Paralleles Programmieren mit Java

Heinz Kredel[*] Akitoshi Yoshida[†]

Zusammenfassung

Parallele Programmierung wird oft mit High Performance Computing assoziiert aber mit dem Programmiersystem Java stehen alle Hilfsmittel für die Nutzung dieser Techniken in beliebigen Anwendungen zur Verfügung. Wir besprechen die parallele Programmierung von Rechnern mit gemeinsamem Speicher, im sogenannten Thread Modell, und von Rechnern mit verteiltem Speicher, im Netzwerk- oder Kommunikations-Modell. Neben den wesentlichen Sprachkonstrukten und Klassen gehen wir auf einige Probleme und ihre Lösung mit den Mitteln von Java ein. Den Abschluß bilden die neuesten Entwicklungen, wie zum Beispiel RMI, CORBA und Java Grande.

1 Einleitung

Auch wenn paralleles Programmieren schon einige Zeit erforscht ist und geeignete Werkzeuge zur Implementierung vorhanden sind, werden diese doch nur von wenigen Programmieren konsequent genutzt. In diesem Artikel wollen wir zeigen, wie mit der Programmiersprache Java [1] und der entsprechenden Bibliotheksumgebung parallele Programme geschrieben werden können. Dabei können wir in der Kürze nur die wichtigsten Sprachkonstrukte und Klassen vorstellen. Es wird aber ausreichen, um Sie von der Leistungsfähigkeit von Java für dieses Gebiet zu überzeugen. Auch wenn die Java Programme hinterher nicht die Performance von numerischen FORTRAN oder C++ Programmen erreichen, so erleichtert Java und seine Umgebung doch sehr die Entwicklung von vielen parallelen Anwendungen. Für Java spricht auch die allgemeine Verfügbarkeit, und die gute Integration aller für die parallele Programmierung wichtigen Hilfsmittel. Somit können auch Auszubildende, ohne Zugang zu einem Parallelrechner zu benötigen, die Grundlagen der parallelen Programmierung zu lernen.

Im Rest dieses Abschnitts führen wir kurz in die Problematik des parallelen Programmierens ein. Dann besprechen wir im nächsten Abschnitt 2 zunächst

[*]Rechenzentrum Universität Mannheim, e-Mail: kredel@rz.uni-mannheim.de
[†]SAP AG Walldorf, e-Mail: Akitoshi.Yoshida@sap-ag.de

Threads und ihre Programmierung und anschließend in Abschnitt 3 die Programmierung der Kommunikation. In Abschnitt 4 gehen wir auf neuere Entwicklungen ein. Eine ausführlichere Besprechung aller notwendigen Konzepte und ihrer Implementierung mit Java finden Sie in unserem Buch [4]. Um den Umfang für diesen Artikel nicht zuüberschreiten, setzen wir im folgenden elementare Kenntnisse in Java voraus.

1.1 Problemstellung

Die Hardware-Entwicklung der letzten Jahre und Jahrzehnte hat zu einer weiten Verbreitung von Multitasking-Betriebssystemen, Multiprozessor Rechnern sowie zu Netzwerken von Workstations und PCs geführt. Eine Konsequenz dieser Entwicklung war es, daß nun verschiedene (Berechnungs-) Aufgaben gleichzeitig und nebeneinander bearbeitet werden können. Die Programmierung dieser Systeme wird als Parallele oder Nebenläufige Programmierung bezeichnet; im Englischen wird etwas treffender von 'concurrent programming' gesprochen. Die Software mit der die Systeme programmiert werden, hat folgende Hilfsmittel, um die gleichzeitige Bearbeitung auszudrücken:

- Erzeugen und Anstoßen von Aufgaben
- Synchronisierung des Zugriffs auf Ressourcen
- Koordinierung der Ressourcen
- Definition von Verbindungen (Kanälen) zur Kommunikation
- Informationsaustausch zwischen nebenläufigen Programmen

Aus der Sicht eines Programmierers besteht die Aufgabe während der Entwicklung von Programmen in der geeigneten wechselseitigen Abstimmung des Algorithmus und der verwendeten Datenstrukturen. In der Parallelen Programmierung gilt es – unabhängig von der zugrundeliegenden Hardware – die verschiedenen Möglichkeiten der Datenhaltung zu berücksichtigen: Daten im gemeinsamen (Haupt-)Speicher oder Daten verteilt auf lokale Speicher in vernetzten Rechnern. Im ersten Fall muß der gemeinsame Zugriff mehrerer Programme auf diese Daten synchronisiert werden; im zweiten Fall müssen die Daten zwischen den Rechnern transportiert werden. Die Optimierung der Programme bedeutet im ersten Fall eine Verringerung der Zugriffskonflikte, im zweiten Fall eine Verringerung des Datentransports.

Zur Implementierung paralleler Programme stehen im Wesentlichen zwei Techniken bereit: Prozesse oder Threads. *Prozesse* sind eigenständige Programme, die vom jeweiligen Betriebssystem unabhängig voneinander zur Ausführung gebracht werden. Unter DOS und Windows sind das also die EXE-Dateien und unter Unix die normalen Binaries (a.out-Dateien). Prozesse können über verschiedene Hilfsmittel mit anderen Prozessen in Kontakt treten: TCP/IP Sockets,

Message Passing Bibliotheken (wie PVM [2] oder MPI [3]), Pipelines (über die Standard-Eingabe und -Ausgabe) oder über gemeinsame Speicherbereiche, die vom Betriebssystem angefordert werden. *Threads*, auch Ausführungsfäden genannt, sind keine eigenständigen Programme, sondern vielmehr Teile von Prozessen. In jedem Thread wird ein Unterprogramm ausgeführt, das uneingeschränkten Zugriff auf alle globalen Daten des umgebenden Prozesses hat. Zu den globalen Daten gehören globale Variablen, Datei-Handles und auch Netzverbindungen. Nur die lokalen Daten der Unterprogramme sind in jedem Thread verschieden.

