Distributed Systems

-

Architekturen verteilter Software Systeme

Hausarbeit

von
Michael Nordhoff
Thomas Wendt

Betreuer

Prof. Dr. Rainer Unland
Dipl.-Wirt.Inform. Stefan Hanenberg

Grundzüge Systems Engineering
Universität Duisburg-Essen
Juli 2003

Inhaltsverzeichnis

1 Einleitung 4

1.1 Motivation 4

1.2 Aufgabenstellung 5

1.3 Übersicht über die Kapitel 5

2 Grundlagen 6

2.1 Client-Server Architekturen 6

2.2 Verteilte Objekte Architekturen 7

2.3 Computer Netzwerke 8

2.4 Grundsätzliches zu Middleware 9

3 CORBA 11

3.1 Einführung 11

3.2 Object Request Broker (ORB) 12

3.3 OMG Interface Definition Language (IDL) 13

3.4 Schnittstellen des ORBs: GIOP & IIOP 15

3.5 Verzeichnisdienst und Namensauflösung anhand eines Beispiels 16

4 .NET 18

4.1 Einführung 18

4.2 .NET Remoting Infrastructure 19

4.3 Assemblies und IL in der Common Language Runtime 20

4.4 Schnittstellen zu anderen Techniken (SOAP) 21

4.5 Beispiel: .NET Anwendung 22

5 Java – RMI 23

5.1 Einführung 23

5.2 RMI Architecture 24

5.3 IDL in RMI 25

5.4 Namensdienst anhand einer Beispielanwendung 25

6 Vergleich der Middleware-Typen 26

7 Resümee 27

8 Quellen 28

1 Einleitung

1.1 Motivation

Bis vor nicht allzu langer Zeit waren die meisten großen EDV-Systeme zentralisiert. Große und relativ leistungsstarke Mainframe-Computer beheimateten mehr oder weniger komplexe Softwareprogramme. Angeschlossen an diese Großrechner waren lediglich Terminals mit geringer oder gar keiner Rechenleistung. So mussten sich Softwareentwickler keine Gedanken machen über die Verteilung von Systemen. Später kam dann die Zeit der „Personal Computers“. Nun liefen ganze Systeme für einzelne Benutzer auf einzelnen Rechnern, den Workstations. Die erledigten lediglich Aufgaben wie Textverarbeitungen, Kalkulationen, Grafikerstellung und Ähnliches. Auch in dieser Zeit gab es erst einmal keinen Anlass, verteilte Systeme zu entwickeln, denn die PCs waren meist autarke Rechner, die zumindest ihre grundsätzlichen Aufgaben für sich lösten.

In einigen Bereichen fing man allerdings an, Rechnersysteme zu verbinden. Dies geschah zu Beginn beim Militär, um wichtige Funktionalität ausfallsicher zu gewährleisten. So entstand das „Internet“. Allerdings waren die Übertragungsmöglichkeiten sehr begrenzt und die Softwaresysteme kooperierten auf sehr niedrigem Grad.

Das Internet wurde mit der wesentlichen Verbesserung der Übertragungs- und Rechnertechniken ausgebaut und von Militär, den Universitäten, Regierungen, Konzernen und Unternehmen und später auch von privaten Interessenten genutzt. Seitdem tauschen unzählige Rechner ihre Daten miteinander aus, PCs bekommen Dienste von Servern zur Verfügung gestellt und kommunizieren miteinander.

Seit etwa Mitte der neunziger Jahre erkennt man deutlich, dass die Anforderungen an die Systeme immer komplexer und massiver werden. Einfache Server-Lösungen zur Bewältigung der Anfragen aus dem Netz der Netze reichen in eher seltenen Fällen aus. Software wird immer komplexer und lässt sich auf einem oder ein paar Rechnern nicht mehr ausführen. Die Entwicklung und der Einsatz von sehr großen und leistungsstarken Servern, die den Anforderungen teilweise gerecht werden könnten, sind zu teuer bzw. zu unwirtschaftlich. Die Lösung dieser Probleme sieht man heute mehr denn je in dem Einsatz von Verteilten Systemen. Sie zeichnen sich aus durch eine Vielzahl wünschenswerter Charakteristika[SOM]:

- Ressourcenteilung (Resource sharing)
- Offenheit (Openness)
- Parallelität (Concurrency)
- Skalierbarkeit (Scalability)
- Fehlertoleranz (Fault tolerance)
- Transparenz (Transparency)

Bei der sich heute stark durchgesetzten objektorientierten Softwareentwicklung bietet es sich an, diese gekapselten Softwareeinheiten in den Systemen zu verteilen und deren Methoden und Schnittstellen über die Grenzen des jeweiligen Hosts hinweg zur Verfügung zu stellen. Damit ergeben sich aber eine Vielzahl von Problemen: die Zugriffe müssen gesteuert werden, Schnittstellen müssen vereinheitlicht werden, Objekte müssen veröffentlicht werden, u.v.a.m. Diese wesentlichen Fragen zur Lösung solcher Probleme sollen in dieser Arbeit grundsätzlich angegangen werden, denn das Prinzip der Verteilten Objekte ist aktueller denn je. Standards, Modelle, Beschreibungs- und Programmiersprachen haben sich nach und nach mit diesem Thema beschäftigt und werden entwickelt, damit sie von Entwicklern bei der Implementierung von verteilten Systemen bzw. Verteilte-Objekte-Systemen genutzt werden können. Lösungsansätze komplexer Probleme können es sich heute nicht mehr leisten, die Vorteile von verteilten Objekten außer Acht zu lassen und nicht zu nutzen.

1.2 Aufgabenstellung

Im Rahmen dieser Arbeit sollen wesentliche Aspekte von verteilten Systemen erläutert werden. Weiterhin soll dann die Idee von verteilten Softwaresystemen angegangen werden. Der Focus soll auf Verteilte-Objekte-Architekturen liegen, wie sie heute beschrieben, entwickelt und angewandt werden. Die drei Lösungsstrategien CORBA, .NET und RMI sollen erklärt, ihre Architektur, ihre grundsätzlichen Funktionen und Dienste herausgestellt werden. Schließlich gilt es die drei verschiedenen Arten von Systemen anhand von wichtigen Kriterien zu vergleichen.

1.3 Übersicht über die Kapitel

Diese Arbeit ist folgendermaßen gegliedert:

Kapitel 2 gibt eine kurze Einführung in die Grundlagen von Verteilten Systemen. Hier werden sukzessiv Begriffe verschiedener Verteilter Systeme eingeführt und erklärt. Des Weiteren gibt es eine allgemeine Einführung in objektorientierter Middleware.

Das dritte Kapitel befasst sich mit der CORBA-Architektur. Hier wird über die Entwicklung und Herkunft gesprochen, die Struktur, die Architektur, die Komponenten, Dienste und Schnittstellen erläutert. Die Realisierung wichtiger Dienste wird intensiver behandelt.

Analog zu Kapitel 3 wird im vierten Kapitel die Architekturvariante .NET behandelt.

…

Die Realisierung vom Verteilte-Objekte-System mittels Java – RMI wird im fünften Kapitel beschrieben.

…

Kapitel sechs ist eine kurze, vergleichende Zusammenfassung der zuvor im Einzelnen behandelten drei Middleware-Typen. Hierauf folgt das abschließende Resümee.

2 Grundlagen

2.1 Client-Server-Architektur

Die Client-Server-Architektur beschreibt die Beziehung und Aufgaben von Computern im Netzwerk. In einer Client-Server Architektur ist eine Anwendung ein Satz von Diensten, die von so genannten Servern zur Verfügung gestellt werden und von mehreren Clients benutzt werden. [SOM] Der Klient muss wissen, dass es die verfügbaren Server gibt, aber über die Existenz von anderen Klienten ist er gewöhnlich nicht informiert. Der Client und der Server sind grundsätzlich verschiedene Prozesse.
Mit „Client“ und „Server“ sind zunächst einmal immer die Prozesse, nicht die Prozessoren gemeint. D.h. wir wollen uns hier auf die logische Client-Server-Architektur konzentrieren. Das Design eines Client-Server-Systems sollte die logische Struktur der Anwendung widerspiegeln. Eine Anwendung kann in 3 Schichten strukturiert werden: „Presentation Layer“, „Application Processing Layer“ und „Data management Layer“. „Presentation Layer“ ist der Programmteil, der sich mit allen Benutzerinteraktionen befasst, „Application Processing Layer“ beinhaltet die gesamte Logik der Anwendung und „Data management Layer“ ist zuständig für alle Datenbankoperationen. Diese drei Schichten sind in zentralisierten Systemen nicht besonders voneinander abgegrenzt. Doch in Verteilten Systemen sollten sie wohl unterschieden werden, denn es ist möglich, jede dieser Schicht auf verschiedene Rechner auszuführen.

Die einfachste Client-Server-Architektur wird zweireihige Client-Server-Architektur genannt. (two-tier client-server architecture). Von dieser Art gibt es im Wesentlichen 2 verschiedene Formen: das Thin-client-Modell und das Fat-client-Modell. Im ersten Fall ist der Client-Rechner lediglich für die Darstellung zuständig, Logik- und Datenschicht sind auf dem Server-Rechner installiert. Bei dem sogenannten „Fat-client-Modell“ ist der Server lediglich für die Datenverwaltung zuständig, die Logik und die Interaktionen werden auf dem Client-Rechner zur Verfügung gestellt.
Der Nachteil des erstgenannten Modells liegt im Wesentlichen darin, dass sowohl der Server, als auch das Netzwerk sehr stark belastet werden. Im Gegensatz dazu wird die Prozessorleistung des Client-Rechners, die heutzutage beim PC recht hoch ist, nicht genutzt und liegt brach. Weiterhin ist das System kaum skalierbar, da der Server 2 Funktionen vereint.
Beim „Fat-client-Modell“ wird zwar die Rechenlast effektiver verteilt, aber das Systemmanagement ist wesentlich komplexer, Wenn die Anwendungssoftware verändert wird, muss sich auf jedem Client neu installiert werden, was zu beträchtlichem Aufwand und Kosten führen kann.

Ein Ausweg aus dieser Misere ist die dreireihige Client-Server-Architektur. Hier sind die Datenschicht, die Logikschicht und die Präsentationsschicht logisch in 3 separate Prozesse getrennt. Logik und Datenverwaltung können zwar noch immer auf einem Server installiert sein, sind aber recht leicht auf separaten Rechnern aufzuteilen, wenn es erforderlich ist. Ein gutes Beispiel für diese Art von Architektur ist ein Homebanking-System. Die Client-Rechner sind die Heim-PCs der Kunden, die über einen Internetbrowser mit dem Banksystem kommunizieren. Software-Verteilung o.ä. spielt hier kaum eine Rolle. Je nach Bedarf können deren Anfragen von unterschiedlich vielen Web-Servern beantwortet werden. Diese Anzahl lässt sich problemlos anpassen. Die Datenbanken der Benutzer- und Konteninformationen liegen auf Datenbankservern. Auch deren Anzahl kann einfach skaliert werden, wenn die Auslastung sich ändert. Die Webserver greifen somit z.B. per SQL auf verschiedene Datenbankserver zu, um die Anfragen, die vom Klienten kommen, zu beantworten und Aktionen bzw. Transaktionen durchzuführen.

2.2 Verteilte Objekte Architekturen

Im Client-Server-Modell sind Klienten und Server unterschiedlich, Clients nutzen Dienste von Servern und nicht von anderen Clients. Server können zwar Dienste von anderen Servern nutzen, aber sie rufen diese nicht von Clients ab. Klienten müssen wissen, welche Dienste auf welchen Servern zur Verfügung stehen und müssen auch wissen, wie sie diese aufrufen. Dieses Modell ist geeignet für viele Anwendungen, doch es schränkt die Flexibilität von Systemdesignern ein, da sie schon im Vorfeld entscheiden müssen, wo Dienste zur Verfügung gestellt werden müssen. Sie müssen auch die Skalierbarkeit mit einplanen, denn wenn mehr Klienten auf wenigen oder gar nur einen Server zugreifen, muss dieser gegebenenfalls durch weitere ergänzt werden.

Ein Ausweg ist es, wenn bei der Systemarchitektur kein Unterschied zwischen Client und Server gemacht wird, sondern von einem System verteilter Objekte ausgegangen wird. Der Gedanke einer Verteilte-Objekte-Architektur basiert darauf, dass Systemkomponenten Objekte sind, die über Schnittstellen Dienste anbieten. Andere Objekte können diese Dienste aufrufen, aber auch selbst Dienste für wiederum andere Objekte anbieten. Diese Objekte können über eine Vielzahl von Computern in einem Netzwerk kommunizieren. Sie bedienen sich einer Art „Software Bus“. Dieser Bus stellt wiederum Dienste zur Verfügung, mit Hilfe derer die Objekte kommunizieren können. Es ist quasi ein Analogon zum Hardwarebus z.B. in PCs, wo Komponenten hereingesteckt und wieder entfernt werden können. Die Komponenten bedienen sich der Funktionalität des Busses, um mit den anderen Komponenten zu kommunizieren. Vorteile dieses Modells sind vor allem:

- Es erlaubt dem Systemdesigner, erst später über das Wo und das Wie von Diensten, die angeboten werden sollen, zu entscheiden

- Es ist eine offene Architektur, die es erlaubt, neue Ressourcen hinzuzufügen, wenn dies nötig ist. Durch geeignete Standards können Objekte verschiedener Programmiersprachen miteinander kooperieren.
- Das System ist flexibel und skalierbar. Verschiedene Instanzen eines Systems können den gleichen Dienst mit verschiedenen Objekten anbieten. Es ist auch möglich, Objekte zu duplizieren, um wachsenden Anfragen gerecht zu werden, ohne das System zu stören.

2.3 Computer Netzwerke

Das ISO-OSI-Referenzmodell der Internationalen Standardisierungs-Organisation (ISO) wurde ab 1977 als Grundlage für die Bildung von Kommunikationsstandards entworfen. Ziel von OSI ist die Kommunikation in heterogener Umgebung, d.h. insbesondere zwischen verschiedenen heterogenen Rechnersystemen darzustellen. Im OSI-Referenzmodell wird die Kommunikation durch Einheiten bzw. Elementen realisiert, die einen festen Platz und eine definierte Aufgabenstellung haben. Elemente mit vergleichbaren Funktionen können auf unterschiedlichen Systemen residieren und werden in so genannten Schichten angeordnet. Jede Schicht beschreibt die Funktionen der Elemente. Da dieses Modell auch Grundlage der Kommunikation von Verteilten Systemen ist, führen wir es an dieser Stelle kurz an, ohne auf die Details einzugehen.

