

ProActive



Martin Steghöfer
10.07.2006

Hauptseminar: Moderne Programmiermethoden

- 1 Grid Computing
- 2 Prinzipien von ProActive
- 3 Actives & Passives
- 4 Migration
- 5 Gruppenkommunikation und -synchronisation
- 6 Konfiguration
- 7 ProActive & Fractal
- 8 Bewertung und Performance

Definition

CoreGRID Network of Excellence:

„Das Grid ist eine voll verteilte, dynamisch rekonfigurierbare, skalierbare und autonome Infrastruktur, die ortsunabhängigen, verlässlichen, sicheren und effizienten Zugang zu einer koordinierten Menge von Diensten anbietet, die Ressourcen (Rechenleistung, Speicher, Geräte, Daten, etc.) virtualisiert um Wissen zu erzeugen.“

Definition

Ian Foster:

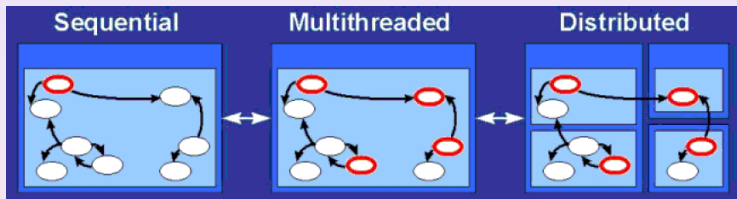
- 1 Koordination von Ressourcen, die keiner zentralisierten Kontrolle unterliegen
- 2 Benutzung von Standard-, offenen, universellen Protokollen und Schnittstellen
- 3 Anbieten von nichttrivialen Diensten

Unterschiedliche Ausprägungen, z.B.

- Data Grid (Verteiltheit von Daten)
- Computing Grid (Verteiltheit von Rechenleistung)

- 70er Jahre: Beginn der Vernetzung von Computern
⇒ Computercluster (HPC) und Ausnutzen von Leerlauf-Rechenzeit (ARPAnet sowie lokale Netze)
- Boom des Internet
⇒ Unmengen an Rechenzeit verfügbar
- Wissenschaftliche Ausnutzung:
Sehr gut parallelisierbare und wenig kommunikationslastige Anwendungen
z.B. Seti@home:
0,5 Mio. Benutzer und 3 Mio. a Rechenzeit (sehr primitives Konzept)
- Peer-to-Peer als Data Grid
- Moderner Ansatz: Remote Method Invokation (Java)
- ProActive: Einfache High-Level-Programmierung

Seamless Computing



- Transparenz von Ort und Aktivität
- Vermeidung von Programmieroverhead
- Wiederverwendung sequenzieller Programme

Prinzipien

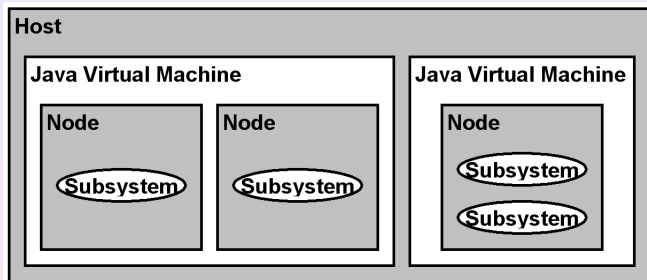
- Keine Modifikation einer Programmiersprache
- Bereitstellung einer Bibliothek
- Aufbau auf RMI
- Erweiterung: Mehr Funktionalität, weniger Overhead
- Alternative Protokolle: JINI, ibis, HTTP,...
- Einteilung in Actives und Passives
- Im Vergleich zu JavaParty:
Lokalität nicht festgelegt, Actives besitzen Thread

Passives

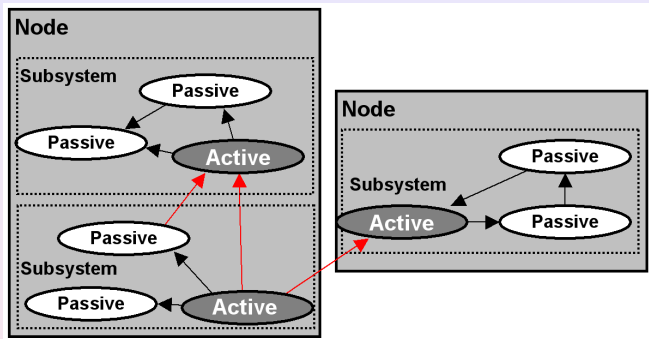
- Nahezu gewöhnliche Java-Objekte
- Teilweise Serialisierbarkeit gefordert
- Zugehörig zu genau einem Active

