Jini:
Programmiermodell

Seminararbeit im Rahmen der Veranstaltung

Verteilte und Parallele Systeme I

im Sommersemester 2001

bei Prof. Dr. Rudolf Berrendorf

Fachhochschule Bonn-Rhein-Sieg
Fachbereich Angewandte Informatik

Bearbeiter:

Hendrik Hasselberg

Abgabedatum:

02.07.2001

Inhaltsverzeichnis

1 Einleitung

Beim Design des Jini-Programmiermodells verfolgten die Entwickler vornehmlich zwei Ziele: zum einen sollte die Entwicklung von verteilten Anwendungen vereinfacht werden, zum anderen sollte ein einheitlicher Programmierstil über alle Jini-Anwendungen hinweg ermöglicht werden.

Im folgenden wird dargelegt, wie diese beiden Ziele erreicht werden sollen. Dazu wird zunächst das Leasing-Interface, dann das Event-Interface und zuletzt das Transaktionsinterface beschrieben.

2 Das Leasing-Interface

Im täglichen Leben wird beim Leasing durch einen der Vertragspartner dem anderen Vertragspartner die Nutzung einer Ressource für einen bestimmten, vorher definierten Zeitraum zugesichert. Dieses Modell lässt sich auch auf das Leasing-Interface bei Jini übertragen. Hier wird von einem Diensteanbieter dem Lookup-Service zugesichert, dass der angebotene Dienst für eine bestimmte Zeit genutzt werden kann.

Das von Jini genutzte Leasing entspricht dem von RMI (*Remote Method Invocation*) aus dem JDK (*Java Development Kit*) 1.2. Die dort definierte Klasse *Lease* dient dazu, den Gültigkeitszeitraum eines Remote-Objects zu spezifizieren. Läuft dieser Zeitraum ab, ohne dass die Lease-Time durch das Objekt verlängert wurde, wird das Object durch den automatischen Java-Garbage-Collector entfernt.

Die Verwendung eines Leasing-Konzeptes ist notwendig, da die Verbindung zwischen zwei Objekten in einem verteilten System jederzeit ausfallen kann, sei es durch einen Netzwerkausfall oder durch den Ausfall eines der Objekte selbst. Gleichzeitig muss ein Objekt (im folgenden *Client* genannt) aber zunächst von der Existenz eines Remote-Objects erfahren, bevor es dessen Funktionen nutzen kann. Dazu ist eine Registrierung des entfernten Objekts nötig, entweder bei einem Lookup-Server (dieses Konzept nutzen RMI und Jini, daher wird im folgenden von der Nutzung dieses Ansatzes ausgegangen) oder direkt bei dem Client.

Bei dieser Registrierung werden gleichzeitig die Bedienungen für das Leasing ausgehandelt, d.h. das Remote-Object teilt innerhalb der vom Lookup-Server festgelegten Grenzen mit, wie lange es voraussichtlich verfügbar sein wird. Läuft diese Zeit ab, muss das Objekt eine Verfügbarkeitsmeldung schicken, damit die Leasing-Zeit um den am Anfang definierten Zeitraum verlängert wird. Bleibt die Meldung aus, so wird die Registrierung vom Server gelöscht, um die verwendeten Ressourcen wieder freizugeben. Alle Clients, die das Remote-Object benutzen, werden von dessen Ausfall informiert.

Probleme können entstehen, wenn ein Remote-Object innerhalb seiner Leasing-Zeit ausfällt. Dieser Fall kann vom Lookup-Server nicht abgefangen werden und bedarf deshalb einer Fehlerbehandlung innerhalb des Clients. Ebenso entsteht durch die Verfügbarkeitsmeldungen eine je nach Anwendung und Netzkapazität nicht unerhebliche Netzlast, die beim Entwurf der Anwendung bedacht werden muss.

