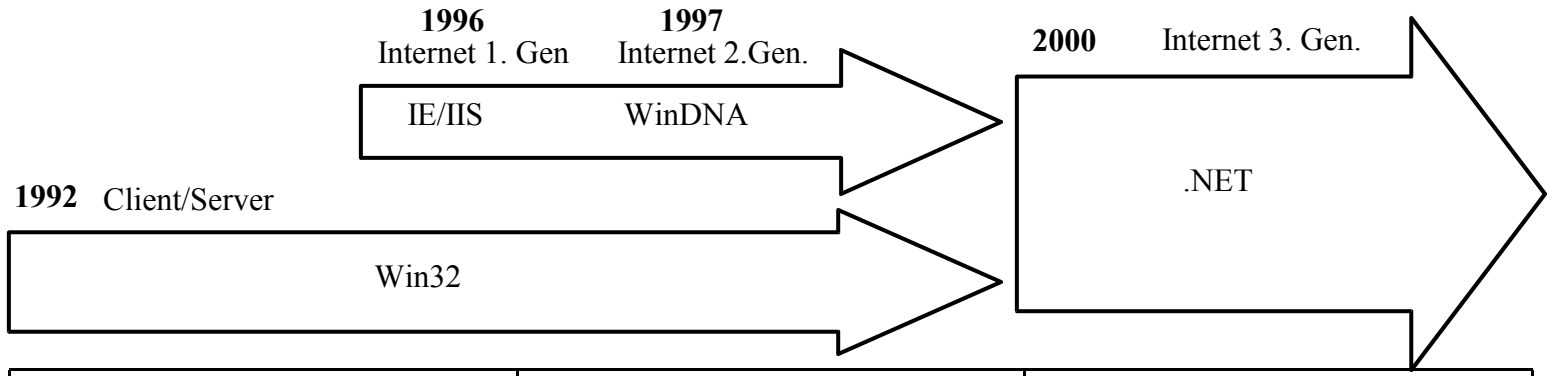
Mircosoft DotNot

Verteilte Anwendungen „the microsoft(cc) way“

Die DotNet-Plattform

- Microsofts zukünftige Plattform für die Erstellung (verteilter) Anwendungen
  - → Zusammenfassung bestehender und zukünftiger Produkte in eine „Architektur“
  - – Sehr starke Anleihen bei Java/J2EE
  - – Ziel: Software → Dienstleistung, Erstellung von Webservices



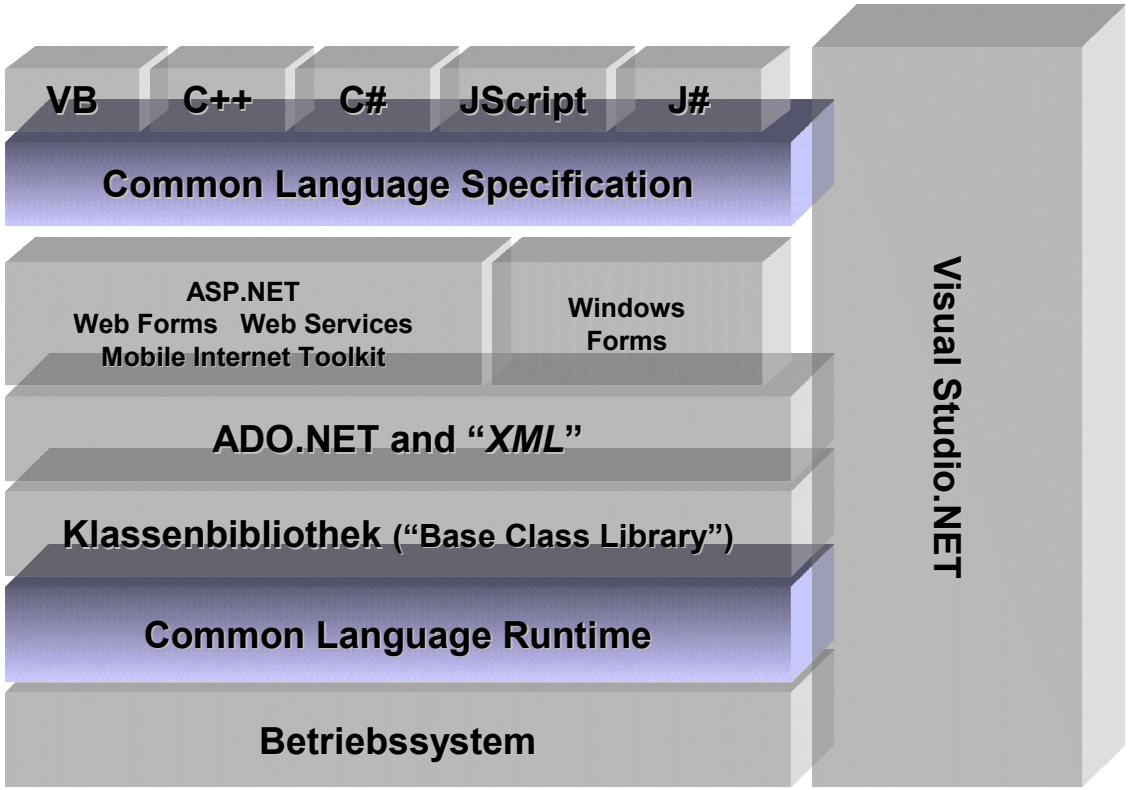
1. Generation 1994-1996	2. Generation 1996-2000	3. Generation 2000-
Statische Seiten	Dynamische Seiten	Nutzung von Diensten
E-Mail, Basisinformationen	Personalisierung, E-Commerce	Web Services
IE/IIS	Windows DNA	Microsoft .NET

