ProActive

Hauptseminar
Moderne Programmiermethoden

Martin Steghöfer

Fakultät für Mathematik und Informatik
Lehrstuhl Prof. Lengauer

10.07.2006

Inhaltsverzeichnis

1 Grid Computing **3**

1.1 Definition 3

1.2 Geschichte 3

2 Prinzipien von ProActive **4**

3 Actives und Passives **6**

3.1 Erzeugung 6

3.2 Methodenaufrufe 7

3.3 Interner Aufbau 8

3.4 Activities 9

4 Migration **10**

4.1 Programmierung 10

4.2 Migration bei Nicht-Serialisierbarkeit 11

5 Gruppenkommunikation und -synchronisation **12**

5.1 Kommunikation 12

5.2 Synchronisation 13

6 Konfiguration **13**

6.1 Interaktive Konfiguration mit IC2D 13

6.2 XML Descriptors 14

7 ProActive und Fractal **14**

8 Bewertung **16**

8.1 Kritik 16

8.2 Performance 17

8.3 Einsetzbarkeit 18

1 Grid Computing

1.1 Definition

In der Literatur existieren zahlreiche Versuche, den Begriff des Grid in einer Definition exakt zu fassen. Als kleinsten gemeinsamen Nenner kann man zusammenfassen, dass es sich um Parallelität und Verteiltheit in einem nicht notwendigerweise homogenen System zur Leistungssteigerung handelt.

Das CoreGRID Network of Excellence [2] stellt sich unter dem Grid „eine voll verteilte, dynamisch rekonfigurierbare, skalierbare und autonome Infrastruktur [vor], 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.“

Ian Foster [1] bietet mit folgender 3-Punkte-Checkliste eine etwas kompaktere Fassung:

- Koordination von Ressourcen, die keiner zentralisierten Kontrolle unterliegen:
  Das Grid integriert Ressourcen aus verschiedenen Kontrolldomänen und regelt dabei Aspekte wie Sicherheit, Regeln, Bezahlung, Mitgliedschaft etc.

- Benutzung von Standard-, offenen, universellen Protokollen und Schnittstellen:
  In einem Grid müssen Protokolle und Schnittstellen für die Authentifizierung, die Suche nach und den Zugriff auf Ressourcen integriert sein. Im Gegensatz zu einem anwendungsspezifischen System müssen diese standardisiert und offen sein.

- Anbieten von nichttrivialen Diensten:
  Das Grid muss in mehreren Hinsichten (z.B. Antwortzeit, Durchsatz, Verfügbarkeit, Sicherheit, etc.) einen qualitativ signifikant höherwertigen Dienst zur Verfügung stellen als die Summe seiner Teile.

So viele Definitionen es für den Begriff Grid gibt, so viele Bezeichnungen gibt es für verschiedene Ausprägungen. Am häufigsten findet man eine Unterscheidung in *Data Grid* (Verteiltheit von Daten) und *Computing Grid* (Verteiltheit von Rechenleistung).

1.2 Geschichte

Bereits in den frühen 70er Jahren, als man damit begann Rechner zu vernetzen, kam die Idee auf, nicht nur Computercluster im High-Performance-Computing-Bereich zum verteilten Rechnen zu benutzen, sondern auch die Leerlauf-Rechenzeit mehrerer Einzelplatzrechner einem gemeinsamen, nicht zeitkritischen Programm zur Verfügung zu stellen. Erste Experimente zu diesem Thema wurden von verschiedenen Instituten und Firmen im ARPAnet, dem Vorläufer des Internet, sowie in lokalen Netzen durchgeführt.

Einen entscheidenden Schub erhielt das Konzept mit dem Boom des Internet, womit plötzlich Unmengen an ungenutzter Rechenleistung auf ihre Nutzung warteten. Daraufhin entwickelten sich einige wenige Projekte, die darauf aus waren die von Nutzern kostenlos zur Verfügung

gestellte ungenutzte Rechenzeit privater PCs für sehr gut parallelisierbare und wenig kommunikationslastige Anwendungen zu nutzen. Das bekannteste und mit einer halben Million Benutzern und 3 Mio. Jahren Gesamt-Rechenzeit größte derartige Projekt ist Seti@home, die Suche nach Hinweisen auf intelligentes außerirdisches Leben im Signal eines Radioteleskops. Allen derartigen Anwendung gemeinsam ist, dass es sich um eine relativ primitive Form des Grid Computing handelt (wenn es sich überhaupt um ein Grid im strengen Sinn handelt). Die Kommunikation beschränkt sich auf das Verteilen von Datenpaketen an die Clients, die damit zwischen Stunden und Wochen rechnen können und das Ergebnis dann an einen zentralen Server zurückliefern.

Parallel dazu entwickelte sich das Peer-to-Peer-Konzept, meist genutzt zur dezentralen Speicherung und dem Austausch von großen Datenmengen. Merkmale heutiger Grid-Systeme (Verteiltheit von Daten, Kommunikation über standardisiertes Protokoll, Ortstransparenz, dynamische Reorganisation,...) sind auch hier schon zu finden.

Modernere Ansätze als reines Parallelrechnen von Computern auf einem vorgegebenen Datenbereich versuchten durch die Unterstützung verteilter Objekte das Potenzial des Grid auch für komplexere Anwendungen nutzbar zu machen, als Beispiel sei die *Remote Method Invocation* (RMI) von Java genannt. Allerdings ist der Programmieroverhead hier noch sehr hoch und damit die Ortstransparenz gering.

An dieser Stelle setzt *ProActive* an, indem dem Programmierer mithilfe einer Bibliothek eine Umgebung geschaffen wird, die Methoden der Kommunikation und Synchronisation zur Verfügung stellt, ohne dass sich der Programmierer um deren Details kümmern muss. So wird die inhomogene parallele verteilte Umgebung einer sequentiellen bzw. lokalen ähnlicher gemacht.