Das Modell besteht aus 7 Schichten. Jede Implementierung einer Schicht auf einem System – auch Instanz genannt – kommuniziert mit der entsprechenden Gegenstelle eines anderen, entfernten Systems. Folgende Schichten bauen von unten nach oben aufeinander auf:

1. Bitübertragungsschicht (Physical Layer). Sie befasst sich vor allem mit physikalische, elektrische und optische Eigenschaften aller Geräte und Komponenten
2. Sicherungsschicht (Data Link Layer). Sie implementiert die einfache Verbindung zwischen zwei Hosts und stellt gesicherte Übertragungskanäle für so genannte Frames zur Verfügung.
3. Vermittlungs- bzw. Netzwerkschicht (Network Layer). Sie implementiert die Verbindungen von vielen Knoten in einem Netzwerk und befasst sich mit dem Vermitteln, dem Routen von Paketen. Sie ist also paketorientiert und verbindungslos.
4. Transportschicht (Transport Layer) Sie ermöglicht die Verbindung zwischen einzelnen Prozessen in verschiedenen Endsystemen. Entsprechende Protokolle sind verbindungslos oder verbindungsorientiert.

   Verbindungsorientierte bieten Operationen zum Öffnen und Schließen derselben an. Den höheren Schichten werden Mechanismen zum Datenaustausch durch Byte-Ströme angeboten. Das heute bedeutendste Protokoll dieser Art im Datenverkehr ist das auf IP (Internet Protocol) aufsetzende TCP (Transmission Control Protocol).

   Verbindungslose Implementierungen bieten lediglich die Möglichkeit, Nachrichten mit fester Länge zwischen den verteilten Hosts in Form von Datagrammen zu senden.

Wenn höheren Schichten nun ein hohes Niveau von Zuverlässigkeit angeboten werden soll, muss die Transportschicht zuverlässigen Transport gewährleisten, sogar dann, wenn darunterliegende Schichten unzuverlässig sind. Das auf IP aufsetzende Gegenstück zu TCP heißt UDP (User Datagram Protocol).

Diese vier Schichten gehören zur Gruppe der transportorientierten Schichten. Auf diesen Schichten setzen dann die drei noch folgenden anwendungsorientierten Schichten auf:

5. Kommunikationssteuerungsschicht (Session Layer) Sie stellt Verbindungen zwischen Komponenten verteilter Systeme her und hält sie aufrecht. Den Prozessen werden Dienste zur Organisieren und Synchronisieren ihres Dialogs während ihrer Sitzung angeboten.
6. Darstellungsschicht (Presentation Layer) Sie löst Unterschiede in der Darstellung von Datenstrukturen auf und ermöglicht so den Transport von komplexen Daten.
7. Anwendungsschicht (Application Layer) Sie ist zuständig für die Verteilten Komponenten und deren Interaktionen und stellt anwendungsorientierte Grunddiensten mit entsprechenden Datenstrukturen und Protokollen unmittelbar zur Verfügung.

2.4 Grundsätzliches zu Middleware

Wenn Anwendungsentwickler von Verteilten Systemen direkt auf der Transportschicht aufsetzend implementieren müssten, dann hätten sie die Kommunikation - sprich das Schreiben in und das Lesen aus Byte-Strömen (TCP) oder Nachrichten-Datagrammen mit fester Länge (UDP) - zwischen den Komponenten mit folgenden Problemen selbst zu lösen:

- Komplexe Parameter wie Zeichenketten (Strings) und Arrays müssen in Byte-Sequenzen umgewandelt werden.
- Verschiedene Kodierungen der Datentypen müssen jeweils umgesetzt werden.
- Objektzeiger (Object references), zusammengesetzt aus Internet-Domänennamen, Portnummern und zusätzlichen Adressinformationen müssen implementieren.
- Komponentenaktivierung muss implementiert werden, damit ein Server antwortet, obwohl die Komponente gerade nicht in einem Prozess des Betriebssystems läuft.
- „Type Safety“ muss realisieren werden. Parameter und andere Bezeichner müssen fehlerlos abgestimmt werden.
- Die Implementierung der Synchronisation ist recht aufwendig und muss realisiert werden, besonders um mit großen Verzögerungen fertig zu werden.
- Fehlermechanismen zur Steigerung der Dienstqualität sind manchmal nötig, wenn die Zuverlässigkeit der Netzwerkschicht nicht ausreicht.

Glücklicherweise müssen sich die Anwendungsentwickler um oben genannte Dinge nicht besonders kümmern, wenn sie ihre Implementierungen auf so genannter Middleware aufsetzen. Middleware vereinfacht die Konstruktion von Verteilten Systemen, indem sie bereits die Darstellungs- und Kommunikationssteuerungsschicht implementiert. Sie übersetzt komplexe Datenstrukturen in übertragbare Form, löst Heterogenität auf, löst logische Adressen in Internet-Domänen-Namen und Portadressen auf und synchronisiert Server- und Clientobjekte und noch einiges mehr.

Die als erste entwickelte Middleware ist die transaktions-orientierte Middleware (transaction-oriented middleware) und wird für Datenbanksysteme zur Bewerkstelligung von „fault-tolerance“ und „load balancing“ genutzt werden.
Eine Weiterentwicklung ist die nachrichten-orientierte Middleware (message-oriented middleware). Anfragen mit zugehörigen Dienst-Parametern und das produzierte Ergebnis werden per Nachricht gesendet, Server und Client sind bereits entkoppelt. Zur Sicherung der Übertragung speichert man die Nachrichten in eine Warteschlange. Wenn der Empfänger nicht empfangen kann, wird die Nachricht später gesendet, wenn sie verloren geht, wird sie wiederholt geschickt.
Die „Remote Procedure Calls“ (RPC) wurden von Sun Microsystems entwickelt. RPCs sind Aufrufe von Prozeduren über die Grenzen von Computern hinweg. Sun implementierte RPC im eigenen Betriebssystem Sun OS (Unix), später beschrieb DCE (=Distributed Computing Environment) einen von Programmiersprachen unabhängigen Standard. RPCs erledigen die Auflösung von Datenheterogenität und das Transferieren von komplexen Datenstrukturen in Blöcken oder Byte-Strömen, das „marshalling“. Die Wiederherstellung von komplexen Datenstrukturen aus Sequenzen oder Blöcken, „unmarshalling“ genannt, wird ebenfalls von den RPCs geleistet.
RPCs sind eine Art Basis für objektorientierte Middleware. Bei dieser Art von Middleware ist z.B., dass es immer eine Art Schnittstellendefinitionssprache gibt. Diese unterstützt das Konzept von Objekttypen als Parameter. Rückgabewerte von Objekten können andere Objekte sein. Klassenhierarchien und Vererbung, wie es bei objektorientierter Softwareentwicklung angewandt wird, wird unterstützt. Auch die Fehlerbehandlung wird aktiv unterstützt, sodass die Anwendungsschicht über die Art des Fehlers informiert wird, statt nur eine Null-Pointer-Exception weiterzugeben, wenn z.B. Server-Objekt nicht gefunden werden kann. Die Darstellungsschicht von objektorientierter Middleware (OO-Middleware) verhält sich ähnlich wie die von RPCs, doch müssen hier auch die Repräsentanten der Objektreferenzen abgebildet und an niedere Schichten weitergereicht werden. Die Sitzungsschicht ist u.a. für das „Mapping“ von Objektreferenzen zu Hostnamen und –Ports zuständig. Weiterhin ist diese Schicht auch für das Starten und Aktivieren von Servern bzw. Serverobjekten zuständig. Dies erledigt meist der so genannte Objektadapter. Synchronisierung wird ebenfalls von der Sitzungsschicht der OO-Middleware gewährleistet.

3 CORBA

3.1 Einführung

Common Object Request Broker Architecture (CORBA), ein Standard der Object Management Group (OMG), definiert eine herstellerunabhängige Architektur und Infrastruktur für Verteilte-Objekte-Systeme. OMG ist ein nicht-profitorientiertes Konsortium und wurde 1989 von elf Firmen gegründet, mittlerweile gibt es etwa 800 Mitglieder. Das von Ihr entwickelte CORBA ist kein Softwareprodukt, es ist eine Spezifikation und ein Modell zur objektorientierten Systementwicklung. Ursprünglich definierte die OMG die Object Management Architecture (OMA), die die Interoperabilität von Software-Anwendungen beschreibt. Eines der Ziele war es, ein Standard zu spezifizieren, wie Objekte im Netz verteilt werden. Dies war die Grundlage für die Entstehung von CORBA. Die OMA besteht im Wesentlichen aus den folgenden Hauptkomponenten:

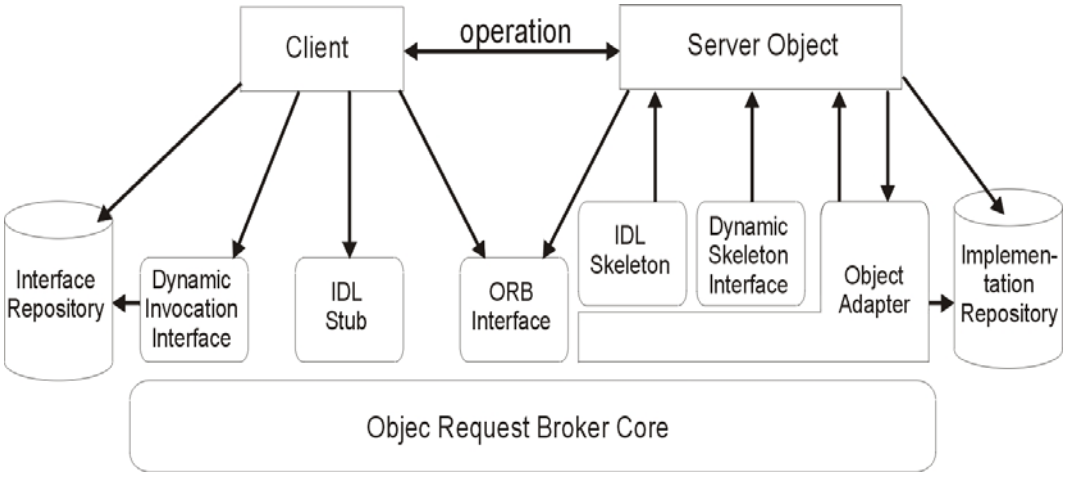
- Der Object Request Broker (ORB) ist ein Softwarebus und für die Kommunikation und den Datenaustausch zwischen den einzelnen CORBA-Objekten verantwortlich.
- CORBA-Services definieren Dienste (Services), die auf der Systemebene des ORBs agieren. Beispiele hierfür sind Security, Naming, Transaction oder Persistent Service.
- CORBA-Facilities definieren Services auf der Anwendungsebene, die branchenübergreifend, branchenspezifisch oder anwendungsspezifisch sein können.
- Geschäftsobjekte (Business Objects) beschreiben Objekte und Anwendungen, die in der realen Welt vorkommen (Flugzeug, Bankkonto oder Auto).

CORBA-Objekte können in den Programmiersprachen realisiert werden, die von den CORBA-Anbietern unterstützt werden, seien es in C, C++, Java, Ada, Cobol oder Smalltalk. Weiterhin sind diese Objekte unabhängig von ihren Plattformen, auf denen sie zur Verfügung gestellt werden bzw. auf denen zugegriffen wird, solange die CORBA-Spezifikation eingehalten wird. Sprach- und Systemunabhängigkeit werden durch die Beschreibung von Interfaces der Objekte über die Beschreibungssprache Interface Definition Language (IDL) ermöglicht. IDL erlaubt alle CORBA-Objekte auf die gleiche Art zu beschreiben. Die einzige Anforderung besteht in der Existenz einer Schnittstelle zwischen der IDL und der jeweiligen Programmiersprache.

Des Weiteren wird von der OMG eine CORBA-Inter-ORB-Architektur beschrieben. Zwei wichtige Teile der Architektur sind das General Inter-ORB Protocol (GIOP) und das Internet Inter-ORB Protocol (IIOP), Protokolle, die der Kommunikation zwischen verschiedenen ORB-Implementierungen dienen.

3.2 Object Request Broker (ORB)

Die nachfolgende Abbildung zeigt die wesentlichen Komponenten der CORBA-ORB-Architektur.



CORBA-ORB-Architektur

Der Object Request Broker (ORB) ist der Kommunikationsbus zwischen CORBA-Objekten, der einen Mechanismus anbietet, um Anfragen des Clients an die entsprechenden Object Implementations weiterzuleiten. Hinter einem einfachen Methodenaufruf des Clients verbirgt sich eine komplexe Netzwerkkommunikation, die vom ORB gehandhabt wird. Eine Anfrage des Clients, bestehend aus dem Methodenaufruf und den Parametern, wird in einem binären Strom umgesetzt (marshaling) und über das Netzwerk an den Server geschickt. Die Informationen werden auf der Server-Seite wieder decodiert (unmarshaling) und die gewünschte Operation ausgeführt. Rückgabewerte werden auf die gleiche Weise wieder über den ORB an den Client gesendet. Zusammengefasst hat der ORB die Aufgabe, die entsprechende Objektimplementierung zu finden, sie zu aktivieren, falls erforderlich, die Anfrage an den Objekt zu leiten und entsprechende Rückgabewerte wieder an den Client zurückzugeben.

Die CORBA-ORB-Architektur auf der Client-Seite besteht aus folgenden Komponenten:

- CORBA IDL Stubs: Sie stellen die Schnittstellen zu den Objektdiensten bereit. Aus der Sicht des Clients verhält sich der Stub wie ein Proxy-Objekt für ein entferntes Server-Objekt. Die Dienste werden mit Hilfe von IDL definiert, ihre zugehörigen Client-Stubs von einem IDL-Compiler generiert. Ein Stub beinhaltet Code für die Umsetzung der Methodenaufrufe mit ihren Parametern in einfache Nachrichtenströme, die an den Server gesendet werden. Die zugrundeliegenden Protokolle oder Dinge wie Datenanordnungen bleiben für den Client vollkommen transparent, der die Dienste als einfache Methodenaufrufe auffasst.