2 Thread Programmierung

Die Thread Funktionalität kann in Programmiersprachen durch eigene Sprachkonstrukte oder durch externe Bibliotheken realisiert sein.

In Java stehen Threads als Basisklassen zusammen mit Spracherweiterungen zur Verfügung. In Ada gehören Threads, hier “Tasks” genannt, zum Sprachumfang. Als minimale Lösung werden in C, C++, Modula-2, FORTRAN und anderen Programmiersprachen Threads durch sprachunabhängige Programmbibliotheken angeboten. Eine standardisierte Bibliothekslösung stellt die POSIX Threads Bibliothek “Pthread” dar. Sie ist in OSF DCE (Open Software Foundation, Distributed Computing Environment) enthalten, wird aber auch auf verschiedenen anderen Systemen angeboten. POSIX bedeutet Portable Operating System IX, ein Standardisierungskommitee von IEEE.

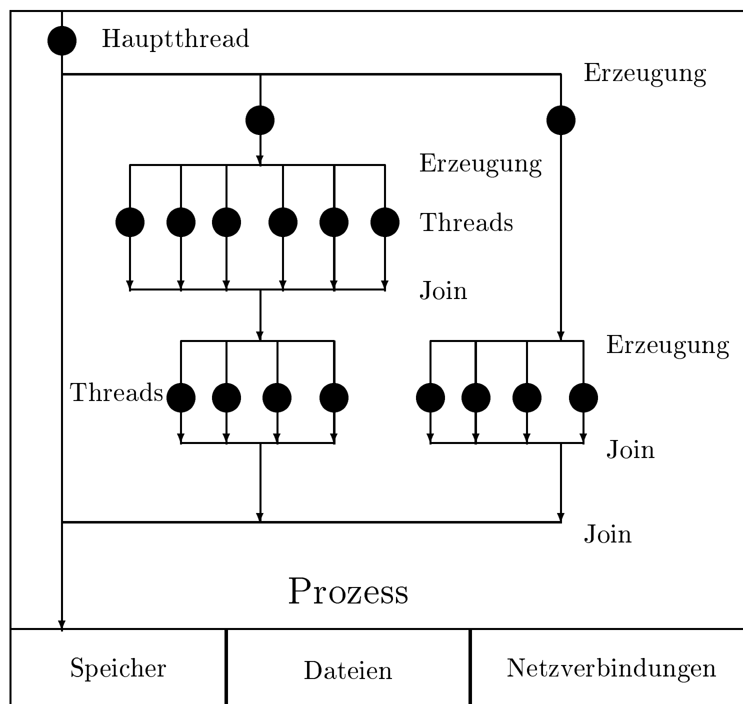
Auch die Java Threads werden - je nach Plattform - mit Hilfe von POSIX Threads implementiert. Thread Bibliotheken mit ähnlicher Funktionalität wie POSIX Threads sind z.B. implementiert auf OS/2 und Windows NT. Wichtig für eine effiziente Implementierung von Threads ist, daß Threads schon vom Betriebssystem bereitgestellt werden und nicht aufgesetzt sind.

In jedem Prozeß (siehe Abbildung 1) wird zunächst vom Betriebssystem ein Hauptthread erzeugt und gestartet. Dieser kann nun weitere Threads erzeugen, die wiederum weitere Threads generieren können. Alle Threads können auf die gemeinsamen Ressourcen (Speicher, Dateien, Netzverbindungen) zugreifen und diese auch modifizieren. Falls ein Thread alle seine ihm zugedachten Aufgaben getan hat, kann er terminieren. Die Zustände, in denen sich Threads (wie auch Prozesse) befinden können, sind: *running*, *blocked*, *ready* oder *terminated*.

Die Java-Implementierung von Threads, die in dem Java-Package `java.lang` enthalten ist, besteht aus folgenden Teilen:

1. der Klasse `Thread` und dem Interface `Runnable`,
2. dem Java-Sprachkonstrukt `synchronized` und
3. den Methoden `wait()`, `notify()` der Basisklasse `Object`.

Abbildung 1: Threads innerhalb von Prozessen



Die Bedeutung und die Anwendung dieser Teile besprechen wir in den nächsten drei Abschnitten.

2.1 Thread Erzeugung

In Java muß zur Erzeugung von Threads in üblicher Weise eine geeignete Klasse instanziiert werden, und dann wird im richtigen Moment eine Methode dieser Instanz aufgerufen. Für die Klasse gibt es zwei Möglichkeiten

- Bildung einer Subklasse der `Thread`-Klasse
- Bildung einer Klasse, die das `Runnable` Interface implementiert.

