Worker-Paradigma

Eine **Menge von gleichartigen Worker-Prozessen**
arbeitet gemeinsam an der Lösung einer Aufgabe.
Jeder bearbeitet einige Teilaufgaben.

Geschwindigkeitsgewinn bei **verteilten Prozessen**.

Anwendung z. B. bei **Lösungsverfahren Branch & Bound**, Divide & Conquer, Backtracking für **kombinatorische Probleme**.

Manager-Prozess
verwaltet die zu lösenden Teilaufgaben und
sammelt die Teillösungen ein.

Worker-Prozess
bearbeitet nacheinander einige Teilaufgaben,
generiert neue Teilaufgaben und
berechnet Teillösungen.

Branch & Bound

Lösungsverfahren Branch & Bound für kombinatorische Probleme (z. B. Travelling Salesman)

Baumstrukturierter Lösungsraum wird nach einer besten Lösung durchsucht.

Allgemeines Schema:

- **Teillösung L erweitern** zu L_1, L_2, ... (z. B. Weg um eine Kante verlängern)
- Ist Teillösung **zulässig**? (z. B. ist der erreichte Knoten neu?)
- Ist L eine **vollständige** Lösung? (z. B. werden alle Knoten erreicht?)
- **MinKosten (L)** = C: jede aus L erweiterte Lösung hat mindestens die Kosten C
  (z. B. Summe der Kantengewichte)
- **Bound**: Kosten der bisher gefundenen besten Lösung

Datenstrukturen: nach MinKosten geordnete Schlange von Teillösungen; Bound-Variable

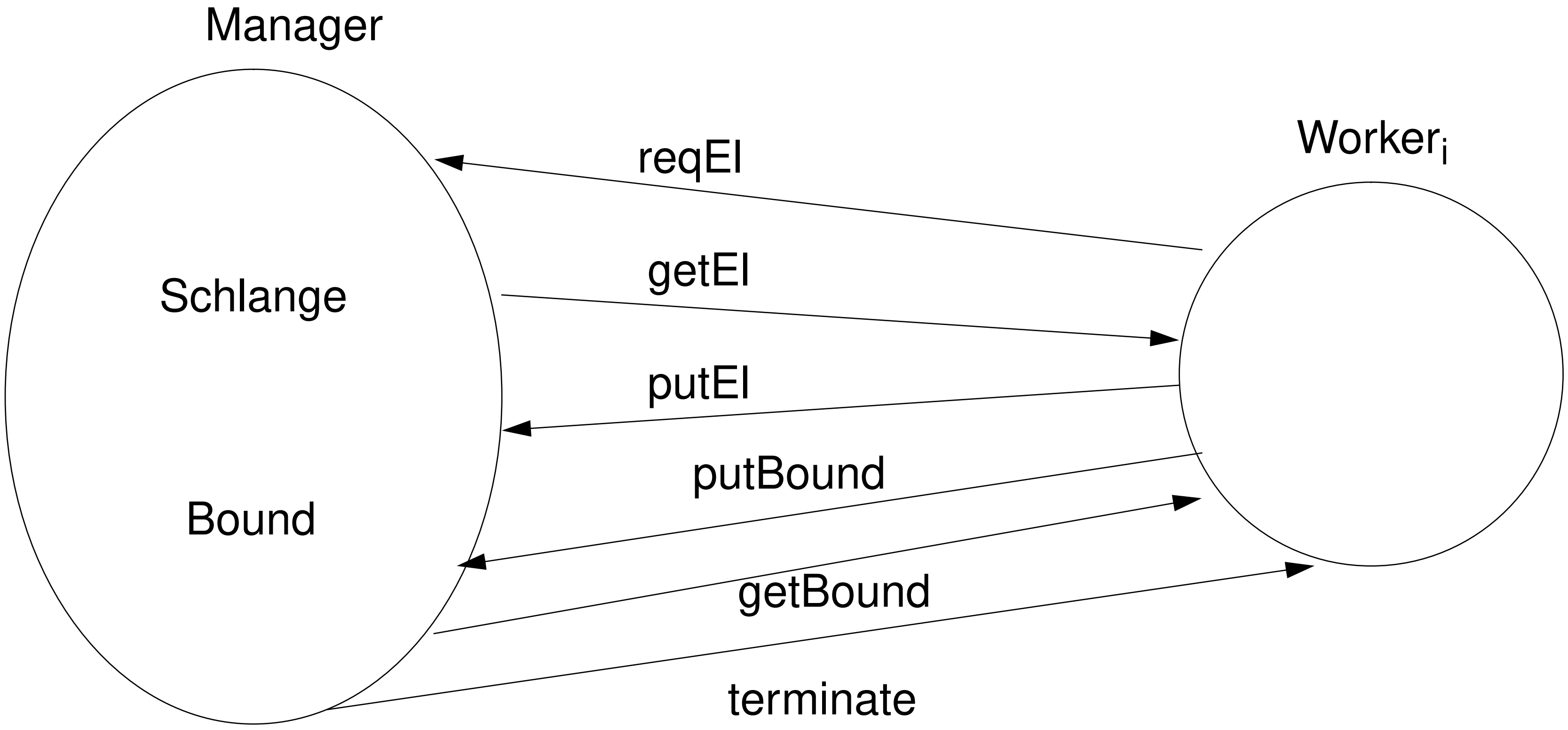
sequentieller Algorithmus:

```
iterieren bis Schlange leer:
    erstes Element erweitern
    neue Elemente prüfen
    ggf. neue Lösung und neuer Bound gefunden
    Schlange aktualisieren
```