- Dynamic Invocation Interface (DII): Dieses erlaubt die vom Server bereitgestellten Dienste zur Laufzeit zu finden und diese zu nutzen. CORBA definiert hierfür Standard-APIs für die Suche der Metadaten, die die Server-Schnittstellen definieren, für die Generierung der Parameter, für das Absetzen des Remote-Aufrufs und für das Empfangen der Ergebnisse. Dynamische Methodenaufrufe erfordern keine Client-Stubs, da sie erst zur Laufzeit zusammengesetzt werden.
- Interface Repository: Dies ist eine verteilte Laufzeit-Datenbank, die maschinenlesbare Versionen der IDL-definierten Schnittstellen, die Metadaten, beinhaltet. Ein API erlaubt, die Metadaten zu lesen, abzuspeichern und zu verändern.

Auf der Server-Seite sind folgende Komponenten zu finden:

- CORBA IDL Skeletons: Sie stellen das Gegenstück zu den Client-Stubs dar. Sie stellen die statischen Schnittstellen für jeden vom Server unterstützten Dienst bereit. Sie beinhalten Code, um eingehende Nachrichten in normale Methodenaufrufe umzusetzen und diese auszuführen. Rückgabewerte werden wieder über die Skeletons in einen binären Strom umgewandelt und an den Client zurückgeschickt. Wie die Stubs werden die Skeletons vom IDL-Compiler aus IDL-Interfaces generiert.
- Dynamic Skeleton Interface (DSI): Es ist das Gegenstück zum DII auf der Client-Seite. Es erlaubt eingehende Methodenaufrufe von einem Client, für die keine entsprechenden IDL-kompilierten Skeletons existieren, dynamisch zu bearbeiten. Seine Aufgabe besteht darin, ankommende Anfragen von einem Client zu analysieren und das dafür vorgesehene Objekt zu finden und den Methodenaufruf abzusetzen.
- Object Adapter: Er nimmt Dienstanfragen der Clients entgegen und aktiviert die entsprechenden Server-Objekte. Er bildet die Laufzeitumgebung der Server-Objekte, übergibt ihnen die Anfragen und weist ihnen Objektreferenzen zu. Der Object Adapter registriert die von ihm unterstützten Klassen und Objekte im Implementation Repository.
- Implementation Repository: Es bildet eine Laufzeitdatenbank, die Informationen über die von einem Server unterstützten Klassen und Objekte enthält. Weitere Informationen, die mit der Implementierung von ORBs zusammenhängen, z.B. Trace-Informationen, Prüfprotokolle, Sicherheit und andere administrative Daten, sind dort ebenfalls zu finden.

3.3 OMG Interface Definition Language (IDL)

Eine einheitliche Schnittstellenbeschreibung der zu verteilenden CORBA-Objekte ist erforderlich, um Sprach- und Betriebssystemunabhängigkeit zu gewährleisten. Diese Beschreibungssprache ist im CORBA-Standard als IDL (Interface Definition Language)

definiert, die an die Syntax von C/C++ angelehnt ist. Operationen und Attribute, die von einem Server zur Verfügung gestellt werden, sind in einem Interface mittels IDL-Syntax beschrieben. Aus einem vollständig beschriebenen IDL-Interface werden mit einem IDL-Compiler Stub-Klassen für die Clients und Skeleton-Klassen für den Server für die jeweilige in der zu realisierenden Programmiersprache erzeugt. Ein IDL-Interface beschreibt lediglich die Schnittstellen der zu veröffentlichenden Dienste und Komponenten des Servers und enthält keine Implementierungsdetails. Dadurch wird ermöglicht, beispielsweise in Java erstellten Clients auf die in C++ realisierten Server-Objekte zuzugreifen. IDL-Compiler werden mit den ORB-Produkten mitgeliefert

Die unten vorliegende IDL-Beschreibung ist ein Überblick über die wesentlichen Merkmale der CORBA-IDL:

```
module <identifier> {                          // definiert einen Namensraum
    <type declarations>;
    <constant declarations>;
    <exception declarations>;

    interface <identifier> [:<inheritance>] {  // definiert ein CORBA-Interface
        <type declarations>;
        <constant declarations>;
        <attribute declarations>;
        <exception declarations>;
        [<op_type>] <identifier>(<parameters>) // definiert eine Operation
            [raises exception][context];
        ...
    };
    ...
};
```

Hier eine kurze Beschreibung der wichtigsten Elemente, die eine IDL-Beschreibung ausmachen:

Ein Modul stellt einen Namensraum bereit, um mehrere Schnittstellen zu gruppieren. Es wird durch das Schlüsselwort **`module`** eingeleitet, dessen Name sich aus einem oder mehreren Bezeichnern zusammensetzt. Dadurch kann eine zusätzliche Ebene in der Hierarchie im IDL-Namensraum eingeführt werden.

Datentypen grenzen die möglichen Werte der Parameter, Attribute, Ausnahmen und Rückgabewerte ein, die sie annehmen können. Diese Datentypen werden als CORBA-Objekte bezeichnet, die Sprach-, Betriebssystem- und ORB-übergreifend verwendet werden. CORBA unterscheidet zwischen einfachen und konstruierten Typen. Zu den einfachen CORBA-Typen zählen **`short`**, **`long`**, **`unsigned short`**, **`unsigned long`**, **`float`**, **`double`**, **`char`**, **`boolean`**, **`octet`**, **`string`**, **`enum`** und **`any`**. Die konstruierten Typen sind **`struct`**, **`array`**, **`union`** und **`sequence`**. Der Typ **`struct`**, ähnlich einer C-Struktur, erlaubt, mit Hilfe von **`typedefs`** jeden komplexen Datentyp zu erstellen. Mit dem Typ **`sequence`** wird ein Array von Objekten variabler Länge definiert. Der Typ **`any`** kann jeden möglichen IDL-Datentyp repräsentieren - einen einfachen, einen konstruierten oder eine

Objektreferenz. Jeder IDL-Datentyp von CORBA wird mit Hilfe von IDL-Compilern auf einen nativen Datentyp abgebildet.
Interfaces definieren Operationen und Attribute. Eine Schnittstelle ist eine rein deklarative Beschreibung der Services, die von einem Client genutzt werden können und enthalten keine Details über ihre Implementierung. Schnittstellen können von einer oder mehreren Schnittstellen abgeleitet werden.
Operationen kennzeichnen Dienste, die ein Client aufrufen kann. Die IDL definiert die Signatur einer Operation, bestehend aus ihren Parametern und Rückgabewert. Ein Parameter kann vom Client zum Server (`in`), vom Server zum Client (`out`) oder in beiden Richtungen (`inout`) übergeben werden. Ein Parameter besitzt einen Datentyp. Der Typ des Rückgabewertes wird durch `op_type` festgelegt. Fehlerfälle können durch das Werfen von Exceptions angezeigt werden. Ein optionaler Context-Ausdruck beinhaltet eine Menge von Werten für Attribute, die den Kontext eines Clients beschreiben. Er ermöglicht einem Client, Informationen, die seine lokale Umgebung beschreiben, an den Server weiterzugeben.
Attribute beschreiben den Zustand eines Interfaces. Für jedes Attribut werden jeweils eine Getter- (wenn das Attribut nicht als read-only deklariert ist) und eine Setter-Methode vom IDL-Compiler erzeugt.

3.4 Schnittstellen des ORBs: GIOP & IIOP

In einer mit CORBA entwickelten Anwendung können mehrere verschiedene ORB-Implementierungen miteinander über ein Netzwerk kommunizieren. Mit Hilfe der ORBs werden die entsprechenden Anfragen des Clients an den Server weitergeleitet. Dieser führt den gewünschten Service aus und gibt die Rückantwort wieder über die ORBs an den Client zurück.

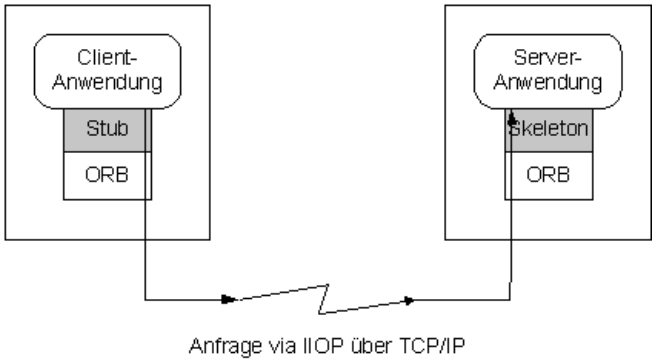


Abbildung: Client/Server-Kommunikation über ORBs

Hinter diesem Aufruf steckt eine komplexe Netzwerkkommunikation, die über die von der OMG spezifizierte CORBA-Inter-ORB-Architektur beschrieben ist. Das General Inter-ORB Protocol (GIOP) und das Internet Inter-ORB Protocol (IIOP) sind zwei wichtige Protokolle dieser Architektur. Das General Inter-ORB Protocol (GIOP) spezifiziert eine Menge von Nachrichtenformaten und Datendarstellungen für die Kommunikation zwischen den

ORBs. Die in IDL definierten Datentypen werden als Common Data Representation (CDR) in eine einfache Nachrichtendarstellung für Netzwerke abgebildet. GIOP definiert außerdem ein Format für Interoperable Object References (IORs). Ein ORB muss eine IOR von einer Objektreferenz erzeugen, wann immer eine Objektreferenz über ORBs weitergegeben wird. Das Internet Inter-ORB Protocol (IIOP) legt fest, wie GIOP-Nachrichten über ein TCP/IP-Netzwerk ausgetauscht werden. Das IIOP ermöglicht daher, die Busarchitektur und die Technologien des Internets selbst als Basis für eine Inter-ORB-Kommunikation zu verwenden. Daneben können einzelne ORBs aber noch weitere Environment-Specific-Inter-ORB-Protocols (ESIOPs) unterstützen.
Entwickler von verteilten Anwendungen werden sich selten mit der Kommunikation der ORBs auseinandersetzen müssen. Interoperabilität wird zwischen ORB-Anbietern garantiert, und nicht zwischen Anbieter und Entwickler.

3.5 Implementierungsbeispiel eines Servers

Im folgenden wollen wir ein ganz simples Beispiel betrachten, bei dem eine Anwendung, die zwei Zahlen addieren und das Ergebnis zurückgeben soll, implementiert wird.
Eine mögliche IDL-Datei sieht so aus:
Add_example.idl:

```
module Add_example {
   interface Adder {
        long generator(in long first, int long second);
   };
};
```

Mit Hilfe des IDL-Compilers von Java (SDK 1.4.1) werden folgende Dateien erstellt:

Adder.java	Interface-Deklaration
_AdderStup.java	Client-Stub, implementiert Interface „Adder“
AdderOperations.java	Interface zur Deklaration der Methoden
AdderPOA.java	Objektadapter, implementiert Interface „AdderOperations“
AdderHelper.java	Hilfsklasse zur Stub/Skeleton-Implementierung
AdderHolder.java	Hilfsklasse zur Stub/Skeleton-Implementierung

Diese Dateien müssen noch erstellt werden:

AdderImpl.java	Eigentliche Serveranwendung, abgeleitet von AdderPOA
AdderServer.java	enthält Main-Methode, zuständig für die ORB-Kommunikation
AdderClient.java	enthält Main-Methode, Implementierung des Client-Programms

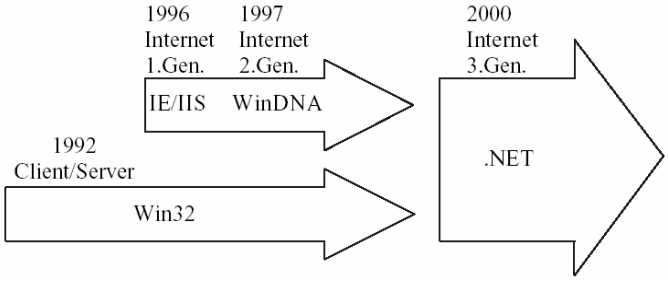
Hier haben wir abschließend die Beispiel-Datei AdderServer.java, dessen Struktur beinahe immer die gleiche ist:

```
//Import der Packages der CORBA-Bibliothek:
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.PortableServer.*;

public class AdderServer {
   public static void main(String args[]) {
       try{

              //Der ORB wird initialisiert
              ORB orb = ORB.init(args, null);

              //Das Serverobjekt, was von Objektadapter abgeleitet ist,
              //wird instanziert und mit dem ORB registriert:
              AdderImpl impl = new AdderImpl(orb);

              //Der Objektadapter (hier:POA) muss aktiviert werden:
              POA rootpoa = POAHelper.narrow(
                orb.resolve_initial_references("RootPOA"));
              rootpoa.the_POAManager().activate();

              //Man  erhält eine Referenz des Serverobjekts
              org.omg.CORBA.Object ref =
                rootpoa.servant_to_reference(impl);
              Adder href = AddHelper.narrow(refadd);

              //Hier wird (der importierte) COS Name Service verwendet,
              //“NameService“ spricht den Naming Server „orbd“ an,
              //der bereits gestartet sein muss
              org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
              NamingContextExt ncRef =
                NamingContextExtHelper.narrow(objRef);


              //Nun wird die Objektreferenz mit dem Namen verbuden
              NameComponent path[] = ncRef.to_name( „Adder“ );
              ncRef.rebind(path, href);

              // Der Server wird gestartet und wartet auf Client-Anfragen.
              orb.run();

       } catch (Exception e) {...}
   }
}
```

4 .NET

4.1 .NET Einführung

Microsoft .NET ist die neueste Weiterentwicklung der Integration von verteilten Objekten in das Windows Betriebssystem. 1992 begann Microsoft mit der Einbindung von Client/Server Techniken in das Microsoft Betriebssystem. Parallel dazu wurde 1996 der Internet Explorer und der Internet Information Server vorgestellt, welche laut Microsoft die 1. Generation von Internet Software darstellt. Die 2. Generation von Internet Software bildete dann Windows DNA (Distributed Internet Application Architecture). Die wichtigsten Bestandteile von Windows DNA sind COM und DCOM, bzw. deren Nachfolger COM+. Diese zwei Entwicklungslinien wurden dann 2000 zusammengeführt in .NET (siehe Grafik).



Microsoft .NET setzt sich aus 4 Hauptbestandteilen zusammen:

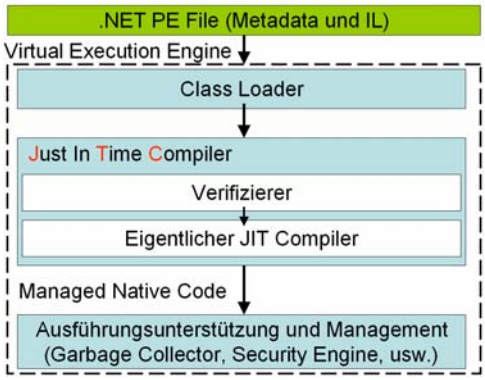
- Framework und Tools
- Building Blocks Services
- Enterprise Server (Für die Webservices)
- Mobile Devices

Unter „Building Blocks Services“ versteht Microsoft fertige Programmteile, die direkt mit in die eigene Website einfließen können. Die Infrastruktur für die Web-Dienste stellt der Enterprise Server und die Mobile Devices sind Geräte die mit Windows CE oder Windows XP Mobile ausgestattet sind.