2 Prinzipien von ProActive

Die Intention von ProActive lässt sich am besten unter dem Stichwort *Seamless Computing* zusammenfassen. ProActive stellt den Anspruch möglichst vollkommener Transparenz von Ort und Aktivität. Das heißt es will die Möglichkeit von verteiltem und parallelem Rechnen zur Verfügung stellen, ohne dass der Programmierer die sonst üblichen Konsequenzen (Programmieroverhead, komplizierte Synchronisation und Kommunikation) tragen muss. Der Programmierer soll möglichst seine sequentiellen Java-Programme durch Modifikation an wenigen stellen in ein verteiltes und paralleles Programm umschreiben können.

Um das zu erreichen, verzichtet ProActive komplett auf die Modifikation einer Programmiersprache, sondern stellt lediglich eine Bibliothek zur Verfügung, die für das Management der Grid-Funktionalität zuständig ist. Dabei findet das eingangs erwähnte RMI-Protokoll Verwendung, ProActive nimmt dem Benutzer aber den bei seiner Verwendung nötigen Programmieroverhead ab und geht mit seinen zusätzlichen Features weit über reines RMI hinaus. Dem Programmierer stehen zudem Alternativ-Protokolle zu RMI (wie z.B. JINI und ibis) zur Verfügung.

Kernstück des ProActive-Konzepts ist die Aufteilung von Objekten in *Active Objects* und *Passive Objects*. Während Passives nahezu gewöhnliche Java-Objekte sind (lediglich Seria-

lisierbarkeit wird unter bestimmten Umständen zusätzlich gefordert), wird mit den Actives Parallelität geschaffen. Zu jedem Active gehört genau ein Thread, der an einem beliebigen Ort einerseits eigene Aktivitäten ausführt und andererseits ortsunabhängig durch Methodenaufrufe von anderen Objekten erreicht werden kann. Der Ausführungsort von Actives kann durch *Migration* auch während der Laufzeit gewechselt werden.

Im Gegensatz dazu gehören Passives fest zu einem bestimmten Active, können immer nur am gleichen Ort liegen wie das zugehörige Active und weder von anderen Actives noch von Passives, die zu einem anderen Active gehören, angesprochen werden. Ein Active bildet zusammen mit allen zugehörigen Passives ein *Subsystem*. Einen möglichen Aufbau eines ProActive-Systems zeigt Abbildung 1: Innerhalb eines Subsystems können Referenzen auf beliebige Objekte existieren, während Referenzen zwischen unterschiedlichen Subsystemen (rote Pfeile) immer nur auf Actives verweisen können.

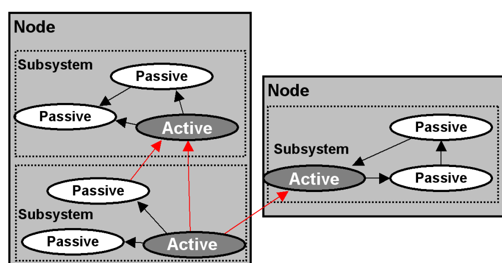

Abbildung 1: Subsysteme innerhalb von Nodes

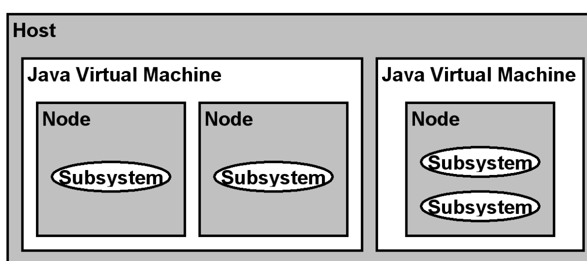

Abbildung 2: Hierarchie der Teilsysteme

Durch diese Nähe der Passives zu gewöhnlichen Java-Objekten im sequentiellen bleibt der sie betreffende Code gegenüber einem Sequentiellen Programm nahezu unverändert. Actives erhalten bei ihrer Erzeugung eine Ummantelung des ehemals sequentiellen Java-Objekts, die die Parallelität und Verteiltheit managt und sich wie das sequentielle Java-Objekt ansprechen lässt. Das schafft die nötige Transparenz und reduziert das explizite Aufrufen der ProActive-Bibliothek, also Veränderungen gegenüber einem sequentiellen Programm, im Wesentlichen auf die Erzeugung der Active Objects.

Die Actives laufen auf sogenannten *Nodes*, die eine Rechneneinheit repräsentieren, von denen aber tatsächlich mehrere auf einem Rechner in einer oder mehreren Java Virtual Machines arbeiten können (siehe Abbildung 2).

Durch die später noch genauer behandelte Aufrufkonvention ist eine implizite High-Level Synchronisation umgesetzt, ebenso existieren aber Methoden zur expliziten Synchronisation.

Portabilität und Einsatzmöglichkeit in inhomogenen Systemen (z.B. eine Mischung aus Linux- und Windowsrechnern) ist durch die Plattformunabhängigkeit von Java bereits gewährleistet.

3 Actives und Passives

3.1 Erzeugung

Die Begriffe Active und Passive betreffen Objekte, nicht Klassen. Daher kann es sowohl aktive als auch passive Instanzen einer Klasse geben. Die Festlegung als Active oder Passive findet meist bei der Erzeugung des Objekts statt. Passives werden weiterhin mit `new` und dem Konstruktor erzeugt, wie gewöhnliche Java-Objekte. Ein Passive von

```
public class A {
  public A() {
  }

  public A(int x, String y) {
  }

  //...
}
```

kann wie gewohnt mit

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

erzeugt werden. Der leere Default-Konstruktor der Klasse `A` ist für alle Klassen, von denen Actives erzeugt werden sollen, für die interne Funktionsweise von ProActive obligatorisch.

Ein Active zu einer Klasse `A` kann mit folgendem Code erzeugt werden:

```
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
}
```

Die eigentliche Erzeugung des Actives geschieht mittels eines Aufrufs der Methode `ProActive.newActive`, dem als erster Parameter der Name der Klasse, von der ein Active Object erzeugt werden soll, übergeben wird. Neben der Erzeugung der notwendigen ProActive-Objekte findet hier der Aufruf eines Konstruktors von `A` statt. Dieses Verfahren, bei dem ein Objekt bereits bei der Erzeugung zum Active wird, nennt sich *Instantiation-Based Creation*.

