In diesem Kapitel:

- *Synchronisation mit JavaSpaces*
  - Koordination von Jini Applikationen mit JavaSPaces
  - Beispiel: ein Multi User Game
- *Transaktionen*
  - ACID Prinzip in Jini Applikationen
  - Beispiel: ein JavaSpaces Relay
- *Fehlertoleranz und Skalierbarkeit*
  - Fehlertoleranz
    - *Transaktionen als Hilfsmittel zur Erhöhung der Fehlertoleranz*
    - *Beispiel: write() und take() als Transaktionen*
  - Skalierbarkeit
    - *Mehrere Spaces als Hilfsmittel zum Erreichen von Skalierbarkeit*
    - *Beispiel: Verteilter Rechner mit mehreren Spaces*
- *Verteilte Datenstrukturen*
  - Das Bag Design Pattern
  - Das Channel Design Pattern

Java Spaces - Praktische Beispiele

1.1. JavaSpaces Praxis

Im Folgenden wollen wir an einzelnen komplexeren Aufgabenstellungen zeigen, wie JavaSpaces zur Lösung komplexerer Aufgaben eingesetzt werden können.

Dabei geht es um Themen wie
- Synchronisation / Koordination
- Transaktionen
- ...

jeweils mit Hilfe von JavaSpaces und Jini Services realisiert.

1.2. Synchronisation

1.2.1.1. Übersicht

Bisher haben wir uns lediglich mit den grundsätzlichen Konstrukten von JavaSpaces befasst. Nun wollen wir JavaSpaces mit Jini kombinieren und das dynamische Laden von Klassen, welches in Jini (und RMI) häufig eingesetzt wird, auch in JavaSpaces Beispielen nutzen.

Nun versuchen wir an einem einfachen Beispiel, wie JavaSpaces mit Jini kombiniert werden können und damit Jini Applikationen koordiniert werden können.

1.2.1.2. Koodination von Jini Applikationen mit JavaSpaces

In vorigen Kapitel haben wir einfach gesehen, wie JavaSpaces funktioniert. JavaSpaces sind netzwerkbasierte Objektspeicher und Objektaustauschkanäle, mit deren Hilfe Prozesse kommunizieren und ihre Aktivitäten synchronisieren können. Koordination ist deine entscheidende Komponenten verteilter Programme.

Diese Probleme kennen wir aus dem täglichen Leben:
- Fahrzeuge, die eine Kreuzung überqueren
- Arbeiter, welche Geräte zusammenbauen
- Spiele (Basket, Fussball, ...)

sind Aktivitäten, welche eine Koordination benötigen.

Verteilte Anwendungen und Prozesse müssen synchronisiert werden, um eine gemeinsame Aufgabe erfüllen zu können. Dazu kommt häufig das Problem, dass mehrere Prozesse auf die selbe, limitierte Ressource zugreifen müssen.

Zur Illustration setzen wir Jini ein und versuchen ein einfaches Multi-User-Game zu realisieren. JavaSpaces können sehr gut zur Kommunikation und Koordination zwischen Java Föderationen (den Entities in Jini Föderationen) eingesetzt werden. Beispielsweise können wir JavaSpaces zur Koordination schnell veränderlicher Jini Umgebungen, Umgebungen also, in denen Entities verfügbar und unverfügbar werden, eingesetzt werden.

Ziel dieses Abschnitts ist es, zu zeigen, wie mit JavaSpaces eine einfache Form der verteilten Synchronisation erreicht werden kann. Als Beispiel dient ein Multi-User-Game. Dieses steht als Jini Dienst zur Verfügung. Es stehen jedoch lediglich eine limitierte Anzahl Plätze (also Spieler) zur Verfügung.

JavaSpaces garantiert Synchronisation auf der Entry Ebene. Durch geschickten Einsatz der Entries können wir also Zugriffe und genereller Aktivitäten synchronisieren.

1.2.1.3. JavaSpaces Operationen und Synchronisation

Koordination kann harte Arbeit sein. Falls eine Applikation lediglich auf einer Maschine läuft, kann das Betriebsystem als zentraler Manager mehrere Threads koordinieren. In einem Netzwerk hingegen, besteht kein zentraler Koordinationspunkt mehr. Die Prozesse laufen auf unterschiedlichen Rechnern, unterschiedlichen Betriebsystemen und unterschiedlichen internen Darstellungen der Basisdatentypen. Falls Sie keinen zentralen Kordinator definieren oder entwickeln, welcher auch eine Gefahr (wegen Ausfällen) darstellen würde, müssen Sie die gegenseitigen Zugriffe der Prozesse auf gemeinsame Ressourcen irgendwie koordinieren. Und dies ist ungemein schwieriger als in zentralen Systemen.

JavaSpaces gestatten auf einfache Art und Weise die Synchronisation verteilter Prozesse, weil die Synchronisation bereits in die Space Operationen eingebaut ist:

- `write`
  schreibt ein Objekt, eine Entry, in ein Space
- `read`
  kopiert eine Entry und lässt das Original im Space unverändert
- `take`
  entfernt eine Entry aus einem Space

Schauen wir einmal, wie mit diesen einfachen Operationen synchronisiert werden kann:

Annahmen: wir haben ein JavaSpace und wollen ein Message Objekt als Entry darin abspeichern. Das Message Objekt wird folgendermassen definiert:

```
public class Message implements Entry {
      public String content;
      public Message() { }
}
```

Nun instanzieren wir eine Message Entry und definieren deren Inhalt:

```
Message msg = new Message();
msg.content = "Mir geht's gut! Wie geht's Dir?";
```

Unser JavaSpace Objekt heisst `space`. Wir rufen dessen `write()` Methode auf, um eine Kopie unserer Message Entry in diesem Space zu speichern:

```
space.write(msg, null, Lease.FOREVER);
```

Nun können andere Prozesse auf dieses Objekt im Space zugreifen und diese Entry lesen:

```
Message template = new Message();
Message result = (Message)space.read(template, null, Long.MAX_VALUE);
```

Lesen können mehrere Prozesse gleichzeitig, da das Original nicht verändert wird. Aber falls ein Prozess diese Entry aktualisieren möchte, muss zuerst die alte Entry aus dem Space entfernt, die Änderung durchgeführt und schliesslich das veränderte Objekt, die Entry, wieder in den Space geschrieben werden.

```
Message result = (Message)space.take(template, null, Long.MAX_VALUE);
result.content = "Mir geht's auch gut! Aber Zucollini geht's schlecht!";
space.write(result, null, Lease.FOREVER);
```

Wichtig ist, dass diese Operationen exklusive erledigt werden können. Falls mehrere Prozesse auf die selbe Entry verändernd zugreifen würden, könnte schnell eine Dateninkonsistenz entstehen. Die `take` Operation setzt ein Lock. Alle anderen, späteren `take()` Aufrufe müssen warten, bis der erste Aufruf abgeschlossen wurde.

Wir haben somit folgendes Grundmuster für die Synchronisation mittels Spaces:
- lesen einer Entry ist jederzeit möglich, auch von mehreren Prozessen
- mutieren einer Entry ist jeweils nur durch genau einen Prozess möglich.
  Dabei wird die Entry zuerst aus dem Space entfernt;
  dann mutiert unddamit exklusive verfügbar
  und schliesslich wird die veränderte Entry in den Space zurück geschrieben.

Mit anderen Worten:
`read`, `take` und `write` Operationen erzwingen einen koordinierten Zugriff auf Entries.

Gleichzeitig muss man sich auch darüber im klaren sein, dass synchronisierte Operationen zeitaufwendiger sind, als nicht-synchronisierte. Aber bei JavaSpaces Anwendungen spielt die Zeit in der Regel nicht die zentrale Rolle: die Operationen sind insgesamt recht zeitaufwemdig.

1.2.1.4. Ein Jini Spielservice

Im Folgenden betrachten wir einen Jini Multiuser Spieleserver, bei dem mehrere Spieler gleichzeitig beteiligt sein können. Allerdings kann eine maximale Anzahl Spieler beim Start vorgegeben werden. Falls diese Anzahl erreicht wird, kann ein weiterer Spieler erst am Spiel teilnehmen, sobald einer der aktiven Spieler das Spiel verlässt.

In seiner einfachsten Form akzeptiert das Spiel einen beliebigen Spieler aus der Warteschlange. Möglich wäre eine trickreichere Verwaltung der interessierten, wartenden Spieler. Aber dies würde lediglich das Beispiel komplexer werden lassen.

Die Skizze unten zeigt, wie das gesamte Spiel etwa ablaufen könnte. Das Jini Spiel verfügt über zwei grundlegende Methoden:

- `joinGame()` und
- `leaveGame()`

Falls ein Spieler am Spiel teilnehmen möchte, sucht er als erstes einen Spielserver, einen Spiele Service Provider, in Jini geschieht dies mit Hilfe eines Lookup Services, und liefert dann ein Proxy Objekt an den Spieler. Mit diesem Proxy Objekt kann der Spieler `joinGame()` aufrufen und einem Spiel beitreten, sich an einem Spiel beteiligen. Falls die Methode erfolgreich ist, dann liefert sie ein Interface zum entfernten Spiele Objekt, mit dessen Hile gespielt werden kann.

Der Spieler ruft die `play()` Methode des `game` Objekts auf. Falls der Spieler diese Methode verlässt, beendet der Spieler das Spiel und ruft die Methode leaveGame() des Proxy-Objekts auf. In unserem Setup wird der selbe Spieler nach einer kurzen Wartezeit dem Spiel wieder beitreten, mit `joinGame()`.

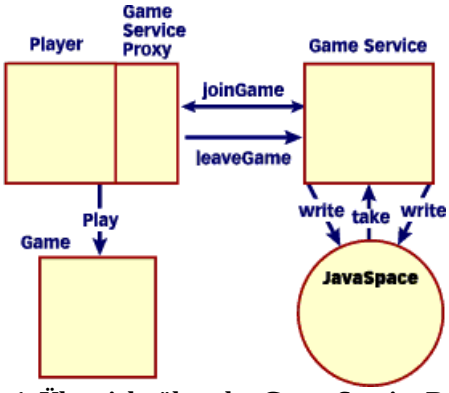


Figure 1. Übersicht über das Game Service Beispiel

Die Programm können mit den Batch Prozeduren gestartet werden. Sie können den Namen des Spieles (einfach eine Zeichenkette) und die maximale Anzahl Spieler angeben.

Fall Sie mehrere Spieler starten, müssen Sie jeden durch seinen Namen kennzeichnen. Falls Sie weniger Spieler starten, als Plätze vorhanden sind, können alle gleichzeitig starten, spielen, pausieren und wieder starten... Falls Sie mehr Spieler starten als Spieler gleichzeitig erlaubt sind, müssen Sie den Zugriff auf das Spiel koordinieren.

1.2.1.4.1. Der Spieler

Als erstes schauen wir uns den Player, den Spieler an. Hier sehen Sie lediglich ein Skelett. Das vollständige Programm finden Sie auf dem Server / der CD.

```
public class Player implements Runnable { //Applet
protected String gameName;
protected String myName;
...
//Konstruktor
public Player(String gameName, String myName) throws IOException {
      this.gameName = gameName; this.myName = myName;
...
// kreiere ein Template, um ein GameServiceInterface zu finden
// setzen eines SecurityManagers
// starten eines Listeners für Discovery Events in der Jini public Gruppe
...
}
// diese Methode wird aufgerufen,
// falls ein neuer Lookup Service gefunden wird
protected void lookForService(ServiceRegistrar lookupService) {
...
// Suche (mit einem Template) nach Proxies
// welche das GameServiceInterface implementieren
...
      GameServiceInterface gameInterface = (GameServiceInterface)proxy;
      while (true) {
            Game game = gameInterface.joinGame(gameName);
            if (game != null) {
                  try {
                        System.out.println("Spiel :" + gameName);
                        game.play(myName);
                  } catch (RemoteException e) { e.printStackTrace(); }
                  gameInterface.leaveGame();
            } else {
                  System.out.println("Das Spiel " + gameName+
                                          " ist nicht erreichbar");
            }
            // 10 Sekunden warten;dann versuchen, am Spiel teilzunehmen
            try {
                  Thread.sleep(10000);
            } catch (Exception e) { e.printStackTrace(); }
      }
}
// kreiere einen Player und starte seinen Thread
public static void main(String args[]) {
      if (args.length < 2) {
            System.out.println("Usage: Player gameName playerName");
            System.exit(1);
      }
      try {
            Player player = new Player(args[0], args[1]);
            new Thread(player).start(); // starte schlafenden Thread
      } catch (IOException e) {
      System.out.println("Fehler beim Kreieren des Spielers: " +
                                                      e.getMessage());
      }
  }
}
```

In der `main()` Methode wird ein neuer Spieler (`Player` Objekt) kreiert. Als Argument erhält dessen Konstruktor den Namen des Spiels, bei dem der Spieler mitspielen möchte, sowie einen Namen, unter dem der Spieler bekannt sein wird. Der Konstruktor kümmert sich auch um die Details des Jini Dienstes und Lookup Prozesses, beispielsweise ein Template zum Suchen des `GameServiceInterface`, der Definition eines Security Managers und der Registration eines Listeners, der den Spieler über Jini Discovery Events informieren soll.