Uns an dieser Stelle interessiert eher das .NET Framework mit seinen Komponenten, deshalb werden wir nicht weiter auf die anderen o. g. Bestandteil eingehen.

Das .NET Framework setzt sich aus folgenden Komponenten zusammen:

- Common Language Runtime (CLR)
- Common Type System (CTS)
- Entwicklungsumgebungen: Visual Studio.NET, .NET SDK
- Web Services
- .NET Remoting



Die Common Language Runtime ist, ähnlich wie bei Sun JAVA, die Laufzeitumgebung für .NET Anwendungen. Auf sie setzten unterschiedliche Programmiersprachen auf und sie ist für die Sicherheit und Kompilierung von Programmen zuständig.

Der .NET fähige Compiler (z.B. im Visual Studio .NET) muss hierbei ein sog. .NET PE File erzeugen, damit die CLR die Klassen laden kann, das Objekt verifizieren kann und dann „Native Code“, also Code den das Betriebssystem ausführen kann, generiert. Ähnlich wie bei Java bringt die CLR eine Menge von Diensten mit, wobei hier nur einige, wie Garbage Collector und Security Engine, genannt sind.

Microsoft hat seit .NET ein einheitliches Typensystem eingeführt, welches die Kompatibilität mit den unterschiedlichsten Programmiersprachen gewährleistet (vorausgesetzt es gibt für die Sprache einen .NET fähigen Compiler). Durch das Common Type System (CTS) ist es nicht mehr nötig, eine Typenwandlung zwischen in unterschiedlichen Programmiersprachen entwickelten Objekten durchzuführen. Durch das CTS benutzten alle Sprachen schon dieselben Typen.

Eine weitere Neuerung sind die Entwicklungsumgebungen Visual Studio .NET und das .NET SDK (Software Development Kid). Das sind allerdings nur die Entwicklungsumgebungen von Microsoft, die direkt zum Framework dazugezählt werden. Andere Umgebungen von Fremdherstellern sind natürlich auch vorhanden, werden allerdings nicht zum Framework gezählt.

Die Web Services im .NET Framework sind eine Komponente, die mit dem Enterprise Server zusammenspielt die hier nicht weiter erläutert werden.

Eine wesentliche Komponente des .NET Frameworks ist das .NET Remoting, welches für die Unterstützung und Bereitstellung von verteilten Objekten zuständig ist.

Nachdem wird einige Komponenten von .NET kennen gelernt haben, schauen wir uns die .NET Remoting Infrastructure genauer an.

4.2 .NET Remoting Infrastructure

Die .NET Remoting Infrastructure (oder .NET Remoting, wie es auch häufig genannt wird) ermöglicht es, dass Objekte anwendungsdomänen-übergreifend kommunizieren.

Als Anwendungsdomäne wird bei Microsoft ein einzelner Prozess angesehen. Es kann somit durch die .NET Remoting Infrastructure sowohl auf Objekte auf derselben Maschine als auch auf einer entfernten Maschine zugegriffen werden. Die .NET Remoting Infrastructure stellt einige Dienste bereit, wie verschiedene Aktivierungsarten, Lease-Time von Objekten oder Kommunikations-Channel, die für die Datenübertragung zwischen Client und Server verantwortlich sind.

Um ein Objekt in ein Remoteobjekt umzuwandeln muss man es von der Klasse MarschalByReferenz ableiten. Die Remoting Infrastructure baut dann bei Aufruf diese Objektes auf der Clientseite automatisch einen Proxy (bei CORBA auch Stub genannt) auf, der sich wie das Entfernte Objekt verhält.

Ein entferntes Objekt kann sowohl Serverseitig aktiviert, als auch durch Clientaufruf aktiviert werden. Des Weiteren unterstützt die .NET Remoting Infrastructure verschieden Arten von Objekten (Singleton, Single Call). Ein „Singleton“ ist dabei ein Objekt, das nur einmal instanziert wird, und dann alle Clientanfragen verarbeitet. Das bedeutet auch,

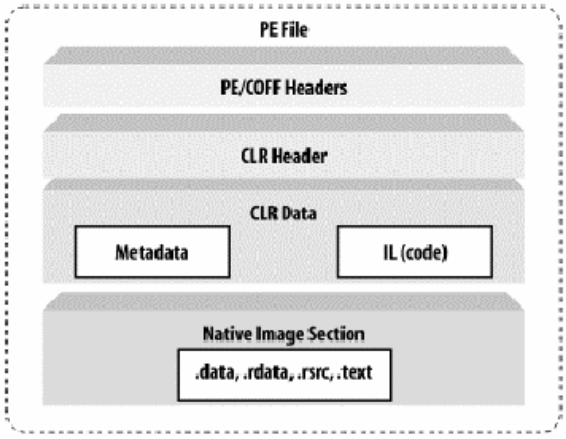
dass alle Clients mit denselben Daten arbeiten. Ein „Single Call“ Objekt ist ein Objekt von dem bei jedem Objektaufruf eine neue Instanz erstellt wird.
Um Daten zwischen Objekten austauschen zu können werden Channel benötigt. Channel sind registrierte TCP „Kanäle“. Channel müssen vom Remoteobjekt registriert werden, sodass Anfragen auf einem bestimmten Channel exklusiv an ein Objekt gehen.

Da wir jetzt einige Dienste der .NET Remoting Infrastructure kennen gelernt haben, schauen wir uns nun an, was für das Hosten eines Objektes benötigt wird, bzw. getan werden muss:

- Objekt muss von MarshalByReferenz abgeleitet werden
- GetLifetimeService muss aufgerufen werden (falls eine Lebensdauer für das Remoteobjekt erstellt werden soll)
- Register Channel
- Register Objekt (durch Tools im .NET Framework)

4.3 Assemblies und IL in der Common Language Runtime

Anders als zurzeit von COM+ oder DCOM kann man in Microsoft.NET zwar auch Klassenbibliotheken einbinden und referenzieren, allerdings funktioniert das nicht mehr nach dem alten DLL Prinzip. Früher konnte es gut sein, das es zwei verschiedene Versionen von derselben DLL gab, und deshalb ein Programm nicht funktionierte. Um dieser so genannten „DLL-Hell“ zu entgehen, gibt es im Microsoft .NET Assemblies (oder auch PE File!). Assemblies werden mit einer Versionsnummer und Public Key des Herstellers versehen und können dann direkt in den GAC (Global Assembly Cache) kopiert werden. Eine Registrierung in der Registry, was früher auch zu vielen Problemen geführt hat, ist somit also nicht mehr nötig. Gleichzeitig können durch die Metadaten im Assembly mehrere Versionen des gleichen Assembly’s existieren.
Diese Assemblies werden durch den Class Loader (Komponente der CLR) in ein Programm mit eingebunden. Nach dieser Einbindung wird durch den Verifier (auch Komponente der CLR) die Gültigkeit des Assembly’s überprüft und sichergestellt, dass die richtige Version der Klassenbibliothek geladen wird.
Assemblies vermeiden also eine Menge Fehlerquellen aus der Zeit vor .NET.
Damit Assemblies verarbeitet werden können wird die CLR benötigt (Assemblies werden auch CLR Executables genannt). Hier die Struktur eines Assemblies (PE Files):

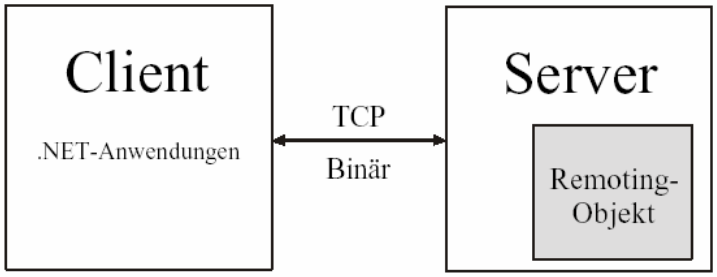


Wie man in der Grafik sieht enthält das PE File auch Metadata und IL.

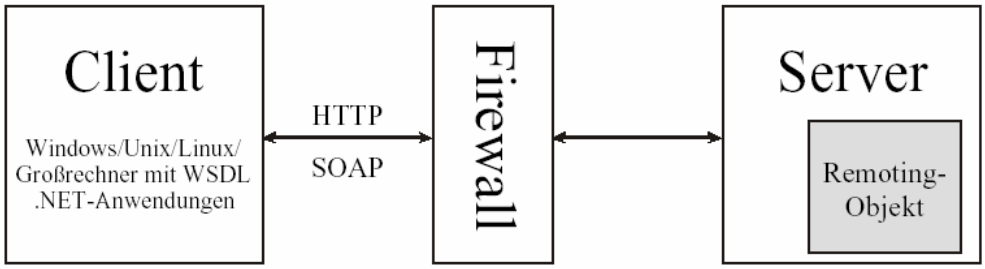
Metadata ist hierbei Information über die Datei selbst. Es enthält die Beschreibung der Interfaces des Programms und der Datentypen. Metadata ist also vergleichbar mit IDL. Allerdings müssen diese Informationen nicht selbst geschrieben werden, denn der .NET Compiler erzeugt diese selbstständig aus dem Originalquellcode.
IL Code ist der Quellcode in Intermediate Language, der von der CLR ausgeführt wird zu Laufzeit. IL Code beinhaltet also den eigentlichen Quelltext. Zusätzlich enthält ein PE File Metadata, welche die Schnittstellen, Klassen und Datentypen beschreibt und anhand derer der Class Loader referenzierte Klassen lädt. Die Native Image Section enthält die Standart Datei-Header, die jedes Windows Executable auch beinhaltet.

4.4 Schnittstelle zu anderen Techniken

.NET Objekte kommunizieren in der Regel über TCP/IP Channel (Siehe Grafik).



Da diese Kommunikation nur zwischen Windows Clients mit .NET Framework funktioniert, ist das nicht die passende Lösung für Webservices, die in .NET unterstützt werden. Damit ein Objekt auch auf einem Webserver verfügbar ist, ist eine andere Technik nötig.
Da die Clients bei Webabfragen nur Verbraucher sind, können sie .NET Objekte über XML Anfrage auf dem Enterprise Server erreichen. Microsoft hält sich bei dieser Web-Objekte Unterstützung an den offenen Standart SOAP. SOAP ist ein Mechanismus, der es ermöglicht Nachrichten zwischen Client und Server über XML auszutauschen. Auf diesem Weg ist es möglich, auf in .NET geschriebene Objekte auf anderen Plattformen zuzugreifen und diese zu benutzen. So wäre es denkbar aus einer CORBA Anwendung ein .NET Objekt über SOAP auf einem Enterprise Server anzusprechen (Siehe Grafik).



Die Kompatibilität zu anderen Plattformen ist somit hergestellt.
Prinzipiell wäre es möglich ein .NET Framework für ein anderes Betriebssystem zu entwickeln. Da die Formate der PE File standardisiert sind, wäre es dann möglich, .NET Objekte und Anwendung auf nicht WIN32 Plattformen zu benutzen.

4.5 Beispiel: .NET Anwendung

Im Folgenden Betrachten wir ein Beispiel (einfache Bank Anwendung)für eine .NET Remote Anwendung. Der Beipielcode ist in C# geschrieben:

-> Remote Anwendung:
ServerHello.cs:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace NetHello

{
// Klasse als Remote-Klasse von MarshalByRefObject ableiten
public class HelloServer : MarshalByRefObject
{
  public Aus()
  {
  Console.WriteLine (".NET Hello Word!");
  }
}
```

-> Client Anwendung:
Client.cs:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace NetHello

{
  public class Client
  {
   public static int Main(string [] args)
   {
   if (args.Length == 1)
   {
  // Channel-Objekt anlegen
     TcpChannel chan = new TcpChannel();
  // Channel registrieren
     ChannelServices.RegisterChannel(chan);
  // Remoteobjekt referenzieren
     HelloServer Hello = (HelloSever)Activator.GetObject(typeof(
     ServerHello.HelloServer), "tcp://" + args[0] + ":8085/HEllo");
     char eingabe = 'q';
     ServerHello.Aus();
   }
   else
     Console.WriteLine ("Aufruf in der Form: client <HOST>");
     return 0;
   }
  }
}
```

Vorbereiten des Servers:
-> server.cs:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace BankBeispiel {
  public class Server {
    public static int Main(string [] args) {
  // Channel-Objekt erstellen
    TcpChannel chan = new TcpChannel(8085);
  // Channel registrieren
```

```
    ChannelServices.RegisterChannel(chan);
  // Objekt registrieren
    RemotingConfiguraton.RegisterWellKnownServiceType(Type.GetType(
    "NetHello.HElloServer,bank"), "Hello",WellKnownObjectMode.Singleton);
    System.Console.WriteLine("<ENTER> zum Beenden...");
    System.Console.ReadLine();
    return 0;
   }
  }
}
```

-> server_cpp.cpp:

```
#using <mscorlib.dll>
#using <System.dll>
#using <System.Runtime.Remoting.dll>
using namespace System;
using namespace System::Runtime::Remoting;
using namespace System::Runtime::Remoting::Channels;
using namespace System::Runtime::Remoting::Channels::Tcp;

void main()
  {
  TcpChannel* chan = new TcpChannel(8085);
  ChannelServices::RegisterChannel(chan);
  RemotingConfiguration::RegisterWellKnownServiceType(Type::GetType(
  "NetHello.HelloServer,Hello"), "Hello",WellKnownObjectMode::Singleton);
  Console::WriteLine("<ENTER> zum Beenden...");
  Console::ReadLine();
}
```

-> build.bat (erzeugen):

```
Build.bat:
@echo off
md bin
csc /target:library /out:bin\hello.dll ServerHello.cs
csc /R:bin\bank.dll /out:bin\client.exe CHello.cs
csc /out:bin\server.exe server.cs
cl /clr /o bin\server_cpp.exe server_cpp.cpp
del server_cpp.obj
```

-> delete.bat:

```
Delete.bat:
@echo off
del bin\client.exe
del bin\server.exe
del bin\server_cpp.exe
del bin\hello.dll
rd bin
```

5 JAVA – RMI

5.1 Einführung

Ähnlich wie bei .NET ist RMI (Remote Method Invocation) eine feste Komponente in JAVA. RMI ist sowohl in der Runtime Environment enthalten, als auch im Software Development Kid. RMI ist das Werkzeug, das JAVA mitbringt, um einen Methodenaufruf auf einer entfernten Maschine so einfach zum machen, als wäre er lokal. RMI ist in JAVA bereits seit der Version 1.1 von 1997 enthalten und bildet auch heute noch einen Kern von JAVA. CORBA war zu diesem Zeitpunkt schon als Standart etabliert, jedoch hat SUN es trotzdem als sinnvoll angesehen, eine speziell auf JAVA abgestimmte

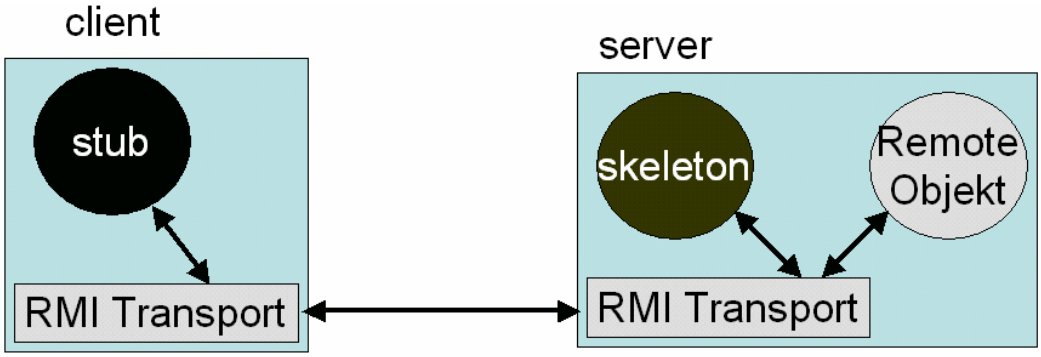
Middelware zu entwickeln. RMI hatte gegenüber CORBA mehrere Vorteile für den die Entwickler:

- Unterstützung eines Distributed Garbage Collectors
- Intuitiv benutzbar
- Kostenlos

Zurzeit als RMI entwickelt wurde, waren noch nicht viele freie CORBA Entwicklungen zu haben, so dass sich RMI auch schnell durchsetzte (in JAVA).
Da RMI mittlerweile das CORBA IIOP Protokoll unterstützt (Seit JRE 1.4), ist es prinzipiell auch offener geworden (RMI ist kein offener Standard).

5.2 RMI Architecture

Da man bei der Entwicklung von Java/RMI den Kerngedanken hatte, das man sowohl Server als auch Clientseitig nur Java Code ausführen wollte, erübrigte sich eine spezielle Interface Definition Language, wie es sie in den anderen beiden Beispielen von Middelware gibt. Weiterhin ist auch keine Serialisierung von Datentypen notwendig, da auf beiden Seiten in derselben Sprache programmiert wird.
RMI bringt alles notwendige, um Objekte entfernt aufzurufen schon mit sich. Im Package **`java.rmi.Remote`** befinden sich die wichtigsten Klassen, die für verteilte Objekte notwendig sind.
Will man also ein Objekt „remote“ verfügbar machen, so ist es notwendig, diese Klasse vom Interface java.rmi.Remote abzuleiten. Eine Klasse, die vom Interface Remote abgeleite ist, kann aus der Ferne aufgerufen werden.
Ähnlich wie in CORBA spricht man bei RMI auch auf der Client Seite von Stubs. Stubs verhalten sich wie ein Proxy auf der Clientseite, der alle Interfaces vom Remoteobjekt lokal abbildet und alle Anfragen an das Remoteobjekt weiterleitet. Serverseitig ist ein Skeleton vorhanden, welcher der Objektadapter auf der Serverseite ist.



Die oben gezeigte Grafik symbolisiert den Ablauf eines Verbindungsaufbaus bei RMI.
Der Stub auf der Clientseite wird automatisch vorgeneriert aus der Serverklasse, sodass sie alle Methoden enthält, die die Server Serverklasse enthält. Ruft der Client dann eine Methode in der Serverklasse auf, ruft er diese tatsächlich im Stub auf, als wäre es die Lokale Serverklasse. Der Stub wiederum baut daraufhin eine Socket Verbindung zum Server auf, oder benutzt eine Verbindung die bereits besteht (wenn ein Methodenaufruf schon vorher getätigt wurde). Der Stub „marshallt“ alle Informationen, den

Methodenname und Parameter eingeschlossen, und sendet diese Informationen über die Verbindung zum Skeleton. Der Skeleton „unmarshallt“ diese Informationen und gibt sie an die Serverklasse weiter. Rückgaben von der Serverklasse werden vom Skeleton wiederum gemarshallt und an den Stub über die Socket Verbindung zurückgesandt. Der Stub „unmarshallt“ diese Informationen und leitet sie an die Clientmethode weiter.

RMI bringt dabei schon alle Werkzeuge mit, um im Quellcode fertige Socket Verbindungen registrieren, Stubs und Skeleton generieren und Serverobjekte registrieren zu können.

5.3 IDL in RMI

Anders als in CORBA und .NET wird in JAVA/RMI keine spezielle Sprache zur Definition der Interfaces benötigt. RMI verwendet die Java Konstrukte, um Interfaces zu beschreiben und zu spezifizieren. Der „rmic“ Compiler generiert dann die Stub- und Skeletonklassen für das entsprechende Remote Objekt.

5.4 Namensdienst anhand einer Beispielanwendung

HelloServer.java:

```
import java.rmi.Remote;
  import java.rmi.RemoteException;

  public interface HelloServer extends Remote {

    public String sayHello() throws RemoteException;

  }
```

HelloServer.HelloServerimp.class:

```
import java.rmi.server.UnicastRemoteObject;
  import java.rmi.RemoteException;
  import java.rmi.Naming;

  import HelloServer.*;

  public class HelloImpl extends UnicastRemoteObject
                         implements HelloServer {

    public HelloImpl() throws RemoteException {
    }

    public String sayHello() {
      return "Hello World";
    }
  }
```

Generierung der Proxy Objekte mit dem Tool `rmic`:

```
rmic de.dpunkt.rmi.hello.server.HelloImp
```

Starten der Registry:

```
rmiregistry
```

Objekt Implementierung beim Namensdienst:

```
public static void main(String args[]) {
    try {
      Naming.rebind("hello-server", new HelloImpl());
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
```

Client Anwendung:

```
import java.rmi.Naming;