Die Geschichte der Computernetzwerke nach Microsoft [SIC!]

Bestandteile der DotNet-Plattform

- DotNet-Framework und Tools
  - Klassenbibliothek
  - CLR
  - ASP.Net
  - Entwicklungstools
- Building Block Services (Foundation Services)
  - Gesamtheit der DotNet-Webservices (Passport® etc.)
- Enterprise Server
  - Infrastruktur
  - gesammelte herkömmliche M$-Produkte
- DotNet Device Software (Mobile Devices)
  - Endgeräte-Anbindung
  - insbesondere Mobile Endgeräte → WinCE

DotNet-Framework



DotNet-Framework Ziele

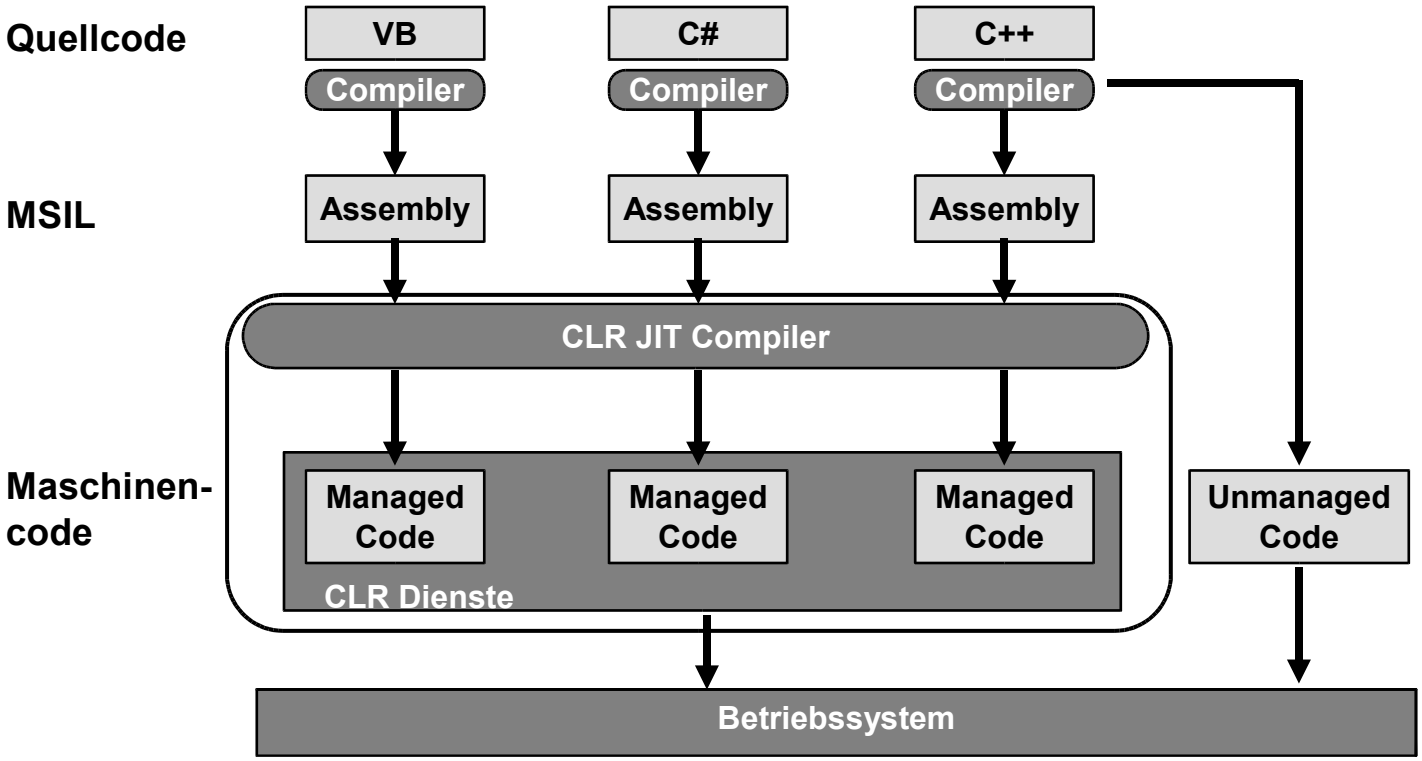
- Mehrsprachigkeit
  - C#, C++, VB.NET
  - „Offenlegung“ von DotNet (CLR und CTS) ermöglicht Anbieten weiterer Programmiersprachen durch Dritte*
- Einfache Entwicklung
  - Common Type System
    - Für alle einheitliches Objektmodell
    - → Gemeinsame Wurzel: `Object`
    - Vererbung zwischen unterschiedlichen Sprachen
  - Verteiltes Garbage Collection
- Einfache Installation
  - Selbsbeschreibende Komponenten
- Stabile Ausführung

