Verteilte Syteme, 3. Teil

C. Kommunikation II

(Aufruf von Code in anderen Prozeßräumen)

Remote Procedure Call (RPC) – Allgemeines Konzept (1)

- **Grundidee**

  ◇ Die elementare **Kommunikation** zwischen **verteilten Prozessen** arbeitet **botschaftsorientiert**.(Nachrichtenaustausch) Sie beruht auf dem **Paradigma der Ein- und Ausgabe** :
  → Verwendung von **I/O-Funktionen** wie **`send()`** und **`receive()`**.

  ◇ Die Programmierung derartiger Kommunikationsbeziehungen ist **relativ umständlich** und **aufwendig**, insbesondere dann, wenn sie in Client-Server-Architekturen zur Aktivierung von Server-Code durch den Client eingesetzt werden. Dies **widerspricht** den Bestrebungen, eine **verteilte Anwendung wie eine zentrale Anwendung erscheinen** zu lassen und entsprechend programmieren zu können. :
  → Der Aufruf von Prozeduren eines entfernten Serverprozesses sollte genauso einfach möglich sein wie der Aufruf von lokalen Prozeduren.

  ◇ Zur Lösung dieses Problems wurde 1984 von **Birrel** und **Nelson** das **Konzept des entfernten Prozeduraufrufs** (***Remote Procedure Call***, **RPC**) entwickelt :
  Ein RPC-System erlaubt es, Prozeduren auf anderen Rechnern prinzipiell in der gleichen Art und Weise wie eine lokale Prozedur aufzurufen, einschließlich der Übergabe von Parametern und der Rückgabe eines Funktionswertes. Ein derartiger entfernter Aufruf basiert auf den Mechanismen eines **synchronen entfernten Dienstaufrufs**. Allerdings bleibt die gesamte hierfür notwendige Netzwerkkommunikation sowie jeglicher damit verknüpfter I/O-Mechanismus vor dem Programmierer verborgen.
  Vielmehr sieht ein *Remote Procedure Call* prinzipiell genauso aus wie ein lokaler Prozeduraufruf.
  Sein Format (Aufruf-Schnittstelle) ist durch die Signatur von Prozeduren gegeben.

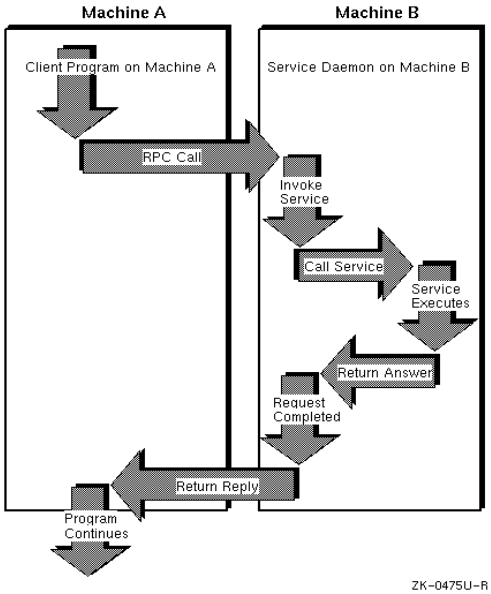
  ◇ Wie bei einem lokalen Prozeduraufruf blockiert – bei traditionellem RPC - der Aufrufer bis der Aufruf beendet ist, d.h. hier : der Client wartet bis er vom Server eine Rückmeldung über das Ende der Prozedurbearbeitung erhält. Damit kann für die Programmierung verteilter Anwendungen das gleiche Paradigma wie für lokale Programme angewendet wird.

- **Definition**

  nach Nelson :

  > **RPC** ist die **synchrone** **Kontrollfluß- und Datenübergabe** in Form von **Prozeduraufrufen** und von aktuellen Parametern zwischen Programmen in **unterschiedlichen Adreßräumen** über einen **schmalen Kanal**

- **Prinzipielle Ablaufkontrolle bei RPC**



aus :
Programming with ONC RPC
Digital Equipment Co. 1996

Remote Procedure Call (RPC) – Allgemeines Konzept (2)

- **Unterschiede zwischen RPC-Aufrufen und lokalen Aufrufen**

  - ▻ Bei RPC ergeben sich durch das beteiligte **Kommunikationssystem** zusätzliche **Fehlerquellen**
  - ▻ Bei RPC liegen Aufrufer und aufgerufene Prozedur in unterschiedlichen Prozessen und damit in **unterschiedlichen Adreßräumen**
    → großer Einfluß auf Parameterübergabe (z.B. ist nur Wert-Übergabe direkt möglich)
    → keine gemeinsamen globalen Variablen
  - ▻ Bei RPC haben Aufrufer und aufgerufene Prozedur **keine gemeinsame Ausführungsumgebung**
    → Einfluß auf die Verwendung von Resourcen (z.B. Dateien !)
  - ▻ Bei RPC haben die beteiligten Prozesse **unabhängige Lebensdauern**
    → Client und Server können unabhängig voneinander terminieren
    → komplexe Fehlersemantik
  - ▻ Wegen des Kommunikations-Overheads **dauern** bei RPC **Prozeduraufrufe** um Größenordnungen **länger** als im lokalen Fall.
  - ▻ Um die Möglichkeiten der Nebenläufigkeit auszunutzen wurden auch **asynchrone RPC-Systeme** realisiert
    → In einem derartigen Fall sinkt die Ähnlichkeit zu lokalen Prozeduraufrufen stark, **anderes Aufruferverhalten** erforderlich

- **Abgrenzung des RPC zur Interprozeßkommunikation mittels Nachrichtenaustausch**

RPC	Nachrichtenaustausch
asymmetrische Kommunikationsbeziehung (Client → Server)	prinzipiell symmetrische Kommunikations-beziehung
i.a. synchron	asynchron oder synchron
1 primitive Operation (RPC-Aufruf)	2 primitive Operationen (`send, receive`)
Nachrichten werden durch das RPC-System konfiguriert	Nachrichten müssen explizit vom Programmie-rer aufgebaut werden
es kann immer nur auf die Beendigung eines "ausstehenden" RPC-Aufrufs gewartet werden (bei synchronem RPC)	im asynchronen Fall kann grundsätzlich auf das Eintreffen mehrerer (Antwort-)Nachrichten gewartet werden

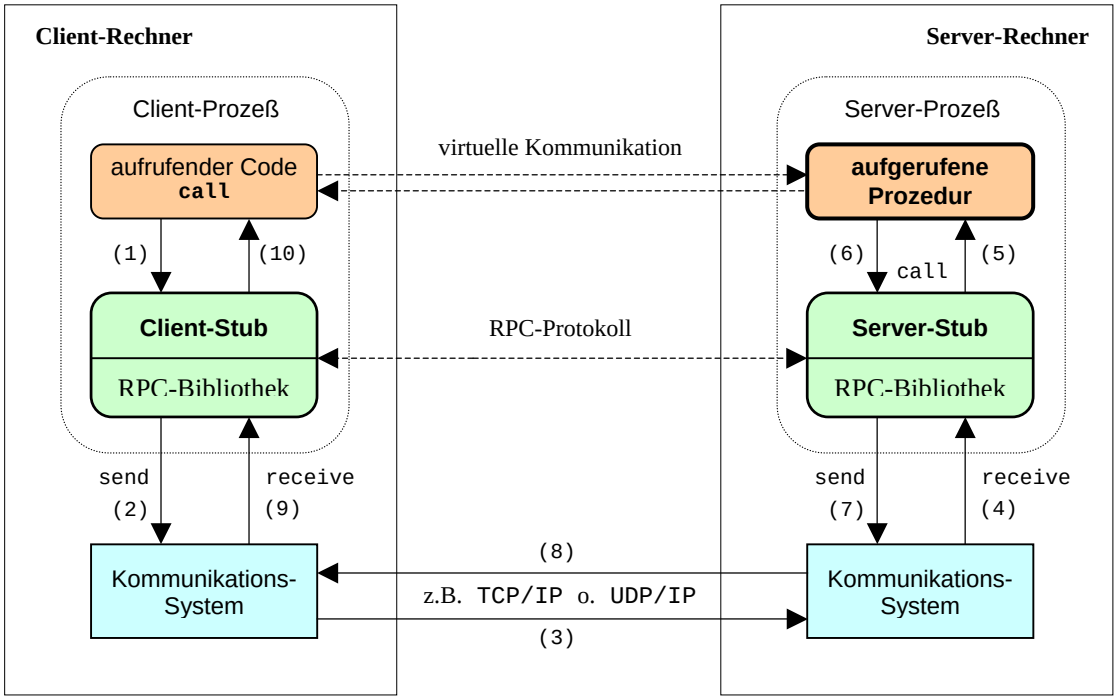
- **Realisierte RPC-Systeme**

  es existieren zahlreiche – meist firmenspezifische – Umsetzungen des RPC-Konzepts.
  Einige Beispiele sind :

  - ▻ ONC-RPC von Sun
  - ▻ HP-RPC
  - ▻ Modula V RPC
  - ▻ Apollo-RPC
  - ▻ DCE RPC (DCE – Distributed Computing Environment)
    von der Open Software Foundation (OSF) als plattformunabhängiges System konzipiert

Remote Procedure Call (RPC) – Allgemeines Konzept (3)

- **Funktionsprinzip des RPC**



◇ Die Implementierung des RPC erfolgt mit Hilfe von lokalen **Stellvertreter-Prozeduren**, den sogenannten **Stubs** :

- ▻ Der **Client-Sub** ist ein **Stellvertreter** der **entfernten Server-Prozedur** auf der Client-Seite
- ▻ Der **Server-Stub** fungiert als **Stellvertreter** des aufrufenden **Client-Codes** auf der Server-Seite.

Die Stubs **verbergen** das **Entferntsein** des Codes der jeweils anderen Seite.

◇ Ein **RPC-Aufruf** läuft grundsätzlich in den folgenden **10 Schritten** ab :

(1) Der Client-Code ruft den lokalen Client-Stub der entfernten Server-Prozedur auf.
Der Client-Stub verhält sich dem Client-Code gegenüber wie die eigentliche Serverprozedur.
Er verpackt die übergebenen Parameter des Prozeduraufrufs in eine Nachricht entsprechend dem RPC-Protokoll.
Dieser Serialisierungsprozeß wird ***Marshalling*** genannt.

(2) Der Client-Stub übergibt die erzeugte Nachricht an das Kommunikationssystem des Clients.

(3) Das Kommunikationssystem überträgt die Nachricht zum Server-Rechner unter Nutzung eines verbindungslosen oder verbindungsorientierten Transportschichtprotokolls (meist UDP/IP oder TCP/IP).

(4) Das Kommunikationssystem des Server-Rechners gibt die Nachricht an den Server-Stub weiter.
Der Server-Stub entpackt die in der Nachricht enthaltenen Parameter gemäß dem RPC-Protokoll.
Diesen Vorgang bezeichnet man als ***Demarshalling***.

(5) Der Server-Stub ruft unter Übergabe der Parameter die eigentliche Server-Prozedur auf (lokaler Aufruf !)

(6) Die Server-Prozedur kehrt nach Abarbeitung unter Rückgabe des Funktionswertes zum Server-Stub zurück.
Dieser verpackt den Rückgabewert in eine Nachricht gemäß dem RPC-Protokoll.

(7) Der Server-Stub übergibt die Rückgabewert-Nachricht an das Kommunikationssystem des Servers.

(8) Das Kommunikationssystem überträgt – aufsetzend auf einem Transportschichtprotokoll - die Nachricht zum Client-Rechner

(9) Das Kommunikationssystem des Clients übergibt die Nachricht an den wartenden Client-Stub.
Dieser entpackt den in der Nachricht enthaltenen Rückgabewert gemäß dem RPC-Protokoll.

