

Universität Karlsruhe (TH)

Forschungsuniversität · gegründet 1825

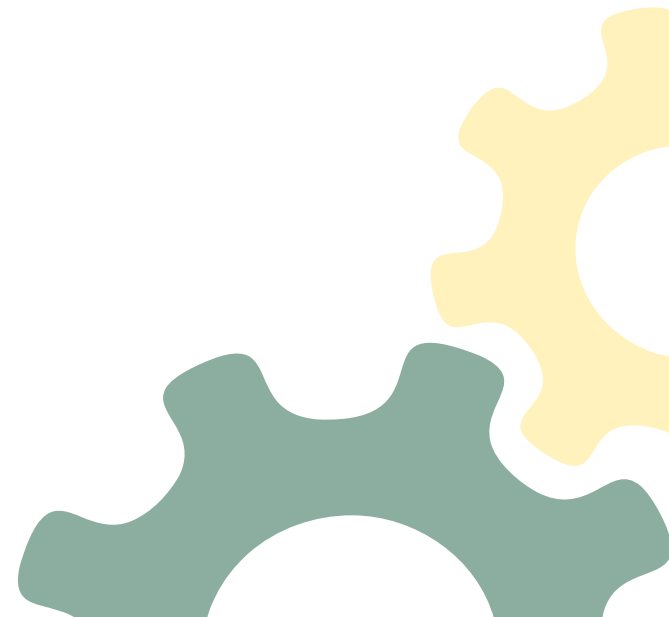
JavaParty: Javas Begleiter für Verteiltes Rechnen

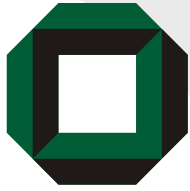
Prof. Dr. Walter F. Tichy
Dr. Victor Pankratius
Thomas Moschny
Ali Jannesari



Fakultät für **Informatik**

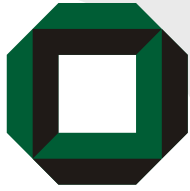
Lehrstuhl für Programmiersysteme





Inhalt

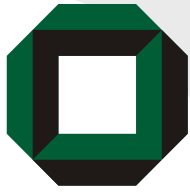
- JavaParty:
 - „Javas Begleiter für verteiltes Rechnen“
 - Übernahme der Java-Konzepte für Parallelität auf verteilte parallele Systeme
 - Ziele und Funktionsweise von JavaParty
 - schneller entfernter Methodenaufruf: *KaRMI*
 - schnelle Objektserialisierung: *uka.transport*
 - transparente, maschinenüberspannende Kontrollfäden
 - replizierte, verteilte Objekte
 - JavaParty Anwendungen
 - Dank:
 - Die ursprüngliche Version dieses Foliensatzes stammt von Bernhard Haumacher.



Wissenschaftliches Rechnen mit Java

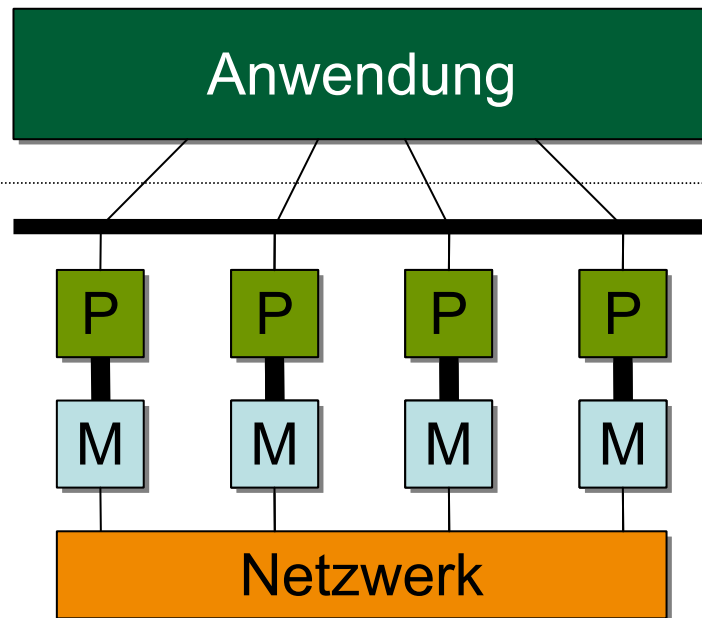
- Sprachdesign
 - klares, objektorientiertes Konzept
 - automatische Speicherbereinigung
- Portabilität: Für gewöhnlich genügt eine Java Virtuelle Maschine auf einer Plattform, damit das Programm läuft.
- Reicher Fundus an Bibliotheken vorhanden.
- Basis-Infrastruktur für verteiltes Rechnen.
- Breite Akzeptanz.
- Java wird gelehrt und gelernt.

JavaGrande.org



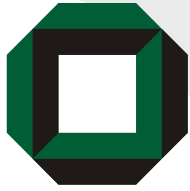
DSM Systeme

- DSM „gaukelt“ der Anwendung transparent einen gemeinsamen Speicher vor.
- DSM: *Distributed Shared Memory*



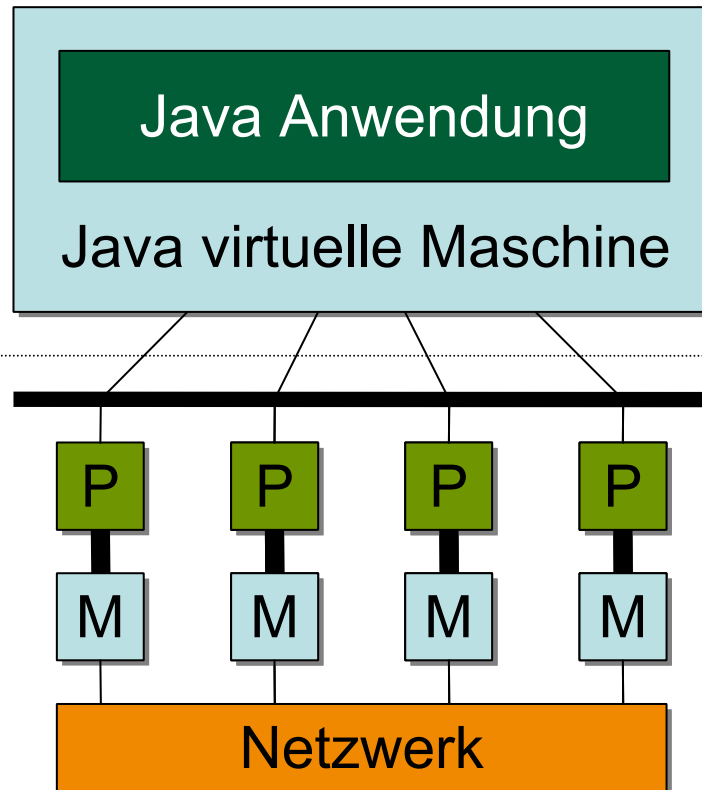
Betriebssystem
der beteiligten
Rechenknoten

DSM Abstraktion

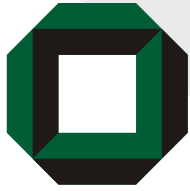


Java DSM Systeme

- Idee: Verwende ein DSM-System und starte die virtuelle Maschine darin, wie jede andere Anwendung.
- z.B. Java/DSM (auf TreadMarks).

