UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Institut für Informatik
Arbeitsgruppe Softwaretechnik
Warburger Str. 100
33098 Paderborn

Seminarausarbeitung:

Verteilte Java Echtzeit-Systeme

Im Seminar:

Trends in der Softwaretechnik für Echtzeitsysteme

Dozent: Dr. Holger Giese
SS2003

Katharina Hojenski

Ingenieur-Informatik, Fachsemester 6
Kontakt: kasia@uni-paderborn.de

Paderborn, September 2003

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In verschiedenen Sektoren der Wirtschaft gewinnen Echtzeit-Systeme[1] eine immer größer werdende Rolle. Z.B. wird die Steuerung und Überwachung vieler Vorgänge in der Industrie mittlerweile von Computersystemen übernommen, welche – um rechtzeitig[2] Informationen verarbeiten und auf diese reagieren zu können – über Echtzeit-Eigenschaften verfügen müssen. Echtzeit-Systeme werden aber auch für viele andere Aufgabenbereiche eingesetzt. Zeitkritische Berechnungen innerhalb von Applikationen und die Anforderungen an Systeme, Signale und Ereignisse zeitgerecht verarbeiten zu können, lassen den Bedarf des Einsatzes von Echtzeit-Systemen kontinuierlich steigen.

Die bereits heute existierenden Anwendungen auf Echtzeit-Systemen stoßen immer häufiger an die Grenzen der vorhandenen Ressourcen, insbesondere an die Grenzen der Prozessorauslastung. Die Anwendungen werden zunehmend komplexer, wodurch sich die Anforderungen an die Hardware erhöhen. Ein Ansatz, dem Mangel an Ressourcen entgegenzuwirken, ist die Übertragung der Anwendungen auf verteilte Systeme, im Falle der Echtzeit-Anwendungen auf verteilte Echtzeit-Systeme.

Der Einsatzbereich verteilter Echtzeit-Systeme erstreckt sich von der industriellen Automatisierungstechnik über den Telekommunikationssektor bis zum Verteidigungssektor. In Zukunft werden verteilte Echtzeit-Systeme mit Sicherheit auch in anderen Bereichen der Industrie und Wissenschaft an Bedeutung gewinnen – so auch in den Bereichen, wo die Java™ Technologie eingesetzt wird.

Im Rahmen dieser Seminararbeit werden zwei mögliche Ansätze vorgestellt, Echtzeit-Anwendungen unter Java zu entwerfen: einerseits eine proprietäre Lösung anhand des *Communicating Sequential Processes* (CSP)-Konzepts und auf der anderen Seite *Distributed Real-Time Specification for Java* (DRTSJ), einen - sich noch in Entwicklung befindenden - Standard für eine verteilte Java Echtzeit-Plattform.

1.1 Grundlagen

Der Begriff **Verteiltes System** beinhaltet ein System, dessen Programmiermodell auf existierenden Objekten basiert, die sich auf mehreren physikalisch von einander getrennten Rechnerknoten befinden und miteinander interagieren. Bestimmen zeitliche Bindungen das rechnerübergreifende Verhalten eines Verteilten Systems so handelt es sich um ein **Verteiltes Echtzeit-System**. Alle von der Zeit abhängigen Merkmale und Zeitbindungen wie Deadline[3], erwartete Ausführungszeit, erreichte aktuelle Ausführungszeit usw. werden in dem englischen, für Echtzeit-Systeme charakteristischen, Begriff **Timeliness** (zu Deutsch: Rechtzeitigkeit, Pünktlichkeit) zusammengefasst. An ein Verteiltes Echtzeit-System wird die Forderung gestellt, alle gegebenen Echtzeit-Eigenschaften und Ende-zu-Ende Bedingungen[4] einhalten zu können (**Optimality of Timeliness**). Hinter der Bezeichnung *Optimality of Timeliness* verbirgt die erste Dimension für die Bewertung der Qualität des verteilten Echtzeit-Systems. Eine weitere zentrale Eigenschaft eines Echtzeit-Systems ist die **Vorhersagbarkeit** dessen. In einer zweiten – die Erste umfassenden – Dimension wird vom System verlangt, das Kriterium zu erfüllen, Aussagen über die Fähigkeit der Einhaltung von Zeitbindungen des Systems, unter Beachtung aller auftretenden und das Zeitverhalten beeinflussenden Faktoren, treffen (Vorhersagbarkeit der Rechtzeitigkeit, engl.: *Predictability of Timeliness Optimality*) und auch das Nicht-Einhalten der Zeitbindungen vorherbestimmen zu können.