(10) Der Client-Stub gibt den Rückgabewert an den wartenden Client-Code zurück

Remote Procedure Call (RPC) – Allgemeines Konzept (4)

- **Aufgaben der Client-Stub-Prozedur**
  - ▻ Darstellung der Parameter im Übertragungsformat (Serialisierung, *Marshalling*)
  - ▻ Aufbau der Aufruf-Nachricht entsprechend dem RPC-Protokoll
  - ▻ Übergabe der Aufruf-Nachricht an das Kommunikationssystem
  - ▻ Warten auf die Antwort-Nachricht vom Kommunikationssystem
  - ▻ Extraktion der Rückgabewerte aus der Antwort-Nachricht und Darstellung derselben im maschinenspezifischen Format (Deserialisierung, *Demarshalling*)
  - ▻ Rückkehr zum aufrufenden Clientcode unter Übergabe der Rückgabewerte
  - ◇ **Prinzipieller Aufbau der Client-Stub-Prozedur** :

```
PROCEDURE proc_clnt(Parameter)
BEGIN
  Zusammensetzen der Aufruf-Nachricht AN aus den Parametern;
  send(serverAdresse, AN);
  receive(RN);               // RN – Antwort-Nachricht
  Extrahieren Rückgabewerte aus RN;
  Return(Rückgabewerte);
END;
```

- **Aufgaben der Server-Stub-Prozedur**
  - ▻ Extraktion der Parameter aus der Aufruf-Nachricht und Darstellung derselben im maschinenspezifischen Format (Deserialisierung, *Demarshalling*)
  - ▻ Aufruf der gewünschten Serverprozedur mit Übergabe der Parameter
  - ▻ Darstellung der Rückgabewerte der aufgerufenen Prozedur im Übertragungsformat (Serialisierung, *Marshalling*)
  - ▻ Aufbau der Antwort-Nachricht entsprechend dem RPC-Protokoll
  - ▻ Übergabe der Antwort-Nachricht an das Kommunikationssystem
  - ◇ **Prinzipieller Aufbau der Servert-Stub-Prozedur** :

```
PROCEDURE proc_srv
BEGIN
  receive(AN);
  Extrahieren Parameter aus AN;
  Call proc(Parameter);
  Zusammensetzen der Antwort-Nachricht RN aus den Rückgabewerten
  send(clientAdresse, RN);
END;
```

- **Prinzipieller Aufbau einer RPC-Aufruf-Nachricht** (Sun-RPC)
  - ▻ Transaktions-Identifier
  - ▻ Nachrichtentyp `CALL` (für Antwort-Nachricht : `REPLY`)
  - ▻ RPC-Versions-Nr.
  - ▻ Kennung der aufzurufenden entfernten Prozedur (Programm-Nr., Versions-Nr., Prozedur-Nr.)
  - ▻ Authentifizierungsinformation
  - ▻ Parameter für die aufzurufende entfernte Prozedur

Remote Procedure Call (RPC) – Parameterübergabe (1)

- **Problem**

  Da bei RPC der aufrufende Code und die aufgerufene entfernte Prozedur unterschiedliche Adreßräume verwenden, kann eine Datenübergabe (Parameter) nicht direkt über einen lokalen Stack oder Register erfolgen
  Vielmehr müssen sie sinnvoll vom Client-Adreßraum in den Server-Adreßraum überführt werden.
  Entsprechendes gilt für die Rückgabewerte.
  Diese Überführung erfolgt mit Hilfe von Stubs und Nachrichten.

- **Wert-Parameter**

  - Im lokalen Fall wird der übergebene Parameter zu einer lokalen Variablen der Prozedur, die mit dem Parameterwert initialisiert ist (*Call by Value*).
    Eine eventuelle Änderung des Werts dieser Variablen durch die Prozedur wirkt sich außerhalb derselben nicht aus.
  - Bei RPC genügt es daher nur den Wert des jeweiligen Parameters in die Nachricht aufzunehmen und zu übertragen.
  - Achtung :
    Bei Arrays (und gegebenenfalls anderen komplexeren Datenstrukturen) müssen tatsächlich die Werte aller Komponenten übertragen werden und nicht nur ein Zeiger auf den Anfang.
    Im lokalen Fall wird eine derartige implizite Referenz-Übergabe von manchen Compilern aus Effizienzgründen vorgenommen (z.B. Übergabe von Arrays in C/C++).

- **Referenz-Parameter**

  - Bei Referenzparametern (*Call by Reference*) werden im lokalen Fall tatsächlich Adressen übergeben.
  - Eine derartige einfache Adreßübergabe ist bei RPC problematisch :
    Eine übergebene Adresse wird im Server-Adreßraum i.a. eine andere Bedeutung als im Client-Adreßraum haben
    → keine sinnvolle Dereferenzierung möglich.
  - Statt der Adresse muß der unter der Adresse abgelegte Wert (bzw Wertefolge) übertragen werden. und dann in eine entsprechende Variable im Server-Stub kopiert werden.
    Falls die durch den ursprünglichen Parameter referierten Daten in der entfernten Prozedur verändert werden sollen, müssen sie vom Server-Stub wieder in die Antwort-Nachricht zurückkopiert und zurück zum Client übertragen werden.
    In diesem Fall wird *Call by Reference* durch *Call by Copy/Restore* ersetzt.
  - Daten, die Adressen von anderen Daten enthalten (z.B. dynamische Datenstrukturen) müssen vollständig dereferenziert ("planarisiert") werden und komplett als Bytefolge übertragen werden.
  - Alternativ kann auch tatsächlich ein Zeiger an den Server übertragen werden.
    Jede Dereferenzierung des Zeigers muß dann aber gesondert per Nachricht vom Client angefordert werden.
    → sehr aufwändige und ineffiziente Methode, die trotzdem in einigen RPC-Systemen implementiert ist.

- **Rückgabewerte**

  - Für Rückgabewerte gelten sinngemäß die gleichen Überlegungen wie für Parameter.
  - Sinnvollerweise sollten nur Werte und keine Zeiger oder Referenzen zurückgegeben werden.

- **Weitere Probleme :**

  - In unterschiedlichen Systemen können Daten mit gleicher Bedeutung unterschiedlich repräsentiert werden :
    - Verwendung unterschiedlicher Zeichencodes (z.B. ASCII-Code, Unicode)
    - Verwendung unterschiedlicher Zahlendarstellungen (z.B. 16-Bit-Integer, 32-Bit-Integer)
    - Ablage von Mehrbytewerten in unterschiedlicher Reihenfolge (*little endian, big endian*)
  - Bei Einsatz unterschiedlicher Programmiersprachen können die zur Verfügung stehenden Datentypen unterschiedlich sein (z.B. Datentyp bool, Ersatz durch int)

Remote Procedure Call (RPC) – Parameterübergabe (2)

- **Transfersyntax**

  ◇ Das Packen/Entpacken von Parametern und Rückgabewerten in/aus Nachrichten setzt voraus, daß Client-Stub und Server-Stub
  - ▻ über die gleiche Schnittstellenbeschreibung für die aufzurufende Server-Prozedur verfügen
  - ▻ die gleichen Datentypen und die gleiche Darstellung der verschiedenen Typen verwenden.

  ◇ Zwischen unterschiedlichen Systemen können Unterschiede bestehen bezüglich
  - implementierten Datentypen
  - Zeichencodierung und Zahlendarstellung
  - Byteordnung

  Zum Ausgleich dieser Unterschiede zwischen Server und Client müssen klare Festlegungen bezüglich der für die Übertragung der Parameter und Rückgabewerte verwendeten Syntax getroffen werden.
  Hierfür existieren prinzipiell zwei Verfahren.

  ◇ **Symmetrisches Verfahren**
  - ▻ Client und Server verwenden für die **Übertragung** ein **einheitlich festgelegtes** – kanonisches – **Format**.
  - ▻ Alle Parameter und Rückgabewerte müssen vor der Übertragung aus der systemspezifischen Darstellung des Sendeprozesses in dieses Format umgesetzt werden.
    Nach dem Empfang sind sie in die systemspezifische Darstellung des Empfangsprozesses umzuwandeln.
  - ▻ Sowohl Client als auch Server benötigen hierfür eigene Konvertierungsroutinen
  - ▻ Die Formatumsetzung gehört zu den Aufgaben von Client- und Server-Stubs
  - ▻ Dieses Verfahren wird beispielsweise bei Sun-RPC angewendet (*External Data Representation*, XDR)
  - ▻ **Nachteil** : Wenn Client und Server über das gleiche – vom Übertragungsformat aber abweichende - Format verfügen, finden jeweils zwei eigentlich überflüssige Konvertierungen statt.
  - ▻ **Vorteil** : Der Server braucht sich nicht um Formate anderer Systeme zu kümmern

  ◇ **Asymmetrisches Verfahren**
  - ▻ Die Clients übertragen die Parameter in ihrem jeweiligen systemspezifischen Format.
    In der Nachricht muß eine Information über das verwendete Format mitgeschickt werden.
  - ▻ Der Server muß die empfangenen Parameter in sein systemspezifisches Format umwandeln.
  - ▻ Vor der Übertragung der Antwort-Nachricht muß der Server die Rückgabewerte in das systemspezifische Format des Clients umwandeln.
  - ▻ Hier benötigt nur der Server entsprechende Umwandlungsroutinen, allerdings jeweils für jedes mögliche Client-Format
  - ▻ **Vorteile** : Clients müssen keine Formatumwandlungen durchführen
    Es finden keine überflüssigen Konvertierungen statt
  - ▻ **Nachteile** : Der Server muß sich um die Formate aller möglichen Clients kümmern.
    Das in der Übertragung verwendete Format muß in der Nachricht mit angegeben werden
  - ▻ Die Asymmetrie kann auch zwischen Server und Clients vertauscht sein. :
    Nur die Clients führen eine Konvertierung in das systemspezifische Format des Servers durch.

Remote Procedure Call (RPC) – Parameterübergabe (3)

- **Beschreibung der RPC-Schnittstellen**

  ◇ Um den in eine Nachricht verpackten Prozeduraufruf korrekt interpretieren und behandeln zu können, muß die **Aufruf-schnittstelle** exakt beschrieben werden.
  Diese Beschreibung, die sogenannte **Signatur**, umfaßt
  - ▻ den Namen der Prozedur
  - ▻ die Typen und Art der Parameter (Eingabe, Ausgabe, Ein-/Ausgabe)
  - ▻ den Typ des Rückgabewerts
  - ▻ eventuell Ausnahmebehandlungsvorschriften für Fehlerfälle.

  ◇ Die Beschreibung muß sowohl für den Client als auch den Server verfügbar sein.

  ◇ Zur Beschreibung derartiger Schnittstellen wurden spezielle **Schnittstellenbeschreibungssprachen** entwickelt.
  Die Syntax dieser Sprachen lehnt sich häufig an die Syntax gängiger Programmiersprachen an (z.B. C)

  ◇ Diese Sprachen werden zum Erstellen von **RPC-Spezifikationsdateien** eingesetzt.
  Mit ihnen lassen sich nicht nur die Signaturen entfernt aufzurufender Server-Prozeduren formulieren, sondern auch **zusätzliche Vereinbarungen** angeben, die eine **automatische Erzeugung** der **Stub-Prozeduren** ermöglichen

  ◇ Die Generierung der Stub-Prozeduren erfolgt mittels sogenannter **RPC-Compiler (RPC-Generatoren).**
  Ein RPC-Compiler erzeugt aus einer RPC-Spezifikationsdatei neben anderen auch die Quelldateien für die Stub-Prozeduren in der für die Entwicklung vorgesehenen Programmiersprache.

  ◇ **Beispiele für Schnittstellenbeschreibungssprachen** :
  - von der ISO : ASN.1 (Abstract Syntax Notation)
  - bei ONC-RPC von Sun : RPCL (RPC Language), eine Erweiterung von XDR (External Data Representation)
  - bei DCE-RPC (OSF) : IDL (Interface Definition Language)

- **Beispiel einer RPC-Spezifikationsdatei** (in RPCL für Sun-RPC)

  Zwei Integer-Zahlen sollen mittels einer entfernt aufzurufenden Funktion `add()` addiert werden.
  Die beiden zu addierenden Werte werden in einer Structure zu einem einzigen Parameter zusammengefaßt

```
/*
  add.x   --  Schnittstellen-Definition für ein RPC-Beispiel
*/

struct params
{ int a;
  int b;
};

program ADD_PROG
{
  version ADD_VERS
  {
    int ADD(params) = 1;
  } = 1;
} = 0x20000001;
```

Remote Procedure Call (RPC) – Binden von RPC-Anwendungen

• Binden

◇ Bevor ein Client einen RPC-Aufruf durchführen kann, muß festgelegt werden, welcher Server-Prozeß den Aufruf bedienen kann.
Gegebenenfalls muß zusätzlich auch der Rechner, auf dem der Server-Prozeß läuft, ermittelt werden.

◇ Diesen Vorgang der Zuordnung zwischen aufrufenden Client-Prozeß und ausführenden Server-Prozeß bezeichnet man als Binden (*Binding*)

◇ Das Binden kann auf unterschiedliche Arten erfolgen

• Arten des Bindens

◇ **Statisches Binden zur Compilezeit**

▻ Die vollständige Serveradresse ist fest im Client-Programm codiert.

▻ sehr effizient, aber auch sehr inflexibel : Bei Änderungen ist eine Neukompilierung erforderlich

◇ **Semi-Statisches Binden beim Programmstart**

▻ Übergabe der Serveradresse als Programmparameter

▻ effizient und weniger inflexibel als statisches Binden, nicht immer anwendbar

◇ **Dynamisches Binden**

- **zur Initialisierungszeit** (Start des Clients)
- **zur Laufzeit** (unmittelbar vor jedem RPC-Aufruf)

▻ Ermittlung der Serveradresse

- durch Befragen aller in Frage kommenden Server (Broadcast oder Multicast)
- durch Nachfragen bei einem Namensdienst (*Name Service*, Binder)
  bzw durch Inanspruchnahme eines Maklerdienstes (Vermittler, *Broker*, *Trader*)

▻ sehr flexibel, vor allem beim Binden zur Laufzeit : unmittelbare Berücksichtigung von Änderungen (Migration des Servers, dynamisches Ersetzen des Servers, alternativer Server bei Serverausfall) aber aufwändiger

• Verwendung eines Namensdienstes (Binders)

◇ Externe Instanz, bei der jeder **Server** bei seinem Start **sich** und die von ihm zur Verfügung gestellten (**exportierten**) **Prozeduren registriert**. (→ **Exportieren** von Schnittstellen)

◇ Der Binder verwaltet alle bei ihm registrierten Server mit ihren exportierten Prozeduren in einem Verzeichnis.
Je nach System kann der Binder zentral für die Server-Prozesse einer Gruppe von Rechnern zuständig sein oder nur die Server-Prozesse eines Rechners, der dann dem Client-Prozeß bekannt sein muß, verwalten.