Die Definition als Subklasse von `Thread` hat den Vorteil, daß bequem alle Methoden der `Thread`-Klasse geerbt und somit verwendet werden können. Der

wesentliche Nachteil dieses Ansatzes wird dadurch verursacht, daß Java keine Mehrfachvererbung erlaubt. Wenn zum Beispiel eine Klasse von der Klasse `Applet` abgeleitet wird, kann sie nicht gleichzeitig von `Thread` abgeleitet werden. In der Regel wird daher ein Thread durch eine Klasse definiert, die das `Runnable`-Interface implementiert. Das `Runnable`-Interface verlangt nur die Implementierung der Methode `public void run()` und hat keine weiteren Anforderungen. Wir bevorzugen meist die zweite Methode. Zur Vereinfachung der Ausdrucksweise werden wir oft von `Runnable`-Klassen sprechen, wenn wir Klassen meinen, die das `Runnable`-Interface implementieren.

Der `Thread`-Konstruktor und die Thread-Methoden `start()` und `join()` haben die folgenden Spezifikationen.

```
public Thread(Runnable target)
public Thread(Runnable target, String name)
public Thread(ThreadGroup group, Runnable target, String name)
public synchronized void start()
public final void join() throws InterruptedException
```

Mit einer `ThreadGroup` lassen sich Gruppen von Threads definieren. Auf die gesamte Gruppe lassen sich dann Operationen zur Steuerung der Mitglieder-Threads anwenden. Falls der Thread keiner besonderen `ThreadGroup` angehören muß und er auch keinen Namen haben braucht, reicht die erste Variante `Thread(Runnable target)`. Mit der `start()`-Methode von Thread wird dann die `run()`-Methode der Klasse `myRunnable` gestartet. Mit der `join()`-Methode von Thread kann auf die normale Beendigung des Threads gewartet werden. Thread kennt noch die `stop()`-Methode, mit der ein laufender Thread abgebrochen werden kann. Normalerweise wird man `stop()` selten benutzen und lieber ein Verfahren implementieren, das zu einer normalen Terminierung der Threads führt. `stop()` soll ab Java 1.2 nicht mehr verwendet werden.

Um einen ersten Eindruck eines parallelen Java-Programms zu bekommen, betrachten wir folgende Programmteile.

```
class Action implements Runnable {
   int var;
   public Action(int v) { var = v; }
   public void run() { doSomeWork(var);  }
}
```

Der erste Teil definiert eine Klasse, die `Runnable` implementiert. Der Konstruktor nimmt einen Parameter entgegen, der dann in der Funktion `doSomeWork()` in der `run()`-Methode verwendet wird. Die `run()`-Methode wird später von der jeweiligen `start()`-Methode gestartet.

```
Thread t1 = new Thread(new Action(1));
Thread t2 = new Thread(new Action(2));
```

```
Thread t3 = new Thread(new Action(3));
try {
    t1.start(); t2.start(); t3.start();
    t1.join(); t2.join(); t3.join();
}
catch (InterruptedException e) { ... }
```

Der zweite Teil zeigt die Erzeugung von drei Threads `t1`, `t2` und `t3`. Die Threads werden jeweils mit einer neuen Instanz der Klasse `Action` erzeugt. Dann werden die Threads der Reihe nach mit `ti.start()` gestartet. Anschließend wird mit drei `ti.join()` auf die Terminierung der Threads gewartet.

2.2 Synchronisation von kritischen Bereichen

Da innerhalb der verschiedenen `run()`-Methoden globale Variablen vorkommen können, kann es passieren, daß gleichzeitig auf ein und dieselbe Variable zugegriffen wird. An diesen Stellen ist einige Vorsicht geboten, da sonst die Werte dieser Variablen am Programmende falsche oder zufällige Werte enthalten können. Um diese unerwünschten Überlappungen zu vermeiden, ist es erforderlich, daß sich Programmteile gegenseitig ausschließen (*mutual exclusion*) oder gegebenenfalls die Abarbeitungsschritte aufeinander abstimmen (*condition synchronization*).

Da wir nicht verhindern können, daß Schreib- oder Lese-Operationen auf den globalen Speicher in nebenläufigen Prozessen stattfinden und sichtbar werden, benötigen wir einen Trick. Dieser besteht darin, den Thread in der Ausführung anzuhalten, der die Zustandsänderungen nicht sehen soll. Dies setzt voraus, daß wir bei Bedarf nicht nur lokal ein Haltekonstrukt einfügen, sondern wir müssen alle gefährdeten "kritischen" Bereiche in *allen* Threads ausfindig machen und dort ebenfalls ein passendes Haltekonstrukt einfügen. Dies ist keine so große Einschränkung wie es zunächst aussehen mag, denn wir können die Variablen, die in kritischen Statements vorkommen, oft in einer Klasse isolieren und ihre Modifizierung durch geeignete Methoden kontrollieren. Das Anhalten der Threads erreichen wir durch das `synchronized`-Sprachkonstrukt.

Das Java-Sprachkonstrukt `synchronized` hat die folgenden Varianten.

```
synchronized (object) { ... }
synchronized (static object) { ... }
synchronized type methodName(...) { ... }
static synchronized type methodName(...) { ... }
```

Die erste Variante benutzt eine Objektinstanz `object` einer beliebigen, von der `Object`-Klasse abgeleiteten Klasse als Haltepunkt. Das heißt: wird in verschiedenen Threads ein `synchronized (object)` auf ein bestimmtes Objekt `object` ausgeführt, so stellt das Java-Laufzeitsystem sicher, daß immer nur maximal *ein* Thread die Statements `{...}` ausführen kann. Man spricht dann auch davon,

daß man einen "lock" das Objekt setzt (es also "verschließt"). Ein "lock" kann immer nur einmal zu einem gegebenen Zeitpunkt aktiv sein. Man spricht dann auch von gegenseitigem Ausschluß ("mutual exclusion"). Die "lock"-Objekte werden dann auch als "mutex" bezeichnet.