Danach wird der Thread offiziell schlafende gesetzt. In Wirklichkeit wartet er nun auf ein Discovery Event, um sich einem Spiel anschliessen zu können. Falls der Thread nicht schlafen würde, wäre das Programm zu Ende bevor es richtig gestartet wurde.

Falls ein Lookup Dienst gefunden wird, wird die `lookupForService()` Methode aufgerufen. Diese Methode verwendet ein Template, um Proxies zu finden, welche das `GameServiceInterface` implementieren. Sobald ein Proxy gefunden wird, beginnt eine Schleife, in der Spieler zuerst die Proxy-Methode `joinGame()` aufrufen und eventuell darauf warten, zum Spiel zugelassen zu werden. Sobald der Spieler zun Spiel zugelassen wurde, erhält er ein remote Objekt von der Methode. Der Spieler ruft dessen `play()` Methode auf, um sein Spiel zu starten. Sobald er mit dem Spiel fertig ist, ruft er die Methode `leaveGame()` auf. Diese gibt im Wesentlichen das Slot des Spielers frei für einen anderen Spieler. Der Spieler schläft dann 10 Sekunden und versucht nachher erneut am Spiel teilzunehmen.

1.2.1.4.2. Der Spiel Service

Alle Jini Funktionalitäten, die wir für den Spiel Service benötigen, inklusive dem Auffinden der Lookup Services und dem Publizieren des Proxies, mit dem die Verbindung zum Spiel aufgebaut wird, werden vom `GameService` erledigt:

```
public class GameService implements Runnable { //Thread
      protected JavaSpace space;
      protected Game game;
      ...

      // Methode, mit deren Hilfe maxPlayers Tickets für das Spiel dem
      // Space hinzugefügt werden

      private void createTickets(String name, int maxPlayers) {
            // instanziere ein Spiel mit dem angegebenen Namen
            try {
                  game = new GameImpl(name);
            } catch (RemoteException e) { e.printStackTrace(); }

            // schreiben der Tickets
            for (int i = 0; i < maxPlayers; i++) {
                  Ticket ticket = new Ticket(name, game);
                  try {
                        space.write(ticket, null, Lease.FOREVER);
                  } catch (Exception e) { e.printStackTrace(); }
            }
      }

      // Methode zum Kreieren des Proxy Objekts,
      // welches das GameServiceInterface implementiert

      protected GameServiceInterface createProxy() {
            GameServiceInterface gameServiceProxy = new GameServiceProxy();
            gameServiceProxy.setSpace(space);
            return gameServiceProxy;
      }

      // Konstruktor
      public GameService(String gameName, int numTickets) throws
                                                      IOException {
            space = SpaceAccessor.getSpace();
            item = new ServiceItem(null, createProxy(), null);
            // ...
            // Security Manager
            // Definition des Listener für Discovery Events
            // ...
            // kreiere die Anzahl Tickets für das best. Spiel
            createTickets(gameName, numTickets);
      }
```

```
      public static void main(String args[]) {
            if (args.length < 2) {
              System.out.println("Usage: GameService gameName numTickets");
             System.exit(1);
            }
            try {
                  String gameName = args[0];
                  int numPlayers = Integer.parseInt(args[1]);
                  GameService gameService = new GameService(gameName,
                                                            numPlayers);
                  new Thread(gameService).start();
            } catch (Exception e) {
                  System.out.println("Spiel Service konnte nicht kreiert
                                    werden: " + e.getMessage());
                  e.printStackTrace();
            }
      }
}
```

In der `main()` Methode wird eine neue Instanz des Spiele Services (`GameService`) kreiert. Als Parameter verwendet der Konstruktor den Namen des Spieles und die Anzahl gleichzeitig möglicher Spieler. Damit das Objekt am Leben bleibt, wird anschliessend eine Endlosschleife gestartet.

Wichtig ist auch, was konkret im Konstruktor geschieht:

- als erstes erhalten Sie eine Referenz auf ein JavaSpace Objekt. Diese Arbeit wird von der hier nicht diskutierten Klasse `SpaceAcessor` (siehe unten) erledigt. Mit dem so erhaltenen Space Objekt werden die Spieler und das Spiel koordiniert.

- dann wird ein `ServiceItem` kreiert, ein Objekt, welches bei den Lookup Services eingetragen wird. Beim Kreieren wird die `createProxy()` Methode aufgerufen, welche die Klasse `GameServiceProxy` instanziert. Diese schafft eine Verbindung zum kreierten JavaSpace und liefert ein Proxy Objekt. Das Proxy Objekt wird ins Service Item eingebettet und wird zur Verbindungsstelle für den Spieler zum Spiel.

- die Jini Schnittstellen werden passend für das Spiel definiert. Wichtig ist, dass immer wenn ein Jini Discovery Event geschieht, sich der Spiele Service mit dem Proxy bei neuen Lokup Services einträgt. Der Dienst trägt sich also wenn immer möglich ein.

- schliesslich ruft der Konstruktor die Methode `createTicket()` auf, mit der spezifiziert wird, wieviele Spieler gleichzeitig spielen dürfen.

Die Koordinations- oder Synchronisations- Idee besteht also eigentlich in den Tickets: es werden nur Spieler beim Spiel zugelassen, welche über ein Ticket verfügen (wie im wahren Leben).

Bevor wir uns mit den Details des Kreierens der Tickets befassen, müssen wir die Klasse `Ticket` verstehen:

```
import net.jini.core.entry.Entry;
public class Ticket implements Entry {

	public String gameName;
	public Game game;

	public Ticket() { }

	public Ticket(String gameName) {
		this.gameName = gameName;
	}

	public Ticket(String gameName, Game game) {
		this.gameName = gameName;
		this.game = game;
	}
}
```

Jedes Ticket enthält zwei Datenfelder:
1) der Name des Spiels
2) eine Referenz auf das Spiel, ein remote Objekt `game`. Damit kann der Spieler am Spiel teilnehmen.

Der leere Konstruktor der `Ticket` Klasse wird wegen JavaSpaces benötigt.

Nun können wir uns anschauen, wie die Methode createTickets() funktioniert:
- zuerst wird ein remote Game Objekt kreiert und der remote Referenz zugeordnet
- mit dieser Referenz wird werden die Tickets kreiert, jeweils mit dem Namen des Spiels und der remote Referenz auf das Spiel und trägt diese in den JavaSpace ein.

Damit steht die Infrastruktur für das Spiel und die ganze Action kann starten.

1.2.1.4.3. Das remote Game

Das Spiel selbst ist eher uninteressant. Es geht ja eigentlich auch n icht ums Spiel! Das Spiel besitzt folgendes Interface (`Game.java`):

```
import java.rmi.*;

public interface Game extends Remote {
      public void play(String playerName) throws RemoteException;
}
```

Das Interface wird in der Klasse GameImpl implementiert (in `GameImpl.java`):

```
import java.rmi.*;
import java.rmi.server.*;

public class GameImpl extends UnicastRemoteObject implements Game {

      private String name;

      public GameImpl(String name) throws RemoteException {
            super();
            this.name = name;
            // Create and install a security manager
            if (System.getSecurityManager() == null) {
                  System.setSecurityManager(new RMISecurityManager());
            }
      }

      // play gibt lediglich eine Meldung aus und schläft dann

      public void play(String playerName) {
            for (int i = 0; i < 5; i++) {
                  System.out.println(playerName + " ist am spielen...");
                  try {
                        Thread.sleep(2000);
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
            }
      }
}
```

Das Beispiel setzt voraus, dass Sie wissen, wie man mit Remote Objekten umgeht. Details finden Sie in den Kursunterlagen zu RMI.

Nett wäre ein Ausbau der `play()` Methode:
- diese könnte beispielsweise ein echtes Spiel starten
- die Informationen der einzelnen Spieler könnten mittels JavaSpace an die anderen Teilnehmer kommuniziert werden.

Alles was noch fehlt, ist der Mechanismus, mit dem einem Spiel mittels Proxy beigetreten wird. Darauf kommen wir jetzt :

1.2.1.4.4. Der Game Service Proxy

Das Game Service Interface (in `GameServiceInterface.java`) sieht folgendermassen aus:

```
public interface GameServiceInterface {
      public void setSpace(JavaSpace space);
      public Game joinGame(String name);
      public void leaveGame();
}
```

Die `GameServiceProxy` Klasse (defininiert in `GameService.java`) implementiert das Interface:

```
class GameServiceProxy implements Serializable, GameServiceInterface {
      private JavaSpace space; private String name; private Game game;

      public GameServiceProxy() {
      }

      public void setSpace(JavaSpace space) {
            this.space = space;
      }

      // hol Dir ein Ticket!
      public Game joinGame(String name) {
            this.name = name;
            System.out.println("Suche ein Ticket für " + name);
            Ticket ticketTemplate = new Ticket(name);
            Ticket ticket;
            try {
                  ticket = (Ticket)space.take(ticketTemplate, null,
                                                      Long.MAX_VALUE);
                  this.game = ticket.game;
                  System.out.println("Ich habe ein Ticket für " + name);
            } catch (Exception e) { e.printStackTrace();
            }
            return game;
      }

      // Beim Verlassen des Spieles -> Ticket zurück

      public void leaveGame() {
            if (game == null) {
                  System.out.println("Ich bin nicht im Spiel!");
            } else {
                  Ticket ticketEntry = new Ticket(name, game);
                  try {
                        space.write(ticketEntry, null, Lease.FOREVER);
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
                  System.out.println("Habe das Spiel "+name+" verlassen");
                  name = null;
                  game = null;
                  }
            }
}
```

Bisher haben wir lediglich den Proxy des Spielservice instanziert und seine `setSpace()` Methode aufgerufen. Damit konnten wir eine Verbindung zum JavaSpace herstellen, in dem die Tickets eingetragen sind.

Die Spieler holen sich in einer Endlosschlaufe ein Ticket, spielen eine Weile, geben das Ticket zurück, schlafen eine Weile und beginnen dann wieder von vorne. Zuvor muss ein Spielservice gesucht werden. Dafür wied ein Lookup Service (von Jini) eingesetzt.

Nun geht es um das Beitreten und Verlassen des Spieles.

Die `joinGame()` Methode akzeptiert den Namen des Spielers, welcher bei einem bestimmten Spiel mitmachen will und speichert diesen für später in einer Variable. Dann wird ein Entry Template kreiert, mit dessen Hilfe (über den Namen) das Spiel gesucht wird. Mit diesem Template kann ein Ticket aus dem JavaSpace geholt werden. Falls aber alle Spielplätze besetzt sind, ist kein Ticket mehr erhältlich. In diesem Fall wartet die Methode `take()` bis endlich ein Ticket im JavaSpace vorhanden ist.

Sobald der Spieler ein Ticket bekommt, kann er darauf nachsehen, wo sich das Spiel befindet: auf dem Ticket befindet sich eine remote Referenz in der Variable `game`. Diese Referenz wird auch beim Methodenaufruf `joinGame()` zurückgeliefert. Im Ticket-Pool wird die Anzahl verfügbarer Tickets um eins reduziert, bis der Spieler nicht mehr spielen möchte und das Ticket zurückgibt.

Da das Spiel äusserst wenig Koordinationsaufwand (auf Seite des programmierers) hat, könnten Sie sehr leicht beliebig viele Spieler berücksichtigen, die Lösung ist also skalierbar.
.

