

Verteilte Systeme

Kommunikationsmodelle

OO Verteilte Anwendungen

OOP und Verteilung

Java RMI

-  J. Nehmer, P. Sturm
Grundlagen moderner Betriebssysteme.
dpunkt, 1998.
-  S. Mullender
Distributed Systems (2nd edition).
ACM Press, 1993.
-  A. S. Tanenbaum, M. van Steen
Distributed Systems.
Prentice Hall, 2002.
-  A. D. Birrel, B. J. Nelson
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems, 2(1), pp. 39–59, 1984.
-  M. J. Flynn
Some Computer Organizations and Their Effectiveness.
IEEE Transactions on Computers, C-21, pp. 948–960, 1972.

Distributed System

- ▶ Definition nach Tanenbaum und van Renesse:
 - ▶ It looks like an ordinary centralized system.
 - ▶ It runs on multiple, independent CPUs.
 - ▶ The use of multiple processors should be invisible (transparent).
- ▶ Definition nach Mullender
 - ▶ Zusätzlich: Not any single points of failures
- ▶ Lamport über verteilte Systeme
 - ▶ „A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.“

Definitionen sind nicht präzise

- ▶ Manchmal ist es schwierig, ein lokales bzw. verteiltes System zu identifizieren
- ▶ Definitionen basieren oft auf bestimmten Eigenschaften, die gerade wichtig erscheinen

Eigenschaften verteilter Systeme

- ▶ Physikalische Verteiltheit der Hardware
- ▶ Gemeinsame Nutzung von Ressourcen
- ▶ Heterogenität – Hardware (Netzwerk, Prozessor, etc.) und Software (Betriebssystem, Programmiersprache)
- ▶ Nebenläufigkeit – gemeinsame Nutzung von Betriebsmitteln

Verteilte Systeme

Wünschenswerte Eigenschaften verteilter Systeme

- ▶ Offenheit
 - ▶ Einheitliche Schnittstellen und Protokolle
- ▶ Skalierbarkeit
 - ▶ Verschiedene Dimension von Skalierbarkeit: Größe, Ausbreitung und Verwaltbarkeit
- ▶ Transparenz
- ▶ Sicherheit

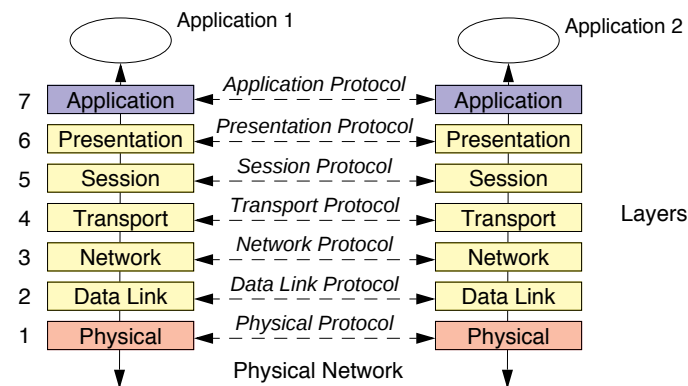
Verteilte Systeme

Transparenz

- ▶ Netzwerktransparenz
 - ▶ Zugriffstransparenz – Zugriff auf lokale und entfernte Betriebsmittel unter Verwendung identischer Operationen.
 - ▶ Ortstransparenz – Zugriff auf Betriebsmittel, ohne ihre Position bzw. ihren Ort kennen zu müssen
- ▶ Nebenläufigkeitstransparenz
- ▶ Replikationstransparenz
- ▶ Fehlertransparenz

Kommunikationsmodelle

Protokollschichten nach dem ISO OSI Referenzmodell



Klassifikation

Synchronität

- ▶ Wird der Sender blockiert bis der Empfänger die Nachricht hat, oder nicht?

