Zusammenfassung „Middleware" SS2001

Michael Jaeger (michael.jaeger@in-flux.de)

29. Oktober 2001

Inhaltsverzeichnis

1 Motivation

Die Einführung moderner Netzwerktechnologien undderen zunehmende Popularität führte um 1980 zu einem neuen Konzept im Software-Enigineering – dem **Client-Server-Modell**. Davor gab es den **Batch-Betrieb**, das **Multiprogramming** und das **Time-Sharing**, die allesamt auf einem Rechner liefen.

1.1 Das Client-Server-Modell

Verteilungs-Konzept

Die Verarbeitung beim Client-Server-Modell ist klar in zwei Teile aufgeteilt: den Server und den Client. Dabei wird der Client meist zur **Darstellung** und der Server zu **Verarbeitung** der Daten eingesetzt. Allgemein ist das **Interaktionsmodell** zwischen Client und Server jedoch perspektivenabhängig, da je nach Infrastruktur und Anwendung der Client mehr oder weniger Arbeit an den Server abgeben kann.

Vorteile **Leistungsfähigkeit der Endgeräte** kann dynamisch berücksichtigt werden und zu einer besseren Lastverteilung führen, und die Anwendung **ausfallsicherer** (?) gemacht werden.

Nachteile Die **Software-Verteilung** kann zum Problem werden (Versionierung) – allgemein erhöht sich der Aufwand für das **Mangement** der Endgeräte (→TCO).

n-Tier-Modelle

Bei der Verteilung der Funktionalität auf Client und Server werdene sog. **n-Tier-Modelle** unterschieden. Das n steht für die Anzahl der logischen (und physischen) Ebenen, die bei der Verteilung der Anwendung auf verschiedene Rechner unterschieden werden. Populär sind 2-Tier-Modelle (z.B. bei X, WWW) und 3-Tier-Modelle (z.B. bei SAP R/3). Eine geläufige Aufteilung ist die nach **User-Interface** (**Präsentation**: GUI, HTML), **Funktion** (**Applikation**: Application-Server, Workflow) und **Daten** (**Datenbank**: SQL).

Thin-Client-Modell

Die Idee hinter dem **Thin-Client-Modell** ist die Verlagerung von Funktionalität weg vom Client (Darstellungs-Ebene) hin zum Server, was zu einer Senkung der **total cost of ownership** (TCO) führen sollte. Mit Unterstützung moderner Middleware wurde es möglich abhängig von den Anforderungen auf Client-Seite verschiedene Zugänge zum System zu schaffen. So kann ein Zugang z.B. über das WWW, was client-seitig nur wenig Voraussetzungen erfordert, oder über eine komplexe GUI erfolgen, was höhere Anforderungen an der Client stellt, dafür aber auch mehr Möglichkeiten in der Präsentation bietet.

Probleme des Client-Server-Wildwuchs

Im Laufe der Zeit entwickelten sich so viele Client-Server-Anwendungen, die allesamt auf ein Repertoire an Daten zugriffen und wiederum untereinander in einer Art und Weise verzahnt waren, die nicht standardisiert und deshalb schwer zu überblicken ist. Das führte zu ernomen Kraft-Anstrengungen, um diese Systeme an neue Technologien wie das WWW oder **ERP**-Programme (Enterprise Ressource Planning) anzubinden, da im Prinzip für jede Anwendung und jede Datenquelle individuell eine Schnittstelle entwickelt werden musste.

Enterprise Application Integration

Durch die Einführung von **Middleware** erhofft man sich einen **einheitlichen Zugriff** auf die Daten und eine **enterprice application integration**, durch die auch „Altlasten" in moderne Infrastrukturen eingebettet werden können. Eine Integration von Anwendungen und Daten soll durch ein **einheitliches Programmiermodell** (→**Komponenten**) und die **Interoperabilität** und den **einheitlichen Datenzugriff** (→Middleware) gewährleistet werden.

Komlexität

Der Einsatz verteiler Systeme führt einen neuen Grad an **Komplexität** mit sich, der sich aus folgenden Tatsachen ergibt:

- **Echte Parallelität der Verarbeitung** kann zu partiellen Ausfällen von Komponenten führen.
- Die **explizit asynchrone, fehlerbehaftete Kommunikation** verhindert die Feststellung eines eindeutigen globalen Zustands.
- **Unabhängige Systemuhren** führen zu dem Fehlen einer eindeutig globalen Zeit.
- **Replizierte Daten** bringen Probleme mit der Datenkonsistenz mit sich.
- Die neue Vielfalt von Angriffspunkten bringen **Sicherheitslücken** mit sich.
- **Autonome Subsysteme** führen zu einem neuen Grad an Komplexität des Netz-, System- und Anwendungsmanagement.

Heterogenität/Interoperabilität

Die **Heterogenität** bezüglich **Netzwerkstandards** (Ethernet, ATM etc.), **Programmiersprachen** (C, Java etc.), **Protokolle** (FTP, HTTP etc.), **Betriebssysteme** (Windows, Unix etc.) und **Mobilität/Kapazität** (Laptop, PDA, Desktop etc.) führt zu einer kaum verwaltbaren Komplexität. Der Ausweg aus dieser Misere ist die **Interoperabilität**, die keine einheitliche Hardware, Betriebssysteme, Programmiersprache oder Kommunikationsprotokolle voraussetzt. Erreicht werden soll diese Unabhängigkeit durch den Einsatz von **Verteilungsplattformen für verteilte Anwendungen** und eine **Transparenz der Verteilung und der Heterogenität**.

Middleware

Wie immer gibt es verschiedene Definitionen, was Middleware ist und leisten soll:

Middleware is the intersection of the stuff that network engineers don't want to do with the stuff that applications developers don't want to do.

Middleware today encompasses everything to the right of the clients and to the left of legacy systems. It resides above the OS and below the application logic.

Middleware is the slash in the client/server.

Middleware ...some think of the Web as the ultimate middleware.

Middlware is connecting systems.

Bei der **Kommunikation** setzt Middleware auf bereits vorhandenen Transportsystemen auf und ist unabhängig von Transportprotokollen. Damit residiert es im ISO/OSI-Modell in Schicht 5 und 6. Sie verbindet **lose gekoppelte, autonome Systeme** und ist meist als **Zusatz zum Träger-Betriebssystem** realisiert.

Zur Abgrenzung unterschieden wir zwischen

Verteilten Betriebssystemen die ein einheitliches Betriebssystem, keinen gemeinsamen Speicher und eine enge Kopplung besitzen. Man spricht auch häufig von einem **single system image** (Middleware stellt soetwas nicht zur Verfügung).

Multiprozessorsystemen die ein zentrales Betriebssystem, gemeinsamen Speicher und eine enge Kopplung besitzen.

Rechnernetzen die Kommunikationsprotokolle besitzen, jedoch über keine Middleware verfügen.

Transparenz

Die durch die Verteilung des Systems entstehende Komplexität versucht man mit Hilfe von **Transparenz** vor dem Benutzer/Programmierer zu verbergen, um so eine vereinfachte Programmierung und Handhabung zu gewährleisten. Die Transparenz betrifft dabei folgende Parameter beim Umgang mit Ressourcen:

- **Ort** (geographische Lokation)
- **Replikation** (von Ressourcen)
- **Migration** (Verlagerung von Ressourcen)
- **Fehler** (Fehlverhalten von Ressourcen)
- **Nebenläufigkeit** (beim gemeinsamen Zugriff auf Ressourcen)
- **Performance** (Unterschiede bei Antwortzeiten zwischen lokal/entfernt)

ODP

Zur Standardisierung der Transparenz wurde von der **ISO** der Standard **ODP** (**Open Distributed Processing**) entworfen, um die grundlegenen Begriffe für die verteilte Verarbeitung zu standardisieren. Die Thematik erwies sich aber als zu komplex, als dass ein Standard alle Facetten gleichzeitig erfassen könnte, so dass nur einzelne Aspekte der Transparent bis heute überlebt haben, wie z.B. der **Trading-Standard** zur Dienstvermittlung.

Transparenz heute

Heutzutage ist in der Regel **Ortstransparenz** in die meiste Middleware integriert und **Migrationstransparenz** wird durch **Indirektionskomponenten** realisiert. Diese neuen Formen der Transparenz bringen aber auch neue Probleme bei **Management**, **Fehlersuche**, **lokalisierten Diensten** und **QoS** mit sich.

Middleware-Arten

Wir unterscheiden im Folgenden zwischen zwei **Middleware-Arten** nämlich der **generischen**:

- **Object Request Borker**, die einen Objektzugriff über das Netz ermöglichen
- **Remote Procedure Calls**, die den Aufruf einer entfernten Prozedur ermöglichen

- **Message Passing**, die eine Send/Receive-Kommunikation ermöglichen
- **Virtual Shared Memory**, das den Zugriff auf einen virtuell gemeinsamen Speichert realisiert
- **Mobile Agenten**.

und der **speziellen**:

- **Dateitransfer** ermöglicht den Fernzugriff auf entfernte Dateien
- **Datenbankzugriff** ermöglicht den Datenzugriff auf entfernte Datenbanken
- **Transaktionsverarbeitung** sorgt für verteilte Transaktionen
- **Groupware** koordiniert die Zusammenarbeit in Gruppen
- **Workflow** organisiert arbeitsteilige Prozesse.

1.2 Middleware-Spezifika

Objektmodell

Das **Objektmodell** modelliert die Welt als Menge interagierender Objekte, wobei ein Objekt **Zustand** (Attribute), **Verhalten** (Reaktionen auf Methodenaufrufe) und **Identität** (Identifier/Adresse) besitzt. **Schnittstellen** (Interfaces) definieren dabei die von außen zugreifbaren Methoden und Attribute, verbergen die Implementierung und stellen einen **Vertrag** zwischen Nutzer und Anbieter dar. An dieser Stelle sei an die **drei Grundelemente der Objektorientierung** erinnert:

- **Kapselung** trennt Schnittstelle und Implementation → Robustheit
- **Vererbung** gewährleistet Wiederwendung von Spezifikation und Code → strukturelle Klarheit
- **Polymorphie** ermöglicht Erweiterbarkeit → Abstraktion

Verteilte Objekte

Die Objektmodellierung impliziert eine mögliche Verteilung. **Objekt-Middleware** ermöglicht den entfernten Methodenaufruf, dabei soll eine **interface definition language** (**IDL**) für eine Unabhängigkeit bzgl. der Programmiersprachen sorgen. Der entfernte Zugriff wird durch **Stubs** und **Proxies** realisiert, die Vertreterobjekte für den entfernten Objektzugriff sind.

RPC

Der Remote Procedure Call ermöglicht einen entfernten Prozeduraufruf transparent für den Programmierer. Dabei wird die notwendige Kommunikation durch die RPC-Middleware versteckt, indem während des Übersetzungsvorgangs ein **Network-Linker** zur Bindung eingesetzt wird, der Client- und Server-seitig einen Stub generiert, über den die Kommunikation läuft.

Message Passing

Das Message Passing erweitert das Programiermodell um die Funktionen `send(message)` und `receive(puffer)`, mit denen Nachrichten der Form [Absender|Empfänger|Typ|Daten] kommuniziert werden können. Seine Urform hat das Message Passing in den **UNIX Sockets**, die ein Grundschema der Form **Request/Reply** ermöglicht – andere Interaktionsmuster sind aber auch möglich.

Gemeinsamer Speicher

In nicht-verteilten Systemen wird **shared memory** häufig für Interprozess-Kommunikation eingesetzt, die jedoch Synchronisationsmechanismen erfordert. In einem verteilten System existiert kein gemeinsamer Adressraum, so dass eine explizite Kommunikation über das Rechnernetz stattfinden muss. Ziel ist die Schaffung der Illusion eines **virtual shared Memory** (also eine Form der Orts-Transparenz), auf die Prozesse mit Bibliotheksfunktionen zum Lesen und Schreiben zugreifen können. Ein Beispiel für eine Realisierungen ist **Linda**, ein System, bei dem Prozesse auf einen gemeinsamen **Tupelraum** lesend und schreibend zugreifen können, indem sie die Funktionen `read()`, `out()` und `in()` verwenden. Tupel sind dabei **Werte-Vektoren** $(N, P_1, \ldots, P_k)$, wobei N der **Aktualparameter** ist, der den Typ/Namen des Tupels festlegt und die P_i aktuelle oder formale Paramter sind. Ein Tupel a passt genau dann zu einem gegebenen Tupel b, wenn es

- identischen Namen
- gleiche Tupel-Länge
- Aktualparameter, die positionsweise identisch sind
- Formal- und Aktualparameter, die positionsweise kompatibel sind

besitzt. Diese Idee wurde von **JavaSapces** erneut aufgegriffe und wird z.B. als Grundlage für **Jini** eingesetzt. Es bietet eine Plattform zum Austausch von Objekten zwischen verteilten Anwendungen und enthält (getypte) Objekte, die über spezielle **Matching-Regeln** identifiziert werden können.

Standardisierung

Bei Standardisierungsgremien unterscheidet man grob zwischen Industrie- und Offiziellen-Standards.

(Offizielle) Internationale Standardisierung ISO, ITU, DIN, ANSI

Semi-offizielle internationale Standardisierung OpenGroup (X/Open bzw. OSF), IEEE, IETF, W3C

Konsortien OSF, OMG

Industrie-Standards

2 Message Queuing

Eine Middleware-Technik, die auf dem Austausch von Nachrichten (request/reply) beruht, ist das **Message Queueing**, das bereits in einer Vielzahl von Produkten eingesetzt wird (z.B. bei MQSeries von IBM). Dabei findet eine **gepufferte Nachrichtenübertragung** statt, die **asynchron** und **verteilungstransparent** ist und sich in eine beliebige Anwendungsstruktur einpassen lässt.

2.1 MQSeries

Das Produkt **MQSeries** unterstützt eine Reihe von Programmiersprachen und Plattformen. Es werden vier verschiedene **Nachrichtenarten** unterschieden: **Auftrag**, **Antwort**, **Unidirektional** und **Report**. Die Anwendungen greifen über das **Message Queueing Interface** (MQI) auf den **Queue Manager** zu, der die Queues verwaltet, die jeweils von **Message Channel Agents** gefüllt bzw. geleert werden.

