⟩+≡
```
Pbp_Bool_Type plp_loadbalancing = PBP_FALSE;
```

⟨*Externe Variablen*⟩+≡
```
extern char *plp_last_host;
```

⟨*Implementierung der externen Variablen*⟩+≡
```
char *plp_last_host = NULL;
```

⟨*Externe Variablen*⟩+≡
```
extern int plp_eval_tid;
```

⟨*Implementierung der externen Variablen*⟩+≡
```
int plp_eval_tid;
```

⟨*Externe Variablen*⟩+≡
```
extern Pbp_Bool_Type plp_eval;
```

⟨*Implementierung der externen Variablen*⟩+≡
```
Pbp_Bool_Type plp_eval = PBP_FALSE;
```

Struct für die Parameter der Hosts in der virtuellen Maschine:

⟨*Externe Variablen*⟩+≡
```
extern struct pvmhostinfo *plp_hostp[15];
```

⟨*Implementierung der externen Variablen*⟩+≡
```
struct pvmhostinfo *plp_hostp[15];
```

⟨*Externe Variablen*⟩+≡
```
extern int plp_max_hosts;
```

⟨*Implementierung der externen Variablen*⟩+≡
```
int plp_max_hosts;
```

⟨*Externe Variablen*⟩+≡
```
extern int plp_encoding;
```

⟨*Implementierung der externen Variablen*⟩+≡

```
int plp_encoding;
```

⟨*Externe Variablen*⟩+≡

```
extern char plp_filename[12];
```

⟨*Implementierung der externen Variablen*⟩+≡

```
char plp_filename[12] = "~/netproset";
```

## C.3.5 Lokale Konstanten

Die folgenden Konstanten sind Indizes für den Array, der die Informationen über die einzufrierenden Komponenten bei der Closurebildung enthält. Der Array ist über `to_freeze` in `Pbu_Info` zugänglich.

#### Anzahl der zu kopierenden Umgebungen

`ToFreeze[ENV_COUNT_INDEX]` enthält die Anzahl der zu kopierenden Umgebungen.

⟨*Lokale Konstanten*⟩≡

```
#define ENV_COUNT_INDEX     0
```

#### Größe der ersten zu kopierenden Umgebungen

`ToFreeze[ENV_SIZE_INDEX]` enthält die Größe der zu kopierenden ersten Umgebung.

⟨*Lokale Konstanten*⟩+≡

```
#define ENV_SIZE_INDEX      1
```

#### Anzahl der zu kopierenden Objekte in der ersten Umgebungen

`ToFreeze[ENV_NUM_INDEX]` enthält die Anzahl der zu kopierenden Objekte in der ersten Umgebung.

⟨*Lokale Konstanten*⟩+≡

```
#define ENV_NUM_INDEX      2
```

#### Index des ersten zu kopierenden Objekts in der ersten Umgebungen

`ToFreeze[ENV_FIRST_ITEM]` enthält den Index des ersten zu kopierenden Objekts in der ersten Umgebungen.

$\langle$*Lokale Konstanten*$\rangle$+≡
```
#define ENV_FIRST_ITEM      3
```

### C.3.6 C Dateien

$\langle$*linda.h*$\rangle$≡
  $\langle$*Externe Variablen*$\rangle$
  $\langle$*Exportierte P-Operationen*$\rangle$

$\langle$*intLinda.h*$\rangle$≡
  $\langle$*Interne P-Operationen*$\rangle$

$\langle$*process.c*$\rangle$≡
  $\langle$*Importe der Quelltextdatei*$\rangle$
  $\langle$*Lokale Konstanten*$\rangle$
  $\langle$*Implementierung der externen Variablen*$\rangle$
  $\langle$*Implementierung der Operationen*$\rangle$

# C.4 Die Linda–Queues

## C.4.1 Einleitung

Dieses Modul enthält Routinen zur Behandlung von Warteschlangen der schwebenden Prozesse.