Neben diesen Anforderungen muss ein verteiltes Echtzeit-System in der Lage sein, auf Ereignisse und Ausnahmesituationen reagieren zu können. Der lokale Verfall einer Zeitkonstante, die Nichterfüllung

[1] **Echtzeit-System** Prozesse erhalten zeitliche Bindungen, das Gesamtsystem ist zeitlich vorhersagbar; Details sind unter [12] im einführenden Seminarthema „Echtzeit-Systeme“ zu entnehmen

[2] **rechtzeitig** heißt in einem (verteilten) Echtzeit-System, dass ein Prozess vor dem Eintreffen einer Zeitschranke terminiert

[3] **Deadline** Zeitangabe oder Zeitschranke, welche die späteste Terminierung eines Prozesses angibt

[4] **Ende-zu-Ende Bedingung** Bedingungen, welche die Interaktion zwischen zwei Netzwerkknoten (Endknoten) festlegen, z.B. die maximale Datenübertragungsdauer u.Ä.

einer Deadline, der Ausfall eines Knotens oder sogar eines ganzen Netzwerkpfades muss wahrgenommen werden. Die sich aus den Ausnahmesituationen ergebenden Fehler müssen von dem System behandelt werden und gegebenenfalls sollte die Ausführung der verteilten Anwendung entsprechend umgeleitet werden. Die Fehlerbehandlung gehört also auch zur Optimierung des gesamten Systems, sie trägt einen großen Faktor zur Vorhersagbarkeit bei.

1.2 Programmiermodell

Verteilte Systeme werden anhand des zugrunde liegenden Programmiermodells kategorisiert. Unter den existierenden Programmiermodellen haben sich im Bereich der Verteilten System folgende bewährt: *networked*, *control flow* und *data flow*. Mindestens eines dieser Modelle muss auf ein Verteiltes System angewendet werden.

Hinter dem Programmiermodell *networked* verbirgt sich das Konzept des Message-Passing. Synchrone oder asynchrone Nachrichten werden zwischen den Einheiten einer Anwendung oder auch zwischen den Knoten eines Netzwerks versendet. Leider ist dieses Konzept nicht mit dem Programmiermodell Javas vereinbar und eignet sich daher nicht zu Integration in verteilte Java Echtzeit-Systeme.
Data flow beschreibt den Datenfluss zwischen den Objekten, ohne den Ausführungsfluss der Anwendung zu betrachten. Dieses Modell eignet sich sehr gut für Anwendungen in denen der Transfer großer Datenmengen stattfindet.
Das *control flow*-Modell bezeichnet den Kontrollfluss, der sich durch Methodenaufrufe zwischen den Objekten eines verteilten Systems ergibt. Es charakterisiert die Bewegung des Ausführungspunktes innerhalb der sich im Programm befindenden Objekte. Dieses Programmiermodell wurde für rechnerübergreifendes Verhalten einer Applikation kreiert. In Java basiert das Kontrollfluss-Modell auf Threads, welche entfernte Methoden aufrufen (*Remote Method Invocation*), und veranlasst diese auf den entfernten Knoten auszuführen.
In einem verteilten System können mehrere nebenläufige Kontrollflüsse auftreten. Sie verfügen über Scheduling-Attribute wie Priorität, Zeitkonstanten, Zeitschranken und Deadlines, die der aktuellen lokalen JVM mitgeteilt werden, sobald der Kontrollfluss die Grenze zu einer JVM überquert.

Andere Programmiermodelle verteilter Systeme wie z.B *Mobile Objects*, *Autonomous Agents* oder *Web Services* - um nur einige zu nennen – sind beim Einsatz auf Systeme beschränkt, welche nicht in Echtzeit agieren.

2 *Communicating Sequential Processes* (CSP) basierte verteilte Echtzeit Java-Anwendungen

Zurzeit gibt es noch keine Standard-Plattform für Java, die es ermöglicht verteilte Echtzeit-Anwendungen unter Java zu implementieren. Eine Spezifikation für eine Verteilte Echtzeit-Java Plattform (Distributed Real-Time Specification for Java) wird derzeit von der Java Community erstellt (siehe Kapitel 3). Um die Java™ Technologie trotzdem in dem Gebiet der verteilten Echtzeit-Systeme einsetzen zu können, bietet das Konzept der *Communicating Sequential Processes* (CSP) eine alternative proprietäre Lösung.