Falls das Objekt als `static` deklariert ist, findet ein "lock" Systemweit nur einmal statt, da das Objekt nur einmal vorhanden ist, sonst bezieht sich der "lock" nur auf eine Instanz eines Objekts. Das heißt, mehrere Instanzen eines Objekts haben dann verschiedene "locks", die untereinander nicht synchronisiert sind.

Die Semantik von `synchronized type methodName(...) { S1; ...; Sn; }` entspricht der von

```
type methodName(...) {
     synchronized(this) { S1; ...; Sn; }
}
```

Dies zeigt auch, daß Methoden einer Klasse, die nicht als `synchronized` deklariert sind, frei auf die Instanzendaten zugreifen können es findet *keine* Synchronisation statt. Damit stellt Java auch keine echten Monitore, wie sie von Hoare entwickelt wurden, zur Verfügung. Nur wenn alle Nicht-`private`-Methoden `synchronized` sind und es nur `private`-Variablen gibt, erhält man etwas ähnliches wie einen Monitor.

2.3 Warten auf Bedingungen

Nachdem wir die kritischen Bereiche im Griff haben, stellen wir uns dem letzten Problem. Wenn wir zum Beispiel eine Summe in einer globalen Variablen bilden wollen, müssen wir den Zugriff darauf mit `synchronized()` kontrollieren, aber bevor wir überhaupt anfangen zu summieren, muß sicher sein, daß die Variable initialisiert ist. Bei dem Summen-Beispiel können wir das machen, in dem wir die Threads erst nach der Initialisierung erzeugen. Bei komplexeren Datenstrukturen werden wir aber unter Umständen die Initialisierung mit in die Threads einbeziehen. In dieser Situation benötigen wir eine Möglichkeit zu warten, bis dieser Schritt abgeschlossen ist.

Zur Verfügung stehen uns die `Object`-Funktionen `wait()` und `notify()` mit den folgenden Spezifikationen:

```
public final void wait() throws InterruptedException
public final void wait(long timeout)
                       throws InterruptedException
public final void notify()
public final void notifyAll()
```

Bei Aufruf von `wait()` wird der aufrufende Thread in den Wartezustand versetzt ("blocked"), und gleichzeitig wird der Lock auf diesen Abschnitt freigegeben. Der Thread wartet nun, bis ein anderer Thread die `notify()`-Methode dieses Objekts aufruft oder bis ein Timeout stattfindet. Dann wartet der Thread eventuell noch einmal, bis er den Lock auf den Abschnitt wieder erhält, und wird dann wieder "ready" ("runnable"). `notify()` weckt *genau einen* Thread im wait-Zustand auf. Falls mehrere Threads warten, ist nicht vorhersehbar oder bestimmbar, welcher Thread aufgeweckt wird.

Die Methode `wait()` darf nur innerhalb eines Abschnitts aufgerufen werden, der mit `synchronized` geschützt ist.

Die `notify` entsprechenden Funktionen in anderen Thread-Paketen (wie `signal` in Pthreads) garantieren nicht, daß nur genau ein Thread aufgeweckt wird. Dort wird spezifiziert, daß *mindestens ein* Thread aufgeweckt wird.

`notifyAll` weckt *alle* an diesem Objekt wartenden Threads auf. Varianten von `wait()` mit `timeout` sind das 'timed wait', bei dem eine Zeit angegeben wird, wie lange maximal auf das Eintreten einer Bedingung gewartet werden soll. Falls die Zeit verstrichen ist, terminiert die Methode, als ob ein `notify()` stattgefunden hätte. Das heißt, auch der Lock wird dann wieder von diesem Thread gehalten. Es muß nun die Bedingung erneut getestet werden, um festzustellen, ob nur die Zeit abgelaufen ist oder die Bedingung tatsächlich erfüllt ist.

Wenn wir auf die Erfüllung eines beliebigen Booleschen Ausdrucks warten wollen, müssen wir für jede semantisch verschiedene Bedingung eine eigene Bedingungsvariable (d.h. ein eigenes Objekt) einführen.

Für jede Bedingung wird dann ihr Test, ob sie wahr ist, an alle (wichtigen) Stellen in das Programm verlegt, an denen die Bedingung wahr geworden sein könnte, und dort wird, falls ja, mit `notify` (oder `notifyAll`) der Eintritt der Bedingung signalisiert. Da von mehreren Threads gleichzeitig oder kurz hintereinander ein `notify` ausgelöst werden kann, ist es für den wartenden Thread wichtig, die Verzögerungsbedingung erneut zu testen, da ein weiterer Thread sie in der Zwischenzeit schon wieder ungültig gemacht haben könnte.

Ein weiteres Problem entsteht dadurch, daß z.B. `notify()` vor `wait()` ausgeführt werden könnte, was den einen Thread auf immer blockieren würde (sogenannte "lost signals"). Eine Lösung zu diesem Problem stellen Semaphore dar, die im Buch [4] in Abschnitt 3.4 besprochen werden.

