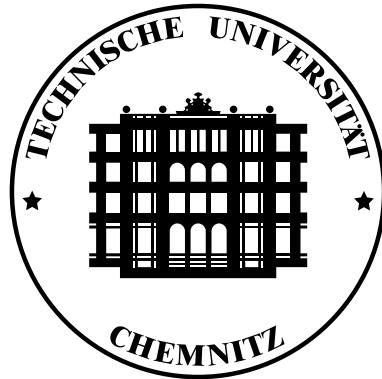


Technische Universität Chemnitz



Fakultät für Informatik

Professur für Rechnernetze und verteilte Systeme

Diplomarbeit

Nutzerorientiertes Management von materiellen und immateriellen Informationsobjekten

Verfasser:

Chris Hübsch

16. Oktober 2001

Betreuer:

Prof. Dr. U. Hübner

Hübsch, Chris:

*Nutzerorientiertes Management von materiellen
und immateriellen Informationsobjekten*

Diplomarbeit, Technische Universität Chemnitz,
2001.

Danksagung

Mein Dank gilt allen, die mich bei dieser Arbeit unterstützt haben. Insbesondere:

- Prof. Hübner für die überaus herausfordernde Aufgabenstellung und die horizonterweiternden Diskussionen,
- Ralph Sontag für seine konstruktiven Kritiken nicht nur an dieser Arbeit,
- Jan und Karsten für ihre Anregungen, wenn die Gefahr bestand, den Horizont aus dem Blick zu verlieren,
- Den Korrekturlesern insbesondere Tino und meiner Mutter,
- Den Angestellten der Bibliothek, die mir trotz Renovierungschaos die nötigen Bücher beschaffen konnten,
- und allen, die spätestens alle 24 Stunden gefragt haben, ob die Arbeit noch immer nicht fertig sei — nun ist sie es!

Inhaltsverzeichnis

Abbildungsverzeichnis	viii
Tabellenverzeichnis	ix
Verzeichnis der Abkürzungen	xi
1. Problemstellung	1
1.1. Aufgabenstellung	1
1.2. Hintergrund	1
1.3. Anforderungen	2
2. Lösungsansätze	3
2.1. Architekturalternativen	3
2.1.1. Terminalsystem mit Fernzugriff	4
2.1.2. Hostsystem mit lokaler Ausgabe	4
2.1.3. Kooperative Verarbeitung	5
2.1.4. Client/(Datenbank)Server	6
2.1.5. Verteilte Daten	6
2.1.6. Voll verteiltes System	7
2.1.7. Realisierungsvorschlag	7
2.2. Kommunikation in verteilten Systemen	8
2.2.1. Socket-Programmierung	10
2.2.2. Klassischer RPC	11
2.2.3. CORBA	13
2.2.4. HTTP	15
2.2.5. XML-RPC	16
2.3. Implementierungsumgebung	17
2.3.1. Programmierparadigmen	17
2.3.2. Programmiersprachen	20
2.3.3. Datenformat	23
2.4. Benchmarking	25
2.4.1. BSD-Sockets	27
2.4.2. Sun-RPC	29
2.4.3. CORBA	29
2.4.4. XML-RPC	32
2.4.5. Vergleich zwischen RPC-Mechanismen	34

3. Realisierung	39
3.1. Basisdienste	40
3.1.1. Dienstkontrolle (BOSSERVER)	41
3.1.2. Dienstauffindung (LOCATIONSERVER)	44
3.1.3. Knotenverwaltung (NODESERVER)	49
3.1.4. Logging (LOGGINGSERVER)	51
3.1.5. Systemmonitoring (LOADSERVER)	51
3.2. Datenbankdienste	52
3.2.1. Anfragesystem (QUERYSERVER)	53
3.2.2. Performanz und Zuverlässigkeit (SLIDESERVER)	54
3.2.3. Persistenz (PERSISTENCYSERVER)	56
3.2.4. Synchronisation (CONCURRENCYSERVER)	60
3.2.5. Zusammenwirken der Datenbankdienste	61
3.3. Anwendungsdienste	63
3.3.1. Objektsuche (SEARCHSERVER)	63
3.3.2. Exemplarinformation	64
3.3.3. Empfehlungen (RECOMMENDATIONSERVER)	64
3.3.4. Rezension	65
3.3.5. Ausleihe	66
3.3.6. E-Mail an Ausleiher (MAILSERVER)	66
4. Problemfelder	69
4.1. Parallele asynchrone Aufrufe	69
4.2. Defizite von XML-RPC	69
4.2.1. Dictionaries in XML-RPC	70
4.2.2. Objekte in XML-RPC	70
4.3. Multithreading, Konsistenz, Deadlocks	72
4.4. SSL	72
4.5. Python im Webserver	73
5. Zusammenfassung und Ausblick	75
5.1. Entwicklungsstand	75
5.2. Weiterführende Aufgaben	75
5.2.1. Mehrdimensionale Suche	75
5.2.2. Unscharfes Stringmatching	76
5.2.3. Synonymsuche	76
5.2.4. Fremdsprachige Suche	76
5.2.5. Umstellung auf RDF im Speicher und für Fremdzugriffe	76
5.2.6. Transaktionsfähigkeit der Datenbank	77
5.2.7. GUI	77

5.2.8. Effizientere Kommunikation	78
5.2.9. Sicherheit	78
5.2.10. Dynamische Knotenkontrolle	78
A. Programmbeispiele	81
A.1. Sun-RPC Benchmark	81
A.2. CORBA Benchmark	83
A.3. XML-RPC Benchmark	85
A.4. Modul defcall	86
B. Messwerte	87
Literaturverzeichnis	98

Inhaltsverzeichnis

Abbildungsverzeichnis

1.	Funktionsgruppen	3
2.	Verteilung: Terminalsystem	4
3.	Verteilung: Hostsystem	5
4.	Verteilung: Kooperative Verarbeitung	6
5.	Verteilung: Client/(Datenbank)Server	6
6.	Verteilung: Verteilte Daten	7
7.	Verteilung: Voll verteiltes System	7
8.	Verteilung: Realisierungsvorschlag	8
9.	XML-RPC-Codierung	18
10.	SOAP-Codierung	19
11.	Codierung mit RDF	25
12.	Codierung mit XML-RPC	26
13.	RPC-Benchmark: Sockets	28
14.	RPC-Benchmark: Sun-RPC	30
15.	RPC-Benchmark: CORBA	31
16.	RPC-Benchmark: XML-RPC	33
17.	XML-RPC-Codierung	34
18.	RPC-Benchmark: Hello World	36
19.	RPC-Benchmark: Liste von Strukturen	37
20.	Systemaufbau: Dienstgruppen	40
21.	Verteilte Dienstsuche	49
22.	Assimilieren eines Knotens	51
23.	Datenbankdienste im Überblick	53
24.	Anlegen eines Datensatzes	62
25.	Methode dump_instance	70
26.	Methode dump_object	71
27.	Methode end_object	71
28.	Anlegen eines SSL-Server-Contexts	73
29.	Aktivieren des Moduls PyApache	74
30.	XDR-Packer	81
31.	XDR-Unpacker	81
32.	Sun-RPC Server	82
33.	Sun-RPC Client	82
34.	CORBA Interfacebeschreibung	83
35.	CORBA Server	83
36.	CORBA Client	84
37.	XML-RPC Server	85
38.	XML-RPC Client	85

Abbildungsverzeichnis

39.	Modul <code>defcall</code>	86
40.	RPC-Benchmark: Ping-Pong	92
41.	RPC-Benchmark: Strukturen	93
42.	RPC-Benchmark: Liste von Integer	94

Tabellenverzeichnis

1.	Knotentypen	25
2.	Codierungszeiten	32
3.	Dienststartoptionen	42
4.	Benchmark: Sockets	87
5.	Benchmark: SunRPC	87
6.	Benchmark: CORBA	88
7.	Benchmark: XML-RPC	88
8.	Benchmark: Ping-Pong	89
9.	Benchmark: Hello World	89
10.	Benchmark: Struktur	90
11.	Benchmark: Liste von Integer	90
12.	Benchmark: Liste von Strukturen	91

Tabellenverzeichnis

Verzeichnis der Abkürzungen

ACID	Atomicity, Consistency, Isolation, Durability
AFS	Andrew File System
ASN.1	Abstract Syntax Notation One
BOS	Basic Overseer Server
CA	Certification Authority
CORBA	Common Object Request Broker Architecture
DC	Dublin Core
DII	Dynamic Invocation Interface
DTD	Document Type Definition
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
IDL	Interface Definition Language
IMAP	Internet Message Access Protocol
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
NCA	Network Computing Architecture
OMG	Object Management Group
ORB	Object Request Broker
RDF	Ressource Definition Format
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLI	Transport Layer Interface
UDP	User Datagram Protocol
URL	Uniform Ressource Locator
WWW	World Wide Web
XDR	eXternal Data Representation
XML	eXtensible Markup Language
XP	eXtreme Programming

Verzeichnis der Abkürzungen

1. Problemstellung

1.1. Aufgabenstellung

Ziel ist die Schaffung einer stabilen, erweiterbaren und skalierbaren Infrastruktur für die Bereitstellung von Diensten im Umfeld von Bibliotheken und ähnlichen wissens anbietenden Einrichtungen.

Die Funktionalität des Systems soll mit einer exemplarischen Implementation der Dienste:

- Empfehlungen
- Rezensionen
- Benachrichtigung der Ausleiher

nachgewiesen werden.

Dabei ist auf eine gute Integrationsmöglichkeit in bisherige und zukünftige Systeme zu achten.

1.2. Hintergrund

Systeme zum Verwalten von Bibliotheken werden in letzter Zeit immer komplexer. Gleichzeitig nimmt deren Flexibilität immer weiter ab. Das führt dann zu Aussagen der folgenden Form:

Alle Module sind voll integriert und werden daher auch nicht einzeln, sondern nur als gesamtes Softwarepaket angeboten. Der Benutzer von LIBERO entscheidet, ähnlich wie bei anderen Windows-Programmen, welche Funktionen der Module benutzt werden sollen.

[1]

Die von LIBERO propagierte Vorgehensweise ist unbefriedigend, weil im Laufe der Zeit immer wieder neue Anforderungen an das System gestellt werden, die eine einfache Anpassbarkeit verlangen.

Diese Anforderungen können sich in verschiedenster Weise bemerkbar machen. Einige sind vom konkreten Anwendungsfall unabhängig, zum Beispiel

- höhere Geschwindigkeit,
- niedrigere Ausfallzeiten oder

1. Problemstellung

- besserer Komfort bei der Bedienung.

Anderere betreffen die entsprechende Anwendung, indem bestehende Dienste erweitert oder modifiziert werden sollen, beziehungsweise neue Angebote realisiert werden müssen.

Der Hintergrund dieser Arbeit ist die Modernisierung des Bibliothekssystems der TU Chemnitz. Das bisherige Ausleih-System soll durch ein moderneres ersetzt werden. Da über dieses System zum Zeitpunkt dieser Arbeit noch nicht genug Details bekannt sind, müssen einige Annahmen gemacht werden.

Das neue System wird offene gelegte Schnittstellen bieten, die es ermöglichen:

- Daten aus dem System abzurufen (und zwar vollständig und maschinell weiterverarbeitbar),
- Daten in das System einzubringen bzw. das System zum Ändern von Daten aufzufordern und
- Externe Systeme über Statusänderungen am System benachrichtigen zu lassen.

Ausschlaggebend für die Entwicklungsrichtung dieser Arbeit war auch, dass in letzter Zeit viele leistungsfähige Nutzerarbeitsplätze (über 70 sehr gut ausgestattete PCs) angeschafft wurden, die ausreichend hohe Leistung bieten, um ungenutzte Rechenzeit und Arbeitsspeicher für Teilsysteme des zu entwickelnden Verwaltungssystems abzuschöpfen. Jedoch ist aus der Sicht des neuen Systems damit zu rechnen, dass diese Arbeitsplatzsysteme plötzlich außer Betrieb gehen, weil sie aufgrund von Wartungsarbeiten am Betriebssystem neu gestartet werden müssen oder von einem Nutzer einfach abgeschaltet wurden.

1.3. Anforderungen

Die Bibliothek der TU Chemnitz wird von rund 30.000 Nutzern in Anspruch genommen. Materielle Informationseinheiten (Bücher, Zeitschriften, etc.) werden an mehreren Standorten ausgegeben. Der Bestand beläuft sich auf über eine Million Einheiten, wobei nur rund 350.000 Exemplare im elektronischen Katalog erfasst sind. Diese Zahlen werden in Zukunft noch ansteigen.

Im Zugriff auf das System sind Antwortzeiten von höchstens 2 Sekunden wünschenswert, 5 Sekunden wären gerade noch zumutbar. Sobald Anfragen längere Bearbeitungszeit benötigen, sollte der Nutzer davon in Kenntnis gesetzt werden.

2. Lösungsansätze

Im folgenden sollen mögliche Ansätze vorgestellt werden, die bei der Realisierung eines solch komplexen Softwaresystems genutzt werden könnten. Zuerst werden verschiedene Arten der Architektur vorgestellt. Danach wird auf Mechanismen eingegangen, die genutzt werden können, wenn die aktiven Komponenten über mehrere Prozesse oder gar Rechner verteilt sind. Schließlich werden verschiedene Implementierungsmöglichkeiten gegeneinander abgewogen.

2.1. Architekturalternativen

Die Architektur des Systems wird durch die Verteilung aktiver Systemkomponenten beschrieben.

In *Distributed Computing Environments* [18] werden 5 Funktionsgruppen angegeben, die bei der Verteilung von Anwendungen zu berücksichtigen sind:

- a Endnutzergerät
- b Endnutzerinterfacesystem
- c Anwendung
- d Datenzugriff und –verwaltung
- e Datenspeicherung

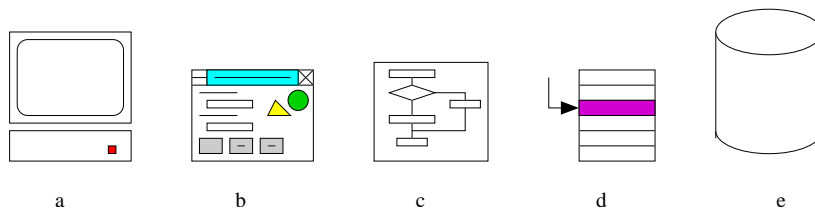


Abbildung 1: Funktionsgruppen

Je nach Verteilung dieser Funktionsgruppen kann man nun von verschiedenen Architekturen sprechen:

2. Lösungsansätze

2.1.1. Terminalsystem mit Fernzugriff

Bei diesem System wird lediglich das Endnutzengerät (a) vom Rest des Systems (bcde) getrennt. Alle anderen Funktionen laufen auf einem zentralen Serversystem ab. Dieses System muss sowohl sehr leistungsfähig als auch extrem zuverlässig sein. Als Endnutzengeräte kämen *X-Terminals* oder klassische alphanumerische Terminals mit beispielsweise einem *Telnet-Clienten* in Frage.

Damit müssen an jedem Arbeitsplatz die notwendigen Terminals oder Terminalemulatoren bereitstehen. Aufgrund der Übertragung jeglicher Nutzeraktionen ist ein relativ hoher Datenverkehr zu erwarten (Mausbewegungen, jede Tastatureingabe). Außerdem entspricht diese Bedienungsphilosophie nicht dem aktuellen Stand der Benutzung, so dass nicht-technische Schwierigkeiten auf Nutzerseite zu erwarten wären. Diese Schwierigkeiten könnten aufgrund ungewohnter Nutzerschnittstellen oder wegen zu langsamer Reaktionen des Systems entstehen.

Daher wird von dieser Architektur Abstand genommen.

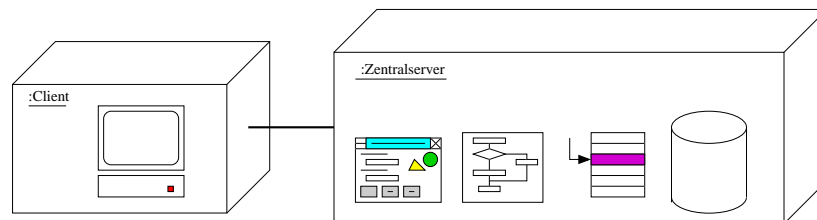


Abbildung 2: Verteilung: Terminalsystem

2.1.2. Hostsystem mit lokaler Ausgabe

Um den zentralen Rechner von einigen Aufgaben zu entlasten und die Ausgabe nutzerorientierter zu gestalten, kann auch die Generierung der Ausgabe auf den Nutzerarbeitsplatz (ab) verlegt werden. Lediglich die Daten werden zur Auswertung an einen Server (cde) geschickt bzw. eine Beschreibung der Ausgabe an den Clienten.

Damit entspricht diese Aufteilung den klassischen 3270 Terminals oder auch einer Realisierung durch Formulare in HTML-Seiten, die mit Hilfe von Programmen im Webserver verarbeitet werden. Auch Spezialprogramme auf RPC-Basis lassen sich unter gewissen Bedingungen in diese Gruppe einordnen.

Besonders die Realisierung der Clientensoftware als quasi „Standardsoftware“ bietet den Vorteil der relativ leichten Installation des Systems auf Clientenseite. Optimalerweise ist keinerlei Installation von besonderer Zusatzsoftware not-

wendig. Allein die Bekanntgabe eines URL (*Uniform Resource Locator*) kann ausreichend sein, um das System den Nutzern zur Verfügung zu stellen, sofern ein Webbrowser installiert ist¹.

Kritisch ist jedoch noch immer die Zentralisierung der Software zu beurteilen. Sowohl die Verarbeitungsleistung als auch die Ausfallsicherheit kann bei einem zentralen System nicht ausreichend genug sein². Gleichzeitig werden jedoch Ressourcen beim Clienten brach liegen gelassen. Dies führt zu einem weiteren Ansatz.

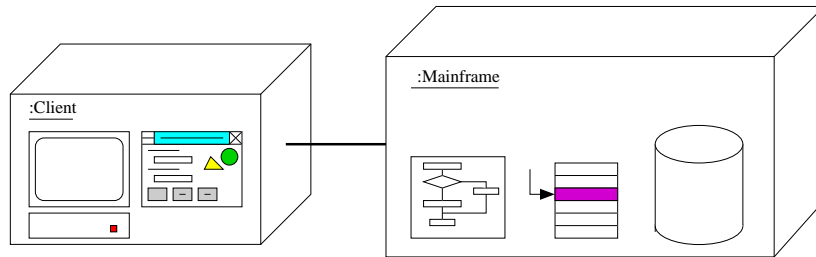


Abbildung 3: Verteilung: Hostsystem

2.1.3. Kooperative Verarbeitung

Um den Server zu entlasten und Rechenressourcen der Clienten zu nutzen, können einige Berechnungen auf den Clienten verlagert werden (Verteilung: abc/cde). Diese Rechnungen können beispielsweise Konsistenzprüfungen bei den Eingaben sein oder die Aufbereitung von Daten in grafischen Darstellungen.

Die Verwendung von clientenseitigen Scriptsprachen (JavaScript), aktiven Inhalten (Java-Applets oder ActiveX-Controls) oder die Implementierung verschiedener Funktionen in speziellen Anwendungsprogrammen ist bei der Umsetzung denkbar. Allerdings setzt dieses Modell eine kompliziertere Software beim Clienten voraus, die möglicherweise nicht immer in der gewünschten Ausführung vorhanden ist (oder überhaupt nicht). Des Weiteren bedingt eine Trennung innerhalb der Anwendung, dass die Anwendung auch die Kommunikation in erheblichen Maße selbst organisieren muss. Das kann beispielsweise proprietäre Übertragungsprotokolle und eine sehr komplizierte und fehleranfällige Steuerung eines globalen Zustands nach sich ziehen.

¹Heutzutage ist das üblich

²Oder einfach zu teuer

2. Lösungsansätze

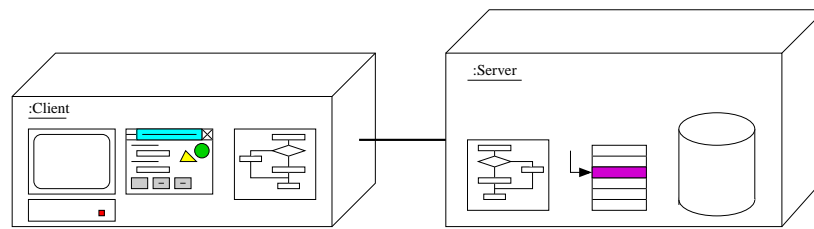


Abbildung 4: Verteilung: Kooperative Verarbeitung

2.1.4. Client/(Datenbank)Server

Um die Kommunikationsarbeit der Anwendung abzunehmen, kann die Schnittstelle zu den entfernten Ressourcen an den Rand der Anwendung verlagert werden. Datenspeicherdienste werden jedoch weiterhin zentral erbracht, um die Sicherheit und Konsistenz der Daten zu gewährleisten (Verteilung: abc/de). Zugriff auf die Daten kann durch Stellvertretermechanismen auf Clientseite in transparenter Form geschehen. Der kritische Punkt stellt jedoch noch immer die Datenbank dar, die als *single point of failure* besondere Aufmerksamkeit verlangt.

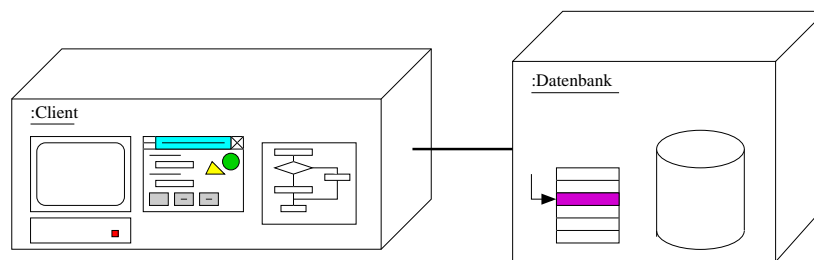


Abbildung 5: Verteilung: Client/(Datenbank)Server

2.1.5. Verteilte Daten

Da viele Anwendungen oft über einen längeren Zeitraum auf denselben Daten operieren, ist es auch denkbar, die Datenverwaltung in gewissen Grenzen auf Clientseite zu verlagern (Verteilung: abcd/de). Primär lässt sich damit das Netzwerk entlasten, jedoch werden Konsistenz sichernde Algorithmen notwendig, die Inkonsistenzen aufgrund der Replikation der Daten verhindern sollen. Aber gerade diese Algorithmen können je nach Verfahren wiederum unerwünschten Datenverkehr hervorrufen.

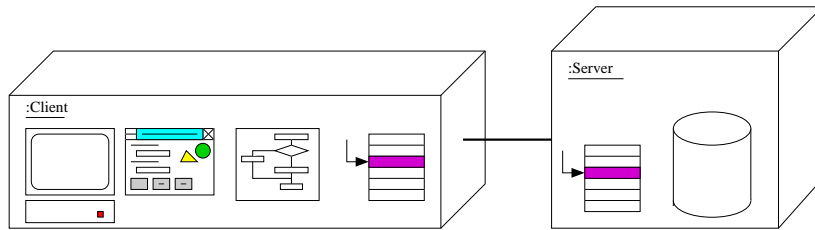


Abbildung 6: Verteilung: Verteilte Daten

2.1.6. Voll verteiltes System

In dieser Extremform sind alle Komponenten über das gesamte Netzwerk verteilt. Die Anwendung kann sowohl lokal als auch auf entfernten Rechnern ablaufen. Ebenso können Daten lokal oder abgekoppelt gespeichert werden. Dieses Modell bietet eine sehr hohe Flexibilität, ist jedoch sehr kompliziert zu realisieren (Verteilung: a/b/c/d/de).

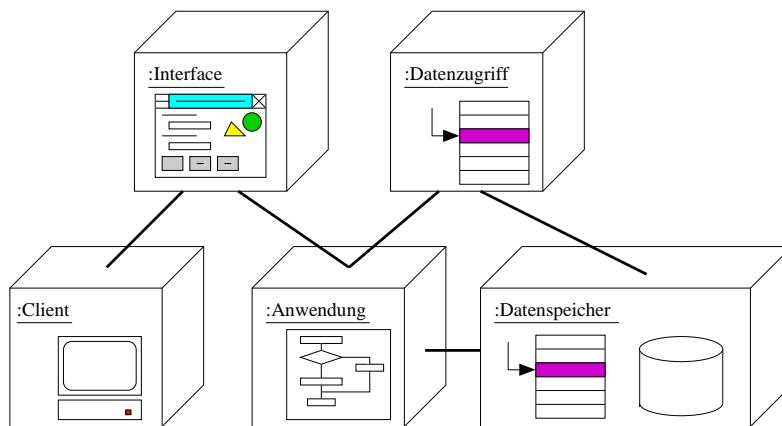


Abbildung 7: Verteilung: Voll verteiltes System

2.1.7. Realisierungsvorschlag

Der von mir aus diesen fünf Alternativen abgeleitete Lösungsvorschlag sieht vor, dass Endnutzergerät und –interfacesystem auf einem einzigen Rechner laufen. Dieser Rechner beherbergt einen WWW–Client in Form eines Webbrowsers. Anwendung, Datenverwaltung und Datenspeicherung werden im Netzwerk verteilt (Verteilung: a/b/c/d/de).

2. Lösungsansätze

Mit dieser Architektur soll sowohl eine leichte Skalierbarkeit als auch eine höhere Ausfallsicherheit aufgrund von redundanten Diensteanbietern erreicht werden. Kapitel 3 wird ausführlich darauf eingehen.

Lamport karikiert: „Ein verteiltes System ist ein System, mit dem ich nicht arbeiten kann, weil irgendein Rechner abgestürzt ist, von dem ich nicht einmal weiß, dass es ihn überhaupt gibt.“ [41] — Diese Schwäche sollte beim Entwurf stets berücksichtigt werden, um dem Zitat nicht zu entsprechen.

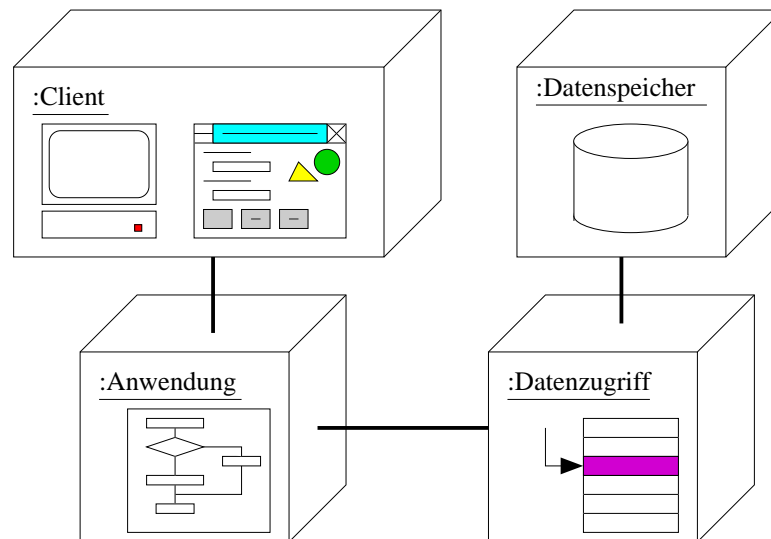


Abbildung 8: Verteilung: Realisierungsvorschlag

2.2. Kommunikation in verteilten Systemen

Im Abschnitt 2.1 wurden verschiedene Architekturen vorgestellt, die Softwarekomponenten auf mehrere physische Knoten verteilen. Wird eine solche Architektur umgesetzt, so benötigt man Technologien, die eine Kommunikation zwischen diesen Komponenten ermöglichen. Diese Technologien werden je nach Umgebung unterschiedlich bezeichnet. Hier soll der Begriff *Remote Procedure Call* (RPC) Verwendung finden, wenngleich die tatsächlich erbrachte Funktion sich vom einfachen Methodenaufruf unterscheiden kann.

Bei der Wahl des RPC-Mechanismus ist zu unterscheiden, ob die Kommunikation zwischen Rechner des System-Nutzers und Server oder zwischen zwei (oder mehr) Servern des Systems erbracht werden muss. Auf Clientenseite können keine zusätzlichen Softwarepakete installiert werden. Auf denen als Server

2.2. Kommunikation in verteilten Systemen

dienenden Maschinen ist das leichter möglich.

Bei der Kommunikation zwischen Nutzer–Rechner und Server ist daher besonders darauf zu achten, dass der RPC–Mechanismus beim Clienten zur Verfügung steht und auch von weit entfernten Rechnern aus verwendbar ist (Firewalls!). Bei Server–Server–Kommunikation stehen Transparenz und Geschwindigkeit im Vordergrund.