Im Allgemeinen stellt CSP ein fundamentales Konzept dar, um Software für verteilte Echtzeit-Systeme anhand eines Formalismus, der zur Beschreibung von Mustern der Interaktion zwischen parallelen und nebenläufigen Prozessen dient, zu entwickeln. CSP enthält eine komplexe mathematische Theorie zur Spezifikation und Verifikation objekt-orientierter Multiprozessor-Echtzeit-Programme. Bei der Verwendung von CSP findet die Interprozesskommunikation über Kanäle – die sogenannten CSP Channels - statt (siehe [17] und [18]). Durch die CSP Channels entfällt in Java die komplizierte und aufwendige Multi-Threaded-Programmierung und wird durch ein vereinfachtes Kommunikationskonzept abgelöst. Die Behandlung von den bisher einem Prozess oder dem Betriebssystem zugeteilten Aufgaben wie Echtzeit-Priorisierung und Scheduling übernimmt nun das Kommunikationskonzept, welches die Bearbeitung dieser Aufgaben in Form eines CSP Channels kapselt.

Im folgenden Abschnitt wird das CSP Channel Konzept – insbesondere das der Java Channels – genauer erläutert. Das CSP Channel-Konzept für Java wurde an der University of Twente in Zusammenarbeit mit der University of Kent entworfen und implementiert. Die Klassenbibliothek ist unter [19] verfügbar.

Das Grundgerüst der CSP Java-Channels wird hauptsächlich durch zwei Pakete gebildet: JCSP (*Java CSP*) und CTJ (*Communicating Threads for Java*), welches einen eingebetteten Echtzeit-Scheduler enthält. Durch die Einbindung dieser beiden Pakete der Klassenbibliothek wird eine Echtzeit-Java Umgebung geschaffen.

2.1 CSP Channel Konzept und Java Channels

CSP beruht auf der Grundidee CSP Channels für die Interprozesskommunikation paralleler und nebenläufiger Prozesse einzusetzen. Dabei wird von den Kanälen das Scheduling und die Synchronisation der Ein- und Ausgaben dieser Prozesse übernommen. Scheduling-Algorithmen werden vollständig in die Kommunikationskanäle integriert, so dass der Scheduler in die Anwendung eingebettet wird. Eine parallele Anwendung, die aus mehreren nebenläufigen Prozessen besteht, welche über CSP-Kanäle kommunizieren, führt also eine Vielzahl von Schedulern aus. Durch die Eingliederung des Schedulers in die Anwendung, wird der in der Java Virtual Machine (JVM) enthaltene Scheduler überflüssig und könnte entfernt werden, wodurch eine Vereinfachung der JVM erreicht wird.

Damit ein Austausch von Nachrichten oder Daten zwischen zwei Prozessen stattfinden kann, müssen beide Prozesse zur Übertragung bereit sein und ein entsprechender Kommunikationskanal zur Verfügung stehen. Indiziert ein Prozess A, dass er bereit ist Daten an einen Prozess B zu senden und ist Prozess B bereit diese Daten zu empfangen, so werden diese Daten über den Kommunikationskanal übertragen. Eine Kommunikation kann nur dann stattfinden, wenn beide Prozesse diese Bedingung erfüllen. Solange sich nur einer der Prozesse im entsprechenden Zustand befindet, ist die Bedingung nicht erfüllt und der Prozess wartet bis der Andere ebenfalls signalisiert, dass eine Kommunikation stattfinden kann. Bei den Java Channels handelt es sich um Passive Objekte, welche von den Aktiven Objekten – den Prozessen – geteilt werden und über welche sie miteinander kommunizieren. Durch Abbildung 1 wird verdeutlicht, dass die Prozesse lediglich `read()`- und `write()`-Operationen auf dem Channel ausführen dürfen.

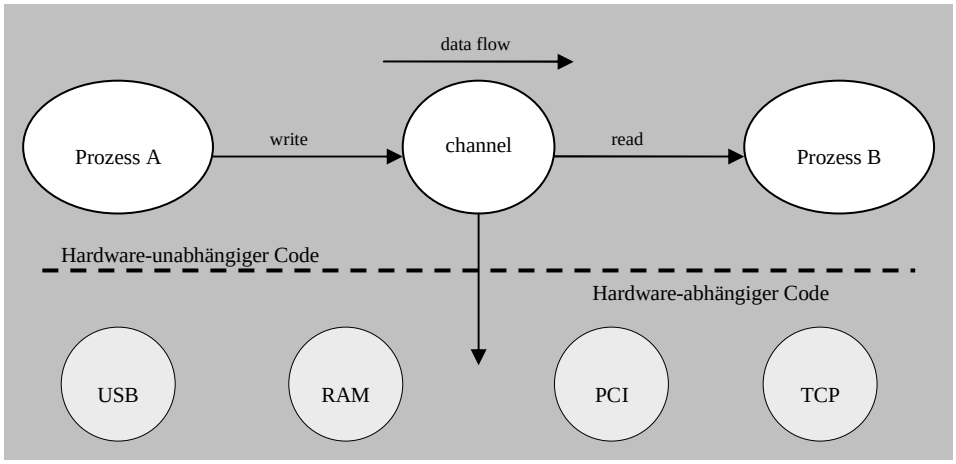

Abbildung 1: Kommunikation über Java Channels; Prozess A sendet Daten und Prozess B empfängt diese über den Kanal; ein Channel führt sowohl hardware-unabhängigen Code () als auch hardware-anhängigen Code () aus