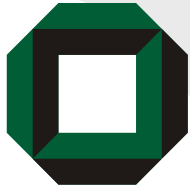


Betriebssystem
der beteiligten
Rechenknoten



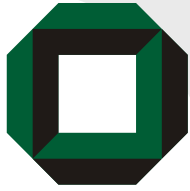
Java DSM Systeme

- Von DSM geerbte Probleme:
 - Deckungsproblem der Speicherkonsistenz des DSM mit den Speicherkonsistenz-Anforderungen der virtuellen Maschine
 - Anpassungen an die Java VM notwendig? möglich?
 - Problem des „False Sharing“:
 - Java Objekte sind nicht deckungsgleich mit Speicherseiten.
 - „Viele kleine“ Objekte in Java verschärfen das Problem.
- Ausweg: Objektbasiertes DSM (Objekte statt Seiten), aber:
 - Zusammenarbeit des DSM mit der VM notwendig, daher:
 - Standard Java VM nicht mehr einsetzbar.
 - Lösung nicht mehr plattformunabhängig.



Der JavaParty Ansatz

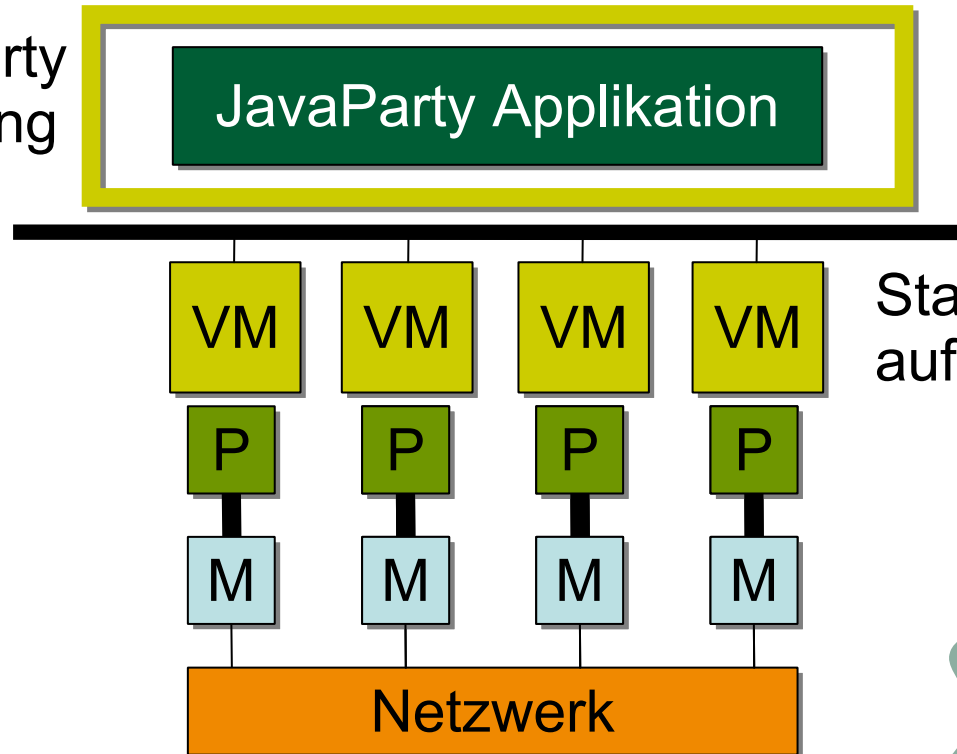
- JavaParty stellt keine besondere Anforderung an das Betriebssystem.
 - Einzige Voraussetzung: Java VM ist verfügbar.
- JP verwendet eine Standard Java VM („pure Java“)
 - Die Sprache JavaParty wird durch eine reine Quellcode-Transformation auf Java abgebildet: JavaParty → Java.
 - Das Laufzeitsystem ist in Java implementiert.
- JP spiegelt dem Anwendungsprogramm eine verteilte Virtuelle Maschine vor
 - Entfernte Objekte werden transparent in der Umgebung verteilt.
- Auf jedem Knoten muss eine virtuelle Maschine gestartet werden.



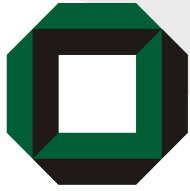
Der JavaParty Ansatz

- JavaParty erzeugt die Illusion einer verteilten Java VM.

Verteilte JavaParty
Laufzeitumgebung

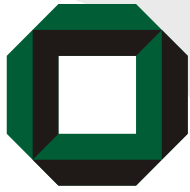


Standard Java VM
auf jedem Knoten



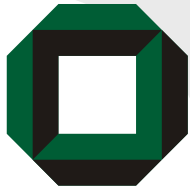
Java für *parallele* Systeme

- Java ist gut geeignet für parallele Programmierung bei gemeinsamem Speicher (SMP Modell)
 - Parallele Kontrollfäden sind in die Sprache integriert:
 - *java.lang.Thread*
 - Synchronisationsmechanismen sind als Sprachprimitive vorhanden
 - synchronisierte Methoden “**synchronized** void foo()”
 - synchronisierte Blöcke “**synchronized** (obj) { ... }”
 - Inter -Thread - Kommunikation
 - *java.lang.Object*: **wait()** & **notify()**



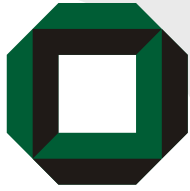
Java für *verteilte* parallele Systeme

- Keine Sprachunterstützung, aber Bibliotheken:
- Alternative 1: Socketbibliothek (*java.net*)
 - ☹ Keine entfernten Objektreferenzen
 - Nur Kommunikationsendpunkte
 - ☹ Keine Methodenaufrufe an entfernt liegende Objekte
 - Kommunikation über Botschaften, daher:
 - Implementierung eines eigenen Kommunikationsprotokolls nötig
 - ☹ Keine automatische Speicherbereinigung
 - Es gibt kein Konzept einer „entfernten Referenz“.



Java für *verteilte* parallele Systeme

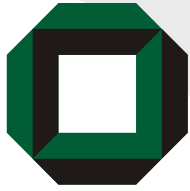
- Alternative 2: Entfernter Methodenaufruf (*java.rmi.**)
 - ☺ Entfernte Referenzen auf Objekte
 - ☺ Kein eigenes Kommunikationsprotokoll notwendig
 - ☺ Verteilte automatische Speicherbereinigung
 - ☹ Explizite Namensbindung (durch eine „Registry“)
 - ☹ Kein entfernter Zugriff auf Instanzvariablen, statische Methoden und statische Variablen
 - ☹ Deklaration zahlreicher zusätzlicher Ausnahmebedingungen
 - ☹ Explizite Objektplatzierung durch lokale Erzeugung
 - ☹ Keine Ortstransparenz (Objekte sind lokal oder entfernt, und der Zugriff muss dementsprechend lokal oder entfernt erfolgen.)
 - ☹ Keine Objektmigration
- nur geeignet für Client-Server-Programmierung



JavaParty

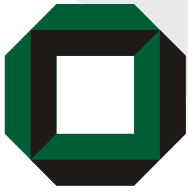
- Vorteile:

- ☺ Keine explizite Namensbindung
- ☺ Entfernter Zugriff auf die gesamte Klasse möglich
- ☺ Keine zusätzlichen Ausnahmebedingungen
- ☺ Entfernte Objekterzeugung, Platzierungsstrategien
- ☺ Ortstransparenz
- ☺ Objektmigration
- ☺ Übernahme von Javas Thread-Konzept

