Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Mit der zunehmenden Vernetzung von Computern in den letzten Jahren haben verteilte Anwendungen einen immer gröSSeren Stellenwert in der Informatik gewonnen. Die Einsatzgebiete sind vielfältig: von der Systemadministration über Lastverteilung in Datenbanken oder Internetanwendungen bis hin zum klassischen Numbercrunching rechenintensiver Prozesse reichen die Anwendungsgebiete.

Durch unterschiedliche Systemarchitekturen und Aufgabenstellungen sind allerdings die am Netzwerk angeschlossenen Rechner oft nur mit groSSem Zusatzaufwand zur Zusammenarbeit zu bewegen.

1.2 Zielsetzung

Im Rahmen meiner Tätigkeit in der Systemadministration in der Informatik der Universität Osnabrück hatte ich Kontakt mit sehr unterschiedlichen Betriebssystemen wie WindowsXX, Linux, Solaris, OpenStep und HP-UX. Nachdem es immer wieder eine spannende Aufgabe war, auch nur einen Netzwerkdrucker im gesamten Netz verfügbar zu machen, wuchs der Wunsch nach einem Werkzeug, daSS eine uniforme Administration möglich macht. Lösungsansätze fanden sich auf der einen Seite im Bereich der verteilten Programmierung mit RMI oder CORBA, auf der anderen Seite bei Programmiersprachen, die für alle Plattformen verfügbar sind.

Dazu zählt auch die Skriptsprache Perl, die im Bereich der Administration weit verbreitet ist. Nach ersten Gehversuchen mit kleineren Programmen lernte ich die schnelle Lösbarkeit komplexer Aufgabenstellungen mit Perl schätzen. Die Vielzahl schon implementierter Module von Datenbankanbindungen und Win32-Bibliotheken bis hin zu grafischen Benutzerinterfaces mit Perl/Tk ermöglicht zum einen die Kommunikation mit systeminternen Mechanismen auf spezifischen Plattformen zum Anderen die einfache Ausbaufähigkeit (Html,Mail).

Daraus entstand die Idee, mit Perl verteilt zu programmieren. Da mit der seit Version 5 die Objektorientierung auch bei Perl Einzug gehalten hatte, lag es nahe, diesen Ansatz zu wählen. Erleichternd kam hinzu, daSS die Grundlagen der Netzwerkprogrammierung (IO::Socket) ebenfalls verfügbar waren.

Ziel der Entwicklung war es, ein kompaktes, erweiterbares System zu schaffen, daSS ohne groSSen Installations- und Konfigurationsaufwand auskommt, und einen guten KompromiSS zwischen Robustheit, Performance und Verwendbarkeit findet. Dieses System wird im folgenden Perl Object Server (POS) genannt.

1.3 Überblick

To be updated

Im Kapitel 2 wird zunächst ein kurzer AbriSS der Geschichte von Perl gegeben, sowie die Systemvoraussetzungen für den Perl Object Server geschildert.

Kapitel 3 enthält einen Überblick über Grundlagen der verteilten Programmierung.

Kapitel 4 beschreibt das Design des Perl Object Servers und die Einordnung dieses Systems

In Kapitel 5 wird die Konfiguration eines POS Systems erläutert.

Kapitel 6 fasst den Stand der Entwicklung zusammen und nennt Möglichkeiten der Anwendung.

2 Voraussetzungen

2.1 Die Programmiersprache: Perl

Am 18. Oktober 1987 gab Larry Wall die Version 1.0 von Perl (Practical Extraction and Report Language), gedacht als Synthese von C, sh, awk und sed frei. Es fand schnell weite Verbreitung. Es folgten zahlenmäSSig groSSe Schritte: 2.0 (1988), 3.0(1989) und 4.0 (1991). Bis 1993 war die Versionsnummer 4.036 erreicht. 1994 erschien mit 5.000 ein vollständig neu implementierter Interpreter mit vielen Erweiterungen, insbesondere Referenzen, mehrdimensionalen Arrays und der Möglichkeit objektorientierter Programmierung. Der aktuelle Stand (Ende März 2000) ist 5.6.0. Die Liste der Portierungen von Perl hat eine beachtliche Länge erreicht [**?**].