Zusammengenommen lässt sich das Leasing-Konzept also durch folgende Punkte charakterisieren:

- Es existiert ein Leasing-Zeitintervall, innerhalb dessen ein Diensteanbieter dem Dienstenehmer seine Verfügbarkeit zusichert.
- Der Anbieter kann seinen Dienst innerhalb des Intervalls beim Dienstenehmer kündigen.

- Der Leasing-Intervall kann erneuert werden, wenn der Dienst weiterhin zur Verfügung steht.
- Der Leasing-Intervall kann verfallen, wenn ein Dienst nicht mehr zur Verfügung steht.

Bei Jini sind diese vier Punkte durch das Interface `net.jini.core.lease.Lease` realisiert. Jeder Diensteanbieter muss dieses Interface implementieren, um die Basisfunktionalitäten zur Verfügung stellen zu können.

```
public interface Lease {
    long FOREVER = Long.MAX_VALUE;
    int DURATION = 1;
    int ABSOLUTE = 2;
    long getExpiration();
    void cancel () throws UnknownLeaseException, RemoteException;
    void renew (long duration) throws LeaseDeniedException;
    UnknownLeaseException();
    RemoteException();
}
```

Durch die Konstante `FOREVER` kann festgelegt werden, dass die Lease-Time des Dienstes niemals abläuft. Die Konstanten `DURATION` und `ABSOLUTE` definieren die Leasing-Zeit relativ bzw. absolut, jeweils in Millisekunden.

Über die Methode `getExpiration` kann die restliche Zeitspanne bis zum Ablaufen des Leasing-Intervalls abgefragt werden. Mit `cancel` kann der Diensteanbierter seinen Dienst beim Lookup-Service kündigen, mit `renew` kann er den Intervall verlängern.

Wie bei Java üblich, kann ein Entwickler das Interface beliebig erweitern. Dadurch wird eine hohe Flexibilität bei der Implementierung erreicht.[1]

3 Event- und Notification-Interface

3.1 Einführung

Bei der Ausführung eines Programmes können Situationen entstehen, bei denen ein Programm von der Reaktion eines Dritten abhängig ist. Solche Reaktionen nennt man Ereignis (*Event*). Beispiele für Events sind u.a. die Ereignisse, die bei einer graphischen Benutzeroberfläche auftreten. Ein Programm (im weiteren *Ereignisempfänger*) muss in solch einer Umgebung ständig auf Benutzereingaben warten, sei es durch Tastatur- oder Maus-Inputs. Es wäre nicht effizient und würde zu einer hohen CPU-Auslastung führen, wenn der Prozess die Eingabegeräte dabei ständig in einer Schleife abfragen würde. Deshalb wurde ein Modell entwickelt, bei dem der entsprechende Prozess durch den sog. *Ereigniserzeuger* sofort informiert wird, sobald ein Ereignis (in diesem Falle eine Eingabe) auftritt. Der Ereigniserzeuger muss anschließend eine Bestätigung über die erfolgreiche Auslösung des Events erhalten.

Natürlich muss ein Ereigniserzeuger darüber in Kenntnis gesetzt werden, dass ein entsprechender Prozess am Empfang von Events interessiert ist. Dazu ist eine Registrierung des Ereignisempfängers notwendig. Außerdem muss dieser ein entsprechendes Interface implementieren.

[1] [Bader, 2000, S. 29]

Das beschriebene Kommunikationsmodell wird auch als *Delegation Event Model* bezeichnet. Normalerweise gilt es sowohl in lokalen als auch in verteilten Umgebungen, im JDK ist es aber nur für eine JVM (*Java Virtual Machine*) implementiert.

3.2 Events bei Jini

Durch Jini wird einem Prozess in einer JVM erlaubt, einen Event in einer anderen JVM auszulösen. Dabei können sich die beiden JVM auch auf unterschiedlichen Systemen befinden, die z.B. durch ein Netzwerk miteinander verbunden sind. Auch können sie von unterschiedlichen Herstellern stammen.