An die RPC–Mechanismen werden von verschiedenen Autoren folgende Anforderungen gestellt:

Transparenz: Die Verwendung von RPC–Methoden darf für den Anwendungsentwickler nicht wesentlich komplizierter sein als der Aufruf lokaler Methoden. Daher ist es nicht sinnvoll, wenn beispielsweise spezielle Call–Objekte angelegt werden müssen, die dann aktiviert werden oder in der Parameter- und Ergebnisübergabe nur bestimmte Typen erlaubt sind. Auch Fehlerzustände (*Exceptions*) sollten in angemessener Weise übertragen werden können.

Die eventuell auftretende Erfordernis, Verteilungsaspekte explizit im Programm zu benennen, muss sich auf die wirklich erforderlichen Parameter beschränken, damit nicht das Gefummel um die Verteilung die Formulierung des Programms in fehlerträchtiger Weise erschlägt. [43]

Ein Verzicht auf Interfacebeschreibungssprachen (*Interface Definition Language* — IDL) ist ebenfalls wünschenswert, um den Entwicklungsprozess nicht unnötig zu verkomplizieren.

Geschwindigkeit: In verteilten Anwendungen wird prinzipbedingt oft über Rechengrenzen hinweg kommuniziert. Daher hat die Geschwindigkeit des Kommunikationsprotokolls einen erheblichen Einfluss auf die Geschwindigkeit der Anwendung. Überflüssige Verzögerungen sollten vermieden werden. Eine Ursache für Verzögerungen sind Übertragungsfehler, die zu einem Verlust der zu übertragenden Daten führt. Daher sollten zumindest für die Kommunikation Client–Server verlässliche Transportmechanismen eingesetzt werden, um eine anwendungsseitige Qualitätssicherung überflüssig zu machen. Ebenso sind Protokolle mit einem hohen Verwaltungsaufwand kritisch zu bewerten.

Eine weitere Verbesserung der Leistung ist mit zustandsbehafteten Servern zu erreichen. Anstatt Daten, die über mehrere Aufrufe hinweg benötigt werden, mehrfach zu versenden, können diese gleich beim Server hinterlegt werden. Damit steigt allerdings auch die Komplexität des Systems.

2. Lösungsansätze

Die Geschwindigkeit lässt sich auch durch intelligentes Zusammenfassen von mehreren Botschaften in eine *Sammelnachricht* oder das Wiederverwenden von Kommunikationskanälen erreichen.

Portabilität: Die eingesetzten Mechanismen müssen auf jeder Plattform ausführbar sein, auf der das Programmsystem eingesetzt wird. Dabei ist es auf der Clientenseite nicht sinnvoll, wenn aufwändige Programmpakete installiert werden müssen. Auch hier gilt wieder, dass die Systemsoftware die entsprechende Funktionalität am besten gleich mitbringt.

Des Weiteren ist es wünschenswert, wenn aus der Wahl des RPC-Mechanismus in der Praxis keine zu starke Bindung an eine Programmiersprache entsteht. Eine theoretische Möglichkeit allein reicht nicht aus — es muss verwendbare Implementationen geben.

Sicherheit: Die Anforderungen an die Sicherheit umfassen ein ganzes Spektrum an Teilanforderungen. Insbesondere Authentifizierung und Autorisierung der Nutzer sowie Integritätssicherung und Vertraulichkeit der Informationen sind dabei zu nennen. Dabei ist es unerheblich, ob es sich um die Kommunikation zwischen Client und Server oder die Interserverkommunikation handelt. Es ist nicht ausreichend, auf ein Protokoll zu verweisen, welches eine binäre³ Codierung der Daten benutzt, um mit einer „Verschleierungstaktik“ Sicherheit vorzugaukeln. Die Realisierung der Sicherheit kann dabei sowohl anwendungsentkoppelt stattfinden (z.B. Vertraulichkeit der Übertragung) als auch innerhalb der Anwendung (Autorisierung zur Nutzung bestimmter Teildienste).

Flexibilität: Neben dem üblichen synchronen 1:1-Aufruf können je nach Anwendungsfall auch Broadcast- und Multicast-Aufrufe von Methoden notwendig sein. Diese Funktionalität sollte bereits im RPC-Mechanismus verfügbar oder leicht nachzuimplementieren sein. Bestimmte Aufgaben erfordern auch asynchrone Aufrufe, die ebenfalls von der RPC-Bibliothek erbracht werden sollten.

Im Folgenden sollen verschiedene RPC-Systeme auf die Anforderungen hin untersucht werden.

2.2.1. Socket-Programmierung

BSD-Sockets oder das *Transport Layer Interface* (TLI) [37] (kurz als Sockets bezeichnet) zählen nicht unbedingt zu RPC-Systemen, sind aber trotzdem ge-

³Im Gegensatz zu Klartextprotokollen wie HTTP, SMTP oder IMAP

2.2. Kommunikation in verteilten Systemen

eignet, um Daten zwischen Programmen über das Netzwerk auszutauschen.

Transparenz kann mit klassischer Socket-Programmierung kaum erreicht werden. Natürlich greifen alle anderen Kommunikationsmethoden auch auf Sockets zurück. Allerdings ist es im Rahmen der hier zu erstellenden Anwendung nicht sinnvoll, eine eigene Kommunikationsbibliothek von Grund auf neu zu entwickeln.

Vorteile von Sockets liegen auf dem Gebiet der Geschwindigkeit. Da man sich recht nahe am Kommunikationsmedium befindet, müssen die Daten weniger langsame Zwischenschichten passieren. Indem einmal geöffnete Kanäle wiederverwendet werden, reduziert sich die Anzahl zeitaufwändiger Prozeduren zum Aufbau von Verbindungen.

Das BSD-Socket-API ist auf den meisten Betriebssystemen und in den gebräuchlichen Programmiersprachen anzutreffen. Selbst Browser, die lediglich HTML-Seiten darstellen, kann man mit Hilfe von aktiven Inhalten⁴ meist zur Kommunikation mit Socket-Servern befähigen. Es fehlen jedoch Mechanismen, die einen Datenaustausch komplexerer Datenstrukturen realisieren. Die einzige bekannte Dateneinheit ist das *Oktett*. Selbst für die Übertragung von Zahlen, die mehr als ein Oktett Speicherplatz belegen, sind schon besondere Vorkehrungen zu treffen.

Sicherheit im Sinne der Anforderungen gibt es bei Sockets nicht. Authentifizierung und Vertraulichkeit könnten mit Hilfe von *Secure Sockets Layer*-Bibliotheken (SSL) [21] realisiert werden. Allerdings ist die Authentifizierung ausschließlich mit Zertifikaten nur zwischen Servern nutzbar. Clienten über ausgegebene Zertifikate zu identifizieren, ist zum aktuellen Zeitpunkt noch nicht praktikabel⁵.

Broad- und Multicast sind auf Socketebene nur mit datagrammorientierten Transportprotokollen möglich, weshalb wieder Aufwand entsteht, um applikationsseitig für eine gesicherte Übertragung zu sorgen. Asynchronität ist hingegen leicht zu realisieren, indem Schreiben und Lesen auf den Socketobjekten von unterschiedlichen Aktivitäten realisiert wird.

Zusammenfassend lässt sich die Aussage treffen, dass Sockets sowohl für die Kommunikation zwischen Clienten und Servern als auch für die Kommunikation zwischen Servern nicht zu empfehlen sind.

2.2.2. Klassischer RPC

Unter dieser Überschrift werden RPC-Systeme wie SunRPC [36] oder NCA-RPC zusammengefasst (und in diesem Abschnitt kurz als RPC bezeichnet). Sie

⁴Java-Applets, ActiveX-Controls

⁵Ein Ausgabe von Zertifikaten auf Chipkarten könnte diese Situation ändern.

2. Lösungsansätze

werden sowohl in Spezialanwendungen als auch in Infrastrukturanwendungen (NFS, YP/NIS, ...) eingesetzt.

RPC verwendet IDLs, um mit Hilfe spezieller Übersetzungsprogramme eine Abbildung von Programmen, Funktionen und komplexen Datentypen in eine Zielsprache zu realisieren. Unterschiede zu lokalem Prozeduraufruf bestehen bei der Verwendung der Funktionen lediglich darin, dass als Parameter stets ein „Verbindungsdeskriptor“ zusätzlich übergeben werden muss, sowie ausschließlich *call by value* möglich ist. Der Verbindungsdeskriptor wird vor dem Aufruf der ersten Funktion des entfernt abgearbeiteten Programms mit einem zusätzlichen Funktionsaufruf erzeugt. Bei der Erzeugung des Deskriptors ist der Rechner zu benennen, auf dem sich der Server befindet, welcher die entfernten Prozeduren abarbeiten wird.

Die Umwandlung aller Daten in ein gemeinsames Austauschformat bringt Geschwindigkeitsnachteile mit sich. Die Umwandlung erfolgt in automatisch aus der IDL generierten Bibliotheken. Da diese Bibliotheken die Codierung in expliziter Form beschreiben, ist der Geschwindigkeitsverlust nicht ganz so hoch wie bei anderen Codierungsverfahren, die den Datenstrom erst noch interpretieren müssen (beispielsweise ASN.1).

In der praktischen Umsetzung von RPC-Programmen ist man auf C festgelegt, wenn man eine komplette Unterstützung aller Fähigkeiten möchte⁶. Damit ist man als Softwareentwickler automatisch darauf angewiesen, Binärprogramme für jede gewünschte Plattform zu produzieren. Angesichts der sich schnell wandelnden Plattformvielfalt (vor 5 Jahren Workstations mit RISC-Prozessoren, heute PCs mit Intel CPUs, in 5 Jahren vielleicht eingebettete Systeme mit Spezial-CPU) ist dies nicht vorteilhaft. Für einige Scriptsprachen gibt es prototypenhafte Unterstützung für RPC. Allerdings beschränkt diese sich zumeist auf die Umwandlung einfacher Datentypen in das Austauschformat. Komplexe Datentypen müssen vom Programmierer explizit durch eigene Umwandlungsroutinen konvertiert werden.

Eine sichere Authentifizierung ist bei RPC mit Hilfe des Diffie-Hellman-Verfahrens oder *Kerberos* realisierbar [20]. Allerdings bieten diese Mechanismen keine Integritäts- und Vertraulichkeitssicherung. Erst in Verbindung mit einem sicheren Transportprotokoll (z.B. SSL) können diese beiden Sicherheitsaspekte realisiert werden.

Mit RPC sind sowohl Broadcast als auch Multicast von Methodenaufrufen möglich. Dabei ist jedoch wie auch bei der Verwendung von Sockets auf Datagrammtransportprotokolle zurückzugreifen (mit allen damit verbundenen Nachteilen). Ebenso werden asynchrone Rufe unterstützt, welche in der Reihenfolge

⁶Teilfunktionalität wird auch für andere Sprachen angeboten.

2.2. Kommunikation in verteilten Systemen

der Inauftraggebung abgearbeitet werden. Diese funktionieren allerdings nur mit datenstromorientierten Transportprotokollen wie zum Beispiel TCP (*Transmission Control Protocol*).

Ein großer Nachteil von RPC ist die Bindung an die Sprache C, wenn man die volle Funktionsvielfalt möchte, und damit die Notwendigkeit von kompilierten Applikationsprogrammen. Außerdem führt die strenge Typisierung der Interfaces zu erheblichem Entwicklungsaufwand, der gerade im *Rapid-Prototyping* störend wirkt.

2.2.3. CORBA

CORBA (*Common Object Request Broker Architecture*) [22, 41] ist ein RPC-System, welches von der *Object Management Group* (OMG) entwickelt wurde. Im Gegensatz zu den bisher vorgestellten Bibliotheken ist CORBA objektorientiert aufgebaut. Aufgrund der Datenkapselung der Objekte lassen sich Probleme in heterogenen Umgebungen leichter handhaben. Aspekte der Verteilung lassen sich vor dem Programmierer besser verbergen, weil Zugriff auf den Zustand der Objekte nur durch entsprechende Methoden erlangt werden kann. Dies sorgt für leichter wartbaren Quellcode.

Der Kern der CORBA-Architektur ist der *Object Request Broker* (ORB). Dieser übernimmt laut [22] unter anderem die Transparenzsicherung für folgende Teilgebiete:

Lokation: Sind die Objekte lokal oder entfernt?

Zugriffspfad: Wie gelangen die Botschaften zu den Objekten?

Relokation: Wie werden Objekte zwischen Maschinen verlagert?

Kommunikation: Welche Protokolle werden verwendet?

Speicherung: Wie werden Objekte persistent gespeichert?

Des Weiteren realisiert der ORB auch die Abstraktion von Betriebssystem, Programmiersprache und Maschinentyp.

Wie auch bei Sun-RPC müssen die Objekte mit einer IDL beschrieben werden. Damit sind dann alle Nachteile von IDLs verbunden. Im Gegensatz zu Sun-RPC existieren allerdings Compiler für verschiedene Zielsprachen.

Das Inter-ORB-Protokoll wird in binärer Form abgewickelt. Dadurch entfällt oft ein aufwändiges Umwandeln der Daten. Ein GIOP (*General InterORB Protocol*) [25] sorgt in den neueren Implementationen für standardisierte Kommunikation zwischen ORBs verschiedener Hersteller. Ältere ORBs verwenden

2. Lösungsansätze

jedoch proprietäre Protokolle, bei denen eine Fehlersuche nur schwer möglich ist, sofern keine Dokumentation bereitgestellt wurde.

Es gibt CORBA-Implementationen für sehr viele Programmiersprachen und Betriebssysteme. Damit ist CORBA trotz seiner Binärlastigkeit überaus portabel (was auch eine der ursprünglichen Anforderungen war). Eine Gefahr besteht, wenn Erweiterungen des ORBs verwendet werden, die nicht im Standard enthalten sind. Man legt sich damit möglicherweise auf einen Anbieter fest.

Sicherheitsmechanismen zur Authentifizierung sind vorgesehen und werden mit speziellen *Services* realisiert. Vertraulichkeit auf Transportebene muss allerdings meist mit SSL oder ähnlichem nachgebessert werden.

Der statische Methodenaufruf erfolgt immer synchron. Der einzige Unterschied zum Aufruf von Methoden an lokalen Objekten besteht darin, dass ein Proxy-Objekt für das entfernte Objekt angelegt werden muss. Mit Hilfe des *Dynamic Invocation Interface* (DII) lassen sich ebenfalls synchrone Aufrufe realisieren, allerdings werden die Aufrufe dynamisch zusammengestellt. Diese Aufrufobjekte können ebenfalls eingesetzt werden, um verzögerte (asynchrone) Aufrufe oder Einwegnachrichten abzusetzen.

CORBA Implementierungen sind sehr komplexe und umfangreiche Softwarepakete. Damit verbunden sind umfangreiche Laufzeitsysteme, die aufgrund der hohen Ressourcenanforderungen einen vom Rechnernutzer unbemerkten Betrieb kaum ermöglichen. Allerdings definiert CORBA eine Reihe von Diensten, deren Funktionalität auch in anderen verteilten Systemen benötigt werden könnte (nach [41]):

- Lifecycle Service (Erzeugen, Löschen, Bewegen und Kopieren von Objekten)
- Relationship Service (Verbundobjekte durch Gruppierungen erschaffen)
- Naming Service (Auflösung von Namen in Objektreferenzen)
- Persistent Object Service (Dauerhafte Speicherung von Objekten)
- Externalization Service (Abspeichern und Einlesen von Objekten)
- Event Service (Ereigniskanal zur Kommunikation)
- Object Query Service (SQL-Schnittstelle)
- Object Properties Service (Dynamisches Hinzufügen von Attributen zu Objekten)
- Object Transaction Service (Transaktionen im Zugriff auf Objekte)

2.2. Kommunikation in verteilten Systemen

- Concurrency Service (Dienste zur Realisierung von Locking)
- Licensing Service (Lizenzverwaltung und Accounting)
- Trader Service (Bestimmung der geeignetsten Alternative)
- Security Service (Authentifikation und Autorisation)
- Secure Time Service (Bereitstellung einer global einheitlichen Zeit)
- Object Collection Service (Verwaltung von Objektlisten, insbesondere für den Query Service)
- Object Startup Service (Starten weiterer Dienste)

2.2.4. HTTP

HTTP (*Hyper Text Transfer Protocol*) [38] ist ursprünglich nicht für den entfernten Aufruf von Methoden vorgesehen gewesen, sondern zum Übertragen von Hypertextdateien und darin eingebetteten Binär-Objekten. Werden die Daten jedoch bei Anforderung dynamisch erzeugt, so kann diese Verwendung ebenfalls als eine Art RPC angesehen werden.

HTTP wird zumeist zwischen speziellen Clienten-Programmen (*Browser*) und entsprechenden Servern eingesetzt. Sollen Server in verteilten Systemen selbst die Rolle von Clienten übernehmen, so müssen diese entsprechende HTTP-Anfragen erzeugen. Dazu können die im HTTP definierten Methoden GET und POST verwendet werden. Parameter für den Methodenaufruf sind vom Programmierer in den URLs zu codieren oder im HTTP-Protokollkopf mitzuschicken. Die Rückgabewerte können in beliebigen Formaten beschrieben sein. Der HTTP-Standard macht dazu keine Vorgaben. Als transparent ist diese Programmiermethode nur dann anzusehen, wenn die entsprechende Sprache ein Metalevel bietet, welches Methodenaufrufe als Datenobjekte verwalten kann.

Die Geschwindigkeit ist stark von der verwendeten Codierung der Daten abhängig. Sollen verschiedene Systemarchitekturen integriert werden, so ist ein möglichst portables Format zu wählen. Anbieten würden sich Binärdaten in einem anwendungsspezifischen Format, eine Binärcodierung nach ASN.1 oder eine Darstellung in einem 7-Bit-Zeichensatz, bei der alle Werte als Text repräsentiert sind⁷. Alle diese Codierungen führen zu mehr oder weniger aufwändigen (De)Codierungsfunktionen.

⁷47 wird also nicht als einzelnes Byte mit dem Wert 47 übertragen, sondern durch die Ziffernsymbole 4 und 7 in Folge.

2. Lösungsansätze

Ein weiterer Einflussfaktor auf die Geschwindigkeit ist die Art und Weise, wie aufeinander folgende HTTP-Anfragen zwischen Client und Server übertragen werden. Die Protokollversion 1.1 erlaubt es, mehrere Anfragen nacheinander über eine einzige TCP-Verbindung zu übertragen. Die Vorgängerversion 1.0 hingegen benötigt für jede Übertragung eine neue TCP-Verbindung.

Da HTTP als textbasiertes Protokoll auf TCP/IP aufgesetzt wird, ergeben sich die gleichen Portabilitätsmerkmale wie für BSD-Sockets.

Verschlüsselte Kommunikation ist ebenfalls nur mit Hilfe von zusätzlichen Protokollen realisierbar. Für die Authentifizierung stehen allerdings Protokollmechanismen zur Verfügung, die in Form spezieller Zusatzinformationen übertragen werden können. Die Auswertung dieser Informationen obliegt allerdings dem Anwendungsprogramm.

Eine andere als die synchrone 1:1-Kommunikation ist aufgrund der Bindung an TCP/IP nicht vorgesehen.

Trotz dieser Nachteile eignet sich HTTP sehr gut für die Kommunikation zwischen dem Rechner des Nutzers und einem Server, der den Zugang zum Anwendungssystem schafft. Der wesentlichste Grund ist die Verfügbarkeit von entsprechenden Clienten-Programmen auf praktisch jedem Arbeitsplatzrechner.

2.2.5. XML-RPC

XML-RPC erweitert HTTP um eine einheitliche Darstellungsschicht für die Anwendungsdaten. Es existieren momentan zwei verschiedene Hauptrichtungen von XML-DTDs (*eXtensible Markup Language-Document Type Definition*) für XML-RPC: SOAP (*Simple Object Access Protocol* [17] und XML-RPC [40] im engeren Sinne. Erstere ist ein wenig neuer und verwendet XML-Namensräume, während letztere einfachere Tags verwendet. Des Weiteren wird bei SOAP der Name der aufzurufenden Methode zusätzlich im HTTP-Header übertragen. Beide Standards bilden Objekte auf Strukturen ab, was für die Entwicklung einer objektorientierten Anwendung nicht von Vorteil ist. Eine Lösung für dieses Problem wird in 4.2 vorgestellt. Außerdem dürfen bei der Codierung im XML-RPC-Format nur Zeichenketten als Schlüsselwerte für Dictionaries verwendet werden sowie keine mehrfach referenzierten Objekte serialisiert werden⁸.

In Bild 9 bzw. 10 sieht man die Codierung eines Aufrufes der Methode `callThisMethod` mit folgenden Parametern:

- Zeichenkette "abc"
- Ganzzahl 1234

⁸SOAP spricht hingegen von *multi-reference Values*.

2.3. Implementierungsumgebung

- Gleitkommazahl 56.78
- Tupel ("a", 2, "c")
- Liste [12, 34, 56]
- Dictionary {"1": 2, "3": 4}

Die restlichen Eigenschaften werden durch die darunter liegende HTTP-Implementierung bestimmt.

Ein großer Vorteil von XML-RPC ist die leichte Implementierbarkeit in vielen Sprachen. Dies wird allerdings mit Geschwindigkeitseinbußen bezahlt. In Systemen, bei denen der Kommunikationsanteil aber nicht sehr hoch ist, spielt dieser Nachteil keine große Rolle. Stattdessen kann der Entwickler von der Portabilität und Flexibilität profitieren.

Auch für XML-RPC gilt, dass echte Transparenz nur dann möglich ist, wenn auf einer Metaebene die Methodenaufrufe abgefangen werden, um diese über die Netzverbindung zu entfernten Rechnern zu leiten. Wenn dies der Fall ist, so kann man ohne IDLs auskommen und schnell einsatzfähige Anwendungsprototypen realisieren.

2.3. Implementierungsumgebung

2.3.1. Programmierparadigmen

Bei der Entwicklung eines komplexen Systems sollte man sich frühzeitig für ein Programmierparadigma und damit auch eine Entwurfsmethode [28] entscheiden. Zur Auswahl stehen der strukturierte Entwurf, das objektorientierte Design oder die — von mir als solche bezeichnete — ereignisorientierte Programmierung.

Strukturierte Programmierung eignet sich bei Projekten, deren Funktionalität zu Beginn der Entwicklungsphase nahezu komplett erfasst werden kann. Während der Entwicklung durchläuft das Projekt mehrere Stufen, wobei mit jedem Übergang verschiedene Transformationen durchgeführt werden. Änderungen an der Spezifikation verursachen einen erneuten Durchlauf des Entwicklungsprozesses.

Beim **objektorientierten Entwurf** kann das System in mehrere Teilkomponenten zerlegt werden, die weitgehend unabhängig voneinander entwickelt werden können. Diese Teilkomponenten können kleiner und damit überschaubarer sein. Damit sinkt der Entwicklungsaufwand. Außerdem wird im objektorientierten Entwurf die Anzahl der Transformationen erheblich reduziert, weil

2. Lösungsansätze

```
<?xml version='1.0'?>
<methodCall>
<methodName>callThisMethod</methodName>
<params>
<param>
<value><string>aaa</string></value>
</param>
<param>
<value><int>1234</int></value>
</param>
<param>
<value><double>56.78</double></value>
</param>
<param>
<value><array><data>
<value><string>a</string></value>
<value><int>2</int></value>
<value><string>c</string></value>
</data></array></value>
</param>
<param>
<value><array><data>
<value><int>12</int></value>
<value><int>34</int></value>
<value><int>56</int></value>
</data></array></value>
</param>
<param>
<value><struct>
<member>
<name>3</name>
<value><int>4</int></value>
</member>
<member>
<name>1</name>
<value><int>2</int></value>
</member>
</struct></value>
</param>
</params>
</methodCall>
```