Da die aufgerufene Methode unabhängig vom gewählten Konstruktor immer die selbe ist, die Parameter aber je nach Konstruktor in Anzahl und Typ variieren können, werden der Methode die Konstruktor-Parameter als `Object`-Array übergeben. Nachdem dieses zwar jeden Typ von Objekt aufnehmen kann, jedoch keine elementaren Datentypen, müssen elementare Datentypen in Form eines Objekts der korrespondierenden Klasse (z.B. Klasse `Integer` statt dem elementaren Typ `int`) übergeben werden. Die Methode `newActive` liefert dann ein Resultat vom Typ `Object` zurück, das zuerst auf den richtigen Typ gecastet werden muss.

Überladen von Konstruktoren ist dabei möglich, die Methode `newActive` wählt auf Basis der Länge des Parameter-Arrays und des Typs der enthaltenen Objekte einen geeigneten Konstruktor aus.

Ebenso wie bei der Instantiierung ist es möglich, bestehende Objekte später zu einem Active Object zu machen. Dieses Vorgehen, *Object-Based Creation*, lässt sich durch Aufruf der ProActive-Methode `turnActive` realisieren:

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

Hier wird das zuvor passive Objekt `a` von ProActive von entsprechenden Objekten eingekapselt, wie das bei der *Instantiation-Based Creation* bereits bei der Erzeugung passiert. Die Methode `turnActive` gibt eine Referenz auf die Einkapselung zurück, über die das Objekt fortan angesprochen werden muss. Da das ursprüngliche Objekt `a` eingekapselt weiterverwendet (und nicht kopiert) wird, darf es nicht mehr direkt, sondern nur noch über die neue Referenz angesprochen werden, um Inkonsistenzen zu vermeiden.

3.2 Methodenaufrufe

Wie bereits im letzten Abschnitt erwähnt, gehören Passives stets zu einem bestimmten Active. Auf ein Passive kann es keine Referenzen in fremden Actives und deren Passives geben, weshalb auch der Aufruf von Methoden fremder Passives unmöglich ist.

Der Aufruf von Methoden eines Passives aus dem eigenen Subsystem erfolgt wie gewohnt: Der Befehl

```
result = aPassive.doSomething(anotherObject);
```

übergibt beispielsweise der Methode `doSomething` eine Referenz auf das Objekt `anotherObject`, das entweder ein Passive des eigenen Subsystems oder ein beliebiges Active sein kann. Der Aufruf findet unmittelbar statt und das Resultat wird sofort in die Variable `result` geschrieben.

Vollkommen anders muss der Aufruf von Methoden fremder Actives erfolgen. Zum einen können hier keine Referenzen auf Passives übergeben werden, da diese nur im eigenen Subsystem gültig sind. Zum anderen ist gegebenenfalls Kommunikation nötig, die von ProActive geregelt werden muss. Zudem steht das Resultat nicht sofort zur Verüfung, nachdem die Kommunikation Zeit benötigt und die Abarbeitung der Anfrage im Thread des fremden Actives - wie eingangs erwähnt ist jedem Active *genau* ein Thread zugeordnet - erfolgen muss, welcher mit anderen Aufgaben beschäftigt sein kann.

Der Tatsache, dass Referenzen auf lokale Passives im aufgerufenen Subsystem nicht verfügbar sein müssen, wird durch Call-by-Value-Parameterübergabe von Passives Rechnung getragen. Beim Methodenaufruf werden die Parameter per deep-copy an den Ziel-Node geschickt und dort auf der Kopie gearbeitet.

Um unnötige Wartezeiten auf das Resultat von Methodenaufrufen zu vermeiden, setzt ProActive hier auf ein Konzept namens *Wait-by-necessity*. Derartige Methodenaufrufe erfolgen stets

asynchron, das heißt die Ausführung des Codes läuft weiter, während parallel die notwendige Kommunikation stattfindet. Da aber bereits beim Methodenaufruf das Resultat in eine Variable gespeichert wird, erzeugt ProActive zunächst einen Platzhalter (*Future*), der über die gleiche Schnittstelle wie das wirkliche Resultatobjekt angesprochen werden kann. Trifft das Resultat ein, erhält das Future eine Referenz darauf und delegiert alles an dieses Objekt weiter, es verhält sich also wie das wirkliche Resultatobjekt. Werden vor Eintreffen des Resultats Methoden des Futures aufgerufen, blockiert es die Ausführung solange, bis das Resultat eintrifft.

Das vorgestellt Konzept scheitert allerdings in bestimmten Fällen. Kann die aufgerufene Methode geprüfte Exceptions werfen, muss abgewartet werden, ob sie tatsächlich welche wirft, da dann der Kontrollfluss anders verläuft. Ebenso kann bei elementaren Rückgabe-Typen der Aufruf nicht asynchron stattfinden, weil der Rückgabewert dann keine Referenz ist, also kein Future zwischengeschaltet werden kann. Ist der Rückgabewert `void`, kann der Aufruf asynchron erfolgen, natürlich wird hierbei jedoch kein Future erzeugt.

Typ d. Rückgabewerts	Gepr. Exception	Asynchrone Kommunikation	Future-Erzeugung
void	ja	nein	nein
void	nein	ja	nein
Elementarer Datentyp	*	nein	nein
Objekt	ja	nein	nein
Objekt	nein	ja	ja

Obwohl Aufrufe von aktiven Objekten im selben Node oder gar im selben Subsystem wesentlich einfacher stattfinden könnten, werden diese aus Gründen der Einheitlichkeit auf die gleiche Weise verarbeitet.

3.3 Interner Aufbau

Ein Active Object zu einer Klasse `A` besteht aus 4 Teilen:

- `A`-Stub
- Proxy
- Body
- Instanz von `A`

Dabei liegen Body und Instanz von `A` auf einem einzelnen Node und verrichten dort die eigentlich Rechenarbeit, Stub und Proxy auf allen Nodes und sorgen dafür, dass das Active `A` von dort aus erreichbar ist.