In verteilten Systemen treten bei der Eventbehandlung bestimmte Problematiken auf, die in Systemen mit einem einheitlichen Adressraum nicht vorkommen. So können die Quittierungen der einzelnen Events z.B. netzwerkbedingt in einer anderen Reihenfolge beim Sender ankommen, als sie tatsächlich aufgetreten sind. Auch kann nicht vorhergesagt werden, wann diese eintreffen werden. Daher kann in einem verteilten System ein Proxy zwischen Sender und Empfänger geschaltet werden, der diverse organisatorische Funktionen übernimmt. Das Problem der nicht vorhersagbaren Reihenfolge wird dabei durch Sequenznummern gelöst, die den einzelnen Events zugeordnet werden und damit eine eindeutige Zuordnung von Ereignis und Bestätigung erlauben.

Damit existieren im Delegation Event Model folgende Elemente:

- ein Objekt, das durch die Registrierung Interesse am Empfang von Events zeigt, aber nicht zwangsläufig auch der Ereignisempfänger sein muss
- der Ereigniserzeuger
- der Ereignisempfänger

Der Ereignisempfänger muss, um den Empfang von Events zu ermöglichen, das Interface `net.jini.core.event.RemoteEventListener` implementieren:

```
public interface RemoteEventListener
    extends java.rmi.Remote, java.util.EventListener
{
    void notify (RemoteEvent theEvent)
          throws UnknownEventException, java.rmi.RemoteException
}
```

Die Methode `notify` dient dabei dazu, den Empfänger darüber zu informieren, dass ein bestimmter `RemoteEvent` ausgelöst wurde (von der folgenden Klasse werden nur die public-Methoden vorgestellt):

```
public class RemoteEvent extends java.util.EventObject
{
    public RemoteEvent (Object source, long eventID, long seqNum, Mar-
        shalledObject handback);
    public long getID();
    public long getSequenceNumber();
    public MarshalledObject getRegistrationObject();
}
```

Dabei bezeichnet `source` das Quellobject, `eventID` charakterisiert die Art des Events und `seqNum` enthält die Sequenznummer. Das `MarshalledObject handback` enthält einen Byte-Stream mit der serialisierten Repräsentation des übergebenen Objektes[2].

Zur Registrierung eines Empfängers beim Ereigniserzeuger existiert die Klasse `EventRegistration`:

```
public class EventRegistration implements java.io.Serializable
{
    public EventRegistration (long EventID, Object eventSource, Lease e-
        ventLease, Long seqNum)
    public long getID();
    public Object getSource();
    public Lease getLease();
    public long getSequenceNumber();
}
```

Hier enthält `eventLease` das Lease-Objekt (vgl. 2) für die Registrierung. Die Erläuterung der anderen Methoden und Argumente ergibt sich aus den bereits oben gegebenen Erklärungen.

Das beschriebene System lässt sich durch Fremdhersteller beliebig erweitern, wobei diese Erweiterungen in Juni als *Agenten* bezeichnet werden. Als Ausgangspunkt wird dabei die Methode `net.jini.core.event.RemoteEventListener.notify()` verwendet. Dadurch, dass dies die einzige Methode des Interfaces ist und diese darüber hinaus auch noch sehr einfach gehalten ist, werden spezifische Implementierungen sehr erleichtert[3].

4 Transaktionsinterface

In einem verteilten System ist natürlich jederzeit ein Ausfall der Verbindung zwischen zwei miteinander verbundenen Prozessen möglich. Wenn die über die Verbindung übertragenen Daten persistent, also dauerhaft, gespeichert werden sollen, müssen dafür spezielle Vorkehrungen getroffen werden. Eine Möglichkeit, dies zu erreichen, ist die Verwendung von Transaktionen, einem Konzept, das bereits seit längerer Zeit in der Datenbank-Welt bekannt ist.