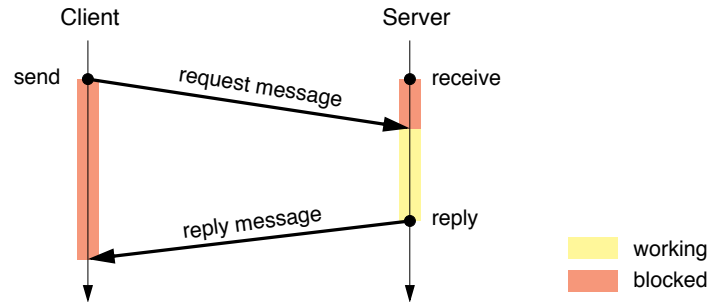
Interaktionsmuster

- ▶ **Message Passing** – eine Nachricht wird von einem Teilnehmer an einen anderen gesendet
- ▶ **Request-Reply** (Client-Server) Interaktion – Nachricht an einen Empfänger und Nachricht zurück an den ursprünglichen Absender

Adressaten

- ▶ Ein Empfänger
- ▶ Mehrere Empfänger (Gruppenkommunikation, Multicast, Broadcast)

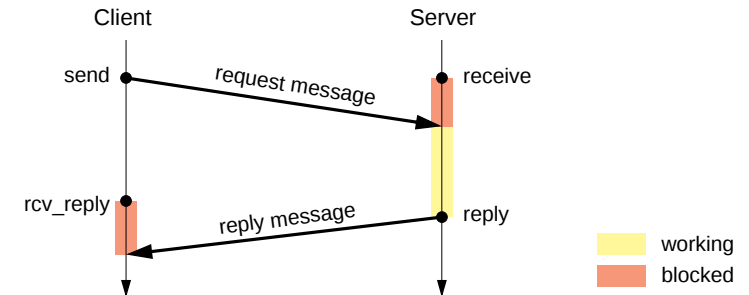
Synchrones Request-Reply Modell



Request-Reply Interaktion bzw. synchrones Senden

- ▶ Sender wartet bis Antwort-Nachricht empfangen wurde
- ▶ Empfänger ist blockiert bis eine Nachricht eintrifft
- ▶ Client und Server arbeiten nicht nebenläufig
- ▶ Zum Beispiel: Remote Procedure Call (RPC)

Asynchrones Request-Reply Modell



Request-Reply Interaktion bzw. asynchrones Senden

- ▶ Client und Server arbeiten nebenläufig
- ▶ Basis für Gruppenkommunikation

Zuverlässigkeit

Nachrichten können verloren gehen wenn keine zuverlässige Verbindung benutzt wird

- ▶ zuverlässige Verbindungen benutzen Acknowledge-Nachrichten (ACK)
- ▶ für einfache Nachrichtenübertragung großer Overhead

⇒ Zuverlässigkeit mit Request-Reply Interaktionsmodell kombinieren

Mögliche Fehler

- ▶ Server Crash (Server verliert alle Informationen über frühere Anfragen)
- ▶ Anfrage-Nachricht geht verloren
- ▶ Antwort-Nachricht geht verloren

Ideale Semantik

- ▶ **exactly-once** die Anfrage wird genau einmal auf Serverseite bearbeitet

Zuverlässigkeit

At-Least-Once Semantik

- ▶ Anfrage wird einmal oder mehrmals bearbeitet
- ▶ Client bekommt nie eine Fehlermeldung aber er erkennt eventuell, dass die Anfrage mehrfach bearbeitet wurde: Operationen müssen idempotent sein!

Implementation

- ▶ wenn der Client nach einiger Zeit (time-out) keine Antwort erhält, wiederholt er die Anfrage
 - ▶ keine zusätzliche Funktionalität auf Server-Seite erforderlich
 - ▶ der Server darf wiederholte Anfrage aber ignorieren, wenn er sie erkennen kann

Zuverlässigkeit

At-Most-Once Semantik

- Die Anfrage wird höchstens einmal (oder überhaupt nicht) bearbeitet

Einfache Implementation (nur auf der Client-Seite)

- wenn das Ergebnis innerhalb einer bestimmten Zeit nicht eintrifft wird dem Aufrufer eine Fehlermeldung zurückgegeben
- sonst wird das Ergebnis zurückgegeben (exactly-once Semantik)