* Offenlegung ermöglicht theoretisch weiterhin Plattformunabhängigkeit, allerdings ist „Base Class Library“ nicht frei und muss dementsprechend von „go-mono“, „Portable.NET“ etc. reengineert werden.

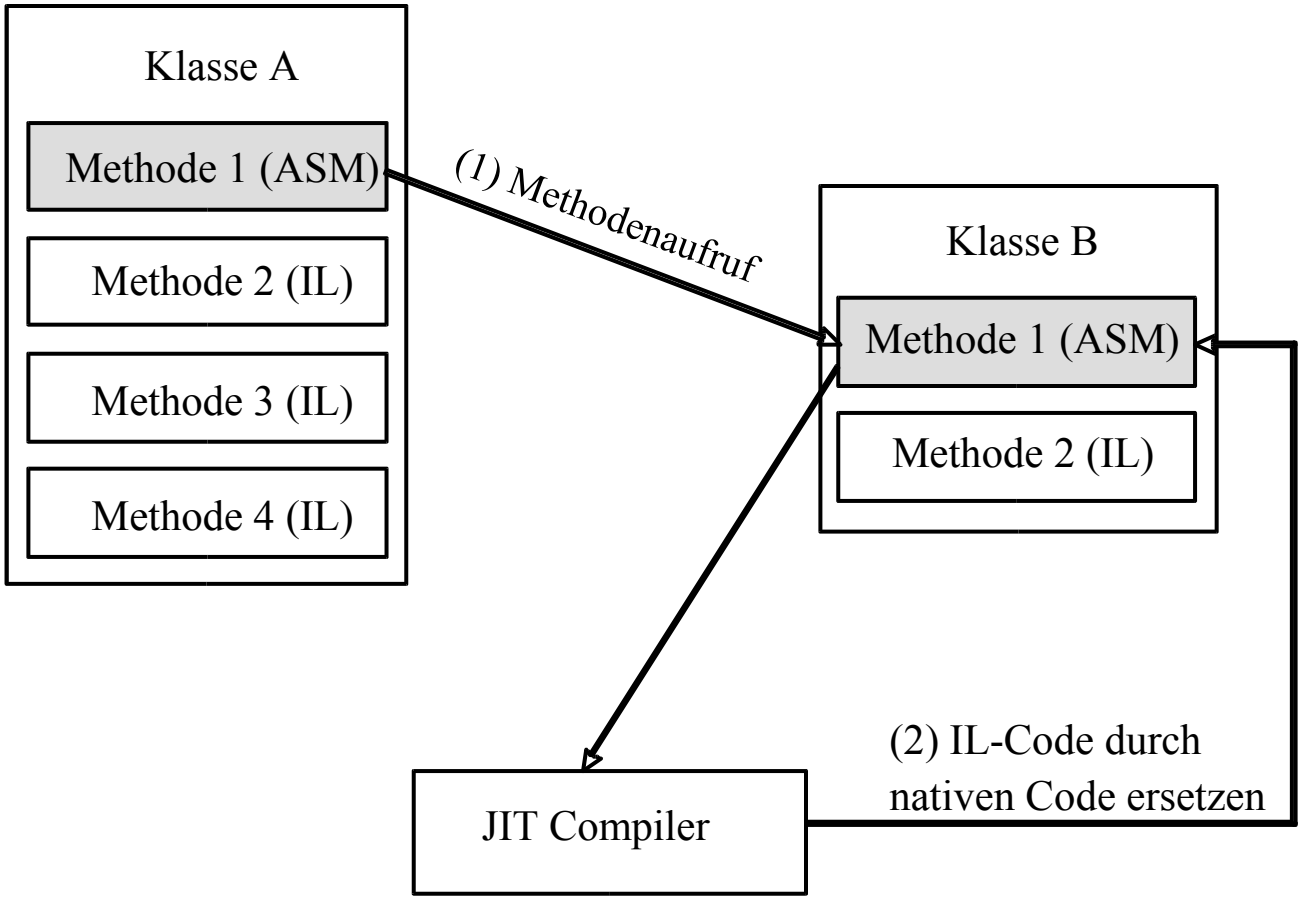
Sprachintegration

- Unterschiedliche Sprachen → unterschiedliche Typsysteme
  - DotNet: *einheitliches Typsystem* (CTS), um Integrationsschicht (XDR) zu vermeiden
  - Einheitliche Aufrufkonventionen
  - → Alle zu verwendenden Sprachen werden an DotNet angepasst
- Compiler erzeugen *Bytecode* (hier: MS Intermediate Language)
  - ermöglicht Vererbung von Klassen anderer Sprachen
  - → sehr gut dekompilierbar! ☺ (daher neuer Markt: „Obfuscation Engines“)
- Ausführung der Zwischensprache in Common Language Runtime (wie JRE)
  - → „*managed Code*“
  - → Nutzung der Dienste der Laufzeitumgebung
  - → JIT-Kompilierung nur der *jeweils benötigten Methoden*! (→ schnell!)
- Nur noch durch VisualC++ Maschinen-Code erstellbar
  - → „*unmanaged Code*“ (Verwendung von RT-Diensten *nicht* möglich)

DotNet Sprachintegration



Methodenaufruf (JIT)