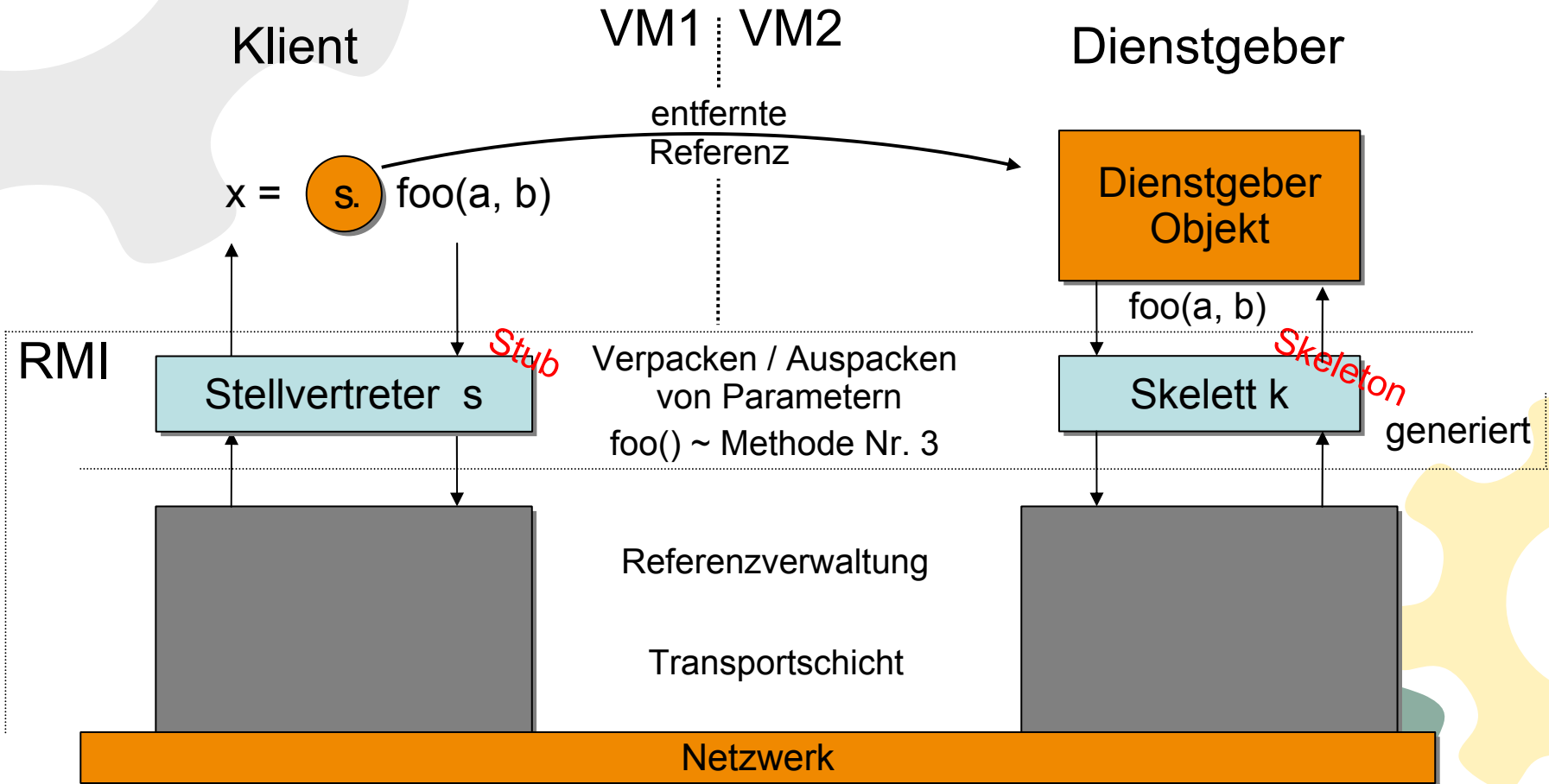


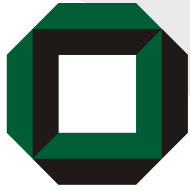
JavaParty

- JavaParty =
 - Verallgemeinert Javas Programmiermodell für eine potentiell verteilte virtuelle Maschine.
 - Verteilt transparent Java-Objekte und Threads.
- Vorteile:
 - Verbirgt Architekturdetails der Hardware-Plattform (SMP, DMP, Cluster, Cluster aus SMP).
 - Erleichtert Umstieg zu DMPs oder heterogenen Systemen.
 - Benutzt Javas (genauer: RMIs) entfernten Methodenaufruf.



Abläufe beim entfernten Methodenaufruf





RMI: Applikationsanpassungen

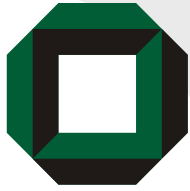
- Ausgangscode

```
class OverdrawnException extends Exception {}

public class BankAccount {
    public static int blz;
    public int accountNumber;

    public void deposit(float amount)
        { ... }
    public void withdraw(float amount) throws OverdrawnException
        { ... }
    public float balance()
        { ... }
}

BankAccount b = new BankAccount( ... );
b.deposit(100);
```



RMI: Applikationsanpassungen

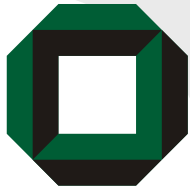
- Deklaration der entfernt aufrufbaren Schnittstelle

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

public interface BankAccountIntf extends Remote {
    /** Zugriffsmethode für blz */
    public int getBLZ() throws RemoteException;

    /** Zugriffsmethode für accountNumber */
    public int getAccountNumber() throws RemoteException;

    public void deposit(float amount) throws RemoteException;
    public void withdraw(float amount)
        throws OverdrawnException, RemoteException;
    public float balance() throws RemoteException;
}
```



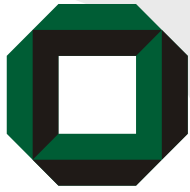
RMI: Applikationsanpassungen

- Änderung der Implementierungsklasse

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

public class BankAccount extends UnicastRemoteObject
    implements BankAccountIntf
{
    public int getBLZ() {return blz; }
    public int getAccountNumber() {return accountNumber; }

    public void deposit(float amount) throws RemoteException
        { ... }
    public void withdraw(float amount)
        throws RemoteException, OverdrawnException
        { ... }
    public float balance() throws RemoteException
        { ... }
}
```



RMI: Applikationsanpassungen

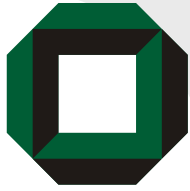
- Anpassungen bei Erzeugung und Benutzung

server-side

```
String url = "rmi://myserver.nowhere.com/account";  
try {  
    BankAccount b = new BankAccount( ... );  
    java.rmi.Naming.bind(url, b);  
}  
catch (RemoteException r) { ... }  
catch (java.net.MalformedURLException m) { ... }  
catch (java.rmi.AlreadyBoundException a) { ... }
```

client-side

```
try {  
    BankAccount b = (BankAccount) java.rmi.Naming.lookup(url);  
    b.deposit(100);  
}  
catch (RemoteException ex) { ... }  
catch (NotBoundException ex) { ... }  
catch (MalformedURLException ex) { ... }
```