Actives

- Parallelität und Verteiltheit
- Zuordnung zu einem Thread
- Besitzt Passives (Subsystem)
- Migration



- Host
- Java Virtual Machine
- Node
- Subsystem
- Actives und Passives



- Subsystem: 1 Active + zugehörige Passives
- Verweise auf Actives: beliebig (innerhalb oder zwischen Nodes, JVMs, Hosts)
- Verweise auf Passives: nur im eigenen Subsystem

Weitere Features

- Objekte wie gewöhnliche Java-Objekte ansprechbar (Einkapselung von Actives)
- Einsatzmöglichkeit in inhomogenen Systemen (Java)
- Spezielle Aufrufkonvention:
Reduziert Wartezeiten und schafft implizite Synchronisation
- Explizite Synchronisation
- Gruppenkommunikation

Grundlage von Actives und Passives: Java-Objekte

```
public class A {  
    public A() {  
    }  
  
    public A(int x, String y) {  
    }  
  
    // ...  
}
```

Erzeugung von Passives:

```
A a = new A(26, "astring");
```

Erzeugung von Actives: Instantiation-Based

```
A a;  
Object[] params = new Object[]  
                { new Integer (26), "astring" };  
try {  
    a = (A) ProActive.newActive("example.A", params);  
} catch (ActiveObjectCreationException e) {  
    // Exception handling  
} catch (NodeException ex) {  
    // Exception handling  
}
```

- newActive mit Klassenname und Konstruktorparameter
- Konstruktorparameter als Objekts
- Exceptions: ActiveObjectCreationException und NodeException
- Cast auf A

Erzeugung von Actives: Object-Based

```
A a = new A(26, "astring");  
a = (A) ProActive.turnActive(a);
```

- `turnActive` mit vorhandenem Objekt
- Cast auf `A`
- Vorsicht: Referenz auf altes Objekt verwerfen!

Probleme beim Methodenaufruf

- Passives in fremden Subsystemen nicht aufrufbar
- Aufruf von lokalen Passives: Standard

```
result = aPassive.doSomething(anotherObject);
```
- Aufruf von Actives (möglicherweise entfernt):
 - Keine Referenzen auf Passives
 - Kommunikation nötig
 - Abarbeitung im fremden Thread \Rightarrow Wartezeit

Konsequenzen

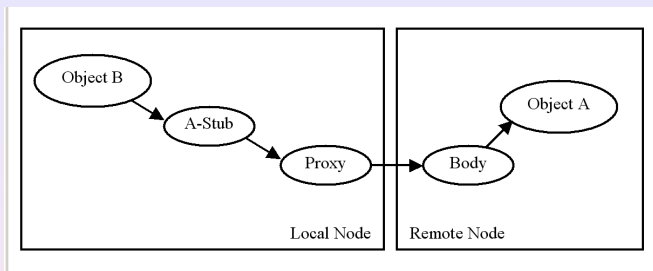
Übergabe von Passives:

- Call-by-Value (Deep-Copy)
- Arbeiten auf Kopie

Wartezeit: Call-by-Necessity

- Aufruf erfolgt asynchron
- Zwischenspeicherung des Aufrufs in Queue (meist FIFO) im aufgerufenen Objekt
- Erzeugung eines Futures (gleicher Typ)
- Eintreffen des Ergebnisses \Rightarrow Füllen des Futures
- Benutzung des Futures vor Eintreffen des Ergebnisses \Rightarrow Blockieren
- Verwendung auch bei lokalem Active

Rückgabetypp	Exception	Asynchron	Future
void	ja	nein	nein
void	nein	ja	nein
Elementar	*	nein	nein
Objekt	ja	nein	nein
Objekt	nein	ja	ja

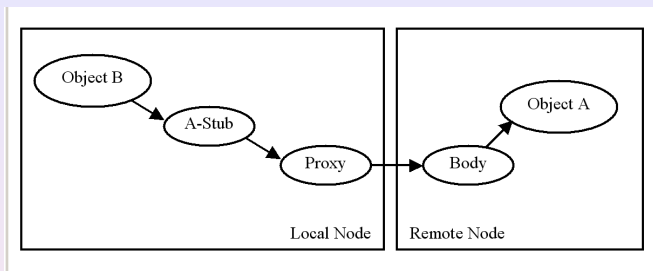


A-Stub:

- Subklasse von A
- Entgegennehmen von Methoden aufrufen
- Erzeugen eines `MethodCall`-Objekts
- Erzeugen von Futures

Proxy:

- Versenden der Anfrage (z.B. über RMI)
- Empfang der Antwort



Body:

- Empfangen der Anfrage
- Zwischenspeichern in Queue
- Scheduling (Activities und Anfragen)
- Versand der Antwort
- Migration

Activities

- Thread für eigene Tätigkeit nutzen, nicht nur Anfragen verarbeiten
- Interface `RunActive` mit `runActivity` (Übergabe des `Body`-Objekts)
- Thread jetzt ständig belegt
⇒ Eigenes Management der Anfragen: `Service`-Objekt

Beispiel-Implementierung

```
public void runActivity(Body body) {  
    Service service = new Service(body);  
  
    while (body.isActive()) {  
  
        while (service.getRequestCount() > 0) {  
            service.serveOldest();  
        }  
  
        continueDoingSomeWork();  
    }  
}
```

Request-Verarbeitung

Service-Objekt:

- Älteste oder neueste Anfrage
- Einzelne oder alle Anfragen
- Blockierend oder nicht-blockierend
- Anfragen-Filter
- Löschen von Anfragen
- Aktuellen Pufferstand abfragen

Eigene Verarbeitungsstrategie

LIFO statt FIFO:

```
public void runActivity(Body body) {  
    Service service = new Service(body);  
    while (body.isActive()) {  
        service.blockingServeYoungest();    }  
    }  
}
```

Initialisierung vor Activity

⇒ Interface `InitActive`

(Methode `initActivity` wird vor Start der Activity aufgerufen)

Aufräumarbeiten nach Activity

⇒ Interface `EndActive`

(Methode `endActivity` wird vor Start der Activity aufgerufen)

Festlegung des Ausführungsorts

Normal:

```
a = (A) ProActive.newActive("example.A", params);  
// bzw.  
a = (A) ProActive.turnActive(a);
```

Node angeben, auf dem das Active erzeugt wird:

```
Node node = NodeFactory.getNode("//minnie/Node1");  
  
a = (A) ProActive.newActive("example.A", params, node);  
// bzw.  
a = (A) ProActive.turnActive(a, node);
```

Änderung des Ausführungsorts zur Laufzeit

```
try {  
    ProActive.migrateTo(destinationNode);  
} catch (MigrationException me) {  
    // Exception handling  
}
```

- Aufruf von `migrateTo`
- Wieder Übergabe eines `Node`-Objekts
- Keine Übergabe des `Actives`!
- Kann nur vom migrierenden `Active` selbst angestoßen werden
- Letzter Befehl in der Methode
- `MigrationException`
- Implementation von `Serializable`

Problem:

Nicht alles kann immer `Serializable` gemacht werden

Beispiel GUI:

- Erwünscht: Bei Migration verschwindet GUI und taucht auf anderem Rechner unverändert auf
- Aber: Zustand von Fenstern im Betriebssystem

Lösung:

- Vor Migration:
 - Abfrage aller relevanten Informationen (z.B. Fenstergröße, Inhalt von Textfeldern, Focus,...)
 - Kontrollierter Abbau der GUI
- Nach Migration:
Aufbau der GUI mit den gespeicherten Informationen

```
public void clean() {  
    // ... save frame-state ...  
  
    // dispose frame  
    frame.dispose();  
    frame = null;  
}
```

```
public void recreate() {  
    // ... restore frame ...  
}
```

Realisierung mit MigrationStrategyManager

```
public void initActivity(Body body) {  
    MigrationManager migrationStrategyManager =  
        new MigrationStrategyManagerImpl((Migratable)body)  
  
    migrationStrategyManager.onDeparture("clean");  
    migrationStrategyManager.onArrival("recreate");  
}
```

- Bisher: Direkte Kommunikation mit einzelner Active (Send und Receive)
- Oft erwünscht: Gleichzeitige Kommunikation mit mehreren Actives (z.B. Broadcast oder Gather)
- Typsicherheit?
- ProActive: Kommunikation mittels Gruppe eines bestimmten Typs

```
Object[][] params = { { ... } , { ... } , ... };  
Node[] nodes = { ... , ... , ... };
```

```
A aGroup = (A) ProActiveGroup.newGroup("A", params,  
                                       nodes);
```