Wichtige Features: OO&Sysprog

3 Verteilte Systeme

Was ist ein verteiltes System?

Zur Einschätzung der Eigenschaften eines verteilten Systems gehört zunächst eine Antwort auf die Frage: Was ist ein verteiltes System überhaupt? Die Definition von Leslie Lamport (sinngemäSS) "Verteilte Systeme sind Systeme, in denen ich durch den Ausfall einer Komponente, von deren Existenz ich bisher nicht wuSSte, in meiner Arbeit beeinträchtigt werde." ist immerhin sehr lebensnah und leider nicht strukturell. Weiter hilft da schon folgende Aussage:

Ein verteiltes System besteht aus einer Anzahl autonomer vernetzter Rechner mit verteilter Systemsoftware.

Verteilte Systemsoftware versetzt die Rechner eines verteilten Systems in die Lage, ihre Aktivitäten zu koordinieren.

Wozu verteilte Systeme?

Die Zielsetzungen verteilter Systeme sind vielfältig;

- Verteilung von Daten zur Bearbeitung
- Erhöhung der Ausfallsicherheit
- Bessere Ressourcenausnutzung
- Schnellere Verarbeitung
- Effizientere Administration

Dementsprechend gibt es verschiedene Arten der Verteilung, die für diese Aufgaben eingesetzt werden können[3]:

- Hardwarekomponenten(RAID,High-Availability-Systeme)
- Last (Loadbalancing bei Webservern)
- Kontrolle (Betriebssysteme)
- Daten(verteilte Datenbanken)
- Verarbeitung(RPC)

Da im Folgenden verteilte Objekte betrachtet werden ...

Im Folgenden werden die beiden letzten Punkte interessant sein, weil bei objektorientierter Programmierung Daten und Methoden auf diesen Daten als Einheit betrachtet werden.

3.0.1 Definition

Eine verteilte Anwendung wird definiert durch[?]:

- Eine Anwendung A, deren Funktionalität in eine Menge von kooperierenden Teilkomponenten $A_1 \dots A_n (n \in N, n > 1)$ zerlegt ist.

- Die Teilkomponenten A_i sind autonome Verarbeitungseinheiten, die auf verschiedene Verarbeitungseinheiten F_i abgebildet werden.
- Teilkomponenten A_i tauschen untereinander Informationen mittels eines Netzes aus.

Eine wichtige Eigenschaft verteilter Anwendungen ist - im Gegensatz zu Einzelplatzprogrammen wie zu zentralisierten Client-Server Anwendungen - ihr hohes MaSS an Komplexität. Deutlich wird dies durch die verschiedenen Arten von Transparenz, die eine verteilte Anwendung auszeichnen kann.

3.1 Transparenz

Mit den Transparenzaspekten werden Eigenschaften verteilter Systeme sinnvoll charakterisiert. Sie beschreiben ebenso die grundlegenden Vorteile verteilter Anwendungen.