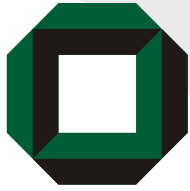
Dasselbe Beispiel mit JavaParty

```
class OverdrawnException extends Exception {}

public remote class BankAccount {
    public static int blz;
    public int accountNumber;

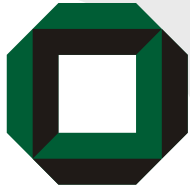
    public void deposit(float amount)
        { ... }
    public void withdraw(float amount) throws OverdrawnException
        { ... }
    public float balance()
        { ... }
}

BankAccount b = new BankAccount( ... );
b.deposit(100);
```

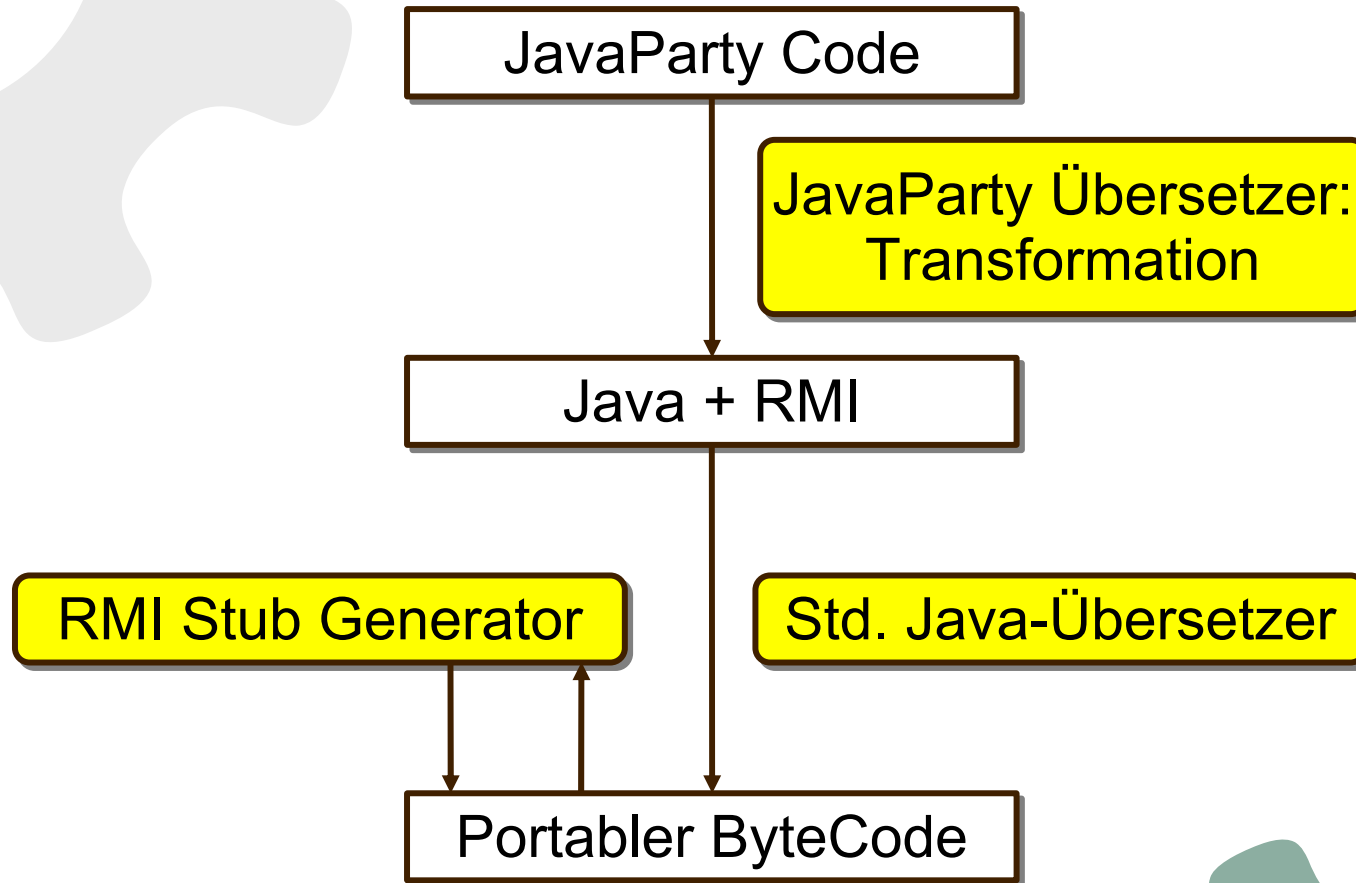


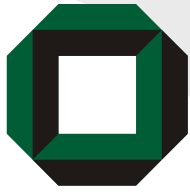
Dasselbe Beispiel mit JavaParty

- Änderungsvolumen:
 - hier: nur 2%
 - dagegen 75% bei RMI-oder bei Socket-Version.
 - Berechnung des Änderungsvolumens hängt natürlich von der Größe des Beispiels ab; die Differenz wird kleiner bei größeren Anwendungen.
- Es wird eine Quellcode-Transformation vorgenommen.
 - Diese Arbeit kann ein mechanischer Übersetzer machen.
- Rückführung auf RMI und pures Java.
 - Das transformierte Programm ist genauso portabel wie die ursprüngliche Anwendung.



Die JavaParty Transformation





JavaParty: Transparente entfernte Objekte

Eine (entfernte) Klasse **R** besteht aus

Instanz -Methoden

```
public int foo(int x) {}  
r.foo(42);
```

RMI (✓)

Instanz -Variablen

```
int a;  
r.a = 13;
```

„generierte get- und set-Methoden“

statischen Methoden

```
static int bar(int x) {}  
R.bar(42)
```

„entfernte Klasse“

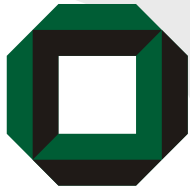
statischen Variablen

```
static int b;  
R.b = 13;
```

Konstruktoren

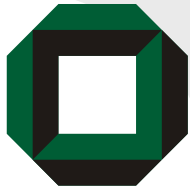
```
public R() {}  
new R()
```

„Konstruktor-Objekte“



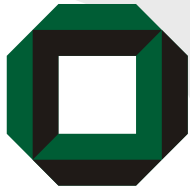
JavaParty: Bestandteile der Transformation

- Instanzanteil...
 - wird transformiert in Instanzobjekt.
 - repräsentiert ein entferntes *Objekt*.
 - enthält dessen
 - Instanzvariablen,
 - nicht-statische (Instanz-) Methoden.
 - ist entfernt zugreifbar (über RMI)
 - über eine generierte entfernte Schnittstelle.
 - Zusätzlich werden set()- und get()- Methoden für die Instanzvariablen generiert.
 - Die RMI-Ausnahmebedingungen werden intern behandelt.



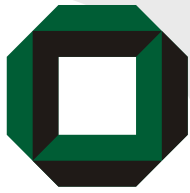
JavaParty: Bestandteile der Transformation

- Statischer Anteil...
 - wird transformiert in Klassenobjekt, das die entfernte *Klasse* repräsentiert.
 - enthält also deren
 - statische Variablen,
 - statische Methoden.
 - existiert genau einmal in der verteilten Umgebung,
 - entspricht der Semantik einer Java-Klasse, ausgedehnt auf die verteilte Umgebung.
 - ist entfernt zugreifbar (über RMI)
 - über eine generierte entfernte Schnittstelle.
 - Zusätzlich werden set()- und get()- Methoden für die statischen Variablen generiert.
 - Die RMI-Ausnahmebedingungen werden intern behandelt.