## C.4.2 Abhängigkeiten

⟨*Importe der Quelltextdatei*⟩≡

```
#include "proset/proset.h"
#include "proset/unit.h"
#include "proset/overloaded.h"
#include "proset/function.h"
#include "proset/handler.h"
#include "proset/module.h"
#include "proset/loop.h"
#include "proset/atom.h"
#include "proset/string.h"
#include "proset/set.h"
#include "proset/tuple.h"
#include "proset/io.h"

#include "linda/linda.h"
#include "../intLinda.h"
```

## C.4.3 Interne Operationen

### plq_check_meets

⟨*Interne Q-Operationen*⟩≡

```
Pli_Op_Type plq_check_meets(Pbp_Object **tup, Pli_Tuple_Space *ts);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist `PLI_MEETINTO`, falls `tup` verändert wurde, sonst `PLI_MEET`. |
| `tup` | ist die Adresse des einzufügenden Tupels. Falls es durch eine `meet`–Operation geändert wird, wird das neue Tupel zurückgegeben. |
| `ts` | ist der Tupelraum in den `tup` eingefügt werden soll. |

**Beschreibung:** `plq_check_meets` testet, ob `tup` mit einem `meet`-Template eines wartenden Prozesses matcht. Ist das der Fall wird der Prozeß aktiviert und der Eintrag in der Liste der wartenden Prozesse gelöscht. Wurde das Tupel dabei nicht verändert (keine `into`-Anweisung) wird weiter getestet.

**Ausnahmen:** keine

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:** `plq_find_temp`, `plp_enable`, `plt_kill_temps`

⟨*Implementierung der Operationen*⟩≡

```
Pli_Op_Type plq_check_meets(Pbp_Object **tup, Pli_Tuple_Space *ts)
{
  Pli_Op_Type optype = PLI_MEET;  /* if the value is PLE_MEETINTO
                           tup was changed */
  Pli_Pend_Proc *FoundTemp = NULL; /* the entry of the process
                      of a matching template */

  /* checking the list of pending processes;
     If a meet template is matching the formals and
        tempNum are set */
  while ((FoundTemp = plq_find_temp (&(ts->pendprocs),
               tup, &optype, ts->atom)) != NULL)
  {
    plp_enable(FoundTemp);  /* enabling the process of the
                     matching  template */
  #ifdef NETDEBUG
    pvm_perror("check_meets: nach plp_enable");
  #endif
    if(optype == PLI_MEETINTO)
    {
```

```
        /* the changed tuple must be checked from
               the beginning of the list */
      #ifdef NETDEBUG
        pvm_perror("check_meets: meetinto");
      #endif
        return(PLI_MEETINTO);
      }
    #ifdef NETDEBUG
      pvm_perror("check_meets: ende der Schleife");
    #endif
    }
    #ifdef NETDEBUG
      pvm_perror("check_meets: meet");
    #endif
    return(PLI_MEET);          /* the tuple is unchanged */
  }
```

**plq_check_fetch**

⟨*Interne Q-Operationen*⟩+≡

```
  extern Pli_Op_Type plq_check_fetch(Pbp_Object *tup,
                                Pli_Tuple_Space *ts);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist `pli_fetch`, falls ein matchendes Template gefunden wurde, sonst `PLI_NOOP`. |
| `tup` | ist das einzufügende Tupel. |
| `tsid` | ist der Tupelraum in den `tup` eingefügt werden soll. |

**Beschreibung:** `plq_check_fetch` testet, ob `tup` mit einem `fetch`-Template eines wartenden Prozesses matcht. Ist das der Fall wird der Prozeß aktiviert, der Eintrag in der Liste der wartenden Prozesse gelöscht und `PLI_FETCH` zurückgegeben.

**Ausnahmen:** Ausnahmebehandlung

**Probleme:** noch nicht

**Abhangigkeiten:** `plq_find_temp`, `plp_enable`, `plt_kill_temps`

⟨*Implementierung der Operationen*⟩+≡

```
Pli_Op_Type plq_check_fetch(Pbp_Object *tup, Pli_Tuple_Space *ts)
{
  Pli_Op_Type optype = PLI_FETCH;    /* plq_find_temp erwartet
                      eine Rueckgabeparameter */
  Pli_Pend_Proc *FoundTemp = NULL;   /* the entry of the
                       process of a matching template */

  if ((FoundTemp = plq_find_temp(&(ts->pendprocs), &tup,
             &optype, ts->atom)) == NULL)
    /* plq_find_temp erwartet einen Rueckgabeparameter
       (deswegen &tup, &optype)*/
  {
    /* no matching fetch template */
    return(PLI_NOOP);
  }
  else
  {
#ifdef Linda_Debug
    printf ("pending PLI_FETCH enabled for ");
    pbi_put(pbp_act_par, 1, tup);
#endif

    plp_enable(FoundTemp);  /*
```

```
        ** enabling the process of
        ** the matching template
        */
    return(PLI_FETCH);
  }
}
```