Bei dem Aufruf von `GetMessage` sind Selektionsparameter möglich und Nachrichten können gruppiert werden, um eine bestimmte **Reihenfolge** zu garantieren. Ausserdem ist es möglich, Nachrichten mit einem Verfallsdatum zu versehen oder diese automatisch zu segmentieren. Ebenso können sie an eine **Verteilerliste** verschickt werden ($1 : n$-Kommunikation). Die **Sicherheit** des Systems basiert auf

- **Authentisierung** zwischen Benutzer und Queue Manager, sowie zwischen MCA und MCA
- **Verschlüsselung** der Nachrichten
- **Message Context** mit Angabe zu Benutzer etc.

Eine Nachricht enthält die folgenden **Deskriptorinformationen**:

- Message ID, Reply to Queue, Verfallsdatum, Formatspezifikation, Sequence-Number.

Die entkoppelte Kommunikation ermöglicht auch einen **Multicast**, also eine $m : n$-Kommunikation. Da der **Abstraktionsgrad sehr niedrig** ist, muss sich der Programmierer selbst um auftretende Fehler kümmern. Dafür kann der Empfänger ausfallen und trotzdem Nachrichten an ihn gesendet werden, ohne dass diese verloren gehen.

Beispiel Reservierungssystem für Reisebuchungen.

Ebenso kann MQSeries als **Resource Manager** für **Transaktionen** eingesetzt werden, so dass Transaktionen über einen **Transaktionsmonitor** gesteuert werden, der auf den Resource Manager (MQSeries, DBMS, Dateien) Zugriff hat. Dabei wird das **2-Phase-Commit-Protokoll** zur Sicherstellung der Transaktion verwendet.

2.2 TIB

Ein anderes Produkt von TIBCO Software Inc. ist **The Information Bus** (TIB). Genau wie MQSeries werden viele Programmiersprachen und Plattformen unterstützt. Es ist sehr stark bei Anwendungen im Finanzsektor vertreten und unterstützt drei **Interaktionsmuster**: **Publish/Subscribe**, **Request/Reply** und **Broadcast Request/Reply**.

Interaktionsmuster

Das Interaktionsmuster Request/Reply wird genauso verwendet wie bei MQSeries und kann in herkömmlichen Client/Server-Anwendungen eingesetzt werden. Beim Publish/Subscribe wir eine $1 : n$-Kommunikation ermöglicht, die mit einer wachsenden Anzahl von Clients besser skaliert als herkömmliche Client/Server-Lösungen. Ausserdem können Anfragen an alle gesendet werden, während nur eine Antwort gewünscht ist, was für **Fehlertoleranz** und **Lastverteilung** eingesetzt werden kann. Antworten werden in diesem Fall mit **Message-Passing** und nicht mit RPC realisiert, was zu einer **Asynchronität** führt.

Subjekt-basierte Adressierung

Bei der **Subjekt-basierten Adressierung** werden die Empfänger von Nachrichten durch Nachrichtenmerkmale idenitifiziert, z.B. anhand eines „Gesprächsthema" oder einer Zeichenkette (inkl. Wildcards). Damit ein Prozess Nachrichten eines bestimmten Subjekt-Typs empfangen kann, muss er sich dafür beim Absender registrieren. Schickt der Absender dann eine Nachricht eines Subjekts-Typs, so erhalten alle registrierten Prozesse diese Nachricht und zwar tranparent bzgl. Ort, Migration und Replikation. Der Empfänger wird mit einer sog. **Callback Function** mit einem Subjekt assoziiert. Auch bei der Subjekt-basierten Kommunikation werden alle drei oben genannten Interaktionsmuster unterstützt.

Systemarchitektur

Bei der Programmierung von TIB-Anwendungen wird über die **TIB-API** auf die **TIB-Library** zugegriffen, die mit dem **TIB-Daemon** kommuniziert. Die TIB-Daemons kommunizieren über einen **Software-Bus** miteinander und können auf diese Art und Weise z.B. auf gemeinsame Resourcen zugreifen.

Beispiel Kunde-Bank-Konto:

Kunde		Bank
`QAntwort` erzeugen		`QSchalter` erzeugen
`receive(QAntwort)`	Neues Konto $\longrightarrow$ `QSchalter`	
		Nachricht bearbeiten
	`QAntwort` Antwort $\longleftarrow$	

Zusammenfassend kann gesagt werden, dass Message-Queueing zu den beliebtesten Middleware-Gattungen überhaupt gehört, da es ein **einfaches Programiermodell**, **Plattform- und Programmiersprachen-Unabhängigkeit** und eine **Integration mit anderen Systemen** bietet.

Dabei muss man sich aber immer darüber im Klaren sein, dass es sich hier nicht um eine komplette Middleware handelt und man sich auf einer sehr niedrigen Abstraktionsstufe bewegt. Dabei erfährt man softwarerechnisch kaum eine Unterstützung.

3 DCE

Das **Distributed Computing Environment** (DCE) wurde von der OSF (heute OpenGroup) ins Leben gerufen und war als Standard-Modell für verteilte Systeme angelegt. Dabei sollte die Kommunikation, die hinter jedem entfernten Prozedur-Aufruf steckt, möglichst versteckt werden, was mit der Einführung des **Remote Procedure Calls** (RPC) erreicht werden sollte. DCE stellt die erste Middleware-Plattform dar und teilweise auch heute noch in Microsoft's **DCOM** weiter. **Kerberos** als Teil von DCE entwickelt und erfreut sich heute, genauso wie das **Andrew Filesystem**, unabhängig von DCE großer Beliebtheit.

3.1 RPC

IDL

Das Design-Ziel hinter dem RPC war, dass Client und Server heterogen sein sollten. Dies realisierte man mit einer speziellen **Interface Definition Language** (IDL), mit Hilfe derer eine automatische **Stub**-Generierung durch den IDL-Compiler möglich wurde. Ein **einheitliches Datenaustauschformat** sollte durch eine Konvertierung der Datentypen in ein **kanonisches Austauschformat** erreicht werden.

Programmierung

Bei der Programmierung von Anwendungen, die RPC nutzen, müssen zunächst die Schnittstellen beschrieben werden. Diese werden dann mit dem IDL-Compiler in Stubs übersetzt, die Client und Server dann beim Link-Vorgang einbinden.

Fehlersematik

Bei RPC-Aufrufen werden vier unterschiedliche **Fehlersemantiken** unterschieden:

- **exactly once** (ein Aufruf wird genau einmal durchgführt)
- **at least once** (ein Aufruf wird mindestens einmal durchgeführt)
- **at most once** (ein Aufruf wird höchstens einmal ausgeführt)
- **maybe** (ein Ausruf wird evt. mehrfach und ohne Rückgabewert ausgeführt).

Beim Arbeiten mit RPC muss der Programmierer umdenken, da anders als beim lokalen Prozeduraufruf keine Pointer übergeben werden dürfen, keine Annahmen über Zeit oder Dauer gemacht werden dürfen (→**Race-Conditions**) und das Testen der Anwendungen sehr schwer ist.

3.2 DCE-IDL

Die IDL ist als Programmiersprache sehr stark an C angelehnt, besitzt als Besonderheit jedoch 16-Bit-lange **Unique Universal Identifier** (**UUID**), die einem Objekt, einer Schnittstelle oder einer Operation eine eindeutige ID zuweisen und per Aufruf vom System erzeugt werden, und sog. **Pipes**, die zur Übertragung großer Datenmengen (analog zu der aus UNIX bekannten `pipe`) eingesetzt werden. In der DCE IDL unterscheidet man zwischen `[in] pipe` und `[out] pipe`, wobei der Anwendungsprogrammierer die dazu definierten zurgehörigen `push()` und `pull()` Funktionen implementieren muss.

Beispiel Kunde-Bank-Konto

Client	Server

	Das Bank-Interface stellt folgende Funktionen zur Verfügung:
	`Konto.anlegen([in] Kunde, [out] Kto)`
	`einzahlen([in] Kto, [in] Betrag)`
	`abfragen([in] Kto)`

Client muss sich an den Server binden, was bereits im Code geschieht: `Handle=Bind('Bank25')`	
Jetzt wird über das Handle ein Prozeduraufruf auf dem Server durchgeführt: `Kto=anlegen(Handle, Name)`	
...und eine Einzahlung kann vorgenommen werden: `einzahlen(Handle, Kto, Betrag)`	

Problematisch bei diesem Beispiel ist, dass ein Handle für die Prozeduraufrufe benötigt wird.

3.3 Datenrepräsentation

Die IDL beschreibt lediglich die **(abstrakte) syntaktische Struktur** einer Server-Schnittstelle. Die Heterogenität des Systems erfordert zusätzlich noch ein **kon-**

kretes kanonisches Austauschformat, in dem Daten übertragen werden können. Damit sollen Probmene in der Datenrepräsentation vermieden werden, wie sie z.B. bei der unteschiedlichen Kodierung von Integer-Zahlen auf verschiedenen Rechnerarchitekturen entstehen können (**Little Endian** vs. **Big Endian**). Beispiele für Standards zur Datenrepräsentation sind OSI (→ASN.1), DCE (→**NDR**, **Network Data Representation**) und SUN RPC (→XDR). Bei der **XDR** werden Integer-Zahlen z.B. einheitlich im Big Endian-Format mit der 2er-Komplement-Darstellung und Gleitkommazahlen im IEEE-Standard (Vorzeichen, Exponent, Mantisse) repräsentiert.

3.4 Threads

Leichtgewichtige Prozesse werden als **Thread** bezeichnet und stellen eine logische eigenständige Aktivität innerhalb eines Prozesses dar. Im Gegensatz zu Prozessen besitzen sie einen gemeinsamen Speicher innerhalb des Prozesses und sind bzgl. der Verwaltung und der Umschaltung effizienter als Prozesse. Sie werden häufig zur Programmierung nebenläufiger Aktivitäten eingesetzt.

POSIX Thread-API

Für die Vewaltung von Threads gibt es eine nach POSIX 1003.4a genormte Schnittstelle, die

- Erzeugung und Zerstörung von Threads
- Nachrichtenaustausch zwischen Threads
- Synchronisation zwischen Threads
- Verschiedene Möglichkeiten des Schedulings

definiert.

3.5 Bindungsvorgang

Für die Bindung von Client und Server ist eine Festlegung der **vom Server verwendeten Kommunikationsprotokolle** und eine **lokale Registrierung der Prozedur** Voraussetzung. Die **Prozedurschnittstellen** werden zum Verzeichnis exportiert und danach initialisiert, woraufhin sie auf Aufrufe warten. Der Client bindet sich mittels der Bindekennung im **Binding Handle** an den Server und ruft danach Prozeduren auf.

Bindungsarten

Es werden die **Bindungsarten** „**explizit**" (der Client bestimmt den Server und gibt explizit eine Bindekennung an), „**implizit**" (der Client wählt einen Server im Verzeichnis aus und legt das **Handle** in eine Variable ab) und „**automatisch**" (das Verzeichnis wählt automatisch einen passenden Server aus) unterschieden.

3.6 Verzeichnisdienste

Allgemein

Für eine flexiblerere Bindung an Server-Dienste wird wie oben schon erwähnt ein sog. **Verzeichnis** benötigt, dass **Objekte verwaltet**, die Namen und evtl. Attribute besitzen. Die **Namen** sind dabei nur in dem zugehörigen Kontext gültig und in der Regel unstrukturiert oder strukturiert. Es werden **flache und hierarchische Namensräume** unterschieden.

Verzeichnisse in DCE

Unter einem **Directory Service** oder auch **Name Service** verstehen wir eine (verteilte) Datenbank, die zur Speicherung von „Objekt"-Informationen dient. Das Erstellen eines Eintrags wird durch einen **Export** und das Lesen eines Eintrags duch einen **Import** realisiert. Es können auch **Objektgruppen** und **Aliase** verwaltet werden.

In DCE werden zwei Arten von Verzeichnissen unterschieden:

- Jede Zelle besitzt ein **local directory** (der sog. **cell directory service**), das Zellen zu organisatorischen Einheiten zusammenfasst.
- Das **global directory** unterstützt zwei Verzeichnisstandards: OSI X.500 und DNS.

X.500

Die **zell-lokalen Namen** eines Objektes setzen sich aus einem Pfad, beginnend mit der sog. **cell root** zusammen, z.B. `/.:/fs/hugo/focds/report.txt`.

DNS

Als verteiltes System zur Abbildung von Namen auf IP-Adressen, definiert der **DNS** eine hierarchische Syntax der Namen und ein hierarchisches Domänen-Konzept, mit dem beliebige Objekttypen benannt werden können. Der Einsatz von **miteinander kooperienden Name Servern** führt durch **Replikation** zu einer Steigerung der Zuverlässigkeit. Anfragen von Clients erfolgen mittels UDP und jeder Name Server kennt mindestens einen Vorgänger-Server. Das **Cachen** von Namensinformationen erhöht die Performanz, kann aber zu veralteten Informationen führen, was durch Angabe einer **time to live** verhindert werden soll.

X.500

Als OSI-Standard von der ISO/ITU-T definiert, stellt **X.500** ein **hierarchischen Domänenkonzept mit attributierter Namenssyntax** dar. Es dient der verteilten Speicherung von Informationen über „Objekte". Objekte können also mittels eines Pfades oder auch durch Angabe von **Datenattributen** angegeben werden. Die Infrastruktur bei X.500 besteht aus **DUA**s (**directory user agents**) und **DSA**s (**directory service agents**), die sich in einer **directory management domain** befinden. Das Directory speichert Informationen beliebiger Objekte, die wiederum selbst Instanzen von Objektklassen sind, die Attribute definieren. Ein **Attribut** ist ein Tupel, bestehend aus dem **Attributtyp** und der **Syntax**. Die **Objektklasse** legt für ein Objekt den Attributtyp und die Syntax in ASN.1 fest. Unter X.500 kann ein globaler Name, beginnend bei „root", folgendermassen aussehen: `.../c=de/o=danet/ou=abc/fs/hugo/docs/report.txt`.

Unter einem **distinguished name** (DN) wird in X.500 eine Attribut-Wert-Zuweisung verstanden (**attribute value assertion**, AVA), die eine Annahme über den Wert eines Attributs beschreibt. Objekte haben dabei herausgehobenen Wert (**distinguished value**) zur Namensbildung. der DN wird dann mit einer Folge von AVAs beschrieben, wobei **relative distinguished name**s (RDN) eine Teilfolge des DN beschreiben.

LDAP

Ein offener Standard für einen Verzeichnisdienst stellt **LDAP** (lightweight directory access protocol) dar. Es besteht im groben aus einer „schlanken" X.500-Implementierung, die den Zugriff auf ein X.500-Verzeichnis über TCP/IP ermöglicht, wobei Dienstprimitive auf TCP/IP-Nachrichten abgebildet werden. Das Informations- und Organisationsmodell ist aber identisch mit leicht reduzierter Funktionalität.