Abbildung 9: XML-RPC-Codierung

2.3. Implementierungsumgebung

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:enc='http://schexas.xmlsoap.org/soap/encoding/'
  xmlns:lab='http://www.pythonware.com/soap/'
  xmlns:xsd='http://www.w3.org/1999/XMLSchema/'
  xmlns:xsi='http://www.w3.org/1999/XMLSchema/instance/'
  env:encodingStyle='http://schexas.xmlsoap.org/soap/encoding/'>
<env:Body>
<callThisMethod>
<v.1 xsi:type='xsd:string'>aaa</v.1>
<v.2 xsi:type='xsd:int'>1234</v.2>
<v.3 xsi:type='xsd:double'>56.78</v.3>
<v.4 enc:arrayType='xsd:ur-type[3] '>
<v xsi:type='xsd:string'>a</v>
<v xsi:type='xsd:int'>2</v>
<v xsi:type='xsd:string'>c</v>
</v.4>
<v.5 enc:arrayType='xsd:ur-type[3] '>
<v xsi:type='xsd:int'>12</v>
<v xsi:type='xsd:int'>34</v>
<v xsi:type='xsd:int'>56</v>
</v.5>
<v.6 xsi:type='lab:PythonDict'>
<k xsi:type='xsd:string'>3</k>
<v xsi:type='xsd:int'>4</v>
<k xsi:type='xsd:string'>1</k>
<v xsi:type='xsd:int'>2</v>
</v.6>
</callThisMethod>
</env:Body>
</env:Envelope>
```

Abbildung 10: SOAP-Codierung

2. Lösungsansätze

die Methodik von der Analyse bis zur Implementierung die gleichen Modellierungselemente verwendet. Eine Klasse — beispielsweise — ist eine Klasse in der Analyse und bleibt eine Klasse im Quellcode.

Im Modell des **ereignisorientierten Ansatzes** wird das objektorientierte System dahingehend erweitert, dass Objekte in Daten-, Dienst- und Kommunikationsobjekte unterteilt werden. Datenobjekte dienen lediglich der Speicherung der Informationen und sind passiv. Dienstobjekte führen Operationen mit den Datenobjekten aus. Auszuführende Transaktionen werden meist von mehreren Dienstobjekten erbracht. Die Kommunikation zwischen den Dienstobjekten wird von Kommunikationsobjekten übernommen.

Diese Trennung ermöglicht eine bessere Verteilbarkeit der Anwendungen, da für die verschiedenen Objektklassen unterschiedliche Realisierungen verwendet werden können. Während Datenobjekte als einfache Strukturen, Zeilen in einer Relationalen Datenbank oder XML-Datensätze repräsentiert werden können, werden Dienstobjekte von Prozessen repräsentiert. Dienstobjekte sind auch dadurch gekennzeichnet, dass sie üblicherweise nicht ihren Ort verändern, während Datenobjekte durch das System geschickt werden können. Datenobjekte sind langlebig und werden oft auf nicht flüchtigen Datenspeichern abgelegt. Kommunikationsobjekte hingegen sind nur kurzzeitig existent, wenn sie zur Kommunikation zwischen Dienstobjekten erzeugt werden. Kommunikationsobjekte müssen nicht unbedingt explizit angelegt werden. Sie können auch mit Hilfe einer Laufzeitbibliothek automatisch generiert werden. Zur Beschreibung von Kommunikationsobjekten können einfache Textbausteine (beispielsweise HTTP-Header), Binärstrukturen (Sun-RPC, CORBA) oder XML-Strukturen (XML-RPC) verwendet werden.

2.3.2. Programmiersprachen

Von nicht untergeordneter Rolle ist in der Praxis auch die Frage der Programmiersprache, die für die Umsetzung eines Projektes genutzt werden soll.

Ein, wenn nicht sogar das Kriterium für die Wahl der Sprache, ist die **Wartbarkeit**. Im Laufe der Zeit werden immer wieder Änderungen und Erweiterungen an den Programmen vorgenommen werden müssen. Daher ist es nicht sinnvoll, eine exotische Sprache zu verwenden, deren Zukunft von nur wenigen Entwicklern abhängig ist und die von nur wenigen Personen beherrscht wird. Ein weiteres Kriterium ist eine hohe **Verfügbarkeit** der Sprache auf vielen Plattformen bzw. die **Integration** der Sprache in angrenzende Technologien (HTTP, RPC-Mechanismen, Datenbankzugriff). Auch die **Geschwindigkeit** der Laufzeitumgebung sollte Einfluss auf die Wahl der Sprache haben, wobei man bedenken sollte, dass innerhalb von etwa zwei Jahren die Leistung der Rechner

sich mehr als verdoppelt haben wird.

In die nähere Auswahl kamen bei dieser Arbeit C++, Java [2], PHP [3] und Python [4].

Lutz Prechel's Artikel über den Vergleich von C, C++, Java, Perl, Python, Rexx und TCL [32] erbringt einen wesentlichen Beitrag für die Entscheidung dieser Frage. Für die Bewertung des Artikels sollte man allerdings bedenken, dass die Untersuchung zwar an ca. 80 Teilnehmern, aber nur an einem einzigen Problem stattfand⁹.

C++: Programme in C++ — so zeigt die Praxis — tendieren dazu, groß und funktionsbeladen zu sein. Eine Ursache könnte darin begründet liegen, dass es erheblich einfacher ist, innerhalb eines Prozesses zu kommunizieren als mit anderen Prozessen Daten auszutauschen. Außerdem erlaubt C++ aufgrund der Wurzeln in der Sprache C sehr undurchsichtige Programme zu schreiben — an manchen Stellen ist dies sogar notwendig, um die gewünschten Ziele zu erreichen. Diese beiden Probleme reduzieren die Wartbarkeit der Programme, was unter den gestellten Anforderungen von großem Nachteil ist. Die Geschwindigkeit von C++-Programmen unterscheidet sich nicht wesentlich von Programmen in den anderen Sprachen. Die Verfügbarkeit eines C++-Compilers ist auch auf den meisten Plattformen gegeben. Zwar unterstützen die aktuellen Compiler den ANSI-Standard, jedoch sind bestimmte Funktionsbibliotheken systemnah „optimiert“ und verhindern so die Portierung umfangreicherer Projekte auf andere Systeme. Dies ist ein weiterer Nachteil, der gegen C++ spricht.

Java: Ein großer Nachteil von Java-Programmen liegt in der relativ hohen Startzeit der sehr umfangreichen Laufzeitumgebung. Ist diese einmal geladen, so werden die Programme dank *Just in Time*-Compilierung meist ausreichend schnell ausgeführt. Auffällig ist auch, dass die Qualität der für den Artikel geschriebenen Java-Programme sehr weit auseinander liegt. Damit kann der Nachteil entstehen, dass nicht die optimale Systemleistung erbracht wird, weil der Programmierer nicht genug „Tricks“ kennt, um bestimmte Systemdefizite zu umgehen. Die Tricks begründen sich aus Eigenarten der *Java Virtual Machine* (JVM), die beispielsweise das Anlegen von sehr vielen kleinen Objekten mit Geschwindigkeitseinbußen bestraft. Auch die Sprachdefinition selbst begründet beispielsweise das Fehlen von Referenzübergabe von primitiven Datentypen, so dass ein üblicher Trick aus der Verwendung von *Wrapper*-Objekten besteht. Die

⁹Die Aufgabe bestand in der Realisierung einer Abbildung von mehrstelligen Zahlen auf existierende Worte eines Wörterbuches, wobei den einzelnen Ziffern bestimmte Buchstaben zugeordnet wurden.

2. Lösungsansätze

Realisierung der Laufzeitumgebung als virtuelle Maschine begründet eine sehr leichte Portierbarkeit der Anwendung auf andere Plattformen, aber auch einen relativ hohen Speicherbedarf der Laufzeitumgebung. Dieser hohe Speicherbedarf ist der wesentliche Grund für die Ablehnung von Java. Die Laufzeitumgebung erbringt jedoch eine sehr große Menge an Zusatzfunktionalitäten, die damit auch auf jeder Systemplattform verfügbar sind. Es werden — bis auf XML-RPC — alle in Abschnitt 2.2 genannten Mechanismen von der Basisbibliothek unterstützt.

PHP: Auf PHP wird in dem genannten Artikel nicht eingegangen. Daher können auch keine konkreten Messwerte angegeben werden, was die Geschwindigkeit betrifft. PHP war ursprünglich als Erweiterung von HTTP-Servern gedacht und wird auch heute noch überwiegend in dieser Form eingesetzt. Es ist aber auch möglich, PHP-Programme in einem eigenständigen Interpreter ablaufen zu lassen. Auch wenn keine Experimente mit dem eigenständigen Interpreter durchgeführt wurden, kann man die Vermutung äußern, dass das Verhalten der PHP-Laufzeitumgebung über lange Laufzeiten kritisch zu bewerten ist. Der Grund für diese Vermutung liegt in der Tatsache begründet, dass die PHP-Laufzeitumgebung in einem HTTP-Server nur kurze Zeit verwendet und danach komplett aus dem Speicher entfernt wird. Dadurch ist es den Entwicklern der Laufzeitumgebung erlaubt, die dynamische Speicherverwaltung nicht zu 100% perfekt zu implementieren, sondern zum Vorteil der Geschwindigkeit auch Speicher nicht oder nur unvollständig freizugeben. Bei länger laufenden Prozessen führt dies aber unweigerlich zu einer Anhäufung des gesamten Speichers bei diesem Prozess. PHP hat aber auch noch andere Nachteile. Einer besteht darin, dass PHP-Programme sehr leicht an Strukturierung verlieren, da von der Sprache selbst keine solche Strukturierung nahegelegt wird. Es ist jederzeit möglich, Klassen, freie Prozeduren und globalen Code miteinander zu mischen. Bei Einbettung in eine HTML-Seite ist zudem eine beliebige Durchmischung der Programmlogik (PHP) mit dem Nutzerinterface (HTML) möglich. Unter dem Gesichtspunkt der Wartbarkeit und Flexibilität ist das eine Katastrophe. Als Scriptsprache lässt sich PHP auf fast jeder Systemumgebung einsetzen bzw. als *Plug-In* in die meisten HTTP-Server integrieren.

Python: Die Sprache Python scheint von den von Prechelt getesteten Sprachen die geeignetste zu sein. Die Geschwindigkeit der Programme ist höher als bei C++ und Java, der Speicherverbrauch ist niedriger als bei Java und die Programme sind erheblich kürzer als in C++ und Java. Da die Anzahl von Zei-

len pro Zeiteinheit, die ein Programmierer mit Python schreibt, sehr hoch ist¹⁰, sind Python-Programme wesentlich effizienter herzustellen. Kürzere Programme sind meist auch besser zu warten, da diese oft leichter zu verstehen sind¹¹. Python bietet selbst eine relativ umfangreiche Klassenbibliothek an, wobei allerdings Bibliotheken für die RPC-Mechanismen nicht im Standardumfang enthalten sind. Es gibt allerdings Drittanbieter, die Python-Implementierungen der gängigen RPC-Mechanismen anbieten. Als Scriptsprache ist Python auch auf allen Plattformen verfügbar, die als Laufzeitumgebung für dieses Projekt in Frage kommen. Als Modul für HTTP-Server kann es sogar die Funktionalität von PHP übernehmen, was die programmgestützte Generierung von HTML-Seiten betrifft.

Im Vergleich der vier zur Auswahl stehenden Sprachen liegt Python vor C++ und Java an erster Stelle und wird als Implementierungssprache Verwendung finden.

2.3.3. Datenformat

Wird ein verteiltes System entwickelt, so stellt sich die Frage, in welcher Form die Daten zwischen den verschiedenen Prozessen ausgetauscht werden sollen.

Bei der Anwendung handelt es sich in der praktischen Umsetzung um ein Bibliothekssystem. Daher sind Stammdaten über Bücher, Zeitschriften und ähnliche materielle Objekte zu übertragen. Immaterielle Objekte wie Rechte, Informationsverweise oder URLs können übertragen werden müssen. Auch Nutzerinformationen und Datensätze von Zusatzdiensten müssen vom System verwaltet werden. Jedoch ist das System nicht auf die Anwendung Bibliothekssystem beschränkt.

Bis zu dieser Stelle wurde bereits die Entscheidung für XML-RPC als Transportmechanismus getroffen. Nun steht noch die Frage zu beantworten, in welcher Form die Daten codiert werden, die die XML-RPC-Schnittstelle übertragen wird. Zur Auswahl stehen drei Alternativen:

1. binäre Speicherabbilder der Laufzeitumgebung,
2. Zeichenketten, die XML-Objekte einer anwendungsspezifischen DTD enthalten,

¹⁰Wenngleich der Artikel eine Quelle nennt, die aussagt, dass die Anzahl der Zeilen pro Zeiteinheit unabhängig von der Sprache ist.

¹¹Im Projekt soll dies noch dadurch unterstützt werden, dass möglichst kompakte Programme jeweils nur einen einzigen Dienst erbringen.

2. Lösungsansätze

3. standardisierte XML-RPC-Datenstrukturen auf ASCII-Basis.

Die erste Methode verwendet Serialisierungsfunktionalität der Laufzeitumgebung. Das kann beispielsweise durch das `pickle`-Modul von Python, die `pack/unpack`-Funktionen von PHP oder auch durch die Objektserialisierung von Java realisiert werden. Die so generierten Binärdaten werden dann per XML-RPC über das Netz übertragen. Der Nachteil dieses Ansatzes besteht darin, dass eine Kommunikation mit Programmen, die in anderen Programmiersprachen geschrieben wurden, gar nicht oder nur nach Entwicklung einer Anpassungsschicht möglich ist.

Anstatt die Datenobjekte binär zu codieren, kann man die Aufrufparameter nach XML codieren. Dazu kann eine anwendungsspezifische DTD verwendet werden. Im Rahmen dieses Projektes böte sich ein *RDF*¹²-*Vocabulary* [29] für *Dublin Core* [30] (DC)¹³ an (Abbildung 11). Auch wenn diese Idee auf den ersten Blick verlockend scheint, so ergeben sich daraus Nachteile. Um die Datenobjekte entsprechend zu codieren, müsste tief in die XML-RPC-Bibliothek eingegriffen werden. Bisher verwendet die Bibliothek Methoden der *Introspection*, um die Daten automatisch zu codieren. Die *Introspection* arbeitet auf Syntaxebene und nicht semantisch. Eine RDF ist jedoch eine semantische Beschreibung der Daten. Alternativ kann man externe Konvertiererroutinen entwickeln, deren Ergebnisse als Zeichenkette noch einmal in der XML-RPC-Bibliothek in XML verpackt werden. Eine Weiterverarbeitung hätte dann ebenfalls eine Decodierung in einem zusätzlichen Modul zur Folge. Nun könnten mit dieser Methode zwar die Datenobjekte codiert werden, aber weitere Attribute der Kommunikationsobjekte würden auf die herkömmliche XML-RPC-Methode codiert werden müssen.

Die dritte Alternative besteht darin, die Codierung der Kommunikationsobjekte komplett der RPC-Bibliothek zu überlassen (Abbildung 12) und damit eine hohe Kommunikationstransparenz erreichen. Durch die Verwendung von standardisierten Codierungen ist es auch möglich, das System XML-RPC-Clienten zu öffnen, ohne dass diese eine doppelte XML-(De)Codierung ausführen müssten.

Eine Verwendung von RDF im Inneren des Systems ist mit XML-RPC weder unter dem Performanz- noch dem Transparenzaspekt sinnvoll zu verbinden. Allerdings könnte RDF für Teile des Persistenzspeichers und für die Kommunikation mit Diensten von fremden Anbietern/Nutzern verwendet werden. Für

¹²Ressource Definition Format

¹³Auch wenn DC ausreichende Möglichkeiten böte, die unmittelbaren Ressourcenattribute zu beschreiben, so ist die Speicherung weitergehender Attribute wie Regalstandort oder Exemplareigenschaften derzeit vollkommen ungeklärt.

```

<?xml:namespace href="http://www.w3c.org/RDF/" as="RDF"?>
<?xml:namespace href="http://purl.org/RDF/DC/" as="DC"?>
<?xml:namespace href="http://.../bib" as="bibo"?>
<RDF:RDF>
<DC:Title>Industriegeographie</DC:Title>
<DC:Creator>Wolfgang Brücher</DC:Creator>
<DC:Publisher>Westermann</DC:Publisher>
<DC:Type>Book</DC:Type>
<DC:Language>de</DC:Language>
<DC>Date>1982</DC>Date>
<bibo:Signatur>S1: 383438</bibo:Signatur>
<bibo:Standort>RB 10714 bru</bibo:Standort>
</RDF:Description>
</RDF:RDF>

```

Abbildung 11: Codierung mit RDF

diesen Zweck wären entsprechende *Externalizer* zu schreiben.

2.4. Benchmarking

Die in Abschnitt 2.2 geäußerten Performanzaussagen der RPC-Mechanismen sollen nun qualitativ unterlegt werden. Dazu standen die fünf in Tabelle 1 gezeigten Hardwarekonfigurationen zur Verfügung.

Typ	CPU	RAM	Ethernet	Rechner
1	Athlon 500	256 MB	100 MBit	malbec, merlot
2	Pentium III 800	512 MB	10 MBit	dem, tan
3	Pentium III 800	512 MB	100 MBit	Clusterknoten
4	Athlon 800	256 MB	100 MBit	bohne, gurke
5	Athlon 950	512 MB	100 MBit	goethe, schiller

Tabelle 1: Knotentypen

Es wurden jeweils fünf Tests für XML-RPC, Sun-RPC und CORBA durchgeführt. Als Python Laufzeitumgebung wurde das Pythonpaket von *RedHat Linux* in der Version 1.5.2 verwendet. Um Auswirkungen von verschiedenen Programmiersprachen zu verringern, wurde für alle Tests Python verwendet.¹⁴ Als XML-RPC wurde klassischer XML-RPC mit Pythonwares Implementation der

¹⁴Bis auf Teile der CORBA-Bibliothek und dem XML-Parser sind alle Bibliotheken und Tests ausschließlich in Python implementiert.

2. Lösungsansätze

```
<object class="BookData.BookData">
<member>
<name>standort</name> <value><string>RB 10714
bru</string></value>
</member>
<member>
<name>date</name> <value><int>1982</int></value>
</member>
<member>
<name>title</name> <value><string>Industriegeographie</string>
</value>
</member>
<member>
<name>language</name> <value><string>de</string></value>
</member>
<member>
<name>author</name> <value><string>Wolfgang Brü-
cher</string></value>
</member>
<member>
<name>publisher</name> <value><string>Westermann</string></value>
</member>
<member>
<name>signatur</name> <value><string>S1:
383438</string></value>
</member>
</object>
```

Abbildung 12: Codierung mit XML-RPC

Version 0.9.8 [34] verwendet, die Sun-RPC-Implementierung stammt aus einer Demo-Anwendung aus dem Python-Quellcodepaket der Version 2.1 und CORBA verwendet das Paket Fnorb 1.1 [5]. Zusätzlich wurden die ersten beiden Tests mit BSD-Sockets implementiert, die zum Standardumfang von Python gehören.

Jeder Test wird im Clienten-Programm 5000-mal in Folge aufgerufen. Dabei werden Resolve-Operationen, die beispielsweise zum Aufsuchen des Servers dienen, vor dem Beginn der Zeitnahme ausgeführt. Jeder der fünf Aufruftypen arbeitet grundsätzlich so, dass eine Datenstruktur zum Server geschickt und von diesem an den Clienten zurück übertragen wird.

Test 1 — Ping-Pong: Es wird eine minimale Methode aufgerufen, die maxi-

mal ein Byte zurückgibt.¹⁵

Test 2 — Hello World: Die Zeichenkette „Hello World“ wird zwischen Client und Server hin und her geschickt.

Test 3 — Struktur: Eine Struktur bzw. ein Objekt mit den Komponenten bzw. Attributen $a=47$, $b=23.5$ und $c=$ „Hello World“ wird übertragen.

Test 4 — Liste von Integer: Eine Liste von 100 ganzen Zahlen wird übertragen.

Test 5 — Liste von Strukturen: Eine Liste von 100 der in Test 3 verwendeten Strukturen wird übertragen.

Außerdem wurde jeder Test einmal so ausgeführt, dass sich Client und Server auf derselben Maschine befanden und zum zweiten bei einer räumlichen Trennung von Client und Server. Um zufällige Verfälschungen der Messwerte aufgrund von unvorhergesehener Aktivität von anderen Prozessen zu verringern, wurde jeder Testlauf dreimal ausgeführt und von diesen drei Werten der Mittelwert bestimmt.

Die Messwerte werden im Folgenden in Diagrammen dargestellt. Im Anhang finden sich dazu entsprechende Tabellen und Programmauszüge.

Die Messergebnisse lassen sich mit einem Satz zusammenfassen: XML-RPC ist sehr langsam. Diese Aussage ist allerdings rein qualitativ im Verhältnis zu den anderen RPC-Mechanismen. Eine quantitative Beurteilung ist also angeraten.

Ganz bewusst wurden keine Tests verwendet, die künstliche Datenpakete in schrittweise erhöhtem Datenumfang austesten, sondern Datenstrukturen, die auch im eigentlichen Projekt auftreten werden. Auch wird nicht explizit die Leistung von Teilsystemen untersucht, sondern ein *Blackboxtest* durchgeführt, weil im Rahmen dieser Arbeit nicht zu tief in die Bibliotheken eingegriffen werden kann.

2.4.1. BSD-Sockets

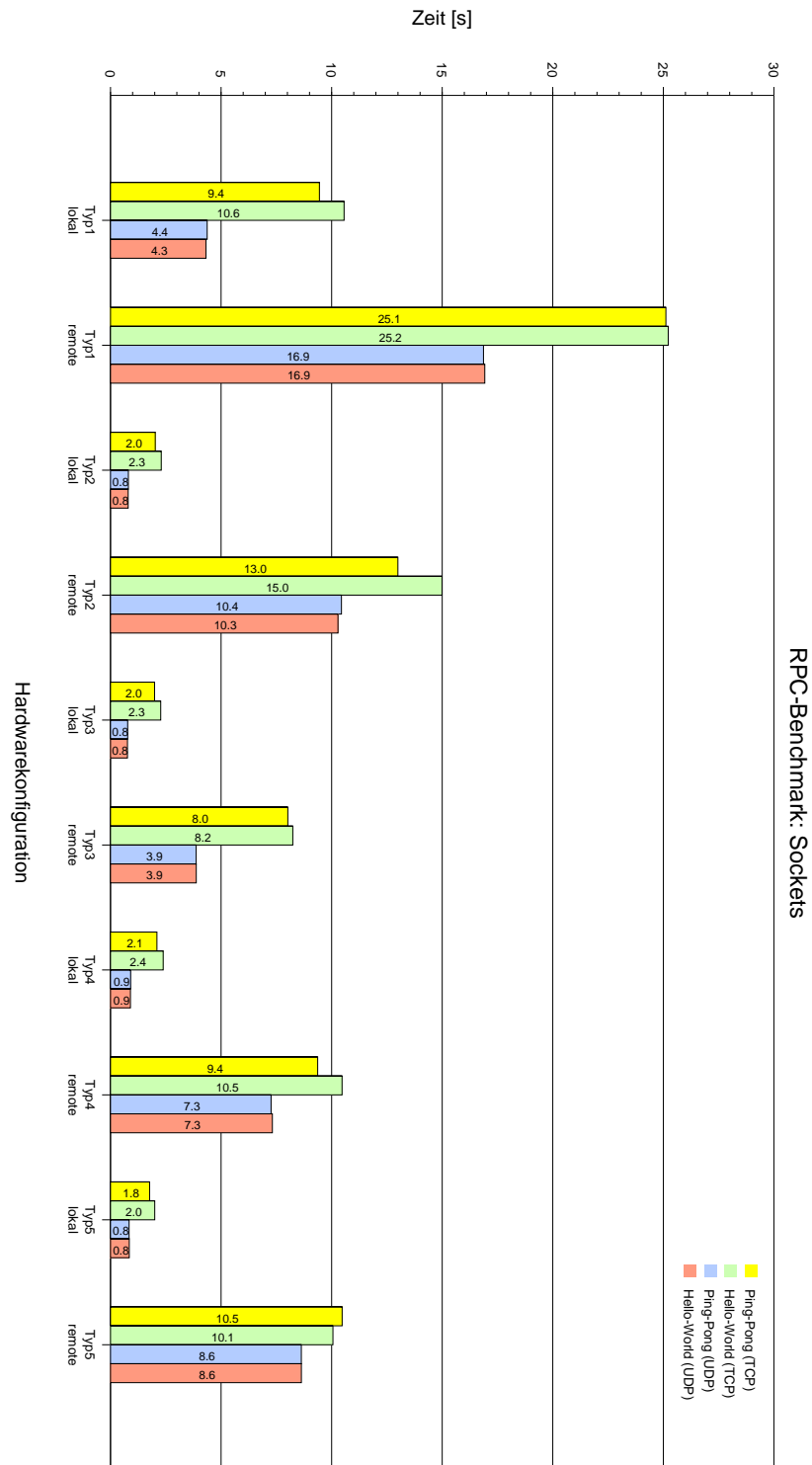
Um zu untersuchen, welche Einfüsse allein aus der Verwendung von BSD-Sockets entstehen, ist in Abbildung 13 ein Geschwindigkeitsvergleich der Tests 1 und 2 auf den verschiedenen Rechnertypen zu sehen.

Die Kommunikation zwischen zwei Rechnern ist langsamer, als wenn Client und Server auf einem Rechner ausgeführt werden. Auffällig ist die Tatsache, dass die Länge der Nachricht kaum einen Einfluss auf die Kommunikationsdauer hat.

¹⁵XML-RPC erlaubt keine Methoden ohne Rückgabewerte.

2. Lösungsansätze

Abbildung 13: RPC-Benchmark: Sockets



Das ist angesichts des recht kleinen Datenvolumens von nur wenigen Bytes auch nicht anders zu erwarten, da sowohl bei Test 1 als auch Test 2 ein einziges IP-Paket für die Nutzdaten ausreichend ist.

Man erkennt auch, dass mit steigender CPU-Leistung die Kommunikationszeit sinkt. Ein Einfluss des Mediums ist nicht so hoch, wie man hoffen könnte: Der Übergang von 10 auf 100 MBit bringt selbst bei UDP keine Verzehnfachung der Kommunikationsleistung.

Erwartungsgemäß werden die Daten per UDP ein wenig schneller übertragen, weil die Protokollschritte zum Aufbau und zur Beendigung der virtuellen Verbindung fehlen. Bemerkenswert ist allerdings, dass die Geschwindigkeit beim Remote-Test im Vergleich zu TCP nicht verdreifacht wird, obwohl mindestens zwei Pakete wegfallen (SYN, FIN).

2.4.2. Sun-RPC

Die vorliegende Implementierung von Sun-RPC codiert die Daten in einer Python-Bibliothek nach XDR. Die Werte aus Abbildung 14 zeigen, dass mit steigender Komplexität der Testdaten auch die Testzeiten steigen. Einzelne zusätzliche Testmessungen haben ergeben, dass von diesen Zeiten allein die Hälfte für das Codieren der Daten benötigt wird. Dabei werden allerdings sehr kompakte Datenblöcke erzeugt. Die Liste von Strukturen ist beispielsweise rund 2800 Byte groß.

Genauere Untersuchungen über die Geschwindigkeit und eine kurze Beschreibung von Sun-RPC finden sich in [33].

An dieser Stelle sei auf einen weiteren Nachteil der Python-Implementierung von Sun-RPC hingewiesen: Da kein RPCGEN für Python existiert, müssen die Codierungs- und Decodierungsfunktionen manuell erstellt werden. Damit steigt natürlich der Programmieraufwand und auch die Gefahr, dass es zu Programmfehlern aufgrund unsauberer Datendecodierung kommt.

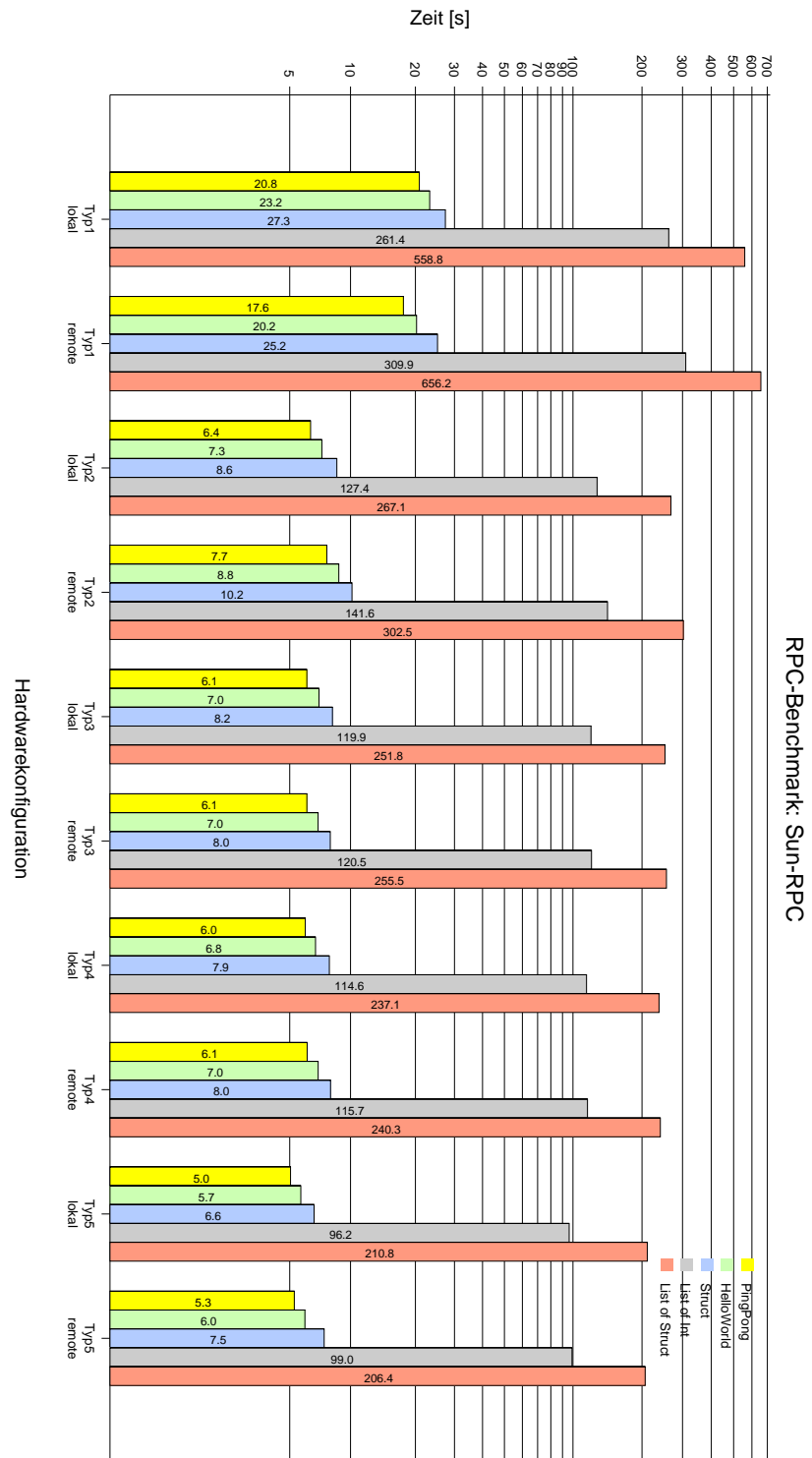
2.4.3. CORBA

Im Gegensatz zum XDR-Paket werden in Fnorb die Daten mit einer in C geschriebenen Erweiterung in ein plattformunabhängiges Format umgewandelt. Das scheint ein möglicher Grund dafür zu sein, dass die Zeiten bei den drei kleineren Datentypen auf einer Plattform nahezu gleich sind — Konvertierungsarbeit ist bedeutungslos und die Datenübertragung erfolgt in einem Paket.

Es fällt auch auf, dass eine unterschiedliche Netzanbindung (Unterschied zwischen Typ2 und Typ3) kaum Einfluss auf die Übertragungsgeschwindigkeit hat. Das lässt den Schluss zu, dass der Hauptaufwand durch die CPU erbracht

2. Lösungsansätze

Abbildung 14: RPC-Benchmark: Sun-RPC



2.4. Benchmarking

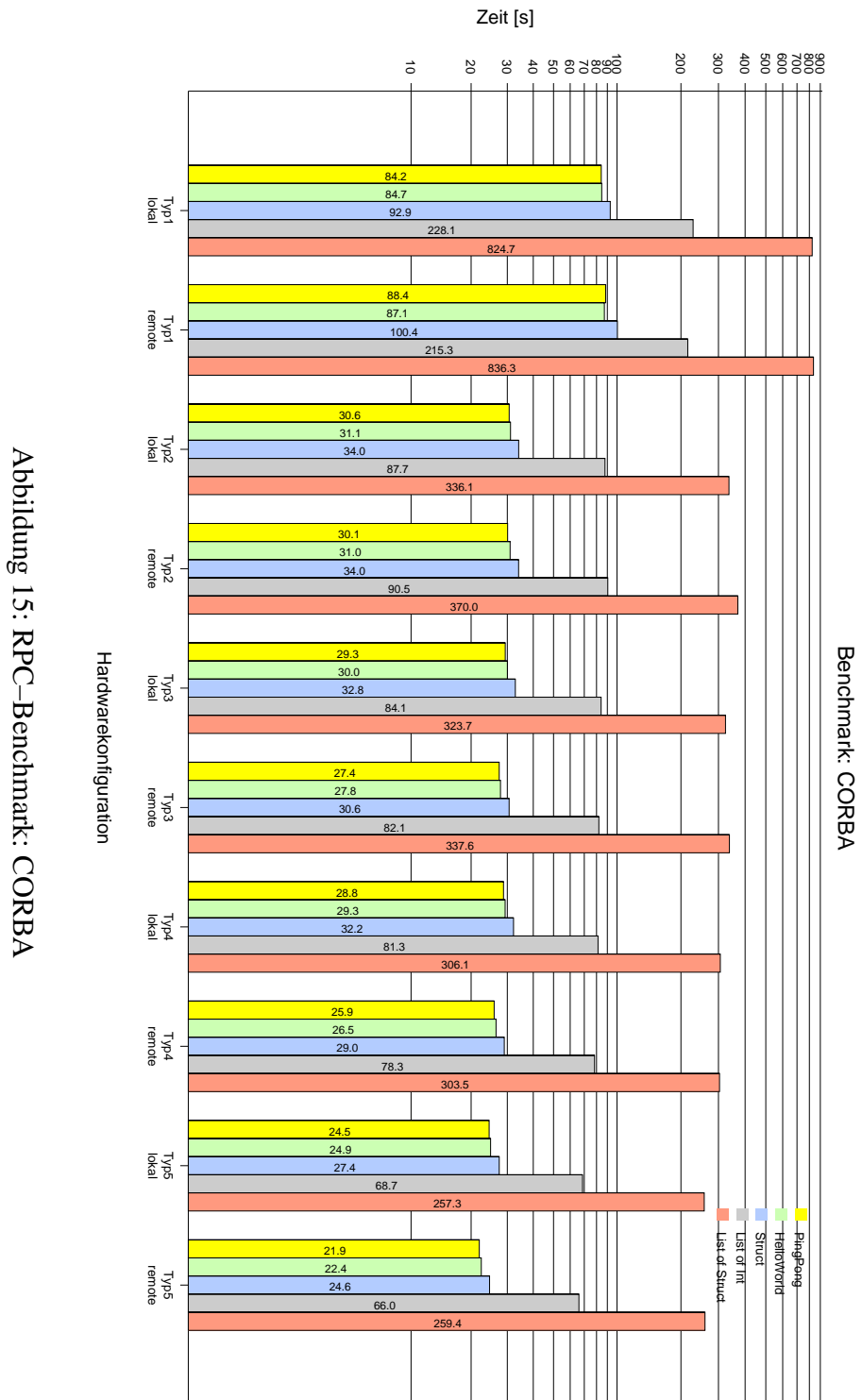


Abbildung 15: RPC-Benchmark: CORBA

2. Lösungsansätze

werden muss. Untermauert wird die Behauptung dadurch, dass der Zugewinn von ca. 20% mehr Rechenleistung zwischen Typ4 und Typ5 ebenfalls rund 20% niedrigere Testwerte ergibt.

2.4.4. XML-RPC

Die hier verwendete XML-RPC-Bibliothek verwendet einen in C geschriebenen XML-Parser namens sgmlop [35]. Dieses Modul ist eigenen Messungen zufolge etwa bis zum Faktor 6 schneller als der in Python geschriebene XML-Parser und ca. 10 bis 30% schneller als expat [23]¹⁶. Tabelle 2 zeigt die benötigten Zeiten, die auf einem Rechner vom Typ 2 für die Codierung und Decodierung von jeweils 5000 der in den Tests zu übertragenden Datenpakete benötigt wird.

Parser	Test 1	Test 2	Test 3	Test 4	Test 5
nur codieren	0.45	0.46	1.25	12.27	94.30
klassisch	9.50	9.50	30.76	337.28	2489.80
expat	2.44	2.43	7.75	71.68	584.70
sgmlop	2.25	2.50	6.85	54.45	466.30

Tabelle 2: Codierungszeiten
[in Sekunden]

Bei XML-RPC lässt sich eine direkte Abhängigkeit zwischen Prozessorleistung und Kommunikationsgeschwindigkeit feststellen. Hier spielt also eindeutig der Codierungsaufwand eine erhebliche Rolle. Einzelmessungen ergaben, dass bei den komplexen Datenstrukturen teilweise über 50% der Rechenzeit für Codierung und Decodierung aufgewendet werden müssen. Ein noch schnellerer Parser ist an dieser Stelle wünschenswert.

Reduziert man die Messwerte um die Codierzeiten, so ist XML-RPC noch immer sehr langsam. Ein Grund dafür könnte das erheblich höhere Datenvolumen sein, welches durch eine Codierung entsteht, deren Umsetzung bis zum Faktor 14 mehr Verwaltungsdaten als Nutzdaten erzeugt. Abbildung 17 zeigt das Ergebnis der Codierung einer einzigen Zahl. Eine Komprimierung während des Transportes könnte an dieser Stelle erhebliche Geschwindigkeitsgewinne bringen. Eine andere Alternative wäre eine kompaktere Darstellung der Daten mit Hilfe einer geänderten DTD. Allerdings würde damit die Kompatibilität zu anderen XML-RPC-Implementierungen verloren gehen. Als Anmerkung sei erwähnt, dass die alternative SOAP-Lib etwa ähnlich umfangreiche Codierungen

¹⁶Von den Messwerten sind die Codierzeiten abzuziehen, wenn die reine Dekodierleistung verglichen werden soll.

2.4. Benchmarking

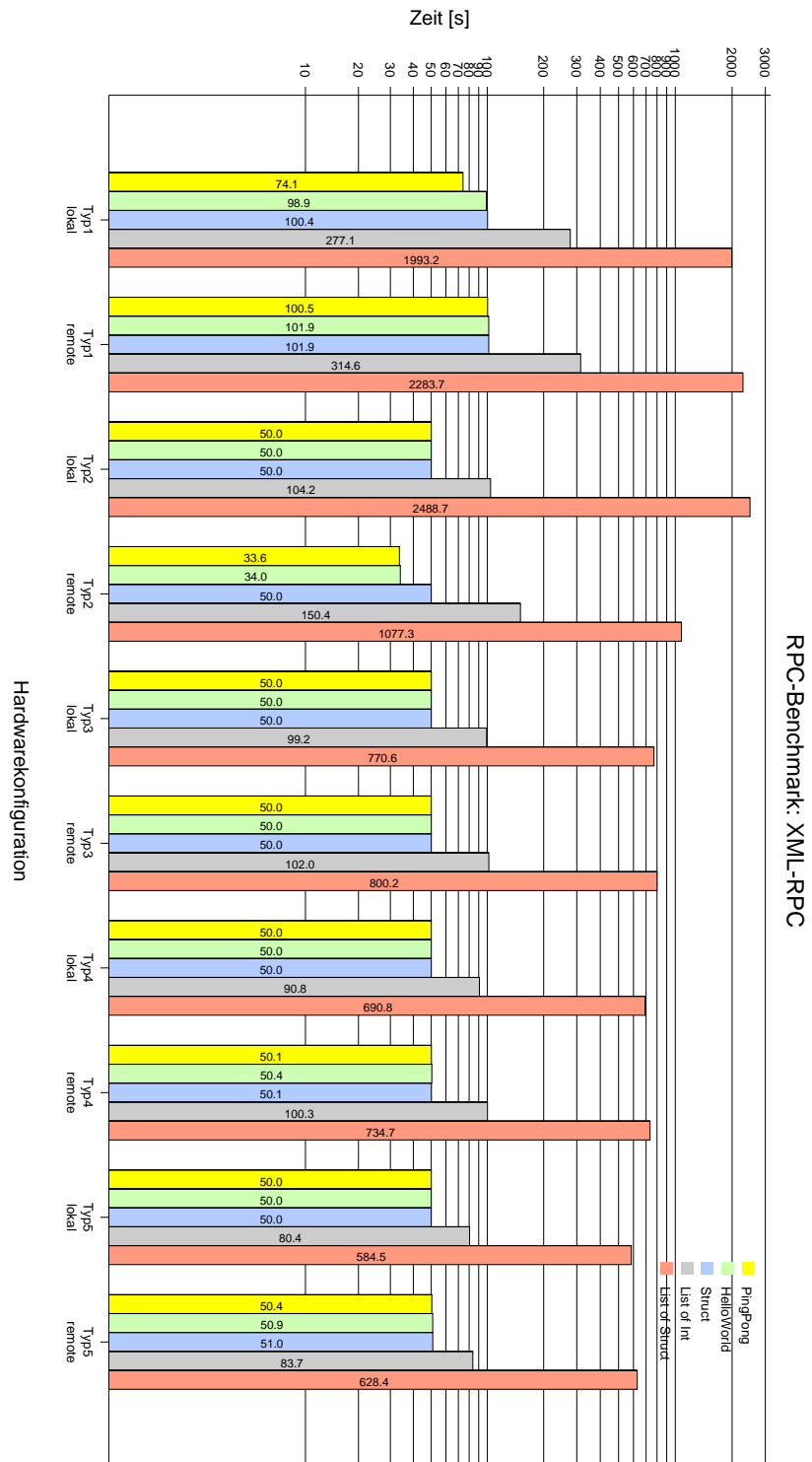


Abbildung 16: RPC-Benchmark: XML-RPC

2. Lösungsansätze