Komplexere Implementation

- Client wiederholt Nachricht nach einem Time-out (verdeckt Nachrichtenverlust auf der Verbindung)
- Client muss Server-Abstürze erkennen (Fehlermeldung an den Aufrufer, at-most-once Semantik)
- Server hebt Antworten auf (Wiederholung bei Verlust der Nachricht)
- Server muss alte Anfragen nach einem Absturz erkennen und ignorieren
- wenn das Ergebnis zurückgegeben wird, haben wir exactly-once Semantik

Remote Procedure Calls

Request-Reply-Modell kann für die Implementierung von RPCs genutzt werden [Birrell and Nelson 1984]

- statt eine Anfrage zu senden wird eine Remote-Prozedur aufgerufen
- statt einer Antwort erhält man die Ergebnisse des Aufrufs

Aufruf einer Prozedur ist ortstransparent

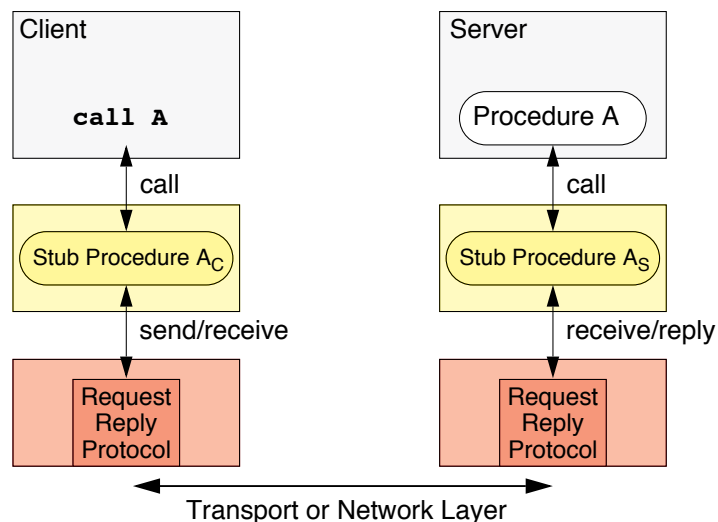
- Syntax für lokale und remote Aufrufe kann identisch sein
- sehr intuitiv
- keine expliziten send()- und receive()-Anweisungen erforderlich

Implementierung von RPCs

- Stub-Prozeduren auf Client- und Server-Seite

Remote Procedure Calls

Implementierung von RPCs mit Stub-Prozeduren



Remote Procedure Calls

Client-Stub

- Marshalling der Parameter (Zusammenstellen einer Anfrage-Nachricht)
- Senden der Anfrage-Nachricht
- Warten auf die Antwort-Nachricht
- Unmarshalling des Ergebnisses
- Implementierung der Zustell-Semantik

Server-Stub

- Marshalling der Parameter (Zusammenstellen einer Anfrage-Nachricht)
- Senden der Anfrage-Nachricht
- Warten auf die Antwort-Nachricht
- Unmarshalling des Ergebnisses
- Implementierung der Zustell-Semantik

Remote Procedure Calls

Probleme bei RPCs

- ▶ Marshalling der Parameter
 - ▶ Zahl und Typen sollten bekannt sein sowie ihre Codierung
- ▶ Parameter-Übergabe-Semantik
 - ▶ **Call-by-value/result**: unproblematisch
 - ▶ **Call-by-reference**: wie implementieren?
- ▶ keine globalen Variablen
- ▶ Semantik – keine exactly-once Semantik
- ▶ Performance
 - ▶ keine Nebenläufigkeit
 - ▶ große Parameter-Daten
 - ▶ kurze Prozeduren
 - ▶ Antwortzeiten

Remote Procedure Calls

Automatische Erzeugung von Stub-Prozeduren

- ▶ Parameter-Marshalling
- ▶ Client-Stub
- ▶ Server-Stub Prozedur
- ▶ Server-Schleife zum Warten auf Anfragen

Binden von Client-Stubs an Server-Stubs

- ▶ Server-Stub hat Netzwerk-Adresse, die Client-Stub kennen muss
- ▶ Problem: woher kennt der Client den Server?