3.7 Sicherheit

Sicherheit

Allgemeine Aufgaben von Netzwerksicherheit sind

- Autorisierung
- Authentisierung
- Vertraulichkeit
- Integrität
- Nachweisbarkeit (Revisionssicherheit),

für deren Realisierung Kryptographische Verfahren und Sicherungsprotokolle eingesetzt werden.

DCE-Sicherheit

Sicherheit ist in DCE ein integraler Bestandteil des Systems. Es werden geboten:

- **Vertraulichkeit und Integrität** durch verschlüsselten RPC
- **Authentisierung** durch Einsatz von Kerberos
- **Autorisierung** durch Einsatz von **Zugriffszertifikaten** (privilege attribute certifictes, PAC) und **Zugriffskontrolllisten** (access control list, ACL)

Verschiedene **Schutzklassen** unterscheiden eine Authentisierung einmal zu Beginn, bei jedem Aufruf, bei jedem Übertragungspaket und bieten einen Schutz gegeben die Modifikation von Nachrichten durch eine vollständige Verschlüsselung. Die Schutzklassen sind dabei per API wählbar. Als Verschlüsselungsverfahren werden in DCE symmetrische Verschlüsselung (de facto DES) eingesetzt.

3.7.1 Kerberos

Kerberos wurde am MIT Mitte der 80er Jahre entwickelt. Die heute aktuelle Version trägt die Nummer 5. Kerberos stellt die Basis für den OSF DCE Sicherheitsdienst dar.

Das zugrunde liegende Problem ist das Folgende[1]: In einer offenen verteilten Umgebung sollen Benutzer von Arbeitsplatzrechnern aus auf Dienste zugreifen können, die von Servern angeboten werden. Dabei soll der Zugriff nur dan erfolgen, wenn eine Berechtigung dazu besteht und der Server den Benutzer authentifizieren kann. Folgende Angriffe sollen also vermieden werden:

- Ein Benutzer versucht Zugriff auf Ressourcen zu erlangen, indem er sich als ein anderer Benutzer ausgibt.
- Ein Benutzer kann seine Netzwerkadresse so ändern, dass er Anfragen an Server abschicken kann, so dass dieser denk, er käme von dem imitierten System.
- Ein Benutzer kann Anfragen abfangen und durch erneutes Senden unberechtigten Zugang erlangen oder Arbeitsvorgänge abbrechen.

Ansätze für Sicherheit

In verteilten Systemen werden im Allgemeinen drei verschieden Ansätze für Sicherheit verwendet:

1. Vertrauen darauf, dass der Client-Rechner die Benutzeridentität überprüft, so dass die Sicherheitsprozedur auf der Benutzer-ID aufbauen kann.
2. Das Client-System muss sich authentifizieren, im Folgenden wird sich aber auf die Benutzer-ID verlassen.
3. Bei jeder Dienstnutzung muss der Benutzer seine Kennung eingeben – das gilt auch für Server.

Anforderungen bei der Entwicklung von Kerberos

In kleinen und geschlossenen Systemen sind 1 und 2 sicherlich ausreichend, in offenen Umgebungen schützen diese Ansätze allerdings nicht die Benutzerdaten und Ressourcen. Die Anforderungen bei der Entwicklung von Kerberos waren demnach:

- **Sicherheit:** Ein Benutzer des Systems darf nicht die Möglichkeit erhalten, den Zugriff eines anderen Benutzers zu missbrauchen.
- **Zuverlässigkeit:** Ein System muss das andere unterstützen, damit höchste Zuverlässigkeit gewährleistet werden kann, da andernfalls eine Einschränkung der Verfügbarkeit anderer von Kerberos unterstützter Anwendung die Folge wäre.

[1] Siehe auch [Stallings2001, Kapitel 4]

- **Transparenz:** Der Benutzer gibt lediglich sein Passwort an – alles weiter läuft transparent im Hintergrunf ab.
- **Skalierbarkeit:** Durch eine verteilte, modulare Architektur soll eine grosse Anzahl von Clients und Servern unterstützt werden.

Kerberos

Um all diese Anforderungen zu erfüllen, ist Kerberos als **vertrauliche, aussendstehende Authentifizierungsfunktion** implementiert, die ein von **Needham und Schroeder** erfundenes Protokoll verwendet.

Die Authentisierung in DCE basiert auf dem **Kerberos**-System. Dort wird zwischen **Kerberos-Akteuren** (Client, Server, Authentication Service, Ticket Granting Server) und **Kerberos-Elementen** (Ticket, Authenticator, Session Key) unterschieden. Das Verfahren basiert auf dem Needham-Schroeder Verfahren und hatte als Zielsetzung einen „Single-Logon". Deshalb werden zwei Dienste eingesetzt: der **authentication service** (AS) und der **ticket granting service** (TGS).

Ein Kerberos-Ticket für Client C bei Server S ist folgendermassen aufgebaut:

$$\{\text{ticket}\,(C,S)\}\,K_{AS} = \{C,S,t_1,t_2,K_{CS}\}\,K_{AS}.$$

Ein Kerberos-Authenticator für C ist

$$\{\text{auth}\,(C)\}\,K_{CT} = \{C,t\}\,K_{CT},$$

wobei A der AS, T der TGS, t ein Zeitstempel, $t_{1/2}$ Start- und Endzeitpunkte der Gültigkeit, N ein Prüfwert (**Nonce**) und $\{\ldots\}\,K_X$ eine Verschlüsselung mit K_X ist.

Der Protokollablauf einer Authentisierung ist dabei wie folgt:

1. $C \to A : C,T,N$
   Client C erbittet vom Authentication Service A ein TGS-Ticket.
2. $A \to C : \{K_{CT},N\}\,K_{AC}, \{\text{ticket}\,(C,T)\}\,K_{AT}$
   A schickt C das den Schlüssel K_{CT} für die Kommunikation von C mit T und einen Prüfwert N, beides verschlüsselt mit dem gemeinsamen Schlüssel K_{AC}. Das Ticket ist verschlüsselt mit dem Schlüssel von A.
3. $C \to T : \{\text{auth}\,(C)\}\,K_{CT}, \{\text{ticket}\,(C,T)\}\,K_{AT}, S, N$
   C schickt das verschlüsselte Ticket zusammen mit S und N und seiner Authentisierung, die mit dem gemeinsamen Schlüssel K_{CT} verschlüsselt wurde, an T.
4. $T \to C : \{K_{CS},N\}\,K_{CT}, \{\text{ticket}\,(C,S)\}\,K_{AS}$
   T schickt an C ein Ticket für S verschlüsselt mit seinem Schlüssel K_{AS} und dem Schlüssel K_{CS} zusammen mit dem Prüfwert N für die Kommunikation mit S, verschlüsselt mit K_{CT}.

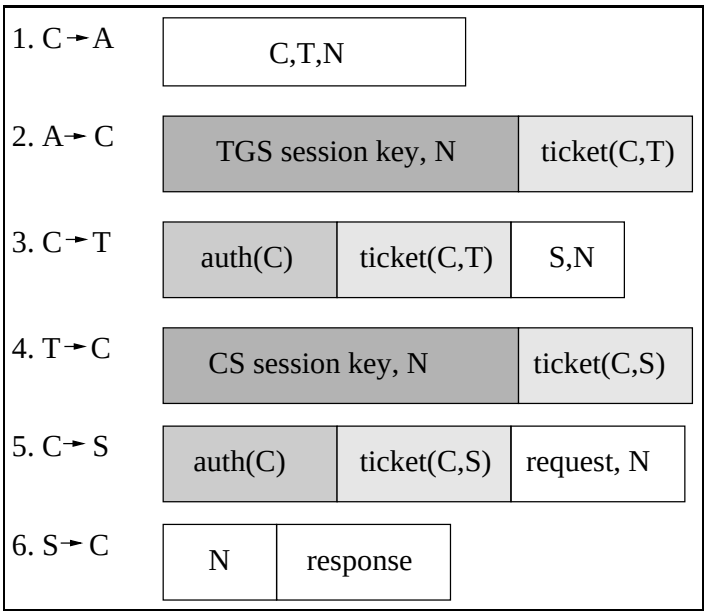


Abbildung 1: Kerberos

5. $C \rightarrow S : \{\text{auth}\,(C)\}\,K_{CS}, \{\text{ticket}\,(C,S)\}\,K_{AS}, \text{request}, N$
   C schickt seinen Service Request zusammen mit N, dem Ticket für S und seiner Authentifizierung an S.
6. $S \rightarrow C : \{N'\}\,K_{CS}$
   S schickt an C einen neue Nounce N' verschlüsselt mit dem gemeinsamen Schlüssel K_{CS}.

Das ganze wird in Abbildung 1 noch einmal grafisch zusamengefasst.

Mögliche Angriffe auf die Netzwerksicherheit beinhalten DoS, Lauschangriffe, Wiederholung, Trojanische Pferde, Masquerading.

Authorisierung

Die Authorisierung in DCE wird über **ACL**s gesteuert, in der jeder Server S seine Zugriffsrechte verwaltet. Der Client erwirbt vom **Privilege Server** ein **PAC** (privilege attribute certificate), in dem Name, Gruppe etc. gespeichert sind. Zur Rechteverwaltung werden ACL's anstelle von Capabilities verwendet, da diese kopierbar sind (was aber durch Einsatz von Verschlüsselung oder durch eine sog. Tagged Architecture verhindert werden kann).

In der DCE-Terminologie werden **Security Server** und **Server** im Allgemeinen unterschieden. Sicherheitsserver sind **Registry**, **Authentication** und **Privilege Server**, die allesamt auf die **Security DB** zugreifen. Server besitzen eine **ACL-Manager**, mit dem sie Zugriffsrechte überprüfen. Die Kommunikation der einzelnen Server mit dem Client ist in Abbildung 2 skizziert. Der genaue Protokollablauf ist in Abbildung 3 aufgezeichnet.

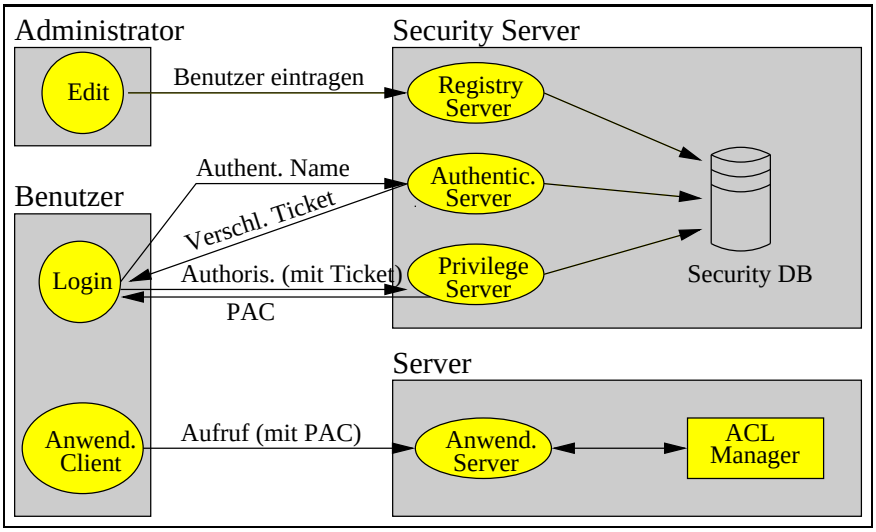


Abbildung 2: DCE Authentisierung und Autorisierung

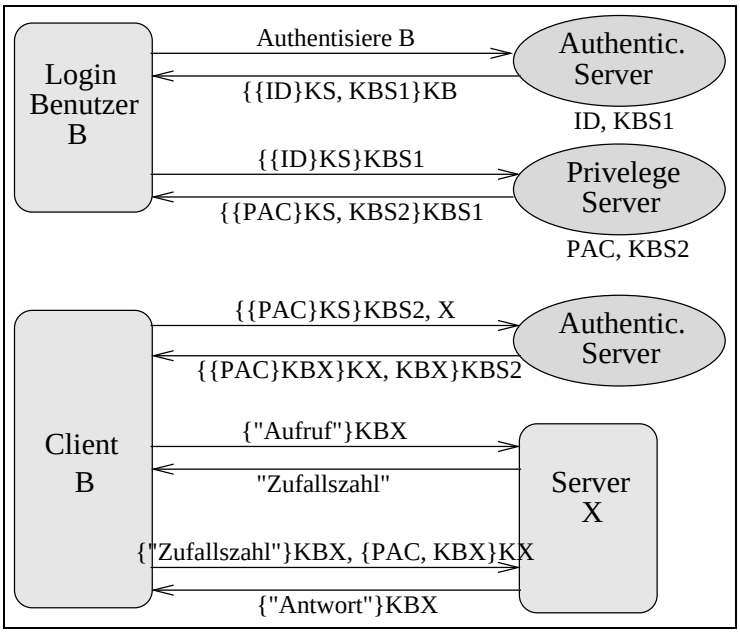


Abbildung 3: DCE Authentisierung (Protokollablauf)

Zusammenfassend werden folgende Angriffe durch Kerberos unmöglich gemacht:

- **Abhören des Passworts**
  Der Client authentifiziert sich gegenüber dem **TGS** (ticket granting server), indem er die Benutzer-ID zusammen mit der TGS-ID an den **AS** (authentication service) übermittelt. Der AS antwortet mit einem Ticket, das mit einem von dem Passwort des Benutzers abgeleiteten Schlüssel verschlüsselt ist, so dass es nur mit Hilfe des Passworts auf dem Client entschlüsselt werden kann. Ist das Passwort korrekt, so ist der Client im Besitz eines Tickets, mit dem er sich gegenüber dem **PS** (privilege service) authentifizieren kann, um ein **PAC** (privilege attribute certificate) für den gewünschten Server zu bekommen. Auf diese Art und Weise wird niemals das Passwort im Netz übertragen.

- **Diebstahl eines Tickets**
  Durch Verwendung von Zeitstempeln kann die Gültigkeit eines Tickets zeitlich begrenzt werden. Zudem ist das Ticken mit einem Schlüssel, den nur der AS und der TGS kennen verschlüsselt, so dass es nicht geändert werden kann.

Die aktuelle Version 5 des Kerberos-Protokolls unterstützt andere Verschlüsselungsverfahren neben DES, ist unabhängig von IP, kodiert Nachrichten in ASN.1 oder BER, vergrössert der maximalen Lebensdauer eines Tickets, Ermöglicht die Weitergabe von Authentifizierungen und vereinfacht Beziehungen zwischen mehreren Kerberos-Bereichen. Daneben werden einige Schwächen der Version 4 behoben wie die doppelte Verschlüsselung von Tickets, Einführung des CBC-Modus für DES, Einführung von Schlüsseln für Teilsitzungen und Verhinderung einer Reihe von Angriffen auf Paswörter.