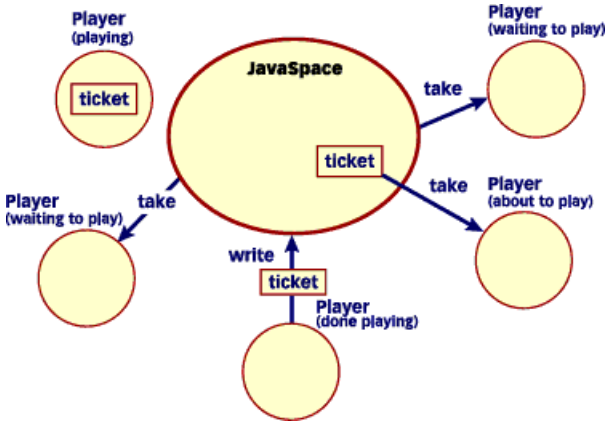


Figure 2. Koordination des Spielens mit einem JavaSpace

1.2.1.4.5. Starten des Beispiels

Auf dem Server / der CD finden Sie Batch prozeduren, welche die einzelnen Dienste und Programme starten. Das Beispiel produziert folgende Ausgaben:

serverseitig:

```
[GameService]Start
[GameService]Spiel : Jini_JavaSpaces Spiel max. Anzahl Spieler : 3
found JavaSpaces = com.sun.jini.outrigger.SpaceProxy@0
Ein neuer Lookup Service wurde gefunden
serviceID : 95a9dff4-e557-4135-a12c-83ccbe748241
[GameService]Ende
[GameImpl]playerName Peter ist am spielen...
[GameImpl]playerName Joanna ist am spielen...
[GameImpl]playerName Peter ist am spielen...
[GameImpl]playerName Joanna ist am spielen...
[GameImpl]playerName Peter ist am spielen...
[GameImpl]playerName Joanna ist am spielen...
[GameImpl]playerName Peter ist am spielen...
[GameImpl]playerName Joanna ist am spielen...
[GameImpl]playerName Nicole ist am spielen...
[GameImpl]playerName Britta ist am spielen...
...
```

clientseitig:

```
[Player]main()
[Player]Player(Jini_JavaSpaces Spiel,Jens)
[Player]run()
[DiscoveryListener<-Player]discovered()
[Player]lookForService(com.sun.jini.reggie.RegistrarProxy@e738ad09)
Es wurde ein passender Service gefunden.
Ein Ticket bitte Jini_JavaSpaces Spiel
Ich habe ein Ticket fuer Jini_JavaSpaces Spiel
Spieler ist aktiv: Jini_JavaSpaces Spiel
Ich verlasse das Spiel Jini_JavaSpaces Spiel
Ein Ticket bitte Jini_JavaSpaces Spiel
...
```

1.2.1.5. Vorteile von JavaSpaces zur Lösung von Koordinationsaufgaben

In diesem Beispiel wurde ein Jini Service eingesetzt, um limitierte Ressourcen (die Spielplätze) zu verwalten. Wie Sie aus dem Programm erkennen können, wurde sehr wenig JavaSpaces spezifischer Programmcode verwendet. Allerdings haben wir viele Aspekte nicht berücksichtigt: Fairness, ein sinnvolles Scheduling usw.

Was auch völlig unberücksichtig blieb sind Fragen der Skalierbarkeit und der Zuverlässigkeit / Fehlertoleranz.

Falls wir den Spielserver erweitern möchten und beispielsweise verschiedene Spiele zulassen möchten, würde sich ein spezieller "Ticket Corner", eine spezielle Klasse oder sogar ein Dienst anbieten, welcher die zusätzliche Funktionalität zur Verfügung stellt.

JavaSpaces gestatten auf leichte Art und Weise das Zusammenspiel mit Jini Diensten, wie wir im Beispiel gesehen haben. Falls Sie beispielsweise Ihre Spielwiese ausbauen und mehr Tickets verkaufen wollen, müssen Sie einfach mehr Tickets in den Space stellen.

1.2.1.6. Zusammenfassung

Dieses Beispiel zeigt Ihnen die Flexibilität der JavaSpaces und wie leicht sich einzelne Prozesse koordinieren lassen. Mehrere Aspekte, wie Leasing, Transaktionen und die Fehlertoleranz werden Sie gleich kennen lernen.

1.3. Transactions - Jini Transaktionen in JavaSpaces

1.3.1. Einleitung

Jini Transaktionen gestatten es, verteilte Applikationen zu entwickeln, welche auch noch korrekt funktionieren, wenn Teile des Systems ausfallen. Jini Transaktionen verfügen über diese Fähigkeit. Allerdings werden Jini Transaktionen kaum eingesetzt.

In diesem Abschnitt wollen wir uns anschauen, wie Jini Transaktionen mit JavaSpaces kombiniert werden können und dieses Konzept auf ein einfaches Beispiel anwenden.

Die Grundidee von Transaktionssystemen besteht darin, dass mehrere Operationen zusammengefasst werden können und damit atomar werden: sie werden entweder ganz oder überhaupt nicht ausgeführt. Falls Transaktionen nicht atomar wären, könnte beispielsweise im Verlaufe einer Lese-, Mutations- und Update-Folge nach dem Lesen jemand die Daten verändern. Damit würden die Daten inkonsistent. In verteilten Systemen kann fast alles schief gehen (Kommunikationsstörungen, Serverausfall, Clientausfall, ...). Daher muss speziell in verteilten Systemen auf Konsistenz geachtet werden. Transaktionen sind ein wertvolles Werkzeug zur Realisierung konsistenzerhaltender Systeme.

Datenbanksysteme verfügen in der Regel über komplexe mehrphasige Transaktionsprotokolle. Jini ist dagegen recht leichtgewichtig, berücksichtigt aber die wesentlichen Elemente. Die Teilnehmer einer Jini Transaktion sind typischerweise Jini Services und Devices. Dazu muss einfach das Jini `TransactionParticipant` Interface implementiert werden.

Transactionen in Jini oder JavaSpaces haben allerdings ein spezielles Problem: sie betreffen in der Regel verteilte Systeme und somit jede Menge möglicher Ausfälle in komplexen Umgebungen (Hardware, Software, ...) und kritische Zeitkonstanten.

In Jini Netzwerken will man in der Regel keinen zentralen Kontrollservice. Daher kann man auch bei Jini Netzwerken versuchen Ausfallsicherheitsmechanismen einzubauen, welche zum Teil bei Datenbanken kaum nötig oder nur schwer realisierbar sind.

1.3.2. Transaktionen und JavaSpaces

Das JavaSpaces Applikations Programmier Interface (API) integriert Jini Transaktionen. Daher können Sie Transaktionen recht problemlos in JavaSpaces Applikationen integrieren.

Um eine Transaktion zu definieren, wird man zuerst einen Transaktionsmanager Service aktivieren müssen. UnterUmständen sind an einer einzigen Transaktion mehrere Spaces beteiligt.

Schreiben von Objekten in ein Space mittels Transaktionen geschehen entweder ganz oder überhaupt nicht. Es kann auch vorkommen, das der Transaktionsmanager eine Transaktion abbricht, beispielsweise weil die Lease Zeit abgelaufen ist, und alle Aktivitäten rückgängig macht. Entries in Transaktionen werden für Space Clients erst sichtbar, falls die Transaktion erfolgreich abgeschlossen werden kann.

Beginnen wir mit der `write()` Methode:

```
space.write(Entry entry, Transaction txn, Lease lease);
```

Diese Methode schreibt die Entry `entry` Space, mit einer Lease Zeit `lease` und mittels des Transaktionsmanagers `txn`.

Falls Sie für `txn` `null` verwenden, besagt dies, dass lediglich eine Operation ausgeführt werden soll, diese Operation. Mit dem `null` Argument steht ein Objekt nach dem Schreiben den Spaces Clients gleich zur Verfügung. Falls Sie einen Transaktionsmanager verwenden, stehen die Objekte erst zur Verfügung nachdem die Transaktionen abgeschlossen sind.

Nun betrachten wir `take()` und `read()`. Beide verwenden ein Template und liefern alle Entries, welche dem Template genügen. `take()` entfernt eine Entry bevor diese an den Aufrufenden übergeben wird. Beim Lesen wird einfach eine Kopie des Objekts geliefert.

1.3.3. Einsatz eines Transaktionsmanagers

Um Transaktionen einsetzen zu können, muss man einen Transaktionsmanager zur Verfügung haben, mit dem man Transaktionen kreieren und unterhalten bzw. ausführen kann. In unserem Fall setzen wir den Jini Lookup und Discovery Manager ein, um einen Transaktionsmanager zu suchen. In diesem Fall suchen wir einen Service, welcher das Interface `TransactionManager` implementiert. Dazu setzen wir eine Hilfsklasse aus dem Buch *JavaSpaces Principles, Patterns and Practice* ein, welche eine Referenz auf den Transaction-Manager Proxy liefert. Im Programm selber genügt eine Zeile:

```
TransactionManager mgr = TransactionManagerAccessor.getManager();
```

Die Methode `getManager()` ist static und liefert ein `TransactionManager` Proxy Objekt. Mit diesem Proxy können Sie eine Transaktion kreieren, welche eine oder mehrere Operationen bei einem oder mehreren Jini Services (oder JavaSpaces) ausführen kann, sofern dieser das Interface `TransactionParticipant` implementiert.

```
Transaction.Created trc = null; //innere Klasse
try {
      trc = TransactionFactory.create(mgr, 300000);// 5 Minuten
} catch (Exception e) {
 System.err.println("Transaktion konnte nicht kreiert werden "+e);
}
```

In der ersten Anweisung wird ein Objekt vom Datentyp `Transaction.Created`, einer inneren Klasse von `Transaction`, kreiert. Dieser Objekttyp wird vom Transaktionsmanager an das Programmfragment zurückgeliefert. Die Transaktion selber wird mit der `TransactionFactory`, speziell deren statischer Methode `create()`, erzeugt. Als Parameter wird, neben dem Transaktionsmanager, die Leasingzeit angegeben, im Millisekunden.
Die innere Klasse der Transaction Klasse sieht folgendermassen aus:

```
public static class Created implements Serializable {
      public final Transaction transaction;
      public final Lease lease;
      Created(Transaction transaction, Lease lease) {...}
}
```

Die Klasse enthält zwei öffentlich zugängliche Datenfelder, `transaction` und `lease`, und einen Konstruktor. Mit dieser Klasse kann man die zwei Return Objekte, die Transaktion und die Leasezeit. Mit der obigen "Hilfsklasse" kann man diese zusammenfassen.

Sie erhalten mit Hilfe dieser Klasse die Transaktion:

```
:
Transaction txn = trc.transaction;
```

und analog dazu kann man die Leaseingdauer erfragen. Leasingdauer bei Transaktionen ist die Zeitdauer, während der der Transaktionsmanager die Transaktion überwachen wird. Falls diese Zeitdauer überschritten wird, wird die Transaktion abgebrochen. Bei verteilten Systemen kann dies durchaus sinnvoll sein.

1.3.4. Ein Beispiel

Als Beispiel für eine Transaktion betrachten wir ein einfaches Space Relay: ein Programm, welches die Nachrichten in ein Source Space einträgt und dann die sich darin befindlichen Messages in ein oder mehrere Target Spaces verschiebt. Die Verschiebung geschieht in Form von Transaktionen: das Entfernen aus dem Source Space und die Eintragungen in den Target Spaces geschieht entweder ganz oder nicht.

Als erstes müssen wir eine Message definieren:

```
package javaspacestransaktionen;

import net.jini.core.entry.Entry;
public class Message implements Entry {
    public String content;

    // Default Konstruktor, wegen der Serialisierung
    public Message() {
    }
}
```

Dann benötigen wir Programmcode, um die Messages in den Source Space zu schreiben:

```
     /* schreibe numMessages Message Entries in den Source Space
     */
    private void createMessages() {

        for (int i = 0; i < numMessages; i++) {

            Message msg = new Message();

            msg.content = "" + i;

            try {

                sourceSpace.write(msg, null, Lease.FOREVER);
                System.out.println("[SpaceRelay.createMessages()]Schreibe
                                    Message " + i + " in " + sourceName);

            } catch (Exception e) {
                System.err.println("[SpaceRelay.createMessages()]Fehler
                        beim Schreiben der " + i + "'ten Message: " + e);

            }

        }

    }
```

Die Methode schreibt `numMessages` in den Source Space. Als Meldung wird einfach die Meldungsnummer verwendet. Diesen Prozess sichern wir nicht transaktionell ab.

Als nächstes kopieren wir diese Messages in die Target Spaces, wobei jetzt Transaktionen eingesetzt werden:

```
/* entnimm numMessages Message Entries aus dem Source Space und schreibe
 * die Entries in jeden der Target Spaces
 */
private void relayMessages() {
      TransactionManager mgr = TransactionManagerAccessor.getManager();
      Message template = new Message();
      Message msg;
      for (int i = 0; i < numMessages; i++) {
            Transaction.Created trc = null;
            try {
                trc = TransactionFactory.create(mgr, 300000);
            } catch (Exception e) {
                System.err.println("[SpaceRelay.relayMessages()]Fehler beim
                                          Kreieren der Transaktion " + e);
                return;
            }
            Transaction txn = trc.transaction;
            try {
                try {
                    template.content = "" + i;
                    // lies die Message der Transaktion
                   msg=(Message)sourceSpace.read(template,null,Long.MAX_VALUE);
                   System.out.println("[SpaceRelay.relayMessages()]Message " +
                                    i + " wurde aus " + sourceName+" gelesen");
                   msg = (Message)sourceSpace.take(template, txn,
                                                          Long.MAX_VALUE);
                    System.out.println("[SpaceRelay.relayMessages()]Message
                          " + i + " wurde aus " + sourceName+" entnommen");
                    // schreibe die Message in die anderen Spaces,
                    // mit der Transaktion
                    for (int j = 0; j < targetSpaces.length; j++) {
                  System.out.println("[SpaceRelay.relayMessages()]Schreiben
                                                 in Space "+targetNames[j]);
                       targetSpaces[j].write(msg, txn, Lease.FOREVER);
      System.out.println("[SpaceRelay.relayMessages()]Message " + i + "
                             wurde in " + targetNames[j]+" eingetragen.");
                    }
                } catch (Exception e) {
System.err.println("\n[SpaceRelay.relayMessages()]Message " + i + " kann
nicht geschrieben werden: " + e);
                    txn.abort();
                    return;
                }
                txn.commit();
            } catch (Exception e) {
                System.err.println("[SpaceRelay.relayMessages()]Die
                                              Transaktion schlug fehl");
                return;
            }
        }
```

Als erstes müssen wir einen Transaktionsmanager bestimmen. Dies geschieht mit der statischen `getManager()` Methode der `TransactionManagerAccessor` Klasse. Diese Methode liefert ein `TransactionManager` Proxy Objekt.