2.4 Zusammenfassung

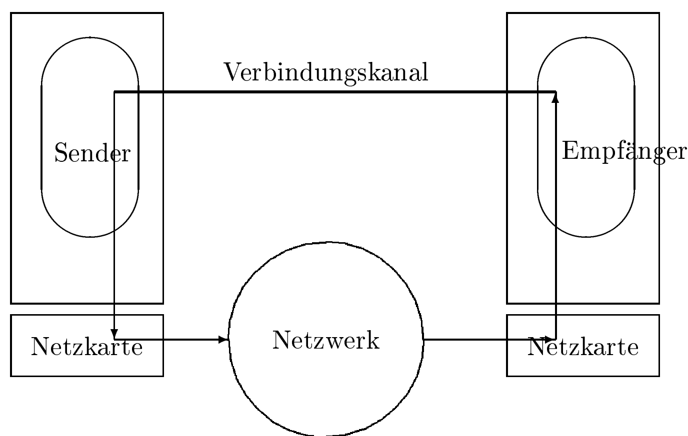
Wir haben die wichtigsten Java Sprachkonstrukte zur Thread Programmierung vorgestellt. Anwendungen sind zum Beispiel Semaphore, Barrieren und das Bounded-Buffer-Problem, sowie die Verwendung von Threads in der Applet-Programmierung, auf die wir hier aber in der Kürze nicht eingehen können. Auch in den Enterprise Java Beans und in den neuen Java Swing Klassen werden Threads extensiv angewendet.

3 Programmierung der Kommunikation

Mehrprozessorsysteme *ohne* gemeinsamen Hauptspeicher (wie z.B. Workstation-Cluster) benötigen spezielle Leitungen zur Kommunikation. (Ethernet und TCP/IP bei Workstation-Clustern, Dateien (bzw. Pipes) bei Unix-Prozessen).

Die Programmierung solcher Systeme erfordert daher die Definition von geeigneten Leitungen und Übertragungsprotokollen. Zur Kommunikation werden explizite Befehle zum Senden und Empfangen von Daten benötigt. Der Schutz von gemeinsamen Variablen ist nicht mehr erforderlich, da alle Programmteile als kritische Bereiche ausgeführt werden. Das Schema des Nachrichtenaustauschs ist in Abbildung 2 dargestellt.

Abbildung 2: Schema des Nachrichtenaustauschs



Die Java-Socket-Kommunikation basiert auf den durch das Internet bekannten TCP/IP Sockets. Diese Sockets werden mittlerweile von praktisch allen Betriebssystemen unterstützt. Auch nahezu jede Treiber-Software von Netzwerk Hardware (Leitungen und Rechnerkarten) bietet Unterstützung für TCP/IP und damit für Sockets. Insbesondere unterstützen auch High Performance Netzwerke TCP/IP – wenn auch manchmal mit schlechterer Performance als speziell angepaßte Software.

Sockets stellen aus Programmiersicht die Software-Schnittstelle zu einem Netzwerk dar. Sie entsprechen etwa den Filehandles, die für den Zugriff auf Plattendateien angelegt und verwaltet werden müssen. Java-Netzwerk-Support befindet sich im Package `java.net`.

Wir beschränken uns in diesem Artikel auf die Erläuterung des einfachsten Falls einer einzigen Punkt-zu-Punkt Verbindung. Zum Beispiel können wir damit

nicht direkt 1-zu-n- oder m-zu-n-Verbindungsnetzwerke realisieren.

3.1 Aufbau von Verbindungskanälen

Zur Implementierung dieser Klassen verwenden wir von Java die Klassen `ServerSocket` und `Socket`. Um eine Kontrolle über einen korrekten Aufbau eines Kanals zu bekommen, wird die Socket-Verbindung unsymmetrisch aufgebaut. Ein Kanalende (der Server Socket) baut seine Datenstrukturen auf und wartet dann auf einen Verbindungswunsch vom anderen Ende. Das andere Kanalende (der Client Socket) baut ebenfalls zuerst seine Datenstrukturen auf und versucht dann, eine Verbindung zur anderen Seite aufzubauen. Gelingt dies auf beiden Seiten, besteht ein zuverlässiger Verbindungskanal. Es wird also nicht erst bei einem folgenden Senden und Empfangen festgestellt, daß eine Verbindung gar nicht zustande kam.

Die Spezifikation des `ServerSocket`-Konstruktors und der benötigten Methode ist wie folgt.

```
public ServerSocket(int port) throws IOException
public Socket accept() throws IOException
```

Der Konstruktor `ServerSocket` erzeugt einen neuen Server Socket an dem angegebenen `port`. Eine Portnummer `0` erzeugt einen Socket an einem beliebigen freien Port. Die Funktion `accept()` wartet auf einen Verbindungswunsch auf dem entsprechenden Port und gibt dann einen `Socket` für die Verbindung zurück. Die Funktion blockiert solange, bis eine Verbindung zustande kommt.

Es folgen die Spezifikationen des `Socket`-Konstruktors und der benötigten Methoden.

```
public Socket(String host, int port)
       throws UnknownHostException, IOException
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
```

Der Konstruktor `Socket` erzeugt einen neuen Client Socket zu dem angegebenen `host` und `port`. Der Socket stellt einen Eingabe- und einen Ausgabe-Strom zur Verfügung, auf die mit den Methoden `getInputStream()` und `getOutputStream()` zugegriffen werden kann. Ein Strom ist eine unformatierte und unstrukturierte Folge von Daten. Die Ströme `InputStream` und `OutputStream` bestehen aus Folgen von Bytes.