Insbesondere die Einführung von Zufallswerten (**Nonce**) verhindert sog. Reply-Attacken und stellt ein wichtiges neues Sicherheits-Feature dar.

3.8 Zusammenfassung

DCE bietet Muster für Middleware-Funktionaltät und war Maßstab bei späteren Entwicklungen. Es besitzt einige richtungsweisende Komponenten, wie die **GSS-API** (eine Programmierschnittstelle für Sicherheitsdienste, die auch als Internet-Standard akzeptiert wurde). Das Programmiermodell ist aber veraltet, da Aspekte zur Intgration und Interoperabilität nicht berücksichtigt wurden. Trotz der „Standardisierung" definiert DCE eine Implementierung und ist deshalb kein offenes System. Heute hat es nur noch eine sehr eingeschränkte Bedeutung fü die Praxis.

4 Mobile Agenten als Middleware

Agenten

Es gibt verschiedene Definitionen, was ein **Software Agent** ist. Nach Wooldridge und Jennings (1995) ist das eine **Software-Einheit, die im Auftrag eines Benutzers handelt** und folgende Eigenschaften besitzt:

- **Autonomie**: agiert ohne Intervention von außen und besitzt Kontrolle über seinen inneren Zustand und seine Aktionen.
- **Soziale Fähigkeiten**: Interagiert mit anderen Agenten über eine ACL.
- **Reaktivität**: Nimmt seine Umwelt wahr und reagiert auf Änderungen.
- **Proaktivität**: Zielgerichtetes Handeln, ergreift selbst die Initiative.
- **Mobilität**: Kann zur Laufzeit selbstständig die Ausführungsplattform wechseln.

Agentensystem

Agenten „leben" in einem sog. **Agentensystem**, das als **Ausführumgebung** für sie dient. Es stellt Mechanismen für **Kommunikation**, **sichere Ausführung**, **Abrechnung** und **Migration** bereit und stellt aus der Sicht verteilter Systeme eine **Verteilungsplattform** (Middleware) dar. Eine aktive Instanz eines Agentensystems wird **Stelle** genannt. Damit besteht ein Agentensystem aus Rechnernetz, Rechner, Stelle, Agent und Benutzer.

Einsatzgebiete

Als **Einsatzgebiete** mobiler Agenten bietet sich folgendes an:

- **Mobile/Nomadic Computing**: Rechner führen oft Ortswechsel durch und Netzverbindungen sind kurzlebig. Agenten entkoppeln Client und Server. Bsp.: Aktienhandel.
- **Datenintensive Anwendungen**: Die Datenverarbeitung findet bereit auf dem Server statt, so dass weniger Daten übertragen werden müssen. Bsp.: Netzmanagement.
- **Elektronische Markt**: Agenten fungieren als Unterhändler, die Angebote einholen, vergleichen und verhandeln. Bsp.: Auktion.

Vergleich mit herkömmlicher Middleware

Bei herkömmlichen RPC-basierten Systemen (RMI, CORBA, DCE) sendet der Client Anfragen an den Server und erhält daraufhin Antworten. Bei mobilen Agenten schickt der Client einen Agenten zum Server, der dann dort vor Ort ausgeführt wird. In herkömmlichen Systemen, werden bei n Servern also n Anfragen gestellt, während bei Agentensystemen ein Agent losgeschickt wird, der dann alle n Server besucht und dann mit dem Ergebnis zurückkommt.

In RPC-Systemen tauschen ortsfeste Clients Daten mit Servern aus, was i.d.R. mittels **synchroner Kommunikation** geschieht, und der **Verteilungstransparenz** zuträglich ist. Bei Agentensystemen sind die Clients mobil und die Kommunikation erfolgt **asynchron** über ein Mailbox-System, was ein Verteilungsbewusstsein auf Client-Seite vroaussetzt.

4.1 AMETAS

Der Fokus bei der Entwicklung von AMETAS wurde auf die **Agenten-Autonomie** und **Sicherheit** gelegt. Die Kommunikation erfolgt **asynchron**. Es gibt **permanente** und **temporäre Stellen**, der **Mikrokernansatz** bietet lediglich wenige Grundfunktionen, so dass neue Funktionalität als Dienst auf Anwendungsebene realisiert werden muss.

Kerndienste

Stellen stellen folgende **Kerndienste** zur Verfügung:

- **Postamt**: Verwaltung von Nachrichten, die als Basis jeglicher Kommunikation dienen. Es sind Unicast-, Multicast- und Broadcast-Nachrichten möglich.
- **Ereignissystem**: Benachrichtigung von Agenten durch Stellen. Agenten müssen sich dafür für Ereignisse registrieren.
- **Mobilität**: Übertragung von Agenten durch die Funktion go(`stellenname`). Dabei werden die Aktionen „Einpacken", „Versenden", „Beenden", „Auspacken" und wieder „Starten" durchgeührt. Es wird der Agentenzustand und die Klasse (transitive Klassenhülle) mittels **Objekt-Serialisierung** übermittelt. Die Daten werden mittels **Signatur** und **Verschlüsselung** sicher gemacht und der Start des Agenten auf der Zielstelle erfolgt an vorher definierten Einstiegstpunkten (→**schwache Migration**).
- **Namensdienst**: Auflösung symbolischer Stellennamen nach Socket-Adressen ähnlich dem DNS (hierarchisch, Delegation an zuständige Namensserver). Temporäre Stellen verlangen eine dynamisch veränderliche Bindung zwischen Stellennahmen und Socket-Adressen erlauben.
- **Mediation**: Vermittlung von Agenten/Diensten durch einfache lokale, Namensbasierte Vermittlung. Neu: Typ-basierte Vermittlung auf Basis einer abstrakten Beschreibung.
- **Sicherheitssystem**: Schutz der Agenten voreinander (vertikales Sandboxing), Schutz des Hosts vor dem Agenten (horizontales Sandboxing), zertifizierte Identitäten, Digitale Signatur des Agenten-Code und -Zustand, verschlüsselte Übertragung (RSA, MD5, IDEA), Nicht-Abstreitbarkeit durch Autorensignaturen, Rechtevergabe durch Capabilities auf der Stelle.

Bereits existierende Erweiterungen sind z.B. Gruppenkommunikation, erweiterter Namensdienst, CORBA-Schnittstelle, KQML und Persistenz.

Stellennutzer

In AMETAS gibt es drei Arten von Stellennutzern: **Agenten** (mobile Komponenten mit sehr beschränkten Rechten), **Dienste** (Erweiterungen der Stellenfunktionalität; gewähren Agenten kontrollierten Zugriff auf System-Ressourcen und externe Dienste) und **Adapter** (Integrieren den Benutzer in das System;

bindet externe Anwendungen auf Client-Seite an). Anwendungen bestehen also immer aus Stellen, Diensten, Agenten, Adapter und Benutzer. AMETAS-Projekt

Beim **IntraManager** werden Agenten zum **proaktiven Netz- und Systemmanagement** eingesetzt. **Health Agents** vewalten Knoten und nutzen neuronale Netze für Vorhersagen. Dadurch wurde der Management-Datenverkehr minimiert und eine robuste, flexible und skalierbare Architektur geschaffen, die ganz zentral auf der **Dezentralisierung** des Systems beruht.

Das **Virtuelle Projektbüro** (VPO) biete eine verteilte Büro-Umgebung für verteilte Teams, in der gewohnte Kommunikationsmechanismen (→communication by chance) möglich sind. Der Benutzer wird in dem System durch persönliche Agenten vertreten..

Eine **Informationsinfrastruktur** wird mittels PDIC möglich, das eine Verteilung von Angeboten und Kaufinteressen in E-Commerce Systemen ermöglicht. Es zeichnet sich durch Offenheit, Dezentralisierung und Interessen-gesteuerte Informationsverbreitung aus. PDIC-Agenten arbeiten als intelligente Informationsverteiler und Käufer-Agenten werden durch verteilte Informationen zielgerichtet zu den interessanten Anbietern geleitet.

5 CORBA

Die **Common Object Request Broker Architecture** wurde von der **OMG** ins Leben gerufen und ist eine Definition von Schnittstellen und einer Architektur. Die OMG liefert keine eigene Implementierung – diese muss innerhalb von zwei Jahren nach Verabschiedung eines „Request for Proposal" unabhängig implementiert werden, ansonsten wird der Standard verworfen.

Ziel beim Entwurf von CORBA war die **Application Integration**. Dazu musste zunächst das Problem der **Interoperabilität** gelöst werden, was mit CORBA versucht wurde. Die **Integration** selbst ist work in progress und soll durch **CORBAservices** und **CORBAfacilities** erreicht werden. Die Haupkonzepte von CORBA sind die OMG IDL, die die **Schnittstelle eines Objektes** beschreibt und der ORB, der die **cross-platform Kommunikation** ermöglicht.

Der Einsatz von CORBA bringt für **Entwickler** folgende **Vorteile**:

- **Integration** von Hard- und Software: alle bisherigen Investitionen können integriert werden. Di Sprachunabhängigkeit verlangt nicht das Lernen einer neuen Programmiersprache.
- Das **OO-Paradigma** beim Entwurf und der Programmierung ist lang erprobt und für den gesamten Entwicklungsprozess definiert.
- **Altlasten Anwendungen** können durch sog. Wrapper einfach integriert werden.

- **Höhere Produktivität**, da bereits viele Standardfunktionen in Diensten zur Verfügung stehen. Der strukturierte Entwicklungsprozess ermöglich auch eine bessere Wiederverwendung. Die Portierung wird durch einen einheitlichen CORBA-Standard einfach möglich.
- **Wiederverwendung von Code**: sowohl Komponenten als auch Klassen ermöglichen einen hohen Grad an Wiederverwendbarkeit.

Für **Benutzer** ergeben sich mit dem Einsatz von CORBA diese **Vorteile**:

- Durch die das einfache Austauschen von Komponenten kann der Benutzer aus eine Auswahl die für ihn **beste Lösung auswählen**.
- Der Einsatz eines Industrie-Standards mach das Unternehmen wettbewerbsfähiger.

Die drei Hauptvoraussetzungen, damit ein Objekt in CORBA quasi plug-an-play eingesetzt werden kann sind **Kapselung**, **Vererbung** und **Polymorphismus**. Dabei hat ein Objekt zwei Facetten: seine Schnittstelle und seine Implementierung, die nach aussen aber unbekannt ist. Dadurch wird eine Ersetzbarkeit der Objekte erreicht.

5.1 Objektmodell

Das **Objektmodell**, das CORBA zugrunde liegt besteht aus

- **Objekt**: Instanz einer Klasse mit Name und Lebenszeit.
- **Klasse**: Schablone für Objekt, Vererbung, Polymorphie.
- **Schnittstelle (Interface)**: Methoden und Attribute, Kapselung der Implementierung.

Vorteile der Nutzung von OOP sind Wiederverwendung, Erweiterbarkeit, Anpassungsfähigkeit und eine bessere Strukturierung.

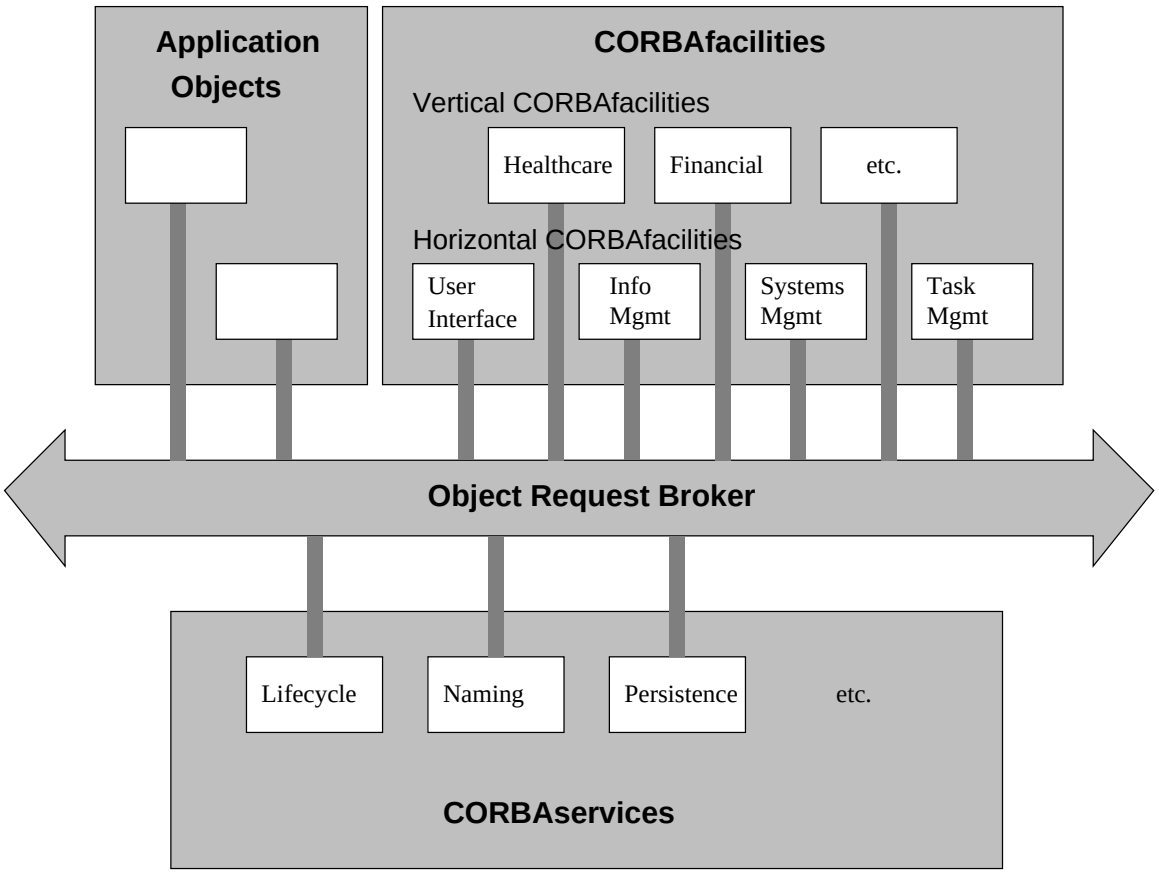
Im Kunde-Bank-Beispiel sind Bank, Benutzer, Konto und Kontoauszuüge Objekte.