- Von Ortstransparenz(location transparency) wird gesprochen, wenn zwei Teilkomponenten A_i miteinander kommunizieren können, ohne Kenntnis davon haben zu müssen, wo die jeweils andere Komponente lokalisiert ist.
- Unter Zugriffstransparenz(access transparency) wird verstanden, wenn sich die Art des Zugriffes auf Teilkomponenten nicht unterscheidet, egal ob diese lokal oder entfernt lokalisiert sind.
- Werden - aus Gründen der Verfügbarkeit - Objekte im System repliziert und synchronisiert, ohne diese Replikation für den Anwender sichtbar werden zu lassen, wird von Replikationstransparenz(replication transparency) gesprochen.
- Unter Fehlertransparenz wird verstanden, wenn in einem verteilten System Teilausfälle im Netzwerk oder Rechnerabstürze abgefangen werden, ohne die Funktion des Gesamtsystems zu beeinträchtigen.
- Kann nebenläufig oder konkurrierend auf gemeinsame Ressourcen im System zugegriffen werden, sind also entsprechende Synchronisationsmechanismen eingebaut, so wird dies mit Nebenläufigkeitstransparenz (concurrency transparency) bezeichnet.
- Besteht die Möglichkeit, Ressourcen in verteilten Systemen zu migrieren, also je nach Netzlast, Rechnerlast oder Performance auf andere Rechner zu lokalisieren, wird von Migrationstransparenz gesprochen.
- ProzeSStransparenz(execution transparency) meint die Unabhängigkeit eines Prozessen vom Ausführungsort.
- Leistungstransparenz (performance transparency) erlaubt eine dynamische Rekonfiguration des Systems zur Verbesserung der Leistung bei Lastveränderung.
- Kann ein System ohne Modifikation der Struktur und der Anwendung erweitert werden, sprechen wir von Skalierungstransparenz.

- Ist die Funktion und Interaktion von Teilkomponenten unabhängig von der verwendeten Programmiersprache , so wird dies als Sprachtransparenz (language transparency) bezeichnet.

Es existiert kein verteiltes System , daSS alle Aspekte der Transparenz implementiert. Insbesondere die - implizit immer vorhandene - Forderung nach hinreichenden Antwortzeiten kann mit der Fehler- bzw. Replikationstransparenz nur schwer vereinbar sein.

3.2 Kommunikation

Zentrale Grundlage verteilter Systeme ist die Kommunikation der Teilkomponenten A_i.

die erste Idee könnte es sein, die Information über eine zentrale Informationsverwaltung abzuwickeln.

3.2.1 Information Sharing

<Bild >

Die Teilkomponenten A_i kommunizieren ausschlieSSlich über die Informationsverwaltung miteinander. Beispiele hierfür wären ein gemeinsames Dateisystem, in dem Daten nach einem gemeinsamen Datenmodell abgelegt werden.

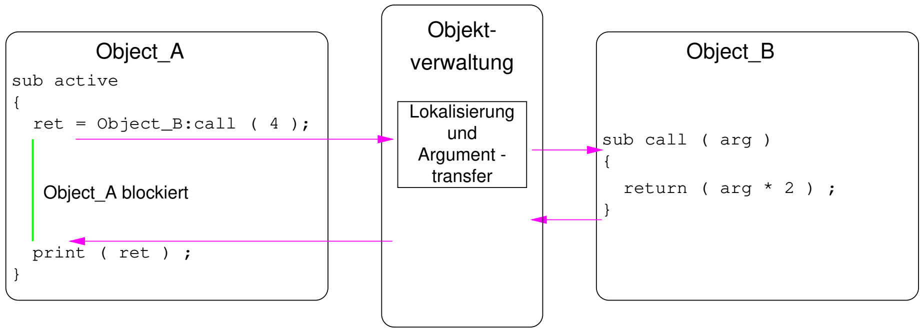
Nach der sehr allgemeinen Charakterisierung von verteilten Systemen wird in diesem Abschnitt ein kurzer Überblick über die Kommunikation von Objekten gegeben.

Zentrale Grundlage verteilter Systeme ist der Nachrichtenaustausch. Im Folgenden werden wichtige Kommunikationsmechanismen aus der Sicht verteilter Objekte erläutert.

Als Beispiel wird folgende einfache Kommunikation zwischen Objekt A und Object B verwendet: A ruft bei B die Methode call mit einer Zahl als Argument auf, die als Ergebnis das Doppelte dieser Zahl an A zurückliefert.

3.2.2 Synchrone Kommunikation

Bei Aufruf einer Methode wird der Aufrufer solange blockiert, bis das aufgerufene Objekt lokalisiert und die Methode evaluiert worden ist. Der Aufrufer wird dann mit dem Ergebnis wieder reaktiviert.