Paralleles Branch & Bound (zentral)

Ein **zentraler Manager-Prozess** verwaltet die Schlange und die Bound-Variable.

Ein Worker-Prozess erweitert ein Element, prüft es, berechnet die Kosten, ggf. neuen Bound

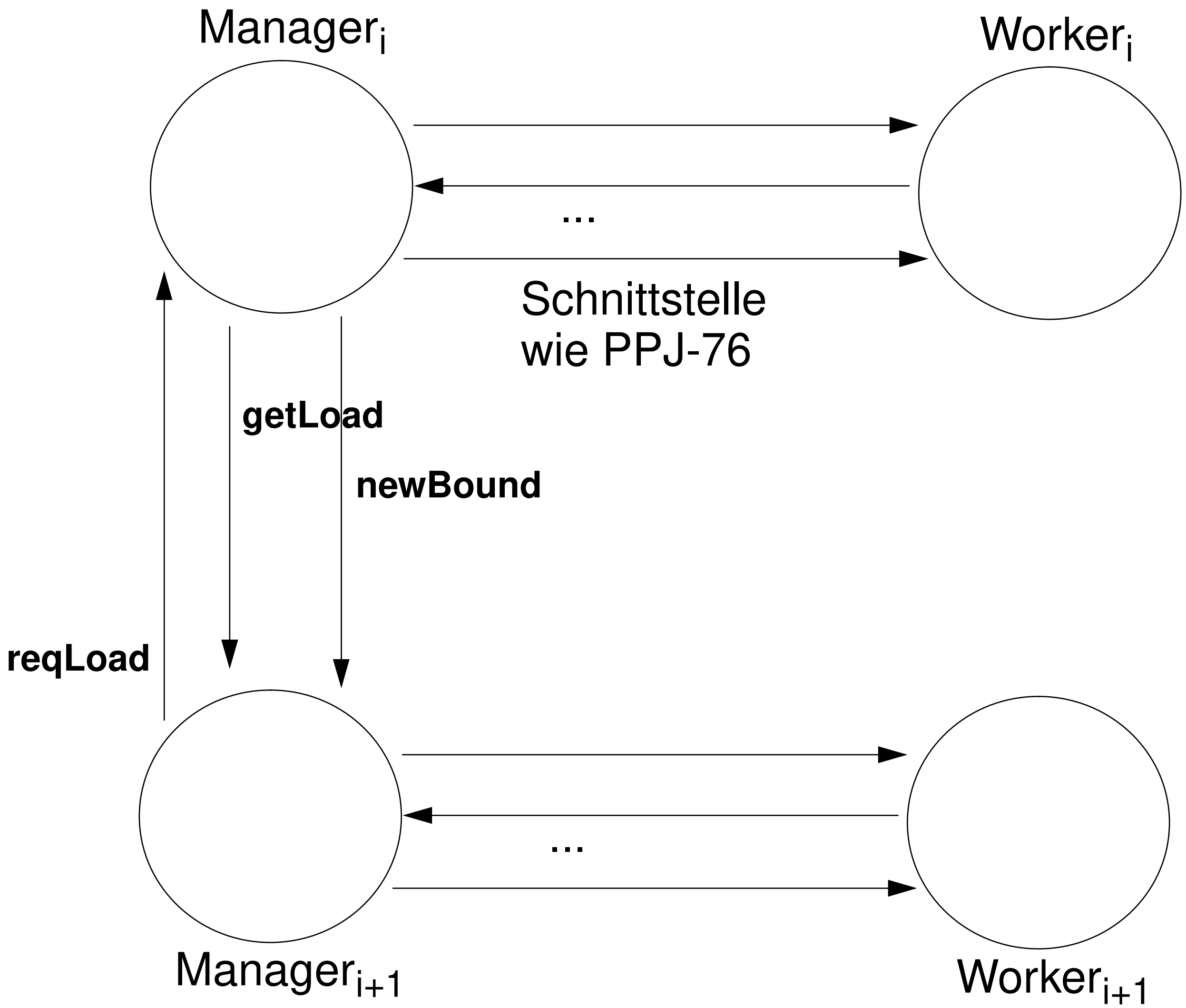


Protokoll: reqEl (getEl [getBound] (putEl | putBound)* reqEl)* terminate

Paralleles Branch & Bound (verteilt)

Verteilte Manager-Prozesse, einer für jeden Worker-Prozess.

Last balancieren mit Nachbarn, z. B. im Ring