Die OMG **Object Management Architecture** (**OMA**) besteht aus zwei Hauptkomponenten: den **lower-level CORBAservices** und den **intermediate CORBAfacilities**. Erstere liefern dabei grundlegende Dienste, die fast jedes Objekt benötigt (Lebenszyklus, verschieben, kopieren, Namen, Verzeichnis, **OLTP** – **Online Transaction Processing** etc.), während letztere Dienste für Aplikationen darstellen (**Compound Documents**), wobei zwischen horizontalen (wie die

Verbund-Dokumente) und vertikalen Diensten (Standardisierung des Management von Informationen, die aus einer bestimmten Domäne kommen) unterschieden wird.

CORBA Dienste

Ein **Software Bus**, der **Object Request Broker** (ORB), organisiert die Kommunikation der Objekte. Die gesamte **Object Management Architecture** (OMA) besteht aus:



- **ORB**:
  - vergibt Objekt-Referenzen
  - nimmt Aufrufe von Clients entgegen
  - Transportiert Aufrufe zum Server
  - aktiviert ggf. Server-Objekte
  - übergibt Aufrufe an Server-Objekte
  - nimmt Ereignisse/Ergebnisse entgegen und übergibt sie dem Client

  - Unterstützt Sicherheits- und Abrechnungsfunktionen.

- **Object Services / CORBAservices** (notwendige Middleware-Funktionen):
  - **Properties**: erweiterte Objekt-Eigenschaften, die nicht im Interface beschrieben werden können (z.B. Systemvoraussetzungen)
  - **Relationships**: Beziehungen zu anderen Objekten (z.B. Abhängigkeiten)
  - **Query**: Abfrage von Objekten
  - **Externalisation**: Rausschreiben von Objektinhalten (überlebt durch Objects by Value)
  - **Lifecycle**:
    * kundenspezifische Objekte die eine Anwendung selbst kreieren kann (→Object Factory, z.B. registrierte und nicht-registrierte Kunden)
    * erzeugt, löscht, kopiert und migriert Objekte (bis auf das Erzeugen ist im Standard nichts weiter festgelegt)
    * einige Objekte werden nicht durch den Lifecycle Service erzeugt, da sie durch Scripts oder beim Systemstart gestartet werden.
    * Object-Factory erzeugt ein CORBA-Objekt bestimmten Typs mit spezifischen Eigenschaften (allokiert Ressourcen, erzeugt Objekt-Instanz, meldet Objekt beim POA, gibt Objektreferenz zurück, signalisiert, dass Objekt aktiviert werden kann)
  - **Naming**:
    * meist bei den Initial Referencen des ORB mitübergeben
    * bildet Name auf Objektreferenz ab
    * reine Abbildungsfunktion ohne Attribute
    * beliebige Realisierung mit Name Server Technik; Name Server arbeiten föderativ zusammen
    * keine Festlegung der Namenssyntax: `struct NameComponent{ string id; string kind; };`. Operationen `bind(Name, Object)`, `resolve(Name)`, `bind_context(Name, NamingContext)`.
  - **Trading**:
    * gelbe Seiten-Funktion mit attribut-basierter Anfrage
    * Standardisiert von ISO (**Open Distributed Processing**, ODP) und OMG (ähnlich ODP)
    * Strukturierung der Objekte z.B. über Konzeptgrapen
  - **Event**:
    * überträgt Ereignismeldungen beliebigen Inhalts vom Erzeuger zum Interessenten
    * entkoppelt Ereignis-Erzeuger und Ereignis-Verbraucher

  - ermöglicht verschiedenste Interaktionsmuster
  - mehrere Erzeuger/Verbraucher können sich registrieren
  - Entkopplung durch Event-Channel:

Rolle	Client-Verarbeitung	Server-Verarbeitung
Publish-Subscribe	Push	Push
Message Queue	Pull	Push
Monitor	Push	Pull
Reines Polling	Pull	Pull

  - leider unterspezifiziert, wurde eingeführt, weil diesynchrone Verarbeitung nicht immer sinnvoll ist, keine Filter, Typisierung der Ereignis-Arten unpraktisch, unklare Semantik
  - Neuer Vorschlag: Notification Service (aufwärts kompatibel zum Event Service)

- **Notification**:
  - Filter, mit denen sich (un-)erwünschte Ereignisse auswählen lassen (auch inhaltlich)
  - **Structured Events**, die Datentypen mit festen und optionalen Feldern enthalten (effizienter als untyed und typed events)
  - Quality of Service, als Garant für die Auslieferung, Lebensdauer, Prioritäten, Auslieferungsreihenfolge (FIFO, Priority, Deadline) sowie Gruppierung bei der Auslieferung von Ereignissen
- **Transactions**:
  - Isolation von Aktionen, Konsistenzerhaltung bei gemeinsamem Zugriff
  - ACID-Prinzip (Atomicity, Consistency, Isolation, Durability)
  - 2-Phasen-Commit (Vorbereitung, Durchführung); der Transaktionskontext ist ein Current-Objekt
  - die OMG **Object Transaction Services** (OTS) erlauben flache und geschachtelte Transaktionen und sind interoperabel mit OTS anderer Hersteller sowei mit legacy transaction systems und C/Open XS, OSI TP, LU 6.2
  - eine Transaction besteht aus den Komponenten **Transaction Client** (→Client: `begin_transaction`, `commit`, `rollback`), **Transaction Object** (→Transactional Server: `rollback`), **Recoverable Object** (→Recoverable Server: `register_ressource`, `rollback`), Ressource (→Recoverable Server: `prepare`, `commit`, `rollback`)
- **Security**:
  - Funktionen werden durch Mechanismen (z.B. ACL, Capabilities) realisiert
  - Unterscheidung in **function** (einzelne Aufgaben), **mechanism** (Realisierung einer Funktion), **service** (Sammlung von Mechanismen), **policy** (Richtlinien, Vorgehensweisen)

  * Funktionen: Authentisierung, Autorisierung, Vertraulichkeit, Auditierung, Nachweisbarkeit, Archivierung, Delegation (Weitergabe von Aufgaben)
  * erfordert Veränderungen des ORBs
  * Elemente: **principal** (Benutzer mit Sicherheitsattributen wie Name, Zugriffsrechte, Rolle Gruppe), **credential** (Behälter für Sicherheitsattribute), **current** (Ausführungskontext einer Operation), **policy** (Handlungsrichtlinie)
  * ORB und Objekt müssen Zugriff auf den Security Service haben
  * Die Server Authentisierung erfolgt mit SSL und stellt die einfachste Form der CORBA Security dar.
  * Die Hauptanforderungen an das Sicherheitssystem sind nach der OMG **Consistency**, **Scalability**, **Regulatory Requirements**, **Enforceability**, **Usability** und **Evaluation Criteria**.
  * Die **Security Functionality** beinhaltet:
    - **Identification and Authentication of Principals**
    - **Authentication of each User of the System**
    - **Authorization and Access Control**
    - **Security Auditing to make Users accountable for their Security related Actions**
    - **Secure Communication between objects**
    - **Cryptography**
    - **Administrative Tools**

- **Persistence Object Service** (**POS**):
  * Beim Persistence Service werden 6 Elemente unterschieden:
    1. **Persistence Object** (**PO**): ein Objekt, das entweder selbst für Persistenz sorgt oder diese Aufgabe an den Client delegiert.
    2. **Persistence Identifier** (**PID**): Identifiziert das PO ähnlich einem IOR, kann jedoch nur im POS eingesetzt werden.
    3. **Client**: Wenn das Objekt das PO-Interace implementiert, dann spielt auch der Client eine Rolle in der Zusammenarbeit mit dem POS.
    4. **Persistence Object Manager** (**POM**): Hier werden Entscheidungen über das Objekt getroffen.
    5. **Persistence Data Service (PDS) and the Protocol**: Implementiert das Protokoll-Interace und koordiniert die Persistenz-Operationen.
    6. **Datastore**: Kann eine Datenbank oder auch einfach ein Dateisystem sein. Da viele Datastores passiv sind, wird der PDS benötigt, um Such- und Speicher-Operationen anzustoßen.
- Time, Licensing, Notification, Concurrency Control.

- **Common Facilities**:
  - horizontale anwendungsnahe Unterstützung (z.B. Compound Documents, System Management).
- **Application Domain Specific Interfaces**:
  - vertikale anwendungsnahe Unterstützung (z.B. Business Objects, Medizin, Finanzindustrie).
- **Anwendungen**:
  - es gibt keine Standardisierung der Anwendungs-Schnittstellen
  - Integration von „Altlasten", Kapselung unf Kompatibilität, Interoperabilität mit ORBs un

Damit ein ORB das Siegel **CORBA compliant** erhält, muss er der **CORBA Core specification** (CORBA Object Model, CORBA Architecture und OMG IDL Syntax und Semantik) und einigen ORB Komponenten (DII, DSI, IR, ORB Interface und BOA) genügen. Ausserdem muss der ORB mindestens ein **Language Mapping** liefern. Ausserdem muss der ORB mit den ORBs anderer Hersteller kommunizieren können, also mindestens eine GIOP-Implemenation (z.B. IIOP oder DCE CIOP) unterstützen.

Object Factories

Im Zusammenhang mit dem Lifecycle Service war die Rede von **Object Factories**. Wenn ein Clientprogramm auf ein Objekt zugreifen möchte, dann stellt sich die Frage, woher dieses Objekt kommt. Dafür gibt es zunächst zwei Möglichkeiten: Das Objekt wurde „von Hand" (oder per Skript) gestartet oder es wurde extra für den Client erzeugt. Für letzteres sind Object Factories zuständig. Anwendungsgebiete sind z.B. Compound Documents, Bestellungen und Reports. Von einer Object Factory spricht man bei jedem Objekt, das andere Objekte erzeugt. Im Zuge der Erzeugung eines Objektes müssen folgende Aufgaben übernommen werden:

- den Ort des Objektes festlegen;
- die Ressourcen (Speicher, System-abhängige Ressourcen etc.), die von dem Objekt benötigt werden, müssen beschafft werden;
- das Objekt muss beim BOA registriert werden, damit es eine Objekt Referenz erhält und dauerhaft gespeichert werden kann;
- das Objekt muss erzeugt werden;
- dem BOA muss signalisiert werden, dass das Objekt für die Aktivierung bereit ist;
- die Objekt Referenz muss an den aufrufenden Client zurückgegeben werden.

- (Das Objekt beim Trading und Naming Service registrieren, so dass andere Clients darauf zugreifen können)

Es gibt also **keine generische Object Factory**, lediglich die Ressourcen, die für die Objekt-Erzeugung benötigt werden können in standardisierter Weise beschrieben werden.

5.2 Schnittstellenaufrufe

SII

Beim Einsatz des **Static Invocation Interface** (**SII**) wird der Stub einer Anwendung, über den auf ein Objekt zugegriffen wird, wird automatisch vom IDL-Compiler für die jeweilige Programmiersprache erzeugt. Dabei muss der Stub selbst nicht zwischen verschiedenen ORBs austauschbar sein, dafür kann aber mit Hilfe eines IDL-Compilers der Stub für jeden ORB neu generiert werden – IDL-Compiler und ORB werden in einem ausgeliefert und gehören zusammen. Das Mapping der IDL zu einer Sprache ist von der OMG standardisiert, so dass der Sourcecode zwischen ORBs austauschbar ist.

DII

Das **Dynamic Invocation Interface** (**DII**) ermöglicht den Aufruf eines Objektes zur Laufzeit, wobei bei der Programmierung noch nicht feststeht, welches Objekt das genau ist. Für diese Freiheit eine Ziel-Objekt, Interface und Methoden zu Laufzeit zu wählen, müssen aber einige Zeilen mehr Code investiert werden. Beim Aufruf von Methoden ist dabei neben dem beim SII möglichen **synchronen oder asynchronen Aufruf** ein **deferred synchronous** Aufruf (nicht blockierend mit Rückgabewert) möglich. Was mit dem DII also nicht möglich ist, geht auch nicht mit dem SII. Auf Objekt-Seite ist dabei transparent, ob der Aufruf über das SII oder das DII gemacht wurde – darum kümmert sich der ORB.

Für den dynamischen Schnittstellenaufruf sind 4 Schritte nötig:

1. **Identifizierung des Objektes, das aufgerufen werden soll**
   Die gewünschte Objektreferenz kann z.B. über einen Trader Service gefunden werden, der ein Register ähnlich den gelben Seiten anbietet.

2. **Besorgen der Schnittstelle des Interfaces**
   Zunächst wirde mit der Objekt-Referenz die ORB-Methode `get_interface` aufgerufen, die die Top-Level Komponenten der Schnittstelle zurückliefert. Mit weiteren Methoden können dann die restlichen Operationen (mit Namen) und Parameter (mit Typen) erfragt werden, wobei allerdings keine Meta-Informationen (wie z.B. was die Methode macht und was Parameter für Wertebereiche haben) mitgeliefert werden, allerdings ist auch nicht vorgeschrieben, wo diese bezogen werden können.

3. **Zusammenbauen des Aufrufs**
   Das DII liefert eine Standard-Schnittstelle für das Zusammenbauen eines Aufrufs. Diese ist in einer **pseudo-IDL** (**PIDL**) formuliert, die von

OMG standardisiert ist. Jedes Objekt besitzt dafür die Standard-Methode `create_request`, mit deren Hilfe dynamische Aufrufe kreiert werden können.

4. **Aufruf durchführen und Ergebnisse empfangen**
   Für den synchronen Aufruf gibt es die Methoden `invoke`, `send` und `get_response`, wobei der Aufruf von `invoke` synchron erfolgt.
   Beim asynchronen Aufruf gibt es die Methoden `send` und `get_response`. Der Aufruf von `send` ist asynchron. Danach kann mit Hilfe von `get_response` auf ein Ergebnis gepollt werden.

CORBA 3.0

Neu in CORBA 3.0 sind **CORBA Components**, **CORBA Messaging**, **Real Time CORBA**, **CORBA Firewall** etc. Weiterhin wurden die Object Services erweitert/verbessert und ein **Minimum CORBA** definiert. Hinzu kamen Standards für E-Commerce, Telekommuniation etc.

Objektreferenzen

Objekt-Implementationen werden in CORBA eindeutig durch **Objektreferenzen** identifiziert. Der ORB bildet eine vom Client präsentierte Objektreferenz auf das zugehörige Objekt ab – so können Objektreferenzen auch als Parameter in der IDL auftauchen. Der ORB ist in der Lage, sich zu jeder Objektreferenz die zugehörige Interface-Spezifikation geben zu lassen, das ORB-Interface bietet hierzu die Funktionen `object_to_string()` und `string_to_object()` an, wobei der String ausgetauscht oder über den Namensdienst ermittelt werden kann.