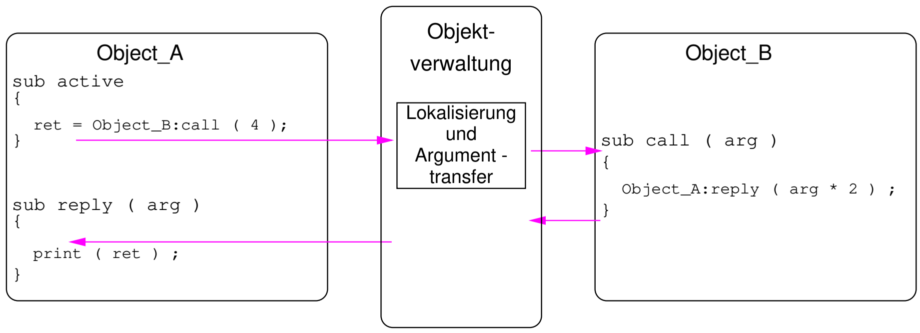


Dieser Ansatz hat den Vorteil, dass Klassen, die ursprünglich für lokale Anwendungen implementiert wurden, ohne groSSen Aufwand portiert werden können.

Nachteilig ist der erhöhte Aufwand in der Objektverwaltung und die Synchronisierug im Netzwerk.

3.2.3 Asynchrone Kommunikation

Im Gegensatz zur synchronen Übertragung wird beim asynchronen Ansatz der Sender nicht blockiert. Damit kann der Sender sofort weiterlaufen, bekommt allerdings auch keinen unmittelbaren Rückgabewert des Empfängers, sondern nur die Quittung der Objektverwaltung über das versenden einer Nachricht - im Folgenden Event genannt.



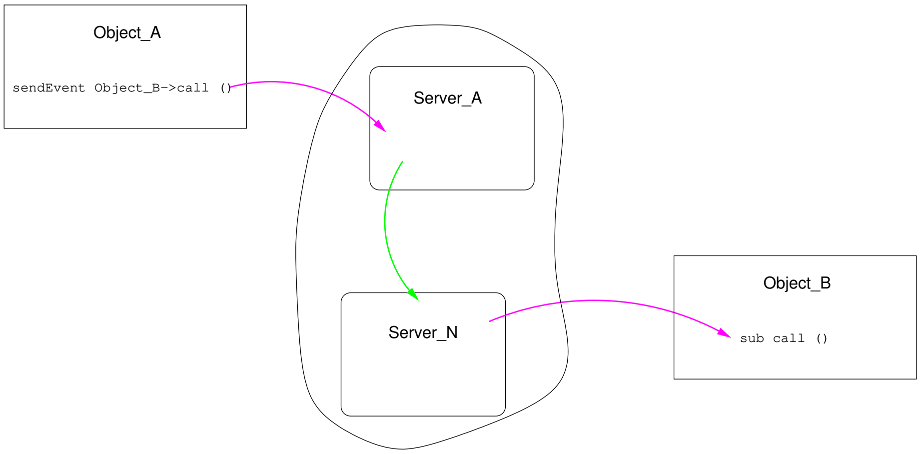
Benötigt ein Objekt einen Rückgabewert, muSS dieser explizit angefordert werden. Im einfachsten Falle muSS das Empfängerobjekt in Eigenregie eine Nachricht zurückschicken.

Vorteil dieses Konzeptes ist die Einfachheit: Die Objektverwaltung versendet Events - ohne Stacks oder Rückgabewerte berücksichtigen zu müssen.

Nachteilig macht sich bemerkbar, daSS für lokale Anwendungen implementierte Klassen nicht ohne gröSSeren Aufwand portiert werden können, da sie speziell auf den Eventmechanismus der Objektverwaltung zugeschnitten werden müssen.