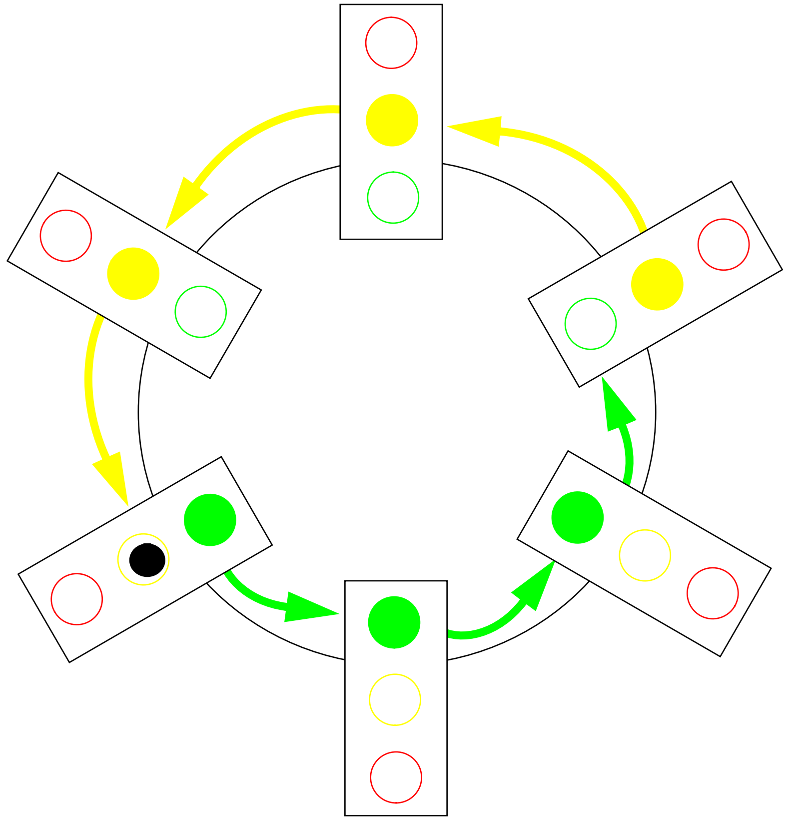


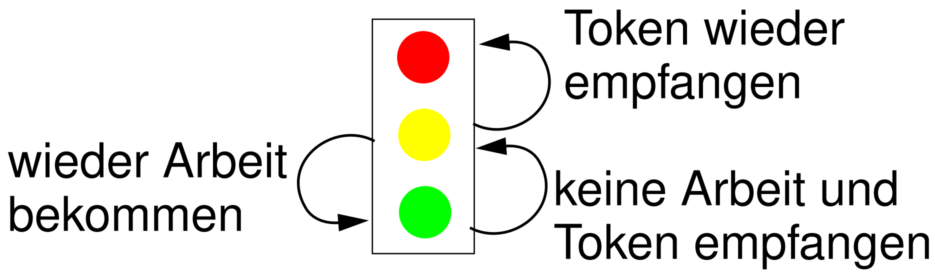
Terminierung?

Terminierung im Ring

Aufgabe: Einen globalen Zustand von Prozessen im unidirektionalen Ring feststellen und allen Prozessen mitteilen, z. B. „alle Prozesse sind inaktiv“.

Idee: Ein Token kreist im Ring und kennzeichnet Prozesse, die den Zustand (inaktiv) eingenommen haben.
In gleicher Richtung kann er wieder zurückgesetzt werden.
Wenn das Token seine eigene Spur erreicht, gilt der Zustand global.