◇ Die gespeicherte Information ist typischerweise ein Tupel bestehend aus :
<Server-(Programm-)Name, Versionsnummer, Server-ID, Handle (Serveradresse),>
Zusätzlich können auch noch weitere Informationen (z.B. bezüglich einer Authentifizierung) enthalten sein

◇ Ein Server kann sich beim Binder auch wieder **deregistrieren**

◇ Ein Client, der einen bestimmten Dienst (Prozedur) benötigt, erfragt beim Binder einen geeigneten Server (*lookup*). Falls ein derartiger Server registriert ist, bekommt er vom Binder einen Handle, über den er die Verbindung zum Server aufbauen kann, sowie die Server-ID (→ **Importieren** von Schnittstellen)

◇ Binder und ihre Verzeichnisse sind unter Umständen verteilt ausgelegt und/oder repliziert implementiert
→ Vermeidung eines "Flaschenhalses" in größeren Systemen,
aber Erhöhung des Verwaltungs- und Kommunikationsaufwands für Registrierung und Deregistrierung

Remote Procedure Call (RPC) – Erzeugung einer RPC-Anwendung (1)

- **Einsatz von RPC-Generatoren**

  ◇ RPC-Generatoren (RPC-Compiler) erleichtern die Erzeugung von RPC-Anwendungen beträchtlich

  ◇ Aus der in einer RPC-Spezifikationsdatei enthaltenen Schnittstellenbeschreibung erzeugen sie die für die Entwicklung von Server- und Clientprozeß benötigten "Interface-Dateien" (als Quelldateien in der jeweils verwendeten Programmiersprache)

  - ▻ eine Header-Datei, die sicherstellt, daß Server- und Client-Programm unter Verwendung derselben benutzerdefinierten Typ-, Daten- und Funktionsvereinbarungen entwickelt werden.
  - ▻ Dateien mit dem Quellcode für Client- und Server-Stubs
  - ▻ je nach System weitere Quelldateien (z.B. mit Vorschlag zur Funktionalität des Server-Programms)

  Beispiel : Bei **Sun-RPC** werden vom RPC-Generator `rpcgen` aus der RPC-Spezifikationsdatei `myprog.x` folgende C-Quell-Dateien erzeugt :

  - ▻ eine C-Header-Datei `myprog.h`
    Sie enthält die benötigten Typ-und Konstantendefinitionen sowie Funktionsdeklarationen
  - ▻ eine C-Quell-Datei `myprog_svc.c` mit der Definition des Server-Stubs und des Server-Hauptprogramms.
    Es wird eine Server-Stub-Funktion pro Server-Programm erzeugt.
  - ▻ eine C-Quell-Datei `myprog_clnt.c` mit der Definition der Client-Stubs.
    Für jede vom Server-Programm implementierte RPC-Funktion enthält die Datei eine eigene Client-Stub-Funktion.
  - ▻ falls als Parameter u/o Rückgabewerte selbstdefinierte Datentypen (z.B. structures) verwendet werden :
    eine C-Quelldatei `myprog_xdr.c` mit der Definition der von Client- und Server-Stubs verwendeten Datenkonvertierungsroutinen

  ◇ Der **Anwendungsprogrammierer** muß nur noch die verbleibenden Programmbestandteile für Server und Client implementieren.
  Bei **Sun-RPC** sind dies :

  - ▻ **Client-Hauptprogramm**
  - ▻ die eigentlichen **RPC-Funktionen** des Servers

- **Erzeugung des Server-Programms**

  Das **Server-Programm** entsteht durch Compilieren und Linken

  - ▻ der Datei mit Server-Stub und Server-Hauptprogarmm
  - ▻ der Datei / den Dateien mit den RPC-Funktionen des Servers
  - ▻ und gegebenenfalls der Datei mit den Daten-Konvertierungsroutinen

  und zusätzlichem Linken der **RPC-Laufzeitbibliothek**

- **Erzeugung des Client-Programms**

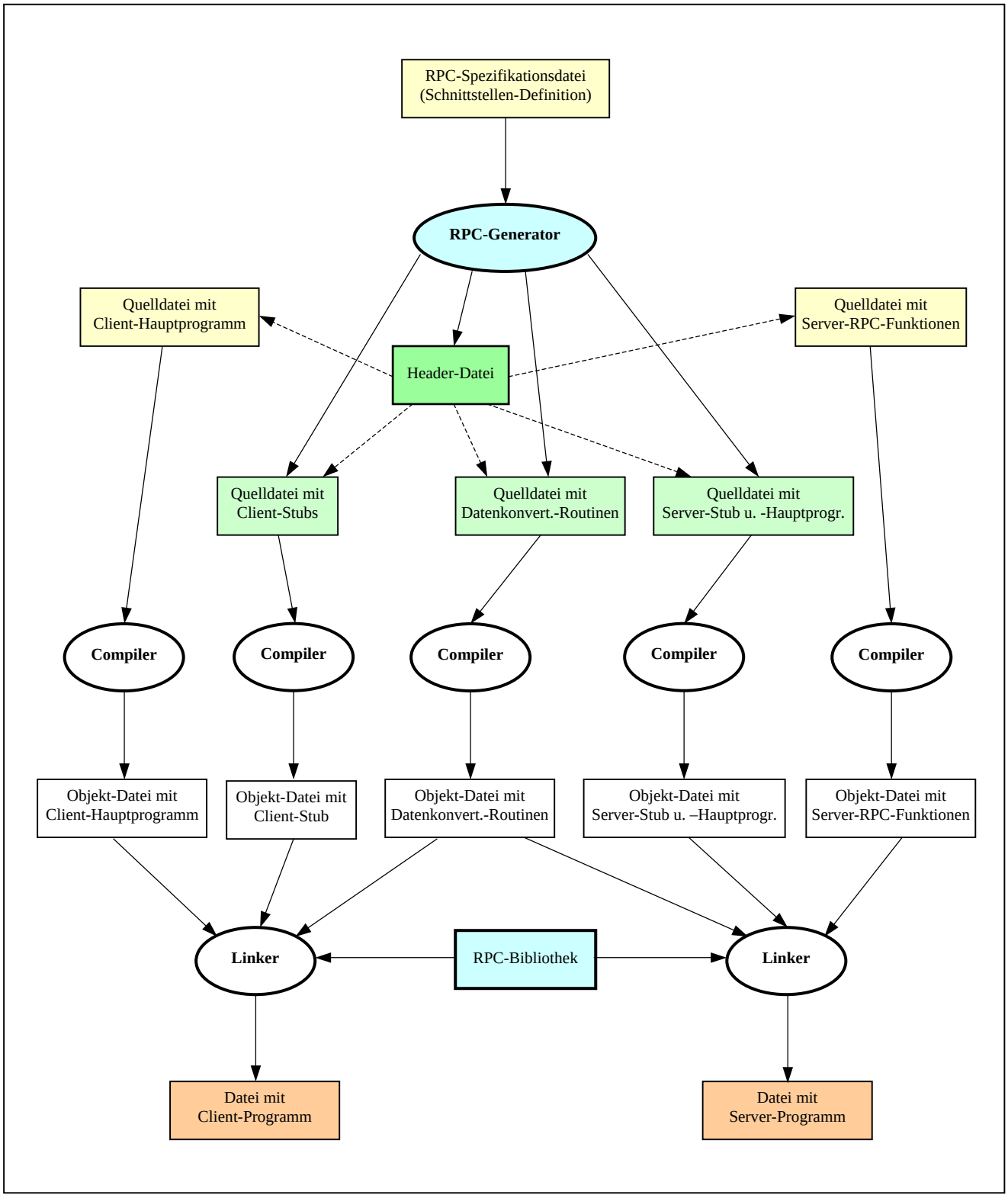
  Das **Client-Programm** entsteht durch Compilieren und Linken

  - ▻ der Datei mit dem Client-Hauptprogramm
  - ▻ der Datei mit den Client-Stubs
  - ▻ und gegebenenfalls der Datei mit den Daten-Konvertierungsroutinen

  und zusätzlichem Linken der **RPC-Laufzeitbibliothek**

Remote Procedure Call (RPC) – Erzeugung einer RPC-Anwendung (2)

- **Erzeugung einer RPC-Anwendung**



Remote Procedure Call (RPC) – Beispiel für Sun-RPC (1)

- **Aufgabenstellung**

  ◇ Nachbildung eines vereinfachten lotto-ähnlichen Spiels :

  ◇ Ein Server soll eine mittels RPC aufrufbare Funktion `play()` zur Verfügung stellen.
  Der Funktion werden drei `int`-Werte, die im Bereich `1 .. 36` liegen müssen, als Eingabeparameter übergeben.
  Die Funktion erzeugt drei im gleichen Werte-Bereich liegende Pseudozufallszahlen und vergleicht sie auf Übereinstimmung mit den übergebenen Parameterwerten.
  Die Anzahl der übereinstimmenden – d.h. richtig geratenen – Werte gibt sie als Funktionswert zurück.

  ◇ Ein Client-Programm fordert in einer Schleife zur Eingabe von jeweils drei `int`-Zahlen auf, liest diese von der Standardeingabe ein und ruft mittels RPC die Funktion `play()` auf, der die eingelesenen Zahlen als Parameter übergeben werden
  Den von der Funktion zurückgegebenen Wert gibt das Programm als Anzahl der Treffer an die Standardausgabe aus.

- **Schnittstellenbeschreibung (Datei `play.x`)**

```
/* ------------------------------------------------------------ */
/*  play.x  --  Schnittstellenbeschreibung für ein RPC-Beispiel */
/*------------------------------------------------------------- */

struct params
{
  int z1;
  int z2;
  int z3;
};

program PLAY_PROG
{
  version PLAY_VERS
  {
    int PLAY(params) =1;
  } = 1;
} = 0x20000001;
```

  ◇ **Anmerkungen :**

  ▻ Sun-RPC ermöglicht nur die Übergabe eines Parameters
  → mehrere Parameter müssen zu einem geeigneten `struct`-Typ zusammengefaßt werden

  ▻ Bei Sun-RPC wird jede über RPC aufrufbare Funktion eindeutig über ein Tripel <Funktions-Nummer, Programm-Nummer, Versions-Nummer> identifiziert.
  Programm- und Versions-Nummer referieren das Server-Programm, das die Funktion zur Verfügung stellt.
  Ein Programm kann mehrere Funktionen zur Verfügung stellen (→ Funktions-Nummer).
  Die Programm-Nummern `0x0 ... 0x1fffffff` werden von Sun verwaltet. Sie sind für Programme, die von allgemeinem Interesse sind und weltweit über eine eindeutige Nummer verfügbar sein sollen, vorgesehen.
  Die Programm-Nummern `0x20000000 .. 0x3fffffff` können von Benutzern für lokale Anwendungen verwendet werden.

- **Aufruf des RPC-Generators**

  ▻ **`rpcgen play.x`**
  → Erzeugung von : `play.h, play_svc.c, play_clnt.c, play_xdr.c`

Remote Procedure Call (RPC) – Beispiel für Sun-RPC (2)

- **Erzeugte Header-Datei `.play.h`**

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _PLAY_H_RPCGEN
#define _PLAY_H_RPCGEN

#include <rpc/rpc.h>


#ifdef __cplusplus
extern "C" {
#endif


struct params {
  int z1;
  int z2;
  int z3;
};
typedef struct params params;

#define PLAY_PROG 0x20000001
#define PLAY_VERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define PLAY 1
extern  int * play_1(params *, CLIENT *);
extern  int * play_1_svc(params *, struct svc_req *);
extern int play_prog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define PLAY 1
extern  int * play_1();
extern  int * play_1_svc();
extern int play_prog_1_freeresult ();
#endif /* K&R C */

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_params (XDR *, params*);

#else /* K&R C */
extern bool_t xdr_params ();

#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_PLAY_H_RPCGEN */
```

Remote Procedure Call (RPC) – Beispiel für Sun-RPC (3)

- **Erzeugte Quelldatei mit Server-Stub und Server-Hauptprogramm `(play_svc.c)`**

  1. Teil : Server-Stub-Funktion `play_prog_1()`

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "play.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void
play_prog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
  union {
    params play_1_arg;
  } argument;
  char *result;
  xdrproc_t _xdr_argument, _xdr_result;
  char *(*local)(char *, struct svc_req *);

  switch (rqstp->rq_proc) {
  case NULLPROC:
    (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
    return;

  case PLAY:
    _xdr_argument = (xdrproc_t) xdr_params;
    _xdr_result = (xdrproc_t) xdr_int;
    local = (char *(*)(char *, struct svc_req *)) play_1_svc;
    break;

  default:
    svcerr_noproc (transp);
    return;
  }
  memset ((char *)&argument, 0, sizeof (argument));
  if (!svc_getargs (transp, _xdr_argument, (caddr_t) &argument)) {
    svcerr_decode (transp);
    return;
  }
  result = (*local)((char *)&argument, rqstp);
  if (result != NULL && !svc_sendreply(transp, _xdr_result, result)) {
    svcerr_systemerr (transp);
  }
  if (!svc_freeargs (transp, _xdr_argument, (caddr_t) &argument)) {
    fprintf (stderr, "unable to free arguments");
    exit (1);
  }
  return;
}
```

Remote Procedure Call (RPC) – Beispiel für Sun-RPC (4)

- **Erzeugte Quelldatei mit Server-Stub und Server-Hauptprogramm `(play_svc.c)`**

  2. Teil : Funktion `main()` (Server-Hauptprogramm)

```
int
main (int argc, char **argv)
{
  register SVCXPRT *transp;

  pmap_unset (PLAY_PROG, PLAY_VERS);

  transp = svcudp_create(RPC_ANYSOCK);
  if (transp == NULL) {
    fprintf (stderr, "cannot create udp service.");
    exit(1);
  }
  if (!svc_register(transp, PLAY_PROG, PLAY_VERS, play_prog_1, IPPROTO_UDP)) {
    fprintf (stderr, "unable to register (PLAY_PROG, PLAY_VERS, udp).");
    exit(1);
  }

  transp = svctcp_create(RPC_ANYSOCK, 0, 0);
  if (transp == NULL) {
    fprintf (stderr, "cannot create tcp service.");
    exit(1);
  }
  if (!svc_register(transp, PLAY_PROG, PLAY_VERS, play_prog_1, IPPROTO_TCP)) {
    fprintf (stderr, "unable to register (PLAY_PROG, PLAY_VERS, tcp).");
    exit(1);
  }

  svc_run ();
  fprintf (stderr, "svc_run returned");
  exit (1);
  /* NOTREACHED */
}
```

◇ **Anmerkungen :**

▻ Mit der RPC-Bibliotheksfunktion **`svc_register()`** wird das Server-Programm bei dem lokalen – als Namensdienst fungierenden – **Portmapper** registriert
Der Portmapper ist ein Dämon, der mit dem Kommando **`portmap`** gestartet wird.