  public class HelloClient {

    public static void main(String args[]) {
      try {
        HelloServer server;
        server = (HelloServer)Naming.lookup("hello-server");
        String result = HelloServer.sayHello();
        System.out.println(result);
      } catch(Exception ex) {
        ex.printStackTrace();
      }
    }
  }
```

6 Vergleich der Middleware-Typen

CORBA, als erstgenannte Middleware, existierte damals als erste Version eher als .Net und RMI. Sie wurde von einem Konsortium verschiedenster Mitglieder entwickelt. Durch die freie Verfügbarkeit des unabhängigen Standards hat sich CORBA verbreitet und wird heute von vielen Programmiersprachen wie z. B. auch C++, Java usw. unterstützt. Durch die plattform-, betriebssystem- und programmiersprachenunabhängige Struktur eignet sich CORBA besonders für den Einsatz in heterogenen Umgebungen. Gleichzeitig ist durch die Entwicklung von einem recht unabhängigen Konsortium die Unabhängigkeit der Middleware gewährleistet. CORBA ist nicht von der Gunst oder der Strategie eines Unternehmens abhängig, sondern wird auch profitunabhängig weiterentwickelt.

JAVA mit seiner direkt in die „Java Virtual Machine“ (JVM) integrierten Middleware erfüllt mit RMI die meisten Grundbedürfnisse eines JAVA Entwicklers. RMI ist schon direkt in Java enthalten und muss nicht mehr zusätzlich integriert werden. Ebenso bring RMI einen eigenen Namensdienst mit und generiert Stubs und Skeletons unkompliziert über mitgebrachte Werkzeuge. RMI wird allerdings nur in JAVA Umgebungen unterstützt. Andere Programmiersprachen werden nicht unterstützt, allerdings ist die JVM für nahezu jedes Betriebssystem zu haben und somit ist RMI auch fast auf jeder Plattform möglich.

Microsoft bietet mit .NET die jüngste Middleware im Vergleich. Allerdings ist .NET auch eine Weiterentwicklung von DOM und COM+, welches es schon seit längerer Zeit gibt. In .NET hat Microsoft seine Middlewarekonzept offener gestaltet, sodass jetzt quasi jede Programmiersprache mit .NET fähigem Compiler unterstützt wird. Durch die Einführung einer eigenen „Virtuellen Maschine“, der Commmon Language Runtime, im .NET Framework vorhanden, wäre sogar eine Unterstützung verschiedener Betriebssysteme denkbar, was allerdings wohl am fehlenden .NET Framework für andere

Betriebssysteme scheitert. So ist .NET eher für homogene WIN32 Umgebungen interessant. Allerdings ist .NET fähig, über den Standard SOAP seine Objekte auch anderen Umgebungen bereit zu stellen.

7 Resümee

Sucht man die richtige Middleware für sein Softwareprodukt, wird man dies von der unterstützten Umgebung und der Programmiersprache, die verwendet werden soll, abhängig machen müssen.

Will man in einer Java Umgebung Software entwickeln und verteilte Objekte einsetzten, so wird die Wahl wohl auf JAV/RMI fallen. Die Einbindung von RMI Objekten ist relativ einfach gestaltet und es sind für den Programmierer auch keine spezifischen IDL Kenntnisse erforderlich.