Der Stub dient dazu, Aufrufe von `A`-Methoden entgegenzunehmen und so auf allen fremden Nodes ein lokales Vorkommen eines `A`-Objekts zu simulieren. Dafür ist der `A`-Stub eine Subklasse von `A`, die alle Methoden von `A` mit Methoden überschreibt, die ein Metaobjekt vom Typ `MethodCall` erzeugen, das den Aufruf repräsentiert und Informationen über die Methode sowie die übergebenen Parameter enthält. Ab diesem Zeitpunkt ist von der Klasse `A` so weit abstrahiert, dass die weitere Verarbeitung generisch erfolgen kann.

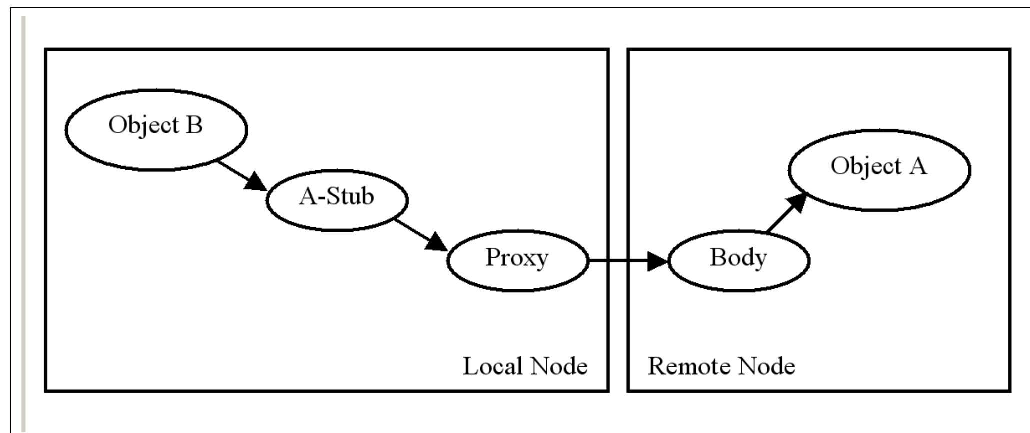

Abbildung 3: Komponenten eines Active Objects einer Klasse A:
Stub, Proxy, Body und A-Instanz

Die Kommunikation über das Netzwerk erfolgt dann zwischen dem Proxy auf der Seite des Callers und dem Body auf der Seite des Callees. Abhängig davon, ob der Aufruf synchron oder asynchron stattfindet, blockiert der Proxy das aufrufende Subsystem entweder oder erzeugt ein Future, das dem Aufrufer dann vom Stub zurückgegeben wird.

Auf der Seite des aufgerufenen Actives sorgt dann der Body dafür, dass alle eingehenden Anfragen in der richtigen Reihenfolge (standardmäßig *First-come-first-served*) abgearbeitet werden, während das Active außerdem seine eigenen Aktivitäten durchführt, und die Resultate an die entsprechende Stelle zurückgeschickt werden.

3.4 Activities

Actives können ihren Thread nicht nur dazu nutzen, auf Anfragen von Actives oder Passives anderer Subsysteme zu reagieren, sondern auch eigene Aktivitäten darin durchführen. Um dies ProActive mitzuteilen, muss es das Interface `RunActive` implementieren, das `runActivity` als einzige Methode enthält. Diese Methode wird nach der Erzeugung von ProActive aufgerufen und das Active kann darin seine Aktivitäten ausführen.

Nachdem der (einzige) Thread des Actives jetzt aber ständig beschäftigt ist, muss sich der Programmierer selbst darum kümmern, zu geeigneten Zeitpunkten die Abarbeitung von Anfragen zuzulassen. Dazu wird der Methode `runActivity` eine Referenz auf den Body des Actives übergeben, mit deren Hilfe eine Instanz der Klasse `Service` erzeugt werden kann, die zahlreiche Methoden zur Abarbeitung zur Verfügung stellt.

Eine typische Implementation von `runActivity` kann folgendermaßen aussehen:

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

```
      continueDoingSomeWork();
    }
  }
```

Das Active in diesem Beispiel führt seine eigene Aktivität, repräsentiert durch `continueDoingSomeWork()`, blockweise aus, solange er von `body.isActive` die Erlaubnis dazu hat. Nach jedem dieser Blöcke wird mit `service.getRequestCount()` abgefragt, wie viele Anfragen vorliegen, und alle Anfragen in FIFO-Strategie abgearbeitet.

Die Klasse `Service` bietet dabei zahlreiche Möglichkeiten die Anfragen in unterschiedlicher Art und Weise abzuarbeiten. Es können einzelne oder alle Anfragen abgearbeitet werden, die ältesten oder die neuesten, blockierend (warten auf das Ankommen einer Anfrage) oder nicht-blockierend (Programmablauf fortsetzen, falls keine Anfrage vorliegt) und Anfragen für das ganze Active oder nur für bestimmte Methoden. Daher ist das Interface `RunActive` auch dazu geeignet, eine eigene Abarbeitungsstrategie zu implementieren, selbst wenn man keine eigene Aktivität durchführen will. Eine LIFO- statt der normalen FIFO-Strategie sieht beispielsweise so aus:

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

Soll das Active vor oder nach der Aktivität Initialisierungen oder Aufräumarbeiten durchführen, muss es lediglich das Interface `InitActive` oder `EndActive` implementieren, das die Methode `initActivity` bzw. `endActivity` enthält, die vor bzw. nach der Aktivität des Actives aufgerufen werden.