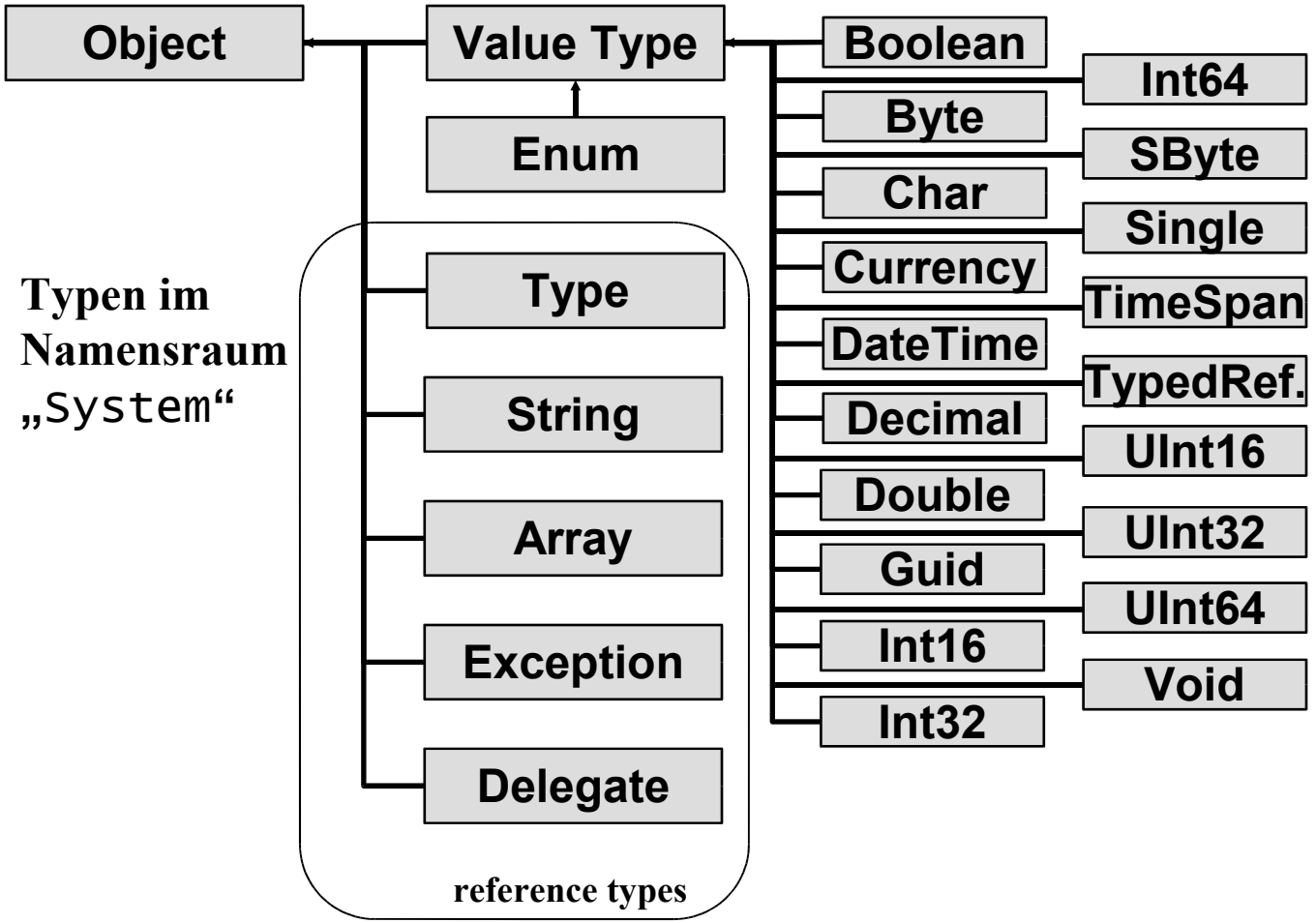


Nach Zimmermann /.NET/

Common Type System

- Einheitliches Typsystem für alle Sprachen
  - → Einheitliche Repräsentation,
  - → Einheitliche Größe eines Datentyps
  - → Keine Konvertierung notwendig
  - → Keine externe Beschreibung
- Zwei unterschiedliche Typklassen (analog Java):
  - *a) Werttypen* (value types)
    - → enthalten einen eigenen Wert
    - → Primitive Datentypen (analog: int, char etc.)
    - → Plus (zu Java): Stack-Typen (enum, struct)
  - *b) Referenztypen* (reference types)
    - → Eigentliche Objekte (enthalten Referenz auf Objekt)
    - → Können Wert „null“ annehmen
    - → Komplexe Datentypen (analog: Integer, String etc.)

Objektmodell



Verteilte Anwendungen mit DotNet
„Remoting“

- RPC/RMI-Mechanismus bei DotNet wird als „Remoting“* bezeichnet
- „Remoting-Framework“ stellt übliche VA-Dienste bereit:
  - Activation
  - Lifecycle Services („lifetime support“*)
  - Zugriff auf Sockets („channels“*)
  - Verteiltes Garbage Collection
  - Etc.
- Kommunikationsarten:
  - Zwischen DotNet-Komponenten, ohne SOAP: Binäres Protokoll über TCP-Socket
  - DotNet-Komponenten mit bel. anderen: SOAP über HTTP-Verbindung
- Verbindungen zwischen Komponenten heißen bei DotNet „*Channel*“*
- Stubs/Skeletons werden aufgeteilt in *Proxy*-Objekte und an „Channel“ gebundene „*Formatter*“*