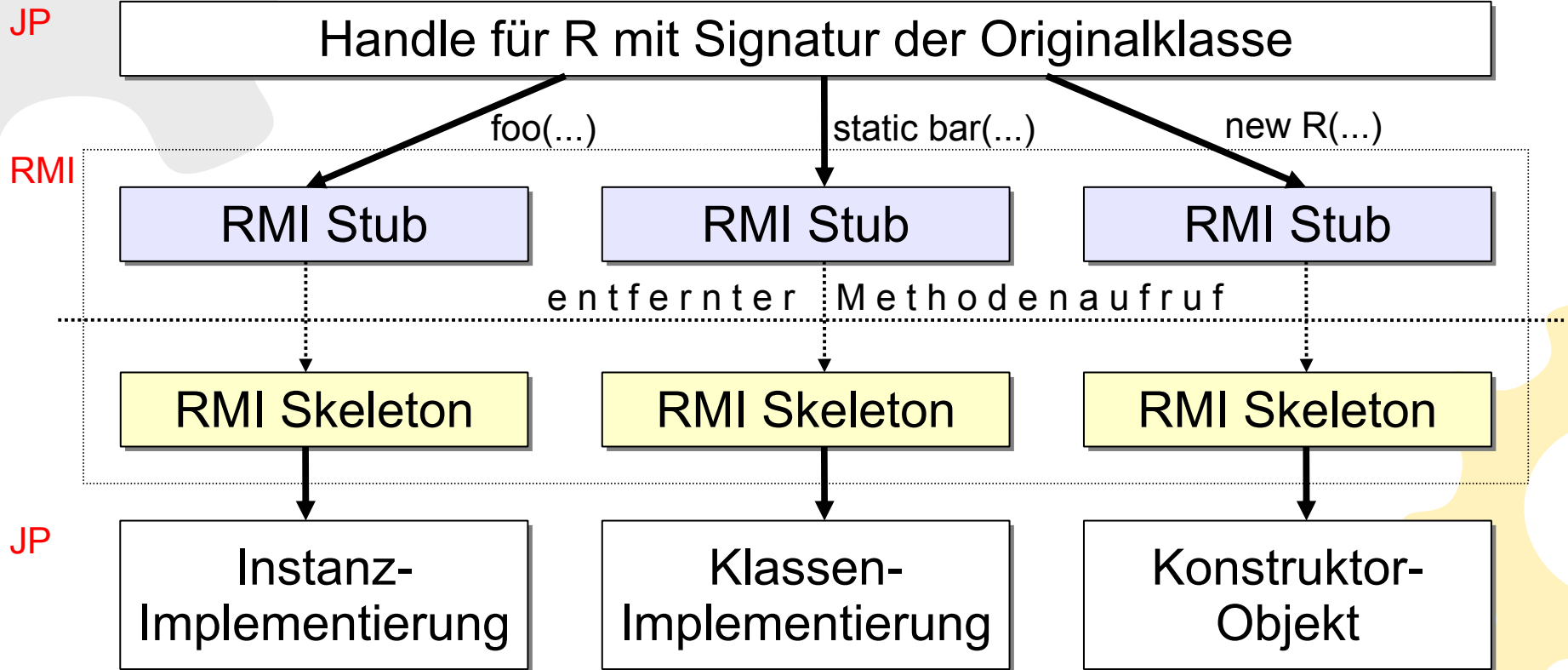
JavaParty: Bestandteile der Transformation

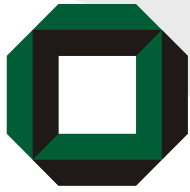
- Konstruktoren...
 - werden transformiert in Konstruktorobjekte.
 - erlauben das entfernte Anlegen von Objekten.
 - Pro Knoten und entfernter Klasse existiert genau ein Konstruktorobjekt.
 - ist entfernt zugreifbar über RMI
 - über eine generierte entfernte Schnittstelle.
 - Pro Konstruktor der Klasse wird eine Fabrikmethode generiert.



Zugriff auf ein entferntes Objekt in JavaParty, Übersicht

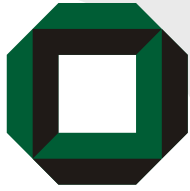
Stellvertreter (JavaParty -Handle): uniformer Zugriff auf entfernte Klasse + Migration





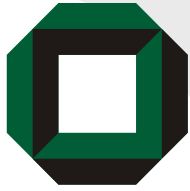
Objektmigration in JavaParty (1)

- Ein Objekt kann nur migriert werden, wenn gerade keine Methoden auf ihm ausgeführt werden.
- Das entfernte Objekt wird serialisiert, zur Ziel-VM geschickt, dort ausgepackt und als neues entferntes Objekt bekannt gemacht. Dieser Vorgang erzeugt eine *tiefe Kopie* des migrierten Objekts.

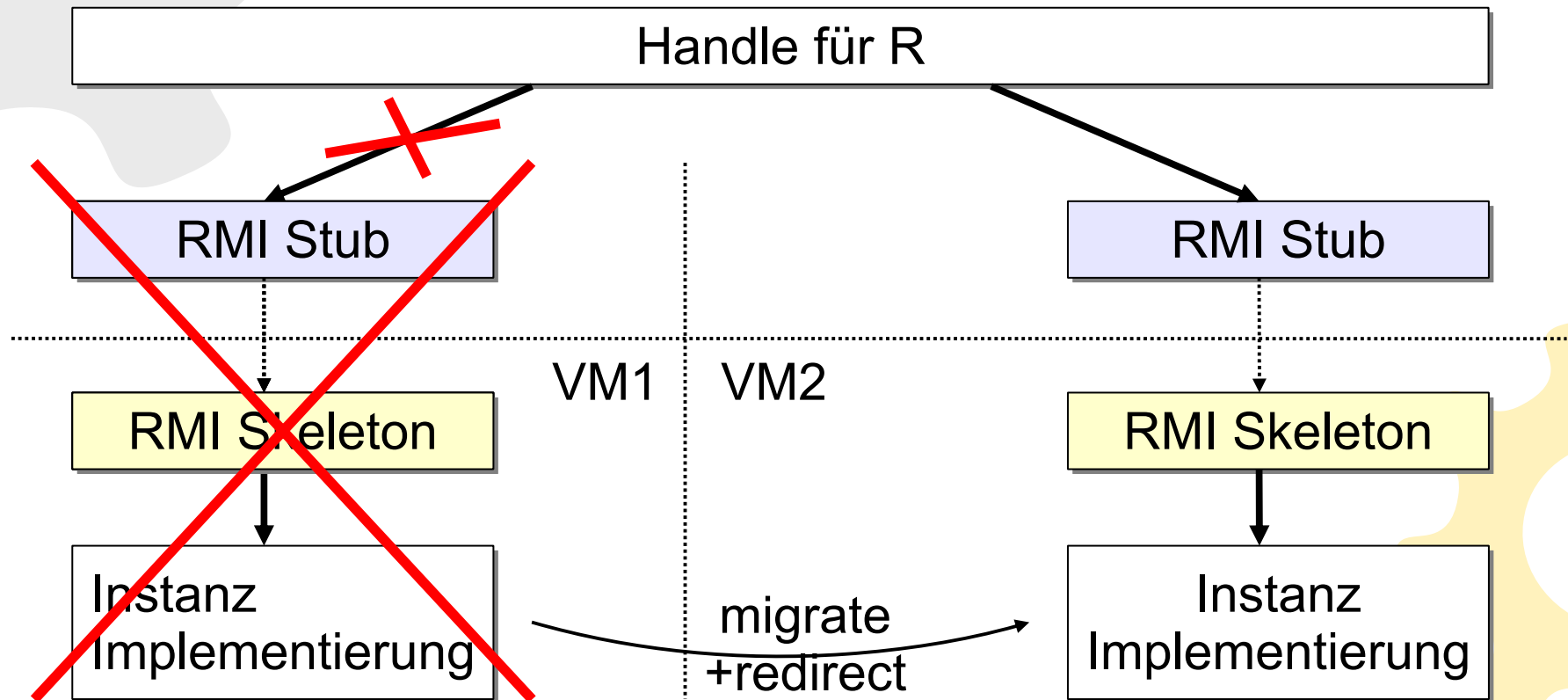


Objektmigration in JavaParty (2)