[2] [SUN, 2000, „Class Marshalled Object"]
[3] [Bader, 2000, S. 31]

4.1 Allgemeine Einführung

Eine Transaktion ist eine Aktion, die aus mehreren Operationen besteht, die in einem logischen Zusammenhang stehen. Daher darf die Ausführung dieser Kette von Operationen nicht unterbrochen werden, um die Konsistenz der übertragenen Daten nicht zu gefährden. Eine Transaktion wird dementsprechend entweder vollständig oder gar nicht ausgeführt. Wenn während der Ausführung einer Transaktion ein Zustand auftritt, der ein weiteres Fortführen der Transaktion nicht gestattet, so werden alle bis dahin durchgeführten Operationen rückgängig gemacht und damit der Ursprungszustand wieder hergestellt. Verläuft die Transaktion erfolgreich, so wird vom Empfänger ein *Commit* an den Sender geschickt, das die erfolgreiche und vollständige Übermittlung bestätigt und garantiert, dass die Daten persistent gespeichert wurden.

4.2 Implementierung in Jini

Um diesen Mechanismus zu implementieren ist eine Unterstützung durch entsprechende Protokolle notwendig. Bei Jini wird das sog. *Two-Phase Commit Protocol* unter Einbeziehung von RMI-Funktionen benutzt, das hauptsächlich bei verteilten Transaktionen über mehrere Datenbanken hinweg verwendet wird. Dabei wird jede Transaktion von einem Koordinator (bei Jini *Transaktionsmanager* genannt) kontrolliert. Wesentlicher Vorteil des Verfahrens ist, dass der Zeitraum, in dem aufgetretene Probleme nur durch eine aufwendige Kommunikation zwischen dem Transaktionsmanager und den Teilnehmern gelöst werden kann, erheblich verkürzt wird.

Dazu wird der Commit in zwei Phasen durchgeführt: nach Abschluß der Datenübermittlung sendet der Transaktionsmanager zunächst ein „*Prepare Commit*“ an alle Teilnehmer. Bis zu diesem Zeitpunkt kann die Transaktion noch an keiner Stelle abgeschlossen sein, d.h. im Falle eines Fehlers kann jeder Teilnehmer die Transaktion selbsttätig abbrechen.

Antwortet einer der Teilnehmer (*Participants*) mit „*OK*“, gerät er in einen Zustand („*Pending Transaction*“), in dem er die Transaktion nicht mehr ohne Mitwirkung des Managers abbrechen kann. Signalisiert nun einer der anderen Teilnehmer ein „*Not OK*“, so schickt der Transaktionsmanager an alle Participants ein *Rollback*, das sie auffordert, sich in den Ursprungszustand zurück zu versetzen. Fällt der Koordinator aus, so müssen beim Wiederanlauf umfangreiche Protokolle ablaufen, um das Ergebnis der Transaktion festzustellen.

Erhält der Transaktionsmanager dagegen von allen Teilnehmern ein „*OK*“, kann er durch das Senden eines „*Final Commit*“ die Transaktion erfolgreich abschließen.

Durch dieses Protokoll wird also sichergestellt, dass eine Transaktion immer als eine Abfolge von voneinander untrennbaren Aktionen realisiert wird, die im Fehlerfalle leicht rückgängig gemacht werden können.

In Jini werden zur Implementierung des Modells Interfaces zur Verfügung gestellt, die sich in den Packages `net.jini.core.transaction` und `net.jini.core.transaction.server` befinden. Diese müssen von den Teilnehmern einer Transaktion ihrer Funktion entsprechend implementiert werden. Dabei ist zu beachten, dass während einer Transaktion erzeugte Objekte nur für die Beteiligten an dieser Transaktion sichtbar sind. Nach Abschluss der Transaktion (egal ob dieser regulär oder durch einen Abbruch geschieht) werden die erzeugten Objekte gelöscht.