Mit dem Kommando **`rpcinfo –p host`** können alle auf dem Rechner `host` registrierten RPC-Server-Programme ermittelt werden.

Mit dem zu einem Broadcast führenden Komando **`rpcinfo –b prognum versnum`** kann die Rechner-IP-Adresse und die Portnummer unter der ein Server-Programm erreichbar ist, ermittelt werden

▻ Die RPC-Bibliotheksfunktion **`svc_run()`** wartet in einer Endlosschleife auf ankommende RPC-Aufrufe und ruft – nach Konvertierung der Parameter mittels der `xdr-`Funktionen – die Server-Stub-Funktion auf.

Remote Procedure Call (RPC) – Beispiel für Sun-RPC (5)

- **Erzeugte Quelldatei mit Client-Stub `(play_clnt.c)`**

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <memory.h> /* for memset */
#include "play.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

int *
play_1(params *argp, CLIENT *clnt)
{
  static int clnt_res;

  memset((char *)&clnt_res, 0, sizeof(clnt_res));
  if (clnt_call (clnt, PLAY,
     (xdrproc_t) xdr_params, (caddr_t) argp,
     (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
     TIMEOUT) != RPC_SUCCESS) {
    return (NULL);
  }
  return (&clnt_res);
}
```

- **Erzeugte Quelldatei mit Datenkonvertierungsroutinen `(play_xdr.c)`**

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "play.h"

bool_t
xdr_params (XDR *xdrs, params *objp)
{
  register int32_t *buf;

  if (!xdr_int (xdrs, &objp->z1))
    return FALSE;
  if (!xdr_int (xdrs, &objp->z2))
    return FALSE;
  if (!xdr_int (xdrs, &objp->z3))
    return FALSE;
  return TRUE;
}
```

Remote Procedure Call (RPC) – Beispiel für Sun-RPC (6)

- **Quelldatei `play_proc.c` mit Server-Funktion `play_1_svc()`**

```
/* ------------------------------------------------------------------- */
/* play_proc.c  ---  RPC-Beispiel : Definition entfernte Server-Prozedur */
/* ------------------------------------------------------------------- */

#include "play.h"

#define ANZ 3

extern int rand36();

int* play_1_svc(params* par, struct svc_req* dummy)
{
  static int hit;
  int guess[ANZ];
  int i, j;
  int num;

  guess[0]=par->z1;
  guess[1]=par->z2;
  guess[2]=par->z3;
  hit=0;
  for (i=0; i<ANZ; i++)
  {
    num=rand36();
    j=0;
    while(j<ANZ)
      if (num==guess[j])
      {
        hit++;
        guess[j]=-1;
        j=ANZ;
      }
      else
        j++;
  }
  return &hit;
}
```

- **Erzeugung des Server-Programms `playserver`**

  ▻ **`cc play_svc.c play_proc.c play_xdr.c –o playserver`**

Remote Procedure Call (RPC) – Beispiel für Sun-RPC (7)

- **Quelldatei mit Client-Hauptprogramm `(rplay.c)`**

```
/* ------------------------------------------------------------ */
/* rplay.c   ---  Client-Programm zum RPC-Aufruf von play()     */
/* ------------------------------------------------------------ */

#include <stdio.h>
#include <rpc/rpc.h>

#include "play.h"

int main(int argc, char** argv)
{
  int ret=0;
  char* server;
  params myGuess;
  CLIENT* cl;

  if (argc<2)
  { printf("\nUsage : rplay server\n");
    ret=1;
  }
  else
  {
    server=argv[1];
    if ((cl=clnt_create(server, PLAY_PROG, PLAY_VERS, "tcp"))==NULL)
    {  printf("\nRPC-Programm nicht gefunden\n");
       ret=2;
    }
    else
    {
      while (printf("\nEin neues Spiel : "),
             scanf("%d%d%d", &myGuess.z1, &myGuess.z2, &myGuess.z3)!=EOF)
        printf("\nMein Spiel : %d Richtige\n", *play_1(&myGuess, cl));
    }
  }
  putchar('\n');
  return ret;
}
```

◇ **Anmerkungen** :

- ▻ Die RPC-Bibliotheksfunktion **`clnt_create()`** ermittelt einen Handle (Typ `CLIENT*`) über den das Server-Programm angesprochen werden kann.
- ▻ Dieser Handle ist beim Aufruf des Client-Stubs (hier : `play_1()`) diesem zusätzlich zum eigentlichen Funktions-Parameter zu übergeben.

- **Erzeugung des Client-Programms `rplay`**

▻ **`cc rplay.c play_clnt.c play_xdr.c –o rplay`**

Verteilte objektorientierte Architekturen (1)

- **Objektorientiertes Programm-Modell**

  ◇ **Objekte** sind **Programm-Einheiten**, die **Daten** und die sie bearbeitenden **Funktionen** zusammenfassen :
  Die Datenkomponenten beschreiben den **Zustand** eines Objektes, die Funktionskomponenten – üblicherweise **Methoden** genannt – legen seine **Fähigkeiten**, sein Verhalten fest.

  ◇ Objekte **kapseln** ihren **inneren Aufbau** (ihre Implementierung, insbesondere ihre Datenkomponenten).
  → **Kapselung** (*encapsulation*)
  Ihre Interna lassen sich daher nicht direkt manipulieren.
  Vielmehr lassen sich Objekte nur über öffentlich zugänglich gemachte Methoden ansprechen : → **Aufrufschnittstelle**

  ◇ Über diese Aufrufschnittstelle stellen Objekte ihre **Dienstleistungen** zur Verfügung.
  Die **Anforderung** einer Dienstleistung erfolgt durch das **Senden einer Botschaft** an das Objekt, dies wird durch den Aufruf einer Methode realisiert.
  Das Objekt reagiert auf die Botschaft mit der Ausführung der entsprechenden Methode, d.h. mit der Wahrnehmung der jeweiligen Dienstleistung.

  ◇ Objekte können von anderen Objekten mittels Ableitung Datenkomponenten und Methoden **erben**.
  (→ **Vererbung**, *inheritance*).
  Abgeleitete Objekte können die Eigenschaften ihrer Vorfahren einerseits erweitern, andererseits im Sinne einer Spezialisierung modifizieren.

  ◇ **Polymorphismus** (*polymorphism*) ermöglicht es, daß unterschiedliche Objekte, die die gleichen Vorfahren haben, auf die gleiche Botschaft unterschiedlich reagieren, d.h. Memberfunktionen mit derselben Schnittstelle unterschiedlich implementieren.

  ◇ Analog den benutzerdefinierten Datentypen in der prozeduralen Programmierung beschreibt man den Aufbau gleichartiger Objekte (gleiche Datenkomponenten, gleiche Memberfunktionen) durch Objekttypen, die man **Klassen** nennt.
  Klassen dienen als "Schablone" zur Erzeugung individueller Objekte (des entsprechenden) Typs zur Laufzeit.

  ◇ Ein **objektorientiertes Programm** besteht aus einer **Ansammlung** von **Objekten**, die miteinander über Botschaften **kommunizieren** und somit voneinander Dienstleistungen anfordern.

- **Objektorientiertes Modell für verteilte Client-Server-Anwendungen**

  ◇ Jede Kommunikationsbeziehung zwischen Objekten kann als **Client-Server-Beziehung** aufgefaßt werden.

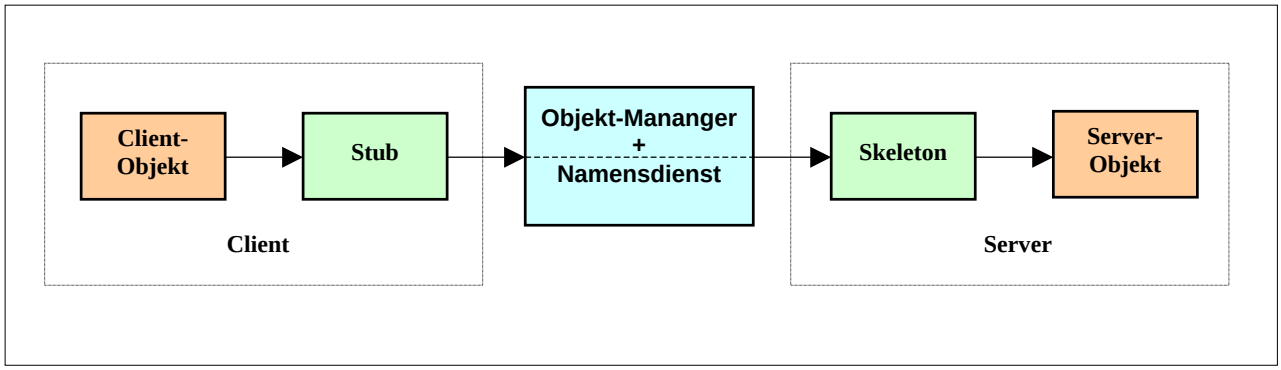
  ◇ Das objektorientierte Modell läßt sich sinnvoll auf **verteilte Client-Server-Anwendungen** übertragen :

  ▻ Ein **Server-Objekt** stellt **Dienstleistungen** zur Verfügung

  ▻ die von **Client-Objekten**, die sich in einem anderen Prozeßraum – i.a. auf einem anderen Rechner – befinden, **angefordert** werden können.

  ◇ Das **Senden von Botschaften** an **entfernte Objekte** führt zu einer **Übertragung** des Prinzips des **entfernten Prozeduraufrufs** (RPC) auf die **Objektsemantik**.

  ◇ Entsprechend dem RPC wird eine Botschaft vom Client-Objekt nicht direkt zum Server-Objekt gesandt, sondern über **Stellvertreter-Objekte** (Proxy-Objekte) des Server-Objekts geleitet :
  dem **Stub** auf der Client-Seite und dem **Skeleton** auf der Server-Seite.
  Die Vermittlung zwischen Client- und Serverseite übernimmt ein **Objekt-Manager**, der einen **Namensdienst** beinhaltet oder mit einem Namensdienst zusammenarbeitet.



Verteilte objektorientierte Architekturen (2)

- **Forderungen an verteilte objektorientierte Systeme**

  - **lokationsunabhängige Programmierung**
    Die Verteilung und Platzierung der Objekte erfolgt erst nach der Programmierung
    → keine Berücksichtigung von Details des späteren Zielsystems bei der Objektentwicklung
    Die Objektkapselung unterstützt ein nebenwirkungsfreies Programmieren, so daß Abhängigkeiten zwischen den Verteilungseinheiten (Objekte) minimal bleiben

  - **lokationstransparenter Methodenaufruf**
    Lokale und entfernte Methodenaufrufe sind syntaktisch gleich.
    Der Server-Stub auf Clientseite besitzt dieselbe Schnittstelle wie das Original-Server-Objekt

  - **Objektmigration**
    Erzeugung eines Objekts auf einem Rechner und Übertragung auf einen anderen Rechner unter Wahrung der Objekt-Identität (Datenkomponenten und gegebenenfalls auch Methoden-Code)
    Für Parameter-Objekte wird unterschieden :
    - ***call by visit*** : Das Parameter-Objekt kehrt nach der Methodenausführung zum früheren Aufenthaltsort zurück
    - ***call by move*** : Das Parameter-Objekt verbleibt am Zielort

    In einem allgemeineren Fall sollte Objektmigration auch für gerade aktive Objekte möglich sein

  - **Entfernte Objekterzeugung**
    Erzeugung von Objekten auf einem entfernten Rechner

  - **Lastoptimierung**
    Eine Lastoptimierung ist mit Objektmigration leichter erreichbar als mit Prozeßmigration

  - **Heterogenität**
    - Objekte können
      - **unabhängig von der Programmiersprache**, in der sie einmal formuliert worden sind
      - und **unabhängig von der Hardwareplattform**, auf der sie sich befinden,

      miteinander kommunizieren.
    - Objekte können zwischen unterschiedlichen Hardwareplattformen migrieren.
      Eine Migration auch von Code setzt die Verwendung von maschinenunabhängigen Bytecodes voraus.