## C.4.4 Lokale Operationen

### plq_find_temp

⟨*Lokale Operationen*⟩≡

```
static Pli_Pend_Proc *plq_find_temp(Pli_Pend_Proc **tstlist,
                                    Pbp_Object **tup,
                                    Pli_Op_Type *optype,
                                    Pbp_Object *tsid)
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist der ausgewählte Prozeß oder `NULL`, falls kein matchender gefunden wurde. |
| `tstlist` | ist die Adresse der Liste der wartenden Prozesse des Tupelraums. |
| `tup` | ist die Adresse des einzufügenden Tupels. Im Fall von `PLI_MEETINTO` wird hier die Adresse des veränderten Tupels zurückgegeben. |
| `optype` | ist als Rückgabeparameter definiert und gibt an, ob es sich um den `PLI_MEET`- oder den `PLI_FETCH`-Fall handelt. Falls tup verändert wurde, wird `PLI_MEETINTO` zurückgegeben. |

**Beschreibung:** `plq_find_temp` sucht für `tup` ein passendes Template aus einem der wartenden Prozesse. Dabei wird unterschieden, ob es sich um `PLI_FETCH` oder `PLI_MEET` Templatelisten handelt. In `optype` ist angegeben, welcher Typ relevant ist. Im Fall von `PLI_MEET` wird durch `optype` `PLI_MEETINTO` zurückgegeben, falls `tup` durch einen `into`-Ausdruck verändert wurde. Zurückgegeben wird der ausgewählte Prozeß oder NULL.

**Ausnahmen:**

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:** `plo_match_list`

⟨*Lokale Operationen*⟩+≡

```
{
  Pli_Pend_Proc *RetPend = NULL,     /* the selected template */
                *prev;               /* and its predecessor */
  int tempNum = 0;          /* number of selected template */

  if(*tstlist == NULL)
  {
    /* no pending process */
    return(NULL);
  }

  /* checking the first entry */
  if(((*tstlist)->optype == *optype) &&
```

```
     ((tempNum = plo_match_list(tup, (*tstlist)->temps,
                optype, tsid, (*tstlist)->proc)) != 0))
  {
    /* the first entry matches */
    RetPend = *tstlist;
    *tstlist = (*tstlist)->next;      /* removing the entry */
    *(RetPend->ptempnum) = tempNum;   /* pass the number of the
                             selected */
                                      /* template to plp_wait */
    return(RetPend);     /* returning the entry of the process */
  }

  /* checking the other entries */
  prev = *tstlist;
  RetPend = prev->next;
  while(RetPend != NULL)
  {
    /* checking the actual entry */
    if((RetPend->optype ==  *optype) &&
       (tempNum = plo_match_list(tup, RetPend->temps,
                 optype, tsid, (*tstlist)->proc)) != 0)
    {
      prev->next = RetPend->next;      /* removing this entry */
      *(RetPend->ptempnum) = tempNum;  /* pass the number of the
                             selected */
                                       /* template to plp_wait */
      return(RetPend);   /* returning the entry of the process */
    }
    /* shifting the entry */
    prev = RetPend;
    RetPend = prev->next;
  }
  return(NULL);                        /* no template matched */
}
```

**plq_kill_task**

⟨*Interne O-Operationen*⟩≡

```
extern void plq_kill_task(Pli_Pend_Proc **pendList, int taskid);
```

**Profil:** `pendlist`

**Beschreibung:**

**Ausnahmen:**

**Probleme:** Ausnahmebehandlung

**Abhangigkeiten:**

⟨*Implementierung der Operationen*⟩+≡