Die Konstruktion der Messages bzw. deren Abfrage mit einem Template geschieht analog zu den bisherigen Beispielen.

Die Schleife entnimmt die Objekte / Messages aus dem Source Space und schreibt sie in die Target Spaces, innerhalb der Transaktion:
- falls die Transaktion erfolgreich abgewickelt werden kann, wird sie committed
- sonst wird sie aborted, also rückgängig gemacht.

Der Rest des Programms dient eignetlich nur der dem Starten und abwickeln der gesamten Transaktionen:
- definieren und kreieren des Source Spaces und der Target Spaces und benennen
- kreieren der Messages und Aufruf der obigen Methoden

```
// Konstruktor
public SpaceRelay(String[] args) throws IOException {
    // lies den Namen des Source Spaces
    // des  Target Spaces und die ANzahl Messages
    sourceName = args[0];
    System.out.println("[SpaceRelay.relayMessages()]SourceSpace="
                                                    +sourceName);
    targetNames = new String[args.length - 2];
    for (int i = 0; i < args.length - 2; i++) {
        targetNames[i] = args[i+1];
        System.out.println("[SpaceRelay.relayMessages()]
                            TargetSpace["+i+"]="+targetNames[i]);
    }
    numMessages = Integer.parseInt(args[args.length - 1]);
    // Zugriff auf Source und Target erstellen
    sourceSpace = SpaceAccessor.getSpace(sourceName);
    targetSpaces = new JavaSpace[targetNames.length];
    for (int i = 0; i < targetNames.length; i++) {
        targetSpaces[i] = SpaceAccessor.getSpace(targetNames[i]);
    }
    // kreiere die Messages und spiechere diese im Source Space
    createMessages();
    // relay Messages vom Source ins Target Space
    relayMessages();
}


public static void main(String args[]) {
    if (args.length < 3) {
        System.out.println("Usage:
        SpaceRelay source target1 target2 ... targetN numMessages");
        System.exit(1);
    }
    try {
        SpaceRelay spaceRelay = new SpaceRelay(args);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Die `main()` Methode instanziert ein SpaceRelay Objekt. Im Konstruktor der `SpaceRelay` Klasse werden die Spaces "getauft", gemäss Angabe auf der Kommandozeile.

1.3.5. Zusammenfassung

In diesem Abschnittt haben wir allgemein und an einem Beispiel erklärt, wie Transaktionen im Kontext der JacaSpaces eingesetzt werden können. Jini Transaktion können leicht mit JavaSpaces eingesetzt werden. Diese gestatten es sinnvolle und ausfallgesicherte Transaktionen durchzuführen. Im Beispiel waren alle Partner der Transaktion Jini Services. Aber genauso hätten wir ohne Jini arbeiten können.

Sofern Ihre Dienste das `TransactionParticipant` API implementieren, können Sie auch verteilte Transaktionen realisieren.

1.4. Fehlertoleranz und Skalierbarkeit

1.4.1. Übersicht

Als Beispiel zur Illustration der Skalierbarkeit und Ausfallsicherheit / Fehlertoleranz verwenden wir ein verteiltes Rechnersystem. Der Rechner soll ein möglichst universelles Berechnungsschema implementieren. Dazu verwenden wir das bereits bekannte Beispiel des verteilten Rechners, einfach zur Illustration der Skalierbarkeit und Fehlertoleranz:

wir rüsten das Beispiel mittels Transaktionen auf, um die Applikation robuster zu gestalten und mittels mehrerer Java Spaces skalierbarer.

Unser Computer Server ist ein leistungsfähiger, allgemein einsetzbarer Rechner, welcher Aufgaben anzeptiert und Ergebnisse zurück liefert. Ein Master-Prozess teilt die Aufgabe auf, in viele kleinere Aufgaben - Entries, welche die Aufgaben beschreiben und Methoden enthalten, welche für die Berechnungen benötigt werden. Diese werden in ein Space geschrieben und von dort von den Workern gelesen, herausgeholt und erledigt. Die Ergebnisse werden anschliessend in den Space zurück geschrieben.

Das Beispiel st sehr einfach, zumindest in dieser Form! Aber es besitzt einige Qualitäten:
- es stellt eine allgemein einsetzbare Plattform zur Verfügung, da die Aufgaben flexibel definiert werden können. Der Rechner kann dauernd in Betrieb sein, da die Aufgaben in den Entries gleich mitgeliefert werden.
- der Rechnerserver ist skalierbar - neue Worker können jederzeit hinzugefügt werden. Worker können auch kommen und gehen, ohne dass das System zusammenbricht.
- der Rechner ist durchaus geeignet, Load Balancing zwischen langsameren und schnelleren Maschinen zu implementieren.

Für den praktischen Einsatz fehlen jedoch einige wichtige Merkmale, mindestens die folgenden zwei:
- der Rechner weiss nicht, wie er auf partielle Ausfälle reagieren sollte. Dies kann man durch den Einbau von Transaktionen verbessern. Die Fehlertoleranz kann damit erhöht werden.
- der Einsatz eines einzigen JavaSpaces führt zu einem Engpass, da dieser auf einer einzigen CPU / JVM läuft. Skalierbarkeit mittels mehrerer Java Spaces könnte auch diese Schwachstelle verbessern.

Schauen wir und die zwei Bereiche der Reihe nach an.

1.4.2. Fehlertoleranz

1.4.2.1. Worker mit Transaktionen

Betrachten wir nochmals den Worker aus dem Projekt JavaSpacesVerteilterRechner:

```
package javaspacesverteilterrechner;

import net.jini.core.entry.Entry;
import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

/**
 * Title:        Distributed Computing
 * Description:  einfacher Rechner, welcher JavaSpaces als
   Kommunikationsplattform, Objektrepository für die Kommunikation zwischen
   Clients und dem Master / Server verwendet.
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class Worker {
    JavaSpace space;

    public static void main(String[] args) {
        System.out.println("[Worker]main() Start");
        Worker worker = new Worker();
        worker.startWork();
    }

    public Worker() {
        space = SpaceAccessor.getSpace();
    }

    public void startWork() {
        System.out.println("[Master]startWork()");
        TaskEntry template = new TaskEntry();

        for (;;) {
            try {
                TaskEntry task = (TaskEntry)
                    space.take(template, null, Long.MAX_VALUE);
                Entry result = task.execute();
                if (result != null) {
                    space.write(result, null, 1000*60*10);
                }
            } catch (Exception e) {
                System.out.println("Task wurde gecancelled");
            }
        }
    }
}
```

Die beiden Operationen `take()` und `write()` verwenden ein null Argument als Transaktion, verlaufen also weniger sicher als Transaktionen. Im Falle eines Ausfalles während einer der

kritischen Operationen oder dazwischen, könnte zu einem undefinierten Zustand des Systems führen.

Als erstes bauen wir Transaktionen ein. Dies ist gemäss dem letzten Abschnitt sehr einfach:

```
package javaspacesfehlertoleranz;
...
public Worker() {
        space = SpaceAccessor.getSpace();
        mgr = TransactionManagerAccessor.getManager();
    }
...
public class Worker {
      private JavaSpace space;
      private TransactionManager mgr;
...
      public Transaction getTransaction(long leaseTime) {
        try {
            Transaction.Created created =
                TransactionFactory.create(mgr, leaseTime);
            return created.transaction;
        } catch(RemoteException e) {
            e.printStackTrace();
            return null;
        } catch(LeaseDeniedException e) {
            e.printStackTrace();
            return null;
        }
      }
}
```

Diese Struktur unterscheidet sich nur unwesentlich vom Beispiel aus dem vorherigen Abschnitt über Transaktionen.

Nun schauen wir uns noch die `startWork()` Methode an, da diese die Transaktion implementieren muss:

```
public void startWork() {
        TaskEntry template = new TaskEntry();
        for (;;) {
            // Transaktion mit 10 Minuten Leasezeit
            Transaction txn = getTransaction(1000*10*60);
            if (txn == null) {
                throw new RuntimeException("Transaktion konnte nicht
                                                    erhalten werden");
            } try { try {
                    // Aufgabe der Transaktion
                    TaskEntry task = (TaskEntry)
                        space.take(template, txn, Long.MAX_VALUE);
                    // Aufgabe ausführen
                    Entry result = task.execute();
                    // Ergebnis in den Space zurück schreiben
                    if (result != null) {
                       space.write(result, txn, 1000*60*10);}
                } catch (Exception e) {
                    System.out.println("Task gecancelled:" + e);
                    txn.abort();}
                txn.commit();
            } catch (Exception e) {
```

```
                System.out.println("Transaktion schlug fehl:" + e);
            }
        }    }
```

In dem in diesem Programmcode beschriebenen Szenario können drei Dinge passieren:

- die Operationen könnten erfolgreich abgeschlossen werden und die Transaktion korrekt ausgeführt werden.
- die Operationen werden nicht korrekt ausgeführt, es wird eine Exception geworfen und die Transaktion gecancelled. Es werden keinerlei Änderungen ausgeführt.
- beim Canceln oder bestätigen der Transaktion geschieht ein Fehler. Es wird eine Exception geworfen und der äussere Catch Block ausgeführt, also eine Meldung ausgegeben. Die Transaktion wird sterben, weil die Leasingzeit verstreicht, ohne dass die Transaktion ausgeführt wurde. Die Transaktion wird somit abgebrochen,es findet keine Operation statt.

Mit der Transaktion haben wir den Programmcode robuster gemacht. Nun müssen wir noch den Master verbessern!

1.4.2.2. Master mit Transaktionen

Zuerst schauen wir uns auch in diesem Fall die alte Version an:

```
package javaspacesverteilterrechner;

import net.jini.core.entry.*;
import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class Master {
    private JavaSpace space;
    public static void main(String[] args) {
        System.out.println("[Master]main() Start");
        Master master = new Master();
        master.startComputing();
    }

    private void startComputing() {
        System.out.println("[Master]startComputing()");
        space = SpaceAccessor.getSpace();

        generateTasks();
        collectResults();
    }

    private void generateTasks() {
        System.out.println("[Master]generateTask()");
        for (int i = 0; i < 10; i++) {
            writeTask(new AddTask(new Integer(i), new Integer(i)));
        }
        for (int i = 0; i < 10; i++) {
            writeTask(new MultTask(new Integer(i), new Integer(i)));
        }
    }

    private void collectResults() {
        System.out.println("[Master]collectResults()");
        for (int i = 0; i < 20; i++) {
            ResultEntry result = takeResult();
            if (result != null) {
                System.out.println(result);
```

```
            }
        }
    }

    private void writeTask(Command task) {
        System.out.println("[Master]writeTask");
        try {
            space.write(task, null, Lease.FOREVER);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    protected ResultEntry takeResult() {
        System.out.println("[Master]takeResult()");
        ResultEntry template = new ResultEntry();
        try {
            ResultEntry result = (ResultEntry)
                space.take(template, null, Long.MAX_VALUE);
            return result;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Sie sehen, beispielsweise in der `writeTask()` Methode, dass die Schrieboperation nicht abgesichert ist. Ob die Operation als Ganzes ausgeführt wurde oder fehlerhaft lässt sich also nicht entscheiden. Im schlimmsten Fall führt ein Fehler zu einem undefinierten Systemzustand. Diese Lösung besitzt also kaum Fehlertoleranz. Falls eine der Aufgaben nicht korrekt in den Space geschrieben oder daraus gelesen wird, kann die Aufgabe als Ganzes eventuell nicht mehr korrekt ausgeführt werden.

Nun stabilisieren wir diese Lösung. Hier die `writeTask()` Methode mit Transaktion:

```
private boolean writeTask(Command task) {

        // Transaktion mit 10 Minuten Leasezeit
        Transaction txn = getTransaction(1000*10*60);
        if (txn == null) {
            throw new RuntimeException("Kann keine Transaktion erhalten");
        }

        try {
            try {
                space.write(task, txn, Lease.FOREVER);
            } catch (Exception e) {
                txn.abort();
                return false;
            }
            txn.commit();
            return true;
        } catch (Exception e) {
            System.err.println("Transaktion schlug fehl");
            return false;
        }
    }
```

Mit dieser Methoden können auch wieder drei Szenarien eintreffen:
- falls die Schreiboperation erfolgreich ausgeführt wird, kann die Transaktion bestätigt werden.
- falls bei der Schreiboperation eine Exception geworfen wird, kann die Transaktion abgebrochen werden. Es geschieht ein Rollback. Der Zustand des Systems bleibt definiert.
- falls die Transaktion aus einem anderen Grund abgebrochen wird, muss die Transaktion abgebrochen werden, das eine Exception geworfen wird und danach die Leasingzeit vergeht, ohne dass die Transaktion bestätigt werden konnte.