Zu den Eingabe- und Ausgabe-Strömen kann nun ein zu den Anforderungen passender Datenstrom zu geordnet werden. Für Folgen von reinen Unicode Zeichen könnten wir die Ströme `Reader` und `Writer` einsetzen. Wir verwenden in diesem Abschnitt `ObjectStreams`, da damit sehr flexibel beliebige (serialisierbare, `serializable`) Objekte über den Kanal ausgetauscht werden

können. Ein Objekt-Strom kann einfache und zusammengesetzte Java-Objekte in einen Daten-Strom umwandeln und typsicher versenden oder empfangen. Daten-Ströme können sowohl Datei-Ströme als auch Netzwerk-Socket-Ströme sein. Es werden nur Objekte, die das `java.io.Serializable` Interface implementieren in den Objekt-Strömen akzeptiert. Objekt-Serialisierung ist eins der wesentlichen neuen Features von Java 1.1.

Falls eine Klasse `Serializable` implementiert, wird die Objekt-Serialisierung von Java automatisch durchgeführt. Überschreibt man die automatische Serialisierung, muß man die Klasse selbst kodieren und senden. Bei manchen Objekten wie Filehandles macht die Serialisierung natürlich keinen Sinn, denn ein Filehandle zeigt vielleicht auf eine Datei, die auf einem anderen Rechner nicht existiert.

Die Spezifikation der benötigten Konstruktoren und Methoden ist wie folgt.

```
public ObjectOutputStream(OutputStream out) throws IOException
public void flush() throws IOException
public ObjectInputStream(InputStream in)
       throws IOException, StreamCorruptedException
```

Der Konstruktor `ObjectOutputStream` erzeugt einen neuen Objekt-Ausgabestrom, zu einem gegebenen `OutputStream`. Zuerst wird ein Strom-Header geschrieben, der eine eindeutige Identifikation der Objekt-Strom-Klasse enthält. Mit `flush()` wird sichergestellt, daß dieser Header unmittelbar verschickt wird, damit die Gegenseite die Daten sofort einlesen kann.

Der Konstruktor `ObjectInputStream` erzeugt einen neuen Objekt-Eingabestrom, zu einem gegebenen `InputStream`. Zuerst wird ein Strom-Header gelesen. Passt die Identifikation im Header nicht zu der eigenen, weil beispielsweise die Daten von einem anderen Rechner mit einer inkompatiblen JDK-Version über das Netz kommen, so wird ein Fehler ausgelöst (hier ist das `StreamCorruptedException`). Der Konstruktor blockiert, bis ein Objekt-Ausgabe-Strom die entsprechenden Daten gesendet hat.

3.2 Senden und Empfangen

Zur eigentlichen Datenübertragung benötigen wir auf einer Seite eine Send-Operation (**send**) und auf der anderen Seite eine Empfangs-Operation (**recieve**). Während der Aufruf der Empfangs-Operation immer blockierend ist, kann die Send-Operation sowohl synchron als auch asynchron arbeiten:

asynchronous send = das Programm wird nach dem Einstellen der Daten in einen Sendepuffer ohne weitere Verzögerung fortgesetzt, d.h., **send** terminiert auch, wenn die Daten erst sehr viel später bei einem Empfänger ankommen.

synchronous send = die weitere Programmausführung wird nach dem **send** so lange blockiert, bis ein entsprechendes **receive** ausgeführt wurde, d.h. bis die Daten von einem Empfänger wirklich abgenommen worden sind.

Das asynchrone **send** erfordert einen potentiell unbeschränkt großen Puffer, während das synchrone **send** nur einen Puffer fester Größe erfordert. Die Verwendung des synchronen **send** ist schwieriger, da es sehr genau auf die zeitlich richtige Reihenfolge aller **send**- und **receive**-Operationen ankommt. Java unterstützt mit den Socket-Klassen nur das asynchrone **send**.

Zur Implementierung der Send-Operation mit Java stehen uns die Funktion `writeObject()` aus der Klasse `ObjectOutputStream` mit der folgenden Spezifikation zur verfügung.

```
public final void writeObject(Object obj)
        throws IOException
```

Die Methode `writeObject()` schreibt ein Objekt auf den entsprechenden Ausgabe-Strom. Der gesamte Graph des Objekts wird zerlegt (serialisiert), verpackt und zusammen mit dem Klassennamen und der Klassensignatur geschrieben.

Für die Empfangs-Operation steht uns die Java-Funktion `readObject()`, aus der Klasse `ObjectInputStream` zur verfügung, die die folgenden Spezifikation hat.

```
public final Object readObject()
        throws OptionalDataException,
               ClassNotFoundException, IOException
```

Die Methode `readObject()` liest ein Objekt von dem entsprechenden Eingabe-Strom. Der gesamte Graph des Objekts wird gelesen, entpackt und rekonstruiert.

3.3 Zusammenfassung

Wir haben in diesem Abschnitt die wichtigsten Java-Hilfsmittel zur Programmierung der Kommunikation zwischen Prozessen kennengelernt. Für parallele Anwendungen, die über das Client-Server-Schema hinausgehen, werden allerdings komplexere Implementierungen benötigt, die wir in [4] im Kapitel 5 diskutieren.