```
<params>
<param>
<value><int>1</int></value>
</param>
</params>
```

Abbildung 17: XML-RPC-Codierung

erzeugt.

Ebenfalls auffällig ist die Tatsache, dass die drei ersten Tests nahezu gleiche Laufzeiten besitzen. Es ist zu vermuten, dass dies an der gleichmäßig langsamen Arbeitsweise der XML-RPC-Implementierung liegt. Detailuntersuchungen im Inneren der XML-RPC-Bibliothek könnten den Anteil von datenunabhängigen Verarbeitungsaufwand ermitteln.

Ein Phänomen stellen die beiden erste Messwert für „Typ2 remote“ dar: Die Kommunikation auf einem Rechner verläuft langsamer, als wenn Client und Server auf zwei verschiedenen Rechnern gestartet werden und über das Netzwerk kommuniziert werden muss. Auch nach mehrmaligen Messungen konnten diese Werte bestätigt werden. Eine Ursache dafür war nicht zu finden.

2.4.5. Vergleich zwischen RPC-Mechanismen

Abschließend sollen noch die verschiedenen RPC-Mechanismen miteinander verglichen werden.

In [Abbildung 18](#) sieht man, dass XML-RPC bis zum Faktor sechs langsamer ist als die direkte Kommunikation über Sockets oder per Sun-RPC, wenn einfache Datenstrukturen über Rechengrenzen hinaus übertragen werden sollen. Im Vergleich zu CORBA ist XML-RPC allerdings nur um den Faktor zwei langsamer. Das ist zu vertreten, wenn man bedenkt, dass in der Anwendung ohnehin die lokale Verarbeitung einen relativ großen Anteil an der Verarbeitungszeit haben wird.

[Abbildung 19](#) zeigt, dass der Geschwindigkeitsnachteil von XML-RPC gegenüber Sun-RPC bei komplexeren Datenstrukturen abnimmt und auch im Vergleich zu CORBA nicht viel schlechter wird.

Im Anhang sind noch drei weitere Grafiken zu finden, die Geschwindigkeitsvergleiche bei den Tests eins, drei und vier darstellen.

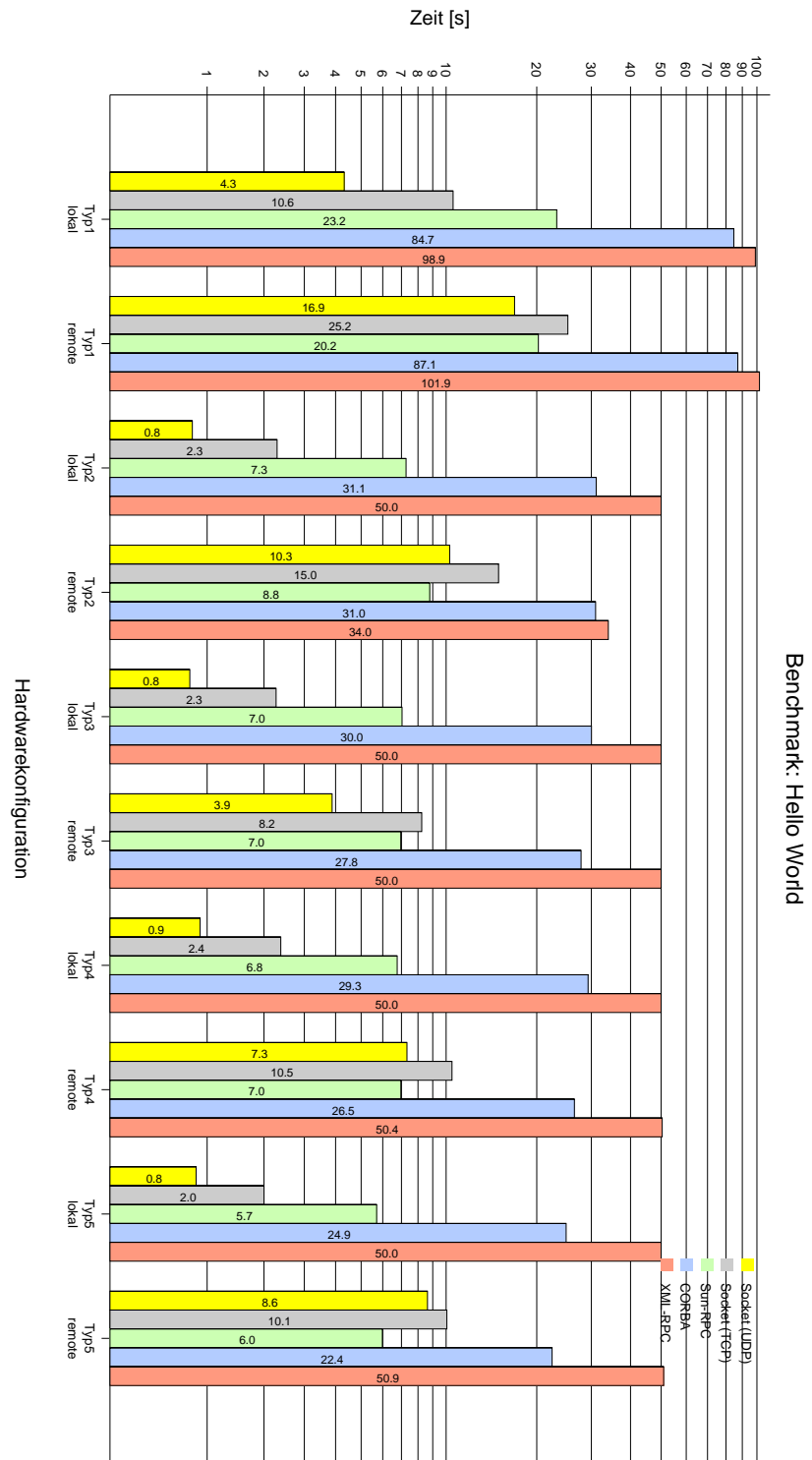
Zusammenfassend kann man festhalten, dass XML-RPC zwar erheblich langsamer ist als die anderen getesteten RPC-Mechanismen, aber dieser Nachteil nicht so stark zu bewerten ist, um in diesem Projekt nicht davon Gebrauch zu machen. Die Vorteile liegen eindeutig bei der sehr einfachen praktischen Pro-

2.4. *Benchmarking*

grammierung, der hohen Portabilität und der relativ unkomplizierten Struktur im Inneren des XML-RPC-Systems.

2. Lösungsansätze

Abbildung 18: RPC-Benchmark: Hello World



2.4. Benchmarking

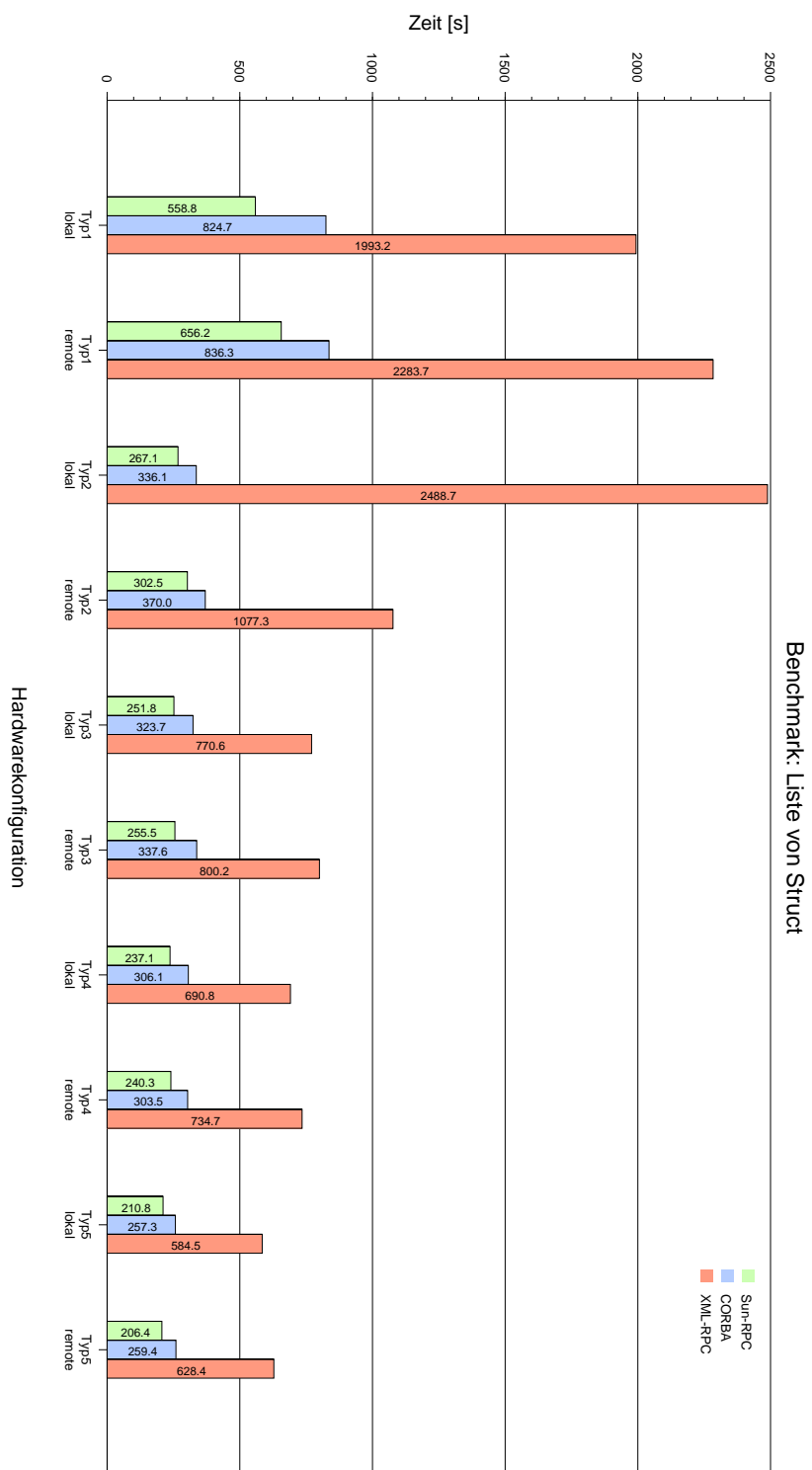


Abbildung 19: RPC-Benchmark: Liste von Strukturen

2. Lösungsansätze

3. Realisierung

Nachdem im letzten Kapitel die Grundlagen geschaffen wurden, soll nun das konkrete Systemdesign beschrieben werden. Es wird auffallen, dass einige der Architekturmerkmale an das *Andrew File System* (AFS) [6] angelehnt sind. Dies ist bewusst so gewählt worden, denn AFS realisiert ein sehr zuverlässiges und skalierbares verteiltes Dateisystem, welches in Aspekten der Replikation und Identifikation dem hier zu entwickelnden System nicht unähnlich ist. Bei der Identifizierung von Diensten werden Bezeichnungen und Funktionalitäten aus dem CORBA-Umfeld eingebracht, wobei allerdings keine Neuimplementierung von CORBA beabsichtigt wird.

Alle Dienste sollen mit einem möglichst kleinen Interface ausgestattet werden, welches gerade ausreichend ist, um die tatsächlich gewünschten Aufgaben zu erfüllen. Es werden keine hypothetischen Anwendungsfälle eingeplant, um den Programmumfang nicht grundlos zu vergrößern. Dieser Ansatz — er entstammt einer Entwicklungsphilosophie, die *eXtreme Programming* [7] genannt wird — kann gewählt werden, da aufgrund der Art und Weise, wie XML-RPC und Python zusammenarbeiten, weitere Funktionen jederzeit hinzugefügt werden können, ohne den Rest des Systems stark zu beeinflussen.

Für jede Aufgabe wird ein neuer Server angelegt, um den Quell-Code der Server überschaubar zu halten. Dadurch entstehen möglicherweise einige Probleme, die mit einem monolithischen System nicht entstanden wären. Diese Probleme sind beispielsweise Synchronisationsprobleme, zeitweise Inkonsistenzen der Daten, eine gewisse Unsicherheit in der Beurteilung der Stabilität des Systems oder häufigerer Datentransport zwischen einzelnen Diensten. Diese Probleme werden in Kauf genommen, um ein flexibleres System zu erschaffen.

Eine generische Basisklasse (XMLMIXIN)¹⁷ definiert ein minimales Interface für alle Server:

ident(): Gibt eine identifizierende Datenstruktur zurück. Im einfachsten Falle enthält diese nur den Namen einer Dienstklasse, den Namen des dienstbringenden Rechners und eine Portnummer, es sind aber auch weitere Attribute möglich, (z.B. Name und Teilbereich einer Datenbank, der vom Dienst verwaltet wird).

ping(): Liefert eine 1 zurück und zeigt damit an, dass der Dienst prinzipiell in der Lage ist, auf Anfragen zu reagieren.

¹⁷Um genau zu sein, wird dieses Funktionsinterface von einer Handlerklasse (GENERICHANDLER) definiert, die dann einige Funktionen an die Serverklasse zurück delegiert.

3. Realisierung

stop(): Stoppt den Dienst, indem keine weiteren Aufträge mehr angenommen werden. Laufende Aufträge werden bei *multithreaded* Servern noch abgeschlossen.

Von dieser Klasse gibt es vier Versionen, die sich über die Auswahl von SSL-Fähigkeit und *Multithreading* definieren: `GENERICSERVER`, `GENERICTHREADINGSERVER`, `GENERICSSLSERVER` und `GENERICTHREADINGSSLSERVER`.

Weiterhin wird für jeden Dienst ein eigener Prozess gestartet. Ziel ist, dass ein mögliches Versagen einzelner Prozesse das Gesamtsystem nur wenig stört. Aufgrund der relativ geringen Ressourcenanforderungen der Python-Laufzeitumgebung ist dies zu vertreten. Ein positiver Nebeneffekt ist, dass sich einzelne Prozesse auch auf anderen Rechnern starten lassen, sobald die Ressourcen auf einem Rechner nicht länger ausreichen.

Es lassen sich mehrere Gruppen von Diensten erkennen, die getrennt betrachtet werden sollen.

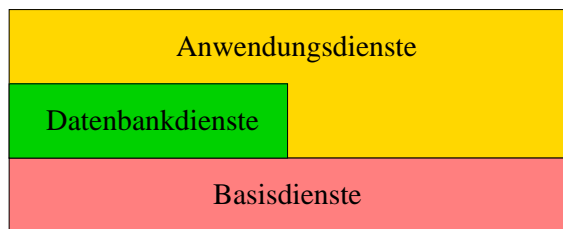


Abbildung 20: Systemaufbau: Dienstgruppen

Basisdienste stellen grundlegende Funktionen als Infrastruktur zur Verfügung. Dazu gehören beispielsweise Funktionalitäten, um Dienste zu starten, den Zustand von Diensten zu überwachen oder Dienste im verteilten System aufzufinden.

Datenbankdienste werden verwendet, um Datenobjekte aufzufinden, zu modifizieren oder persistent zu speichern.

Anwendungsdienste realisieren Algorithmen zur Behandlung von Anwendungsfällen und stellen den Zugangspunkt für Nutzer zum System dar.

3.1. Basisdienste

Die Basisdienste bilden das Fundament des Systems. Man erkennt sie insbesondere daran, dass sie wesentliche Infrastruktur zur Verfügung stellen und auf jedem Knoten im System eine Instanz von ihnen zu finden ist.

3.1.1. Dienstkontrolle (BOSSERVER)

Vorüberlegungen Der erste Dienst, der auf jedem Knoten gestartet wird, ist — in Anlehnung an AFS — der BOSSERVER. Aufgabe des BOSSERVERS (BOS = *Basic Overseer Service*) besteht darin, andere Prozesse zu starten und deren Existenz zu sichern. Ein Dienst wird dabei durch eine Kommandozeile, die den Prozess starten kann, und einem Namen gekennzeichnet. Der mit der Kommandozeile gestartete Prozess darf die Kontrolle erst dann an den rufenden Prozess zurückgeben, wenn der Prozess beendet wurde.

Dazu bietet der BOSSERVER zwei Startphilosophien an:

erhaltend: Wird der Prozess beendet oder versagt er, so wird er vom BOSSERVER erneut gestartet.

einmalig: Geht der Prozess zuende, so wird er nicht wieder gestartet.

Der erhaltende Dienststart wird den Regelfall darstellen. Besonders andere Basisdienste werden mit dieser Option gestartet, um im Falle eines Versagens einen möglichst raschen Ersatz zu gewährleisten.

Der einmalige Dienststart wird verwendet, wenn die Semantik des zu startenden Dienstes dies verlangt oder der Dienstneustart von weiterführenden Einflüssen bestimmt wird¹⁸.

Eine weitere Option für den Start von Diensten besteht darin, festzulegen, ob ein Dienst bei Neustart des BOSSERVERS ebenfalls erneut gestartet werden soll oder nicht. Damit lassen sich Dienste permanent an Rechner binden oder Basisdienste festlegen, die immer mit dem BOSSERVER gekoppelt laufen sollen.

Sofern die Dienste beim Start des BOSSERVERS gestartet werden, kann auch festgelegt werden, dass diese Dienste auf jedem Knoten gestartet werden sollen.

Die drei Startoptionen *erhaltend/persistent* (p), bei Neustart/*withbos* (w) und auf jedem Knoten/*anywhere* (a) lassen damit acht in Tabelle 3 gezeigten Szenarien zu.

Dienstobjekt-Interface Der BOSSERVER stellt folgendes Interface zur Verfügung¹⁹:

bos_register(name, cmdline, persistent=0, withbos=0, anywhere=0):
 Registriert einen Dienst mit dem angegebenen Namen und der zugehörigen Kommandozeile. Der Dienst wird noch nicht gestartet.

¹⁸So wäre es denkbar, den Dienst gleich auf einem anderen Rechner neu zu starten, weil die Ressourcen auf dem bisherigen Rechner nicht länger ausreichen.

¹⁹Der Name des Dienstes wird im Namen der Methode immer mit angegeben, um zufällige Verwechslungen mit gleichlautenden Methodenbezeichnern anderer Dienste zu vermeiden.

3. Realisierung

p	w	a	Erläuterung	Anwendung
o	o	o	Der Dienst wird auf diesem Knoten ausgeführt, bis er beendet wurde und muss nach einem Neustart des BOSSERVERS nicht wieder zur Verfügung stehen.	Migrierende Dienste in dynamischer Lastverteilung und Redundanzsicherung.
x	o	o	Der Dienst wird nach einem Beenden erneut gestartet, steht nach einem Neustart des BOSSERVERS allerdings nicht mehr zur Verfügung.	Migrierende Dienste, deren Existenz einzig auf diesen Knoten beschränkt ist.
o	x	o	Der Dienst wird mit dem Start des BOSSERVERS einmalig gestartet.	Initiale Knotenüberprüfung, wie beispielsweise ein Abgleich von Replikaten.
x	x	o	Der Dienst wird nach jedem Neustart des BOSSERVERS geladen und nach Beendigung erneut gestartet.	Ortsfeste Dienste, deren Existenz zugesichert werden soll.
o	o	x	<i>keine Bedeutung</i>	
x	o	x	<i>keine Bedeutung</i>	
o	x	x	Der Dienst wird auf jedem BOSSERVER einmalig gestartet.	Abgleich von Dateien, wenn kein verteiltes Dateisystem verwendet werden würde.
x	x	x	Der Dienst wird auf jedem BOSSERVER gestartet und wird nach Beendigung neu gestartet.	Namensdienst, der Informationen über die Dienste im System kennt.

Tabelle 3: Dienststartoptionen

persistent: Der Dienst wird nach Beendigung erneut gestartet.

withbos: Dienst wird bei Neustart des BOSSERVERS auf diesem Knoten neu gestartet. Für jeden Knoten, auf dem ein BOSSERVER läuft, existiert eine entsprechende Konfigurationsdatei²⁰.

anywhere: Die Einstellungen von `withbos` gelten für alle Knoten und nicht nur für den aktuellen. Diese Einstellungen werden in einer zweiten Konfigurationsdatei abgelegt, die allen Knoten beim Start verfügbar gemacht werden muss²¹.

bos_deregister(name): Entfernt den angegebenen Dienst aus der Liste der Dienste.²²

bos_start(name, extraArgs=None): Startet den Dienst mit dem angegebenen Namen. Mit `extraArgs` werden zusätzliche Parameter über die Kommandozeile übergeben.

bos_suspend(name): Sorgt dafür, dass Dienste, die als persistent gekennzeichnet sind, nach einem Abbruch nicht neu gestartet werden.

bos_status(): Listet alle Dienste und deren Status auf.

Implementierung Im Programm wird diese Überwachung der gestarteten Dienstprogramme dadurch realisiert, dass jeder zu startende Dienst von einem *Thread* des BOSSERVERS (Klasse BOSWATCHER) gestartet wird, der so lange blockiert wird, bis der Dienstprozess beendet ist.

Ein kleines Problem tritt auf, sobald eine Authentifizierung der Aufrufe verlangt wird. Da der Authentifizierungsdienst noch nicht gestartet wurde, kann auf dessen Funktionalität auch noch nicht zurückgegriffen werden. Da jedoch SSL verwendet wird, kann anstelle der Authentifizierung innerhalb der Anwendung eine Authentifizierung auf Basis von SSL-Zertifikaten erfolgen.

Alle Dienste, die SSL verwenden sollen, kennen zertifizierte Client-Zertifikate. Die Zertifikate werden dazu von einer *Certification Authority* (CA) zertifiziert. Nur Zertifikate, die von dieser CA zertifiziert wurden, bekommen Zugriff auf den Dienst. Alle anderen Zertifikate werden abgewiesen. Später wäre auch eine Erweiterung dahingehend denkbar, dass eine Liste von gültigen

²⁰Damit wird das System statisch vorkonfiguriert. Weitere Dienste können während der Laufzeit dynamisch gestartet werden.

²¹Entweder durch explizites Kopieren bei unabhängigen Dateisystemen oder mit Hilfe eines verteilten Dateisystems.

²²Der Dienst wird nur aus der Liste entfernt, aber nicht gestoppt. Zum Stoppen muss der Dienst direkt kontaktiert werden.

3. Realisierung

Client-Zertifikaten verwaltet wird. Damit kann auch das Problem von unsicher gewordenen — z.B. gestohlenen — Zertifikaten umgangen werden.

3.1.2. Dienstauffindung (LOCATIONSERVER)

Vorüberlegungen Ein Problem ergibt sich, wenn ein Dienst einen anderen Dienst in Anspruch nehmen möchte und ermittelt werden soll, wo dieser Dienst erbracht wird. Da XML-RPC auf TCP-Sockets aufsetzt, dienen IP und Portnummer als eindeutige Identifikation des Dienstannahmepunktes. Man könnte nun die Portnummern bestimmten Diensten zuteilen und damit für die Anwendung eine Liste von *well-known Ports* definieren. Dieses Vorgehen hat jedoch Defizite:

- der vorgesehene Port kann beispielsweise von einer anderen Anwendung bereits reserviert sein
- es werden mehrere Instanzen des Dienstes gewünscht, die natürlich nicht alle einen einzigen Port nutzen können
- in bestimmten Umgebungen werden Ports durch Firewalls blockiert, so dass eventuell auf andere Ports oder gar Anwendungs-Proxies ausgewichen werden muss
- bei den meisten TCP-Implementationen bleibt die Portnummer nach Schließen eines Sockets für eine gewisse Zeit reserviert, bevor erneut ein Socket mit diesem Port angelegt werden darf

Aus diesen Gründen werden Ports dynamisch vergeben. Nun wird jedoch ein Verfahren benötigt, welches Dienstnamen auf Portnummern abbildet.

Das Verfahren kann auch verwendet werden, um die IP-Nummern (oder auch Hostnamen) für die jeweiligen Rechner herauszufinden, wenn der Dienst nicht auf demselben Rechner erbracht wird. Auch hierfür könnte ein statisches Registrieren in einer Datei oder einem zentralen Register realisiert werden, aber dies setzt voraus, dass das Register aktuell ist und die Dienste auch immer auf den registrierten Rechnern ablaufen. In einem verteilten System mit mehreren Rechnern ist mit der Erfüllung der beiden Annahmen jedoch nicht zu rechnen. Die bereits zitierte Aussage von Lamport beschreibt die wahrscheinliche Folge.

CORBA definiert für diesen Zweck den Namensdienst, der eindeutige Namen auf Objektreferenzen abbildet. Die hier benötigte Funktionalität geht aber noch ein Stück weiter, da nicht nur einfache Namen verwendet werden, sondern Dienste auch mit Attributen versehen sein können. Das Verzeichnis besitzt somit

eine gewisse Ähnlichkeit mit einem LDAP-Server [44], wenn man die Struktur der abzuspeichernden Daten betrachtet.

Im Betrieb des Systems muss die Liste aktuell gehalten werden. Es bieten sich sowohl optimistische als auch pessimistische Verfahren an.

Ein optimistisches Verfahren würde darin bestehen, dass der Namensdienst selbst keinerlei Überprüfung der Existenz der registrierten Dienste übernimmt. Erst dem Nutzer des angeforderten Dienstes obliegt die Aufgabe, im Falle eines Versagens des angeforderten Dienstes den Namensdienst über den Fehlerzustand zu informieren. Unter dem Gesichtspunkt der Netzbelastung ist dieses Verfahren zu bevorzugen, sofern die Wahrscheinlichkeit eines Ausfalls sehr gering ist oder temporäre Störungen auftreten können, die nicht zu einer Störung der regulären Nutzung führen.

Pessimistische Verfahren versuchen das Problem im Vorfeld zu erkennen und zu beheben, bevor ein Client durch das Versagen eines Dienstes Schaden nimmt. Es gibt eine Reihe von alternativen Verfahren:

1. Der Namensdienst schickt in regelmäßigen Abständen *ping*-Anfragen an alle registrierten Dienste. Sobald eine oder mehrere dieser Anfragen nicht beantwortet wird/werden, kann davon ausgegangen werden, dass der Dienst nicht länger vorhanden ist.
2. Der Namensdienst sucht in regelmäßigen Abständen nach Diensten, indem an jeden möglichen Port eine *ident*-Anfrage geschickt wird.
3. Die Dienste melden sich in regelmäßigen Abständen beim Namensdienst. Sobald dieser eine bestimmte Zeit lang keine Nachricht von einem Dienst erhält, wird der Dienst aus der Liste entfernt.

Der Vorteil der ersten Methode besteht darin, dass sie recht einfach zu implementieren ist. Ein Nachteil besteht darin, dass es zu einer Situation kommen kann, in der der Dienst funktionsfähig erscheint, aber in Wirklichkeit ein vollkommen anderer Dienst dessen Platz eingenommen hat. Zu dieser Situation kann es kommen, wenn mehrere Dienste beendet wurden und neue Dienste auf den bisherigen Ports gestartet werden. Dieses Problem lässt sich umgehen, indem anstatt eines „ich bin da“ eine Identifizierung geschickt und diese mit der bisherigen verglichen wird. An dieser Stelle findet eine eindeutige Identifikationsnummer Verwendung.

Das zweite Verfahren hat den Vorteil, dass auch Dienste entdeckt werden, die sich nicht von selbst angemeldet haben, sich aufgrund eines Fehlers nicht anmelden konnten oder noch keinen Namensdienst vorgefunden haben. Ein Nachteil

3. Realisierung

besteht darin, dass bei nur wenigen Diensten auf einem Knoten das Verhältnis zwischen gesuchten und gefundenen Diensten sehr gering ist. Ein weiterer Nachteil besteht darin, dass mit diesem Vorgehen keine Dienste auf entfernten Rechnern überwacht werden können.

Der Nachteil beim dritten Verfahren besteht darin, dass die Dienste erst einmal wissen müssen, wo sich der Namensdienst befindet. Außerdem wird die Verantwortlichkeit für die Korrektheit der Datenbasis damit in die Obhut der Dienste verlagert, welche unter Umständen dieser Aufgabe nicht nachkommen.

Bei der Abfrage des Verzeichnisses müssen Bedingungen formuliert werden, die festlegen, welche Objekte ausgewählt werden sollen.

Zur Auswahl stehen die Übergabe von Programmcode, der mit einer Steuerungsfunktion ausgewertet wird (besonders einfach bei Scriptsprachen) oder eine spezielle Abfragesprache. Als Abfragesprachen kämen beispielsweise SQL [8] oder XQuery [19] in Betracht.

Für die Verwendung dieser Sprachen benötigt man jedoch einen Parser sowie die notwendige Programmlogik, die die entsprechenden Ausdrücke zu interpretieren vermag. SQL ist zudem noch ungeeignet, weil es für die Zusammenarbeit mit relationalen Datenbanken entwickelt wurde und nicht für Objektstrukturen. Man müsste also die Semantik einiger Anweisungen umdefinieren. Eine solche Modifikation sorgt jedoch nicht für eine kürzere Einarbeitungszeit und leichter zu wartende Programme. Das gleiche Argument trifft auch auf XQuery zu, wobei dessen Syntax noch viel komplizierter ist als die von SQL. Letztendlich ist auch nicht unbedingt mit einem Geschwindigkeitsgewinn zu rechnen, wenn eine abstrakte Abfragesprache mit Hilfe einer Scriptsprache interpretiert wird.

Von der Übertragung von beliebigem Programmcode geht allerdings eine sehr große Gefahr aus: Es ist einem Angreifer jederzeit möglich, zerstörerischen Code in das System einzuschleusen. Von Systemseite muss demzufolge eine Art *Sandbox* geschaffen werden, die derartige Angriffe abwehren kann.

Datenobjekte Jeder Dienst wird durch eine Reihe von Attributen in einer Struktur namens `SERVICEDATA` beschrieben, die teilweise schon aus dem `BOS-SERVER` bekannt sind:

Name: Benennt die Klasse des Dienstes

Host: Rechnername, auf dem dieser Dienst zu finden ist

Port: TCP-Port, hinter dem dieser Dienst Kommandos entgegen nimmt

SSL: Die Kommunikation wird über eine mit SSL gesicherte Verbindung geführt

UniqueID: Zufällig gewählte, aber eindeutige Identifikationsnummer

weitere Attribute: In der Form Schlüssel → Wert, die in eine Suchanfrage eingebunden werden können²³.

Dienstobjekt-Interface Der Dienst, der diese Dienstauffindung realisiert, wird `LOCATIONSERVER` genannt. Er bietet fünf Funktionen an:

`location_register(service)`: Registriert den Dienst unter dem angegebenen Namen mit den Attributen beim `LocationServer`.

`location_deregister(service)`: Hebt die Registrierung des Dienstes wieder auf.

`ServiceData[] location_locate(name, expr=None)`: Findet Dienste mit dem angegebenen Namen, die den angegebenen Ausdruck erfüllen.

`ServiceData[] location_locate_local(name, expr=None)`: Findet Dienste mit dem angegebenen Namen, die den angegebenen Ausdruck erfüllen. Dabei wird allerdings die Suche auf den lokalen Rechner begrenzt.