*Bei diesen Begriffen handelt es sich um M$-Wortschöpfungen deren Gebrauch sicherlich bald Lizenrechtliche folgen haben wird.

3 Arten entfernter Objekte

- Serveraktivierte Objekte/Well known-Objekte (.EXEs)
  - *Single Call*-Objekte
    - Ein Objekt pro Anfrage
    - atomare Anfragen
    - aka stateless session beans
  - *Singleton* Objekte
    - Mit *Conversational State*
    - dauerhaft oder *„Lease based Lifetime“* (→ Lebensdauermanager)
    - Sind einzigartig (singleton) → existieren nur einmal pro VM
    - Sinnvoll für komplexe Objekte, deren Instanziierung „teuer“ ist
    - Ähnlich entity beans (ohne Persistenz, keine Lastverteilung)

  Beide: Anmeldung im BS bei Serverstart durch Server

- Clientaktivierte Objekte (.DLLs)
  - Aktivierung bei vorliegendem Request
  - Generierung von clientseitigen Proxies aus „*ObjRef*“
  - *„Lease based Lifetime“*

Entfernte Zugriffe

- Austausch von Daten innerhalb einer „*Application Domain*“:
  - Alle Objekte (reference types) „Passed by reference“
  - Alle einfachen Datentypen (value types) „passed by value“

- Austausch über Grenzen der „*Application Domain*“ hinweg immer „by value“
  → Entfernt zugreifbare Objekte müssen das `[serializable]` Attribut tragen oder das Interface `ISerializable` implementieren

- Lokale, nicht serialisierbare Objekte sind außerhalb der „Application Domain“ nicht zugreifbar und daher „*nonremotable*“*