Bootstrap

Damit ein Client in CORBA starten kann, müssen vom ORB **initiale Objektreferenzen** zurückgegeben werden können (über die Funktion `resolve_initial_references()`): Interface Repository, Naming Service, Trading Service, Notification Service (Erweiterungen sind zugelassen).

Interface Repository

ORB

Das **Interface Repository** liegt...

Der ORB-Kern transportiert Aufrufe und Ereignisse zwischen Clients und Objekt-Implementationen und sorgt für **Transparenz hinsichtlich Objekt-Lokation, Art der Objekt Implementation, Aktivierungszustand und Kommunikationsprotokolle**. Als Implementierungsvarianten stehen **dezentrale ORBs** (der ORB wird in Form von Bibiliotheksroutinen oder als Teil des BS realisiert), **zentrale ORBs** (der ORB wird als separater Prozess realisiert) und **lokale ORBs** (Objektimplementierungen sind als lokale Bibiliothek vorhanden und benötigen keine Stubs) zur Verfügung.

5.3 Client/Server-Struktur

Der Client-Prozess greift über einen **Object Proxy** auf das Objekt zu. Der Objekt Proxy verwendet da **Invovation Interface**, um über den ORB mit der Objekt Implementierung zu kommunizieren. Auf Server Seite kommuniziert die **Objekt Implementierung** im Server-Prozess über das **Server Skeleton** und den

Object Adapter und den ORB mit dme Client. Der **Object Adapter** bietet hier eine Schnittstelle zwischen dem ORB und der Objekt-Implementation und erlaubt es dem ORB Objekte für den Zugriff vorzubereiten und Objekten den ORB zu benachrichtigen, sobald sie bereit sind (siehe [Siegel1996, Abschnitt 5.3 ff.]).

IDL

Mit Hilfe der **Interface Definition Language** (IDL) werden Schnittstellen von Objekten in CORBA definiert. Die Sprache ist **rein deklarativ** und besitzt eine **strenge Typbindung**. Sie wird für die automatische Generierung von Stubs und Skeletons eingesetzt. **Stubs** beinhalten Funktionsdeklarationen, während **Skeletons** Funktionsaufrufe beinhalten. Die Verbindung von Stub und Skeleton zu dem ORB sind dabei proprietär.

Aufrufmechanismen

Es werden **statische** und **dynamische Aufrufmechanismen** unterschieden:

- **statische Aufrufmechanismen** kennen alle Typ-Informationen bereits zur Compile-Zeit (ähnlich RPC, Stub wird benötigt, kaum ein Unterschied zum lokalen Aufruf).
- bei dem **Dynamic Invocation Interface** (DII) gibt der Programmierer an, welches Objekt er nutzen und welche Operationen samt Argumenten er aufrufen möchte.

Dynamische Interfaces

Der Ablauf eines **dynamischen Aufrufs** erfolgt in sechs Schritten:

1. Beschaffung der Objektreferenz (z.B. über den Naming Service)
2. Bestimmung der Schnittstelle (z.B. über das Interface Repository)
3. Erzeugung eines **Request Pseudo Object**, der als Behälter für die zum Aufruf gehörenden Informationen dient
4. Übergabe des Pseudo Objects an den ORB (Aufruf starten)
5. Warten auf das Egebnis (optional)
6. Request Pseudo Object löschen oder wiederverwenden

Pseudo Object

Ein **Pseudo Object** ist eigentlich kein CORBA Objekt. Sein Interface wird in der OMG IDL beschrieben, es kann jedoch nicht als Parameter für eine Methode übergeben werden und kann nicht über das DII angesprochen werden. Die IDL, mit er es beschrieben wird heisst **Pseudo IDL** (**PIDL**). Ein Pseudo Objekt kann man sich als Interface vorstellen, das von jedem Objekt geerbt wird. Die Methoden werden vom ORB ausgeführt, so dass sich der Objekt-Programmierer nicht darum kümmern muss. Methoden wären z.B. `duplicate()` oder `release()`.

Auf Server Seite wird ein **dynamic skeleton interface** (DSI) benötigt, das nicht alle Objekt-Implementierungen kennen muss. Eine Aufrufübergabe erfolgt dann

mit eienr Spezifikation des aktuellen Inhalts. So können z.B. Gateways, die als Client und Server agieren, realisiert werden. Beim dynamischen Aufruf unterscheidet man zwischen dem **synchronen Aufruf** (der Client wird bis zur Antwort blockiert, wie beim RPC) und dem **asynchronen Aufruf** (der Client blockiert nicht, so dass der Zustand des **Request Objects** aktiv abgefragt werden muss). Beim sog. **oneway-Aufruf** wird der Aufruf abgesetzt, ohne dass das Ergenis abgefragt wird.

CORBA Messaging

Mit dem **CORBA Messaging** wurden in CORBA 2.3 zwei weitere Aufruftechniken eingeführt:

- **Callback**: der Client gibt beim Aufruf eine Objektreferenz für die Antwort mit, die der Server zum Abliefern der Antwort zurückgibt.
- **Polling**: Der Aufruf gibt sofort ein Pseudo-Objekt zurück, auf dem der Client das Egebnis abfragen kann.

Dabei werden **Quality of Service-Vorgaben** beachtet, die für einzelne Objekte oder alle Aufrufe gemacht werden können (bzgl. Priorität, Deadline, Hop Count etc.)

Interoperabilität

Das oberste Ziel von CORBA war die Interoperabilität, die sich auf folgende Bereiche erstreckt:

- zwischen Objekten (der ORB ermöglicht Methodenaufrufe)
- zwischen Programmiersprachen (durch IDL-Compiler)
- zwischen ORBs (durch standardisierte Inter-ORB-Protokolle)
- zwischen Verteilungsplattformen (Gateways, z.B. nach DCOM, AMETAS).

Inter-ORB-Kommunikation

Die Kommunikatin zwischen ORBs erfolgt mittels standardisierter Inter-ORB-Protokolle:

- das **General Inter-ORB Protocol** (GIOP) spezifiziert eine Transfersyntax und ein Nachrichtenformat und kann ein beliebiges verbindungsorientiertes Transportsystem verwenden (wird also dann auf ein konkretes Transportprotokoll abgebildet). Die Daten werden in der **common data representation** (CDR) dargestellt.
- das **Internet Inter-ORB Protokoll** (IIOP) stelle eine Implementierung des GIOP auf TCP/IP dar und wird auch häufig als Intra-ORB-Protokoll verwendet. Das **Protokollprofil** spezifiziert die IIOP-Version, den Namen des Servers, die TCP-Portnummer und die Object-ID.
- das **Envrionment Specific Inter-ORB Protocol** (ESIOP) wird z.B. für OSF DCE eingesetzt.

Im GIOP gibt es folgende **Nachrichtentypen**: Request($\rightarrow$), Reply($\leftarrow$), LocateRequest($\rightarrow$), LocateReply($\leftarrow$), CancelRequest($\rightarrow$), CloseConnection($\leftarrow$), MessageError($\leftrightarrow$), Fragment($\leftrightarrow$). Der Aufruf eines Clients wird über das IDL-Interface in einen GIOP-Request umgesetzt, der dann mittels des IIOP übertragen wird, auf Server-Seite in einen GIOP-Request ausgepackt wird und als Aufruf über das IDL-Interface an die Objekt-Implementation weitergereicht wird.

IOR

Die **interoperable object references** stellen ein gemeinsames Zwischenformat für Objektreferenzen dar.

Object Adapter

Die Aufgaben des **Object Adapter** sind:

- Registrierung von Objektimplementationen
- Generierung von Objektreferenzen
- Aktivierung von Server-Prozessen
- Aktivierung der Objekte
- Demultiplexen und Weitergabe von Aufrufen (upcalls)
- Rückgabe von Ergebnissen.

BOA, POA

Der ursprünglich **Basic Object Adapter** (BOA) genannte standardisierte Adapter war als Minimal-Adapter gedacht und sollte durch spezifische Adapter ersetzt werden – mit CORBA 2.2 wurde der **Portable Object Adapter** (POA) eingeführt, der bisherige Schwächen des BOA ausbügeln sollte, der unterspezifiziert war, keine klare Trennung zwischen „Objekt", „Server" und „Implementation" vorgab und zu nicht-portablen Objekt-Implementationen führte:

- Klare Spezifikation und Trennung zwischen Objekt, Objektreferenz und Implementierung (**Servant**)
- ein Servant kann mehrere Objekte bedienen
- transiente Objekte sind möglich
- virtuelle Objektreferenzen können erzeugt werden
- die Aktivierung von Objekten erfolgt im Bedarfsfall transparent
- Hierarchien mehrerer POAs sind möglich
- POAs werden von POA Managern gesteuert.

Eine Nachricht an ein Objekt wird also von dem ORB an den **Root POA** weitergeleitet, der sie an den konkreten POA weitergibt. Dieser wählt anhand der **Active Object Map** den benötigten Servant aus und übergibt diesem die

Nachricht. Eine **POA-Objektreferenz** enthält dabei alle Informationen des zuständigen Servant, wie IP-Adresse, Port Nummer, Implem. Name, POA Name, Objekt ID.

Interface Reposito

Das **Interface Repository** speichert die IDL-Spezifikation für Schnittstellen als Objekte und ist selbst ein CORBA-Objekt. Es kann zur Laufzeit benutzt werden, um Schnittstellen-Informationen zu erhalten, was z.B. bei dynamischen Aufruf-Schnittstellen verwendet wird.

Implementation Repository

CORBA Server registrieren sich im **Implementation Repository**. Hierzu gibt es keine Vorgaben zum Interface oder zum Inhalt, es wird lediglich vom Object Adapter benutzt und ist ORB-spezifisch. Es enthält Angaben zu Executable-Dateien, Zugriffsrechten, Aktivierungskommandos, Aktivierungsparametern und Aktivierungsmodi.

Objects by Value

Neu ab CORBA 2.3 sind die sog. **Objects by Value**, da bis dato nur Objekt Referenzen, nicht jedoch Objekte als Parameter übergeben werden konnten. Bei bestimmten Datenstrukturen ist das auch nicht sinnvoll (z.B. bei Graphen), es gibt jedoch Ausnahmen. Hier wird der Objektzustand beim Empfänger abgelegt, wo ein Pseudo-Objekt vom Typ `valueType` angelegt wird, das keine eigene Identität und keine automatische Rückwirkung auf das Original ermöglicht. Das Verhalten des Objekts muss dynamisch nachgeladen werden und ist nicht näher spezifiziert.

MICO

Ein Beispielimplementation für CORBA 2.3 ist MICO, das aufgrund seiner Mikroker-Orientierung sehr anpassungsfähig ist und vor allem in Forschung und Lehre eingesetzt wird.

5.4 Anwendungen mit CORBA

Die Entwicklung einer Anwendung mit CORBA erfolgt nach folgendem Ablauf:

1. Definition der Schnittstellen in IDL
2. Generierung der Stubs und Skeletons durch den IDL Compiler
3. Implementierung des Clients
4. Implementierung des (Service-)Objekts
5. Registrierung des Servers beim ORB
6. Methodenaufruf des Client auf dem Objekt.

IDL

Die Datentypen in der IDL unterscheiden sich zwischen **basic** und **constructed**, wobei die Basistypen sind

- (unsigned) long, (unsigned) short, float, double, char,boolean, octet, any.

Die Datentypen constructed setzen sich aus

- struct, union, enumeration, sequence, string, template, array

zusammen.

Die Syntax eines Eintrags einer Operation in IDL ist definiert durch:

```
<op_dcl> := [<op_attribute>] <op_type_spec> <identifier> <param_dcls>
[<raises_expr>] [<context_expr>]
```

Ein sog. **Language-Mapping** bildet CORBA-IDL-Typen auf Datentypen in der Zielprogrammiersprache ab. **Module** gruppieren zusammengehörige Schnittstellen und definieren einen eigenen Namensraum. Auch Module werden auf Programmiersprachen-Konstrukte abgebildet, wie z.B. `namespaces` in C++. Der Datentyp `any` kann jeden beliebigen Typ enthalten.

Interface Repository

Das DII benötigt das **Interface Repository** (IR), das ein Archiv für Schnittstellenbeschreibungen darstellt. Einträge im IR enthalten Objekte, die IDL-Definitionen enthalten und ist selbst ein CORBA-Objekt. Es ist nicht definiert, wie die Informationen in das IR gelangen – meist werden sie automatisch über den IDL-Compiler abgelegt. Die Schnittstellendefinitionen werde duch einen eindeutigen Namen oder durch eine global eindeutige **RepositoryID** identifiziert, die eine Eindeutigkeit über Grenzen der IRs hinweg garantiert.

5.5 CORBA Services

Die **Object Management Architecture** schreibt sog. **CORBA Object Services** vor, das sind notwendige Middleware-Funktionen. Sie sind in der **Common Object Service Specification** (COSS) definiert die in 5.1 vorgestellten Dienste. Diese sind die Infrastrukturfunktionen, die jedoch noch von keinem Hersteller vollständig implementiert wurden, da die Spezifikationen sehr umfangreich sind.

Im Gegensatz zu DCE baut die OMG auf die Erprobung vor der Verabschiedung von Standards, während bei DCE konkrete Implementierung den (Binär-)Standard bilden.

5.6 CORBA und DCOM

Das **Component Object Model** (COM) von Microsoft ist das Basismodell für Anwendungsintegration und diente anfangs der Erstellung von Software-Komponenten. Es bildete die Grundlage von OLE, ActiveX und DNA und ist für eine Reihe von Programmiersprachen wie Visual C++, Visual J++, Visual Basic etc. verfügbar. Es definiert einen **binären sprachunabhängigen Standard** für den Zugriff auf Objekte durch die Art und Weise:

- wie Schnittstellen aufgebaut, identifiziert, erzeugt, freigegeben etc. werden
- wie Clients auf die Schnittstellen zugreifen können (lokal, anderer Adressraum, entfernt)
- wie Fehlercodes verwendet werden.

Objektmodell

Das **Objektmodell** von COM legt seinen Schwerpunkt auf die Schnittstelle des Objekts, was nicht dem klassischen Objektmodell entspricht, das einen Zustand hat und instantiiert werden kann. Ein Objekt kann dabei eine oder mehrere Schnittstellen anbieten, die durch einen weltweit eindeutigen Identifikator idenitifiziert wird und von der Schnittstelle `IUnknown` abgeleitet wird. Eine Schnittstelle ist im Prinzip eine Tabelle mit Funktionszeigern einer vorgegebenen Struktur.