3.2.4 Kommunikation im POS

Beim POS wurde aufgrund der einfacher zu implementierenden Objektverwaltung als Grundlage die aynchrone Kommuikation gewählt. Ziel ist des damit, eine plattformübergreifende einheitliche Eventfläche zu schaffen: Objekt A schickt einen Event mit Ziel Objekt B in das System hinein. Der POS hat dann die Aufgabe, Objekt B zu lokalisieren, und den Event - gegebenenfalls über eine Netzwerkverbindung - an die POS- Instanz, die Objekt B verwaltet, zu versenden. Diese stellt den Event dann zu.

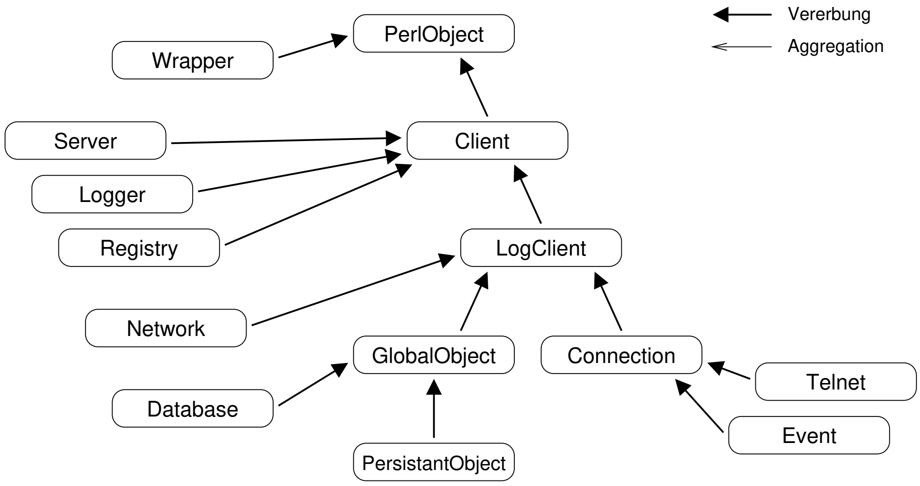


4 Design und Klassenbibliothek

Da der POS einfach aufgebaut sein sollte ist das Design sehr simpel gehalten. Alle Objekte inklusive des Servers teilen sich einen ProzeSS, eine lokale Instanz des POS ist also einfach ein laufendes Perl- Programm. Das POS- System wiederum besteht aus zwei Hauptkomponenten: Die lokale Objektverwaltung (Server.pm) registriert Objekte behandelt lokale Ereignisse. Die Netzwerkkomponente (Network.pm) lokalisiert Objekte und stellt die Kommunikation mit anderen Servern her.

Der ursprüngliche Ansatz zum Entwurf von POS war durchgehend objektorientiert. Es zeigte sich allerdings schnell, daSS Perl als interpretierte Sprache schnell an Performance verlor. Das Ergebnis ist ein hybrides System.

Als Beispiel mag die Repräsentation einer Nachricht dienen:



4.1 Events

Ein Event ist ein Informationspaket mit minimal der Empfängeradresse und dem eigentlichen Gehalt, den Argumenten. Damit ist eine Zustellung eines Events schon möglich. Im POS sind Events als Array implementiert, in dem weitere Informationen übermittelt werden:

Index	Typ	Format	Beschreibung
0	scalar	integer	Laufende Nummer
1	ARRAY	S,O,M	Absender
2	ARRAY	[S,]O,M	Empfänger
3	ARRAY	[argA, argB,...]	Argumentliste
[4]	ARRAY	[S,]O,M	Reply To

Beim Format der Indices 1,2 und 4 sind die Bezeichnungen abgekürzt: S bezeichnet den Namen eines Servers, O den Namen eines Objektes und M den

Namen einer Methode, jeweils als String. Damit kann innerhalb der Strukur von POS insbesondere ein Objekt eindeutig lokalisiert werden.

Bei Index 0 steht eine vom lokalen Server vergebene eindeutige laufende Nummer.

Index 1 beinhaltet ein weiteres Array, dessen Inhalt den Absender identifiziert. Die eingetragene Methode ist die, deren Aufruf den Event getriggert hat, also i.A. verschieden von derjenigen, die den Event versendet.