* Siehe vorhergehende Folie

Remote Objects

- Abgeleitet von `MarshalByRefObject`
  - Übertragung eines Proxy-Objekts zum Client bei Request
  - Alle Aufrufe finden am Proxy-Objekt statt und sehen lokal aus
  - Proxy-Objekt marshallt / serialisiert Parameter und kommuniziert mit Peer

- Abgeleitet von `MarshalByValue`
  - Übertragung einer vollständigen Kopie des Objektes und Nutzung auf Clientseite

Proxy Objekte

Transparent Proxy vs. *Real Proxy*

1. Bei Aktivierung eines Remote Objects legt CLR auf Clientseite zunächst einen *Transparent Proxy* an, dessen Methoden Client aufruft
2. Bei Aufruf einer Methode des Remote Objekt am Transparent Proxy wird durch CLR überprüft, ob Remote Objekt *lokal* vorliegt oder *entfernt* ist

a) Liegt Remote Objekt lokal vor werden *simple lokale Methodenrufe* durchgeführt

d) Ist Remote Objekt entfernt werden

5) Parameter in ObjRef *gemarshallt*,

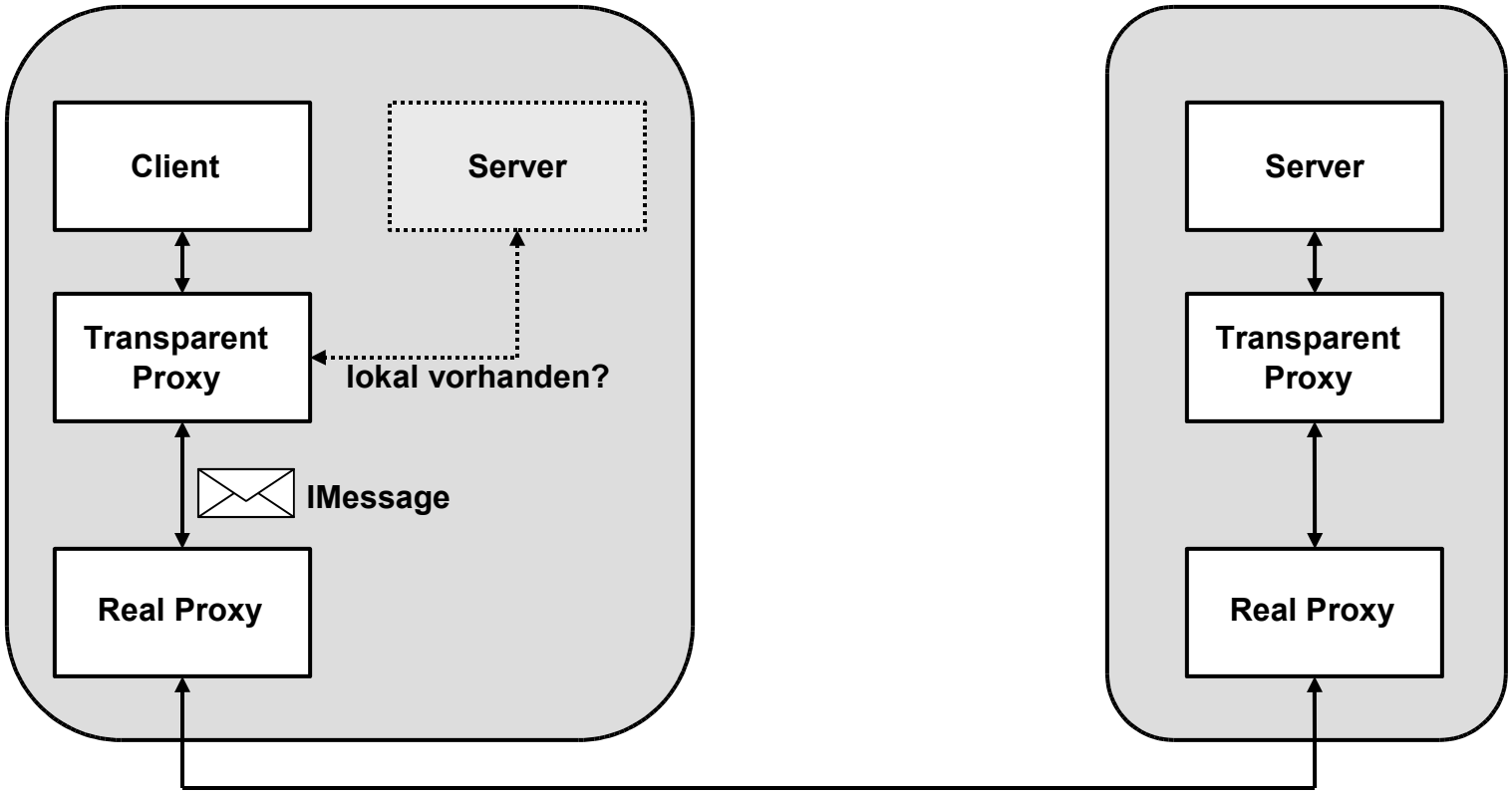
6) In *IMessage*-Objekt verpackt,

7) Ein *RealProxy* generiert,

8) An diesen das IMessage-Objekt übergeben und von diesem die *Kommunikation* übernommen