4 Migration

4.1 Programmierung

Die Erstellung von Actives nach dem oben angegebenen Schema erfolgt zunächst in einem beim Programmstart erzeugten Standard-Node. Durch Angabe eines Node-Namens als zusätzlichem Parameter im Aufruf von `newActive` oder `turnActive` können sie jedoch auch auf einem bestimmten anderen Node erzeugt werden. Zudem bietet ProActive die Möglichkeit, den Aufenthaltsort von Actives während ihrer Laufzeit durch Migration zu ändern. Das geschieht durch einen Aufruf der Funktion `migrateTo`:

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

Das Codebeispiel bewirkt eine Migration des Actives, aus dem heraus `migrateTo` aufgerufen wurde, zu dem im String `destinationNode` angegebenen Node. Der Aufruf muss zwingend der letzte der Methode sein und muss von dem Objekt aus stattfinden, das migrieren will. Er wirft im Fehlerfall (z.B. bei Angabe eines nicht existierenden Nodes) eine `MigrationException`.

Voraussetzung für die Migration eines Actives ist, dass es das Interface `Serializable` implementiert und wie alle Actives einen leeren Standardkonstruktor besitzt.

4.2 Migration bei Nicht-Serialisierbarkeit

Das Konzept der Migration baut darauf auf, dass das migrierende Active mit allen zugehörigen Passives serialisiert und sein Zustand auf einem anderen Node wiederhergestellt wird. In bestimmten Fällen befindet sich der Zustand von Komponenten jedoch teilweise außerhalb der Reichweite des Programmierers. Bei GUI-Komponenten befindet sich der Zustand von Fenstern beispielsweise im Betriebssystem. Daher gibt es oft Komponenten, die nicht serialisiert werden können.

Dennoch bietet ProActive Unterstützung bei der Migration derartiger Objekte. Die Idee dabei ist, dass die Zustandsinformationen der nicht-serialisierbaren Objekte (bei GUI-Komponenten z.B. die Fenstergröße sowie der Inhalt von Textfeldern) zu Beginn einer Migration von außen so weit abgefragt werden, dass sie nach vollzogener Migration entsprechend wieder aufgebaut werden können.

Dazu wird ein `MigrationStrategyManager` an das Active angehängt, der unter anderem dazu dient, vor und nach der Migration definierbare Methoden des Actives aufzurufen.

```java
public void initActivity(Body body) {
  // add a migration strategy manager on the current active object
  MigrationManager migrationStrategyManager =
                  new MigrationStrategyManagerImpl((Migratable)body);

  // specify what to do when the active object is about to migrate
  // the specified method is then invoked by reflection
  migrationStrategyManager.onDeparture("clean");
  migrationStrategyManager.onArrival("recreate");
}

public void clean() {
  // ... save frame-state ...

  // dispose frame
  frame.dispose();
  frame = null;
}

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

Der `MigrationManager` wird an das eigene Active angehängt, indem ihm bei der Erzeugung als Konstruktor-Argument eine Referenz auf den eigenen Body übergeben wird. Der Name der vor bzw. nach der Migration aufzurufenden Methode wird ihm per `String`-Parameter des Methodenaufrufs `onDeparture` bzw. `onArrival` mitgeteilt. Der Aufruf der übergebenen Methode (hier `clean` zum Abbauen eines `Frame`s bzw. `recreate` zum Wiederaufbauen des `Frame`s) erfolgt dann über Reflection.

5 Gruppenkommunikation und -synchronisation

5.1 Kommunikation

Direkte Kommunikation mit einem Active ist mit dem ProActive-Konzept mittels Methodenaufrufen des Actives möglich. Oftmals ist es aber erwünscht mit vielen Actives einer Klasse `A` (oder Subklassen) gleichzeitig zu kommunizieren („Broadcast“ oder „Gather“). Das funktioniert in ProActive mit Methodenaufrufen einer sogenannten Gruppe, die (ebenso wie `A`-Stubs) eine Subklasse von `A` ist und den Aufruf an alle beteiligten Objekte weiterleitet. Dadurch, dass die Gruppe nur Objekte vom Typ `A` und Subtypen enthalten kann und selbst vom Typ `A` ist, bleibt die Typsicherheit (zumindest nach dem Anlegen der Gruppe) hier erhalten.

Die Erzeugung einer neuen Gruppe kann mit der ProActive-Methode `newGroup` erfolgen:

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

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

Der Aufruf von `newGroup` ist dem von `newActive` sehr ähnlich und erfüllt im Wesentlichen die gleiche Aufgabe. Allerdings wird hier nicht nur ein einzelnes Active erzeugt, sondern eine bestimmte von der Länge der übergebenen Arrays abhängige Anzahl. Die Dimension der Arrays `params` und `nodes` ist entsprechend um 1 höher als beim Aufruf von `newActive`. Die hier erzeugten Actives sind sogleich Mitglieder der neu erzeugten Gruppe und können über das zurückgegebene `A`-Objekt gemeinsam angesprochen werden.

Mehr Kontrolle über die Gruppenverwaltung bieten die `Group`-Objekte. Ein solches erhält man durch Aufruf von `getGroup` mit der zu verwaltenden Gruppe als Argument. Damit lassen sich sämtliche Mengenoperationen (Vereinigung, Schnitt und Differenz) zwischen Gruppen durchführen und einzelne Actives hinzufügen und entfernen:

```
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);
```

Durch den Aufruf von `newGroup` mit nur einem Parameter wird eine leere `A`-Gruppe erzeugt, deren `Group`-Objekt mit `getGroup` erzeugt wird. Mit dessen Hilfe werden ein passives und ein aktives `A`-Objekt sowie ein aktives `B`-Objekt (`B` ist eine Subklasse von `A`) der `A`-Gruppe hinzugefügt. Die Operationen haben unmittelbare Auswirkungen auf die `A`-Gruppe `aGroup`, ohne dass zu dem `Group`-Objekt wieder eine Gruppe erzeugt werden muss, was jedoch bei bedarf mittels