  In realisierten verteilten objektorientiertenArchitekturen werden meist nicht alle der obigen Forderungen umgesetzt, z. Teil sind sie auch nur unvollständig implementiert.

- **Beispiele verteilter objektorientierter Architekturen**

  - **Remote Method Invocation (RMI) von Java**
    Keine Quellsprachen-Unabhängigkeit, dafür Hardwareplattform-Unabhängigkeit (Java-Bytecode !)

  - **Voyager** von ObjectSpace
    Eine auf Java basierende und zu Java vollkommen kompatible Laufzeitumgebung und Bibliothek für die Programmierung verteilter Anwendungen

  - **CORBA** (Common Object Request Broker Architecture)
    Entwickelt von der *Object Management Group* (OMG)
    Quellsprachen-Unabhängigkeit.
    Objekt-Schnittstellen werden in einer implementierungssprachen-unabhängigen Beschreibungssprache formuliert (*Interface Definition Language*, IDL)

  - **COM/DCOM** von Microsoft

  - **.NET** von Microsoft

Java-RMI - Eigenschaften

- **RMI (Remote Method Invocation)**
  - ◇ **Java-spezifische** Implementierung eines Systems für verteilte Objekte
    "Natürliche" Integration des verteilten Objekt-Modells in die Sprache Java unter Ausnutzung ihrer Konzepte und Beibehaltung ihrer Objekt-Semantik
  - ◇ Bestandteil des **Java-API** (ab JDK 1.1, modifiziert und erweitert in JDK 1.2)
  - ◇ Von den an verteilte objektorientierte Systeme gestellten **Forderungen** werden **erfüllt** :
    - ▻ **lokationsunabhängie Programmierung** (Java-Motto : "write once, run anywhere")
    - ▻ **lokationstransparenter Methodenaufruf**
    - ▻ **Heterogenität** bezüglich der **Hardware-Plattform**, nicht jedoch bezüglich der Programmiersprache

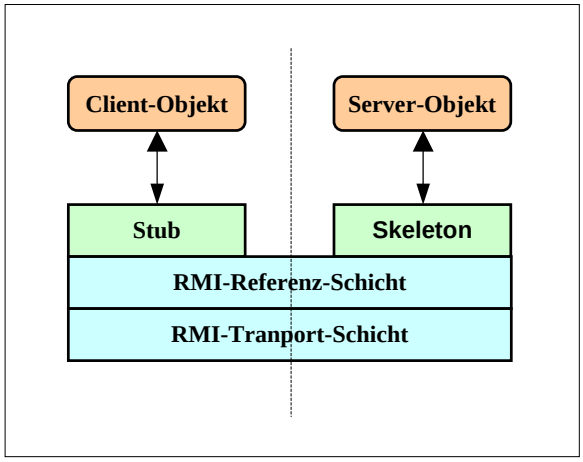
- **Realisierung der grundlegenden Fähigkeiten eines verteilten Objektsystems**
  - ◆ **Lokalisierung entfernter Objekte**
    Anwendungen stehen zwei Mechanismen zur Erlangung von Referenzen auf entfernte Objekte zur Verfügung :
    - ▻ Nachfragen bei einem **RMI-eigenen Namensdienst** (`rmiregistry`)
    - ▻ Übergabe als **Parameter** und/oder **Rückgabewerte** von entfernten Funktionsaufrufen
  - ◆ **Kommunikation mit entfernten Objekten**
    Die eigentliche Kommunikation wird vom RMI-Laufzeitsystem durchgeführt, Einzelheiten bleiben der eigentlichen Anwendung (und ihrem Programmierer) verborgen.
    → Ein entfernter Methodenaufruf sieht wie ein normaler lokaler Methodenaufruf aus
    (**transparenter entfernter Methodenaufruf)**
  - ◆ **Übergabe von Objekten als Parameter und Rückgabewerte**
    Das RMI-Laufzeitsystem stellt die notwendigen Mechanismen zur Übertragung von Objekt-Daten und -Code zur Verfügung (→ **Kopieren** von Objekten)
    Diese beruhen auf
    - ▻ dem Java-Konzept der **Objekt-Serialisierung** (*object serialization*)
    - ▻ sowie der Java-Eigenschaft des **dynamischen Ladens von Code** (*dynamic code downloading*)
      → Erzeugung von Objekten, deren Klassen-Implementierung (Java-Byte-Code) lokal nicht verfügbar ist

- **Weitere wesentliche Eigenschaften von RMI**
  - ◇ Das dynamische Laden von Code ermöglicht die **dynamische Änderung des Verhaltens** von Server und/oder Client
  - ◇ Unterstützung von **Callbacks** vom Server zum Client
    Übergabe von Referenzen auf – im Client vorhandene – entfernte Objekte als Parameter im entfernten Methoden-aufruf
  - ◇ **Verteilte Garbage Collection**
    Einbeziehung von entfernten Objekten in den Garbage Collection Mechanismus von Java
    RMI stellt sicher, daß dann und nur dann entfernte Objekte (*remote objects*) vernichtet werden, wenn weder eine lokale noch eine entfernte Referenz auf das jeweilige Objekt existiert.
  - ◇ Einbettung in die **Sicherheitsmechanismen** der Java-Plattform
    Verwendung von Security Manager, Class Loader und Policy-Files zur Überprüfung der Zulässigkeit des Ladens von entferntem Code und des Ausführens von Code (Resourcenzugriff)
  - ◇ **Flexibilität** und **Erweiterbarkeit**, z.B.
    - ▻ Implementierung **unterschiedlicher Referenz-Semantiken** für entfernte (Server-) Objekte
      (z.Zt im Standard-API implementiert : `UnicastRemoteObject` und `Activatable`)
    - ▻ Einsatz unterschiedlicher **Socket-Typen** in der Transportschicht von RMI (Ersatz des standardmäßig verwendeten Socket-Typs (TCP-Protokoll) durch SSL-Sockets oder selbstdefinierte Socket-Typen)

Java-RMI - Funktionsweise (1)

- **RMI -Architektur**

  ◇ Das RMI-System besteht aus **drei Schichten** :
  - ▻ Stub-/Skeleton-Schicht
  - ▻ RMI-Referenz-Schicht
  - ▻ RMI-Transport-Schicht



◇ Die **Stub-/Skeleton-Schicht** bildet die Schnittstelle zwischen den Bestandteilen der verteilten Applikation (Client-Objekt und Server-Objekt) und dem restlichen RMI-System (RMI-Laufzeitsystem).

Der **Stub** ist ein Stellvertreter-Objekt des Server-Objekts auf der Client-Seite, der die gleiche Methoden-Aufruf-Schnittstelle wie das Server-Objekt anbietet.
Er nimmt die RMI-Aufrufe des Clients entgegen und leitet sie an die RMI-Referenzschicht weiter.
Umgekehrt erhält er von der RMI-Referenzschicht die Rückgabewerte eines RMI-Aufrufs und leitet diese an das aufrufende Client-Objekt weiter.

Das **Skeleton**-Objekt nimmt auf der Serverseite die RMI-Aufrufe von der RMI-Referenzschicht entgegen, ruft die entsprechenden im Server-Objekt implementierten Methoden auf, nimmt deren Rückgabewerte entgegen und leitet diese an die RMI-Referenzschicht weiter.

Die Funktionalität des Skeletons wird in neueren Java-Versionen durch einen allgemeinen Verteiler ersetzt.
Es muss daher kein eigenes Skeleton-Objekt erzeugt werden.
Der Stub muss dagegen an das jeweilige Server-Objekt angepaßt sein.
Seine Klasse wird aus der Server-Objekt-Klasse mittels eines **RMI-Compilers** `(rmic)` automatisch generiert.
(Der RMI-Compiler erzeugt auch die Klasse für ein Skeleton-Objekt, die aber nicht mehr benötigt wird)

◇ Die **RMI-Referenz-Schicht** ist für die **Lokalisierung** des jeweiligen Kommunikationspartners (Server bzw Client) zuständig. Sie beinhaltet den **RMI-Namensdienst** `(Registry)` und verwaltet die **Referenzen** auf entfernte Objekte.
Unter Berücksichtigung der jeweiligen Referenz-Semantik gibt sie die vom Stub erhaltenen RMI-Aufrufe einschließlich deren Parameter an die RMI-Transportschicht weiter.
Auf der Serverseite nimmt sie die RMI-Aufrufe und deren Parameter von der RMI-Transportschicht entgegen, aktiviert gegebenenfalls das Server-Objekt und leitet die Aufrufe an das Skeleton-Objekt weiter
Analog ist sie an der Übermittlung der Rückgabewerte beteiligt.

◇ In der **RMI-Transport-Schicht** werden die **Kommunikationsverbindungen** verwaltet und die eigentliche **Kommunikation** zwischen verschiedenen Adreßräumen (JVMs) abgewickelt.
Üblicherweise werden hierfür **Sockets** eingesetzt.

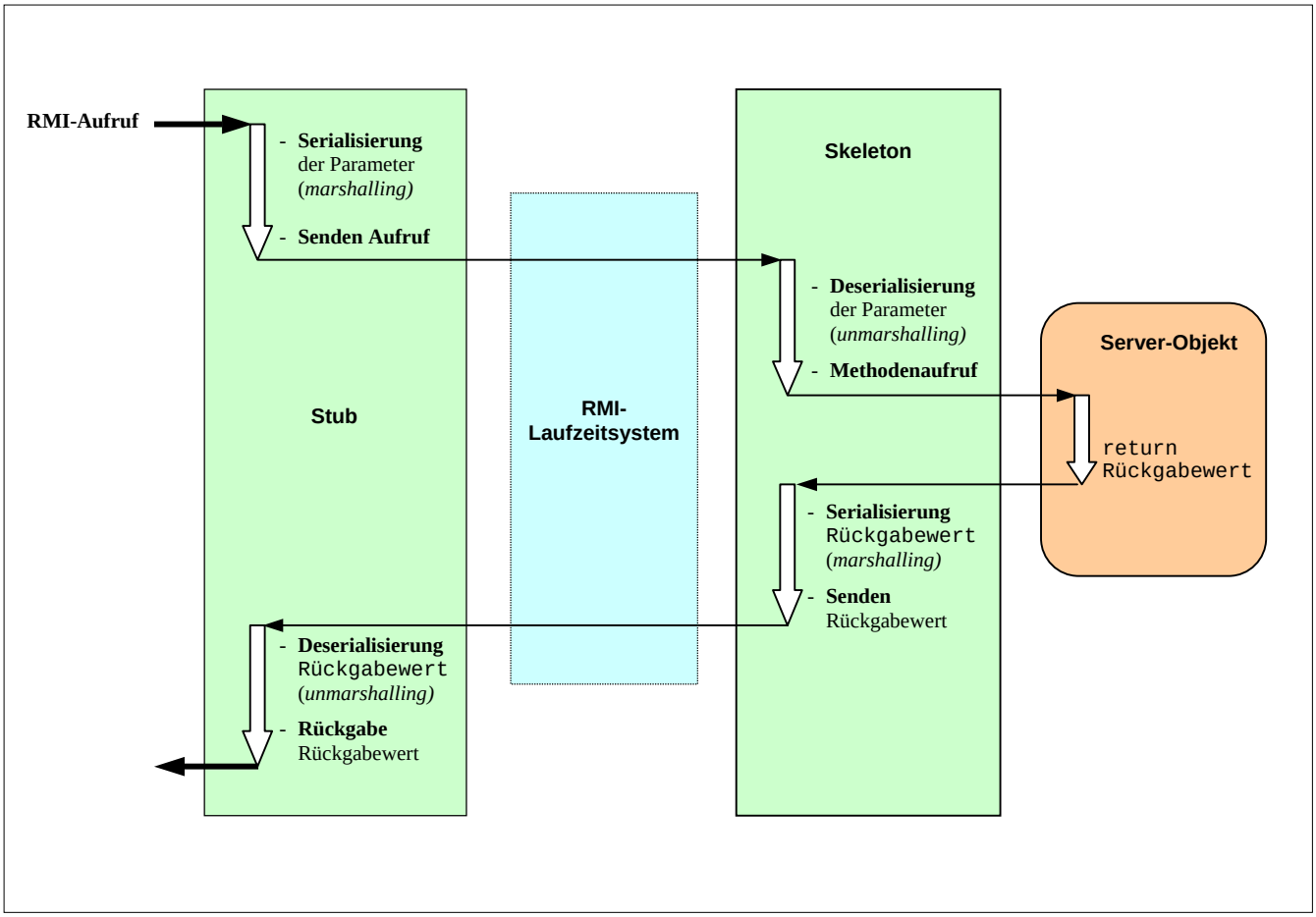
Java-RMI - Funktionsweise (2)

- **Zusammenspiel zwischen Stub und Skeleton**

  ◇ Der **Stub** wandelt die Methoden-Aufruf-**Parameter** mittels **Objekt-Serialisierung** in einen seriellen **Byte-Stream** um (*marshalling*).
  Dadurch können die Parameter zusammen mit den für den Aufruf der Serverfunktion benötigten Informationen (Identifikation des Ziel-Objekts und der aufzurufenden Methode) über das RMI-Laufzeitsystem zum Server-Skeleton übertragen werden.