Index 2 gibt den Empfänger des Events an. Dabei ist die Angabe des Servernamens zunächst optional (siehe **??**).

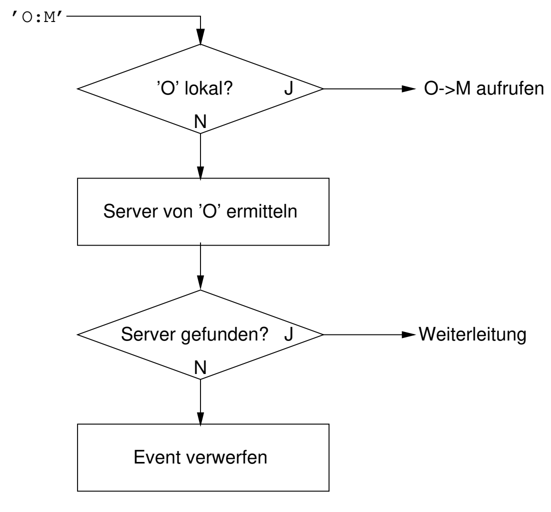
Index 3 enthält die Argumentliste für den Methodenaufruf im Empfänger.

Index 4 ist optional und belegt mit einer Reply-To Adresse (siehe **??**).

4.1.1 Die Versendung

Ein Event wird versendet, indem im Server die Methode 'S:O:M' aufgerufen wird, das heiSSt Servername, Objektname und Methodenname werden mit Doppelpunkten als Trenner in einem String geschrieben. Der entstehende String ist sicher keine Methode, die im Server tatsächlich vorhanden ist. Der Server implementiert die Methode AUTOLOAD, die bei einem unbekannten Methodenaufruf getriggert wird. Dort wird der Aufruf analysiert, in einem Event gespeichert, in die interne Queue übernommen und die Kontrolle wieder dem Aufrufer gegeben.

Bei der Suche nach dem Empfängerobjekt wird unterschieden, ob der Servername angegeben war oder nicht. Im ersten Falle wird der Event direkt an den entsprechenden Server weitergeleitet. Im zweiten Fall wird zunächst lokal nach einem Objekt des angegebenen Namens gesucht.

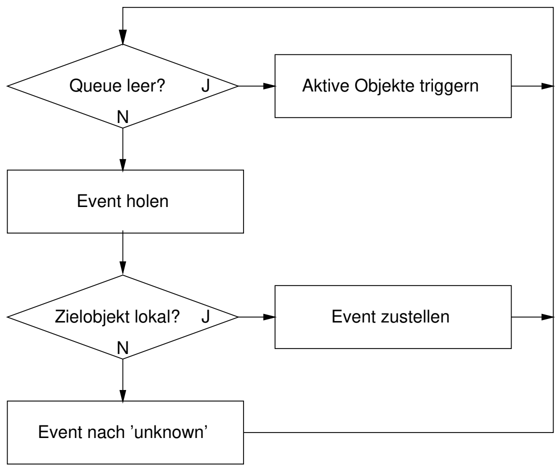


4.1.2 Der Empfang

Nachdem der Event auf richtigen Server angekommen ist, wird die angegebene Methode aufgerufen. Diese kann ihrerseits Events versenden, die zunächst gequeuet werden. Nach Beendigung der Methode wird der Rückgabewert zunächst gespeichert. Ist beim Event Index 4 belegt, wird dieser Rückgabewert an die dort angegebene Adresse gesendet.

4.2 Lokales Eventhandling: Server.pm

Server.pm hat zunächst die Aufgabe, Clients lokal zu verwalten, eingehende Events entgegenzunehmen und weiterzuleiten. Hier ist die Hauptschleife von POS implementiert. Dieses Modul kennt noch kein Netzwerk und ist standalone funktionsfähig. Die prinzipiellen Funktionen der Hauptschleife zeigt **??**.