`ServiceData[] location_list()`: Listet alle Dienste, die auf dem lokalen Rechner laufen, auf.

Implementierung Nach der Registrierung eines Dienstes mit der Methode `location_register` wird dieser in eine Liste der verfügbaren Dienste eingetragen.

Im Projekt wird ein pessimistisches Verfahren eingesetzt, wobei bei jedem *Ping* die Identität des Dienstes (einschließlich der `UniqueID`) abgefragt und mit der Datenbasis verglichen wird. Freie Parameter, die im Betrieb des Systems zu optimieren wären, sind die Anzahl der Ping-Versuche, die Wartezeit auf eine Antwort und der zeitliche Abstand zwischen zwei solchen Versuchen.

Die Funktion `location_locate` ermittelt eine Liste von Diensten, die den angegebenen Namen tragen und den zugehörigen Ausdruck erfüllen.

²³Denkbar wären der Name der Relation bei einem Datenbankdienst und der Teilbereich dieser Tabelle.

3. Realisierung

Dieser zu übergebende Ausdruck muss ein berechenbarer Ausdruck sein, der in der Sprache Python beschrieben ist. Dem Ausdruck wird die Variable `x` als Referenz auf ein `Service-Data`-Objekt mitgegeben. `X` wird in die in die Rückgabeliste übernommen, wenn der Ausdruck als logisch wahr evaluiert wird. In dieser Liste können mehrere dienstbeschreibende `SERVICEDATA`-Objekte enthalten sein. Welcher Wert aus dieser Liste verwendet wird, entscheiden die Nutzer der Dienste.

Wird stets der erste verfügbare Dienst verwendet, so ergibt sich eine Art von Backup-Konstellation. Bei alternierender Auswahl eines Dienstes der Liste kann eine Lastverteilung realisiert werden.

Wurde bei einer Suche nach einem Dienst mindestens eine Instanz dieses Dienstes in der lokal vorhandenen Liste gefunden, dann wird sofort eine Liste mit den passenden `SERVICEDATA`-Objekten an den Anfrager zurückgeliefert. Ist der Dienst lokal nicht bekannt, dann muss auf den anderen Rechnern im System danach gesucht werden. Dazu dient folgender Algorithmus:

1. Lokalen `NODESERVER` auffinden und von diesem die Liste der im System vorhandenen Rechner anfordern.
2. Auf jedem Rechner der Liste nach dem dortigen `LOCATIONSERVER` suchen.
3. Jeden dieser `LOCATIONSERVER` nach dem gesuchten Dienst befragen (mit Hilfe von `location_locate_local`, um Rekursionen zu vermeiden).²⁴
4. Jeden gefundenen Dienst der Ergebnisliste hinzufügen, wenn dieser Dienst noch nicht enthalten ist und in die eigene Liste der Dienste eintragen und damit auch die Überwachung starten.

Damit hat der `LOCATIONSERVER` eine Information zu diesem Dienst in seinem eigenen Cache und spart bei einer erneuten Suche den aufwändigen Suchalgorithmus. Das Überprüfen der Verfügbarkeit kann nach einem ähnlichen Verfahren wie bei lokalen Diensten erfolgen, sofern das Verfahren vom `LOCATIONSERVER` ausgelöst wird.

Ein kleines Problem ergibt sich jedoch, wenn mehrere Dienste mit dem gleichen Namen auf verschiedenen Rechnern angeboten werden und die Dienste zu

²⁴Es wird nicht direkt der Dienst nach seiner Identität gefragt, da einige Dienste nur über SSL verfügbar sind, jedoch der `LocationServer` ohne SSL arbeitet. Damit kann die Zeit für das Aufbauen der SSL-Verbindung gespart werden.

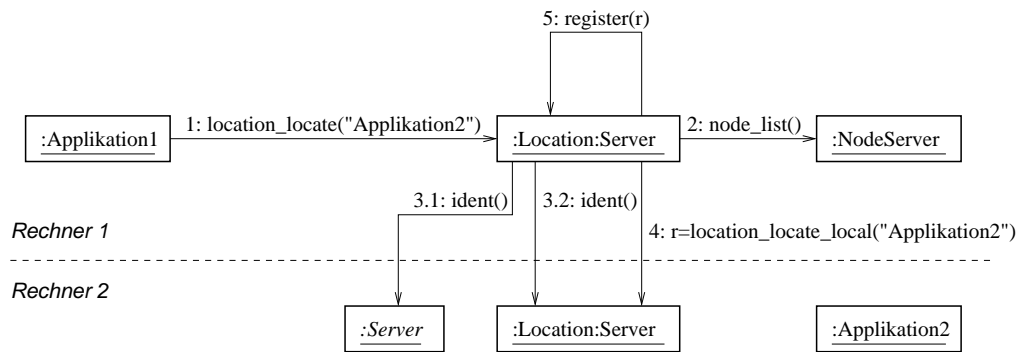


Abbildung 21: Verteilte Dienstsuche

unterschiedlichen Zeiten gestartet wurden. Es kann passieren, dass die erste Instanz des Dienstes auf Rechner A gestartet wurde. Danach sucht Rechner Z im Netzwerk nach dem Dienst, findet diesen auf Rechner A und trägt eine Referenz in seinem Cache ein. Wird nun eine zweite Instanz auf Rechner B gestartet und sucht Rechner Z erneut nach dem Dienst, so bedient er die Anfrage aus seinem Cache. Der Dienst auf Rechner B würde nicht gefunden.

Dieses Problem lässt sich lösen, indem man die Einträge nach einer gewissen Zeit aus dem Cache löscht, selbst wenn der Dienst noch erreichbar ist. Nach einem Cache-Timeout werden dann beide Instanzen des Dienstes gefunden²⁵.

Eine Abwandlung dieses Problems entsteht, wenn ein `LOCATIONSERVER` nach einem Dienst gefragt wird, der in einer Instanz auf demselben Knoten wie der `LOCATIONSERVER` und in einer weiteren Instanz auf einem anderen Knoten existiert. Weil stets der Eintrag für den lokal existierenden Dienst vorhanden sein wird, muss nach dem vorgestellten Verfahren keine Suche im gesamten System ausgeführt werden. Der Dienst auf dem anderen Knoten würde also nie gefunden werden.

3.1.3. Knotenverwaltung (NODESERVER)

Vorüberlegungen Im Abschnitt 3.1.2 (`LOCATIONSERVER`) wurde schon kurz angedeutet, dass mehr als ein Rechner am System beteiligt sein wird. Die Verwaltung der im System verwendeten Rechner (auch Knoten genannt), wird mit Hilfe des `NODESERVERS` realisiert. Jeder `NODESERVER` besitzt eine Liste aller im System aktiven Rechner.

²⁵Eine ähnliche Idee steckt hinter dem *Timeout* von *Address Resolution Protocol* (ARP) in lokalen Ethernet-IP-Netzwerken.

3. Realisierung

Dienstobjekt-Interface

node_assimilate(host): Öffnet zu dem angegebenen Rechner eine interaktive Sitzung und startet auf diesem Rechner einen BOSSERVER. Dieser BOSSERVER *kann* dann wiederum einen NODESERVER starten.

NodeData[] node_list(): Gibt die Liste der im System bekannten Rechner zurück.

node_register(node): Registriert den angegebenen Knoten beim NODESERVER. Der NODESERVER schickt daraufhin in regelmäßigen Abständen, ähnlich wie der LOCATIONSERVER, *ident*-Anfragen²⁶. Sobald ein NODESERVER unerreichbar scheint, schickt der entdeckende NODESERVER eine *node_deregister*-Nachricht an die übrigen NODESERVER.

node_deregister(host): Entfernt den Rechner aus der Liste der Rechner und stellt gegebenenfalls die Überwachung ein.

Implementierung Das Hinzufügen eines neuen Rechners erfolgt nach folgendem Algorithmus:

1. NODESERVER auf Rechner h1 wird (von einem anderen Dienst oder einem Administrator) angewiesen, Rechner h3 in das System einzugliedern.
2. Per *Secure Shell* (SSH) — oder ähnlichem — wird zu Rechner h3 eine Verbindung geöffnet und ein BOSSERVER gestartet, wobei diesem der Hostname von Rechner h1 mitgeteilt wird.
3. Der BOSSERVER startet einen NODESERVER mit einem Hinweis auf Rechner h1.
4. Der neue NODESERVER fordert vom NODESERVER von Rechner h1 eine Liste der bekannten Knoten im System an.
5. Der neue NODESERVER registriert sich bei allen anderen NODESERVERN im System und fängt an, jeden dieser anderen Knoten zu überwachen.

²⁶Dieses Verfahren führt zu exponentiellem Wachstum der Anfragen bei steigender Anzahl von Rechnern im System. Effizientere, aber aufwändigere Alternativen sind in [41] zu finden.

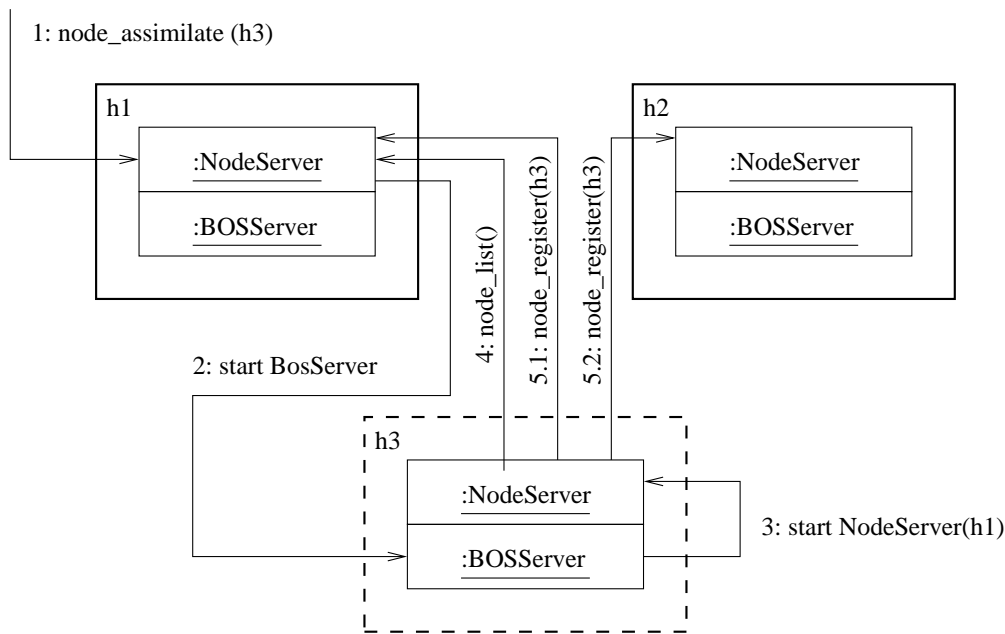


Abbildung 22: Assimilieren eines Knotens

3.1.4. Logging (LOGGINGSERVER)

Vorüberlegungen Ein weiterer Basisdienst übernimmt das Mitschreiben von entfernten Methodenaufrufen. Dieser LOGGINGSERVER kann dazu verwendet werden, um Messwerte für Statistiken über die genutzten Dienste zu erfassen. Diese Messwerte können von Administratoren oder adaptiven Lastverteilern verwendet werden, um Leistungsparameter des Systems langfristig zu verbessern.

Im Moment ist dieser Dienst noch nicht implementiert.

3.1.5. Systemmonitoring (LOADSERVER)

Vorüberlegungen Zum Zwecke der dynamischen Lastverteilung ist ein Dienst notwendig, der die Auslastung von Speicher und CPU bestimmen kann. Diese Aufgabe übernimmt der LOADSERVER.

Dienstobjekt-Interface

load_memory(): Gibt die Größe des verwendbaren Speichers zurück. Der Wert wird in Byte angegeben. Kann kein Wert ermittelt werden, so wird pauschal 100.000.000 zurückgegeben.

3. Realisierung

load_cpu(): Gibt die Last der CPU als Liste von drei Dezimalwerten zurück. Die drei Werte geben die Auslastung des Knotens in den letzten 1, 5 und 15 Minuten an. Der Wert von 1 entspricht voller Auslastung. Können keine Werte ermittelt werden, so wird dreimal 0.0 geliefert.

Implementierung Beide Funktionen arbeiten sehr betriebssystemnah und müssen gegebenenfalls angepasst werden, wenn unterschiedliche Betriebssysteme unterstützt werden sollen. Läuft der Dienst auf einem Rechner mit Linux als Betriebssystem, so werden beide Werte aus Einträgen im `proc`-Filesystem ermittelt.

Die Bestimmung des freien Speichers ist nicht einfach, da beispielsweise *Linux* freien Speicher für Buffers und Caches verwendet. Dieser Speicher kann aber freigegeben werden, sobald ein Prozess Speicher benötigt. In der von mir verwendeten Funktion wird der freie Speicher so berechnet, dass der vom Betriebssystem als frei angegebene Speicher um die Werte für Buffers und Caches erhöht wird, wobei diese Werte nur zur Hälfte in die Summe eingehen. Diese Einschränkung wird getroffen, damit keine Überbuchung der Knoten ausgelöst wird und dadurch Leistungsschmälerungen aufgrund zu wenig freien Speichers eintreten.

3.2. Datenbankdienste

Ein wesentlicher Bestandteil des Systems findet sich in der verteilten Objektdatenbank. Die Objektdatenbank besteht aus Anwendungssicht aus mehreren benannten Gruppen von Objekten, die *Kollektion* genannt werden. Für die Anwendung ist es möglich, Objekte in dieser Kollektion abzulegen, aufzufinden oder aus dieser Kollektion zu entfernen. Aus Systemsicht erbringen mehrere Dienste die Funktionalität der Datenbank.

QUERYSERVER: Realisiert die Kernfunktionalität des Objektzugriffs.

SLICESERVER: Stellt mit Hilfe von Replikation und Caching Funktionssicherheit und Performanz her.

PERSISTENCYSERVER: Speichert Objekte permanent auf einem nicht flüchtigen Medium.

CONCURRENCYSERVER: Hilfsdienst für Objektsperren in Transaktionen.

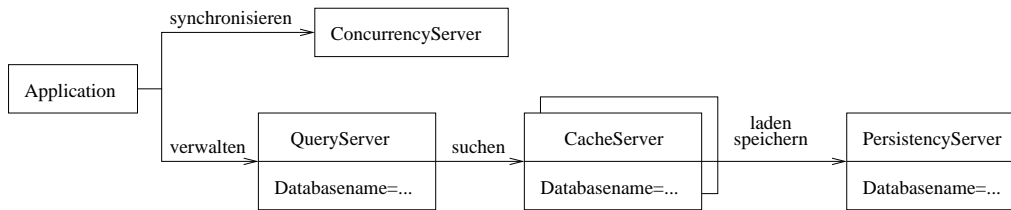


Abbildung 23: Datenbankdienste im Überblick

3.2.1. Anfragesystem (QUERYSERVER)

Vorüberlegungen Der QUERYSERVER stellt den Anwendungen das Datenbankinterface zur Verfügung. Für jede Gruppe von Objekten existiert ein oder mehrere QUERYSERVER. Die Auswahl der entsprechenden Servergruppe geschieht durch das zusätzliche Attribut `database` des `SERVICEDATA`-Objektes.

Datenobjekte Das gesamte Datenbanksystem arbeitet mit beliebig strukturierten Objekten²⁷. Die einzige Forderung an die Objekte ist, dass diese ein abzählbares und möglichst lückenlos vergebenes Attribut namens `ID` besitzen müssen, welches eindeutig ein Objekt der Kollektion bezeichnet.

Dienstobjekt-Interface

Object[] query_select(expr, prep, first=1, last=N):

Findet die Objekte, die dem angegebenen Ausdruck entsprechen und gibt aus dieser Liste die ersten `last-first` Objekte zurück, wobei bei `first` begonnen wird²⁸. Der Ausdruck `prep` dient der Vorbereitung der Suche.

query_store(object): Speichert das übergebene Objekt in der Datenbank.

query_delete(object): Löscht das übergebene Objekt aus der Datenbank.

Implementierung Mit `query_select(expr)` erhält man vom QUERYSERVER eine Liste von Objekten, welche der in `expr` angegebenen Filterregel

²⁷ Aus diesem Grund wird in der Interfacebeschreibung ganz abstrakt von `OBJECT` gesprochen.

²⁸ Der Wert für `N` wird auf 2147483647 (maximal möglicher `Int`-Wert) festgelegt, Anwendungen sollten sinnvollerweise kleinere Werte verwenden.

3. Realisierung

entsprechen. Bei `expr` handelt es sich um einen Ausdruck in der Sprache *Python*, der seinen Wahrheitswert auf der Variable `x` arbeitend bestimmen kann. Evaluiert sich dieser Ausdruck zu einer wahren Aussage, dann wird das zugehörige Objekt in die Ausgabeliste übernommen²⁹. Die Vorbereitung mit Hilfe von `prep` kann das Erzeugen von Hilfsobjekten einschließen, die in der eigentlichen Suchanfrage verwendet werden.

`Query_store(objekt)` speichert ein Objekt in der Datenbank ab. Diese Methode ist nur dann erfolgreich, wenn sichergestellt wurde, dass das Objekt tatsächlich auf (irgend)einem Datenträger gespeichert wurde und bei der nächsten Suche nach diesem Objekt das geänderte Objekt wiedergefunden wird. Wenn das übergebene Objekte eine ID von 0 trägt, so geht die Datenbank davon aus, dass es sich um ein neues Objekt handelt und vergibt eine neue ID.

Ein mit `query_delete(object)` gelöscht Objekt wird aus allen SLICESERVERN und PERSISTENCYSERVERN entfernt.

3.2.2. Performanz und Zuverlässigkeit (SLIDE SERVER)

Vorüberlegungen In der Anforderungsanalyse wurde als ein wesentlicher Aspekt eine niedrige Antwortzeit verlangt. Diese Anforderung kann ab einer bestimmten Menge an zu verwaltenden Objekten nicht länger von nur einem einzigen Rechner erbracht werden. Eine Parallelisierung der Algorithmen kann dieses Problem lösen. Damit wird eine Replikation der Daten notwendig.

In [24] werden verschiedene praktische Verfahren vorgestellt.

Datenverteilung Aus einer Primärdatenbank werden mehrere Kopien erzeugt, die jedoch lediglich einen Ausschnitt der Datenbank enthalten. Auf den Kopien wird nur lesend gearbeitet. Schreiboperationen werden mit der Primärdatenbank durchgeführt. Konsistenzprobleme aufgrund von laufenden Transaktionen sind nicht möglich, da lediglich auf der Primärdatenbank gearbeitet werden muss. Allerdings besteht die Gefahr, dass die Kopien veralten.

Datenkonsolidierung Mehrere Datenbanken werden an einer zentralen Stelle zu einer Lesekopie zusammengeführt. In dieser Konstellation entsteht die Gefahr, dass Transaktionen, die mehrere Datenbanken betreffen, zu einer inkonsistenten Sicht führen, wenn Aktualisierungen der Lesekopie nicht synchron durchgeführt werden.

²⁹Eine Begründung für dieses Vorgehen wurde im Abschnitt 3.1.2 über die Dienstauffindung bereits gegeben.

Gemeinsame Datenteilung Jeder Knoten besitzt alle Daten, die er benötigt und ist gleichzeitig für einen Teil dieser Daten auch verantwortlich. Bei Änderungen an den eigenen Daten müssen diese Änderungen weitergegeben werden³⁰. Es treten die gleichen Konsistenzprobleme wie bei der Datenkonsolidierung auf.

Datenintegration Alle Daten sind auf jedem Knoten verfügbar. Es darf sowohl gelesen als auch geschrieben werden. Mit diesem Verfahren ist eine Konsistenzsicherung nur mit sehr komplizierten Algorithmen möglich. Nach Ausfall eines Knotens sind umfangreiche *Recovery-Maßnahmen* notwendig, um den Speicherzustand mit den anderen Knoten abzugleichen.

Stand-By-Daten Die Verwendung von Stand-By-Daten zielt lediglich auf die Erhöhung der Verfügbarkeit ab. Alle Operationen werden auf der Primärkopie der Datenbank getätigt. In regelmäßigen Abständen wird diese auf eine Sekundärkopie übertragen. Fällt die Primärkopie aus, so kann auf die Sekundärkopie umgeschaltet werden.

Für dieses Projekt wird ein Verfahren verwendet, welches weitgehend der Datenverteilung entspricht. Die Aktualisierung der Kopien könnte sowohl vom QUERYSERVER als auch vom PERSISTENCYSERVER angestoßen werden. Da der QUERYSERVER schon für die Anfragen mit mehreren SLICESERVERN kommunizieren muss, kann auch die Aktualisierung in diesem Server angesiedelt werden.

Die Datenbank wird zur Lastverteilung in mehrere Teilbereiche untergliedert. Die Bestimmung der Teilbereiche der Datenbank kann ebenfalls nach verschiedenen Strategien geschehen. In [24] betrachtet Gallersdörfer dazu drei Kennzeichen der Daten:

Änderungscharakteristik: Ändern sich die Daten oft (*flüchtige Daten*, Bewegungsdaten) oder relativ selten (*Stammdaten*, Versionsdaten)?

Knotenzuordnung: Sind die Daten einem bestimmten Teil des Systems zuzuordnen (knotenabhängige Daten) oder werden sie von allen Teilen gleichmäßig genutzt (knotenunabhängige Daten)?

Verarbeitungsreichweite: Müssen die Daten für andere Knoten sichtbar sein (globale Daten) oder nicht (lokale Daten)?

³⁰Dieses Verfahren wird im AFS verwendet, um *readwrite*- und *readonly*-Replikate von *Volumes* zu verwalten.

3. Realisierung

Aus der Sicht der Anwendung sind alle Daten als nicht flüchtig zu bezeichnen, eine Zuordnung der Daten zu bestimmten Knoten kann nicht gefunden werden, und fast alle Knoten müssen potenziell auf alle Daten zugreifen können.

Aus diesen Kennzeichen werden acht Ausprägungsformen gebildet. Für eine Anwendung wie dieses Projekt (nicht flüchtige Daten, keine Knotenzuordnung und potenzieller Zugriff durch alle Knoten), wird von Gellersdörfer eine zentrale Datenspeicherung mit asynchroner Replikation vorgeschlagen.

Dienstobjekt-Interface Das Interface entspricht im Wesentlichen dem des `QUERYSERVERS`.

Object[] slice_select(expr, prep, first=1, last=N):

Findet die Objekte, die dem angegebenen Ausdruck entsprechen und gibt aus dieser Liste die ersten `last-first` Objekte zurück, wobei bei `first` begonnen wird³¹.

slice_store(object): Speichert das übergebene Objekt im Cache

slice_delete(object): Löscht das übergebene Objekt aus der Datenbank

Beim Start des `SLICESERVERS` wird diesem per Kommandozeile mitgeteilt, welche Datenbank und ggf. welchen Teilbereich dieser Datenbank der `SLICESERVER` bedienen soll. Daraufhin versucht er, einen `PERSISTENCYSERVER` mit dieser Datenbank zu finden, um von diesem dann die entsprechenden Objekte zu lesen.

3.2.3. Persistenz (`PERSISTENCYSERVER`)

Vorüberlegungen Der `PERSISTENCYSERVER` stellt die Verbindung zum Dateisystem her. Ihm obliegt die Speicherung der Objekte auf einem nicht flüchtigen Speicher und das Laden der Daten von diesem Speicher.

Es sind dabei zwei Entscheidungen zu treffen:

1. In welchem Format werden die Daten auf dem Datenträger gespeichert?
2. In welchen Einheiten wird der Platz auf dem Datenträger vergeben?

³¹Der Wert für N wird auf den größten möglichen Int-Wert — 2147483647 — festgelegt.

Datenformat Die Daten müssen auf dem Speichermedium in einem geeigneten Format abgelegt werden. Für die Eignung ausschlaggebend sind mehrere Aspekte:

- Benötigter Speicherplatz
- Portabilität des Datenformats
- Performanz beim Lesen und Schreiben
- Stabilität der Daten bei Fehlern
- Vielseitigkeit des Datenformats

Für das hier zu entwickelnde System stehen Performanz und Portabilität im Vordergrund. Auf den Gebieten Speicherplatz und Vielseitigkeit können Abstriche gemacht werden. In [2.3.3](#) wurden bereits Formate zur Datenübertragung vorgestellt. Diese Formate stehen auch für die Speicherung der Daten zur Verfügung, müssen jedoch noch einmal unter dem Gesichtspunkt der langfristigen Speicherung betrachtet werden.

Datendump in Klartext–Darstellung: Dieses Datenformat kann die verschiedensten Formen aufweisen. Eine im derzeitigen Bibliothekssystem eingesetzte Variante verwendet beispielsweise Zahlen im Bereich von 000 bis 999 als Schlüssel (sowie den gleichen Bereich noch einmal als Unterschlüssel, der dann mit einem Punkt vom Hauptschlüssel getrennt wird) und mit einem Doppelpunkt von dieser Zahl abgetrennte Werte. Das Ende eines Attributes erkennt man daran, dass ein neues Attribut begonnen hat — ein Zeilenumbruch ist also nicht notwendigerweise das Ende eines Attributes. Die Datenätze werden mit 999 und einer folgenden Leerzeile beendet. Unter dem Gesichtspunkt der Speichernutzung ist ein solches Format sehr effizient. Die Portabilität geht jedoch verloren, sobald nicht-ASCII-Zeichen (Umlaute, Sonderzeichen) in den Daten vorkommen. Die Codierung und Decodierung der Daten muss speziell für die jeweiligen Datenobjekte programmiert werden. Mehrfach ineinander geschachtelte Datenstrukturen oder beliebig lange Listen sind mit diesem Format nicht realisierbar. In den Schlüsselzahlen darf kein Fehler auftreten, weil dieser nicht kompensiert werden kann. Fehler, die Datenwerte beschädigen, beeinträchtigen die Lesbarkeit der Daten allerdings nicht.

Binärdarstellung der Laufzeitumgebung: Auf dieses Format wurde ebenfalls bereits in [2.3.3](#) eingegangen. Ergänzend ist dazu anzumerken,

3. Realisierung

dass nach einer zufälligen Beschädigung der binären Darstellung keine Aussage getroffen werden kann, ob die Daten weiterhin verwendbar sind oder unwiderruflich zerstört wurden. Eine Reparaturfähigkeit der fehlerhaften Datensätze unterliegt vielfältigen Zufällen³².

XML–Codierung nach XML–RPC: Eine Codierung in diesem Format ist aus Implementationssicht am geeignetsten, weil keine zusätzlichen Bibliotheken eingesetzt werden müssen. Nachteilig erweist sich allerdings die fehlende Semantik der *XML–Tags*, wenn die Daten von anderen Anwendungen verarbeitet werden sollen.

XML–Codierung nach RDF: Perspektivisch gesehen ist eine Codierung in diesem Format sinnvoll. Zum Zeitpunkt dieser Arbeit fehlen allerdings noch die notwendigen DTDs, um alle benötigten Attribute abzuspeichern. Für die Erzeugung der XML–Daten müsste außerdem ein entsprechender Generator entwickelt werden³³.