  ◇ Das – durch einen allgemeinen Verteiler realisierte – **Skeleton deserialisiert** den empfangenen Byte-Stream und gewinnt dadurch die Parameter zurück (*unmarshalling*).
  Anschließend ruft es die durch den übertragenen Methoden-Identifikator ausgewählte Methode des Server-Objekts mit den deserialisierten Parametern auf.

  ◇ Die Übertragung des **Funktionsrückgabewertes** erfolgt analog vom Server (Serialisierung) zum Client (Deserialisierung)

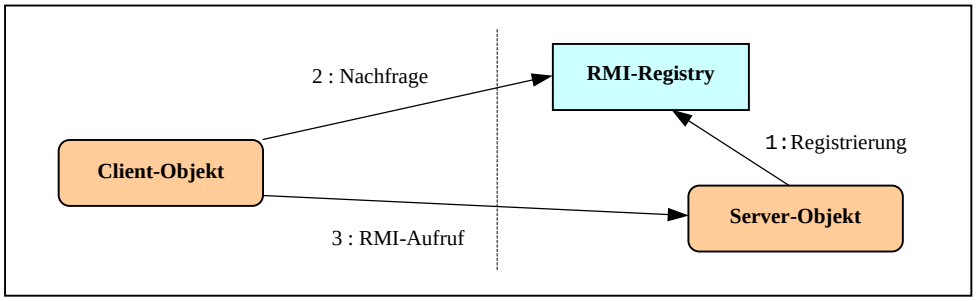


Java-RMI - Funktionsweise (3)

- **Übertragung von Parametern und Rückgabewerten**
  - ◇ Die Parameter und Rückgabewerte von RMI-Aufrufen werden mittels Objekt-Serialisierung als Byte-Streams übertragen.
    →Übertragbare Objekte müssen **serialisierba**r sein, d.h. das Interface `java.io.Serializable` implementieren.
  - ◇ **einfache Datentypen** : Übertragung der Werte (*Call by Value*)
  - ◇ **lokale Objekte** (*non-remote objects*) : Übertragung einer Kopie des Objekts (*Call by Copy*)
    Für Rückgabewerte wird beim Aufrufer ein neues Objekt erzeugt
  - ◇ **entfernte Objekte** (*remote objects*) : Übertragung einer Referenz auf das Objekt (*Call by Reference*)
    Die Referenz ist das zum Objekt gehörende **Stub-Objekt**. (das seinerseits per Kopie übertragen wird)
  - ◇ In dem im Byte-Stream für ein übertragenes Objekt enthaltenen Klassen-Deskriptor ist auch die **URL** über die der **Klassen-Code** (Java Byte Code) **zugänglich** ist, vermerkt.
    Dies ermöglicht es dem empfangenden System, den **Klassen-Code dynamisch zu laden**, falls er nicht lokal verfügbar ist. (***Dynamic Code Downloading***).
    Eine derartige Situation kann auftreten, wenn das übertragene Objekt eine Instanz eines Untertyps des deklarierten Parametertyps ist. Ein Untertyp in diesem Sinn ist entweder eine Klasse, die ein Interface, das als Parametertyp deklariert ist, implementiert oder eine von der Parameterklasse abgeleitete Klasse.

- **RMI-Registry**
  - ◇ Die RMI-Registry ist ein einfacher **Namensdienst**, der es Clients ermöglicht, entfernte Objekte über ihren Namen zu lokalisieren.
  - ◇ Eine RMI-Registry muß **lokal** auf dem System, das Objekte für RMI-Aufrufe (entfernte Objekte) bereitstellt (RMI-Server) existieren.
  - ◇ Üblicherweise wird die RMI-Registry in einer eigenen JVM als Hintergrundprozeß gestartet.
    Defaultmäßig "horcht" sie auf Port `1099`.
    Sie kann aber auch mit einer anderen Port-Nummer gestartet werden.

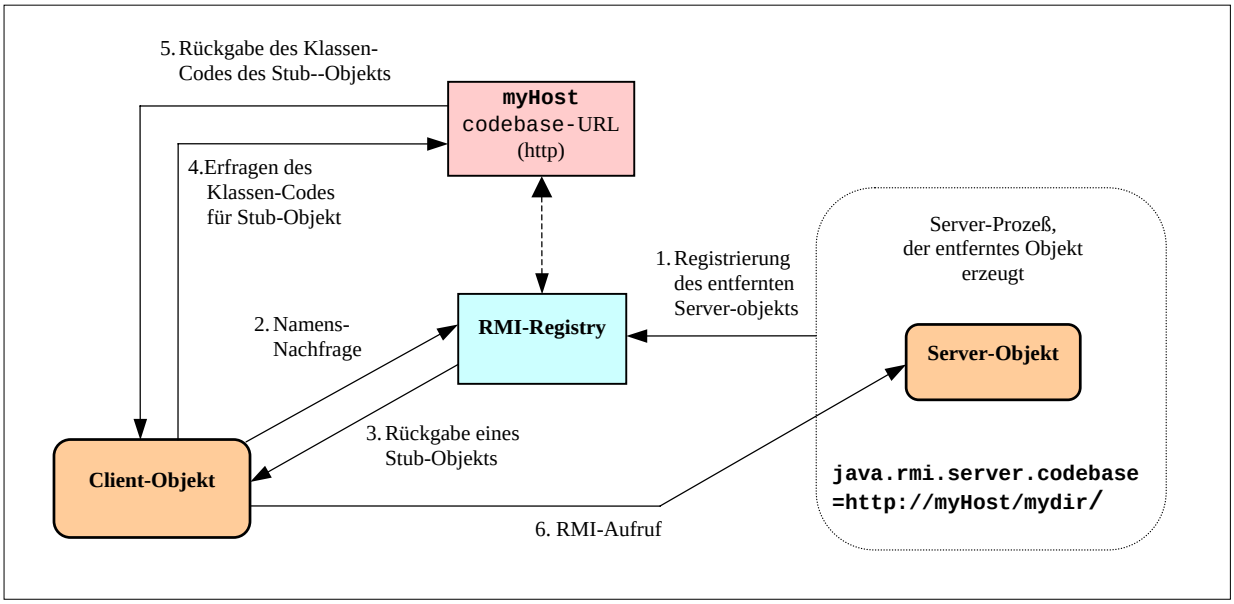
    Kommando : `start rmiregistry [port]` (Windows)
    `rmiregistry [port] &` (Linux)
  - ◇ Ein entferntes Objekt, das über die Registry zugänglich sein soll, muß vom Server-Prozeß der Registry bekannt gemacht (registriert, "**gebunden**") werden.
  - ◇ Ein Client, der einen RMI-Aufruf ausführen möchte, benötigt eine **Referenz** auf das entsprechende (entfernte) Server-Objekt. Diese erhält er durch eine Nachfrage ("**lookup**") bei der RMI-Registry des Server-Rechners.
    Falls für das – über einen Namen referierte – Objekt ein Eintrag vorhanden ist, liefert die Registry sein als Referenz dienendes **Stub-Objekt** zurück.
  - ◇ Typischerweise wird die RMI-Registry nur zur Lokalisierung des ersten vom Client referierten entfernten Objekts verwendet.
    Weitere gegebenenfalls benötigte entfernte Objekte können dann über dieses Objekt ermittelt werden (als Parameter bzw Rückgabewerte von RMI-Aufrufen)
  - ◇ Die Kommunikation mit der RMI-Registry erfolgt üblicherweise über das von der Java-Klasse `java.rmi.Naming` bereitgestellte Interface (statische Methoden)
  - ◇ Tatsächlich stellt die RMI-Registry bereits selbst ein **besonderes entferntes Objekt** dar, zu dem auch mittels RMI, das durch die `Naming`-Funktionen gekapselt wird, zugegriffen wird.



Java-RMI - Funktionsweise (4)

- **Dynamisches Laden von Code (*Dynamic Code Downloading*)**
  - ◇ Eine hervorstechende Eigenschaft von Java ist die Fähigkeit, dynamisch Code von jeder URL in eine laufende Java Virtual Machine (JVM) zu laden. I.a. wird die URL auf ein anderes physikalisches System verweisen.
    → Eine JVM kann Code ausführen, der nie auf ihrem eigenen System installiert worden ist.
  - ◇ Diese Fähigkeit wird auch vom RMI-System genutzt :
    - ▻ Zum **Laden** des **Codes** für den **Stub** des Server-Objekts durch den Client.
      → Die Stub-Klasse muß nicht lokal zur Verfügung stehen
    - ▻ Zum **Laden** des **Codes** für als **Parameter** in oder **Rückgabewerte** von RMI-Aufrufen übergebene Objekte (sowohl lokale Objekte als auch entfernte Objekte)
  - ◇ Das dynamische Laden von Code erfordert das Setzen der **Property** `java.rmi.server.codebase` in der JVM des das Objekt bzw die Klasse zur Verfügung stellenden Prozesses :
    - bei Stubs von Server-Objekten : in der JVM des RMI-Server-Prozesses, der ein entferntes Objekt registriert
    - bei als Parameter übergebenen Objekten : in der JVM des RMI-Client-Prozesses
    - bei Objekten, die von RMI-Aufrufen zurückgegeben werden : in der JVM des Server-Prozesses

    Als Wert der `codebase`-Property ist die URL, von der der Klassen-Code geladen werden kann, anzugeben. Diese URL kann sich auf ein drittes System beziehen.
  - ◇ Wenn der Server-Prozeß ein entferntes Objekt unter einem Namen bei der RMI-Registry registriert, wird der in der Server-JVM gesetzte `codebase`-Wert (URL !) zusammen mit der Objekt-Referenz (Stub-Objekt) gespeichert.
    Voraussetzung : Der Code der Stub-Klasse darf nicht über die `CLASSPATH`-Environment-Variable der JVM der RMI-Registry erreichbar sein.
  - ◇ Der Client-Prozeß erhält auf seine Nachfrage bei der RMI-Registry das Stub-Objekt als Referenz auf das Server-Objekt Falls er den Stub-Klassen-Code nicht über seine `CLASSPATH`-Environment-Variable lokal findet, versucht er den Code von der mit dem Stub-Objekt verbundenen URL (`codebase`-Wert) zu laden.
    Voraussetzung : Der RMI-Client-Prozeß hat einen Security Manager installiert, der das Laden des Stubs erlaubt (z.B. `RMISecurityManager`). In Java 2 (ab JDK 1.2) erfordert dies zusätzlich ein entsprechend konfiguriertes Security Policy File
  - ◇ Gegebenenfalls werden mit dem Stub des Server-Objekts auch alle weiteren von ihm benötigten Klassen-Codes von der `codebase`-URL geladen.
  - ◇ Analog erfolgt das Laden von Klassen-Codes für Objekte, die dem Server-Objekt vom Client-Objekt als Parameter in RMI-Aufrufen übergeben werden.
    Voraussetzungen : Der `codebase`-Wert im Client-Prozeß ist auf die richtige URL gesetzt und das Server-Objekt kann den Code nicht lokal über seine `CLASSPATH`-Environment-Variable finden. Außerdem muß im Server ein Security Manager installiert sein.