- Alle Handles (mit Ausnahme des Handles, das die Migration initiierte) zeigen nun auf das falsche (alte) entfernte Objekt. Dieses bleibt deshalb erhalten, führt aber keine Methoden mehr aus, sondern wirft stets Ausnahmen vom Typ `MigratedException`, wenn man versucht es zu benutzen.
- Diese Ausnahmen werden vom Handle abgefangen und nicht der Anwendung präsentiert. Nach und nach erfahren so alle Handles von der Migration und Ort des neuen entfernten Objekts.
- Schließlich kann das alte entfernte Objekt vom verteilten Speicherbereiniger entfernt werden.



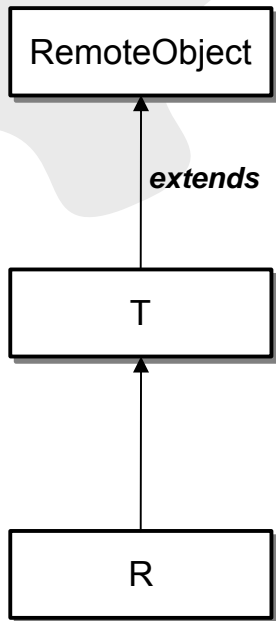
Objektmigration in JavaParty (2)



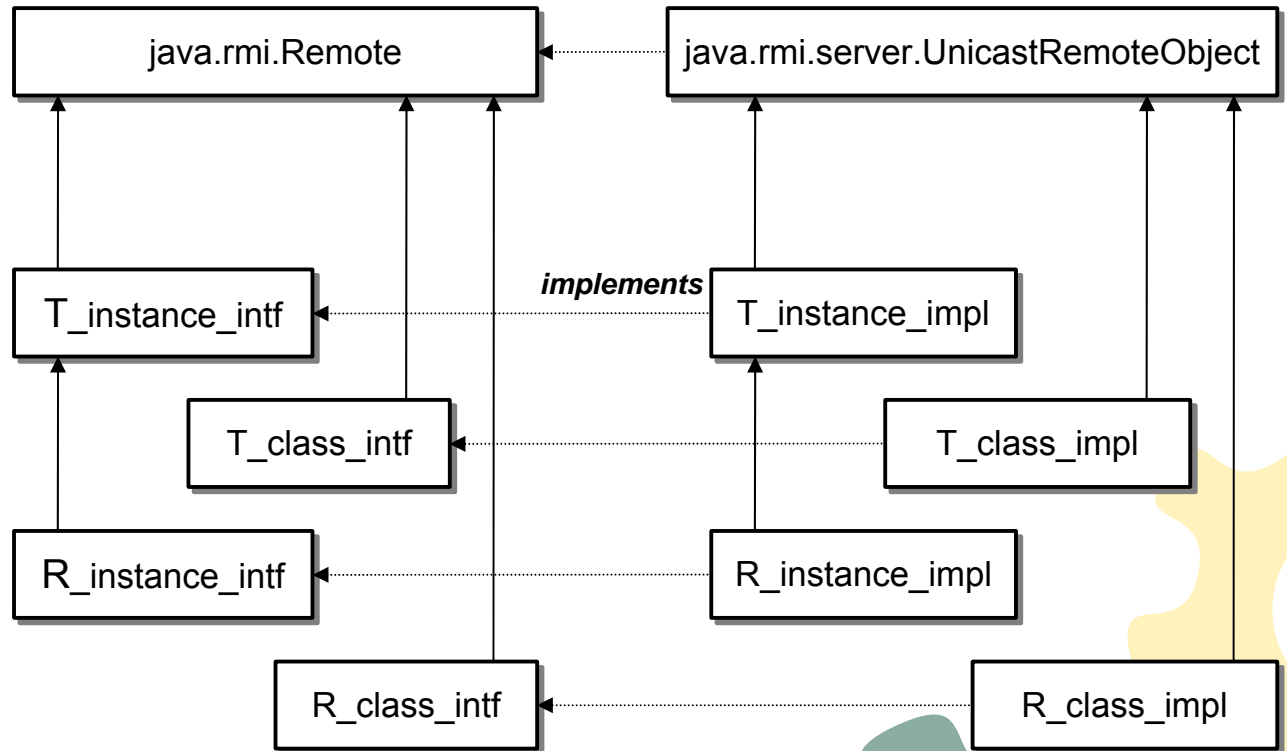


Klassendiagramm vor und nach der JavaParty Transformation

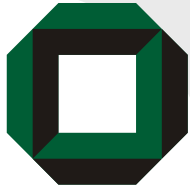
```
remote class T { ... }  
remote class R extends T { ... }
```



JP Fassade

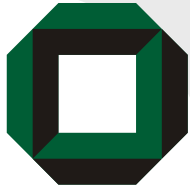


RMI Implementierung



Das Laufzeitsystem

- Die JavaParty-Transformation benutzt
 - Methodenfernaufrufe (RMI)
 - Objektserialisierung (Marshalling)
- **Problem:** Methodenaufrufe mit Objektargumenten über Ethernet zeigen hohe Latenz.
- RMI ist nicht „cluster-tauglich“
 - Fokus von RMI liegt auf WAN-/Internet-Applikationen
 - mit anonymen Klienten,
 - instabilen Netzwerken,
 - unterstützt nur Ethernet.
 - Fokus der Objektserialisierung:
 - persistente Speicherung,
 - Objektversendung zu anonymen Klienten.