Die Kommunikation innerhalb einer Transaktion hat einen vorgegebenen zeitlichen Ablauf, der im Folgenden beispielhaft anhand eines Sequenzdiagramms dargestellt wird:

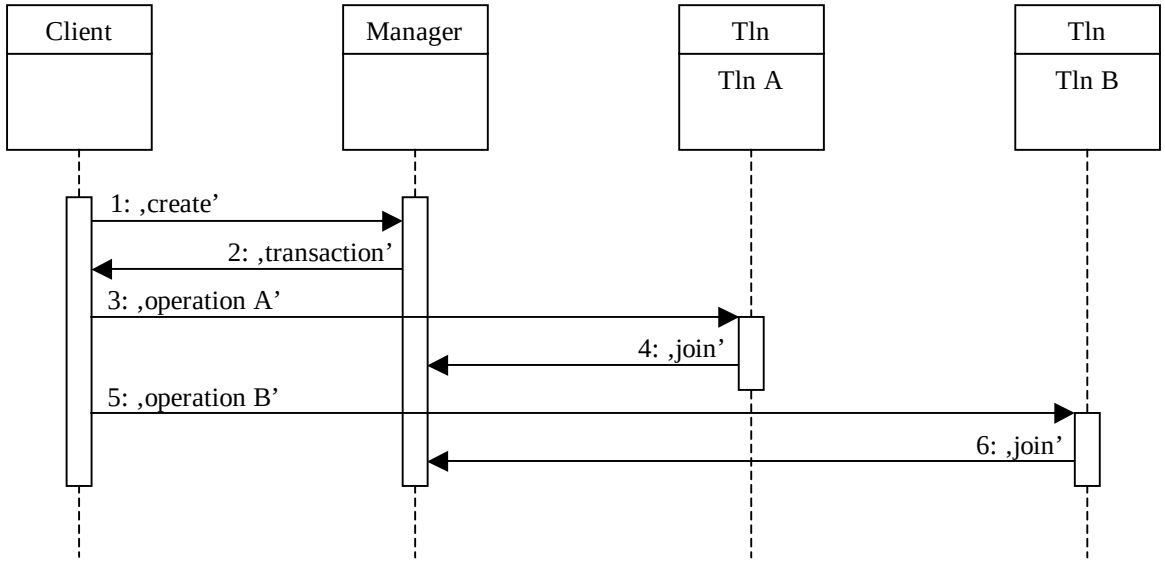

Abbildung 4.2.1[4]

Im Beispiel wird der Transaktionsmanager durch die `create`-Methode des Clients angewiesen, eine neue Transaktion zu beginnen. Durch Aufruf der `join`-Methode wird Dritten ermöglicht, an der Transaktion teilzunehmen.

Damit wird deutlich, dass drei verschiedene Typen von Beteiligten an einer Transaktion existieren, auf die in den folgenden Abschnitten näher eingegangen wird:

- der Erzeuger (Client)
- der Koordinator (Transaktionsmanager bei Jini)
- die Teilnehmer

4.2.1 Erzeuger einer Transaktion (Client)

Ein Client kann einen Transaktionsmanager, der das Interface `TransactionManager` implementieren muss, über einen Lookup-Service ermitteln. Folgende Methoden muss der Transaktionsmanager dabei zu Verfügung stellen[5]:

[4] [Bader, 2000, S. 39]
[5] [JINI, 2000, „Interface TransactionManager"]

```
public interface TransactionManager
extends java.rmi.Remote, TransactionConstants
{
    void abort(long id);
    void abort(long id, long waitFor);
    void commit(long id);
    void commit(long id, long waitFor);
    TransactionManager.Created create (long lease);
    getState(long id);
    void join(long id, TransactionParticipant part, long crashCount);
}
```

Mit Hilfe der `create`-Methode ist der Client nun in der Lage, aus diesem Transaktionsmanager heraus eine neue Transaktion zu starten. Dabei kann er über das Argument `leaseFor` den Zeitraum angeben, den eine Transaktion voraussichtlich einnehmen wird. Es wird ein `Created`-Object zurückgegeben, das die Return-Werte der `create`-Methode enthält. Dabei erhält eine Transaktion u.a. eine für den jeweiligen Manager eindeutige ID.