Java-RMI - Interfaces und Klassen (1)

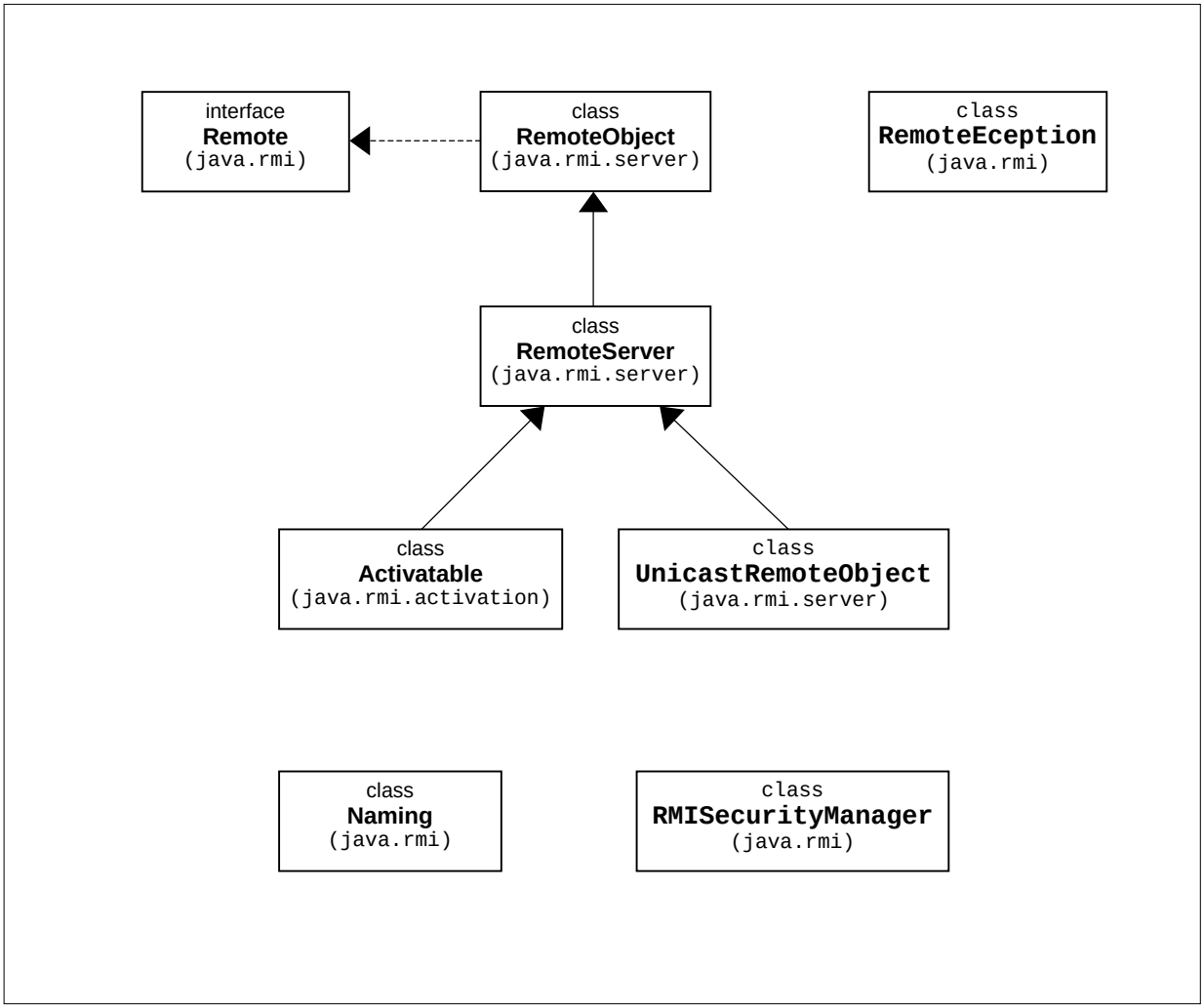
- **RMI-Packages**

  Das RMI-API von Java wird durch die folgenden Packages zur Verfügung gestellt :

  - **`java.rmi`**
    grundlegendes Package
  - **`java.rmi.activation`**
    Unterstützung für die RMI Object Activation
  - **`java.rmi.dgc`**
    Klassen und Interfaces für die RMI Distributed Garbage Collection
  - **`java.rmi.registry`**
    Klassen und Interfaces für die RMI-Registry
  - **`java.rmi.server`**
    Klassen und Interfaces zur Unterstützung der Server-Seite von RMI

- **Überblick über die wichtigsten Interfaces und Klassen des RMI-API**

  Für Anwenderprogramme sind vor allem die folgenden Interfaces und Klassen von Bedeutung :



Java-RMI - Interfaces und Klassen (2)

- **Die wichtigsten Interfaces und Klassen des Java-RMI-API - Kurzbeschreibung**

  ◇ **`interface Remote`** (Package : `java.rmi`)

  "Basis"-Interface für alle Interfaces, deren Methoden mittels RMI aufrufbar sein sollen.
  Alle Klassen, deren Objekte mittels RMI ansprechbar sein sollen, müssen direkt oder indirekt – über abgeleitete Interfaces – dieses Interface implementieren.
  Mittels RMI können nur die Methoden, die in einem von Remote abgeleiteten Interface (→ *remote interface*) spezifiziert sind, aufgerufen werden.

  ◇ **`class RemoteObject`** (Package : `java.rmi.Server`)

  Abstrakte Klasse, die einige Methoden der Klasse `java.lang.Object` für entfernt nutzbare Objekte (*remote objects*) überschreibt und damit die entsprechenden Fähigkeiten für derartige Objekte implementiert.
  (spezielle Implementierungen der Methoden `hashCode(), equals()` und `toString()`)

  ◇ **`class RemoteServer`** (Package : `java.rmi.Server`)

  Abstrakte Superklasse der Klassen für entfernt nutzbare Server-Objekte.
  Sie stellt ein gemeinsames Framework zur Unterstützung unterschiedlichster Semantiken für entfernte Referenzen zur Verfügung, die durch die von ihr abgeleiteten Klassen implementiert werden

  Die zur Erzeugung und Exportierung von Objekten an das RMI-Laufzeitsystem benötigten Methoden (Konstruktoren und gegebenenfalls weitere Methoden) werden durch die von dieser Klasse abgeleiteten Klassen bereitgestellt.

  ◇ **`class UnicastRemoteObject`** (Package : `java.rmi.Server`)

  Von der Klasse `RemoteServer` abgeleitete Klasse für einfache – nicht replizierbare – entfernte Objekte, deren Referenzen nur gültig sind, solange die Objekte aktiv sind.
  Sie stellt die Unterstützung für Punkt-zu-Punkt-Verbindungen zu aktiven Objekten (bezüglich Aufrufe, Parameter, Rückgabewerte) unter Verwendung von TCP-Streams bereit
  Eigene Klassen, die diese Referenz-Semantik bereitstellen sollen, müssen – unter Implementierung eines geeigneten *remote interfaces* - von dieser Klasse abgeleitet werden.

  ◇ **`class Activatable`** (Package : `java.rmi.Activation`)

  Von der Klasse `RemoteServer` abgeleitete Klasse für entfernte Objekte, deren Referenzen auch gültig sind, wenn die Objekte inaktiv und persistent ausgelagert sind Diese Objekte können bei Bedarf wieder aktiviert werden.
  Eigene Klassen, die diese Referenz-Semantik bereitstellen sollen, müssen – unter Implementierung eines geeigneten *remote interfaces* - von dieser Klasse abgeleitet werden.

  ◇ **`class RemoteException`** (Package : `java.rmi`)

  Superklasse der kommunikationsorientierten Exceptions, die bei RMI auftreten können.
  Alle Methoden eines *remote interfaces* müssen diese Exception weiterreichen können, d.h. in ihrem Funktionskopf muß eine entsprechende Exception-Spezifikation (`throws`-Klausel) enthalten sein.

  ◇ **`class Naming`** (Package : `java.rmi`)

  Nicht ableitbare Klasse (`final`), die statische Methoden zum Zugriff zur RMI-Registry bereitstellt

  ◇ **`class RMISecurityManager`** (Package : `java.rmi`)

  Beispiel eines von RMI-Anwendungen für das dynamische Laden von Code benötigten Security Managers.
  Der RMI Class Loader kann ohne installierten Security Manager keinen Code von URLs, die auf einen entfernten Rechner verweisen, laden.

Erzeugung einer RMI-Anwendung (1)

- **Auszuführende Schritte**
  - ► Definition eines **Interfaces** für die RMI-Aufrufe (*remote interface*)
  - ► Erstellung eines **Servers** (Implementierung des Interfaces und Server-Start-Funktionalität)
  - ► Entwicklung eines **Clients**, der unter Nutzung dieses Interfaces RMI-Aufrufe ausführt
  - ► Erzeugung der **Stub**- und **Skeleton**-Klassen
  - ► Starten der **RMI-Registry** auf dem Server-Rechner
  - ► Start des Servers und des Clients

- **Definition eines Interfaces für die RMI-Aufrufe**

  ◇ Alle Methoden des Server-Objekts, die über RMI aufrufbar sein sollen, sind in einem **Interface** zu deklarieren

  ◇ **Eigenschaften** dieses RMI-Interfaces :
  - ► Das Interface muß **`public`** deklariert werden
  - ► Das Interface muß von dem Interface **`java.rmi.Remote`** abgeleitet sein (→ *remote interface*).
  - ► Jede Methode des Interfaces muß die Exception **`java.rmi.RemoteException`** als werfbar deklarieren (Exception-Deklaration : `throws java.rmi.RemoteException`)

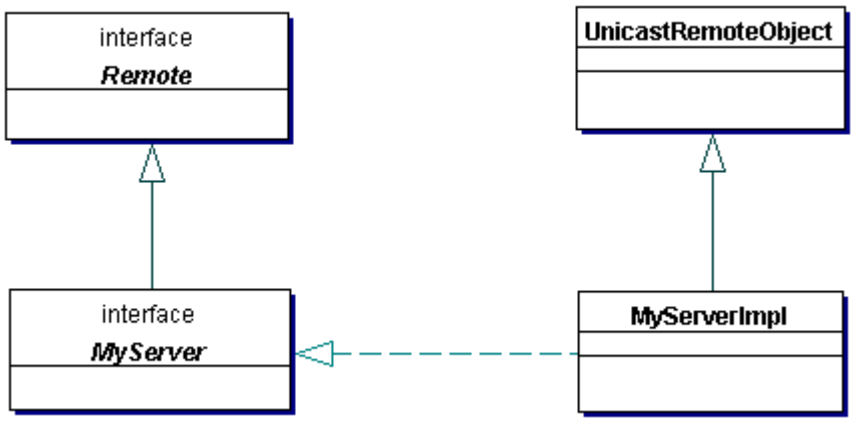
  ◇ Die Klasse der RMI-Server-Objekte muß dieses Interface implementieren (RMI-Server-Klasse).
  Nur die Methoden der Server-Objekte, die in diesem Interface enthalten sind, sind auch entfernt aufrufbar.
  Alle weiteren eventuell vorhandenen Methoden sind nur lokal ansprechbar.

  ◇ Der Client muß das Stub-Objekt, das er als lokale Referenz auf das entfernte Server-Objekt erhält, immer als eine Instanz dieses Interfaces referieren
  Der hierfür notwendige Type-Cast ist zulässig, da auch die Stub-Klasse dieses Interface implementiert.

  ◇ **Übersetzung** der Java-Quell-Datei (z.B. `MyServer.java`)
  → Erzeugung einer Java-Bytecode-Datei (`MyServer.class`)

```
javac MyServer.java
```

- **Klassendiagramm für einen einfachen RMI-Server**



Erzeugung einer RMI-Anwendung (2)

- **Erstellung eines Servers**

  - Zur Realisierung eines RMI-Servers werden benötigt :
    - Eine das RMI-Interface implementierende Klasse (**RMI-Server-Klasse**)
    - Die **Server-Start-Funktionalität**.
      Häufig wird die Server-Start-Funktionalität durch die `main()`-Methode der RMI-Server-Klasse realisiert.
      Diese `main()`-Methode kann aber auch in einer anderen – speziell hierfür vorgesehenen - Klasse enthalten sein.

  - Die **RMI-Server-Klasse** muß
    - wenigstens ein **RMI-Interface** *(remote interface*) **implementieren**
    - von einer Unterklasse der Klasse **`java.rmi.server.RemoteServer`** abgeleitet sein
    - einen **Konstruktor** definieren, der die Exception **`java.rmi.RemoteException`** werfen kann
    - alle im implementierenden RMI-Interface enthaltenen **Methoden** definieren

  - Der **Konstruktor** der RMI-Server-Klasse muß den Konstruktor der Basisklasse aufrufen (per default).
    Dieser **exportiert** ein neu erzeugtes Objekt an das RMI-Laufzeitsystem. Dadurch ist das Objekt in der Lage, ankommende RMI-Aufrufe (über den Stub) zu empfangen.

  - Die **Server-Start-Funktionalität** beinhaltet :
    - Erzeugung und Installation eines **Security Managers**
    - **Erzeugung** einer oder mehrerer **RMI-Server-Objekte** (Instanzen der RMI-Server-Klasse)
    - **Registrierung** wenigstens eines dieser Objekte bei der RMI-Registry

  - Erzeugung und Installation eines **Security Managers**
    In jeder JVM, in der Code von anderen Quellen als den durch die `CLASSPATH`-Environmentvariable festgelegten geladen werden soll, muß ein Security Manager installiert sein.
    Falls noch kein Security Manager installiert ist, muß der Server einen installieren – entweder eine Instanz der Klasse `RMISecurityManager` oder einer selbstdefinierten Security Manager Klasse

```
if (System.getSecurityManager()==null)
{
  System.setSecurityManager(new RMISecurityManager());
}
```

  - **Registrierung** eines RMI-Server-Objektes bei der lokalen RMI-Registry
    Mittels der statischen Methode **`rebind()`** der Klasse **`Naming`** :

```
public static void rebind(String name, Remote obj) throws RemoteException,
                                                          MalformedURLException
```

    Parameter : `name` - URL-formatierter Stríng, der das zu registrierende Objekt bezeichnet
    - Allgemeine Form : `"//host:port-nr/ObjectName".`
    - Wenn die RMI-Registry auf dem Default-Port `1099` läuft, können die Angabe des Hosts (`host`) und der Port-Nr. (`port-nr`) weggelassen werden, es reicht die Angabe von `"ObjectName"`.
    - `"ObjectName"` ist ein beliebiger Name unter dem das Objekt registriert werden soll.

    `obj` Name des Objekts unter dem es angelegt worden ist und im Programm referiert wird.

    Beispiel : `Naming.rebind("MyServer", serv);`

    Die von der Methode werfbaren Exceptions müsssen gefangen werden.