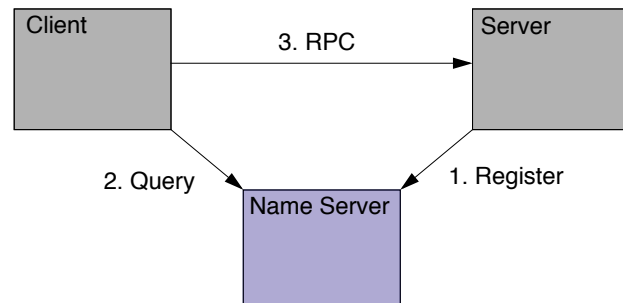
Name-Server

- ▶ Symbolische Namen werden in Netzwerk-Adressen umgesetzt

Name-Server und Binden

Bekannter Name-Server wandelt Namen in Adressen um

- ▶ Client kennt Namen seines Servers und die Adresse eines Name-Servers
- ▶ Name-Server wandelt Namen in eine (dynamische) Netzwerk-Adresse um
- ▶ Client kann sich immer an Server binden
- ▶ Server muss seine Netzwerkadresse beim Name-Server registrieren



OO Verteilte Anwendungen

Objektorientierte Anwendungen im verteilten System

- ▶ Verteilung auf
 - ▶ verschiedene Rechner
 - ▶ verschiedene Prozessoren
 - ▶ verschiedene virtuelle Maschinen
 - ▶ verschiedene Adressräume
- ▶ Objektinteraktionen zwischen objektorientierten Programmen oder Aufteilung eines Programmes in interagierende Programmteile?

OOP und Verteilung

Formen/Ausprägungen von Objektinteraktion im Verteilten System

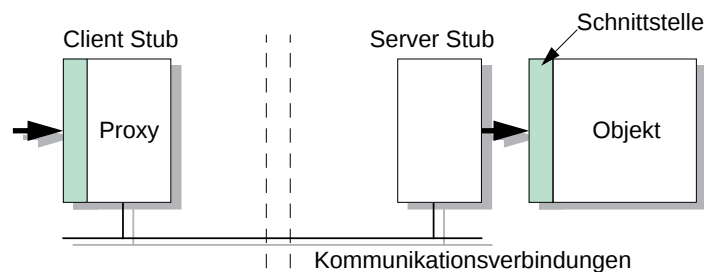
- ▶ explizit
- ▶ implizit
- ▶ orthogonal
- ▶ nicht-orthogonal
- ▶ uniform
- ▶ nicht-uniform
- ▶ transparent
- ▶ nicht-transparent

explizite, orthogonale Interaktion

- ▶ weit verbreitete Vorgehensweise
- ▶ von klassischen Interprozesskommunikations-Mechanismen geprägt
 - ▶ Nachrichten (Datagramm-Sockets, Messages,)
 - ▶ Verbindungen (Stream-Sockets, Pipes,)
- ▶ Vorteile
 - ▶ meist weit verbreitete, etablierte Infrastruktur vorhanden
 - ▶ kein „unsichtbarer“ Overhead
- ▶ Nachteile
 - ▶ Programmierung aufwändig
 - ▶ Bruch im objektorientierten Paradigma
 - ▶ Serialisierung von Parametern, Verlust von Typ-Information
 - ▶ Verteilung wird durch die Programmierung „fest verdrahtet“
 - ▶ Software sehr unflexibel in Bezug auf Änderungen

implizite, nicht-orthogonale Interaktion

- ▶ Interaktion zwischen lokalen und verteilten Objekten unterscheidet sich prinzipiell nicht
 - ▶ nur ein Interaktionsmechanismus: Methodenaufruf
- ▶ grundlegendes Konzept: Remote Procedure Call
 - ▶ Verteilung wird durch Vermittler- oder Stellvertreterobjekte vor den Kommunikationspartnern (weitgehend) verborgen
 - ▶ Proxy/Stub-Prinzip