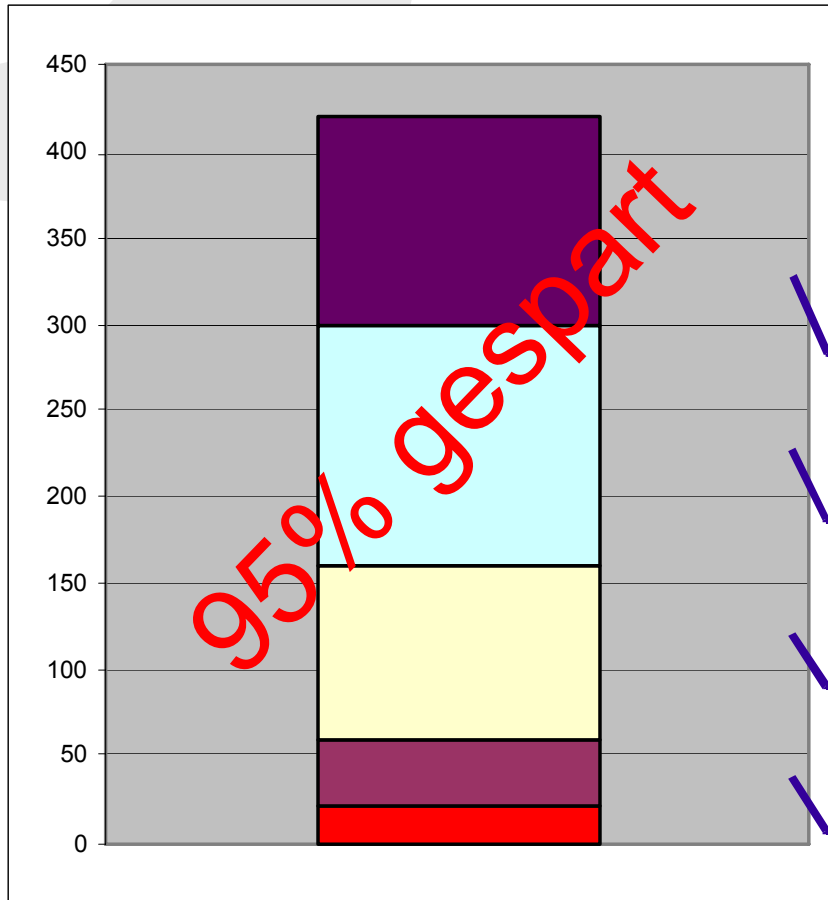


Das Laufzeitsystem

- Javas Objektserialisierung ist...
 - sehr benutzerfreundlich:
 - Der Programmierer muss im wesentlichen nur "**extends Serializable**" hinzufügen.
 - Die Objektserialisierung benutzt zur Laufzeit dynamische Typ-Introspektion.
 - zur persistenten Objektspeicherung geeignet:
 - speichert ausführliche Typinformation.
 - aber nicht für Netzwerkübertragung optimiert:
 - keine optimierter Zugriff auf Sende- und Empfangspuffer,
 - keine Wiederverwendung von übertragener Typinformation in Folgeaufrufen.



Effiziente Objektserialisierung uka.transport



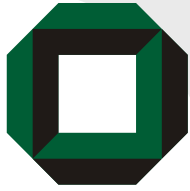
Exemplarisch: Zeit für
einen einfachen entfernten
Methodenaufruf in μs .

Ohne Typintrospektion

Schlanke Typ-Codierung

Direkter Pufferzugriff

Ohne Typ-Wiederholung



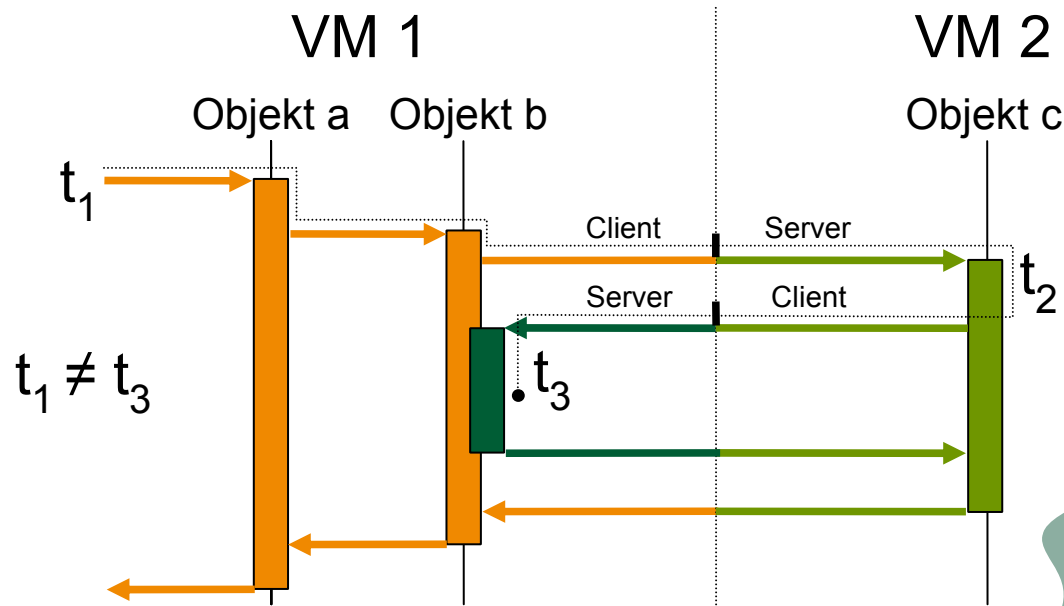
Effizienter entfernter Methodenaufruf: KaRMI

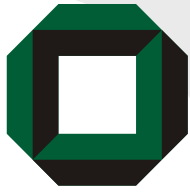
- KaRMI ist ein Ersatz für RMI, der speziell auf den Einsatz in Rechnerbündeln hin ausgelegt ist. Es...
- ... benutzt die effiziente Objektserialisierung `uka.transport`.
 - `uka.transport` bietet außerdem eine steckbare Transportschicht mit Adaptern für
 - Ethernet (TCP)
 - Myrinet (GM/ParaStation)
 - weitere denkbar, z.B. Infiniband, Gigabit Ethernet, ...
- ... implementiert einen schnellen transparenten **lokaler** Methodenaufruf an entfernten Objekten.
 - ...also an solchen, die zwar semantisch entfernt, aber physikalisch lokal sind, d.h. sich in derselben virtuellen Maschine befinden.
- ... integriert die RMI Stellvertreter (Stubs) und die JavaParty Stellvertreter (Handles) in eine Klasse.



Transparente maschinen- überspannende Kontrollfäden (1)

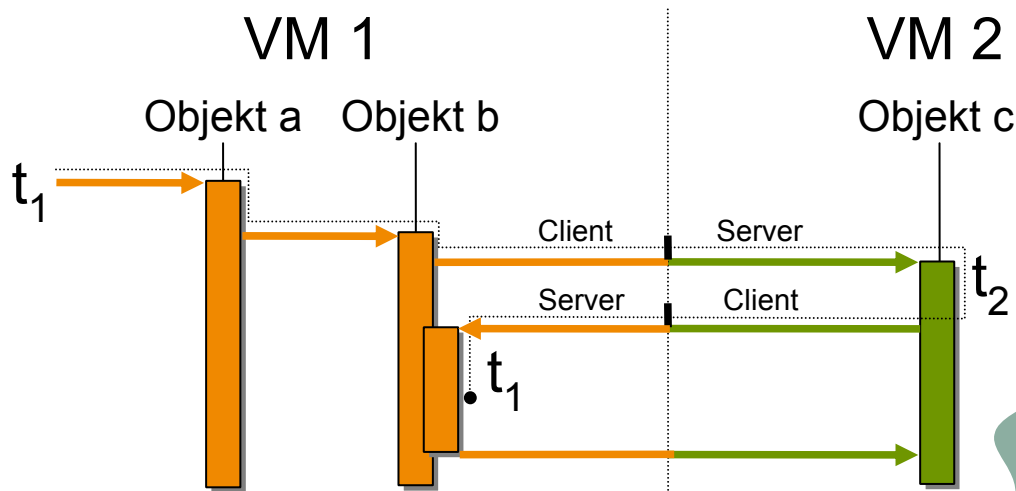
- Während eines Fernaufrufs wechselt der Kontrollfluss auf einen anderen Knoten.
 - Problem: Java Threads sind an ihre virtuelle Maschine gebunden.
 - RMI: Simulation durch mehrere lokale Kontrollfäden

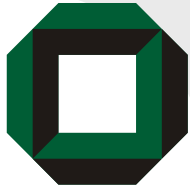




Transparente maschinen- überspannende Kontrollfäden (2)

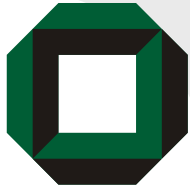
- Problem: t_1 und t_3 haben keine Beziehung
 - Sperren, die t_1 besitzt, können von t_3 nicht betreten werden
- KaRMI-Lösung: Verwende einen eindeutigen Stellvertreter t_1 für den maschinenüberspannenden Kontrollfaden auf VM 1.