  - **Übersetzung** der Java-Quell-Datei (`MyServerImpl.java`)
    → Erzeugung einer Java-Bytecode-Datei (`MyServerImpl.class`) (hier Annahme nur eine Server-Klasse)

```
javac MyServerImpl.java
```

Erzeugung einer RMI-Anwendung (3)

- **Entwicklung eines Clients**

  ◇ Wesentliche **Funktionalität** des Clients :

  ► Erzeugung und Installation eines **Security Managers**

  ► Erfragen einer **Referenz** auf das entfernte RMI-Server-Objekt bei der **RMI-Registry**

  ► **Aufruf** der **RMI-Methoden** über die erhaltene Referenz

  ◇ Die Erzeugung und Installation eines **Security Managers** erfolgt analogt wie beim Server :

```
if (System.getSecurityManager()==null)
{
  System.setSecurityManager(new RMISecurityManager());
}
```

  ◇ Erfragen einer Referenz auf das entfernte RMI-Server-Objekt bei der **RMI-Registry**
  Mittels der statischen Methode **`lookup()`** der Klasse **`Naming`** :

```
public static Remote lookup(String name) throws NotBoundException,
                                                MalformedURLException, RemoteException
```

  Parameter : `name` - URL-formatierter Stríng, der das zu suchende Objekt bezeichnet
  - Allgemeine Form : `"//host:port-nr/ObjectName".`
  - Als Host (`host`) ist der Rechner (Name oder IP-Adresse) anzugeben, auf dem sich das gesuchte Objekt und die RMI-Registry befindet.
  - Wenn die RMI-Registry auf dem Default-Port `1099` läuft, kann die Angabe der Port-Nr. (`port-nr`) weggelassen werden
  - "`ObjectName`" ist der Name des Objekts, unter dem es in der Registry registriert worden ist.

  Die Methode `Naming.lookup()` erzeugt unter Verwendung von Host und Port-Nr. ein Registry-Stub-Objekt, über das die Methode `lookup()` für die Registry `mit "ObjectName"` als Parameter aufgerufen wird.
  Diese gibt – sofern ein Eintrag in der Registry vorhanden ist – ein Stub-Objekt für das gesuchte RMI-Server-Objekt zurück.
  `Naming.lookup()` gibt dieses Stub-Objekt als Referenz auf das RMI-Server-Objekt an den aufrufenden Client zurück (als Instanz des Interfaces `Remote`).
  Der Stub-Code wird – sofern er nicht lokal über die `CLASSPATH-`Environmentvariable gefunden wird - von der in der Registry für die Stub-Klasse abgelegten Codebase geladen.
  Der Client wandelt den Typ des Stub-Objekts in den Typ des vom RMI-Server-Objekt implementierten RMI-Interfaces um.

  ◇ Der Client referiert das erhaltene Stub-Objekt als Instanz des vom RMI-Server-Objekt implementierten RMI-Interfaces. Über diese Instanz erfolgt ein Aufruf der RMI-Methoden.

  ◇ **Übersetzung** der Java-Quell-Datei (`MyClient.java`)
  → Erzeugung einer Java-Bytecode-Datei (`MyClient.class`)

```
javac MyClient.java
```

  Mit der **`"d"`**-Option kann zusätzlich das **Directory** festgelegt werden, in dem die erzeugten Bytecode-Dateien abgelegt werden sollen. z.B. :

```
javac –d G:\MyJava\classes MyClient.java
```

Erzeugung einer RMI-Anwendung (4)

- **Erzeugung der Stub- und Skeleton-Klassen**

  ◇ Mittels des **RMI-Compilers** (Stub-Generators) **`rmic`** aus der das RMI-Interface implementierenden RMI-Server-Klasse und dem RMI-Interfacc

  ◇ Dem RMI-Compiler ist der **voll-qualifizierte Package-Name** der als Java-Byte-Code vorliegenden **RMI-Server-Klasse** als Parameter zu übergeben.

  ◇ Die den Java-Byte-Code der **RMI-Server-Klasse** enthaltende Datei (`class`-file) sowie die Datei mit dem Java-Byte-Code (oder mit dem Quell-Code) des **RMI-Interfaces** müssen über die **`CLASSPATH`**-Environment-Variable erreichbar sein.

  ◇ Zusätzlich kann beim Aufruf von `rmic` mit der **`"-d"` Option** angegeben werden, in welchem **Ausgangs-Directory** die erzeugten Java-Byte-Code-Dateien (`class`-files) für Stub und Skeleton abgelegt werden sollen
  (Ausgehend von diesem Directory erfolgt die Ablage in einer dem vollqualifizierten Package-Namen entsprechenden Directory-Struktur).
  Ohne diese Angabe dient das aktuelle Directory als Ausgangs-Directory.

  ◇ Die von `rmic` erzeugten Klassen-Dateien für Stub und Skeleton tragen den um **`_Stub`** bzw **`_Skel`** ergänzten Hauptnamen der RMI-Server-Klassen-Datei

  ◇ **Beispiel** : (ohne Verwendung eines Package-Namens)

  ```
  rmic –d G:\MyJava\classes MyServerImpl
  ```

  Hierdurch werden die Dateien **`MyServerImpl_Stub.class`** und **`MyServerImpl_Skel.class`** im Directory `G:\MyJava\classes` erzeugt.

- **Starten der RMI-Registry**

  ◇ Üblicherweise wird die RMI-Registry als eigener Prozeß (in einer eigenen JVM laufend) auf dem Server-Rechner von der Kommandozeile aus mit dem Kommando **`rmiregistry`** gestartet.
  Falls die Registry nicht auf dem Default-Port `1099` arbeiten soll, muß die Port-Nr. beim Aufruf als Parameter angegeben werden.

  ◇ **Beispiel** : **rmiregistry** (→ Registry läuft auf Port-Nr. `1099`)
  rmiregistry `1248` (→ Registry läuft auf Port-Nr. `1248`)

  ◇ Alternativ kann die Registry auch lokal für den Server-Prozeß von diesem erzeugt und gestartet werden
  (Statische Methode `createRegistry()` der Klasse `LocateRegistry`)

- **Start des Servers und des Clients**

  ◇ Auf dem Server- und dem Client-Rechner sind jeweils eine **JVM** mit der **Klasse**, die die jeweilge **`main()`**-Methode enthält, zu starten.
  Dabei ist der **vollqualifizierte Klassenname** anzugeben.
  Die Klassendatei muß über die **`CLASSPATH`**-Environment-Variable **erreichbar** sein.

  ◇ Als **Optionen** können die **Codebase** (URL) und die zu verwendende **Policy-Datei** (Dateipfad) angegeben werden

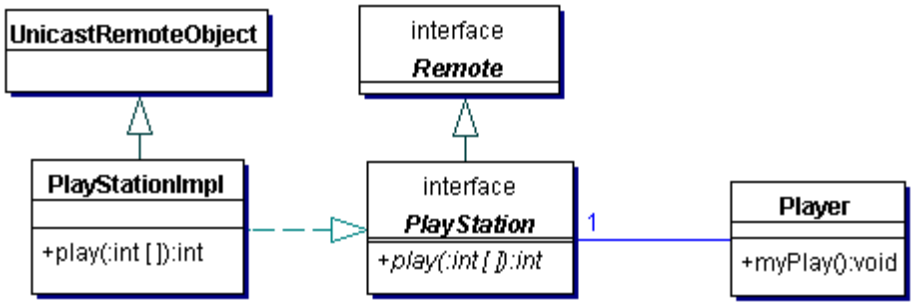
  ◇ **Beispiel** (für Server, kein Package-Name) :

  ```
  java –Djava.rmi.server.codebase=http://cd2/~thomas/classes/
       -Djava.security.policy=my.policy MyServerImpl
  ```

RMI von Java – Beispiel (1)

- **Aufgabenstellung**
  - ◇ Nachbildung eines vereinfachten **lotto-ähnlichen Spiels** :
  - ◇ Ein Server-Objekt soll eine mittels RMI aufrufbare Funktion `play()` zur Verfügung stellen.
    Der Funktion werden drei `int`-Werte, die im Bereich `1 .. 36` liegen müssen, als Eingabeparameter übergeben.
    Die Funktion erzeugt drei im gleichen Werte-Bereich liegende Pseudozufallszahlen und vergleicht sie auf Übereinstimmung mit den übergebenen Parameterwerten.
    Die Anzahl der übereinstimmenden – d.h. richtig geratenen – Werte gibt sie als Funktionswert zurück.
  - ◇ Ein Client-Programm fordert in einer Schleife zur Eingabe von jeweils drei `int`-Zahlen auf, liest diese von der Standardeingabe ein und ruft mittels RMI die von einem Remote-Objekt implementierte Funktion `play()` auf, der die eingelesenen Zahlen als Parameter übergeben werden
    Den von der Funktion zurückgegebenen Wert gibt das Programm als Anzahl der Treffer an die Standardausgabe aus.

- **Lösung**
  - ◈ Definition eines Remote-Interfaces `(PlayStation)`, das die Funktion `play()` zur Verfügung stellt
  - ◈ Definition einer Remote-Klasse `(PlayStationImpl)`, die das Remote-Interface implementiert.
    Die `main()`-Funktion dieser Klasse wird als Serverprogramm gestartet.
    Sie erzeugt ein Objekt ihrer Klasse und meldet dieses auf der RMI-Registry ihres Host-Rechners an ("binden").
    Das Objekt wartet anschließend auf Aufrufe durch einen Client.
  - ◈ Definition einer Client-Klasse `(Player)`.
    Die `main()`-Funktion dieser Klasse wird als Client-Programm gestartet.
    Sie erzeugt ein Objekt ihrer Klasse und ruft für dieses Objekt eine die eigentliche Client-Aktivität realisierende Member-Funktion `(myPlay())` auf.
    Diese Funktion beschafft sich durch Anfrage bei der RMI-Registry des Server-Rechners eine Referenz auf das Remote-Objekt (das ist dessen Stub-Objekt). Für dieses ruft sie für jeden eingelesenen Datensatz die Funktion `play()` auf.



- **Quelldatei mit Definition de Remote-Interfaces `(PlayStation.java)`**

```
// Einfaches-RMI-Beispiel : Definition des Remote-Interfaces PlayStation

package rt.remoteplay;

import java.rmi.*;

public interface PlayStation
  extends Remote
{
  public int play(int[] guess) throws RemoteException;
}
```

RMI von Java – Beispiel (2)

- **Quelldatei mit Server-Klasse `(PlayStationImpl.java)`**

```
// Einfaches RMI-Beispiel : Server-Klasse PlayStationImpl

package rt.remoteplay;

import java.rmi.*;
import java.rmi.server.*;

import rt.utilities.IntRandom;


public class PlayStationImpl
  extends UnicastRemoteObject
  implements PlayStation
{
  IntRandom rand36;     // Objekt zur Erzeugung von Int-Zahlen im Bereich (1..36)

  public PlayStationImpl() throws RemoteException
  { super();
    rand36=new IntRandom(1,36);
  }

  public int play(int[] guess)
  {
    int hit=0;
    int cnt=guess.length;
    int i, j;
    int num;

    for (i=0; i<cnt; i++)
    { num=rand36.nextVal();
      j=0;
      while (j<cnt)
        if (num==guess[j])
        { hit++;
          guess[j]=-1;
          j=cnt;
        }
        else
          j++;
    }
    return hit;
  }

  public static void main(String[] args)
  {
    if (System.getSecurityManager() == null)
      System.setSecurityManager(new RMISecurityManager());
    try
    {
      PlayStationImpl remPlay = new PlayStationImpl();
      Naming.rebind("RemPlayStation", remPlay);
      System.out.println("RemPlayStation bound in registry");
    }
    catch (Exception e)
    {
      System.out.println("PlayStationImpl error : " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

RMI von Java – Beispiel (3)

- **Quelldatei mit Client-Klasse `(Player.java)`**

```
// Einfaches RMI-Beipiel : Client-Klasse Player

package rt.remoteplay;

import java.rmi.*;

import rt.utilities.TextInputStream;

public class Player
{
  String server;

  public Player(String serv)
  {
    super();
    if (serv==null)
      server="";
    else
      server="//"+serv+"/";
    server=server+"RemPlayStation";
  }

  public void myPlay()
  {
    try
    {
      int myGuess[]={0,0,0};
      PlayStation playstat=(PlayStation)Naming.lookup(server);
      TextInputStream ein=new TextInputStream(System.in);
      System.out.print("Ein neues Spiel : ");
      while (!ein.eof())
      {
        myGuess[0]=ein.readInt();
        myGuess[1]=ein.readInt();
        myGuess[2]=ein.readInt();
        ein.readLine();
        System.out.print("Mein Spiel : ");
        System.out.print(playstat.play(myGuess));
        System.out.println(" Richtige");
        System.out.print("\nEin neues Spiel : ");
      }
    }
    catch(Exception e)
    {
      System.out.println("Player Exception "+e.getMessage());
      e.printStackTrace();
    }
  }

  public static void main(String[] args)
  {
    String host=null;
    if (System.getSecurityManager() == null)
      System.setSecurityManager(new RMISecurityManager());
    if (args.length!=0)
      host=args[0];
    Player me = new Player(host);
    me.myPlay();
  }
}
```