Nun müssen wir die Methode `generateTask()` verbessern.

```
private void generateTasks() {
        boolean written;

        for (int i = 0; i < 10; i++) {
            written = false;
            while (!written) {
          written = writeTask(new AddTask(new Integer(i), new Integer(i)));
            }
        }
        for (int i = 0; i < 10; i++) {
            written = false;
            while (!written) {
         written = writeTask(new MultTask(new Integer(i), new Integer(i)));
            }
        }
    }
```

Als erste einfache Verbesserung bauen wir eine Endlosschleife ein, welche erst endet, sobald die Aufgabe in den Space geschrieben wurde.

Aber auch die `takeResult()` Methode ist noch verbesserungswürdig.Dazu verwenden wir Transaktionen:

```
protected ResultEntry takeResult() {
        // 10 Minuten Lease
        Transaction txn = getTransaction(1000*10*60);
        if (txn == null) {
    throw new RuntimeException("Transaktion wurde nicht korrekt bestimmt");
        }
        ResultEntry template = new ResultEntry();
        ResultEntry result;
        try {
            try {
                result = (ResultEntry)
                    space.take(template, txn, Long.MAX_VALUE);
            } catch (Exception e) {
                txn.abort();
                return null;
            }
            txn.commit();
            return result;
        } catch (Exception e) {
```

```
            System.err.println("Transaktion schlug fehl");
            return null;
        }
    }
```

Nun müssen wir auch noch die Methode `collectResults()` verbessern.

```
private void collectResults() {
        ResultEntry result;
        for (int i = 0; i < 20; i++) {
            result = null;
            while (result == null) {
                result = takeResult();
            }
            System.out.println(result);
        }
  }
```

Auch hier verbessert die `while`-Schleife die Methode: sie garantiert, dass die Ergebnisse aus dem Space geholt werden.

Damit haben wir den Verteilten Rechner stabilisiert, fehlertoleranter gemacht.
Es bleibt noch die Verbesserung der Skalierbarkeit.

1.4.3. Skalierbarkeit - mit Hilfe mehrerer Spaces

Skalierbarkeit ist partiell bereits vorhanden: je mehr Worker wir haben, auf desto mehr Rechner kann die Arbeit verteilt werden. Aber in einem anderen Aspekt sind die Lösungen noch nicht skalierbar: es wird lediglich ein Java Space eingesetzt.

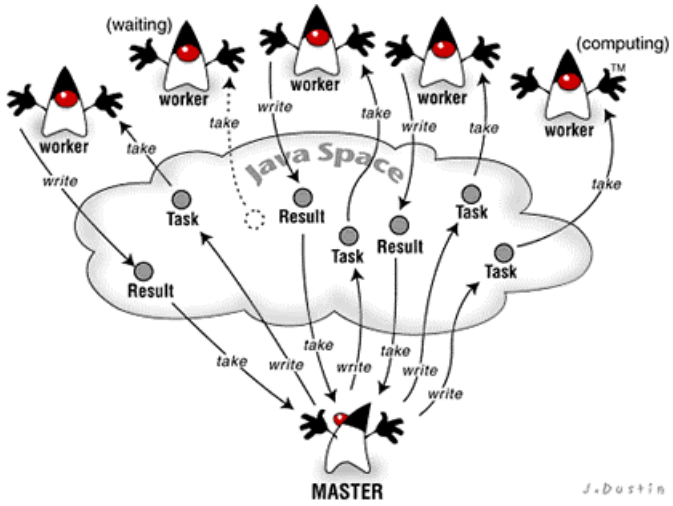
Optimal wäre eine Lösung, bei der mehrere CPUs ein JavaSpace teilen könnten, die Arbeit somit auf mehrere CPUs verteilt werden könnte. Die aktuelle Implementation der JavaSpaces sieht so etwas nicht vor : JavaSpaces sind single CPU Systeme.

Anstatt auf eine solche single Space / multiple CPU Lösung zu warten, können wir die Techniken aus den vorherigen Abschnitten anwenden: den Einsatz mehrerer Spaces.

Der Master muss in diesem Fall in mehrere Spaces schreiben können und die Resultate daraus holen. Wie die Verteilung geschieht, ist sicher ein wichtiges Thema. Man könnte beispielsweise ein Load Balancing damit kombinieren und erst jeweils feststellen, ob ein Space bereits ausgelastet ist oder weitere Aufgaben aufnehmen kann.

In diesem Beispiel beschränken wir uns auf eine denkbar einfache Lösung: jede Task wird auf Grund einer TaskID (int) den Spaces zugeteilt. Auch das Ergebnis der Task mit der ID i stammt aus diesem selben Space.

Mittels Jini werden die verfügbaren Spaces gesucht (Lookup & Discovery). Der Worker selber wird ein einfaches Round Robin Scheduling implementieren: er entnimmt zuerst dem ersten Space eine Aufgabe und liefert die Ergebnisse dort ab; dann entnimmt er dem zweiten Space eine Aufgabe und berechnet das Ergebnis und liefert dieses in den zweiten Space zurück ....



Ein Verteilter Rechner- Service
© J. Dustin

1.4.3.1. Zugriff auf mehrere Spaces

Master und Worker Prozesse verwenden Jini, um die verfügbaren JavaSpace Services zu finden. Dies geschieht mit einem Aufruf:

```
ServiceMatches spaces = SpaceAccessor.getSpaces();
```

Die Klasse `SpaceAccessor` ist eine der Hilfsklassen aus dem Buch von E. Freeman, S. Hupfer et al über JavaSpaces und deren Design Patterns. Mit Hilfe der statischen Methode `getSpaces()` werden die verfügbaren Spaces gesucht.

```
public static ServiceMatches getSpaces() {
   try {
       if (System.getSecurityManager() == null) {
           System.setSecurityManager(new RMISecurityManager());
       }

       Locator locator = new com.sun.jini.outrigger.DiscoveryLocator();
       ServiceRegistrar lookupService = (ServiceRegistrar)locator.locate();

       Class[] types = { JavaSpace.class };
       ServiceTemplate template = new ServiceTemplate(null, types, null);
       ServiceMatches matches = lookupService.lookup(template, 50);
       return matches;
   } catch (Exception e) {
       System.err.println(e.getMessage());
   }
     return null;
}
```

Zuerst wird mit der `locate()` Methode der Jini Lookup Dienst gesucht. Anschliessend wird mit dem `Template` der Datentyp `JavaSpace` gesucht, maximal 50 in unserem Fall (zweites Argument in der `lookup()` Methode). Die Spaces selber werden mit der Methode `getSpaces()` als `matches` erhalten (Rückgabewert). Jedes `ServiceItem` besitzt eine ServiceID.

1.4.3.2. Zerlegung der Aufgaben durch den Master

Als erstes muss der Master die Spaces bestimmen.

```
package javaspacesskalierbarkeit;

import java.rmi.RemoteException;

import net.jini.core.entry.*;
import net.jini.core.lease.*;
import net.jini.core.lookup.*;
import net.jini.core.transaction.*;
import net.jini.core.transaction.server.*;
import net.jini.space.JavaSpace;


public class Master {
    private ServiceItem[] spaceServices = null;
    private int numSpaces = 0;
    private TransactionManager mgr;

    public static void main(String[] args) {
        Master master = new Master();
        master.startComputing();
    }

    private void startComputing() {
        ServiceMatches serviceMatches = SpaceAccessor.getSpaces();
        if (serviceMatches == null) {
            System.out.println("Es wurden keine Spaces gefunden...
                                Das Programm wird abgebrochen.");
        } else {
            spaceServices = serviceMatches.items;
            numSpaces = serviceMatches.totalMatches;
            System.out.println("Es wurden " + numSpaces +
                                                " Spaces gefunden");
            mgr = TransactionManagerAccessor.getManager();

            generateTasks();
            collectResults();
        }
    }


    private void generateTasks() {
        boolean written;
        ServiceItem spaceService;
        JavaSpace space;
        ServiceID id;
        Integer num;

        for (int i = 0; i < 10; i++) {
            spaceService = spaceServices[i % numSpaces];
            space = (JavaSpace)spaceService.service;
            id = spaceService.serviceID;
            num = new Integer(i);

            written = false;
            while (!written) {
                written = writeTask(space, new AddTask(num, num));
```

```
            }
            System.out.println("Die Additionsaufgabe " + i +
                                " wurde in Space " + id+" eingefuegt");

            written = false;
            while (!written) {
                written = writeTask(space, new MultTask(num, num));
            }
            System.out.println("Die Multipliikationsaufgabe " + i +
                            " wurde gerade in Space " + id+" eingefuegt");
        }
    }



    private void collectResults() {
        ResultEntry result;
        for (int i = 0; i < 10; i++) {
            // bestimme zwei Ergebnisse: eine Addition, eine Multiplikation
            for (int j = 0; j < 2; j++) {
                result = null;
                while (result == null) {
                    result = takeResult(i);
                }
            }
        }
    }



    private boolean writeTask(JavaSpace space, Command task) {

        // Transaktion mit 10 Minuten Lease
        Transaction txn = getTransaction(1000*10*60);
        if (txn == null) {
            throw new RuntimeException("Transaktion konnte nicht generiert
                                                                    werden");
        }

        try {
            try {
                space.write(task, txn, Lease.FOREVER);
            } catch (Exception e) {
                txn.abort();
                return false;
            }
            txn.commit();
            return true;
        } catch (Exception e) {
            System.err.println("Transaktion schlug fehl");
            return false;
        }
    }



    protected ResultEntry takeResult(int i) {

        // try to get a transaction with a 10-min lease time
        Transaction txn = getTransaction(1000*10*60);
        if (txn == null) {
```

```
            throw new RuntimeException("Es konnte keine Transaktion
                                                    bestimmt werden");
        }

        ServiceItem spaceService = spaceServices[i % numSpaces];
        JavaSpace space = (JavaSpace)spaceService.service;
        ServiceID id = spaceService.serviceID;
        ResultEntry template = new ResultEntry(i);
        ResultEntry result;


        try {
            try {
                result = (ResultEntry)
                    space.take(template, txn, Long.MAX_VALUE);
            } catch (Exception e) {
                txn.abort();
                return null;
            }
            txn.commit();
            if (AddResult.class.isInstance(result)) {
                System.out.println("Lesen der Addition " + i +
                                                    " aus Space " + id);
            } else {
                System.out.println("Lesen der Multiplikation " + i +
                                                    " aus Space " + id);
            }
            return result;
        } catch (Exception e) {
            System.err.println("Die Transaktion schlug fehl");
            return null;
        }
    }



    private Transaction getTransaction(long leaseTime) {
        try {
            Transaction.Created created =
                TransactionFactory.create(mgr, leaseTime);
            return created.transaction;
        } catch(RemoteException e) {
            e.printStackTrace();
            return null;
        } catch(LeaseDeniedException e) {
            e.printStackTrace();
            return null;
        }
    }

}
```