DCOM

Die verteilte Version von COM stellt **Distributed COM** (DCOM) dar, in der eine **Mircosoft IDL** (MIDL), die auf DCE IDL basiert (aber nicht 100% kompatibel ist) und zusätzliche Datentypen einführt (z.B. Interface Pointer). Das **DCOM Protokoll** basiert auf DCE RPC und unterstützt zusätzliche Aufrufparameter und Services für Prozessaktivierung, Finden eines Objekts und Fehlerbehandlung etc. Es werden eindeutige Schnittstellen- und Klassen-IDs verwendet. DCOM wurde von der OpenGroup als offener Standard festgeschrieben.

Client/Server-Konfiguration

In DCOM können der Client und der Server unterschiedlich konfiguriert sein

- im gleichen Prozess: der Server ist als .exe oder .dll realisiert
- auf dem gleichen Rechner, aber in unterschiedlichen Prozessen: die Kommunikation erfolgt über einen optimierten Leightweight-RPC mittels Local **Object Proxies** und **Stubs**
- auf verschiedenen Rechnern: der Client-Prozess greift über ein Interface Proxy und die COM Library (Service Control Manager, Registry) und Object-RPC und den Stub im Server-Prozess auf das Ziel-Objekt zu.

CORBA vs. DCOM

Eine Übersicht über Bestandteile und Terminologie einer Middleware gibt Tabelle 1.

Die Zielsetzung bei der Entwicklung beider Middleware-Modelle war die gleiche, Ziel war **Integration**, **Interoperabilität** und **Komponenten-Software**. Beide verfügen über ein verteiltes Objektsystem mit transparentem Zugriff auf entfernte Objekte, eine (M)IDL, statische sowie dynamische Zugriffe und ein Schnittstellenverzeichnis.

Im Unterschied zu CORBA ist DCOM proprietär, hat ein anderes Objektmodell (in DCOM können Objekte mehrer Schnittstellen haben, eine CORBA Object

Tabelle 1: DCOM-CORBA

Konzept	DCOM	CORBA
Basisklasse	IUnknown	CORBA::Object
Objekt-Identif.	CLSID	Interface-Name
I/F-Identifier	IID	Interface-Name
Objekt-Zeiger	interface pointer	Objektreferenz
Name→Implementation	Registry	Implementation Repository
Typinformation	Type Library	Interface Repository
Lokalisieren	SCM	ORB
Aktivieren	SCM	Object Adapter
Client Stub	proxy	stub (proxy)
Server Stub	stub	skeleton

Reference entspricht nicht einem DCOM Interface Pointer) und ist ein Binärstandard. Der Schnittstellenvererbung von CORBA steht die Schnittstellen-Aggregation von DCOM gegenüber und während CORBA mit beliebigen Protokollen arbeitet, funktioniert DCOM nur mit dem RPC. DCOM verfügt ausserdem über eine Garbage Collection, die mit Referenzzählern realisiert ist.

Es gibt bereits Bridges zwischen DCOM und CORBA (z.B. der OLE Broker bei ORBIX), die die beiden Welten wo es geht miteinander verbinden.

COM+

Ein neues Komponentenmodell von Microsoft ist **COM+**. Es besteht aus COM/DCOM (**Komponentenmodell mit Laufzeitumgebung**) und dem **Microsoft Transaction Server** (MTS) – es fehlen allerdings die Einbettung transaktionaler Anwendungen und die Integration von Legacy Applications. der MTS arbeitet als Transaktionsmonitor und stellt die Middle-Tier der dreistufigen Microsoft **Distributed Internet Application** Architectur (DNA) dar. Er dient als Container für Komponenten, zur Transaktionssteuerung und für weitere Komponentendienste.

Die Nachteile beim Arbeiten mit Komponenten (großer Aufwand für „Drumherum", Skalierbarkeit ist nicht Teil des Komponentenmodells) versucht COM+ durch folgende Mechanismen zu lindern:

- **Just-in-time-Aktivierung** (Skalierbarkeit durch Sparsamkeit)
- **Object Pooling** (Skalierbarkeit durch Recycling, z.B. Datenbankverbindungen)
- **Lastbalancierung** (Skalierbarkeit durch Lastverteilung)
- **Queued Components** (asynchrone Aufrufe mit MSMQ)
- **Rollen-basierte Sicherheit** (Erweiterung der Gruppenstrukturen)
- **Datenbank im Speicher** (Performancegewinn durch Caching)

- **Ereignisse mit Publish/Subscribe** (Skalierung durch Entkopplung)

Portierung

Zunächst gab es DCOM nur auf der Windows-Plattform, eine Portierung (**EntireX**) wurde von der Software AG durchgeführt. Sie kombiniert die Message Broker Technologie mit DCOM und bietet auch Sprachanbindungen an beliebige nicht-OO Sprachen. Die Integration von legacy Applications geschieht durch Wrapper und es werden IMS und CICS Transaktionen integriert. Ausserdem existieren Schnittstellen zu APPC und MQSeries und ein Sicherheitssystem (SAF).

6 Java Middleware

Vorteile der Programmiersprache Java ist: sauber objektorientiert, typsicher, standardisiert. Ergebnis des Übersetzungvorgangs ist ein plattformunabhängiger Bytecode, der von einem Bytecode-Interpreter ausgeführt oder in Binärcode übersetzt wird.

6.1 RMI

Die **Remote Method Invocation** (RMI) stellt einen Aufrufmechanismus für verteilte Java-Objekte dar und zeichnet sich durch Plattformunabhängigkeit aus. der Methodenaufruf erfolgt **ortstransparent** und die Implementierung ist vollständig Java-basiert (und benötigt deshalb keine IDL). Methoden werden über URL-basierte Namen aufgerufen. Es werden sowohl Daten, als auch Code wenn notwendig dynamisch, geladen. Klasseninformationen werden in der **RMI-Registry** gespeichert. Ein einheitlicher Exception-Mechanismus sorgt für eine konsistente Fehlerbehandlung.

Naming

RMI benutzt eine nicht-persistente RMI-Registry, in der URLs als Namen gespeichert werden. Clients können sich dann mittels `bind()`, `unbind()` und `rebind()` registrieren und mittels `lookup()`, `list()` suchend auf die Registry zugreifen (entfernte Clients können die Registry nur abfragen).

Kommunikation

Möchten Clients auf ein Objekt zugreifen, das sie in der Registry gefunden haben, können sie sich zur **Laufzeit Stubs von dem Server herunterladen**. Der **Stub Code selbst kann ggf. auch dynamisch vom Server erzeugt werden**. Die Klassen werden dann mit dem Objekt-Serialisierungs-Mechanismus übertragen.

Sicherheit

Sicherheit ist auf die Befugnisse der Stubs eingeschränkt – das **Stub loading** ist allerdings auch abschaltbar. Der **SecurityManager** ist ein vorgeschriebener Teil des RMI und erlaubt Einschränkungen hinsichtlich dem Laden von Klassen. Allgemein gilt, dass Stubs nicht:

- Verbindungen annehmen
- Ports abhören

- Thread ausserhalb ihrer Threadgruppe manipulieren
- weitere Prozesse starten
- ClassLoader erstellen
- DLLs anbinden
- die VM beenden
- Dateien öffnen
- ...

dürfen. Serverseitig ist der Ablauf bei der RMI wie folgt: Ablauf bei der RMI

1. Server aktiviert seinen RMI Security Manager (**setSecurityManager**)
2. Server registriert sein **remote object** bei einem RMI Naming Service (**Naming.bind**)
3. Server ist bereit für beantwortung von Anfragen.

Ist der Server bereit so kann der Client auf ihn zugreifen:

1. Client aktiviert seinen RMI-SecurityManager (**setSecurityManager**)
2. Client besorgt sich die URL des **remote objects**
3. Client präsentiert beim Naming Service die URL des **remote objects** (**Naming.lookup**)
4. Client erhält lokalen Proxy für das Objekt und der Stub wird ggf. geladen
5. Client führt Aufrufe auf das Objekt aus (über den lokalen Proxy).

Vergleich mit CORBA

Vorteilhaft bei RMI ist, dass

- keine zusätzlichen Übersetzungsschritte benötigt werden
- eine kompakte Implementierung möglich ist
- sowohl Daten als auch Code übertragen werden kann
- die Client Stubs dynamisch vom Server laden können.

Nachteile gegenüber CORBA sind:

- Einschränkung auf Java

- propritärer Standard
- stellt nur rudimentäre Infrastrukturdienste zur Verfügung
- schlechte Performance (fast doppelt so langsam wie CORBA).

Eigentlich ist ein Vergleich von CORBA mit RMI nicht gerade fair, da Java ein Programmiersprache ist und CORBA ein Technologie zur Integration von Software. Aus diesem Grund wird die Java-Entwicklung seitens der OMG kritisch beäugt – SUN strebt jedoch eine Annäherung an und bietet bereits IIOP als Teil von Java 2 an. Natürlich gibt es auch Java Orbs und Java und CORBA schliessen sich ganz und gar nicht aus. Die gute Integration in das Internet-Umfeld prädestiniert Java gerade für Anwendungen in diesem Umfeld, während sich CORBA und DCOM in diesem Gebiet noch etwas schwer tun.

6.2 Object Web

Die umständlichen Mechanismen, derer man sich im WWW bedienen muss (CGI, Formulare), führten zu einer Reihe proprietärer Server-Erweiterungen, die wegen des grundsätzlich zustandslosen HTTP-Protokolls fragwürdige Kontextsicherungen über **Cookies** und **Session Objects** realisierten. An dieser Stelle wäre ein Integration einer Middleware wie Java/CORBA/WWW sinnvoll – z.B. mittels CORBA-Clients als Java-Applets. Problem bei der Sache ist die „Schwergewichtigkeit" von CORBA und Probleme, die bei der Kommunikation über Firewalls entsteht.

6.3 Jini

Suns Konzept einer **dynamisch verteilten Systemarchitektur** stellt **Jini** dar. Es definiert eine Menge von APIs und ist als Erweiterung der Java 2-Plattform realisiert. Als **Abstraktionsschicht zwischen Applikation und Betriebssystem** kann Jini als Middleware bezeichnet werden, die Java in drei Dimensionen erweitert: **Infrastruktur**, **Programmiermodell** und **Dienste**. Dabei werden vorrangig **dynamische, kooperative und spontan vernetzte Systeme** unterstüzt. Es basiert auf JavaSpaces und RMI zur Kommunikation zwischen Java-Objekten und nutzt Sicherheitssystem von RMI. Voraussetzung dafür ist natürlich die Existenz einer Java VM.

Dienste

Das gesamte Konzept von Jini ist **dienstorientiert**: alles (Hardware, Software, Benutzer) ist ein Dienst und ein Jini-System ist eine **Föderation von Diensten.** Mobile Proxy-Objekte regeln den Zugriff auf die Dienste, die sich **spontan** zusammenfinden. In dieser offenen, verteilten und dynamischen Welt bietet die Laufzeitinfrastruktur Mechanismen, um Dienste zu finden, zu verwenden, zu registrieren und zu entfernen. Dabei können Dienste den Clients **Dienstproxies**

injezieren, die eine **Dienstnutzung ohne vorhergehende Softwareverteilung** ermöglichen (Bsp.: Druckertreiber). Die Dienste werden mit Typ und Eigenschaften beschrieben.

Spontane Vernetzung

In Jini bilden die Aktuere nur eine zeitlich befristete Gemeinschaft – mit den zugehörigen Problemen wie:

- kein a priori-Wissen über Existenz, Schnittstelle, Funktionalität und Vertrauenswürigkeit eines Dienstes
- kein Wissen über die Lokalität eines anderen Akteurs
- „Dienst-Treiber" müssen geladen werden.

Im Gegensatz dazu weiss der Server beim Client/Server-Modell fast nichts über seine Clients, der Client kennt jedoch die Schnittstelle des Servers und weiss um dessen Vertrauenswürdigkeit.

Lookup Service

Aus diesem Grund wird von Jini der **Lookup Service** zur Verfügung gestellt. Er bildet das Kernelement jeder Jini-Föderation und arbeitet ähnlich wie die RMI Registry oder der CORBA Naming Service. Zu seinen Aufgaben gehört:

- als Anlaufstelle für Clients zur Verfügung zu stehen
- Registrierungen von Diensten entgegen zu nehmen und publik zu machen
- Dienste zu vermitteln
- Server Proxies zur Verfügung zu stellen.

Die Lookup Services können dabei ähnlich wie im DNS hierarchisch und redundant angeordnet werden.

Einen groben Überblick, wie das Einfügen und Nutzen eines Jini-Dienstes schematisch abläuft gibt Abbildung 4.

Discovery Protokoll

Das **Discovery Protokoll** dient zum Finden eines Lookup Service in einem Netz, über das keine Details bekannt sind. Dabei wird ein **Multicast** an eine bekannte Adresse (well-known address/port) geschickt. Daraufhin antwortet ein Lookup Service mit einem serialisierten JavaRMI-Proxy-Objekt, das für den Zugriff auf den Lookup Dienst benötigt wird. Die Kommunikation mit dem Lookup Service erfolgt an über dieses Proxy Objekt.

Join Protokoll

Möchte ein Dienst eine eigene Dienstleistung beim Lookup Service anmelden, dann teilt er diesem seinen eigenen Dienstproxy inklusive der genauer qualifizierenden Attribute mit. Daraufhin erhält er eine **Lease** für seinen Eintrag, mit der der Dienst eine Zugangsberechtigung für eine gewisse Zeitspanne hat (die Lease enthält ein Verfallsdatum und muss dann erneuert werden). Im Folgenden kann der Dienst dann im Jini-Verbund gefunden und verwendet werden.