Zur Implementierung des Servers gehören einige Listen, die verschiedene Aufgaben haben. Allen gemeinsam ist ihr Inhalt: es sind Methoden von registrierten Clienten, denen spezielle Aufgaben zukommen. Im Folgenden werden diese Listen genauer beschrieben

4.2.1 log [Stack]

Die Methode, die oben auf dem Stack liegt, erhält alle Meldungen, die der Server entgegennimmt. Jede Meldung besteht aus zwei Argumenten: level und message, wobei level die Priorität der Meldung beinhaltet. Der log-Stack wird bei der Initialisierung des Servers mit der internen Methode poslogfunc vorbelegt, die alle Meldungen nach STDOUT schreibt.

4.2.2 unknown [Stack]

Die Methode die auf unknown liegt, erhält alle Events, die nicht lokal verarbeitet werden können. Initialisiert wird unknown mit der Methode Server->error() die einen Dump des Events als Fehler an log versendet.

4.2.3 timed [Priority-Queue]

Die sortierte Liste aller Methoden, die periodisch aufgerufen werden (Auflösung 1 sec).

4.2.4 active [List]

Die Liste aller Methoden, die aufgerufen werden, wenn die Event-Queue leer ist. Initial leer.

4.3 Globales Eventhandling: Network.pm

Neben Server.pm ist Network.pm das zweite Hauptmodul des POS- Systems. Es lokalisiert Objekte im Netzwerk und versendet Events an andere Server.

4.4 Verbindungsaufnahme: Connection.pm

Die Connection- Klasse bildet die Basisklasse für die Kommunikation des POS-Systems - derzeit ausschlieSSlich via TCP - mit anderen Servern, telnet- Sessions oder Html- Browsern.

Die eine Besonderheit dieser Klasse besteht darin, daSS Connection zwei Basisklassen hat. Um die Socket- Funktionalität zu erhalten, wird IO::Socket::Inet eingebunden. Die zweite Quelle ist die Eigenimplementierung LogClient, um die Methoden zur Serialisierung und des Logging verwenden zu können.

Die Zweite Besonderheit ergibt die sich aus der Kombination des zeilenorientierten Lesens von einem Socket in Kombination mit der Verwendung des select() - Systemaufrufes. Der Rückgabewert eines can_read() sind alle Sockets, von denen gelesen werden kann. Werden die ankommenden Zeilen mithilfe der stdio-Bibliothek gelesen, verschwinden alle Zeilen im Filepuffer, und das can_read blockiert - obwohl unter Umständen noch Zeilen im Puffer zwischengelagert sind. Als Lösung hat jedes Connection- Objekt einen eigenen Puffer, in dem ankommende Zeilen wieder zusammengesetzt und gespeichert werden.

4.5 Einstellungssache: Configuration.pm

4.6 Geordneter Start: Wrapper.pm

Das Wrapper- Objekt implementiert den Startup des POS

5 Konfiguration

Alle Einstellungen des POS werden in einer zentralen Datei zusammengefaSSt.

6 Zusammenfassung und Ausblick

Dies ist die Zusammenfassung.

A Betriebssysteme

Die Funktion des Perl Object Servers wurde auf folgenden Plattformen getestet:

Plattform	Version	Single	Netz	Master
Linux Suse 6.x	5.005	+	+	+
WinNT 4.0	5.005	+	+	
Win95b	5.005	+	+	
Solaris 7	5.005	+	+	+
OpenStep 4.2	5.004	+	+	*
FreeBSD	5.005	+	+	+

Auf den Win*- Systemen ist die Installation des TCP/IP-Netzwerks optional: Für die Vernetzung von POS ist diese notwenig.

* Unter OpenStep fehlt in Version 4.2 die POSIX- Bibliothek, die `setsid` implementiert.

B Parameter

Parameter

Literatur

[1] http://www.perl.com/pub/doc/manual/html/pod/perlport.html

[2] Uwe M. Berghoff, Johann Schlichter: "Rechnergestützte Gruppenarbeit - Eine Einführung in verteilte Anwendungen", Springer Verlag (1995)