Speichereinheiten Orthogonal zur Frage des Datenformats der einzelnen Datensätze steht die Frage, in welcher Granularität die Datensätze im Dateisystem abgelegt werden sollen.

Die beiden Extrema sind in diesem Falle auf der einen Seite eine einzige Datei, in der alle Datensätze abgelegt werden, und auf der anderen Seite eine Datei pro Datensatz.

Die Variante der Containerdateien ist bei SQL–Datenbanken eine übliche Vorgehensweise. Jedoch besteht die Gefahr, dass das Dateisystem Dateien in der benötigten Größe nicht unterstützt³⁴. Außerdem sind die einzelnen Datensätze im Gegensatz zu SQL–Speicherstrukturen nicht von konstanter Länge, so dass keine absolute Adressierung innerhalb der Datei möglich ist. Dieses Problem ist mit indirekter Adressierung lösbar. Der Speicherplatz fragmentiert bei der Modifikation oder Löschung von Datensätzen noch immer. Letztendlich läuft dieser Ansatz auf ein virtuelles Dateisystem in einer Datei hinaus.

Anstatt nun ein eigenes Dateisystem zu implementieren, kann man auch gleich auf vorhandene und gut erprobte Dateisysteme für Datenhaltung zurückgreifen. *ReiserFS* [9] stellt beispielsweise effiziente Verfahren zum Speichern vieler kleiner Dateien zur Verfügung. Bei den in der Einleitung genannten Zahlen von 350.000 und mehr Objekten ist die Belastung für das Dateisystem noch immer sehr hoch. Um in dieser Menge von Objekten die Navigation ein wenig

³²Z.B. Stelle der Beschädigung, Redundanz der Information, Prüfsummen, Codierung.

³³In CORBA wird dieser Dienst als *Externalization–Service* bezeichnet.

³⁴2 GB ist eine häufig anzutreffende Grenze für Dateigrößen.

performanter zu gestalten (und auch vernünftige Leistungen auf Dateisystemen zu erhalten, die nicht auf schnelle Suche hin optimiert wurden), können die Dateien in Gruppen zusammengefasst und pro Gruppe ein Unterverzeichnis angelegt werden. Für diese Partitionierung bietet sich die ID-Nummer der Datensätze an. Ein mehrstufiger Index kann so leicht mit Hilfe von Modulo-Operationen³⁵ berechnet werden.

Ein Kompromiss zwischen diesen beiden Grenzwerten lässt sich durch das Speichern einer bestimmten Anzahl von Datensätzen pro Datei finden. Eine hohe Anzahl von Datensätzen reduziert die Anzahl von `open`- und `close`-Aufrufen, jedoch steigt der Aufwand für Positionierungen des Zeigers in der Datei. Die Ermittlung einer optimalen Anzahl an Datensätzen sei einer weiterführenden Arbeit vorbehalten.

Dienstobjekt-Interface

`Object[] persistency_fetch(start=1, end=2147483647):`

Lädt die angegebene Objektmenge einschließlich des durch `start`, aber ausschließlich des durch `end` identifizierten Datensatzes³⁶.

`persistency_store(object):` Speichert das übergebene Objekt in der Datenbank.

`persistency_delete(object):` Löscht das übergebene Objekt aus der Datenbank.

Implementierung Für die Codierung der Daten in den Dateien wird die Darstellung nach XML-RPC verwendet, wobei jeder Datensatz in eine einzelne Datei geschrieben wird. Die Dateien werden in einer zweistufigen Verzeichnishierarchie abgelegt.

Beim Speichern von Objekten mit einer ID mit dem Wert Null wird eine neue ID ermittelt und dem Datensatz zugeteilt. `Persistency_store(object)` gibt die ID des Datensatzes als Rückgabewert aus.

Zum Löschen von Datensätzen wird die Länge der zugehörigen Datei auf Null zurückgesetzt. Die Datei bleibt aber erhalten, um eine erneute Vergabe der ID zu verhindern und ein schnelles Auffinden der maximal vergebenen ID zu ermöglichen³⁷.

³⁵entweder im Dezimalsystem oder im Binärsystem

³⁶Diese Interpretation eines Wertebereiches entspricht der Interpretation der *Range*-Objekte in Python.

³⁷Es wird eine Art binäre Suche nach der letzten vergebenen ID ausgeführt.

3. Realisierung

3.2.4. Synchronisation (CONCURRENCYSERVER)

Vorüberlegungen Große Datenbanken bieten üblicherweise ein Konzept mit der Bezeichnung *Transaktion* an. In [42] wird eine Transaktion durch mehrere Eigenschaften gekennzeichnet:

Atomarität: Eine Transaktion wird entweder komplett ausgeführt oder gar nicht.

Konsistenzerhaltung: Eine Transaktion überträgt eine Datenbank von einem konsistenten³⁸ Zustand in einen anderen konsistenten Zustand.

Isolation: Transaktionen können parallel ablaufen und führen zu dem Ergebnis, das auch entstehen würde, wenn die Transaktionen in jeder beliebigen Reihenfolge nacheinander ausgeführt worden wären.

Persistenz: Alle Änderungen bleiben erhalten — auch im Falle von Systemfehlern.

Aus den englischen Begriffen für diese vier Eigenschaften (*atomicity, consistency, isolation, durability*) leitet sich die Abkürzung ACID ab.

Diese Eigenschaften bei parallelem Zugriff zu garantieren, erreicht man für die Eigenschaften Konsistenz und Isolation mit Hilfe von Serialisierungen der Transaktionen. Für die Gewährleistung der Serialisierbarkeit gibt es die verschiedensten Verfahren.

Diese Verfahren lassen sich in optimistische und pessimistische Verfahren unterteilen [24].

Pessimistische Verfahren versuchen Konflikte frühzeitig zu erkennen und zu vermeiden. Dies geschieht üblicherweise durch Sperrverfahren. Um dabei so genannte *Deadlocks* zu vermeiden, setzt man beispielsweise ein Verfahren namens *Two-Phase-Lock* ein.

Optimistische Verfahren lösen die Probleme nach dem Auftreten. Dazu arbeiten die Transaktionen zumeist mit drei Phasen: Lesen und Verarbeiten, Validieren, Schreiben. Optimistische Verfahren bieten sich insbesondere dann an, wenn Konflikte nur sehr selten auftreten.

Eine Schwierigkeit, die sich bei der Verwendung pessimistischer Verfahren ergibt, entsteht dadurch, dass die Datenbank einen Ablaufplan der Transaktion

³⁸konsistent im Sinne der Integritätsbedingungen

entwickeln muss. Deshalb ist es notwendig, die Transaktionsschritte in einer abstrakteren Ebene als die der Programmiersprache zu verfassen. Weit verbreiteter Standard ist die *Structured Query Language* (SQL). Alternativ könnte man die klassischen Methoden auf Metaebene abfangen und erst nach entsprechender Serialisierung weiterverarbeiten.

Die Implementierung eines CONCURRENCYSERVERS würde den Umfang dieser Arbeit übersteigen. Diese Aufgabe sei weiterführenden Arbeiten überlassen.

3.2.5. Zusammenwirken der Datenbankdienste

Es ergeben sich vier wesentliche Szenarien, die von den Datenbankdiensten erfüllt werden müssen:

1. Anlegen eines neuen Datensatzes
2. Suchen nach einem/mehreren Datensätzen
3. Modifikation eines bestehenden Datensatzes
4. Löschen eines Datensatzes

Anlegen eines Datensatzes Ein Datensatz wird von dem jeweiligen Anwendungsdienst als normales Datenobjekt angelegt. Das Objekt muss ein ID-Attribut kennen, welches mit dem Wert 0 belegt wird. Danach wird der geeignete QUERYSERVER aufgesucht und diesem das Objekt mit `query_store` übergeben. Der QUERYSERVER speichert das Objekt dann in einem PERSISTENCYSERVER mit `persistency_store`. Der Rückgabewert dieser Operation ist die ID des Objektes. Mit dieser ID können dann die geeigneten SLICESERVER gefunden werden, die für das Caching dieses Objektes verantwortlich sind. Diesen Servern wird dann das Objekt mit `slice_store` ebenfalls übergeben. Falls mehr als ein PERSISTENCYSERVER verwendet werden, müssen diese ebenfalls das neue Objekt abspeichern³⁹.

Suchen nach Datensätzen Wenn ein Anwendungsdienst ein oder mehrere Objekte sucht, so stellt dieser eine Anfrage an den QUERYSERVER. Dieser stellt

³⁹An dieser Stelle kann es zu Konsistenzproblemen kommen, da zwei PERSISTENCYSERVER parallel die gleiche ID für zwei verschiedene Datensätze generieren könnten. Mit der Einführung von Transaktionsunterstützung kann dieses Problem gelöst werden.

3. Realisierung

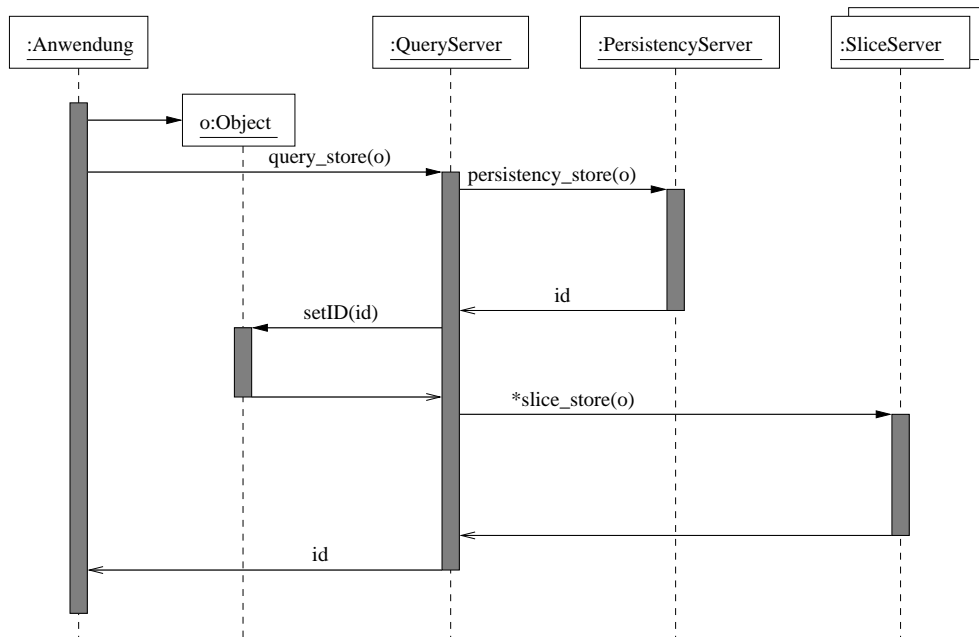


Abbildung 24: Anlegen eines Datensatzes

die Anfrage **parallel** an alle⁴⁰ SLICESERVER. Jeder SLICESERVER liefert daraufhin eine Liste von Objekten, die den gewünschten Merkmalen entsprechen. Der QUERYSERVER vereinigt diese Teilergebnisse zu einem Gesamtergebnis, wobei Wiederholungen entfernt werden.

Modifikation eines Datensatzes Zum Speichern modifizierter Datensätze wendet sich die Anwendung an den QUERYSERVER und ruft bei diesem `query_store` auf. Der QUERYSERVER geht dann nach demselben Algorithmus wie beim Anlegen eines Datensatzes vor.

Löschen eines Datensatzes Wenn eine Anwendung einen Datensatz löschen möchte, dann muss beim QUERYSERVER die Methode `query_delete(object)` aufgerufen werden. Der QUERYSERVER gibt die Löschanforderung an alle SLICESERVER und den bzw. die PERSISTENCYSERVER weiter. Die frei

⁴⁰Optimierte Algorithmen könnten hier nur maximal n Server mit dem gleichen Datenbestand pro Anfrage auswählen oder bei der Suche nach einer bestimmten ID nur die entsprechenden SLICESERVER ansprechen.

werdende ID-Nummern wird nicht neu vergeben⁴¹.

3.3. Anwendungsdienste

Anwendungsdienste nutzen die Infrastruktur aus Basis- und Datenbankdiensten, um dem Nutzer anwendungsspezifische Dienstleistungen zu erbringen. Die Anwendungsdienste sind zumeist in zwei Bestandteile untergliedert. Ein Teil ist für die Realisierung der Anwendungslogik verantwortlich. Davon programmtechnisch abgekoppelt wird das Nutzerinterface erzeugt. An welcher Stelle dieser Schnitt durchgeführt wird, hängt auch von den Fähigkeiten des Nutzerinterfacegenerators ab. Wird beispielsweise eine Beschreibungform gewählt, die keinerlei algorithmische Komponenten (wie Wiederholungen oder Alternativen) kennt, eingesetzt, so sind diese Aufgaben im Teil der Anwendungslogik zu realisieren.

Im folgenden sollen einige Dienste exemplarisch vorgestellt werden, die zur Demonstration der prinzipiellen Machbarkeit implementiert werden.

3.3.1. Objektsuche (SEARCHSERVER)

Vorüberlegung Der Informationsraum der zu verwaltenden Datenobjekte wird in verschiedenen Datenbanken abgelegt. Mit Hilfe des SEARCHSERVERS soll dem Nutzer die Möglichkeit gegeben werden, die gesuchten Objekte aufzufinden. Dazu verwendet der SEARCHSERVER eine Metasuche auf allen geeigneten⁴² Datenbanken.

Datenobjekte Der SEARCHSERVER liefert eine Liste von *Dictionaries* zurück. In den *Dictionaries* stehen zwei Schlüssel-Wert-Paare.

Der eine Wert wird unter dem Schlüsselwort *database* angelegt und bezeichnet die durchsuchte Datenbank.

Der zweite Wert beschreibt eine Liste von Suchergebnissen und ist unter dem Schlüssel *data* abrufbar.

Dienstobjekt-Interface

Object[] search_for(item): Durchsucht alle geeigneten Datenbanken nach Objekten, die alle in *item* angegebenen Worte enthalten.

⁴¹Langfristig entsteht damit eine Fragmentierung des ID-Raumes, der besonders bei rasch anwachsenden Lücken entgegengewirkt werden muss.

⁴²Welche Datenbank geeignet ist, wird im Programm festgelegt.

3. Realisierung

Implementierung Die derzeitige Implementierung der Suchfunktionalität erstellt aus der übergebenen Anfrage einen Ausdruck, der mit Hilfe von regulären Ausdrücken alle Objekte findet, bei denen alle übergebenen Worte gleichzeitig im Titel enthalten sind. Als kleine Hürde gegen Missbrauch werden nur die Anfragen bearbeitet, die minimal aus drei Zeichen bestehen. Des Weiteren werden nur maximal 250 Treffer übermittelt, um eine Überlastung sowohl des Systems als auch des anfragenden Nutzers zu verhindern.

3.3.2. Exemplarinformation

Vorüberlegung Mit diesem Dienst sollen dem Nutzer alle mitteilungswürdigen Informationen zu einem Informationsobjekt übersichtlich dargeboten werden. Dazu gehören die Attribute, die beim Objekt hinterlegt sind, aber auch Informationen über dieses Objekt, die aus anderen Datenbanken herbeizuholen wären. Dieser Dienst stellt gleichzeitig einen Anknüpfungspunkt für weitere Anwendungsdienste dar, die im Folgenden vorgestellt werden.

Für den Dienst der Exemplarinformation sind keine besonderen Dienstobjekte erforderlich. Die Funktionalität kann mit einem direkten Zugriff auf die entsprechenden QUERYSERVER oder SLICESERVER durch die Nutzerinterfacekomponente realisiert werden.

3.3.3. Empfehlungen (RECOMMENDATIONSERVER)

Vorüberlegung Der Empfehlungsdienst soll es einem Nutzer ermöglichen, einen oder mehrere Nutzer auf ein Informationsobjekt hinzuweisen. Diese Empfehlungen könnten den Nutzern per E-Mail oder auf einer personalisierten Webseite mitgeteilt werden.

Im praktischen Einsatz ist es notwendig, dass der empfehlende Nutzer sich authentifiziert hat, damit kein Missbrauch mit dem Dienst getrieben werden kann.

Wird eine Empfehlung per Webseite gewünscht, so müssen die Empfehlungen in einer Datenbank hinterlegt werden. In welcher Form die Empfehlungen den Nutzer erreichen, könnte dieser in einer Personalisierungskomponente festlegen.

Datenobjekte Zu einer Empfehlung gehören folgende Attribute:

By: Nutzer, der die Empfehlung ausspricht.

To: Nutzer, an den die Empfehlung gerichtet ist.

Date: Datum, an dem die Empfehlung angelegt wurde.

What: Referenz auf ein Datenobjekt, welches empfohlen wird. Bestehend aus ID und database.

Comment: Kleine Textnotiz, die zusätzlich übermittelt wird.

Dienstobjekt-Interface Der Empfehlungsserver wird von einem Dienst namens RECOMMENDATIONSERVER erbracht.

recommendation_recommend(recommendation): Übergibt die angegebene Empfehlung dem RECOMMENDATIONSERVER zur Bearbeitung.

Implementierung Wie der Empfehlungsdienst im Detail realisiert wird, ist nach einer gründlichen Anwendungsfallanalyse zu entscheiden. Für diesen Test werden die Empfehlungen zum Abrufen über eine Webseite abgespeichert. Über diese Webseite können die Empfehlungen vom Empfänger auch wieder gelöscht werden.

Für die Authentifizierung wird die Authentifizierungsfunktionalität des Webservers verwendet.

3.3.4. Rezension

Vorüberlegung Ein weiterer Anwendungsdienst ist die Rezension von katalogisierten Datenobjekten. Nutzern soll die Möglichkeit gegeben werden, ihre Meinung über ein bestimmtes Datenobjekt anderen Nutzern mitzuteilen.

Eine Authentifizierung der rezensierenden Nutzer ist nicht zwingend notwendig, sollte jedoch in Betracht gezogen werden, um Missbrauch zu vermeiden.

Bevor eine Rezension allen Nutzern zur Ansicht gestellt wird, ist eine Prüfung durch eine vertrauenswürdige Person sinnvoll.

Datenobjekte Eine Rezension besteht in dieser Beispielimplementierung aus folgenden Attributen:

By: Nutzer, der die Rezension abgeben möchte.

Date: Datum, an dem die Rezension angelegt wurde.

What: Referenz auf ein Datenobjekt, welches bewertet wird. Bestehend aus ID und database.

3. Realisierung

Comment: Kleine Textnotiz, die die Meinung des Bewertenden ausdrückt.

Quality : Wert auf einer Skala von 1 bis 10. (1 sehr schlecht, 10 sehr gut, -1 keine Angabe)

Approved: Wenn auf 1 gesetzt, dann wurde die Rezension kontrolliert und ist öffentlich lesbar. Ansonsten muss sie noch genehmigt werden.

Implementierung Die Rezension kommt ebenfalls ohne speziellen Anwendungsserver aus. Für die beiden Rollen Rezensierer und Bewerter der Rezensionen ist eine Nutzerverwaltung notwendig, welche entsprechende Autorisierungsmöglichkeiten bietet. Vorerst wird dem Webserver diese Aufgabe zugeteilt.

3.3.5. Ausleihe

Vorüberlegung Dieser Dienst ist für die Testimplementierung notwendig, um die Funktion „E-Mail an Ausleiher“ umsetzen zu können.

Der Funktionsumfang erstreckt sich nur auf Ausleihe und Rückgabe von ausleihbaren Objekten. Kontrolle der Leihfristen, Verlängerungen von Fristen oder Vorbestellungen sind nicht implementiert.

Datenobjekte

By: Nutzer, der das Objekt ausgeliehen hat.

Date: Datum, an dem die Ausleihe getätigt wurde.

What: Referenz auf ein Datenobjekt, welches ausgeliehen wurde. Bestehend aus ID und database.

Prolongation: Anzahl der Verlängerungen der Ausleihfrist.

Implementierung Auch dieser Dienst kann momentan noch ohne einen Anwendungslogik realisierenden Dienst implementiert werden. Alle Funktionen sind vollständig im Nutzerinterface hinterlegt.

3.3.6. E-Mail an Ausleiher (MAILSERVER)

Vorüberlegung Mit diesem Dienst soll es für einen Interessenten für ein ausleihbares Objekt möglich werden, dem Ausleiher eine E-Mail zukommen zu lassen, ohne die Identität des Ausleihers preiszugeben.

3.3. Anwendungsdienste

Damit soll es möglich sein, kurzzeitig und ohne viel Aufwand in den Besitz des betreffenden Objektes zu gelangen, ohne gleich eine Vorbestellung in der Bibliothek auslösen zu müssen. Anwendung fände diese Funktionalität besonders bei Recherche nach Büchern, wenn diese sich in längerfristiger Ausleihe befinden und eigentlich vom Interessenten nur kurz eingesehen werden möchten.

Dienstobjekt-Interface Für die Übermittlung der E-Mail steht ein Anwendungsdienst namens MAILSERVER mit folgendem Interface bereit:

mail_mailto(*by*, *to*, *text*): Schickt an den durch *to* bezeichneten Nutzer eine Mail mit dem Inhalt *text* und trägt als Absender *by* ein.

Implementierung *by* und *to* bezeichnen für diesen Test Nutzerkennzeichen im Rechenzentrum der TU Chemnitz, so dass eine Zustellung an die Adresse *to@hrz.tu-chemnitz.de* die gewünschte Person erreicht. Ebenso ist der Absender *by@hrz.tu-chemnitz.de* automatisch der richtigen Person zugeordnet.

In der Implementierung für den produktiven Einsatz ist auch an dieser Stelle auf eine entsprechende Nutzerverwaltung zurückzugreifen.

3. Realisierung

4. Problemfelder

4.1. Parallele asynchrone Aufrufe

In einigen Algorithmen ist es sinnvoll und notwendig mehrere Methodenaufrufe parallel abzusetzen. Beispielsweise kann der `QUERYSERVER` nur dann einen Vorteil aus der Parallelisierung ziehen, wenn er gleichzeitig auf mehrere `SLICE-SERVER` zugreift.

Python bietet aber ursprünglich keine Unterstützung für den gleichzeitigen Aufruf von Methoden an. Aus diesem Grund wurde eine Hilfsbibliothek geschrieben, die diese Funktionalität erbringt.

Die Bibliothek `defcall`⁴³ bietet dazu den Funktionsaufruf `defcall(servers, methodname, *params)` an.

Servers: enthält eine Liste von Objekten, denen die entsprechende Botschaft übermittelt werden soll.

Methodname: ist der Name der Methode, die durch die Botschaft aktiviert werden soll.

***params:** enthält die übrigen Funktionsparameter, die der Methode mitgegeben werden sollen.

Die Funktion `defcall` legt für jeden Server nun einen eigenen *Thread* an, der als einzige Aufgabe die Ausführung der Methode an diesem Server übernimmt. Nachdem jeder *Thread* ein Ergebnis des Methodenaufrufes bekommen hat, werden die Ergebnisse in einer Liste zusammengefasst und der aufrufenden Anwendung zurückgegeben.

Zukünftige Verbesserungen dieses Moduls müssten sich mit der Behandlung von *Exceptions* und der Realisierung einer zeitlichen Beschränkung der Ausführungszeiten befassen.

Im Anhang ist der Quellcode dieses Moduls zu finden.

4.2. Defizite von XML-RPC

Die XML-RPC-Bibliothek besitzt zwei wesentliche Probleme, die zum Teil auch im Standard begründet sind.

Langfristig ist eine entsprechende Korrektur in der XML-RPC-Bibliothek beziehungsweise im Standard anzustreben.

⁴³von englisch *deferred* — verzögert

4. Problemfelder

4.2.1. Dictionaries in XML-RPC

Das erste Problem betrifft die Verwendung von *Dictionaries*.

Es ist nicht möglich, andere Typen als Zeichenketten als Schlüsselwerte für diese Datenstrukturen zu verwenden. Besonders innerhalb der Datenbankschicht ist dies ein Nachteil, weil die eindeutige Identifikationsnummer der Datenobjekte eine Zahl ist. Der Programmierer muss an dieser Stelle stets die Zahl in eine Zeichenkette konvertieren, bevor er sie als Schlüssel verwendet.

4.2.2. Objekte in XML-RPC

Der XML-RPC-Standard macht keine Aussagen über die Behandlung von allgemeinen Objekten. Die vorliegende Python-Implementierung setzte Objekte unbekannter Klassen ursprünglich in Strukturen um, bei denen aus den Attributen der Objekte Elemente der Strukturen wurden. Mit dieser Herangehensweise verliert die gesamte Objektorientierung ihre Konsistenz, da aus Methoden nun Funktionen werden, die in entsprechenden Bibliotheken zusammengefasst werden müssten. Die Zuordnung der Datenstrukturen zu den jeweiligen Funktionsbibliotheken müsste vom Programmierer in jedem Programm manuell realisiert werden. Damit entsteht unnötiger Ballast, der zu Fehlern führen kann.

Aus diesem Grund wurde die XML-RPC-Bibliothek um die Dateneinheit des Objektes erweitert. Es mussten die Klassen `Marshaller` und `Unmarshaller` entsprechend erweitert werden.

Die Serialisierung eines Objektes erfolgt in der `dump_instance`-Methode (Abbildung 25), welche die Methode `dump_object` (Abbildung 26) aufruft. Letztere erwartet einen Klassennamen sowie ein Dictionary bestehend aus Attributnamen und Attributwerten des Objektes.