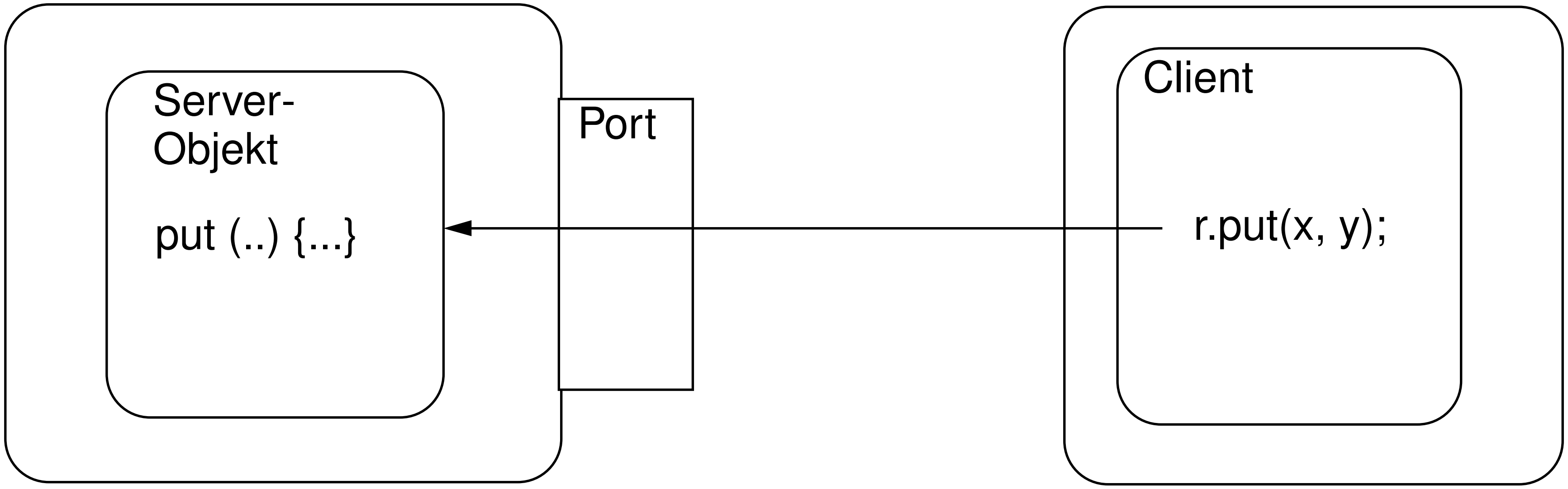


Methodenaufrufe für Objekte auf anderen Rechnern (RMI)

Remote Method Invocation (RMI): Aufruf von Methoden zu Objekten auf anderen Rechnern

In Java durch die Bibliotheken java.rmi verfügbar.

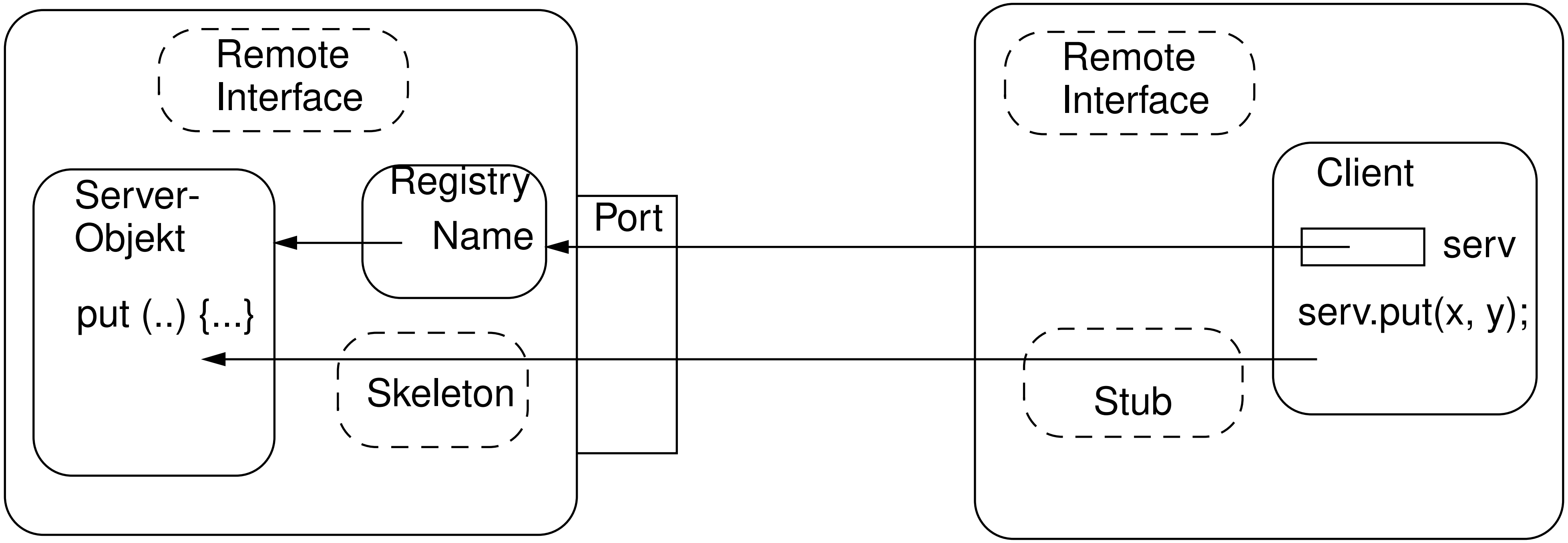
Vergleichbare Techniken: CORBA mit IDL, Microsoft DCOM mit COM