```
void plq_kill_task(Pli_Pend_Proc **pendList, int taskid)
{
 Pli_Pend_Proc *killList = NULL;  /* das zu loeschende
                                    Listenelement */
 Pli_Pend_Proc *hList = NULL;   /* Hilfszeiger auf den Rest
                                  der Liste */
 Pli_Pend_Proc *prevList = NULL;  /* Zeiger auf den Vorgaenger
                                    von killList */

  if ((*pendList)->proc == taskid)
  {
     killList = *pendList;
     *pendList = (*pendList)->next;
     plt_kill_temps(killList->temps);   /* kill the templates */
     free(killList);                    /* kill this entry */
     return;
  }else{
     prevList = *pendList;
     hList = (*pendList)->next;
     while(hList != NULL)
     {
       if (hList->proc == taskid)
       {
          killList = hList;
          hList = killList->next;
          prevList->next = hList;
          plt_kill_temps(killList->temps); /* kill the templates*/
          free(killList);    /* kill this entry */
          return;
```

```
        }
        /* shifting hList and prevList*/
        prevList = hList;
        hList = hList->next;
      }
    }
  }
```

### C.4.5 C Dateien

$\langle$*linda.h*$\rangle\equiv$

$\langle$*intLinda.h*$\rangle\equiv$
$\langle$*Interne Q-Operationen*$\rangle$

$\langle$*queue.c*$\rangle\equiv$
$\langle$*Importe der Quelltextdatei*$\rangle$
$\langle$*Lokale Operationen*$\rangle$
$\langle$*Implementierung der Operationen*$\rangle$

# C.5 Die Linda–Templates

## C.5.1 Einleitung

Die nachfolgenden Funktionen generieren Templates.

## C.5.2 Abhängigkeiten

⟨*Importe der Quelltextdatei*⟩≡
```
#include "proset/proset.h"
#include "proset/unit.h"
#include "proset/overloaded.h"
#include "proset/function.h"
#include "proset/handler.h"
#include "proset/module.h"
#include "proset/loop.h"
#include "proset/atom.h"
#include "proset/string.h"
#include "proset/set.h"
#include "proset/tuple.h"
#include "proset/io.h"

#include "linda/linda.h"
#include "../intLinda.h"
```

## C.5.3 Exportierte Operationen

### plt_actual

⟨*Exportierte T-Operationen*⟩≡
```
extern Ple_Temp_Comp *plt_actual(Pbp_Object *rvalue);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist das neugenerierte Actual. |
| `rvalue` | enthält den Wert des `actual`. |

**Beschreibung:** `plt_actual` generiert ein Actual aus einem Rvalue.

**Ausnahmen:** keine

**Probleme:** noch nicht

**Abhängigkeiten:** `pbo_copy`

⟨*Implementierung der Operationen*⟩≡
```
Ple_Temp_Comp *plt_actual(Pbp_Object *rvalue)
{
  Ple_Temp_Comp *result;                /* the new actual */

  /* creating and initializing the new actual */
  if ((result = malloc(sizeof(Ple_Temp_Comp))) == NULL)
  {
    pbp_fehler("plt_actual", PBP_FATAL, "cannot allocate memory");
  }
  result->type = PLE_ACTUAL;

  /* copy the value */
  result->comp.actual = pbo_copy(rvalue);
  pbp_set_use(result->comp.actual);

  result->next = NULL;
  return (result);           /* returning the new component */
}
```

## plt_formal

⟨*Exportierte T-Operationen*⟩+≡
```
extern Ple_Temp_Comp *plt_formal(Pbp_Object *into, Pbp_Env dL,
         Pbp_Exit *RetStat, Pbp_Env *RetAd,
         Pbp_Object **lvalue);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist das neugenerierte Formal. |
| `into` | ist die Funktion, die den Wert des Feldes des passenden Tupels verändert, falls es sich um den `PLI_MEETINTO`-Fall handelt. Im anderen Fall ist der Wert von `into` `NULL`. |
| `lvalue` | nimmt den Wert des passenden Tupels auf. |

**Beschreibung:** `plt_formal` generiert ein Formal aus einer `into`-Funktion un einem Lvalue.

**Ausnahmen:** keine

**Probleme:** noch nicht

**Abhängigkeiten:** keine

⟨*Implementierung der Operationen*⟩+≡