```
def dump_instance(self, value):
    # check for special wrappers
    if value.__class__ in WRAPPERS:
        value.encode(self)
    else:
        self.dump_object(value.__class__, value.__dict__)

dispatch[InstanceType] = dump_instance
```

Abbildung 25: Methode `dump_instance`

Die Deserialisierung erfolgt in der Methode `end_object` (Abbildung 27) der Klasse `Unmarshaller`. Dazu wird zuerst das *Package* geladen, in welchem die Klasse des zu ladenden Objektes definiert wurde. Das ist auch der

```

def dump_object(self, classname, value):
    self.container(value)
    write = self.write
    write("<value><object class='%s'>"%(classname))
    for k, v in value.items():
        write("<member>")
        write("<name>%s</name>" % escape(k))
        self.__dump(v)
        write("</member>")
    write("</object></value>")

```

Abbildung 26: Methode dump_object

Grund weswegen mit XML-RPC zu verwendenden Klassen immer in einem eigenen *Package* definiert werden müssen. Danach wird eine Instanz dieser Klasse angelegt. Dazu wird der Konstruktor ohne Angabe von Parametern aufgerufen. Deshalb muss der Konstruktor der zu serialisierenden Objekte ohne Parameter aufrufbar sein. Entweder indem der Konstruktor keine Parameter kennt oder indem die möglichen Parameter mit Vorgabewerten belegt sind.

```

def end_object(self):
    mark = self._marks[-1]
    del self._marks[-1]
    fullclassname = self._attr[-1]
    del self._attr[-1]
    (package, classname) = string.split(fullclassname, ".")
    packageref = __import__(package)
    objref = packageref.__dict__[classname]()
    items = self._stack[mark:]
    for i in range(0, len(items), 2):
        objref.__dict__[items[i]]=items[i+1]
    self._stack[mark:] = [objref]
    self._value = 0
    dispatch["object"] = end_object

```

Abbildung 27: Methode end_object

Es sollte darüber nachgedacht werden, diese Erweiterungen in den Standard einfließen zu lassen.

4.3. Multithreading, Konsistenz, Deadlocks

Eine schwierige Entscheidung steht an, wenn festgelegt werden soll, ob ein Dienst als *singlethreaded* oder *multithreaded* Server implementiert werden soll.

Da zur Überprüfung der Funktionsbereitschaft von Diensten ein regelmäßiger Aufruf der `ident()`-Methode vom `LOCATIONSERVER` vorgenommen wird, ist es zwingend notwendig, dass auf diese Anfragen rechtzeitig geantwortet wird. Sollte ein Dienst Methoden anbieten, die eine sehr lange Abarbeitungszeit haben, so ist die Verwendung von *Threads* anzuraten.

Gegen die Verwendung von *Threads* spricht allerdings die Gefahr von Inkonsistenzen im Datenbestand, die beim parallelen Zugriff durch mehrere Klienten eintreten könnten.

Eine genaue Analyse der eingesetzten Algorithmen ist demzufolge dringen angeraten. Insbesondere die Datenbankschicht sollte auf diese Gefahren hin noch einmal gründlich untersucht werden, sobald die Transaktionsfähigkeit nachgerüstet wird.

Verklemmungen (*Deadlocks*) können sowohl innerhalb als auch zwischen zwei Anwendungen auftreten, wenn zyklische Abhängigkeiten zwischen einzelnen *Threads* oder Diensten entstehen. Da es erheblich schwieriger ist, in einer verteilten Anwendung Verklemmungen zu erkennen und zu lösen, sollte im Voraus darauf geachtet werden, dass keine entstehen können.

4.4. SSL

Für Python existieren mehrere SSL-Implementierungen. Jedoch einzig das Paket `M2Crypto` [16] hat einen Zustand erreicht, den man als einsetzbar bezeichnen kann. Mit `M2Crypto` mitgeliefert wird eine SSL-fähige XML-RPC-Bibliothek, die auf Klientenseite ebenfalls verwendet wird.

Für das Anlegen eines SSL-fähigen Servers wird die Klasse `TCPServer` aus dem Paket `SocketServer` erweitert. Dazu wird der ursprünglich verwendete `Socket` durch eine so genannte `Connection` aus dem SSL-Paket ersetzt. Diese `Connection` verlangt einen SSL-Kontext als Parameter, der wiederum einige Initialisierungen erfordert, die in Abbildung 28 dargestellt sind.

Die Funktion der entsprechenden Aufrufe ist aus der Dokumentation zu `OpenSSL` [14] zu entnehmen, da die Dokumentation von `M2Crypto` bis zum jetzigen Zeitpunkt lediglich aus einigen unkommentierten Beispielen besteht.

Die Klientenseite ist erheblich einfacher zu programmieren. Als Server-Proxy wird eine Instanz der Klasse `Server` aus der bei `M2Crypto` mitgelieferten `xmlrpc.lib2` angelegt, die ein entsprechendes Kontextobjekt übergeben bekommt. In das Kontextobjekt ist lediglich das entsprechende Zertifikat mit

```

ctx = SSL.Context('sslv23')
ctx.load_cert('../certs/server.pem')
ctx.load_client_ca('../certs/ca.pem')
ctx.load_verify_info('../certs/ca.pem')
ctx.set_verify(SSL.verify_peer | SSL.verify_client_once |
               SSL.verify_fail_if_no_peer_cert, 1)
ctx.set_allow_unknown_ca(0)
ctx.set_session_id_ctx('ssl')
ctx.set_tmp_dh('../certs/dh1024.pem')

```

Abbildung 28: Anlegen eines SSL-Server-Contexts

`load_cert` zu laden.

Die Verwendung von SSL unterliegt einem erheblichen Mangel: Der Aufwand für die Initialisierung der gesicherten Verbindung braucht erheblich mehr Zeit als der Aufbau einer herkömmlichen TCP-Verbindung. Der zeitliche Nachteil beläuft sich eigenen Messungen zufolge auf den Faktor 10. Erst mit der Verwendung von HTTP 1.1 kann dieser Mangel ausgeglichen werden, indem einmal angelegte Verbindungen mehrfach nacheinander genutzt werden.

4.5. Python im Webserver

Für die Erstellung und Auslieferung der HTML-Seiten für die HTTP-Klienten ist ein entsprechender Server notwendig. Es kommen sowohl ein in Python realisierter Spezial-HTTP-Server als auch ein Standard-HTTP-Server (z.B. *Apache*) in Frage.

Ein in Python realisierter HTTP-Server hat den Vorteil, dass er unmittelbar die notwendige Funktionalität zur Verbindung mit dem System aufnehmen kann. Ein Standard-Server muss um diese Funktionalität erst erweitert werden. Allerdings haben diese Standard-Server den Vorteil einer sehr langen praktischen Erprobung und bieten viele nützliche Fähigkeiten an (Umfangreiche Protokollierungsfunktionalität, Authentifizierung, geschicktes Subprozessmanagement, SSL-Fähigkeit, Virtuelle Server), die bei den meisten Python-Servern noch nicht implementiert wurden.

Der populärste Standard-Server ist der *Apache HTTP Server* [10]. Für diesen gibt es mehrere Python-Module, von denen drei näher betrachtet werden sollen:

1. `Mod_Snake` [11]
2. `Mod_Python` [12]

4. Problemfelder

3. PyApache [13]

Nach Aussage von der FAQ zu `mod_python` sei dies schneller und wird von mehr Nutzern eingesetzt, während `mod_snake` mehr Konfigurationsoptionen und modernere Techniken bietet. Beide Module greifen aber sehr tief in die internen Strukturen von Apache ein und erlauben damit die Bearbeitung der Anfragen in jeder beliebigen Stufe der Abarbeitung im Server. Diese Fähigkeit ist aber überflüssig. Es wird lediglich eine Möglichkeit gesucht, Python-Skripte auszuführen und deren Ausgabe per HTTP weiterzureichen ohne für jeden Aufruf eine neue Python-Umgebung anlegen zu müssen. Für diese einfache Aufgabe reicht das weitaus kleinere PyApache vollkommen aus.

PyApache startet für jeden Kindprozess des Servers einen eigenen Python-Prozess. Für jede Anfrage wird ein neuer Subinterpreter gestartet. Dieser hat einen eigenen Namensraum, der von anderen Subinterpretern nicht erreichbar ist. Für Daten, die für mehrere Anfragen innerhalb eines Kindprozesses von Interesse sein könnten, gibt es die Variable `__persistdict__` im Modul `__builtins__`⁴⁴.

Um Python-Skripte ausführen zu können, muss für das entsprechende Verzeichnis die Option `ExecCGI` aktiviert sowie die Skripte als ausführbar gekennzeichnet werden.

Mit vier Anweisungen (Abbildung 29) in der Konfigurationsdatei des Apache-Servers wird das Modul aktiviert.

```
AddHandler application/x-python-httpd-cgi .py
AddHandler application/x-python-httpd-cgi .pyc
LoadModule PyApache_module modules/mod_pyapache.so
AddModule mod_pyapache.c
```

Abbildung 29: Aktivieren des Moduls PyApache

⁴⁴Natürlich kann damit keine Sitzungsverwaltung realisiert werden, weil nicht gesichert ist, dass die zu einer Sitzung gehörenden Aufrufe stets zum selben Kindprozess geleitet werden.

5. Zusammenfassung und Ausblick

5.1. Entwicklungsstand

Der implementierte Prototyp zeigt, dass eine verteilte Anwendung mit einer Scriptsprache unter Verwendung von offenen Protokollen realisierbar ist. Es wurde eine Anwendungsarchitektur geschaffen, die sowohl fehlertolerant als auch skalierbar ist. Die offenen Schnittstellen ermöglichen eine leichte Erweiterbarkeit um zusätzliche Funktionalitäten und die Einbindung bisheriger Systeme.

Die prototypenhaft implementierten Anwendungsdienste zeigen die Verwendbarkeit des Systems.

Um das System in den Produktionsbetrieb zu übernehmen, bedarf es allerdings noch erheblicher Erweiterungen und Verbesserungen. Insbesondere die Anwendungsdienste erfordern eine Anpassung an praktische Anforderungen. Im nächsten Abschnitt werden weitere Ansatzpunkte für folgende Arbeiten aufgezeigt.

5.2. Weiterführende Aufgaben

5.2.1. Mehrdimensionale Suche

Der SEARCHSERVER stellt nur eine äußerst begrenzte Suchmöglichkeit bereit. Anstatt nur nach Worten im Titel zu suchen, wäre auch eine Suche nach allen anderen Attributen und einer beliebigen logischen Verknüpfung dieser Attributbelegungen wünschenswert.

Um diese Funktionalität zu implementieren sind mehrere Teilprobleme zu lösen:

1. Wie werden diese Suchanfragen vom Nutzer formuliert?
 - Gibt es eine spezielle Grammatik?
 - Wird die Anfrage in einer dynamisch anpassbaren Oberfläche zusammengestellt?
2. In welcher Form werden die Suchanfragen an den QUERYSERVER gegeben?
 - Ist Python an dieser Stelle wirklich die beste Wahl?
 - Welche Vorteile bieten Alternativen wie XQuery oder ein objekt-orientierter SQL-Dialekt?

5. Zusammenfassung und Ausblick

3. Wie kann die Suche effizient nach mehreren Parametern erfolgen?

- Ist es sinnvoll, die Datenbasis in einer Iteration mit einem komplexen Auswahlfilter zu bearbeiten?
- Bringt ein mehrstufiges Filterverfahren Geschwindigkeitsvorteile?

5.2.2. Unscharfes Stringmatching

Bei der Suche nach Text ist es leicht möglich, dass der Nutzer sich bei der Eingabe vertippt. Weiterhin ist nicht sichergestellt, dass die Einträge in der Datenbank fehlerfrei sind oder ob nicht der zu suchende Text tatsächlich in einer von der Norm abweichenden Schreibweise erfasst wurde.

Wenn die Suche bei solchen Anfragen brauchbare Ergebnisse liefern soll, dann müssen geschicktere Verfahren als einfaches *Patternmatching* verwendet werden. In [31] werden einige Ansätze zur Lösung dieses Problems beschrieben.

5.2.3. Synonymsuche

Eine weitere Möglichkeit, die Qualität der Suche zu verbessern, stellt die Möglichkeit der Suche nach Synonymen dar. Dabei ist es denkbar, dass sowohl nach bedeutungsverwandten Worten gesucht wird, als auch nach Oberbegriffen.

Zum Beispiel könnte bei einer Suche nach *Eiszeit* automatisch auch nach *Vereisungsperiode* und *Erdzeitalter* gesucht werden.

Zu ermitteln wäre, ob diese Suche automatisch ausgeführt werden soll, wenn die Anzahl der Ergebnisse sehr klein ist oder der Nutzer dies manuell auslösen soll. Des Weiteren ist zu klären, wie die Synonymsuche im System effizient eingebettet werden kann.

5.2.4. Fremdsprachige Suche

Der Synonymsuche nicht unähnlich ist die Suche nach dem jeweiligen Begriff in verschiedenen Sprachen. Sowohl fremdsprachige Eingaben als auch Datenobjekte, die in einer anderen Sprache als die der Suchanfrage beschrieben sind, sollen gefunden werden.

Für die Umsetzung gelten die gleichen Anforderungen wie bei der Synonymsuche.

5.2.5. Umstellung auf RDF im Speicher und für Fremdzugriffe

Die bisherige Implementierung nutzt eine externe Datendarstellung, die mit Hilfe der xmlrpc-Bibliothek generiert wird. Wie im Beispiel auf Seite 25 gezeigt,

ist die Codierung rein syntaktisch. RDF erlaubt eine semantische Codierung, die beispielsweise bei der Weiterverarbeitung mit anderen Werkzeugen von Vorteil ist. Sobald die entsprechenden *Vocabularies* verfügbar sind, lassen sich aus diesen auch automatisch Teile des Nutzerinterfaces gestalten⁴⁵.

Um RDF einsetzen zu können, müssen:

- entsprechende *Vocabularies* ausgewählt oder entwickelt werden,
- Codierer und Decodierer geschrieben werden sowie
- Dienstschnittstellen angepasst werden, um die Datenobjekte in RDF-Codierung abrufen zu können.

5.2.6. Transaktionsfähigkeit der Datenbank

Bisher ist die Datenbank noch nicht transaktionsfähig. Das ist ein großer Nachteil, sobald komplexere Operationen ausgeführt werden sollen.

In [42] und [26] werden dazu verschiedene Strategien vorgestellt, von denen es gilt, die geeignetste zu finden.

Diese Verfahren sind danach im CONCURRENCYSERVER zu implementieren und durch den QUERYSERVER nutzbar zu machen.

5.2.7. GUI

Die Erzeugung der HTML-Seiten wird im Moment durch Python-Scripte realisiert, die im Webserver ausgeführt werden. Es werden keinerlei Bibliotheken verwendet, die beispielsweise *Sessionmanagement*, Personalisierung des Angebotes oder eine automatische Konvertierung der XML-Strukturen realisieren.

Um die Gestaltung der Oberfläche zu flexibilisieren, sollte nach alternativen Techniken gesucht werden. Eine solche Technik könnte XSLT[15] sein. Aber auch andere templatebasierte Technologien wären einsetzbar. Als dritte Gruppe in diesem Kontext würden alle Arten von *serverside Scripting* einzuordnen sein.

Bei der Bewertung dieser Technologien ist darauf zu achten, dass natürlich leichter Zugriff auf die XML-RPC-Schnittstelle der verschiedenen Dienste ermöglicht werden muss. Ein weiteres Kriterium ist eine ausreichende Performanz bei angemessenem Hardwareaufwand. Nicht zu unterschätzen ist außerdem eine leichte Erlernbarkeit, die gegeben sein muss, um dieses Teilsystem langfristig von geeigneten Personen⁴⁶ pflegen zu können.

⁴⁵Festlegung, was angezeigt wird, wonach gesucht werden kann.

⁴⁶Ein GUI wird besser von Arbeitspsychologen oder Designern und nicht Informatikern entworfen.

5. Zusammenfassung und Ausblick

5.2.8. Effizientere Kommunikation

Die derzeit verwendete XML-RPC-Bibliothek verwendet HTTP 1.0 [39] zum Übertragen der Anfragen. Die Spezifikation dieses Protokolls legt fest, dass jede Übertragung aus zwei Phasen besteht: einer Anfrage und einer Antwort. Nachdem diese beiden Datenübertragungen abgeschlossen sind, wird die Verbindung getrennt. Eine neue Übertragung erfordert den Aufbau einer neuen Verbindung.

Dieses Vorgehen ist nicht sinnvoll, wenn viele kleine Datenpakete übertragen werden müssen oder regelmäßig ein Datentransfer stattfinden muss. Auch aus diesem Grund wurde HTTP Version 1.1 [38] entwickelt.

Um HTTP 1.1 nutzen zu können, müssen nur die xmlrpc-Bibliotheken angepasst werden. Der Rest des Systems bleibt unverändert.

5.2.9. Sicherheit

Ein Thema, was in der gesamten Arbeit nur am Rande betrachtet wurde, ist die Sicherheit. In Kapitel 2.2 wurde bereits aufgeführt, welche Teilaufgaben ein Sicherheitssystem zu erfüllen hat.

Die **Authentifizierung** der Nutzer gegenüber der Anwendung kann auf zwei Wegen geschehen. Entweder wird die Authentifizierung dem Webserver überlassen oder die Anwendung realisiert eine eigene Authentifizierungskomponente. Die Vor- und Nachteile beider Verfahren sind gegeneinander aufzuwiegen.

Die Komponente der **Autorisierung** kann ebenfalls in den beiden zuvor genannten Komponenten realisiert werden. Es könnte sich aber als zweckmäßig erweisen, den Webserver nur zu einer Art Vorauswahl der berechtigten Nutzer heranzuziehen und die Details der Autorisierung den jeweiligen Anwendungen überlassen. Im Zusammenspiel mit der Authentifizierung wäre beispielsweise eine Art Tokensystem denkbar.

Integritätssicherung und **Vertraulichkeit** sind mit Hilfe von SSL gut herstellbar. Jedoch wurde in der Umsetzung des Prototypen an vielen Stellen darauf verzichtet, von SSL gesicherte Verbindungen aufzubauen, da der Verbindungsaufbau mit SSL erhebliche Zeit beansprucht. Mit der Implementierung von HTTP 1.1 würde die Anzahl der aufzubauenden Verbindungen reduziert, so dass die zusätzlich gewonnene Zeit in zusätzliche Sicherheit investiert werden kann.

In [27] wird eine mögliche Sicherheitsarchitektur vorgeschlagen.

5.2.10. Dynamische Knotenkontrolle

Der derzeitige Prototyp ist noch nicht in der Lage, die einzelnen Dienstobjekte automatisch zu verteilen. Für einen praktischen Einsatz wäre jedoch eine Auto-

matisierung der Verteilung notwendig, die in unterschiedlichen Stärken realisiert werden könnte:

- Automatische Hinzufügung von Knoten zum System
- Verteilung beliebiger Datenbank- und Anwendungsdienste
- Verteilung von SLICESERVERN der Datenbanken

Eventuell sind Methoden der künstlichen Intelligenz nutzbringend anwendbar.

Des Weiteren müssen verschiedene Optimierungskriterien in Betracht gezogen werden, wie beispielsweise:

- Sind alle notwendigen Dienste verfügbar?
- Ist die Systemperformanz ausreichend?
- Können „billige“ Poolrechner eingesetzt werden?
- Wie ist das tägliche Lastprofil des Systems?
- Behindert das System andere Nutzer?

Schließlich ist zu erörtern, ob die dynamische Lastkontrolle und Dienstverteilung innerhalb des Systems angesiedelt wird oder besser als unabhängiges System agiert.

5. Zusammenfassung und Ausblick

A. Programmbeispiele

A.1. Sun-RPC Benchmark

```
class MyPacker(Packer):
    def pack_mydata(self, data):
        self.pack_int(data["a"]);
        self.pack_float(data["b"]);
        self.pack_string(data["c"]);
    def pack_mydatalist(self, data):
        self.pack_list(data, self.pack_mydata)
```

Abbildung 30: XDR-Packer

```
class MyUnpacker(Unpacker):
    def unpack_mydata(self):
        data = {}
        data["a"] = self.unpack_int();
        data["b"] = self.unpack_float();
        data["c"] = self.unpack_string();
        return data
    def unpack_mydatalist(self):
        return self.unpack_list(self.unpack_mydata)
```

Abbildung 31: XDR-Unpacker

A. Programmbeispiele

```
class S(UDPServer):
    def handle_4(self):
        arg = self.unpacker.unpack_mydatalist()
        self.turn_around()
        self.packer.pack_mydatalist(arg)

s = S('', 0x20000000, 1, 0)
s.packer = MyPacker()
s.unpacker = MyUnpacker('')
s.register()
try:
    s.loop()
finally:
    s.unregister()
```

Abbildung 32: Sun-RPC Server

```
import sys, T
host = sys.argv[1]
class C(UDPClient):
    def call_4(self, arg):
        return self.make_call(4, arg,
                               self.packer.pack_mydatalist,
                               self.unpacker.unpack_mydatalist)

c = C(host, 0x20000000, 1)
c.packer = MyPacker()
c.unpacker = MyUnpacker('')
element = { "a": 47, "b": 23.5, "c": "hello world" }
data = []
for i in range (0,100):
    data.append(element)
T.TSTART()
for i in xrange(0,5000):
    c.call_4(data)
T.TSTOP()
```

Abbildung 33: Sun-RPC Client

A.2. CORBA Benchmark