uniforme / nicht-uniforme Interaktion

nicht-uniforme Interaktion

- ▶ unterschiedliche Methodenaufrufe für lokale und remote-Referenzen
- ▶ unterschiedliche Semantik bei der Parameterübergabe
 - ▶ by reference
 - ▶ by value
 - ▶ Problem: Übergabe von lokalen Objektreferenzen

uniforme Interaktion

- ▶ keinerlei Unterschied zwischen lokalen und remote-Aufrufen

transparente / nicht-transparente Verteilung

volle Transparenz

- ▶ Anwendungsentwickler sieht keinerlei Unterschied zwischen lokalen und verteilten Objekten
- ▶ Probleme:
 - ▶ im verteilten Fall können spezielle Fehler auftreten
 - ▶ unabhängiger Objektausfall signalisiert durch Remote Exceptions
 - ▶ verteilte Interaktion ist implizit signifikant teurer
 - ▶ Transparenz kann zu Ineffizienz führen
 - ▶ Verteilung ist häufig ein Entwurfskriterium
 - ▶ Verbergen der Verteilung in der Implementierung ist unsinnig
- ▶ Fazit:
 - ▶ bei der Programmierung sollte zwischen potentiell verteilten und definitiv lokalen Objekten unterschieden werden können

Herausforderungen

- ▶ Überwindung heterogener Hardware- und Softwarestrukturen
 - ▶ verschiedene Hardware
 - ▶ verschiedene Betriebssysteme
 - ▶ verschiedene Programmiersprachen
- ▶ Ortstransparenz
 - ▶ statische Konfiguration
 - ▶ Objektmigration
- ▶ Globale Dienste
 - ▶ z.B. Namensdienste, Transaktionsdienst, Persistenz, Kontrolle der Nebenläufigkeit

Java Remote Method Invocation (Java RMI)

- ▶ Erweiterung des Java Programmiermodells für Verteilung
- ▶ mehrere JVMs spannen eine verteilte Rechenplattform auf
- ▶ Verteilungseinheiten
 - ▶ Java Objekte
- ▶ Verteilungstransparenz
 - ▶ orts- und zugriffstransparente Methodenaufrufe an verteilten Java Objekten

Literatur



Sun Microsystems

Java Remote Method Invocation Specification

<http://java.sun.com/javase/6/docs/platform/rmi/spec/rmiTOC.html>



J. Waldo

Remote procedure calls and Java Remote Method Invocation

IEEE Concurrency, 6(3):57, 1998.

Java Remote Method Invocation (Java RMI)

Infrastruktur zur Unterstützung von verteilten Java-Anwendungen

- ▶ Server-Anwendung
 - ▶ erzeugt Objekte
 - ▶ macht Objektreferenzen verfügbar
 - ▶ wartet auf Methodenaufrufe
- ▶ Client-Anwendung
 - ▶ besorgt sich Remote-Referenz
 - ▶ ruft Methoden an entferntem Objekt auf

Java Remote Method Invocation (Java RMI)

Probleme

- ▶ Erzeugen eines verteilten Java Objekts (RMI-Objekts)
- ▶ Entfernte Objekte finden
- ▶ Methodenaufuf
- ▶ Übergabe von Objekten bzw. Aufrufparametern

Verteilte Java Objekte

Beispiel

```
public interface Printer implements java.rmi.Remote{
    public void print( String s )
        throws java.rmi.RemoteException;
}
```

Entfernt ansprechbare Methoden

- ▶ entfernt ansprechbar sind nur Methoden von Interfaces welche von Remote erben
- ▶ andere Methoden und Instanzvariablen sind nur lokal ansprechbar

Referenzen

- ▶ Referenzen nur mit dem Namen eines Remote-Interfaces typisieren
- ▶ nicht mit dem Klassennamen der Implementierung