```
Ple_Temp_Comp *plt_formal(Pbp_Object *into, Pbp_Env dL,
         Pbp_Exit *RetStat, Pbp_Env *RetAd,
         Pbp_Object **lvalue)
{
  Ple_Temp_Comp *result;                /* the new formal */

  /* creating and initializing the new formal */
  if ((result = malloc(sizeof(Ple_Temp_Comp))) == NULL)
  {
    pbp_fehler("plt_formal", PBP_FATAL, "cannot allocate memory");
  }
  result->type = PLE_FORMAL;
  result->comp.formal.lvalue = lvalue;
  result->comp.formal.into = into;
  result->comp.formal.dL = dL;
  result->comp.formal.RetStat = RetStat;
  result->comp.formal.RetAd = RetAd;
  result->next = NULL;
  return(result);          /* returning the new component */
}
```

### plt_cre_temp

⟨*Exportierte T-Operationen*⟩+≡

```
extern Ple_Template *plt_cre_temp(Pbp_Object *condition,
               Pbp_Env dL,
                Pbp_Exit *RetStat, Pbp_Env *RetAd,
                  const unsigned ParamNo, ...);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist das neugenerierte Template. |
| `condition` | ist die Funktion, die die Bedingung zum Template prüft. |
| `dL` | ist der dynamische Vorgänger von `condition`. |
| `RetStat, RetAd` | sind Ausnahmebehandlungsparameter. |
| `ParamNo` | ist die Anzahl der Komponenten des Template. |
| `...` | sind die Komponenten des Template. |

**Beschreibung:** `plt_cre_temp` generiert ein Template aus einer Bedingungsfunktion und einer Anzahl von Komponenten.

**Ausnahmen:** keine

**Probleme:** noch nicht

**Abhängigkeiten:** keine

⟨*Implementierung der Operationen*⟩+≡

```
Ple_Template *plt_cre_temp(Pbp_Object *condition, Pbp_Env dL,
                           Pbp_Exit *RetStat, Pbp_Env *RetAd,
                           const unsigned ParamNo, ...)
{
  register unsigned compNum;/*the number of the actual component*/
  Ple_Template *result;             /* the new template */
  Ple_Temp_Comp *lastNOTom = NULL;  /* the last component which
                                 is not om */
  Ple_Temp_Comp *tokill;            /* first of the last om */
  Ple_Temp_Comp *next;
  Ple_Temp_Comp **last;             /* the last component */
  va_list ap;                       /* the parameter list */

  va_start(ap, ParamNo);            /* initializing ap */

  /* creating and initializing the new template */
  if ((result = malloc(sizeof(Ple_Template))) == NULL)
  {
    pbp_fehler("plt_cre_temp", PBP_FATAL,
                "cannot allocate memory");
  }
  result->next = NULL;
```

```
result->arity = ParamNo;
result->comps = NULL;
result->condition = condition;
result->dL = dL;
result->RetStat = RetStat;
result->RetAd = RetAd;

/* adding the components */
last = &(result->comps);
for (compNum = 1; compNum <= ParamNo; compNum++)
{  /* adding the next component */
  *last = va_arg(ap, Ple_Temp_Comp *);
  if(((*last)->type == PLE_FORMAL) ||
     (pbp_type((*last)->comp.actual) != PBP_OM))
  {
    lastNOTom = *last;
  }
  last = &((*last)->next);              /* shifting last */
}
va_end (ap);

if (lastNOTom == NULL)
{
  if (ParamNo == 0)
  {
    return(result);
  }
  else
  {
    tokill = result->comps;
    result->comps = NULL;
  }
}
else
{
  tokill = lastNOTom->next;
}
```

```
  while(tokill != NULL)
  {
    if(tokill->type == PLE_FORMAL)  /* nur zu Testzwecken */
    {
      pbp_fehler("plt_cre_temp", PBP_COMPILER,
                          "om wird nicht erkannt");
    }
    pbo_kill(tokill->comp.actual);   /* killing the object */
    result->arity--;
    next = tokill->next;           /* shifting next */
    free(tokill);                    /* killing tokill */
    tokill = next;                   /* shifting tokill */
  }

  return(result);             /* returning the new template */
}
```