Mit Hilfe der Methoden `commit` und `abort` können die im obigen Modell beschriebenen Funktionen zur Steuerung der Transaktion ausgeführt werden. Die Methode `getState` liefert den aktuellen Status der Transaktion zurück, und `join` dient einem Transaktionsteilnehmer dazu, einer Transaktion, die von diesem Transaktionsmanager verwaltet wird, beizutreten.

Das Interface `Transaction` selbst, das zur Abwicklung einer Transaktion implementiert werden muss, enthält wiederum nur die bereits beschriebenen Methoden `abort` und `commit`. Auf weitere Erklärungen wird deshalb verzichtet. Eine Klasse, die dieses Interface bereits implementiert, ist `net.jini.core.transaction.server.ServerTransaction`.

Um Subtransaktionen ausführen zu können, muss das Interface `NestableTransaction` implementiert werden.

4.2.2 Transaktionsmanager

Wie bereits mehrfach erwähnt, werden Transaktionen in Jini durch den sog. Transaktionsmanager verwaltet. Dieser muss das Interface `net.jini.core.transaction.server.TransactionManager` (vgl. 4.2.1) implementieren. Zur Verwaltung von Subtransaktionen ist dagegen die Implementation des Interfaces `net.jini.core.transaction.server.NestableTransactionManager` notwendig.

Alle weiteren Funktionen des Managers entsprechen dem oben beschriebenen Modell.

4.2.3 Transaktionsteilnehmer

Alle Teilnehmer an einer Transaktion müssen das Interface `net.jini.core.transaction.server.TransactionParticipant` implementieren. Diese stellt folgende Funktionen zur Verfügung[6]:

[6] [JINI, 2000, „Interface TransactionParticipant“]

```
public interface TransactionParticipant
extends java.rmi.Remote, TransactionConstants
{
    void abort(TransactionManager mgr, long id);
    void commit(TransactionManager mgr, long id);
    int prepare(TransactionManager mgr, long id);
    int prepareAndCommit(TransactionManager mgr, long id);
}
```

Die Methoden `abort` und `commit` haben dabei die schon mehrfach beschriebenen Funktionen. Mittels `prepare` kann ein Teilnehmer aufgefordert werden, in die erste Phase des Commits einzutreten (s.o.), während `prepareAndCommit` eine Kombination der beiden namensgebenden Methoden darstellt, die benutzt werden kann, wenn alle anderen Participants bereits „*OK*“ gemeldet haben und nur noch die Antwort eines Teilnehmers fehlt.

Die oben aufgeführten Methoden unterstützen den Programmierer nur bei der Behandlung der Problematik eines Serverausfalls. Jini bietet zwar Möglichkeiten, einen Ausfall zu erkennen und den davor vorhandenen Systemstatus abzufragen, die konkrete Implementierung ist aber Sache des Programmierers, dem dafür sehr viele Freiheiten bei der Gestaltung der persistenten Speicherung gewährt werden[7].

5 Fazit

Wie dargestellt, bietet Jini mannigfaltige Funktionen an, die die Programmierung innerhalb eines verteilten Systems erleichtern können. Dennoch bleibt dem Programmierer viel Arbeit, da oft nur Interfaces existieren, deren Funktionen implementiert werden müssen. Insofern ist Jini mehr als ein Baukasten zur Implementierung verteilter Systeme anzusehen, was sicherlich beim Design aber auch beabsichtigt wurde, um dem Programmierer größtmögliche Freiheiten zu gewähren.

[7] [Bader, 2000, S. 35]

6 Literaturverzeichnis

[Bader, 2000] Bader, H.; Huber, W..: *Jini – Die intelligente Netzwerkarchitektur in Theorie und Praxis*. München 2000

[JINI, 2000] n.n.: *Jini™ Technology 1.1 API Documentation*. Palo Alto 2000

[SUN, 2000] n.n.: *Java™ 2 Platform, Standard Edition, v 1.3, API Specification*. Palo Alto 2000