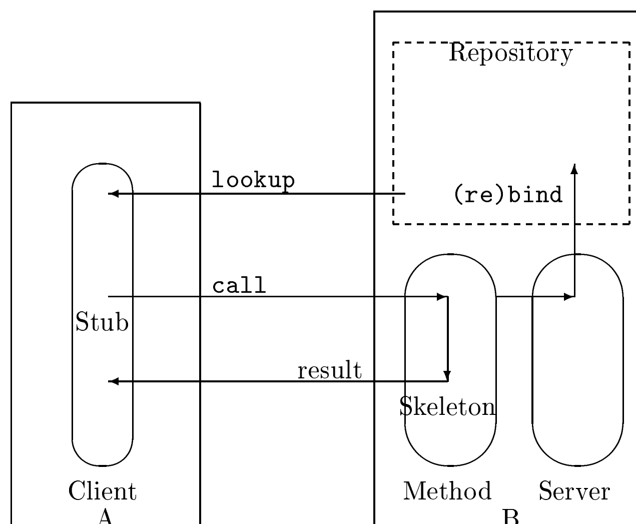
4 Ausblick

Im letzten Abschnitt geben wir einen Überblick über die Themen RMI, CORBA und Java Grande, die im Zusammenhang mit paralleler Programmierung wichtig sind.

4.1 RMI

Die Java Remote Method Invocation Technik (RMI), erlaubt die Ausführung von Methoden auf einem entfernten Rechner. Die Eingabeparameter werden über eine Netzverbindung zu einem entfernten Rechner geschickt, die entsprechende Methode wird dort ausgeführt und anschließend wird der Rückgabewert über die Netzverbindung zurück geschickt. Java RMI ist das Analogon zu Remote Procedure Call (RPC), das in der Kommunikationstechnik schon viele Jahre eingesetzt wird.

Abbildung 3: Remote Method Invocation



Die RMI Architektur wird in Abbildung 3 gezeigt. Computer 'A' bezeichnet den Rechner, von dem aus eine Methode auf dem entfernten Rechner 'B' aufgerufen werden soll. 'Client' bezeichnet den Benutzerprozess in Rechner A. 'Server' bezeichnet den Prozeß, der die aufrufbare Methode bereitstellt.

Damit die verwendbaren Methoden nicht im Programmcode auf der Server Seite fest angegeben werden müssen, wird ein Verzeichnis, genannt 'Repository', auf Rechner B geführt. Dieses Verzeichnis muß vor beginn aller RMI Aktivitäten gestartet sein. Server, die Methoden bereitstellen wollen verwenden `rmi.Naming.bind()` oder `rmi.Naming.rebind()`, um eine Klasseninstanz mit einem Namen bei dem Repository zu registrieren. Clients, die Methoden verwenden wollen sehen mit `rmi.Naming.lookup()` bei dem Repository nach ob ein gewünschtes Objekt vorhanden ist und verknüpfen es mit einem lokalen Objekt.

Beim Aufruf einer entfernten Methode im Client wird zunächst eine sogenannte 'Stub' Methode aktiviert. Diese sendet die Eingabeparameter (mit einer Bezeichnung der entfernten Methode) an eine 'Skeleton' genannte Methode auf dem Server. Die Skeleton Methode ruft dann auf dem Rechner B die gewünschte Methode auf und schickt den Returnwert nach der Terminierung der Methode an die Stub Methode zurück. Diese übergibt den Wert dann an die aufrufende Prozedur. Der Einsatz von Stub und Skeleton erfolgt transparent, d.h. ohne spezielle Aufrufe seitens der Programmierers.

4.2 CORBA

Nachdem es zunächst aussah, als ob Java RMI als direktes Konkurrenzprodukt zu CORBA entwickelt werden würde, hat sich mittlerweile die Einsicht durchgesetzt, daß eine Zusammenarbeit für beide Seiten viele Vorteile bringt. Mit dem JDK 1.2 wird Java volle CORBA Unterstützung bieten. Zusätzlich entwickeln Sun und IBM die Unterstützung von RMI über das IIOP (Internet Inter-ORB Protokoll). Damit könnte auch über RMI auf CORBA Objekte zugegriffen werden.

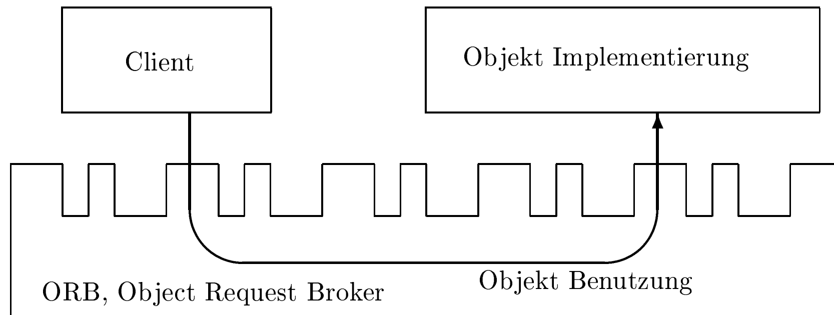
Der Name CORBA ist die Abkürzung für **C**ommon **O**bject **R**equest **B**roker **A**rchitecture. CORBA ist eine Spezifikation, das heißt eine Definition von Schnittstellen, die das Zusammenwirken von von Objekten in verteilten Umgebungen beschreibt. Die wesentlichen Punkte dieser Spezifikation sind:

- CORBA Objekte sind durch die ORB (Object Request Broker) Kommunikations-Infrastruktur transparent von Clients und Servern zugreifbar und benutzbar.
- CORBA ist unabhängig von einer bestimmten Programmiersprache. Es gibt unter anderem CORBA Anbindungen für Ada, Cobol, C++, Smalltalk und Java.
- CORBA ermöglicht sogenannte 'Multi-Tier' Systeme. Das heißt mehrschichtige verteilte Software-Architekturen. Statt einfacher Client-Server System lassen sich mehrstufige Systeme aus Clients, mehreren verschiedenen Anwendungsservern und Hintergrundprozessen wie Datenbanken realisieren.