- Beispiel: Die Bankanwendung

Transparent und Real Proxy



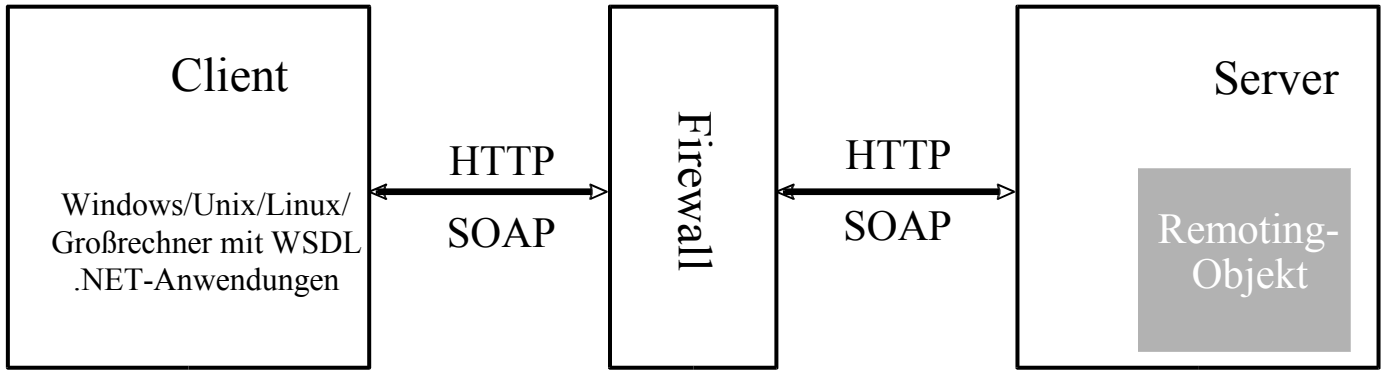
Channel

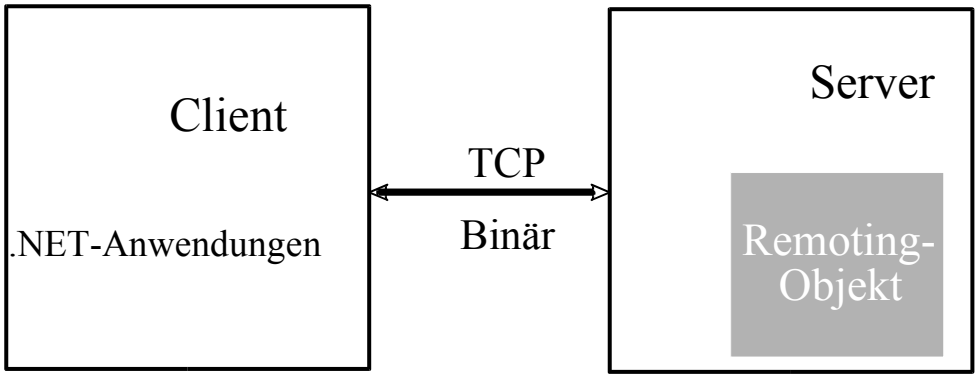
- Channel transportieren Messages
  - Setzen auf verschiedenen Schichten auf
  - Benötigen „Formatter“ fürs Kodieren der Nachrichten

- unterschiedliche Arten:
  - HTTP-Channel
    - Transportieren SOAP-Nachrichten
    - für Kommunikation im Internet gedacht (SOAP, Port 80, Firewalls! ☺)
    - Nur Simplex oder Request-Response Paare, keine permanente Verbindung

  - TCP-Channel
    - entsprechen permanenter Socket-Verbindung
    - nur zwischen DotNet-Komponenten möglich
    - schnelle Kommunikation
    - für LAN-Bereich gedacht