1.4.3.3. Round Robin Zugriff auf die Spaces

Der Worker muss ebenfalls leicht angepasst werden:

```
package javaspacesskalierbarkeit;

import java.rmi.RemoteException;

import net.jini.core.entry.*;
import net.jini.core.lease.*;
import net.jini.core.lookup.*;
import net.jini.core.transaction.*;
import net.jini.core.transaction.server.*;
import net.jini.space.JavaSpace;



public class Worker {
    private ServiceItem[] spaceServices = null;
    private int numSpaces = 0;
    private TransactionManager mgr;

    public static void main(String[] args) {
        Worker worker = new Worker();
    }

    public Worker() {
      ServiceMatches serviceMatches = SpaceAccessor.getSpaces();
      if (serviceMatches == null) {
          System.out.println("Es wurden keine Spaces gefunden... Das
                                      Programm wird abgebrochen.");
        } else {
            spaceServices = serviceMatches.items;
            numSpaces = serviceMatches.totalMatches;
            System.out.println("Es wurden " + numSpaces +
                                              " Spaces gefunden.");
            mgr = TransactionManagerAccessor.getManager();
            startWork();
        }
    }


    public void startWork() {
        ServiceItem spaceService;
        JavaSpace space;
        ServiceID id;

        TaskEntry template = new TaskEntry();

        for (int i=0; ; i++) {
            spaceService = spaceServices[i % numSpaces];
            space = (JavaSpace)spaceService.service;
            id = spaceService.serviceID;

            // Transaktion mit 10 Min Leaseingdauer
            Transaction txn = getTransaction(1000*10*60);
            if (txn == null) {
                throw new RuntimeException("Transaktion konnte nicht
                                                    generiert werden");
            }
```

```
            try {
                try {
                    // Task gehört zur Transaktion
                    TaskEntry task = (TaskEntry)
                        space.take(template, txn, Long.MAX_VALUE);

                    // Ausführen der Task
                    Entry result = task.execute();

                    if (AddTask.class.isInstance(task)) {
                        System.out.println("Additionsaufgabe " +
                                           ((AddTask)task).a +
                                " aus Space " + id + "...wird
                                                bearbeitet...");
                    } else {
                        System.out.println("Multiplikationsaufgabe " +
                                                ((MultTask)task).a +
                                " aus Space " + id + "...wird
                                                    bearbeitet...");
                    }

                    // schreiben der Ergebnisse in den Space
                    if (result != null) {
                        space.write(result, txn, 1000*60*10);
                    }
                } catch (Exception e) {
                    System.out.println("Task wurde gecancelled:" + e);
                    txn.abort();
                }
                txn.commit();
                System.out.println("Schreibe das Ergebnis in Space " + id);
            } catch (Exception e) {
                System.out.println("Transaktion schlug fehl:" + e);
            }
        }
    }

    public Transaction getTransaction(long leaseTime) {
        try {
            Transaction.Created created =
                TransactionFactory.create(mgr, leaseTime);
            return created.transaction;
        } catch(RemoteException e) {
            e.printStackTrace();
            return null;
        } catch(LeaseDeniedException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

1.4.4. Zusammenfassung

Damit haben wir den Verteilten Jini / JavaSpaces Rechner robuster und skalierbarer gemacht. Mit den Transaktionen und dem Einsatz mehrerer JavaSpaces ist es recht einfach möglich, Jini / JavaSpaces Anwendungen wesentlich zu verbessern, praxisnaher zu machen.

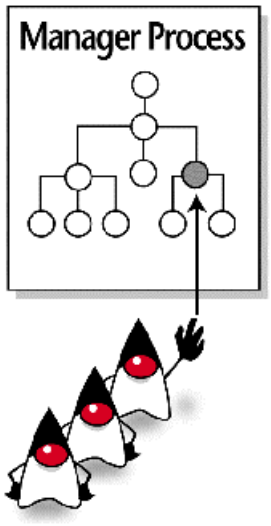
1.5. Verteilte Datenstrukturen

1.5.1. Einleitung

Bisher haben wir an Beispielen gesehen, wie JavaSpaces Programmierung aussehen könnte. In diesem Abschnitt geht es nun noch darum, *Distributed Data Structures* einzuführen, als Datenstrukturen oder Building Blocks, aus denen Space basierte Programme aufgebaut werden.

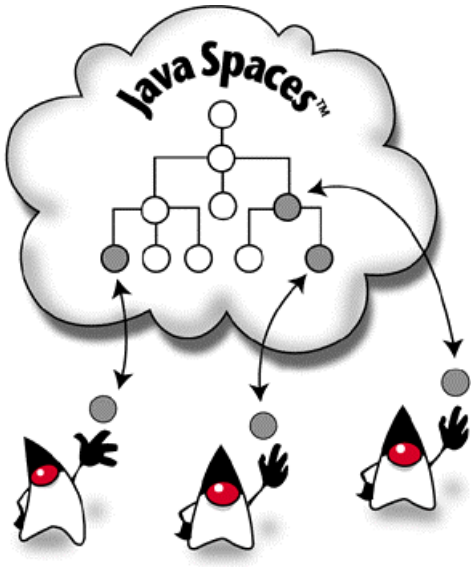
Als Beispiel besprechen wir eine *Channel* verteilte Datenstruktur und illustrieren, wie man damit eine verteilte MP3 Applikation realisieren könnte.

Jede Anwendung, jedes Programm verwendet bestimmte Datenstrukturen, im Falle von Spaces sind dies verteilte Datenstrukturen. Tyischerweise werden an Spaces Anwendungen viele Prozesse beteiligt sein. Oft werden in klassischen Anwendungen die Daten hinter einen Manager gestellt. Dieser verwaltet die Datenstruktur und ist für deren Lebenszyklus zuständig. Der Nachteil ist, dass alle Worker sich einreihen müssen und auf die Gunst des Manager-Prozesses warten müssen.



1. Konventionelle Systeme verstecken Daten hiner einem zentralisierten Manager- Prozess
(Graphik von Sun Microsystems)

Falls man verteilte Datenstrukuren einsetzt, geht man grundsätzlich anders vor. In diesem Fall betrachtet man das System in der Regel als Sammlung interagierender Objekte. Daher spielen Fragen der Koordination, Synchronisation, Fehlertoleranz und Skalierbarkeit entscheidende Rollen. Die einzelnen Objekte warten nicht oder kaum auf die Befehle eines zentralen Koordinators, sie agieren autonom.



2. JavaSpaces unterstützen Concurrent Zugriff auf verteilte Datenstrukturen .
(Graphik von Sun Microsystems)

Neben der Datenstruktur wird in diesem Fall aber auch ein Protokoll benötigt, also Regeln für die Koordination der Objekttätigkeiten.

1.5.2. Aufbau verteilter Datenstrukturen mit Hilfe von Entries

Als erstes betrachten wir ein Array, in das wir Daten speichern wollen. Dieses Array soll von mehreren Prozessen simultan genutzt werden können.

Praktisches Beispiel:
Wir möchten die Temperaturdaten aller europäischen Länder oder aller Kantone oder bestimmer Messpunkte abspeichern. Die Daten sollen von mehreren Wetterbüros genutzt werden können. Jedes Wetterbüro hat zudem Zugriff auf einige der Wetterstationen und kann verschiedene Parameter setzen und abfragen. Als erstes wollen wir lediglich die Maximaltemperaturen gemeinsam nutzen:

```
public class MaximalTemperatur implements Entry {
      public Integer[] values;
      public MaximalTemperatur() {
      }
}
```

Falls eine der Wetterstationen ihren Wert modifizieren möchte, muss sie das gesamte Array lesen (und blockieren). Wir haben also einen Flaschenhals, einen zentralen Kontroll- oder Management- Prozess.

Wir wollen daher eine Alternative suchen und das Temperaturarray in eine Collection von Entries zerlegen. Jede Entry repräsentiert ein einzelnes Array Element.

Diese Datenstruktur könnte folgendermassen definiert werden:

```
public class MaximalTemperatur implements Entry {
      public Integer index;
      public Integer value;
      public MaximalTemperatur () {
      }

      public MaximalTemperatur (int index, int value) {
            this.index = new Integer(index);
            this.value = new Integer(value);
      }
}
```

Der Index der Entry (die Position im Array) und der Wert (die Temperatur) sind die einzigen Datenfelder der Datenstruktur / Klasse.

Die Initialisierung der Objekte könnte etwa folgendermassen aussehen:

```
for (int i = 1; i <= 10 ; i++) {
      MaximalTemperatur temp = new MaximalTemperatur (i, -99);
      space.write(temp, null, Lease.FOREVER);
}
```

wobei wir den `try ... catch` Block weggelassen haben. Die Schleife schreibt 10 Objekte in den Space, jeweils mit einem Temperaturwert von -99 zur Initialisierung.

Wollten wir den fünften Wert bestimmen, könnte unser Programmfragment etwa folgendermassen aussehen:

```
// Template zum Lesen der Objekte
MaximalTemperatur template = new MaximalTemperatur ();

template.index = new Integer(5);

// Lesen des 5ten Objekts aus dem Space
MaximalTemperatur temp = (MaximalTemperatur)space.read(template, null,
                                                       Long.MAX_VALUE);
```

Falls wir die Temperatur modifizieren möchten, entfernen wir die Entry aus dem Space, modifizieren den Wert und schreiben anschliessend die Entry wieder in den Space:

```
// Entfernen des Elements
MaximalTemperatur temp = (MaximalTemperatur)space.take(template, null,
                                                       Long.MAX_VALUE);
// Modifizieren des Wertes
temp.value = new Integer(32);

// zurückspeichern der Entry
space.write(temp, null, Lease.FOREVER);
```

Das Beispiel zeigt, wie einfach verteilte Datenstrukturen konstruiert werden können. Da die einzelnen Entries im Space unabhängig sind, können mehrere Prozesse gleichzeitig an der Datenstruktur arbeiten, mehrere Wetterstationen ihre neuen Werte eintragen.

In diesem einfachen Beispiel ist das Protokoll fast trivial:

- lesen der Entry aus dem Space (`take()`)
- modifizieren der Entry (mutieren des Temperaturwertes)
- schreiben der Entry in den Space (`write()`)

Das Protokoll könnte leicht erweitert werden, beispielsweise indem man auch noch die maximale oder aktuelle Anzahl der Entries im Space mitspeichern würde.

Verteilte Datenstrukturen können sich wesentlich von konventionellen Datenstrukturen unterscheiden. Eine typische verteilte Datenstruktur sind die *Bags*, welche Task und Resultat Entries enthalten.

1.5.3. Ungeordnete Strukturen: Bags

Im Gegensatz zu sequentiellen Programmen verwenden Space-basierte Programme oft ungeordnete Sammlungen von Objekten - Collections oder *Bags* (Beutel, Einkaufstaschen).

In diesen ungeordneten Datenstrukturen kann man im Wesentlichen zwei Operationen ausführen:
- Objekte entnehmen (`get()`) oder
- Objekte hinzufügen (`put()`)

Die Reihenfolge der Objekte ist unwichtig!

Anwendungen dieser Datenstruktur haben wir bereits gesehen: der Master beim Rechnerbeispiel zerlegt die Aufgabe in Teilaufgaben und schreibt Task Entries in einen Task Bag im Space. Ein oder mehrere Worker entnehmen die Tasks, bearbeiten diese und schreiben die Ergebnisse zurück. Der Master Prozess zerlegt die Aufgabe in Teilaufgaben und sammelt die Ergebnisse wieder ein.

Ein Anwendungsmuster für Bags wären Aufgaben, in denen es unwichtig ist, wer die Teilaufgabe löst. Wir hätten also Teilprozesse, welche produzieren und andere Prozesse, welche konsumieren, wobei den Konsumenten egal ist woher das Produkt stammt.

An Stelle eines Produkts kann auch eine Dienstleistung erbracht werden, wie beispielsweise beim verteilten Rechner.

Sie können sich sicher viele weitere brauchbare Anwendungen für dieses Pattern vorstellen oder ausdenken.

1.5.4. Geordnete Strukturen: Channels

Typischerweise wird man sich beim Studium des JavaSpaces API fragen, wie man alle Entries eines Spaces bestimmen kann. Leider gibt das API darauf keine Antwort! Spaces sind grosse Bags, ungeordnete Listen oder Collections von Objekten.

Eine Übersicht über den Inhalt eines Spaces könnte man leichter erhalten, falls man eine bestimmte Ordnung in den Spaces definieren könnte oder wenn man eine geordnete verteilte Datenstruktur zur Verfügung hätte. Dann könnte man recht leicht die Entries auflisten.

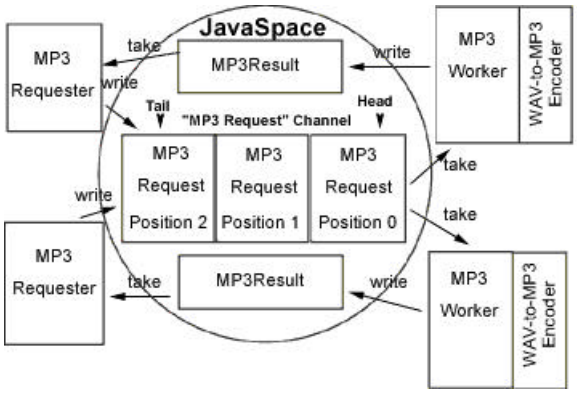
Eine solche verteilte aber geordnete Datenstruktur ist der *Channel.* Ein Channel ist etwa mit einer Pipe, einer Röhre vergleichbar, welche am einen Ende Objekte aufnehmen kann und am anderen Ende in der selben Ordnung / Reihenfolge wieder abgeben kann. Beim Eingang benötigen wir einen oder mehrere Prozesse, welche die Objekte einlagern; beim Ausgang benötigen wir einen oder mehrere Prozesse, welche die Objekte auslagern, lesen und entfernen.

Als Anwendung dieser Channel Datenstruktur werden wir im folgenden Beispiel eine WAV Datei in eine MP3 Datei umwandeln, verteilt, mit mehreren Prozessen!

1.5.5. Eine verteilte MP3 Verschlüsselungsapplikation

MP3 ist ein populäres Format für die Sound Komprimierung. Die Qualität der MP3s ist fast mit jener einer CD vergleichbar. Aber die MP3 Dateien sind wesentlich kleiner als jene im WAV oder anderen Formaten. Typischerweise wandelt man die Audio Dateien der CDs in MP3 auf dem PC um. Diese Umwandlung kann sehr zeitintensiv sein. Daher wollen wir eine solche Verschlüsselung mit Hilfe einer verteilten Datenstruktur, eines Channels realisieren.

Das Verschlüsseln einer gesamten CD kann problemlos parallel mit mehreren Workern und unter Zuhilfenahme eines Spaces gelöst werden. Die Architektur wird in der folgenden Skizze wiedergegeben:



3. Architektur der verteilten MP3-Verschlüsselungs Applikation

Auf den ersten Blick sieht es so aus, als könnten wir die Bag Struktur anwenden, gemäss folgendem Protokoll:

- mehrere Requester plazieren ihre Anfragen im Space
- die Worker entnehmen die Aufgabe dem Space und bearbeiten diese
- die Worker stellen die Resultate in den Space
- der Requester holt sich das Ergebnis ab.

Aber das Schema hat einen entscheidenden Nachteil: es gibt keine Fairness. Es ist nicht sicher, ob zuerst die alten Anfragen bearbeitet werden oder einfach immer gleich die neusten (FIFO versus First In Last Out).

Besser wäre sicher die Channel Struktur: die Anfragen werden in den Channel (Space) gestellt und der Reihe nach bearbeitet.

Aus Fig. 3 oben ist grob ersichtlich, dass für diese Datenstruktur ein Kopf und ein Ende Marker (*head, tail*) benötigt werden:

- die `MP3Requester` Prozesse fügen neue Anfragen für die MP3 Kodierung hinten an
- der `MP3Worker` verarbeitet die Anfragen mit Hilfe externer Software, der MP3 Encoding Software. Er entnimmt die Aufgabe dem Kopf des Channels. Die Ergebnisse werden in den Resultate Bag geschrieben. Hier ist die Bag Struktur sinnvoll, weil die reihenfolge in der die Ergebnisse gelesen werden, eher unwichtig für diesen gesamten Prozess ist.
- der `MP3Requester` überprüft periodisch den Space, um festzustellen, ob seine Aufgabe bereits erledigt wurde.

Soweit die Übersicht. Nun folgt die konkrete Implementation.

1.5.5.1. Der MP3 Request Channel

Um den MP3 Request Channel zu implementieren, benötigen wir eine geordnete Collection von `MP3Request` Entries. Jede `MP3Request` Entry enthält seine Position innerhalb des Channels. Neue Anfragen werden hinten angehängt. Zudem benötigen wir eine Entry, die Head Entry, welche sich die Nummer des vordersten Requests merkt. Analog dazu existiert eine Entry, welche das Ende des Channels markiert.

1.5.5.2. Die MP3 Request Entry

Die MP3Request Entry ist folgendermassen definiert:

```
package javaspacesdatenstrukturen;

import net.jini.core.entry.*;

public class MP3Request implements Entry {
    public String channelName;  // Empfänger des REquests
    public Integer position;    // Positinsnummer im Channel
    public String inputName;    // Dateipfad
    public byte[] data;         // Inhalt der DAtei
    public String from;         // Request von

    public MP3Request() { // Default Konstruktor
    }

    public MP3Request(String channelName) {
        this.channelName = channelName;
    }

    public MP3Request(String channelName, Integer position) {
        this.channelName = channelName;
        this.position = position;
    }

    public MP3Request(String channelName, Integer position,
        String inputName, byte[] data, String from)
    {
        this.channelName = channelName;
        this.position = position;
        this.inputName = inputName;
        this.data = data;
        this.from = from;
    }

}
```

Jede `MP3Request` Entry enthält fünf Datenfelder: `channelName`, `position`, `inputName`, `data` und `from`. Die Bedeutung der Felder sollte weitestgehend aus dem Kontext und den Kommentaren im Programmcode.

1.5.5.3. Start und Ende des Channels verwalten

Die Verwaltung des Channels (Head und Tail Entry) geschieht mit Hilfe der `Index` Klasse.

```
package javaspacesdatenstrukturen;

import net.jini.core.entry.Entry;

public class Index implements Entry {
    public String type;        // head oder tail
    public String channel;
    public Integer position;

    public Index() {
    }

    public Index(String type, String channel) {
        this.type = type;
        this.channel = channel;
    }
    public Index(String type, String channel, Integer position) {
        this.type = type;
        this.channel = channel;
        this.position = position;
    }
    public Integer getPosition() {
        return position;
    }
    public void increment() {
        position = new Integer(position.intValue() + 1);
    }
}
```

Die Entry besitzt drei Datenfelder:
- `type` : identifiziert den Typus des Indexes: Head oder Tail des Channels,
- `channel` : enthält den Namen des Channels,
- `position` : enthält die Position (der Head oder Tail Entry) im Channel.

Die Methode `getPosition()` liefert die Position der Head oder Tail Entry; `increment()` erhöht die Position um eins.

Wie kann man nun den Channel verwalten?
1) Falls der Channel beispielsweise 5 `MP3Request` Entries enthält, auf Position 1 bis 5, ist der Head an Position 1, Tail ist Position 5.

2) Wenn nun ein neuer Request eingetragen wird, muss die Tail Position um eins erhöht werden.

3) Falls ein Request bearbeitet, also aus dem Channel entfernt wird, muss die Head Position um eins erhöht werden.

4) Falls die Requests schneller bearbeitet werden als neue Request eingetragen werden, kann es passieren, dass Head und Tail zusammenfallen.

1.5.5.4. Kreieren eines MP3Request Channels

Der Channel wird von einem Administrator kreiert. Dieser muss die `ChannelCreator` Klasse starten. Diese Klasse ist folgendermassen definiert:

```
package javaspacesdatenstrukturen;

import net.jini.space.*;
import net.jini.core.lease.*;

public class ChannelCreator {
    private JavaSpace space;

    public static void main(String[] args) {
        ChannelCreator creator = new ChannelCreator();
        creator.createChannel(args[0]);
    }

    private void createChannel(String channelName) {
        space = SpaceAccessor.getSpace();
        Index head =
                new Index("head", channelName, new Integer(1));
        Index tail =
                new Index("tail", channelName, new Integer(0));

        System.out.println("Kreiere einen neuen Channel: " +
                                                    channelName + ".");
        try {
            space.write(head, null, Lease.FOREVER);
            space.write(tail, null, Lease.FOREVER);
        } catch (Exception e) {
            System.out.println("Fehler beim Kreieren des Channels.");
            e.printStackTrace();
            return;
        }
        System.out.println("Channel " + channelName + " wurde kreiert.");
    }
}
```

Die Arbeit wird von der Methode `createChannel()` erledigt. Diese benötigt einen Parameter, den Namen des Channels. Und wie sie sehen, besitzt der Head die höchste, die Tail Entry die tiefste Nummer.

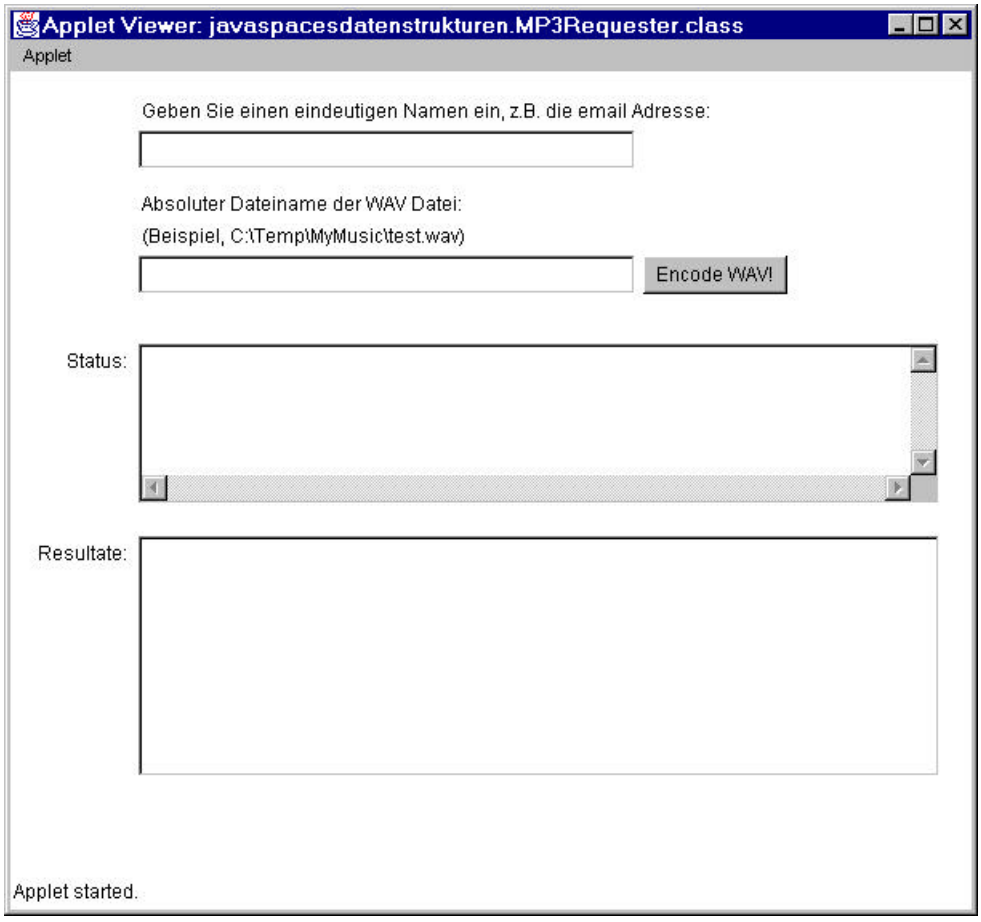
1.5.5.5. Der MP3 Requester

Nachdem der Channel nun existiert, können wir Request darin platzieren. Dazu wird ein Applet eingesetzt, das `MP3Requester` Applet. Dieses besitzt das unten abgebildete GUI.

Wenn man eine MP3 Verschlüsselung durchführen möchte, muss man einen Benutzernamen und die Dateinamen angeben. Der Benutzername dient der Identifizierung des Requests, muss also eindeutig sein.

Der Name der Datei muss absolut eingegeben werden, zum Beispiel:
`C:\Windows\Desktop\wavs\drpepper.wav`

Dann klicken Sie auf den Encode Knopf. Damit wird der MP3Requester den Request in den Channel stellen.



4. Das MP3Requester GUI

Das Applet verwendet JavaSpace und die oben besprochenen verteilten Datenstrukturen. Beim Anklicken des Encode Buttons wird die `actionPerformed()` Methode ausgeführt. Zuerst wird der eindeutige Namen des Requesters bestimmt, beispielsweise die email Adresse des Benutzers. Anschliessend wird die WAV Datei geöffnet und die Rohdaten bestimmt. Der Request wird mit der `append()` Methode in den Channel hineingestellt.

```
package javaspacesdatenstrukturen;

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import net.jini.space.*;
import net.jini.core.lease.*;

public class MP3Requester extends Applet
        implements ActionListener, Runnable
{
    private JavaSpace space;
    private Thread resultTaker;
    private String from; // eindeutiger Name für die MP3 verlangende Person

    // GUI
    private String newline = System.getProperty("line.separator");
    private Label label1;
    private Label label2;
    private Label label3;
    private Label statusLabel;
    private Label mp3Label;
    private TextField userTextField;
    private TextField fileTextField;
    private Button encodeButton;
    private TextArea statusOutput;
    private List mp3Links;

    public void init() {
        space = SpaceAccessor.getSpace();

        // Thread zum Verteilen und Sammeln der Ergebnisse
        if (resultTaker == null) {
            resultTaker = new Thread(this);
            resultTaker.start();
        }

        // GUI Setup
      setLayout(null);
      setSize(600,500);

        // Label
        label1 = new Label("Geben Sie einen eindeutigen Namen ein,
                            z.B. die email Adresse:", Label.LEFT);
        label1.setBounds(80,12,600,25);
        add(label1);

        // Benutzername
      userTextField = new TextField();
      userTextField.setBounds(80,37,310,24);
      add(userTextField);

      // Labels
      label2 = new Label("Absoluter Dateiname der WAV Datei:", Label.LEFT);
      label2.setBounds(80,70,600,25);
      add(label2);
      label3 = new Label("(Beispiel, C:\\Temp\\MyMusic\\test.wav)",
                                                          Label.LEFT);
      label3.setBounds(80,90,600,25);
```

```
      add(label3);

      // Textfeld für den Dateinamen
      fileTextField = new TextField();
      fileTextField.setBounds(80,115,310,24);
      add(fileTextField);

      // Encode Button
      encodeButton = new java.awt.Button();
      encodeButton.setLabel("Encode WAV!");
      encodeButton.setBounds(395,115,90,24);
      encodeButton.addActionListener(this);
      add(encodeButton);

      // Status Label
        statusLabel = new Label("Status:",Label.RIGHT);
        statusLabel.setBounds(0,170,75,21);
        add(statusLabel);

        // Status Zeile
        statusOutput = new TextArea();
        statusOutput.setBounds(80,170,500,100);
        add(statusOutput);

        // Ergebnislabel
        mp3Label = new Label("Resultate:",Label.RIGHT);
        mp3Label.setBounds(0,290,75,21);
        add(mp3Label);

        //// Text Area für Resultate
        //mp3Output = new TextArea();
        //mp3Output.setBounds(80,290,500,300);
        //add(mp3Output);

        // Liste mit den MP3 Dateinamen
        mp3Links = new List(10, false);
        mp3Links.setBounds(80,290,500,150);
        add(mp3Links);
    }

    // "Encode" wurde gedrückt
    public void actionPerformed(ActionEvent event) {

        from = userTextField.getText();
        if (from.equals("")) {
            statusOutput.append("Geben Sie einen eindeutigen
                                        Benutzernamen ein." + newline);
            return;
        }

        String inputName = fileTextField.getText();
        if (inputName.equals("") || (!inputName.endsWith(".wav"))) {
            statusOutput.append("Geben Sie den absoluten Namen der WAV
                                                Datei ein." + newline);
            return;
        }
        fileTextField.setText("");