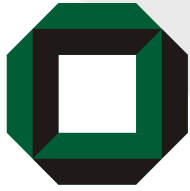
Replizierte Objekte (1)

- **Entfernte Objekte...**
 - sind auf bestimmten Knoten angelegt.
 - Sie können migriert werden, aber das ist eine teure Operation.
 - werden an ihrem aktuellen Aufenthaltsort benutzt.
 - verursachen Fernzugriffe bei ihrer Verwendung.
- **Lokale (Java-) Objekte...**
 - sind auf den Knoten beschränkt, auf dem sie angelegt wurden.
 - sind nur lokal, dafür aber schnell und direkt zugreifbar.
 - werden bei Übergabe in entferntem Methodenaufruf kopiert.
- **Neu: Transparent replizierte Objekte...**
 - sind auf allen Knoten lokal (als Kopie) verfügbar.
 - werden lokal angesprochen.
 - gibt es pro Knoten genau einmal.
 - werden **über Synchronisation** konsistent gehalten.



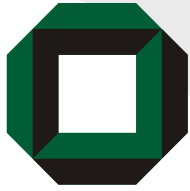
Replizierte Objekte (2)

- **Deklaration** mittels neuem Schlüsselwort **"replicated"**.
 - Es wird immer eine ganze Klasse als repliziert gekennzeichnet.
- **Anlegen** eines replizierten Objektes: **"new P"**
 - liefert "echte" Java-Referenz auf das lokale Replikat
 - kein Proxy-Objekt,
 - kein Umschreiben von Variablenzugriffen in get()/set()-Methoden.
 - Implizit werden Replikate auf allen (anderen) Knoten angelegt.
 - in Fernaufrufen werden allerdings die Referenzen durch die Referenz im Zielknoten ersetzt.
- **Synchronisation**: Neue Schlüsselwörter **"shared"**, **"exclusive"** und **"collective"**.
 - Neue/erweiterte Semantik für die Synchronisation replizierter Objekte.



Replizierte Objekte (3)

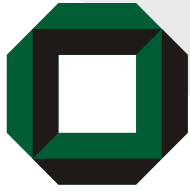
- Warum benötigt man neue/erweiterte Semantik für die Synchronisation?
- (Standard-)Java-Synchronisation ist immer exklusiv. Für Replikation nicht sinnvoll, denn:
 - Replikation wurde ja eingeführt insbesondere für **verteilten Lesezugriff**.
 - Wenn immer nur *ein* Zugriff erlaubt wäre, ginge **mögliche Parallelität** verloren.
 - Wenn vor jedem Zugriff alle *anderen* Replikate gesperrt werden müssten, **wäre der Zugriff nicht mehr lokal**.
- Javas Synchronisationsprimitive unterstützen kein Protokoll, das viele Leser, aber höchstens einen Schreiber zulässt.



Replizierte Objekte (4)

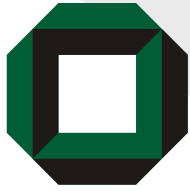
- Es gibt neue Synchronisationsmodi für repl. Objekte. Diese können in zwei Varianten verwendet werden.
- Die beiden Modi dürfen **nicht gemischt** (d.h. gleichzeitig auf dem selben Objekt angewendet) werden!
- Hier wird nur ein grober Überblick gegeben, weitere Details sind wichtig: wie z.B. die Verarbeitung von Signalen in beiden Varianten.

Die Details werden z.B. im Cluster-Praktikum im Sommersemester erklärt.



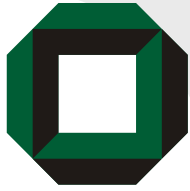
Replizierte Objekte (5)

- Variante 1 mit zwei möglichen Mustern:
 - Paralleler, verteilt lokaler Lesezugriff
 - viele Leser über eine "geteilte" Sperre
 - verwende **shared** **synchronized(p)** überall
 - Exklusiver lokaler Schreibzugriff
 - ein Schreiber über eine "exklusive" Sperre
 - anschließende Zustandsfortschreibung aller Replikate
 - verwende **exclusive** **synchronized(p)** beim Schreiber
 - erwende **shared** **synchronized(p)** sonst



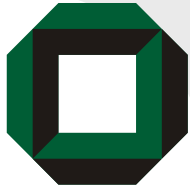
Replizierte Objekte (6)

- Variante 2:
 - Kollektive Modifikation
 - viele Schreiber auf nicht überlappenden Teilen des Replikates
 - anschließende Zustandsfortschreibung aller Replikate
- verwende **collective synchronized(p)** überall



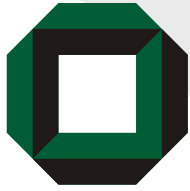
Unterschiede zur Java-Synchronisation

- Bei replizierten Objekten **muss** die Synchronisation **immer an demjenigen Objekt** stattfinden, das **geändert** wird, bzw. dessen **Zustand gelesen** wird.
 - **Zustandsfortschreibung** geschieht vor der Freigabe einer exklusiven Sperre **für das Objekt, an dem synchronisiert wurde.**
 - Als Daumenregel hatten wir das auch schon bei der Java-Synchronisation empfohlen.
- **Exklusive** Synchronisation ist teuer.
 - Egal, ob es Wettbewerb um die Sperre gibt, oder nicht.
 - Exklusive Synchronisation ist mit Kommunikation verbunden!
- **Geteilte** Synchronisation ist (relativ) billig (entspricht Java-Synchronisation).



JavaParty Anwendungen

- Partnerprojekt am IPD Gruppe Lockemann:
 - Paralleles DataMining auf Rechnerbündeln in JavaParty
 - SQL- und DataMining-Anfragen mit visueller Anfrageoberfläche
 - Parallelisierung:
 - Kontrollparallelität: Operatoren
 - Datenparallelität: Tabellenzerteilung (Partitionierung)
- Kooperation mit dem Impact Projekt
 - Fragestellung: Wie kann allgemein ein datenparalleler Algorithmus (SPMD) formuliert werden, der auf einem verteilten Objektgraphen operiert?
 - Website: <http://impact.sourceforge.net>: Explicit Time Integration General Purpose Finite Element Program
 - In neueren Versionen von JavaParty Unterstützung für replizierte Objekte (siehe vorne).



JavaParty: Zusammenfassung

- JavaParty bietet
 - Javas Programmiermodell verallgemeinert für eine potentiell verteilte virtuelle Maschine
 - Beherrschung des verteilter Adressraumes durch transparente entfernte Objekte
 - entfernte maschinenüberspannende Kontrollfäden nutzen Rechenleistung
 - optimiertes Kommunikationssystem durch
 - effiziente Objektserialisierung
 - schnellen entfernten Methodenaufruf
 - schnellen entfernten Methodenaufruf für lokale Objekte
 - Ausnutzung von schnellen Netzwerktechnologien
 - Replikation bietet Ansätze für datenparallele JavaParty-Programme.