- Erstellung von allen Actives einer Gruppe
- Aufruf ähnlich dem von `newActive`,
nur Arraydimension um 1 höher
- Homogene Gruppe
- Subklasse von `A` \Rightarrow Typsicherheit

```
A a1 = new A();
A a2 = (A) ProActive.newActive('A', paramsA, node);
B b = (B) ProActive.newActive('B', paramsB, node);

A aGroup = (A) ProActiveGroup.newGroup("A");
Group groupObject = ProActiveGroup.getGroup(aGroup);

groupObject.add(a1);
groupObject.add(a2);
groupObject.add(b);
```

- Manipulation von Gruppen mit `Group`-Objekt
- Hinzufügen, Entfernen, Schnitt, Vereinigung,...
- Auswirkungen unmittelbar
- Inhomogene Gruppen

Übergabe von Parametern: „Broadcast“

Rückgabe eines Wertes: „Gather“

- Typsicherheit?
- Rückgabewert: Gruppe aller Rückgabewerte
- Gruppenoperationen oder Extraktion mit `Group`-Objekt

Synchronisation

- Implizite High-Level-Synchronisation
- Für SPMD: Barrier-Synchronisation

Total Barrier

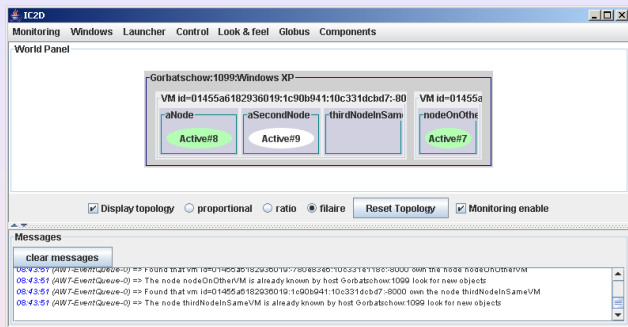
```
ProSPMD.barrier("identification");
```

Neighbor Barrier

```
ProSPMD.barrier("identification", neighborGroup);
```

Neighbor Barrier

```
ProSPMD.barrier({ "method1", "anotherMethod" });
```



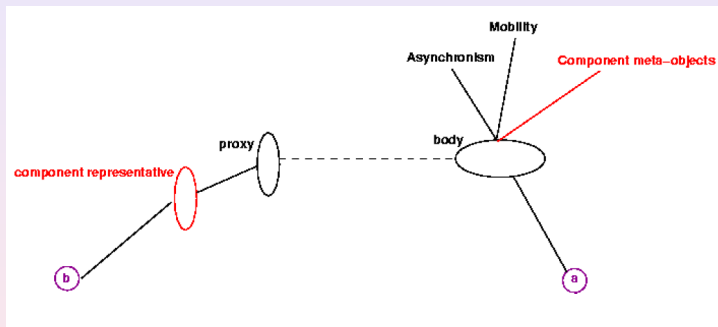
- Zustand von Hosts, Virtual Machines, Nodes und Actives (z.B. bei Actives: Selbst rechnend, Anfrage bearbeitend, auf Resultat wartend, auf Anfrage wartend, migrierend, Anzahl der wartenden Anfragen)
- Topologie von Hosts, Virtual Machines, Nodes und Actives (z.B. Kommunikation)
- Starten von Anwendungen über XML Descriptors

- Erzeugung von JVMs mit Prozessen (auch auf remote Hosts)
- Definition von Prozessen
- Erzeugen von Nodes
- Nodes auf JVMs verteilen

ProActive & Fractal

- ProActive auch Fractal-Implementierung (Level 3.2, keine Erzeugung durch Templates)
- Actives als Komponenten
- Kein Ausbau von Julia möglich
- Erweiterung von purem Fractal: Parallelität und Verteiltheit

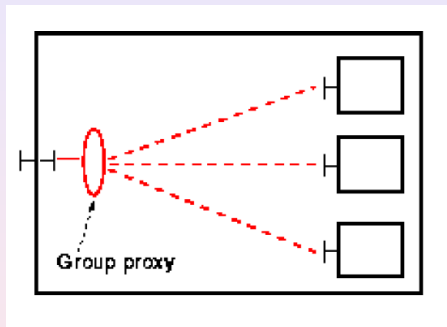
Actives als Komponenten



Modifikation von Actives:

- Stub wird ersetzt durch *component representative*
- Body erhält weitere Meta-Funktionalität

Parallele Komponenten



Parallelität:

- Automatisch durch Wait-by-Necessity
- Fractal-Erweiterung: Parallele Komponente
 - Hierarchische Komponente
 - Parallele Verarbeitung eingehender Anfragen

Typsicherheit beim Erstellen von Actives

```
public class C {  
    public C (int i) {}  
    public C (Integer i) {}  
    public C (int i, String s) {}  
    public C (int i, Vector v) {}  
}
```

```
Object[] params = new Object[] { new Integer (26) };  
a = (A) ProActive.newActive("example.A", params);
```

```
Vector vector = null;  
Object[] params = new Object[]  
                    { new Integer (26), vector };  
a = (A) ProActive.newActive("example.A", params);
```

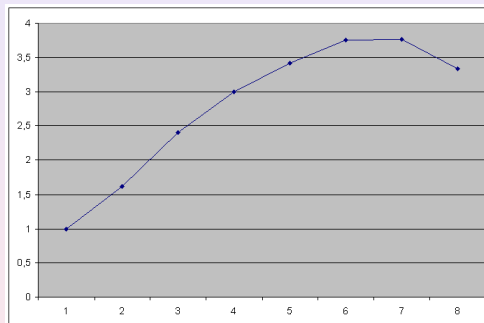
Zusammenfassung

- + Typsicherheit bei Kommunikation
- Keine Typsicherheit bei Aufruf von ProActive-Methoden (Reflection, Casts)
- Einschränkung bei Actives:
 - Keine `final`-Methoden
 - Keine `public`-Attribute
 - Löschen von Referenz auf Originalobjekt bei `turnActive` nötig
- + Gute Unterstützung von Polymorphie (z.B. inhomogene Gruppen, Actives und Passives haben gleichen Typ,...)

Anwendungstest

- Raytracing
- SPMD: Aufteilung des Bildbereichs auf verschiedene Actives
- Mehrere Render-Actives: Bildberechnung (je 1 Host)
- 1 Dispatcher-Active (1 Host):
 - Dynamische Lastverteilung
 - Zusammenfügen des Bilds
 - Zeitmessung
- CIP-Pool der FMI (Pentium 4, 2,8 GHz, 512KB Cache, 1024MB RAM, 100MBit/s-Ethernet)
- Jeweils 32 Durchläufe

Anwendungstest



Engines	1	2	3	4	5	6	7	8
Laufzeit	2,1s	1,3s	0,9s	0,7s	0,6s	0,6s	0,6s	0,6s
Speed-Up	1	1,62	2,40	3,00	3,42	3,75	3,77	3,34

Migration

- Getestet in CIP-Pool
- Migration auf lokalen Nodes
- Migration auf Nodes im LAN

	lokal		remote	
	erster	Aufrufe	erster	weitere
Zeit [ms]	209,5	24,7	213,3	28,9
St.abw. [ms]	72,7	12,0	136,0	15,0

Methodenaufruf

- Test im CIP-Pool
- Synchrone Methodenaufrufe
- Unterschiedliche Datenmengen als Parameter
- Lokal und im LAN

Länge [bytes]	4	40	400	4000	40000	400000
Zeit (lokal) [ms]	5,9	5,4	5,2	6,1	9,6	21,0
St.abw. [ms]	26,9	16,7	15,7	29,5	29,0	146,3
Zeit (remote) [ms]	5,0	3,7	3,7	8,0	7,1	44,3
St.abw. [ms]	21,8	7,1	7,0	11,4	7,2	34,8

- Hohe Startup-Kosten
- Kaum Unterschiede zwischen lokalen und remote Aufrufen

Einsetzbarkeit:

- Keine größeren ProActive-Projekte auffindbar
- Beispiele des ProActive-Teams:
 - Monte Carlo
 - 3D Renderer
 - n-Körper-Problem
 - Elektromagnetische Felder
 - Bildgenerierung mit Fractal
- „2nd GRID Plugtest“
 - n-Damen-Problem
 - FlowShop

Vielen Dank für die
Aufmerksamkeit!

Noch Fragen?