Hat man jedoch den Wunsch, seine Software nur auf Windowsoberflächen einzusetzen, und die bevorzugte Programmiersprache ist eine andere als JAVA, so ist .NET zu empfehlen. Das Implementieren von Remote Objekten ist in .NET ähnlich einfach wie in JAV/RMI. Auch hier sind nicht unbedingt IDL Kenntnisse erforderlich, wenn man sie besitzt, kann man hier allerdings auch mit einer IDL ähnlichen Sprache arbeiten.

Hat man jedoch im Sinn, eine Software für eine völlig heterogene Landschaft, also für unterschiedliche Programmiersprache und Betriebssysteme, zu schreiben, so wird wohl kein Weg an CORBA vorbei gehen. Allerdings ist CORBA etwas schwierig zu implementieren, hat man keine Kenntnisse der IDL. Ist man allerdings erst einmal eingearbeitet, sind auch CORBA Anwendungen problemlos und in der bevorzugten Programmiersprache, Plattform übergreifend, schnell implementiert.

8 Quellen

***Engineering Distributed Objects**, 1999*
Wolfgang Emmerich, Stefan Tai, Springer
ISBN 3-540-41792-3

***Software Engineering, 6th Edition,** 2001*
Ian Sommerville, Addison Wesley
ISBN 0-201-39815-X

***Engineering Distributed Objects**, 2000*
Wolfgang Emmerich, John Wiley & Sons, Ltd
ISBN 0471986577

OMG CORBA Specifications
http://www.omg.org

SUN Microsystems
http://developer.java.sun.com/developer/technicalArticles/releases/corba/

FH Darmstadt
http://www.fbi.fh-darmstadt.de/~schuette/Vorlesungen/VerteilteSysteme/Skript/4_Objektdienste/JavaIDL.pdf

***Microsoft .NET Remoting**, 2003*
Scott McLean, James Naftel, Kim Williams, Microsoft Press
ISBN 0735617783

***.NET Framework Essentials**, Second Edition, February 2002*
Thuan L. Thai, Hoang Lam, O'Reilly
ISBN 0-596-00302-1, 320 pages

***.NET als alternative Middleware**, Hauptseminar, 2002*
Professor Dr. D. Reschke, Technische Universität Ilmenau

***Java RMI**, First Edition October 2001*
William Grosso, O'Reilly
ISBN: 1-56592-452-5, 572 pages

***Java Programmierhandbuch und Referenz für die Java-2-Plattform**, Standard Edition, DPunkt Verlag,* 3. Auflage 2002

Stefan Middendorf, Reiner Singer, Jörn Heid
http://www.dpunkt.de/java/

An Architectural View of Distributed Objects and Components in CORBA, Java RMI, and COM/DCOM

František Plášil, Charles University Faculty of Mathematics and Physics, Department of Software Engineering
http://nenya.ms.mff.cuni.cz

Michael Stal, Siemens AG Corporate Technology, München

SUN Microsystems

http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html

Distributed Systems
-
Architekturen verteilter Software Systeme

Michael Nordhoff

Thomas Wendt

Grundzüge des Systems Engineering 2

Universität Duisburg-Essen Juli 2003

Inhalt

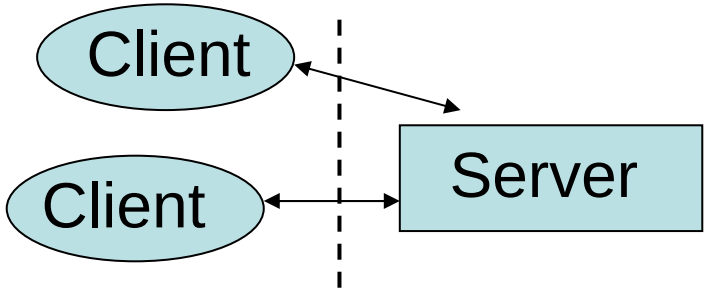
1. Einleitung
2. Architekturen Verteilter Systeme
3. Middleware
   3.1 CORBA
   3.2 .NET
   3.3 Java – RMI
4. Vergleich der Middleware
5. Resümee

Verteilte Systeme – Charakteristika

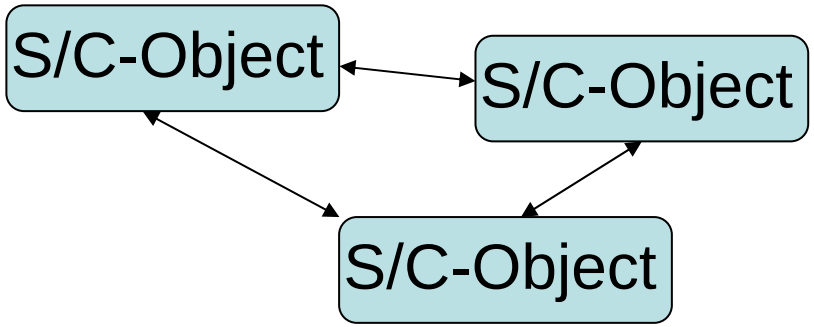
- Resource sharing (Ressourcenaufteilung)