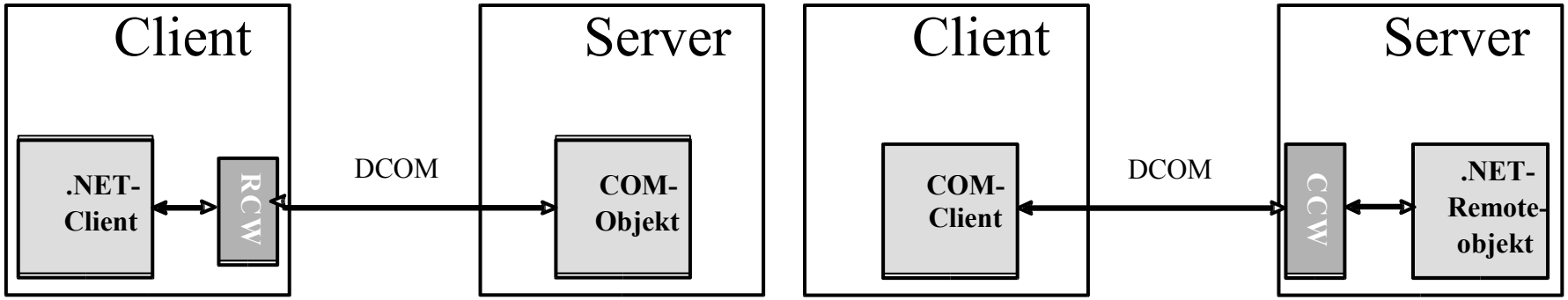
Channel-Kommunikation





COM-Integration

- Nutzung von COM-Komponenten durch .NET-Anwendungen über „Runtime Callable Wrapper“ (Wrapper für die COM-Komponente)
- Nutzung von .NET-Komponenten durch COM-Anwendungen über „COM Callable Wrapper“ (Wrapper für die DotNet-Komponente)



Kommunikationsszenarien

Client	***Server***	Nutzlast	Protokoll
.NET-Komponente	.NET-Komponente	SOAP/XML	HTTP
.NET-Komponente	.NET-Komponente	Binär *(nur lokal!)*	TCP
Verwaltet/nicht verwaltet	.NET-Webdienst	SOAP/XML	HTTP
.NET-Komponente	Nicht verwaltete herkömmliche COM-Komponente	NDR (Network Data Representation, Netzwerkdatendarstellung) über RCW	DCOM
Nicht verwaltete herkömmliche COM-Komponente	.NET-Komponente	NDR über CCW	DCOM

Beispiel: Die Bankanwendung

- Remote Object „Bank“
  - Schlichte Implementierung einer veränderlichen Klassenvariablen
  - Hat Methoden: `einzahlen(double betrag)`, `abheben(double betrag)` und `get_kontostand()`
- Hostinganwendung (Server)
  - Binden des Remote Objects an CLR
  - Bereitstellen der „Channels“
  - Aktivierung des Remote Objects als Singleton
- Client
  - Einfacher Zugriff auf Remote Object

→ Keine Schnittstellendefinition (CTS...)

Das Remote Object: Die Bank

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace BankBeispiel{
    public class BankServer : MarshalByRefObject //von MarshalByRefObject ableiten
    {
          double kontostand;
          public BankServer(){
                kontostand = 0.0;
          }
          public bool einzahlen (double betrag){
                [...]
          }
          public bool abheben (double betrag){
                [...]
          }
          public double get_kontostand(){
                [...]
          }
    }
}
```

Server

```
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) {

          TcpChannel chan = new TcpChannel(8085);
          // Channel-Objekt erstellen und registrieren
          ChannelServices.RegisterChannel(chan); // Objekt registrieren (Port binden)

          RemotingConfiguration.RegisterWellKnownServiceType(Type.GetType(„
                    BankBeispiel.BankServer,bank"), "Bank", WellKnownObjectMode.Singleton);
      System.Console.WriteLine("<ENTER> zum Beenden...");
      System.Console.ReadLine();
      return 0;
    }
  }
}
```

Client

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

namespace BankBeispiel{
  public class Client{
    public static int Main(string [] args){
      if (args.Length == 1){
            TcpChannel chan = new TcpChannel(); // Channel-Objekt anlegen
            ChannelServices.RegisterChannel(chan); // Channel registrieren

            BankServer bank =
                    (BankServer)Activator.GetObject(typeof(BankBeispiel.
                    BankServer), "tcp://" + args[0] + ":8085/Bank");
            // Remoteobjekt referenzieren

          bank.einzahlen (Convert.ToDouble(Console.ReadLine()));
          bank.abheben (Convert.ToDouble(Console.ReadLine()));
          Console.WriteLine ("Aktueller Kontostand: " + bank.get_kontostand());
      }
  }
}
```