```
aGroup = (A) groupObject.getGroupByType();
```

möglich ist.

Bei Methoden mit Rückgabewert wird natürlich von jedem Objekt der Gruppe ein Wert zurückgegeben. Daher wird bei Gruppenoperationen statt eines einzelnen Rückgabewerts vom Typ `V` eine `V`-Gruppe zurückgegeben. Alle Features betreffend asynchronem Aufruf und Erzeugung von Futures bleiben bei Gruppenkommunikation erhalten.

5.2 Synchronisation

Neben der impliziten Synchronisation, die durch die Art der Abarbeitung von Methodenaufrufen gegeben ist, bietet ProActive speziell für SPMD-Programmierung die Möglichkeit der *Barrier-Synchronisation*. Es stehen 3 Arten von Barriers zur Verfügung.

Die *Total Barrier* blockiert nach dem Aufruf der Barrier-Methode das Active so lange, bis alle beteiligten Actives die Barrier überschritten haben. Im Unterschied zu bekannten Systemen wie MPI wird das Active jedoch nicht sofort blockiert, sondern erst nachdem die aktuelle Anfrage bearbeitet wurde. Zur eindeutigen Unterscheidung verschiedener Barriers in einem Programm wird ein Identifikator-String übergeben:

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

Die *Neighbor Barrier* schrenkt die Blockade auf das Warten auf Actives einer bestimmten Gruppe ein, die im Methodenaufruf angegeben wird:

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

Bei der *Method Barrier* wird die Abarbeitung anderer Anfragen so lange gestoppt, bis alle angegebenen Methoden aufgerufen wurden:

```
ProSPMD.barrier({'foo', 'bar', 'gee'});
```

Die Reihenfolge, in der die Methoden angegeben bzw. aufgerufen werden, spielt dabei keine Rolle.

6 Konfiguration

6.1 Interaktive Konfiguration mit IC2D

Zur Überwachung des Grids steht dem Benutzer die Software IC2D zur Verfügung. Hierin werden Topologie und Zustand von Hosts, Virtual Machines, Nodes und Actives angezeigt, bei

Actives die Anzahl der wartenden Anfragen sowie der Zustand des Objekts: Selbst rechnend, eine Anfrage bearbeitend, auf ein Resultat wartend, auf eine Anfrage wartend oder migrierend.

Einfache Aktionen wie das Starten einer neuen Anwendung über einen XML Descriptor (siehe nächster Abschnitt) oder die Migration von Actives lassen sich direkt per Mausklick vornehmen.

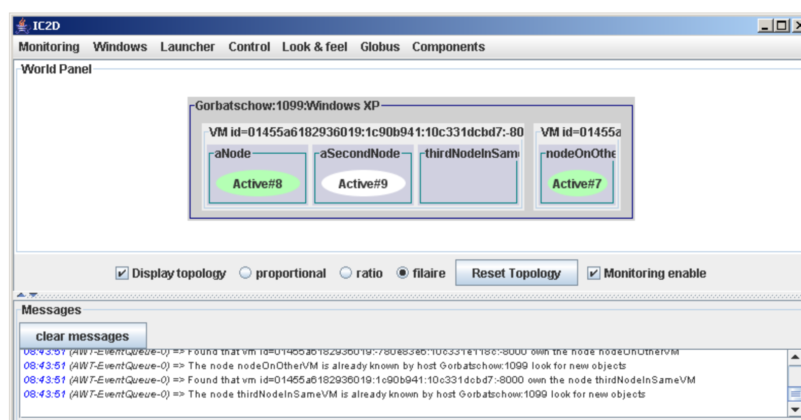

Abbildung 4: Monitoring mit IC2D
Auf einem Host „Gorbatschow“ laufen 2 JVMs mit insgesamt 4 Nodes, auf die 3 Actives verteilt sind. Dabei verarbeitet „Active#9“ gerade einen Request (weiße Färbung), „Active#7“ und „Active#8“ gehen ihren eigenen Aktivitäten nach (grüne Färbung).

6.2 XML Descriptors

Neben dem Erzeugen der Grid-Struktur mittels Java-Code bietet ProActive die Möglichkeit, das Grid per XML Descriptor zu konfigurieren und automatisch hochfahren zu lassen.

In einem derartigen Descriptor können Java Virtual Machines gestartet, Nodes erzeugt und auf die JVMs verteilt werden, Actives erzeugt und auf die Nodes verteilt und die verwendeten Protokolle festgelegt werden. Mithilfe von SSH oder ähnlichem kann das nicht nur lokal, sondern auch auf entfernten Rechnern geschehen.

7 ProActive und Fractal

ProActive ist nicht nur eine Bibliothek zur Unterstützung von Grid-Features in Java-Anwendungen, sondern insbesondere eine Implementierung des *Fractal*-Komponentenmodells (siehe [6]). Dabei sind Parallelität und Verteiltheit von Komponenten die Eigenschaften, mit denen sich ProActive von anderen Fractal-Implementierungen abhebt.

Um Verteiltheit zu erreichen, sind die Fractal-Komponenten in ProActive als Active Objects implementiert. Da Actives aber, wie in Abschnitt 3.3 aufgezeigt, aus mehreren Java-Objekten bestehen, wobei Benutzer stets nur Referenzen auf den Stub halten, kann ProActive keine

Erweiterung einer bestehenden Fractal-Implementierung (wie z.B. Julia) sein, da diese dann nur auf den Stub um Fractal-Funktionalität erweitern würden.

Stattdessen wurde die Fractal-Funktionalität in die bei Active Objects ohnehin vorhandene Meta-Ebene integriert. Der Stub weicht hierbei einem *Component Representative*, der das Active nach außen hin als Fractal-Komponente darstellt (notwendige Interfaces implementiert etc.), der Proxy ist generisch genug um weiterhin der Weiterleitung von Anfragen dienen zu können und der Body wird um die Implementierung der Fractal-Funktionalität erweitert, die der Stub nach außen zur Verfügung stellt. Dank diesem Aufbau unterstützen die Fractal-Komponenten in ProActive weiterhin alle Features von normalen Actives, also insbesondere Migration und asynchrone Methodenaufrufe.

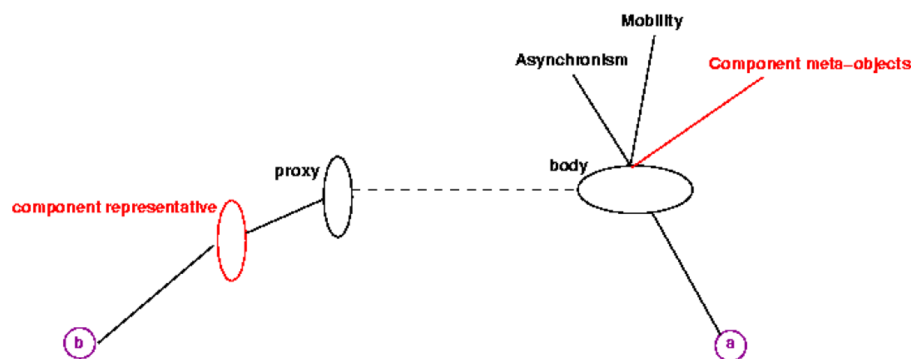

Abbildung 5: Aufbau von Komponenten in ProActive

Parallelität wird mit dieser Technik allerdings kaum erzielt. Zwar besitzen nun durch die Implementierung als Actives alle Komponenten ihren eigenen Thread, allerdings sieht Fractal keine Activities vor, weshalb alle Komponenten nur aktiv werden, wenn sie Requests erhalten. Die Parallelität, die dadurch entsteht, dass die aufrufende Komponente dank des Wait-by-Necessity-Konzepts kurzfristig parallel weiterarbeiten kann, ist relativ beschränkt. Daher sieht ProActive als Erweiterung von Fractal *Parallele Komponenten* vor.

Parallele Komponenten in ProActive sind aufgebaut wie normale hierarchische Komponenten in Fractal, wobei die Teilkomponenten allerdings externe Anfragen, die von einem *Group Proxy* verteilt werden, parallel verarbeiten.

Die Fractal Provider-Klasse heißt in ProActive *Fractive* und fungiert als Bootstrap-Komponente, als generische Factory zur Instantiierung von Komponenten und als Anbieter von einigen statischen Hilfsmethoden zu Fractal.

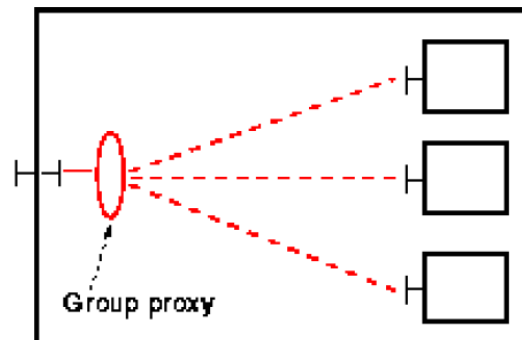

Abbildung 6: Parallele Komponente

8 Bewertung

8.1 Kritik

ProActive bietet dem Programmierer eine Bibliothek reichen Umfangs, die es einfach macht, verteilte und parallele Programme aus sequentiellen zu erzeugen oder neu zu entwerfen.

Konzepte der Objektorientierung wie z.B. Polymorphie werden dabei gut unterstützt. Nachdem der `A`-Stub eine Subklasse von `A` ist, können beispielsweise `A`-Actives vollkommen polymorph zu `A`-Passives verwendet werden. Ebenso ist bei der Gruppenkommunikation möglich, Objekte verschiedener Klassen in einer Gruppe vom Typ einer gemeinsamen Oberklasse zu vereinigen.

Allerdings offenbaren sich bei genauerem Hinsehen auch einige Fallstricke, die der Programmierer kennen und umgehen muss. Zum einen ist die Typsicherheit durch zahlreiche Casts eingeschränkt. Die Funktionen der Klasse `ProActive` arbeiten zumeist mit Reflection, womit sie viele Casts erfordern und viel Raum für Laufzeitfehler bieten.

Die Übergabe von Konstruktorparametern als `Object`-Array macht zudem manches Overloading zweideutig. Dadurch, dass auch elementare Datentypen hier auch über Objekte der korrespondierenden Klasse (z.B. `Integer` statt `int`) übergeben werden, geht die Unterscheidung zwischen diesen beiden Typen verloren. Wird für einen Parameter `null` übergeben, kann die `ProActive`-Klasse hier ebenfalls keinen Typ mehr feststellen. Bei folgender Klasse besteht also eine Zweideutigkeit zwischen Konstruktor 1 und 2, falls als Parameter ein `Integer`-Objekt übergeben wird, und zwischen Konstruktor 3 und 4, falls als zweiter Parameter `null` übergeben wird:

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

Auch die Art und Weise, wie die Kommunikation mit Actives geregelt ist, fordert Fehler heraus. Die Einfachheit der Kommunikation mit entfernten Actives wird dadurch erreicht, dass in jedem Subsystem ein `A`-Stub-Objekt vorhanden ist, in dem jede Methode von `A` durch eine Kommunikationsroutine ersetzt wurde. Das funktioniert aber einerseits nicht bei Methoden, die als `final` deklariert wurden und erreicht andererseits nur Methoden und keine Objektvariablen. Zugriffe auf `final`-Methoden oder Objektvariablen beziehen sich also nur auf das Stub-Objekt und nicht, wie man vermuten könnte, die Methoden und Variablen des entfernten Actives. Derartige Zugriffe können daher unvorhersehbare Konsequenzen haben.

8.2 Performance

Alle folgenden Benchmarks wurden im CIP-Pool der FMI (Pentium 4, 2,8 GHz, 512KB Cache, 1024MB RAM, 100MBit/s-Ethernet) durchgeführt.

Eine Benchmark-Anwendung, die Teil ProActive-Pakets ist, misst die Zeit, die benötigt wird, um ein bestimmtes Bild mittels Raytracing zu rendern. Dabei können mehrere Render-Engines als separate Actives existieren und jeweils einen Teil des Bilds rendern, der dann zu einem Gesamtbild zusammengesetzt wird. Die Aufgabenverteilung sowie das Zusammenfügen des Bildes wird von einem separaten Active, dem Dispatcher, vorgenommen.

Im Test stand jedem Active (dem Dispatcher sowie einer bestimmten jeweils angegebenen Anzahl von Render-Engines) die volle Rechenleistung eines Rechners zur Verfügung.

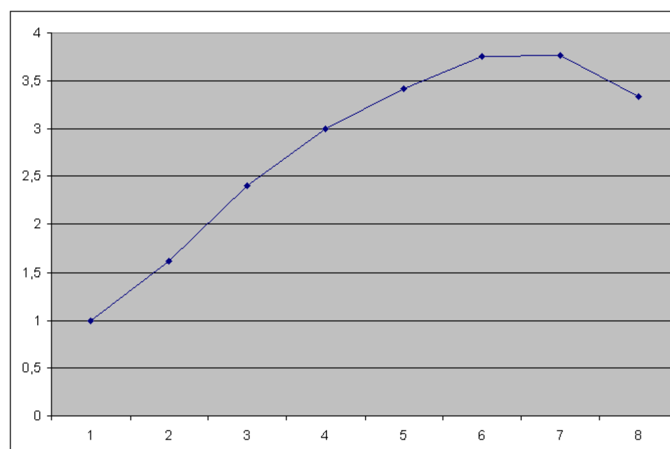

Abbildung 7: Speedup beim Rendering-Benchmark
Speedup gegenüber der sequentiellen Variante in Abhängigkeit der Anzahl der verwendeten Engines

Verwendete Engines	1	2	3	4	5	6	7	8
Laufzeit	2,101s	1,300s	0,876s	0,701s	0,614s	0,560s	0,558s	0,630s
Standardabweichung	0,276s	0,228s	0,115s	0,159s	0,079s	0,095s	0,096s	0,116s
Speed-Up	1	1,62	2,40	3,00	3,42	3,75	3,77	3,34

Weitere Benchmarks hatten zum Ziel, die Zeit, die für bestimmte ProActive-Aktivitäten benötigt wird, zu messen, im Speziellen für Migration und Remote-Methodenaufruf. Die Tests

wurden wieder auf den genannten Rechnern durchgeführt. Es ergaben sich folgende Ergebnisse:

	Migration lokal		Migration remote	
	erster Aufruf	weitere Aufrufe	erster Aufruf	weitere Aufrufe
Zeit [ms]	209,5	24,7	213,3	28,9
Standardabweichung [ms]	72,7	12,0	136,0	15,0

Wie an den Messwerten für die Migration zu erkennen ist, wird bei der ersten Migration zu einem Node die Verbindung aufgebaut, was erheblich länger dauert als die eigentliche Migration eines kleinen Objekts. Die unterschiede zwischen der Migration zwischen Nodes auf einem gemeinsamen Rechner (aber in unterschiedlichen JVMs) und der Migration zwischen unterschiedlichen Rechnern ist marginal.

Nachrichtenlänge [bytes]	4	40	400	4000	40000	400000
Zeit (lokal) [ms]	5,9	5,4	5,2	6,1	9,6	21,0
Standardabweichung (lokal) [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
Standardabweichung (remote) [ms]	21,8	7,1	7,0	11,4	7,2	34,8

Bei den Methodenaufrufen auf Actives in fremden JVMs wird deutlich, dass zwischen der Kommunikation innerhalb eines Rechners und der über das Ethernet-Netzwerk bei kleineren Nachrichten kaum ein Unterschied besteht. Die Unterschied im Bereich bis zu einer Parameter-Größe von 4000 Byte sind auf statistische Abweichungen zurückzuführen (große Standardabweichung).

Ebenfalls wird deutlich, dass die Startup-Zeit für Methodenaufrufe eine wesentlich größere Rolle spielt als die Größe der übergebenen Daten. Die Zeiten für Methodenaufrufe sind bis zu einer übergebenen Datenmenge von 40000 Byte nahezu konstant, erst dann beginnt die benötigte Zeit mit der Datenmenge zu wachsen.

8.3 Einsetzbarkeit

Derzeit sind leider keine größeren Projekte, die ProActive benutzen, öffentlich zu finden. Einen echten Nachweis seines Potenzials bleibt ProActive daher schuldig. Allerdings lassen sich mögliche Einsatzgebiete in zahlreichen Beispielen des ProActive-Teams selbst sowie in Tests des „2nd GRID Plugtests“ des ETSI ausmachen.

Auf der ProActive-Website werden neben einfachen Tutorial-Beispieln zur Demonstration bestimmter Features größere Beispiel-Projekte vorgestellt, unter anderem eine Monte Carlo Simulation, ein 3D Renderer, eine Simulation für das n-Körper-Problem der Physik, eine Simulation elektromagnetischer Felder und Bildgenerierung mit Fractal. Im „GRID Plugtest“ wurden klassische Programmier-Probleme wie das n-Damen-Problem und FlowShop für das GRID gelöst und die Performance getestet.

Abbildungsverzeichnis

1 Subsysteme innerhalb von Nodes 5
2 Hierarchie der Teilsysteme 5
3 Komponenten eines Active Objects einer Klasse A: 9
4 Monitoring mit IC2D 14
5 Aufbau von Komponenten in ProActive 15
6 Parallele Komponente 16
7 Speedup beim Rendering-Benchmark 17

Literatur

[1] *What is the Grid? A Three Point Checklist*, I. Foster, Argonne National Laboratory & University of Chicago, Chicago 2002.

[2] *About CoreGRID*, `http://www.coregrid.net/mambo/content/view/2/25/`.

[3] *Grid Computing - The evolution*, `http://www.grid.org/about/gc/evolution.htm`.

[4] *ProActive - A Comprehensive Solution for Grid Computing*, D. Caromel et al., Nice 2006, `http://www-sop.inria.fr/oasis/proactive/doc/release-doc/pdf/ProActiveManual.pdf`.

[5] *Open Source Middleware for the Grid: ObjectWeb ProActive*, D. Caromel, Nice 2006, `http://www-sop.inria.fr/oasis/proactive/doc/presentation/ProActive-Current-Complete-Talk-Jan-27.ppt`.

[6] *Das Komponentenmodell Fractal*, C. Zengler, Hauptseminar Moderne Programmiermethoden, Passau 2006.