- Openness (Offenheit)
- Concurrency (Parallelität)
- Scalability (Skalierbarkeit)
- Fault tolerance (Fehlertoleranz)
- Transparency (Transparenz)

Architekturen

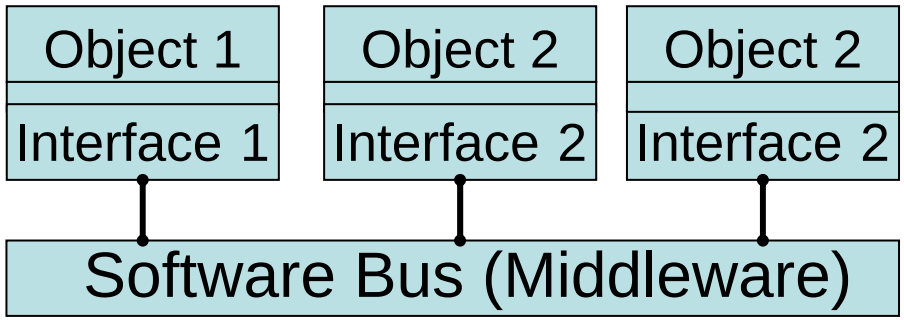
- **Client-Server**
  - Aufgaben u. Funktionen verschieden



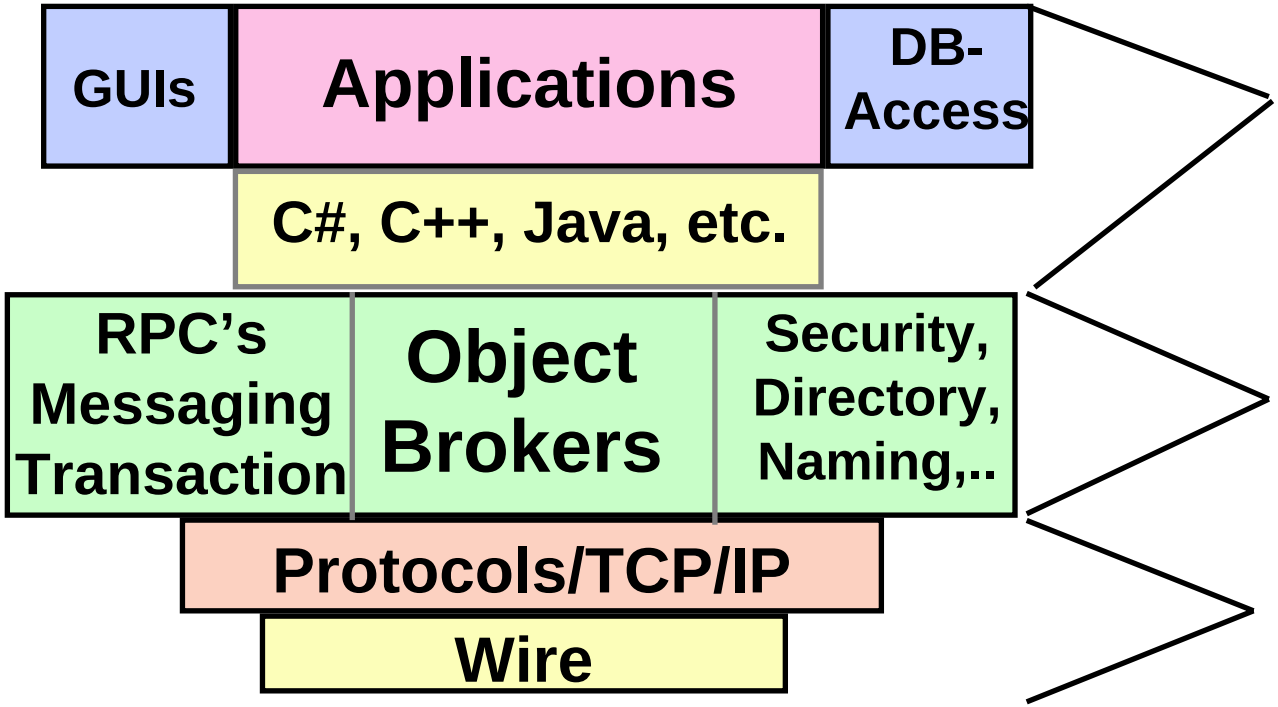
- **Verteilte Objekte**
  - Keine C/S-Unterscheidung
  - Hohe Flexibilität
  - Hohe Skalierbarkeit



  - Kommunikation mittels „Middleware“



Middleware im Schichtenmodell



Components
(Layer 7)

"The Middle"
(Layer 5-6)

Transmission
(*Layer 1-4)*

Middleware´s Aufgaben:

- Parameterkonvertierung
- Bekanntmachung von Objekten
- Synchronisierung
- Fehlerbehandlung

Middleware-Entwicklung

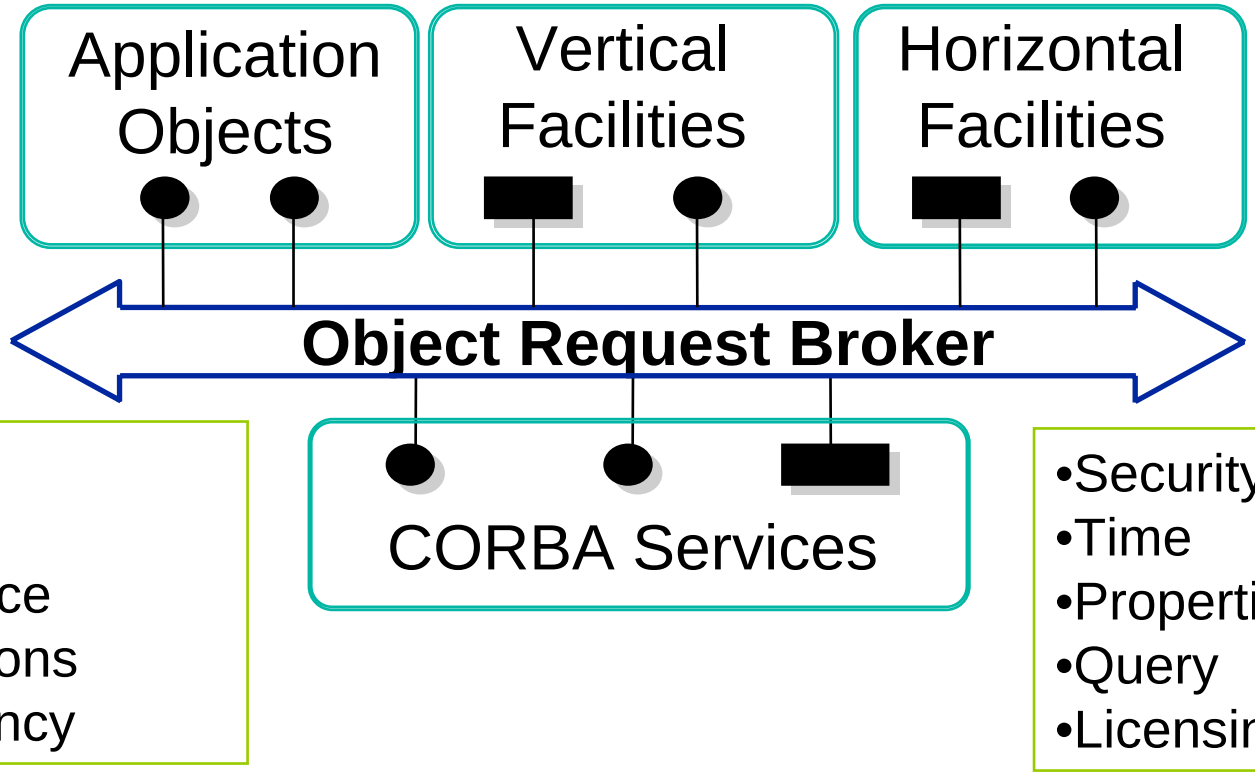
- **transaktions-orientierte Middleware**
  - Einsatz bei verteilten Datenbanksystemen zur Lastverteilung und Fehlertoleranz
- **nachrichten-orientierte Middleware**
  - Server- und Clientprozesse bereits entkoppelt
- **Remote Procedure Calls** (RPC)
  - Prozeduraufruf über Computergrenzen hinweg
  - Schnittstellendefinitionssprache bereits programmiersprachenunabhängig
- **objektorientierte Middleware**
  - Konzept der Vererbung
  - IDL unterstützt Konzept der Objekttypen

CORBA: OMA - Übersicht

- Not standardized
- Scope is Single application

- Healthcare
- Finance
- Telecommunication

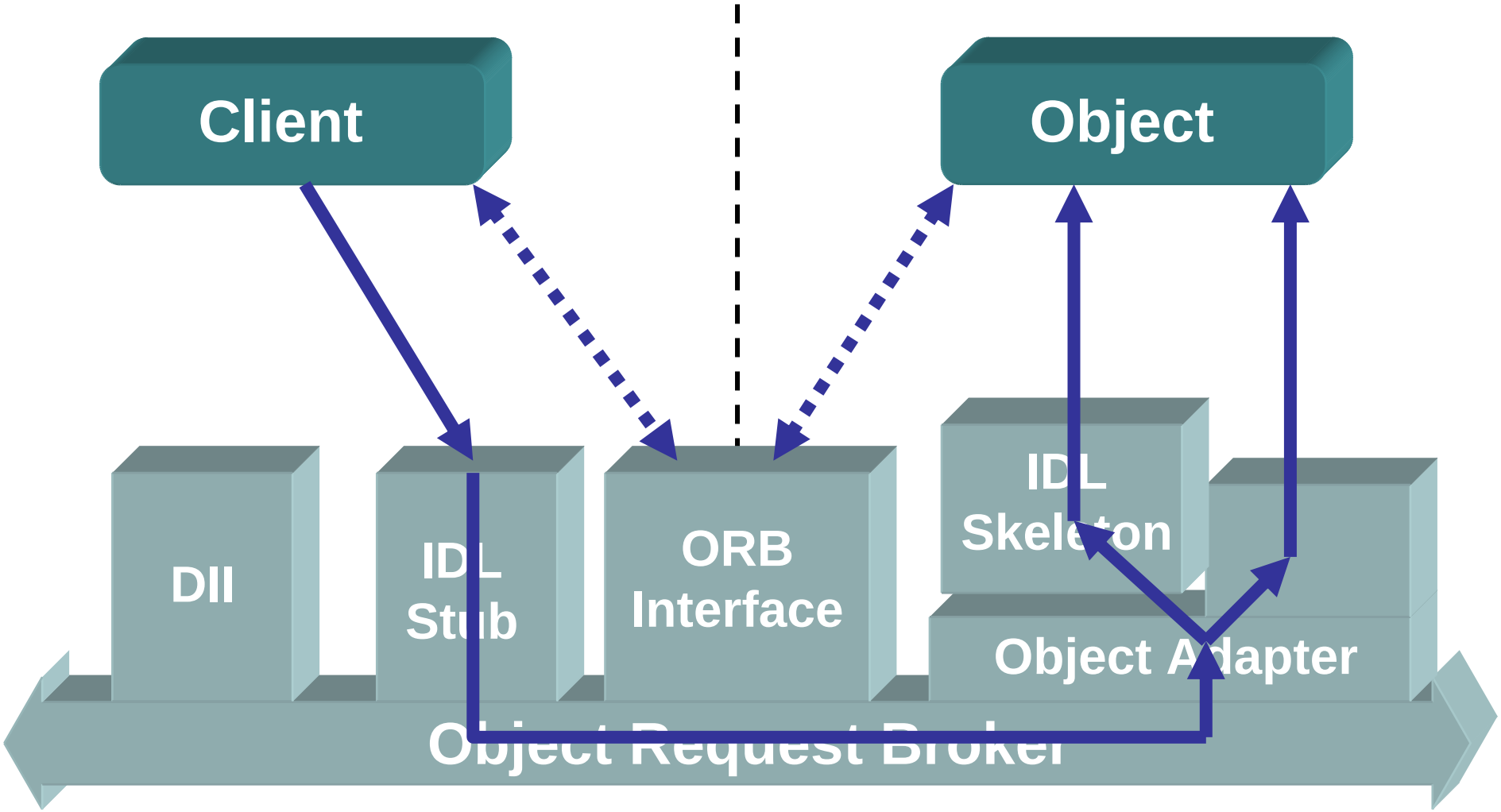
- Object Linking
- Help Facilities
- Desktop Mgmt



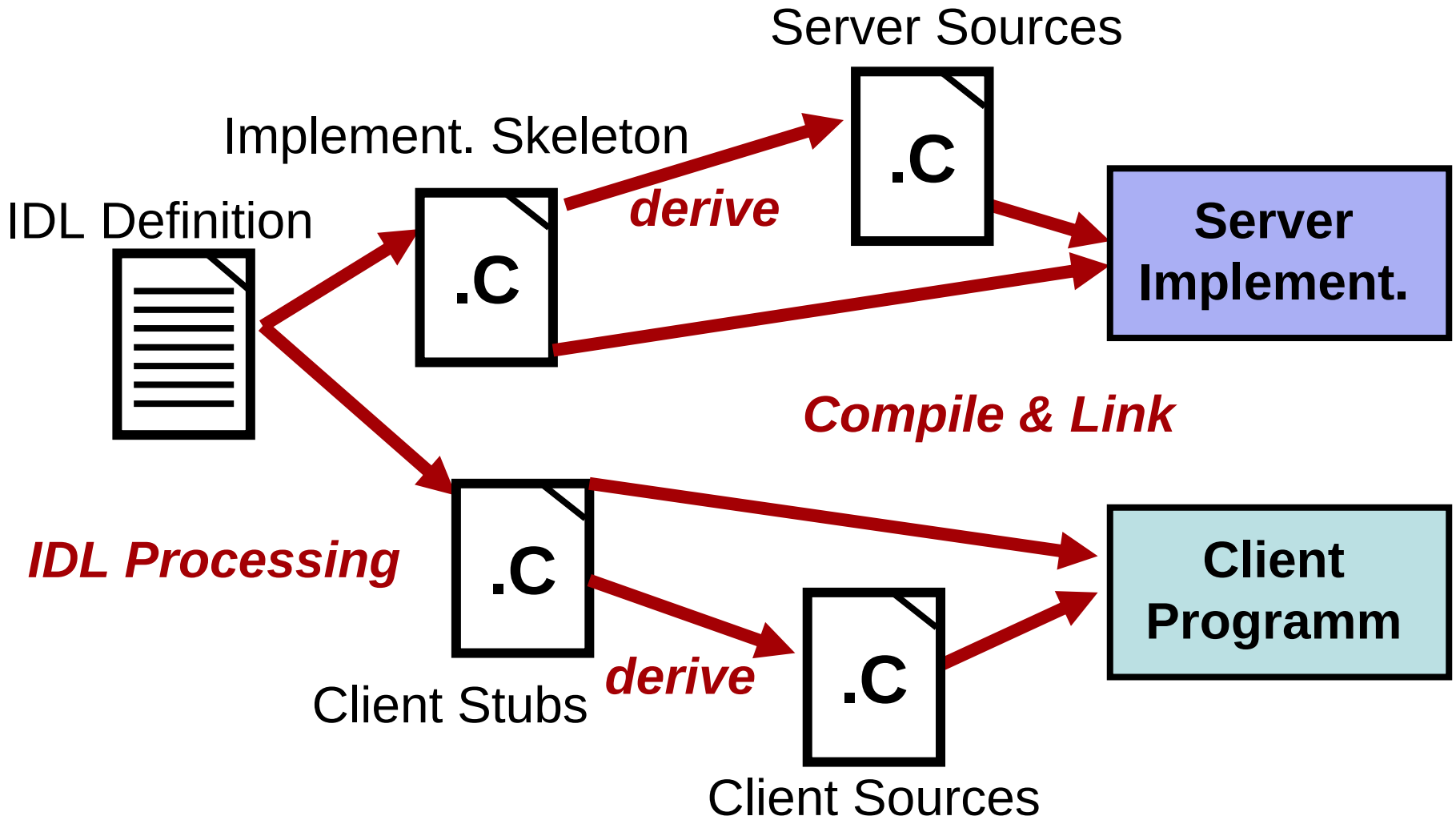
- Events
- Naming
- Persistence
- Transactions
- Concurrency

- Security
- Time
- Properties
- Query
- Licensing

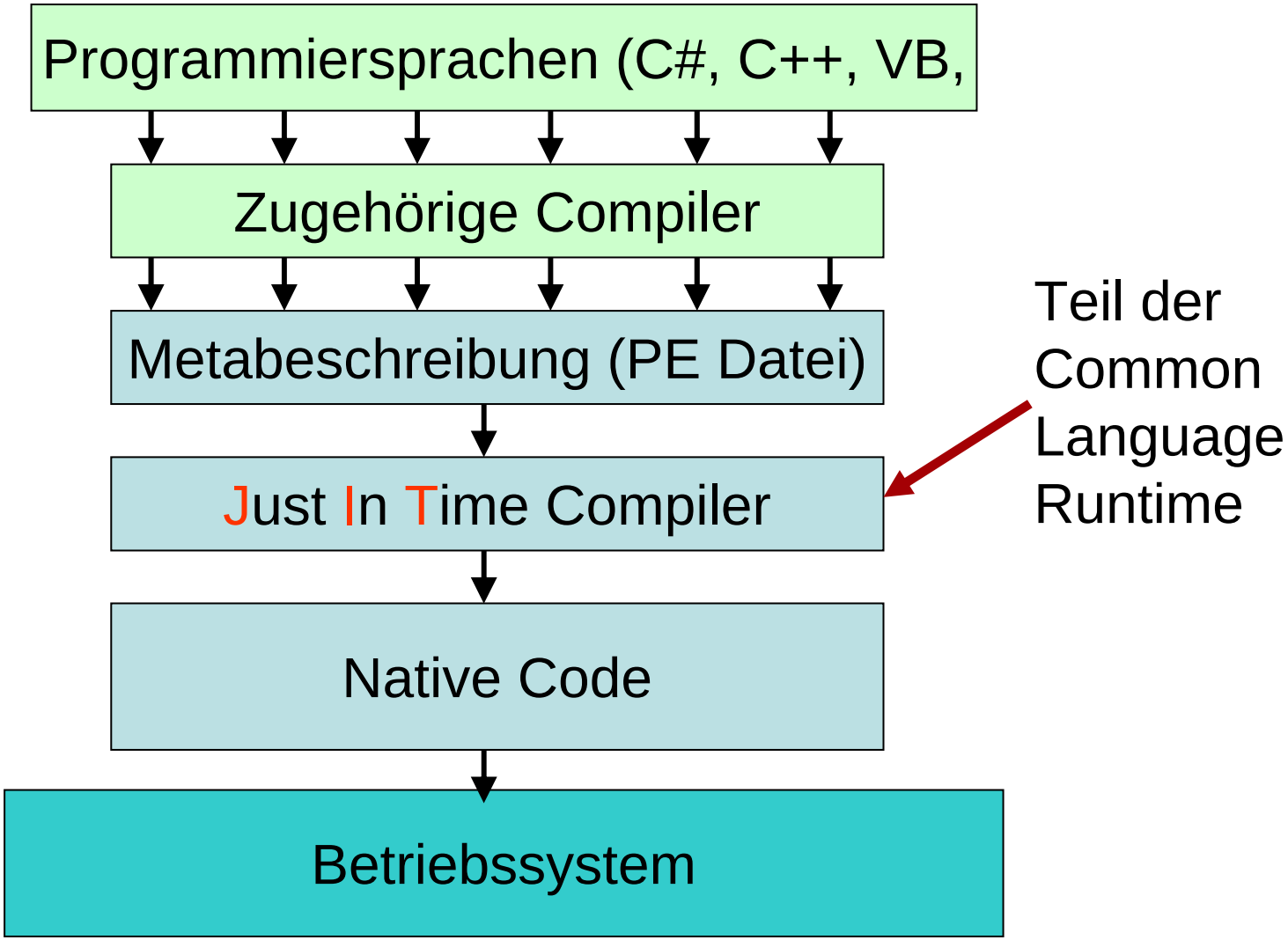
CORBA-ORB


Client
Object
DII
IDL Stub
ORB Interface
IDL Skeleton
Object Adapter
Object Request Broker

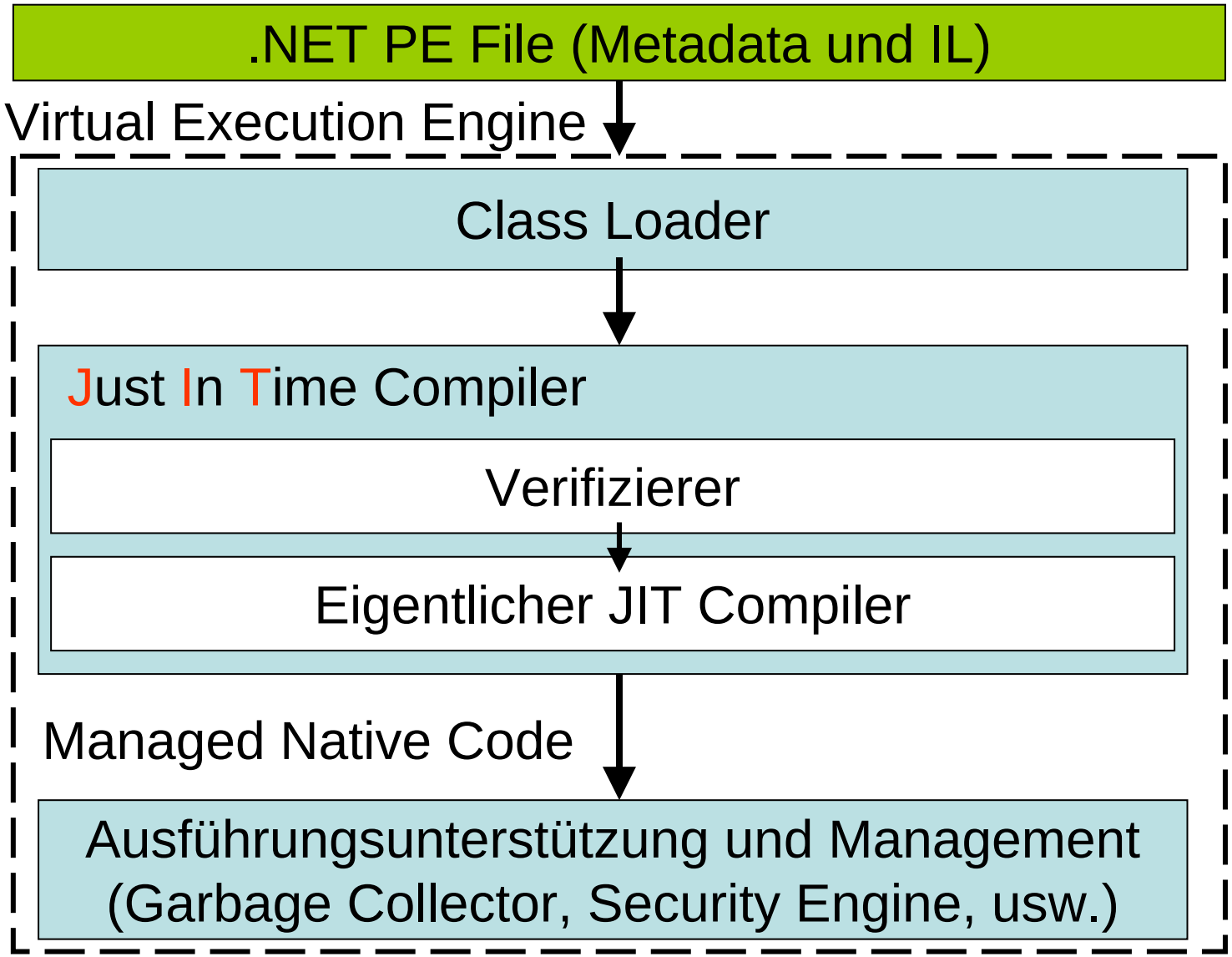
Implementieren von Corba-Objekten



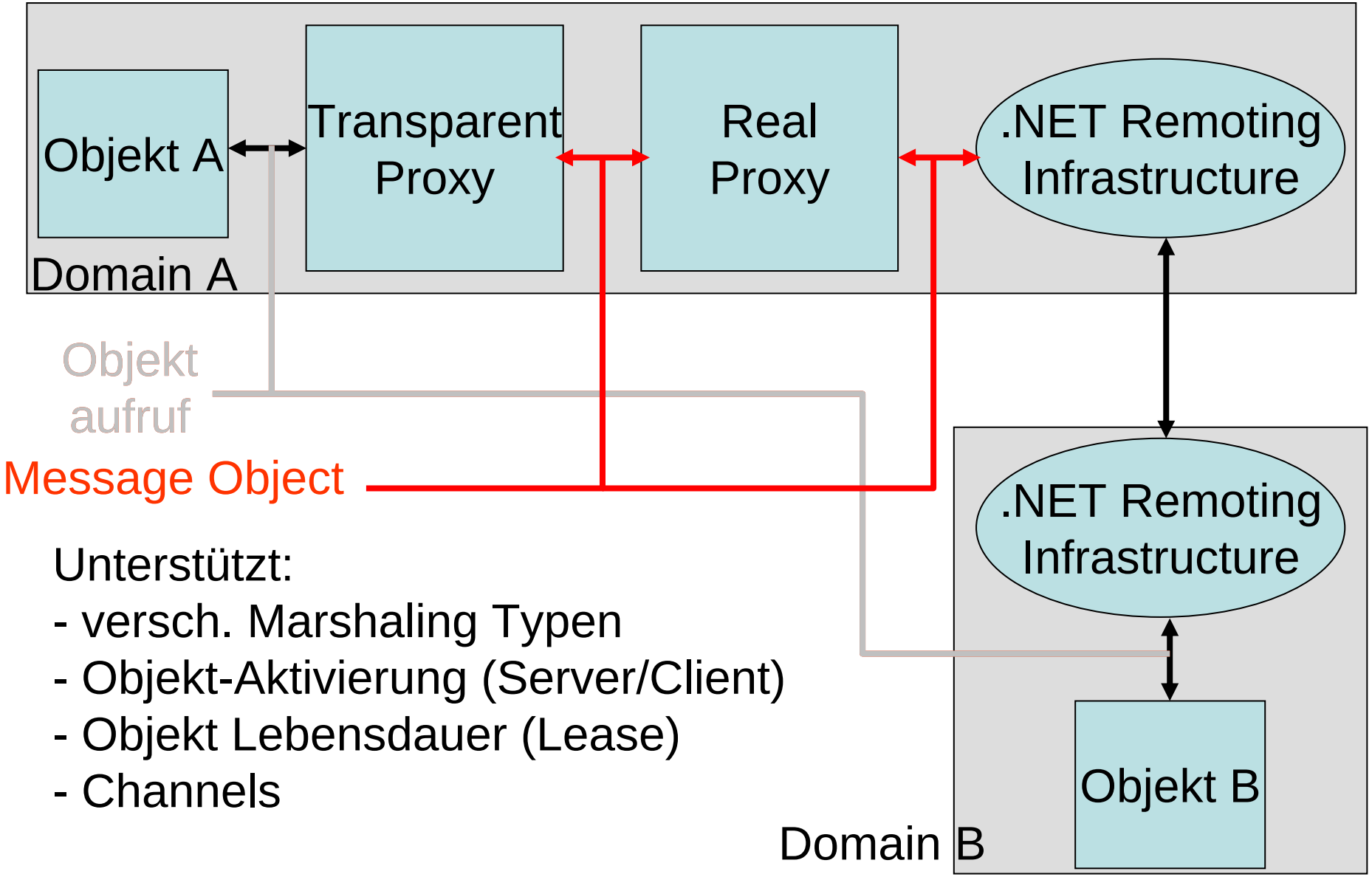
.NET Framework



Common Language Runtime

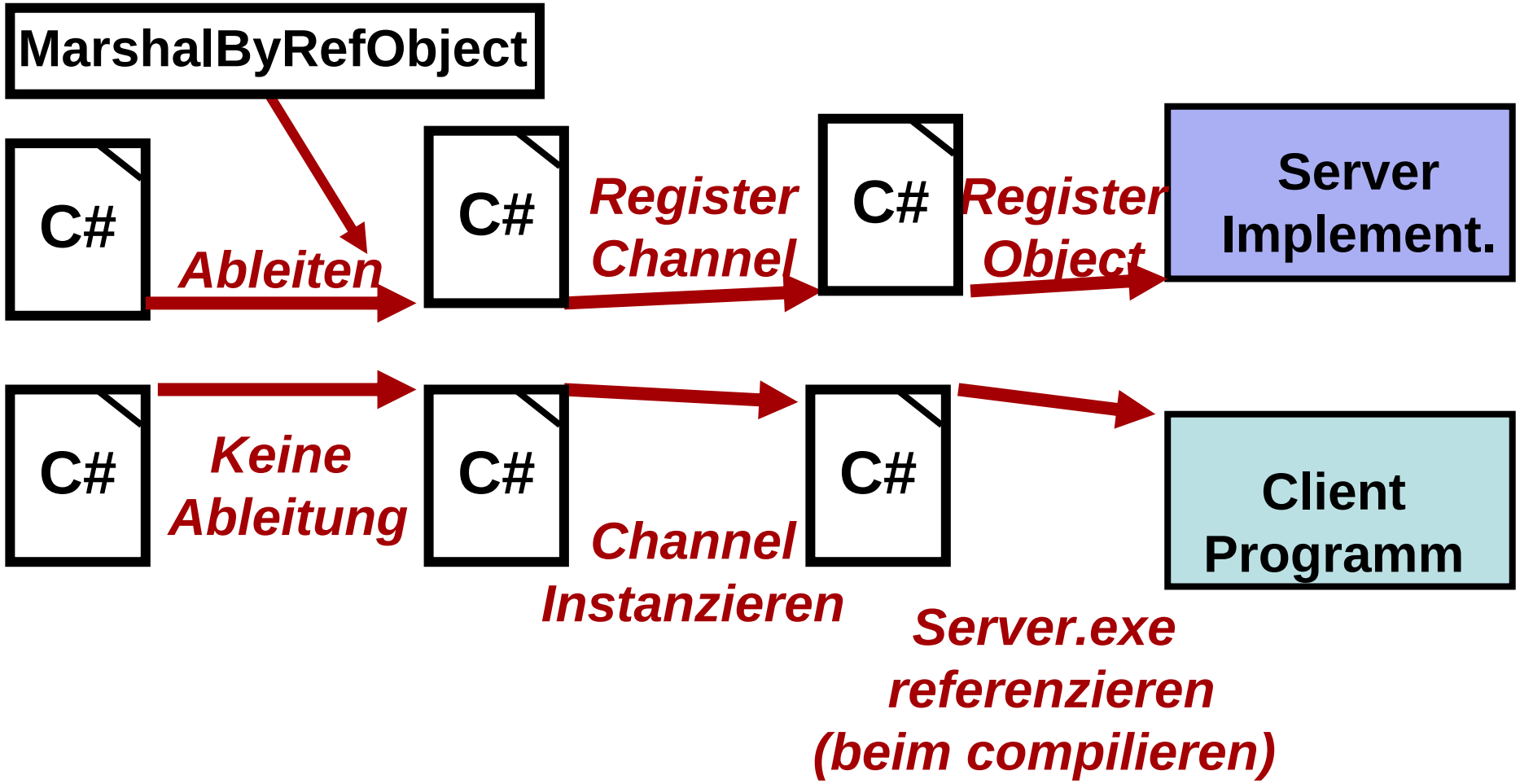


.NET Remoting Infrastructure

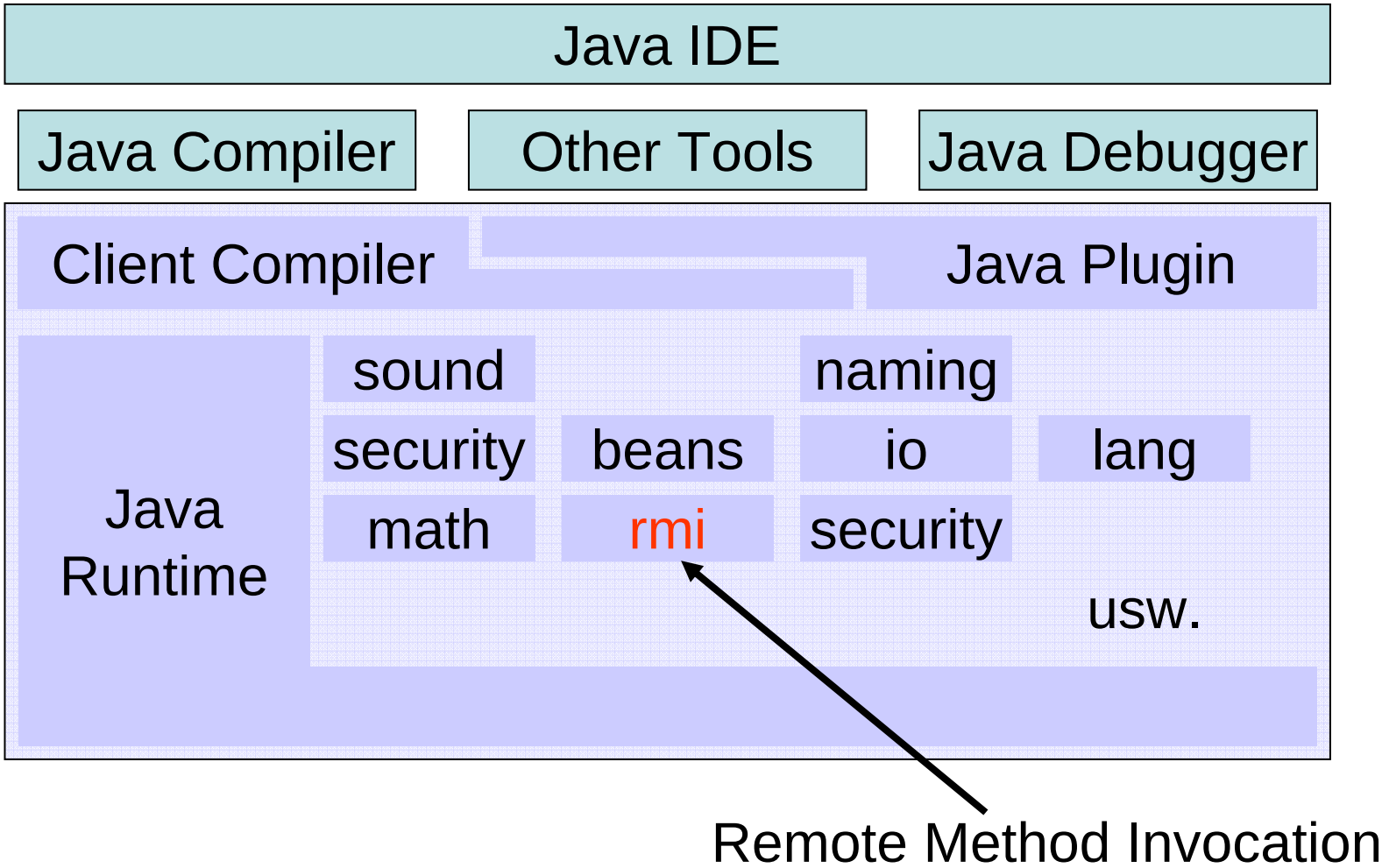


Unterstützt:
- versch. Marshaling Typen
- Objekt-Aktivierung (Server/Client)
- Objekt Lebensdauer (Lease)
- Channels

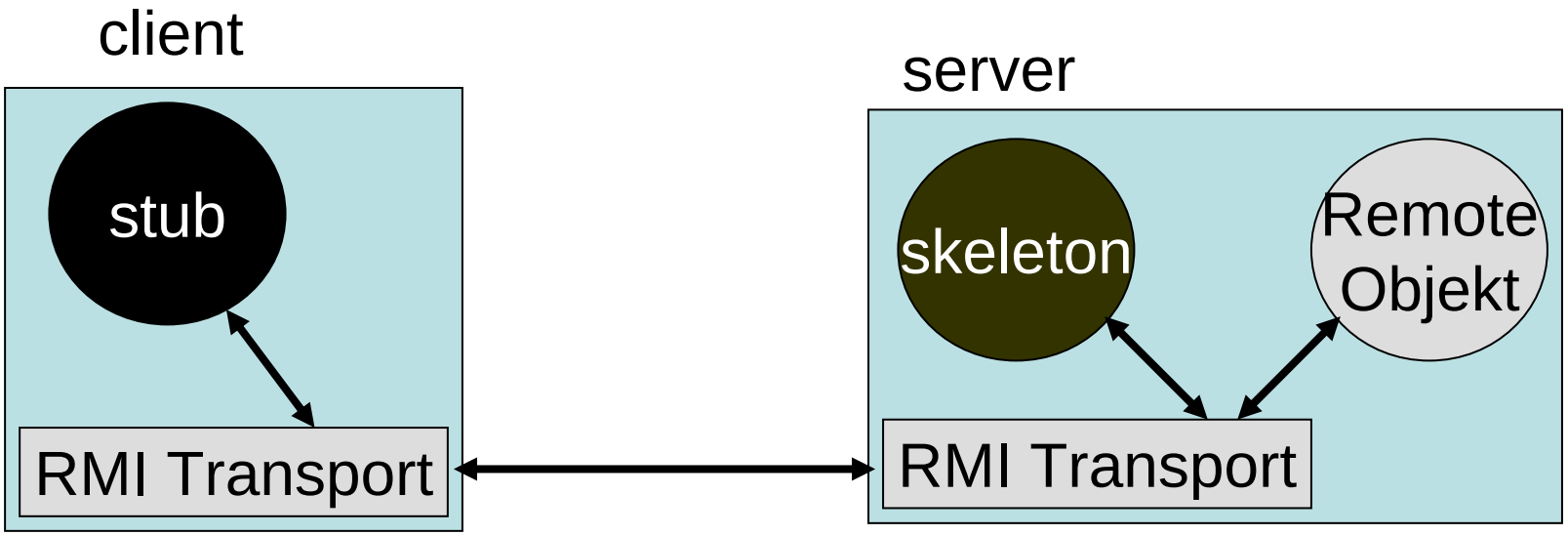
Implementieren von .NET-Objekten



Java Overview

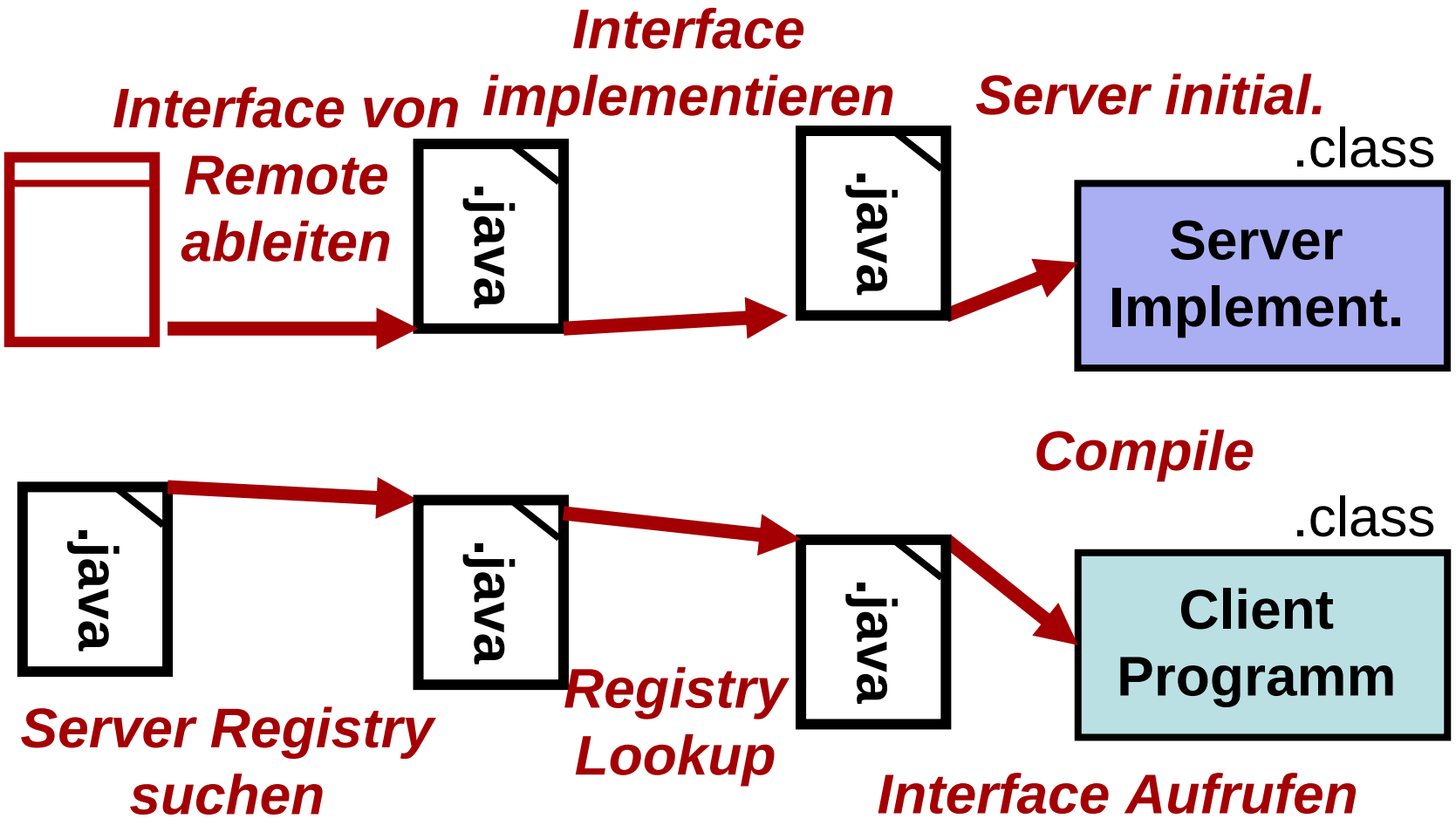


RMI (Remote Method Invocation)



- keine IDL Definition notwendig
- Marshaling durch RMI
- keine Typenwandlung
- Registry und Nameservice vorhanden

Implementieren von RMI-Objekten



Vergleich der Middleware

CORBA

- Plattform übergreifend (für Heterogene Systeme geeignet)
- Unterstützt viele Programmiersprachen (z.B. C++, Java, OOCobol usw.)
- Konsortium OMG

.NET

- Unterstützt viele Programmiersprachen (C++, C#, Visual Basic 7.0)
- Nur auf Win32 Umgebungen
- In .NET Framework integriert

RMI

- Unterstützt nur Java
- Kein Offener Standard
- In die JVM integriert

Resümee

- Heterogene/Komplexe Systeme

➔CORBA

- Applets/Java

➔RMI

- Win32/ASP

➔ .NET