Zur Kommunikation zwischen ORBs und Objekten mit Internet Mitteln gibt es das Internet Inter-ORB Protokoll (IIOP). Das Prinzip der CORBA Architektur ist in Abbildung 4 zusammen gefaßt. Der ORB definiert die Infrastruktur für das Zusammenwirken von Anwendern (Clients) und den durch verteilte Objekte bereitgestellten Diensten (Objekt Implementierungen).

Vollständige Informationen zu CORBA finden Sie im Internet auf dem Web-Server der OMG: `http://www.omg.org/`. Eine deutschsprachige Einführung in CORBA gibt auch das Buch von Sayegh [6]. Eine freiverfügbare, vollständige Implementierung des CORBA 2.0 Standards ist MICO [5].

Abbildung 4: Object Request Broker



4.3 Java Grande

Obwohl es unbestreitbar ist, daß Java zur Zeit im Vergleich zu FORTRAN oder C++ um mehr als den Faktor 10, also deutlich langsamer ist, werden dessen ungeachtet mittlerweile viele Java Anwendungen entwickelt, deren Design Ziele zunächst auf umfangreiche Funktionalität ausgerichtet waren. Diese Anwendungen verlangen aber an bestimmten Stellen nummerische Berechnungen, zum Beispiel zu graphischen Darstellung der Ergebnisse.

Zur Lösung dieses Problems, d.h. zur Verbesserung der nummerischen Leistungen von Java haben sich Anfang 1998 verschiedene Entwickler zum "Java Grande Forum" zusammen gefunden. Unter andrem sind James Gosling von Sun und Marc Snir von IBM im Forum vertreten. Java Grande soll Java für folgende Bereiche verbessern.

- High Performance Network Computing
- technisch-wissenschaftliches Rechnen
- verteilte Modellierung und Simulation
- paralleles und verteiltes Rechnen
- Anwendungen mit sehr großen Datenmengen
- rechenintensive kommerzielle Anwendungen

Die Ziele des Java Grande Forums sind:

- Stärkung des Potentials von Java als Entwicklungsumgebung für "Grande applications". D.h. Java als bessere Entwicklungsumgebung als FORTRAN oder C++ für große umfangreiche Anwendungen.

- Das Forum soll Konsens herstellen und Empfehlungen ausarbeiten für Weiterentwicklungen von Java selbst, oder für die Entwicklung von Standards (Frameworks) für "Grande" Bibliotheken oder Dienste.
- Die Java Sprachänderungen und Frameworks sollen die beste je verfügbare Programmierumgebung für große umfangreiche (nummerische) Anwendungen werden.

Unter dem Java Grande Forum haben sich zur Zeit zwei Arbeitsgruppen gebildet: eine mit Schwerpunkt Numerik und eine mit Schwerpunkt Parallel- und Verteiltes-Computing. Die wichtigsten Themen, die in den Arbeitsgruppen derzeit behandelt werden sind: volle Unterstützung der IEEE 754 Floating Point Spezifikation durch Java, Arrays, Interfaces zu MPI, PVM, sowie zu Lapack und BLAS, Tools, Verbesserung des Laufzeitsystem und der verteilten Garbage Collection, bessere Berücksichtigung der Speicherhierarchie (Register, Caches, Hauptspeicher). Die zweite Arbeitsgruppe befaßt sich vorrangig mit der Verbesserung von RMI, Support für Multicast, Unterstützung der Programmiermodelle SPMD und MIMD; Verbesserung der Virtuellen Java Maschine, schnellere Synchronisierung, bessere Skalierbarkeit der Threads.

Näheres zu Java Grande finden Sie im Internet unter dem folgenden URL[1] `http://www.npac.syr.edu/javagrande/`.

Zusammenfassung

Wir haben in diesem Artikel kurz umrissen, daß zum Lernen von paralleler Programmierung Java gut geeignet ist. Wo höhere Performance notwendig ist können Programmteile in C++ implementiert werden, auf die dann mittels Java Native Interface (JNI) zugegriffen werden kann. In der Java Grande Initiative wird an der Verbesserung von Java für Numerische und Parallele Anwendungen gearbeitet.

Literatur

[1] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM 3.3 Usersguide.* Oak Ridge, USA, 1995.

[3] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable parallel programming with the Message Passing Interface.* MIT Press, Cambridge, Mass., 1995.

[1] Stand: 16. September 1998

[4] H. Kredel and A. Yoshida. *Thread- und Netzwerk-Programmierung mit Java. Ein Praktikum für die Parallele Programmierung.* dpunkt.verlag, 1998.

[5] A. Puder and K. Römer. *MICO Is CORBA: A CORBA 2.0 compliant implementation.* dpunkt.verlag, 1998.

[6] A. Sayegh. *CORBA Standard, Spezifikation, Entwicklung.* O'Reilly, 1997.