```

module Benchmark {
  interface BenchmarkIF {
    struct Data {
      short a;
      double b;
      string c;
    };

    typedef sequence<short> intVector;
    typedef sequence<Data> dataVector;

    short ping();
    string call2(in string arg);
    Data call3(in Data arg);
    intVector call4(in intVector arg);
    dataVector call5(in dataVector arg);
  };
};

```

Abbildung 34: CORBA Interfacebeschreibung

```

import sys
from Fnorb.orb import BOA, CORBA
import Benchmark, Benchmark_skel

class BenchmarkServer(Benchmark_skel.BenchmarkIF_skel):
  def call5(self, arg):
    return arg

orb = CORBA.ORB_init(argv, CORBA.ORB_ID)
boa = BOA.BOA_init(argv, BOA.BOA_ID)
obj = boa.create('fred', BenchmarkServer._FNORB_ID)
impl = BenchmarkServer()
boa.obj_is_ready(obj, impl)
f = open('Server.ref.%s'%(sys.argv[1]), 'w')
f.write(orb.object_to_string(obj))
boa._fnorb_mainloop()

```

Abbildung 35: CORBA Server

A. Programmbeispiele

```
import sys, T
from Fnorb.orb import CORBA
import Benchmark

orb = CORBA.ORB_init(argv, CORBA.ORB_ID)
stringified_ior = open('Server.ref.server =
orb.string_to_object(stringified_ior)
if server is None:
    raise 'Nil object reference!'
if not server._is_a('IDL:dstc.edu.au/Benchmark/BenchmarkIF:1.0'):
    raise 'This is not a BenchmarkIF server!'

element = server.Data(42, 23.5, "hallo world")
data = []
for i in range (0,100):
    data.append(element)
T.TSTART()
for i in range (0,5000):
    server.call5(data)
T.TSTOP()
```

Abbildung 36: CORBA Client

A.3. XML-RPC Benchmark

```
import rpclib

class BenchmarkServer(rpclib.GenericServer):
    pass

class BenchmarkRequestHandler(rpclib.GenericRequestHandler):
    def echo(self, data):
        return data

server = BenchmarkServer(8000, BenchmarkRequestHandler)
server.serve_forever()
```

Abbildung 37: XML-RPC Server

```
import sys, xmlrpclib, T

server = xmlrpclib.Server("http://%s:8000"%(sys.argv[1],))

data = []
for i in range(0,100):
    element = { "a": 47, "b": 23.5, "c": "hello world"}
    data.append(element)
T.TSTART()
for i in xrange(0,5000):
    server.echo(data)
T.TSTOP()
```

Abbildung 38: XML-RPC Client

A. Programmbeispiele

A.4. Modul defcall

```
from threading import Thread, Event
class Call(Thread):
    def __init__(self, function, args, waker=None):
        Thread.__init__(self)
        self.function=function
        self.args=args
        self.result=None
        self._hasResult=0
        self.waker=waker
        self.start()
    def run(self):
        self.result = apply(self.function, self.args)
        self._hasResult=1
        if self.waker:
            self.waker.set()
    def hasResult(self):
        return self._hasResult
    def getResult(self):
        return self.result

def defcall(servers, methodname, *params):
    waker = Event()
    calls = []
    for s in servers:
        method=s.__getattr__(methodname)
        calls.append(Call(method, args=params, waker=waker))
    res = []
    while calls:
        waker.wait()
        for call in calls:
            if call.hasResult():
                res.append(call.getResult())
                waker.clear()
                calls.remove(call)
    return res
```

Abbildung 39: Modul defcall

B. Messwerte

Knoten	UDP		TCP	
	Ping-Pong	Hello World	Ping-Pong	HelloWorld
Typ1 lokal	9.45	10.56	4.37	4.32
Typ1 remote	25.12	25.22	16.87	16.93
Typ2 lokal	2.02	2.29	0.81	0.80
Typ2 remote	13.00	14.99	10.44	10.29
Typ3 lokal	1.99	2.27	0.78	0.77
Typ3 remote	8.02	8.24	3.87	3.88
Typ4 lokal	2.10	2.38	0.91	0.90
Typ4 remote	9.37	10.47	7.26	7.32
Typ5 lokal	1.77	2.00	0.84	0.85
Typ5 remote	10.48	10.06	8.63	8.63

Tabelle 4: Benchmark: Sockets
[in Sekunden]

Knoten	Ping-Pong	Hello World	Struktur	Liste v. Int	Liste v. Strukt
Typ1 lokal	20.84	23.22	27.31	261.36	111.75
Typ1 remote	17.63	20.25	25.18	309.92	131.23
Typ2 lokal	6.39	7.27	8.60	127.44	53.42
Typ2 remote	7.69	8.79	10.16	141.55	60.50
Typ3 lokal	6.12	7.03	8.19	119.87	50.36
Typ3 remote	6.12	6.96	7.99	120.51	51.09
Typ4 lokal	6.01	6.76	7.91	114.62	47.43
Typ4 remote	6.14	6.97	8.01	115.72	48.06
Typ5 lokal	5.05	5.70	6.65	96.24	42.17
Typ5 remote	5.28	5.98	7.46	99.04	41.29

Tabelle 5: Benchmark: SunRPC
[in Sekunden]

B. Messwerte

Knoten	Ping-Pong	Hello World	Struktur	Liste v. Int	Liste v. Strukt
Typ1 lokal	84.25	84.66	92.89	228.08	824.68
Typ1 remote	88.41	87.14	100.40	215.35	836.26
Typ2 lokal	30.64	31.09	34.04	87.75	336.07
Typ2 remote	30.11	30.96	34.01	90.45	369.96
Typ3 lokalk	29.32	30.02	32.80	84.11	323.71
Typ3 remote	27.42	27.81	30.62	82.07	337.65
Typ4 lokal	28.76	29.29	32.15	81.34	306.05
Typ4 remote	25.91	26.47	29.00	78.31	303.47
Typ5 lokal	24.49	24.90	27.42	68.67	257.28
Typ5 remote	21.93	22.43	24.61	66.03	259.37

Tabelle 6: Benchmark: CORBA
[in Sekunden]

Knoten	Ping-Pong	Hello World	Struktur	Liste v. Int	Liste v. Strukt
Typ1 lokal	73.15	98.88	100.37	277.12	1993.22
Typ1 remote	100.47	101.87	101.87	314.57	2283.68
Typ2 lokal	50.01	50.00	50.00	104.18	2488.71
Typ2 remote	33.60	33.99	49.98	150.44	1077.29
Typ3 lokal	50.02	50.01	50.02	99.20	770.56
Typ3 remote	50.01	50.01	50.02	102.01	800.21
Typ4 lokal	50.01	50.00	50.02	90.81	690.83
Typ4 remote	50.07	50.39	50.07	100.27	734.72
Typ5 lokal	50.01	50.00	50.00	80.43	584.47
Typ5 remote	50.38	50.94	50.98	83.75	628.37

Tabelle 7: Benchmark: XML-RPC
[in Sekunden]

Knoten	UDP-Sockets	TCP-Sockets	SunRPC	CORBA	XML-RPC
Typ1 lokal	4.37	9.45	20.84	84.25	74.15
Typ1 remote	16.87	25.12	17.63	88.41	100.47
Typ2 lokal	0.81	2.02	6.39	30.64	50.01
Typ2 remote	10.44	13.00	7.69	30.11	33.60
Typ3 lokal	0.78	1.99	6.12	29.32	50.02
Typ3 remote	3.87	8.02	6.12	27.42	50.01
Typ4 lokal	0.91	2.10	6.01	28.76	50.01
Typ4 remote	7.26	9.37	6.14	25.91	50.07
Typ5 lokal	0.84	1.77	5.05	24.49	50.01
Typ5 remote	8.63	10.48	5.28	21.93	50.38

Tabelle 8: Benchmark: Ping-Pong
[in Sekunden]

Knoten	UDP-Sockets	TCP-Sockets	SunRPC	CORBA	XML-RPC
Typ1 lokal	4.32	10.56	23.22	84.66	98.88
Typ1 remote	16.93	25.22	20.25	87.14	101.87
Typ2 lokal	0.80	2.29	7.27	31.09	50.00
Typ2 remote	10.29	14.99	8.79	30.96	33.99
Typ3 lokal	0.77	2.27	7.03	30.02	50.01
Typ3 remote	3.88	8.24	6.96	27.81	50.01
Typ4 lokal	0.90	2.38	6.76	29.29	50.00
Typ4 remote	7.32	10.47	6.97	26.47	50.39
Typ5 lokal	0.85	2.00	5.70	24.90	50.00
Typ5 remote	8.63	10.06	5.98	22.43	50.94

Tabelle 9: Benchmark: Hello World
[in Sekunden]

B. Messwerte

Knoten	SunRPC	CORBA	XML-RPC
Typ1 lokal	27.31	92.89	100.37
Typ1 remote	25.18	100.40	101.87
Typ2 lokal	8.60	34.04	50.00
Typ2 remote	10.16	34.01	49.98
Typ3 lokal	8.19	32.80	50.02
Typ3 remote	7.99	30.62	50.02
Typ4 lokal	7.91	32.15	50.02
Typ4 remote	8.01	29.00	50.07
Typ5 lokal	6.65	27.42	50.00
Typ5 remote	7.46	24.61	50.98

Tabelle 10: Benchmark: Struktur
[in Sekunden]

Knoten	SunRPC	CORBA	XML-RPC
Typ1 lokal	261.36	228.08	277.12
Typ1 remote	309.92	215.35	314.57
Typ2 lokal	127.44	87.75	104.18
Typ2 remote	141.55	90.45	150.44
Typ3 lokal	119.87	84.11	99.20
Typ3 remote	120.51	82.07	102.01
Typ4 lokal	114.62	81.34	90.81
Typ4 remote	115.72	78.31	100.27
Typ5 lokal	96.24	68.67	80.43
Typ5 remote	99.04	66.03	83.75

Tabelle 11: Benchmark: Liste von Integer
[in Sekunden]

Knoten	SunRPC	CORBA	XML-RPC
Typ1 lokal	558.77	824.68	1993.22
Typ1 remote	656.16	836.27	2283.69
Typ2 lokal	267.12	336.07	2488.71
Typ2 remote	302.51	369.96	1077.30
Typ3 lokal	251.78	323.71	770.56
Typ3 remote	255.46	337.65	800.21
Typ4 lokal	237.14	306.05	690.83
Typ4 remote	240.31	303.46	734.72
Typ5 lokal	210.84	257.28	584.47
Typ5 remote	206.43	259.37	628.37

Tabelle 12: Benchmark: Liste von Strukturen
[in Sekunden]

B. Messwerte

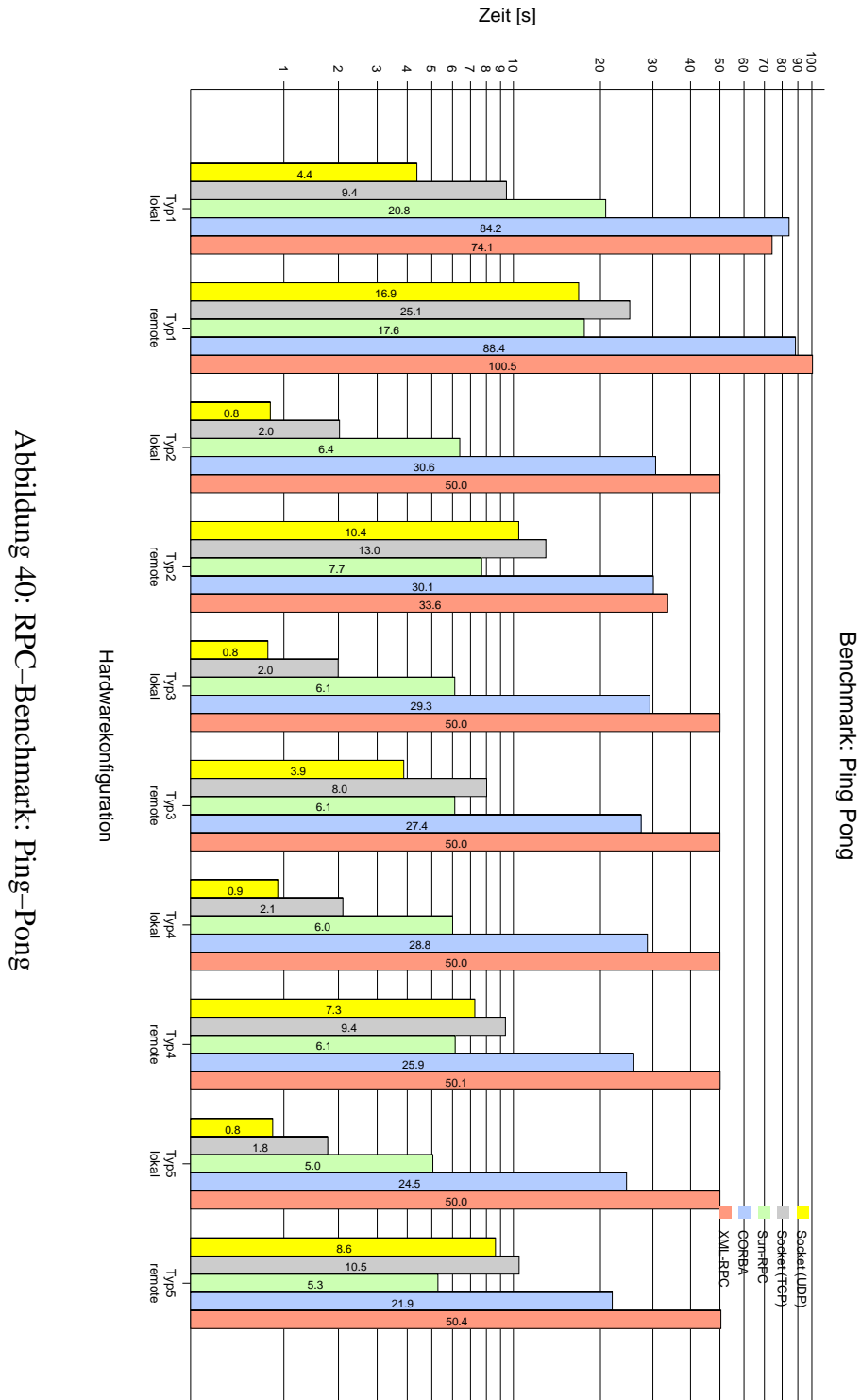


Abbildung 40: RPC-Benchmark: Ping-Pong

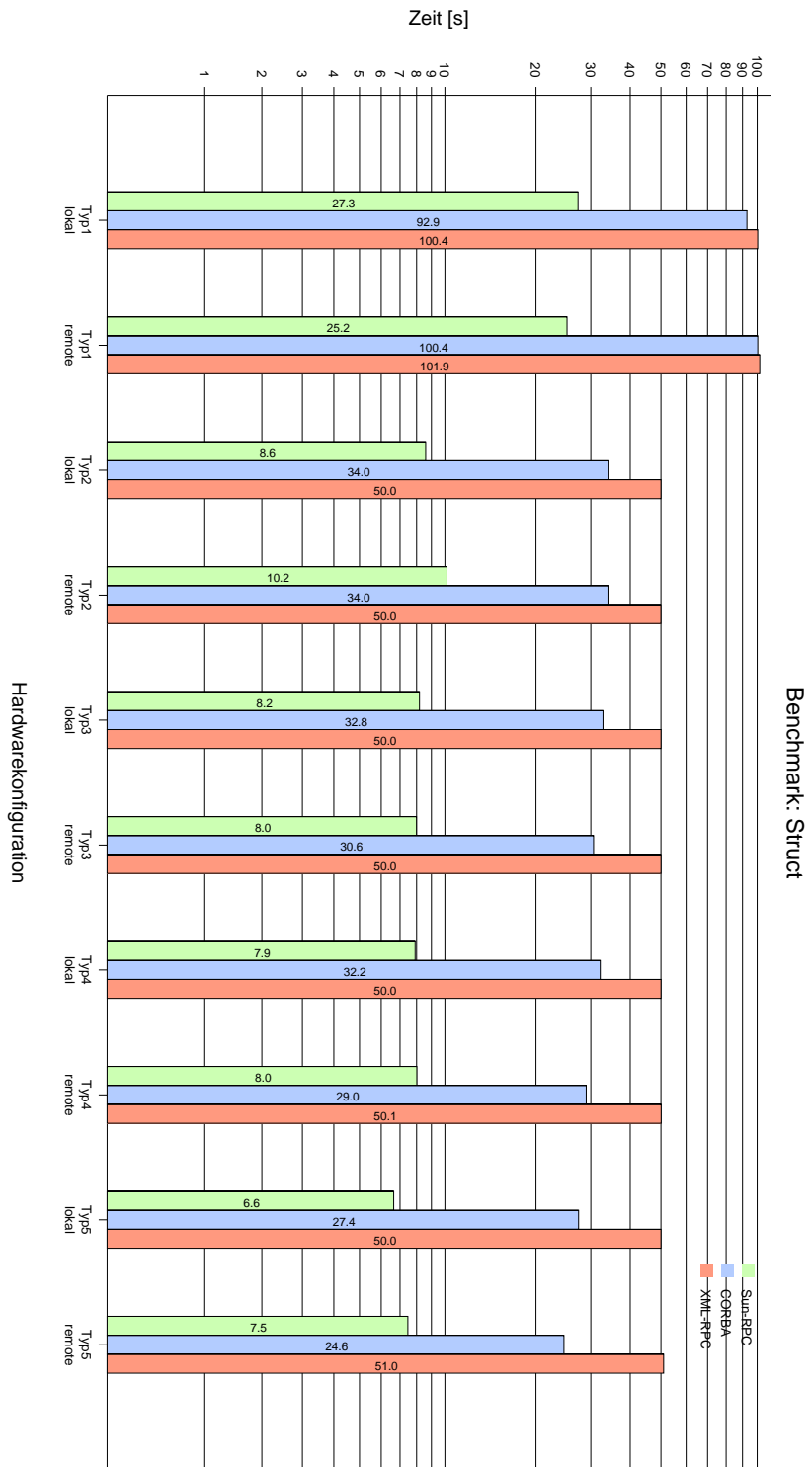


Abbildung 41: RPC-Benchmark: Strukturen

B. Messwerte

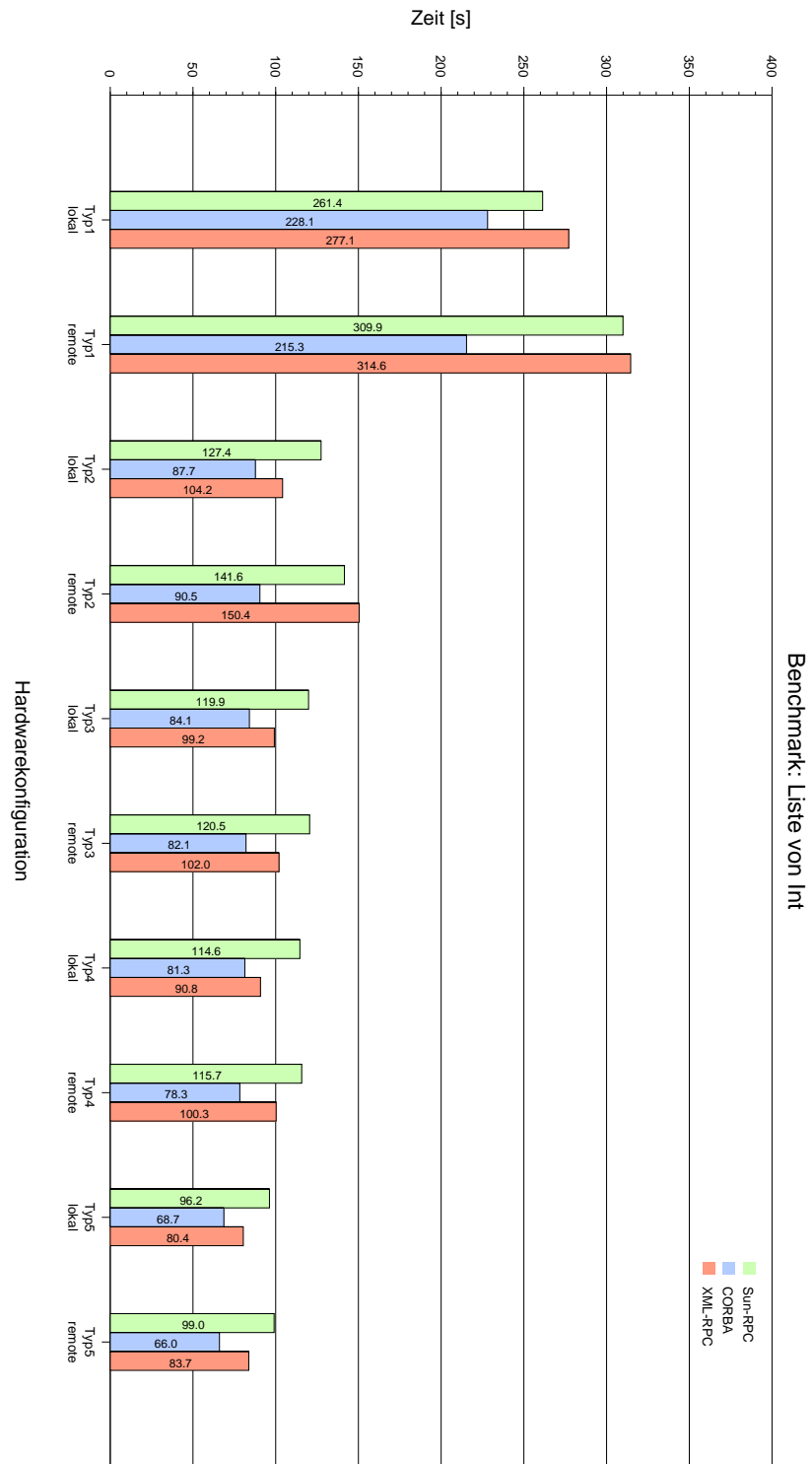


Abbildung 42: RPC-Benchmark: Liste von Integer

Literatur

- [1] LIBERO.
<http://www.libero.de/> pages 1
- [2] Javasoft.
<http://java.sun.com/> pages 21
- [3] PHP Hypertext Processor.
<http://www.php.net> pages 21
- [4] Python Language.
<http://www.python.org/> pages 21
- [5] Fnorb.
<http://www.fnorb.com/> pages 26
- [6] The AFS File System.
<http://www.transarc.ibm.com/Product/EFS/AFS/afsoverview.html> pages 39
- [7] Extreme Programming: A gentle introduction.
<http://www.extremeprogramming.org/> pages 39
- [8] SQL.
<http://www.sql.org/> pages 46
- [9] ReiserFS.
<http://www.namesys.com/> pages 58
- [10] The Apache Software Foundation.
<http://www.apache.org/> pages 73
- [11] Mod Snake.
<http://modsnake.sourceforge.net/> pages 73
- [12] Mod Python.
<http://www.modpython.org/> pages 73
- [13] PyApache.
<http://bel-epa.com/pyapache/> pages 74
- [14] The OpenSSL Project, 1999.
<http://www.openssl.org> pages 72

Literatur

- [15] XSL Transformations, 1999.
<http://www.w3.org/TR/xslt> pages 77
- [16] M2Crypto = Python + OpenSSL + SWIG, 2001.
<http://www.post1.com/home/ngps/m2/> pages 72
- [17] Don Box. Simple Object Access Protocol (SOAP) 1.1, 2000.
<http://www.w3.org/TR/SOAP/> pages 16
- [18] Dan Cerutti, Herausgeber. *Distributed Computing Environments*. McGraw-Hill, Inc., New York, 1993. ISBN 0-07-010516-2. pages 3
- [19] Don Chamberlin. XQuery 1.0: An XML Query Language, 2001.
<http://www.w3.org/TR/xquery/> pages 46
- [20] A. Chiu. Authentication Mechanisms for ONC RPC, 1999.
<http://www.ietf.org/rfc/rfc2695.txt> pages 12
- [21] Netscape Communications Corporation. Introduction to SSL, 1998.
<http://developer.netscape.com/docs/manuals/security/sslin/contents.htm> pages 11
- [22] Jon Crowcroft. *Open Distributed Systems*. UCL Press Limited, London, 1996. ISBN 1-85628-229-9. pages 13
- [23] Expat XML Parser.
<http://expat.sourceforge.net/> pages 32
- [24] Rainer Gallersdörfer. *Replikationsmanagement in verteilten Informationssystemen*. 1997. ISBN 3-89601-424-1. pages 54, 55, 60
- [25] Object Management Group. General Inter-ORB Protocol, 1995.
<http://www.infosys.tuwien.ac.at/Research/Corba/OMG/giop.htm> pages 13
- [26] Abdelsalam A. Helal. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1996. ISBN 0-7923-9800-9. pages 77
- [27] Matthias Käding. *Sicherheitsarchitektur für Verteilte Systeme*. 1991. pages 78
- [28] Petr Kroha. *Softwaretechnologie*. Prentice-Hall, Haar b. München, 1997. pages 17

- [29] Ora Lassila. Resource Description Framework, 1999.
<http://www.w3.org/TR/REC-rdf-syntax/> pages 24
- [30] Eric Miller. Dublin Core Examples in RDF, 1998.
<http://www.dstc.edu.au/Research/Projects/rdf/dc-in-rdf-ex.html> pages 24
- [31] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
<http://www.dcc.uchile.cl/~gnavarro/ps/acmcs01.1.ps.gz> pages 76
- [32] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl. 2000.
http://www.ipd.ira.uka.de/~prechelt/Biblio/Biblio/jccpprt_computer2000.ps.gz pages 21
- [33] David Purdue. Transparency and Performance Issues in Sun RPC. 1995.
<http://www.auug.org.au/auugn/voll16nol/sunPaper.html> pages 29
- [34] secret labs. XML-RPC for Python, 2001.
<http://www.pythonware.com/products/xmlrpc/index.htm> pages 26
- [35] secret labs. The sgmlp Module, 2001.
<http://www.pythonware.com/products/xml/sgmlp.htm> pages 32
- [36] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2, 1995.
<http://www.ietf.org/rfc/rfc1831.txt> pages 11
- [37] W. Richard Stevens. *Unix Network Programming*, Band 1. Prentice–Hall, Inc., Englewood Cliffs, 1990. pages 10
- [38] J. Mogul J. Gettys R. Fielding T. Berners-Lee, H. Frystyk. Hypertext Transfer Protocol – HTTP/1.1, 1997.
<http://www.ietf.org/rfc/rfc2068.txt> pages 15, 78
- [39] R. Fielding T. Berners-Lee, H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0, 1996.
<http://www.ietf.org/rfc/rfc1945.txt> pages 78

Literatur

- [40] Inc. UserLand Software. XML-RPC Specification, 1999.
<http://www.xml-rpc.com/spec> pages 16
- [41] Michael Weber. *Verteilte Systeme*. Spektrum Akademischer Verlag GmbH, Heidelberg, 1998. ISBN 3-8274-0221-2. pages 8, 13, 14, 50
- [42] Thorsten Wittkugel. *Synchronisationsmechanismen in verteilten Objektsystemen*. 1994. pages 60, 77
- [43] Thomas Wolff. *Transparente Verteilung objektorientierter Programme*. VDI Verlag GmbH, Düsseldorf, 1996. ISBN 3-18-345710-5. pages 9
- [44] W. Yeong. X.500 Lightweight Directory Access Protocol, 1993.
<http://www.ietf.org/rfc/rfc1487.txt> pages 45

Selbständigkeitserklärung

-----BEGIN PGP SIGNED MESSAGE-----

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Chemnitz, 16. Oktober 2001

-----BEGIN PGP SIGNATURE-----

Version: 2.6.3in

Charset: noconv

iQCVAwUBPBOY+DrY3KOpLvExAQHVtgP+Lb0Wf80W4N+HlMphKlqUoKyUtN7v4+rI
2tUlv3glw4VZ2E/zEbSP8G5/lqx4axeH+VbTNYycAbeTl8P5GYw+X2LgDTVqqBcz
+mSSb2GQ43btQF11qC9Vbi/ooCx3YlpoEM9xe/iihCR+B/uQ4lyK4Jnv2WzfJWl
xKnsbdKTHfo=

=YEHp

-----END PGP SIGNATURE-----