Aufgaben:

- Objekte über Rechnergrenzen hinweg **identifizieren** (Objektverwaltung, Naming Service)
- **Schnittstelle** für Fernzugriffe und ausführbare Repräsentanten der Objekte (Skeleton, Stub)
- **Methodenaufruf**, Parameter und Ergebnis übertragen (Object Serialization)

RMI in Java



Remote Interface: Spezielle Anforderungen an Schnittstellenmethoden

Registry: Prozess auf Systemebene für Rechnerund Port;
leistet Zuordnung von Namen zu Objektreferenzen

Server Skeleton: Repräsentant des Servers für Außenzugriffe auf Server-Objekte,
leistet E/A-Umsetzung Server-seitig,

Client Stub: Repräsentant des Servers, leistet E/A-Umsetzung Client-seitig

RMI Entwicklungsschritte

Am Beispiel: **`Hashtable`** als Server-Objekte verfügbar machen

1. Remote Interface definieren:

```
public interface RemoteMap extends java.rmi.Remote
{ public Object get (Object key) throws RemoteException; ...}
```

2. Adapterklasse zur Anpassung der Server-Klasse an Remote Interface entwickeln:

```
public class RemoteMapAdapter extends UnicastRemoteObject
       implements RemoteMap
{ public RemoteMapAdapter (Hashtable a) { adaptee = a; }
  public Object get (Object key) throws RemoteException
  { return adaptee.get (key); }
  ...
}
```

3. Server-Hauptprogramm erzeugt das Server-Objekt und trägt es in die Registry ein:

```
Hashtable adaptee = new Hashtable();
RemoteMapAdapter adapter = new RemoteMapAdapter (adaptee);
Naming.rebind (registeredObjectName, adapter);
```

4. Skeleton und Stub aus der adaptierten Server-Klasse erzeugen;
   Client-Stub zum Client kopieren:

```
rmic RemoteMapAdapter
```

RMI Entwicklungsschritte (Forts.)

5. Client identifiziert das Server-Objekt auf einem Zielrechner und ruft Methoden auf:
```
Registry remoteRegistry = LocateRegistry.getRegistry (hostName);
RemoteMap serv = (RemoteMap) remoteRegistry.lookup (remObjectName);
v = serv.get (key);
```
6. Auf Server-Rechner eine Registry starten:
```
rmiregistry [port] &
```
Default Port ist 1099
7. Auf Server-Rechner einige Server starten.
8. Auf Client-Rechnern ggf. mehrere Clients starten

Objekte als Parameter von RMI-Aufrufen

Parameter und Ergebnis von RMI-Aufrufen werden über E/A-Ströme übertragen.

Das ist problemlos für Werte von Grundtypen und String.

Bei allgemeinen Objekten wird der Datengehalt der Objektvariablen übertragen und daraus aus der Empfangsseite ein neues Objekt erzeugt.
Die Klasse solcher Objekte muss das Interface Serializable implementieren:

```
import java.io.Serializable;

class SIPair implements java.io.Serializable
{  private String s;
   private int i;

   public SIPair (String a, int b) { s = a; i = b; }
   public String toString () { return s + "-" + i; }
}
```