Verteilte Java Objekte

Schnittstelle

- ▶ verteilte RMI-Objekte implementieren spezielle Java-Interfaces
 - ▶ mindestens ein Java-Interface, das direkt oder indirekt von `java.rmi.Remote` erbt
 - ▶ dient als Marker, enthält keine Methoden
- ▶ alle Methoden in von `java.rmi.Remote` erbenden Interfaces müssen die Exception `java.rmi.RemoteException` deklarieren
 - ▶ zeigt Fehler bei entfernten Aufrufen an
- ▶ es kann auch eine Basisklasse deklariert werden, z.B. `java.io.IOException` oder `java.lang.Exception`

Erzeugen eines RMI-Objekts

Zwei Vorgehensweisen

- ▶ Implementierungsklasse implementiert mindestens ein Remote-Interface
- ▶ Variante A:
 - ▶ Implementierungsklasse erbt von `java.rmi.server.UnicastRemoteObject`
 - ▶ Objektinstanzen sind sofort nach der Erzeugung exportiert!
- ▶ Variante B:
 - ▶ manuelles Vorbereiten für entfernte Aufrufe:
 - ▶ Aufruf von `java.rmi.server.UnicastRemoteObject.exportObject(obj,...)`
 - ▶ Implementierung verschiedener Methoden von Objekt unter Umständen nötig

Erzeugen eines RMI-Objekts

Beispiel für Vererbungsvariante:

```
public class MyPrinter extends java.rmi.UnicastRemoteObject
implements Printer {
    public void print(String s) throws java.rmi.RemoteException{
        ...
    }
    ...
}
```

RMI Compiler

- ▶ erzeugt Subklassen für Clientseite
 - ▶ ab Java 1.5 nicht mehr nötig da generische Stubs
- ▶ erzeugt Skeletonklassen für Serverseite
 - ▶ ab Java 1.2 nicht mehr nötig da generische Skeletons
- ▶ Kommando: `rmic`
 - ▶ wird mit der Implementierungsklasse aufgerufen

Entfernte Objekte finden

Namesdienste zum Registrieren von Objekten

- ▶ Klasse `java.rmi.registry.LocateRegistry`
 - ▶ erlaubt das Erzeugen einer Namensdienst-Instanz
 - ▶ ermöglicht das Erzeugen einer Objektreferenz zu einem Namensdienst
 - ▶ benötigt Hostname und Portnummer
- ▶ Objektreferenz ist vom Remote-Interface `java.rmi.registry.Registry`
 - ▶ `bind(String s, Remote obj)` registriert Objekt `obj` unter Name `s`
 - ▶ `lookup(String s)` liefert Objektreferenz für Objekt mit Name `s`

Entfernte Objekte finden

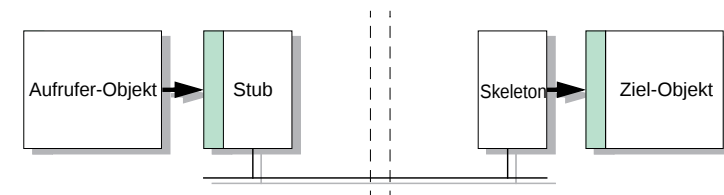
Standard Mechanismen zum Registrieren und Finden von Objekten

- ▶ `rmiregistry` einfacher Namensdienst zur Umwandlung von Namen in Remote-Referenzen
- ▶ Name = URL, bestehend aus Registry-Host[:Port] und Objektnamen
- ▶ Spezielle Klasse `java.rmi.Naming` für transparenten Zugriff auf Nameserver
 - ▶ Server meldet Objekt bei seiner Registry an
 - ▶ `void Naming.bind(String name, Remote obj)`
 - ▶ `void Naming.rebind(String name, Remote obj)`
 - ▶ Client bekommt Referenz von Registry
 - ▶ `Remote Naming.lookup(String name)`

Alternative

- ▶ Ein Objekt erhält Remote-Referenz als Parameter oder als Ergebnis eines Methodenaufrufs