        // Daten des WAV Files in eine ENtry einpacken
        byte[] rawData = null;

        rawData = Utils.getRawData(inputName);
```

```
        // einen MP3 Request generieren
        if (rawData != null) {
            append("MP3 Request", inputName, rawData, from);
        }
    }

    private void append(String channel, String inputName,
            byte[] rawData, String from) {
        statusOutput.append("Suche \"" + channel + "\"..." + newline);
        Integer num = getRequestNumber(channel);
        MP3Request request =
            new MP3Request(channel, num, inputName, rawData, from);

        statusOutput.append("Sende Request " + num +
            " an \"" + channel + "\"..." + newline);
        try {
            space.write(request, null, Lease.FOREVER);
        } catch (Exception e) {
            statusOutput.append("Fehler beim Schreiben in den Channel." +
                                                                newline);
            e.printStackTrace();
            return;
        }
        statusOutput.append("Anfrage " + num +
            " an \"" + channel + "senden \"... Erledigt." + newline);
    }


    private Integer getRequestNumber(String channel) {
        try {
            Index template = new Index("tail", channel);
            Index tail = (Index) space.take(template, null,
                                                  Long.MAX_VALUE);
            tail.increment();
            space.write(tail, null, Lease.FOREVER);
            return tail.getPosition();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }


    // Thread zeigt MP3 Links an
    public void run() {
        MP3Result template = new MP3Result(from);
        MP3Result result = null;
        String outputName = "";   // Name der MP3 Dateien

        while(true) {
            try {
                result = (MP3Result)
                    space.take(template, null, Long.MAX_VALUE);
                statusOutput.append("Ergebnis fuer: " + result.inputName +
                                                        "." + newline);
                int pos = result.inputName.indexOf(".wav");
                outputName = result.inputName.substring(0, pos) + ".mp3";
                Utils.putRawData(result.data, outputName);
                displayResult(outputName);
            } catch (Exception e) {
                e.printStackTrace();
```

```
            }
        }
    }


    private void displayResult(String outputName) {
        statusOutput.append("Datei " + outputName + " wurde geschrieben." +
                                                                    newline);
        mp3Links.addItem(outputName);
    }


   public boolean action(Event event, Object arg){
        if (event.target == mp3Links){
             String fileName = (String)arg;
             System.out.println("Action - mit Datei: " + fileName);

             try {
                String[] cmdArray = { "\"C:\\Programme\\Winamp\\Winamp\"",
                                                                  fileName};
                Process subProcess = Runtime.getRuntime().exec(cmdArray);
             } catch (Exception e) {
                System.out.println("Fehler beim Apspielen der Datei.");
                e.printStackTrace();
                return false;
             }
         }
        return true;
    }

}
```

Die Ergebnisse der Konversion werden mittels *Result Tracker Thread* verfolgt. Dessen `run()` Methode entfernt laufend abgeschlossene WAV -zu - MP3 Requests. Diese Aktivitäten werden im Fortschrittsfenster des Applets angezeigt. Diese Aktivitäten müssen in einem separaten Thread abgewickelt werden, damit der GUI Thread immer auf Benutzereingaben reagieren kann.

1.5.5.6. Eintrag eines Requests in den Channel

Schauen wir uns die Methoden noch etwas genauer an. `append()` fügt einen Request ans Ende des Channels an. Zuerst wird die aktuelle Positionsnummer des Endes bestimmt, um eins inkremenriert und der neue Request mit dieser Nummer gekennzeichnet.

Die `getRequestNumber()` Methode bestimmt die Positionsnummer eines neuen Request:

```
private Integer getRequestNumber(String channel) {
        try {
            Index template = new Index("tail", channel);
            Index tail = (Index) space.take(template, null,
                                              Long.MAX_VALUE);
            tail.increment();
            space.write(tail, null, Lease.FOREVER);
            return tail.getPosition();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```

Zuerst wird ein `Index` Template kreiert, mit dem wir das Ende des Channels suchen. Anschliessend entfernen wir mit `take()` die letzte Entry des Channels (die Tail Entry). Dessen Zähler wird um eins erhöht und zurück geschrieben. Damit haben wir die Positionsnummer des neuen Requests! Am Anfang steht der Index / Zähler auf 0, wird also bei der ersten Einlagerung auf 1 erhöht: der erste Request erhält die Kennziffer 1.

Mit dieser Methode implementieren wir die `append()` Methode:

```
private void append(String channel, String inputName,
            byte[] rawData, String from) {
        statusOutput.append("Suche \"" + channel + "\"..." + newline);
        Integer num = getRequestNumber(channel);
        MP3Request request =
            new MP3Request(channel, num, inputName, rawData, from);

        statusOutput.append("Sende Request " + num +
            " an \"" + channel + "\"..." + newline);
        try {
            space.write(request, null, Lease.FOREVER);
        } catch (Exception e) {
            statusOutput.append("Fehler beim Schreiben in den Channel." +
                                                                newline);
            e.printStackTrace();
            return;
        }
        statusOutput.append("Anfrage " + num +
            " an \"" + channel + "senden \"... Erledigt." + newline);
    }
```

Diese Methode hat als Parameter den Namen des Channels ("MP3Request"), den Dateinamen des zu verschlüsselnden WAV Files und den Namen des Requesters.

Als erstes wird die Methode `getRequestNumber()` aufgerufen, mit dem Channel Namen als Parameter. Damit wird der Channel wie oben erwähnt aktuell gehalten.

Dann wird ein `MP3Request()` kreiert. Jetzt wollen wir sehen, wie die Einträge bearbeitet werden.

1.5.5.7. Die MP3 Worker

Der Administrator startet einen oder mehreren MP3 Worker. Diese entnehmen die MP3 Requests aus dem Channel Kopf. Die Kodierung geschieht mit Hilfe eines externen Spezialprogramms. Nach der Kodierung wird das Ergebnis in den Space geschrieben. Dieser Space wird wie oben erwähnt, von allen MP3Requestern laufend beobachtet.

Und so sieht der Worker aus:

```
package javaspacesdatenstrukturen;

import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;
import java.io.*;
import java.lang.Runtime;

public class MP3Worker {
    private JavaSpace space;
    private String channel;

    public static void main(String[] args) {
        MP3Worker worker = new MP3Worker();
        worker.startWork();
    }

    public void startWork() {
        space = SpaceAccessor.getSpace();
        channel = "MP3 Request";

        while(true) {
            processNextRequest();
        }
    }
}
```

Die Hauptarbeit erledigt die `processNextRequest()` Methode:

```
private void processNextRequest() {
        Index tail = readIndex("tail", channel);
        Index head = removeIndex("head", channel);

        if (tail.position.intValue() < head.position.intValue()) {
            // es liegen keine Anfragen vor
            writeIndex(head);
            return;
        }

        // neue Anfrage
        MP3Request request = removeRequest(channel, head.position);
        head.increment();
        writeIndex(head);

        if (request == null) {
            System.out.println("Kommunikationsfehler.");
            return;
        }

        // MP3 Request, rufe BladeEnc Programm auf
        // Konvertiere WAV in MP3
```

```
        System.out.println("Anfrage von " + request.from + " fuer: " +
                                                        request.inputName);

        String inputName = request.inputName;
        String from = request.from;
        byte[] inputData = request.data;
        byte[] outputData = null;
        String tmpInputFile = "./tmp" + request.position + ".wav";
        String tmpOutputFile = "./tmp" + request.position + ".mp3";
        Process subProcess = null;

        Utils.putRawData(inputData, tmpInputFile);
        try {
            String[] cmdArray = { "\"C:\\Programme\\BladeEnc\\BladeEnc\"",
            "-quit", "-quiet", "-progress=0", tmpInputFile, tmpOutputFile};
            subProcess = Runtime.getRuntime().exec(cmdArray);
            subProcess.waitFor();
            } catch (Exception e) {
            System.out.println("Fehler beim Encoding.");
            e.printStackTrace();
            return;
      }

      System.out.println("Prozess wurde beendet mit Programmcode " +
                                                  subProcess.exitValue());
            if (subProcess.exitValue() != 0) {
                System.out.println("Fehler beim Kodieren.");
                return;
            }

      // MP3 in den Space stellen
        outputData = Utils.getRawData(tmpOutputFile);
        MP3Result result = new MP3Result(inputName, outputData, from);
        try {
            space.write(result, null, Lease.FOREVER);
            System.out.println("MP3 für " + inputName + " steht im
                                                          Space.");

          // delete the temporary files
          new File(tmpInputFile).delete();
          new File(tmpOutputFile).delete();
        } catch (Exception e) {
            System.out.println("Fehler beim Schreiben in den Channel.");
            e.printStackTrace();
            return;
        }
    }
```

Zuerst wird geschaut ob Header und Tail unterschiedlich sind und zwar so, dass Tail > Head. Sonst ist der Channel leer und es muss nichts getan werden. Falls jedoch Requests im Channel sind, wird `processNextRequest()` aufgerufen und `removeRequest()`.

Wie Sie oben erkennen können, werden zwei temporäre Dateien angelegt, eine für die WAV Eingabedatei und eine für die MP3 Ausgabedatei.

Zusätzlich benötigen wir ein paar Helper Methoden wie beispielsweise `removeRequest()`. Das Listing finden Sie oben. Die Methode ist soweit dokumentiert, dass ein weiterer Kommentar wohl überflüssig ist.

1.5.5.8. Die MP3Result Entry

Die Worker speichern ihre Ergebnisse in `MP3Result` Entries. Hier eine Definition dieser Entries:

```
package javaspacesdatenstrukturen;

import net.jini.core.entry.*;

public class MP3Result implements Entry {
    public String inputName;    // Datei, welche kodiert wird
    public byte[] data;         // MP3 Daten
    public String from;         // wer sandte den Request?

    public MP3Result() {  // Default Konstruktor
    }

    public MP3Result(String from) {
        this.from = from;
    }

    public MP3Result(String inputName, byte[] data, String from) {
        this.inputName = inputName;
        this.data = data;
        this.from = from;
    }
}
```

Wie sie sehen, enthält die Entry den Dateinamen, die MP3 Bytes und den Namen der Person, welche die Anfrage gestellt hat.

1.5.5.9. Sammeln und Anzeige des MP3 Resultats

Wie oben erwähnt sucht ein Hintergrundthread laufend nach Ergebnissen im Space und entfernt die fertigen Ergebnisse. Dies geschieht mit folgenden Programmzeilen:

```
public void run() {
        MP3Result template = new MP3Result(from);
        MP3Result result = null;
        String outputName = "";   // Name der MP3 Dateien

        while(true) {
            try {
                result = (MP3Result)
                    space.take(template, null, Long.MAX_VALUE);
                statusOutput.append("Ergebnis fuer: " + result.inputName +
                                                        "." + newline);
                int pos = result.inputName.indexOf(".wav");
                outputName = result.inputName.substring(0, pos) + ".mp3";
                Utils.putRawData(result.data, outputName);
                displayResult(outputName);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Als erstes wird ein passende Template konstruiert, mit den Namen des Auftraggebers aus dem GUI. Anschliessend werden MP3 Result Entries gesucht.

Falls ein Result Entry gefunden wird, entfernt der Thread dieses und schreibt das Ergebnis in die Ausgabedatei. Die Ausgabedatei besitzt einen Namen, der sich leicht aus dem der Eingabedatei herleitet. Beispiel:
Eingabe `C:\Temp\MyMusic\wavs\drpepper.wav`
Ausgabe `C:\Temp\MyMusic\wavs\drpepper.mp3`.

Mit der Methode `displayResult()` wird der Namen ins Fenster des GUIs eingetragen. Falls der Benutzer doppelklickt, wird ein fix einprogrammierter MP3 Player gestartet.

1.5.5.10. Zusammenfassung

In diesem Beispiel haben wir ein reales Problem mit verteilten Datenstrukturen gelöst. Die Verteilten Datenstrukturen lassen sich leicht praxisnah umsetzen.

Das Channel Pattern unterscheidet sich vom Master / Worker Pattern, welches wir weiter vorne eingesetzt haben:
- beim Channel Pattern haben wir viele Requester und viele Worker
- beim Master / Worker Pattern haben wir einen Master (= Requester) und viele Worker.

Das Channel Pattern gestattet eine FIFO basierte Behandlung von Problemen. Im Master / Worker Pattern besteht keinerlei Ordnung, also eine Bag basierte Datenstruktur.

Verteilte Datenstrukturen bilden das Fundament für Space.basierte Programmierung. Wir können jederzeit auch komplexere Strukturen, wie beispielsweise hierarchische Baumstrukturen, auf Entries aufbauen.

1.6. Schlussbemerkung

Die Technologie verbreitet sich sehr schnell heute. Heute gibt es e-Mail Systeme, die Jini und JavaSpace basiert sind, Simulatoren für, na was wohl (DOD Tanks). Die Reise startet erst. Es wird noch interessant werden. Viel Spass beim Mitreisen.

Nun liegt es an Ihnen, neue und innovative Anwendungen dieser neuartigen Techniken zu finden und die Informatik voranzutreiben!