Lookup Protokoll

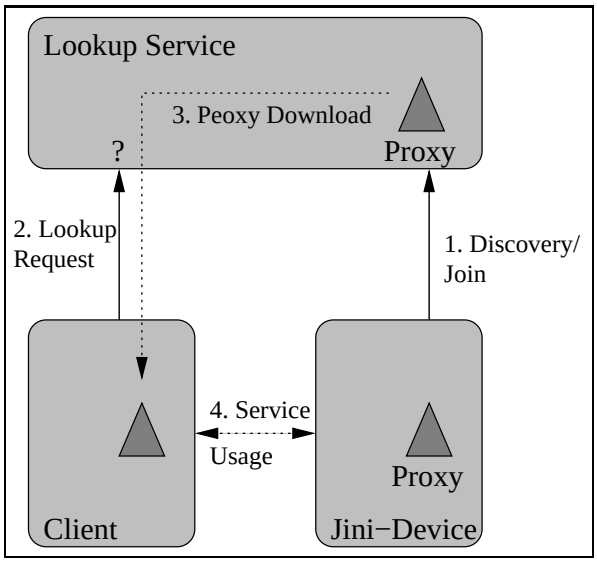


Abbildung 4: Jini Überblick

Möchte ein Client einen Dienst finden und kennt er bereits den Lookup Service, so verwendet er das **Lookup Protokoll**, mit dem Dienste über sog. **Service-Templates** gefunden werden können. Dabei wird ein Matching nach Registriernummer und/oder Typ und/oder Attributen durchgeführt. Wildcards sind erlaubt, ansonsten wird ein „exact-match" durchgeführt. Der Lookup Service liefert daraufhin eine Menge von Treffern zurück, aus denen der Client dann lokal einen Dienst auswählt, auf den er über dessen Protokoll Proxy zugreift.

Das Programmiermodell basiert auf dem von JavaBeans: Es gibt das **Leasing-Interface** (Reservierung/Freigabe), das **Event- und Notification-Interface** (ereignisgesteuerte Kommunikation) und das **Transaction-Interface** (alles oder nichts). Dienste werden von **Providern** oder **Brokern** verwaltet, bei denen der Service ein Interface (**Proxy**) als Java-Objekt registriert, das durchaus auch mehrere Interfaces implementieren kann. Die Vereinigung aller registrierten Ressourcen wird mit **Djinn** bezeichnet.

6.4 JavaSpaces

Eine Ebene über RMI, aber noch unterhalb des Lookup Service sind die **JavaSpaces** angeordnet, auf denen der Lookup Service aufbaut. Vom Prinzip her modellieren die JavaSpaces den **Objektfluss zwischen Anwendungen** (z.B. bei Producer/Consumer-Anwendungen). Es werden die Direktiven `write()`, `read()`, `take()` und `notify()` neu eingeführt, mit denen Objekte geschrieben, gelesen, entnommen oder sich über das Auftauchen eines neuen Objekts benachrichtigt werden kann. Damit werden beliebige Interaktionsmuster zwischen

Aktueren ermöglicht. Die Idee hinter JavaSpaces basiert auf Ideen aus Linda („Tuple Space"). Eigenschaft

Zusätzlich zu den erwähnten Eigenschaften ermöglichen JavaSpaces

- persistente Speicherung von Objekten in einem JavaSpace
- atomare Schreiben/Lese-Operationen
- Zugriffe auf JavaSpaces können Teil von Transaktionen sein
- Melden von Ereignissen.

Ein Objekt in einem JavaSpace passt zu einem Template, wenn die Typen kompatibel sind (identischen oder Subtyp) und die Attributwerte positionsweise identisch oder im Template mit Wildcards angegeben sind.

Es kann durchaus mehrere JavaSpaces geben, auf die Sender und Empfänger zugreifen können. Ein denkbares Interaktionsmodell vom Typ „Message Passing" stellt das Einstellen eines Objektes in des Space dar und die Benachrichtigung eines Empfängers, der sich dafür registriert hat.

6.5 Java Message Queue

Eine weitere Java Middleware stellt die **Java Message Queue** dar. Sie stellt eine Kommunikationstopologie dar, in der sog. **Message Router** Nachrichten zuverlässig zustellen. Sie bilden ein virtuell voll-verbundenes Netz von Message Routern, auf die Clients asynchron und Server sowohl synchron als auch asynchron zugreifen. Beim asynchronem Zugriff registriert sich der Server als MessageListener beim Message Router und liefert diesem die Nachrichten. Kommunikation

Als Kommunikationsmuster steht sowohl die Punkt-zu-Punkt, als auch die Publish-Subscribe-Kommunikation zur Verfügung, wobei bei ersterer eine Queue und bei letzterer ein „Topic" adressiert wird. Nachrichten bestehen aus einem **Header** (der Nummer, Ziel, TTL, Typ, ReplyTo etc. enthält), den **Properties** (beliebige (name,value)-Paare; dienen zur Filterung) und dem **Body** (er enthält die eigentlichen Daten).

Sessions müssen explizit gestartet und beendet werden, wobei eine Session **transaktionsgesichert** werden kann, also eine Auslieferungsgarantie für eine Sequenz von Nachrichten auch über Systemabstürze von Clients hinweg gewährleistet werden kann.

6.6 Zusammenfassung

In Java existieren **viele Modelle** für Middleware-Ansätze, die fast alle auf der **Übertragung von Daten und Code** basieren. Die **nahtlose Integration** in die

Programmiersprache, das Programmiermodell und das Internet machen sie für den Einsatz sehr interessant. Leider sind sie allesamt **sprachgebunden**, **proprietär** und **nicht performant**.

7 Ausblick

7.1 Nachteile von DCOM/IIOP

Da DCOM „sehr" verbindungsorientiert ist, muss viel Aufwand betrieben werden, um Sitzungen aufzubauen und zu unterhalten. Ausserdem gibt es DCOM nicht für alle Plattformen. Das DCOM und IIOP sehr komplexe Protokolle sind, die aufwendige Lauzeitumgebungen benötigen, ist die Portierung der Protokolle auf andere Plattformen schwer und zieht einen hohen Administrativen Aufwand nach sich – von der bereits angesprochenen Firewall-Problematik ganz zu schweigen.

7.2 HTTP+XML=SOAP

Seit einiger Zeit gilt **HTTP** als neue Stern am Himmel der Schicht 7-Protokolle gesehen, das wegen des Internets einen hohen Verbreitungsgrad hat. Der Hauptvorteil von HTTP ist mit Sicherheit seine **Einfachheit** sowohl beim Verbindungsaufbau als auch für die Unterhaltung einer Session. Dabei unterstützt HTTP einfache, aber effektive Sicherheitsmechanismen und verträgt sich wunderbar mit Firewalls.

XML

Als „Nachfolger" von HTML wurde XML eingeführt, das eine universelle textbasierte Datenbeschreibung definiert. Es ist **plattformunabhängig**, **weitgehend akzeptiert und einfach erweiterbar**. Transformationen können mit Hilfe von XSLT in definierter Weise durchgeführt werden und Werkzeuge und APIs existieren mittlerweile für alle wichtigen Plattformen.

SOAP

Oft als XML-RPC bezeichnet, bietet das **Simple Object Access Protocol** (SOAP) ein Protokoll, das auf XML und HTTP basiert. Es wurde als **minimaler Konsens für ein Protokoll zum Aufruf von Methoden** eingeführt und schreibt **keine API oder Laufzeitumgebung vor** – kein ORB oder Web Server wird benötigt. Auch ein **Interaktionsmodell wird nicht vorgeschrieben**, wie es sonst bei RPC-artigen Mechanismen der Fall ist. Durch die plattformunabhängigkeit verursacht das Protokoll wenig Aufwand in Wartung und Implementierung. Die **SOAP-Zwiebel** stellt die Schichten dar, aus denen sich SOAP zusammensetzt:

1. XML 1.0 + Namespaces
2. XML Schema Definition Languag (opt)
3. Element Normal Form

4. SOAP: Envelope
5. Serialized Instance as Method Request
6. SOAP over HTTP Mapping.

Dabei gibt es drei Ansichten eines SOAP-Aufrufs:

- als Objekt-RPC (der Aufruf enthält `in` und `inout` Parameter, die Antwort `inout` und `out`)
- als „messaging" Protokoll (Aufruf und Antwort enthält serialisiertes Aufruf-Objekt)
- als XSLT „über Draht" (Aufruf enthält XML-Dokument, Antwort enthält Transformation)

7.3 SOAP vs. DCOM vs. CORBA

Die Funktionalität von SOAP ist eine Obermenge von CORBA GIOP/IIOP: SOAP=GIOP+multi-interface objects.

Von DCOM ist die Funktionalität von SOAP eine Untermenge: SOAP=DCOM-pinging/garbage collection.

Aber: zunächst führt SOAP zu einer aufwendigen XML-Konversion pro Methodenaufruf und verursacht viel Netzlast, was durch Einsatz einer Binär-Codierung (WBXML) gelindert werden soll.

Noch hat SOAP seine primäre Bedeutung im B2B-Verkehr.

7.4 Microsoft .NET

Von Microsoft als Rahmenwerk für die Erstellung moderner Komponentenbasierter Anwendungen gedacht, soll .NET die Entwicklung klassischer Windows-Applikationen, moderner Web-Anwendungen, wiederverwendbarer Komponenten und von Web Services vereinfachen. Dabei soll eine Fusion neuer Technologien und Anwendungsanforderungen möglich gemacht werden. Durch eine **Common Language Runtime** wird eine Programmiersprachenunabhängigkeit erreicht, da Source Code in **Managed Code** (gemäß der **Common Language Specification**) übersetzt wird, der dann in der **Common Language Runtime** mittels eines JIT-Compilers in Nativen Code umgesetzt wird.

Integration von COM/COM+

Jede .NET-Klasse kann auch ein COM Objekt sein, das von Anwendungen weitgehend transparent mit Diensten von COM+ genutzt werden kann. Hinzu kommen die Verwendungen von Transaktionen, Object Pooling und die automatische Registrierung und Aktivierung.

7.5 Web Services

Unter einem Web Service ist

> „a programmable application component accessible via standard Web protocols"

zu verstehen. Funktionalität wird also über das Web integriert, wozu XML und SOAP eingesetzt werden (XML, um einen RPC zu realisieren und SOAP als zugrunde liegendes Aufrufprotokoll).

7.6 Diverses

In der Forschung beschäftigt man sich unter anderem mit **QoS**, die verschiedene **nicht-funktionale Eigenschaften** eines Dienstes betreffen kann. Beim QoS Management werden folgende Phasen durchlaufen:

- Specification and Negotiation
- Monitoring, Adaption and Resource Control
- Accounting, Billing, Profiling, Planning.

MAQS

Bei der Entwicklung von **MAQS** waren die Ziele

- CORBA-Erweiterungen für dynamisches QoS-Management mit einem generischen Framework für viele QoS-Kategorien
- klare Trennung von Anwendung und QoS-Funktionalität
- adaptives Verhalten je nach Umgebung
- Philosophie: QoS als Aspekt im Sinne des Aspect Oriented Programming (AOP).

AMETAS

Beim AMETAS werden mobile Agenten als Verteilungsplattform eingesetzt. Eine konkrete Anwendung ist **NetDoctor**, ein Programm zum System- und Netzmanagement. Bei dem Personalized Distributed Interest Catalgue (PDIC) verteilen sich Informationen dynamisch im Netz mittels sog. Verteileragenten. Suchagenten durchstöbern das Netz nach Informationen, die sie auf den Knoten finden. Insgesamt wird der elektronische Markt als eines der Haupteinsatzgebiete mobiler Agenten gesehen.

7.7 Ausblicke

Für die Integration mit Komponentenmodelen stehen Java Beans, CORBA Components, COM+ etc. zur Verfügung. Neue Interaktionsmuster (Messaging, Publish/Subscribe, Multicast) und QoS (Echtzeit, Dienstgüte, Verbrauchserfassung, Abrechnung) werden eingeführt. Dynamische Strukturen verlangen neue Technologien wie Jini und UPAP. Neue Paradigmen wie Mobile Agenten und JavaSpaces suchen neue Einsatzgebiete und bei allem müssen das Internet und XML dabei sein.

8 Fragen

- **Vorteil von MQ gegenüber RPC/CORBA/OO?**
  Vorteile des Message Queueing sind Asynchronität, Flexibilität, geringer Overhead und Entkopplung – das war's aber schon. Transparenz bzgl. der Plattform oder der Datenrepräsentation sind hier nicht gegeben, das Abstraktionslevel ist hier also sehr niedrig und der Programmierer muss sich mit sehr vielen Aspekten auseinander setzen, die ihm bei RPC, CORBA und OO (?) abgenommen werden.

- **Was sind Java Message Queues?**
  Java Message Queues sind einfach MQ in Java realisiert. Dazu gibt es die **Java Messaging Service** (**JMS**), ein Standard von Sun, in dem sog. Provider Nachrichten (messages) austauschen. Bei Java Message Queues handelt es sich hier um einen solchen Provider, der als kommerzielles Produkt, das nicht in das Standard JDK übernommen wurde, verkauft wird. Grundsätzlich sind hier zwei Arten der Kommunikation möglich: Punkt-zu-Punkt und publish/subscribe. Bei den JMQ sind alle Clients untereinander anonym, lediglich eine Authentifizierung beim Server ist möglich, so dass dieser Nachrichten speichern und später versenden kann, wenn ein Client gerade nicht online ist (**durable subscriber**). Der JMS-Standard macht keine Aussagen zu Kommunikation zwischen Servern, Fehlertoleranz, Fehlerbenachrichtigung, Administration oder Sicherheit – das ist Sache der der JMQ-Hersteller.
  Eine Nachricht besteht aus den 3 Teilen Kopf (header), Eigenschaften (attributes) und der eigentlichen Information (body). Dabei ist die eigentliche Nachricht schreibgeschützt, bestimmte Teile (wie der Empfänger) können aber geändert werden, damit eine Nachricht z.B. weitergeleitet werden kann. Weitere Informationen sind Hersteller-spezifische Daten.
  Ablauf: Der Client fordert eine Connection zum JMS-System nämlich dem Provider an und authentifiziert sich ihm gegenüber (z.B. über Username, Passwort). Der Provider kann z.B. über die JNDI gefunden werden. Danach werden eine oder mehrere Sitzungen eröffnet, die auch transaktionsbasiert sein können (entsprechende Direktiven stehen in der API zur

Verfügung). Über eine Sitzung kann ein Client nun als MessageProducer oder MessageConsumer agieren. Um sofort über neu ankommende Nachrichten informiert zu werden, muss der Client das MessageListener-Interface implementieren.

- **Was sind Race-Conditions?**
- **Wann werden die UUID's in der IDL neu generiert?**
- **Wo liegen CORBA Skeletons und Proxies?**
  Die Stubs und Skeletons werden beim Linken der Programme in die Client/Server-Programme integriert und liegen je nach dem auf dem Client oder dem Server.
- **Wo liegt das Interface Repository?**
- **Muss der POA selbst programmiert werden?**

Literatur

[Stallings2001] W. Stallings: „Sicherheit im Internet", Addison Wesley 2001.

[Siegel1996] J. Siegel: „CORBA – Fundamentals and Programming", Wiley 1996.