Methodenaufruf



Klassische Stub-/Skeleton-Technik

- ▶ Stub-Klasse wird bei Bedarf automatisch vom Server geladen (`RMIClassLoader`)
- ▶ Ausführungssemantik: „at most once“
 - ▶ bei Fehler wird `RemoteException` ausgelöst

Parameterübergabe

Parameterübergabesemantik bei lokalen Aufrufen

- ▶ **Call-By-Value** für Nicht-Objekttypen (z.B. Integer)
- ▶ **Call-By-Object-Reference** für Standard-Java-Objekte, lokale und entfernte RMI-Objekte
- ▶ entfernte RMI-Objekte
 - ▶ Referenz auf Stubobjekt wird übergeben
 - ▶ Client verwendet Stubobjekt wie lokales Objekt

Parameterübergabe

Parameterübergabesemantik bei entfernten Aufrufen (fortges.)

- ▶ nicht-serialisierbare Java-Objekte
 - ▶ nicht übertragbar: `MarshalException`
- ▶ lokale, exportierte RMI-Objekte
 - ▶ Einfügung eines geeigneten Stubobjekts in den Bytestrom
 - ▶ Regenerierung des Stubobjekts auf der Empfängerseite
 - ▶ **Call-By-Object-Reference** Semantik für lokale RMI-Objekte
- ▶ entfernte, exportierte RMI-Objekte
 - ▶ Einfügung des vorhandenen Stubobjekts in den Bytestrom
 - ▶ **Call-By-Object-Reference** Semantik für entfernte RMI-Objekte

Parameterübergabe

Parameterübergabesemantik bei entfernten Aufrufen

- ▶ **Marshalling** der Parameter
 - ▶ Parameter werden serialisiert (Java Serialization)
 - ▶ Erzeugung eines Bytestroms aus den Parametern
 - ▶ Übertragung des Bytestroms
 - ▶ Regenerierung der Datentypen und Java-Objekte aus Bytestrom
- ▶ **Call-By-Value** für Nicht-Objekttypen (z.B. Integer)
 - ▶ Datentypen werden kopiert
- ▶ serialisierbare Java-Objekte und lokale, nicht-exportierte RMI-Objekte
 - ▶ Serialisierung führt zu vollständiger Kopie eines Objekts und seiner untergeordneten Objekte
 - ▶ evtl. große transitive Hülle zu übertragender Daten aber Erhaltung der Objektstruktur über alle Parameter
 - ▶ **Call-By-Copy** Semantik

Programmiermodell

Erweiterung des Java-Programmiermodells

- ▶ Wesentliche Änderungen gegenüber Java
 - ▶ Interface obligatorisch
 - ▶ Exception-Handling obligatorisch
- ▶ veränderte Parameterbergabesemantik bei übergebenen Nicht-RMI-Objekten
 - ▶ Zugriffstransparenz verletzt
- ▶ Verändertes locking z.B. keine Erkennung von doppelten Object-Locks
 - ▶ Deadlock beim Wiedereintritt in das selbe RMI-Objekt

Programmiermodell

Bewährte Konzepte aus dem Java-Programmiermodell

- ▶ Objekte
 - ▶ Methodenaufrufe syntaktisch gleich
- ▶ Parameterübergabesemantik für übergebene RMI-Objekte und Nicht-Objekte (primitive Datentypen wie Integer)
- ▶ Typisierung
 - ▶ volle Typisierung im verteilten Fall
 - ▶ (dynamische) Type-Casts auch für RMI-Objekte möglich
 - ▶ Stubobjekte besitzen alle Methoden des Originals
- ▶ Garbage Collection
 - ▶ transparente Speicherbereinigung
 - ▶ verteiltes Verfahren zur Speicherbereinigung basierend auf Leases

Resume

- ▶ einfacher, speziell auf Java abgestimmter Fernaufrufmechanismus
- ▶ durch dynamisches Laden von Code hohe Flexibilität
- ▶ implizite, nicht-orthogonale Interaktion (durch Remote-Referenzen)
- ▶ Verteilung nicht transparent