**plt_cre_tl**

⟨*Exportierte T-Operationen*⟩+≡

```
extern Ple_Template *plt_cre_tl(const unsigned ParamNo, ...);
```

**Profil:**

| | |
|---|---|
| `Rückgabewert` | ist die komplette Liste der Templates. |
| `ParamNo` | ist die Anzahl der Templates. |
| `...` | sind die zu verkettenden Templates. |

**Beschreibung:** `plt_cre_tl` generiert eine Liste von Templates.

**Ausnahmen:** vielleicht: `no_templates`

**Probleme:** Ausnahmebehandlung

**Abhängigkeiten:** keine

⟨*Implementierung der Operationen*⟩+≡

```
Ple_Template *plt_cre_tl(const unsigned ParamNo, ...)
{
  register unsigned tempNum;  /* the number of the actual
                                    template */
  Ple_Template *result,                /* the new template list */
               *last;         /* the last template of the list */
  va_list ap;                          /* the parameter list */

  va_start(ap, ParamNo);               /* initializing ap */

  if (ParamNo == 0)                    /* compiler error? */
  {
    pbp_fehler("plt_cre_tl", PBP_COMPILER, "no templates?");
  }

  result = va_arg(ap, Ple_Template *);  /* the first template */
  last = result;

  /* adding the others */
  for (tempNum = 2; tempNum <= ParamNo; tempNum++)
  {   /* adding the next component */
    last->next = va_arg(ap, Ple_Template *);
    last = last->next;                       /* shiftin last */
  }
  va_end (ap);
```

```
    return(result);        /* returning the new template list */
  }
```

### plt_kill_temps

⟨*Interne T-Operationen*⟩≡

```
  extern void plt_kill_temps(Ple_Template *tokill);
```

**Profil:** `tokill` ist die zu löschende Liste von Templates.

**Beschreibung:** löscht die Templateliste und gibt den Speicher frei.

**Ausnahmen:** keine

**Probleme:**

**Abhängigkeiten:** `plt_kill_temp`

⟨*Implementierung der Operationen*⟩+≡

```
  void plt_kill_temps(Ple_Template *tokill)
  {
    Ple_Template *next;         /* the next member of the list */

    /* killing the whole list */
    while (tokill != NULL)
    {
      plt_kill_temp(tokill->comps);   /* killing the components */
      next = tokill->next;            /* shifting next */
      free (tokill);                  /* killing tokill */
      tokill = next;                  /* shifting tokill */
    }
  }
```

### plt_kill_temp

⟨*Lokale Operationen*⟩≡

```
  static void plt_kill_temp(Ple_Temp_Comp *tokill)
```

**Profil:** `tokill` ist die zu löschende Liste von Templatekomponenten.

**Beschreibung:** `plt_kill_temp` löscht alle Komponenten eines Templates und gibt den Speicher frei.

**Ausnahmen:** keine

**Probleme:**

**Abhängigkeiten:** `pbo_kill`

⟨*Lokale Operationen*⟩+≡
```
{
  Ple_Temp_Comp *next;       /* the next member of the list */

  /* killing the whole list */
  while(tokill != NULL)
  {
    if (tokill->type == PLE_ACTUAL)
    {
    #ifdef NETDEBUG
      pvm_perror("kill_temp: vor pbo_kill");
    #endif
      pbo_kill(tokill->comp.actual); /* killing the object */
    }

    next = tokill->next;            /* shifting next */
    free(tokill);                   /* killing tokill */
    tokill = next;                  /* shifting tokill */
  }
}
```

### C.5.4 C Dateien

⟨*linda.h*⟩≡
  ⟨*Exportierte T-Operationen*⟩

⟨*intLinda.h*⟩≡
  ⟨*Interne T-Operationen*⟩

$\langle$*template.c*$\rangle\equiv$
  $\langle$*Importe der Quelltextdatei*$\rangle$
  $\langle$*Lokale Operationen*$\rangle$
  $\langle$*Implementierung der Operationen*$\rangle$