Im folgenden Code-Beispiel (Abbildung 2) ist die Implementierung der Kommunikation zweier nebenläufiger Prozesse über CSP-Channels beispielhaft aufgeführt. Der Producer-Prozess (`ProcessA`, Abbildung 2a) und der Customer-Prozess (`ProcessB`, Abbildung 2b) werden parallel ausgeführt (siehe Main-Programm, Abbildung 2c). Der Producer-Prozess sendet 10.000 Integer-Werte über den Kanal (`ChannelOutput_of_Integer channel`), welche vom Customer-Prozess über den gleichen Kanal (`ChannelInput_of_Integer channel`) empfangen werden.

```
import csp.lang.*;
import csp.lang.Integer;
import csp.lang.Process;
import csp.io.IOException;

public class ProcessA implements Process
{
  ChannelOutput_of_Integer channel;
  public ProcessA(ChannelOutput_of_Integer out)
  {
    channel := out;
  }

  public void run()
  {
    Integer object = new Integer();
    try
    {
      while (object.value < 10000)
      {
        object.value ++;
        channel.write(object);
      }
    }
    catch (IOException e) {  }
  }
}
```

Abbildung 2a

```
import csp.lang.*;
import csp.lang.Integer;
import csp.lang.Process;
import csp.lang.System;
import csp.io.IOException;

public class ProcessB implements Process
{
  ChannelInput_of_Integer channel;
  public ProcessA(ChannelInput_of_Integer in)
  {
    channel := in;
  }

  public void run()
  {
    Integer object = new Integer();
    try
    {
      while (object.value < 10000)
      {
        channel.read(object);
        System.out.println(object.value);
      }
    }
    catch (IOException e) {  }
}}
```

Abbildung 2b

```
import csp.lang.*;
public class Main
{
  public static void main (String[] args)
  {
    Channel_of_Integer channel = new Channel_of_Integer();
    Parallel par = new Parallel(new Process[]
    {
      new ProcessA(channel);
      new ProcessB(channel);
    });
    par.run();
  }
}
```

Abbildung 2c

Abbildung 2a: public class ProcessA. illustriert einen Producer-Prozess `ProcessA` welcher 10.000 Zahlen auf den Kanal schreibt;
Abbildung 2b: public class ProcessB illustriert einen Customer-Prozess. Dieser liest die über den Kanal ankommenden Daten, hier 10.000 Integerwerte und gibt diese aus;
Abbildung 2c: Die Klasse Main. In der `main`-Methode werden die beiden Prozesse A und B erzeugt und parallel gestartet.

Ein Channel darf von mehreren Producer- und Customer-Prozessen genutzt werden, das Scheduling der Prozesse unterschiedlicher Prioritäten sowie die Prozess-Priorisierung werden von dem Java Channel übernommen.

Das in Abbildung 2 aufgeführte Beispielprogramm kann in dieser Form nur auf einem Prozessor ausgeführt werden. Um die Interprozesskommunikation zweier Prozesse auf verschiedenen Prozessoren – also auf einem verteilten System - zu erzielen, muss das Channel-Konzept um sogenannte *Link Driver* erweitert werden, welches im folgenden Kapitel genauer beschrieben wird.

2.2 Link Driver Konzept

Neben der Kommunikation, Synchronisation und dem Scheduling dienen die Channels dazu, die Ressourcen anzusprechen und allen hardware-abhängigen Code in ihnen einzudämmen. Die in Abbildung 2 aufgeführten Prozesse sind hardware-unabhängig. Da diese Prozesse auf einem Prozessor ausgeführt werden, benutzen die Channels für die Übertragung der Daten einen Shared Memory Treiber. Kommunikationskanäle zwischen Prozessen auf verschiedenen Prozessoren benötigen periphere Treiber – so genannte *Link Driver*. Link Driver sind hardware-abhängige Objekte, die als Einzige Zugriff auf die Hardware erlauben. Durch das Konzept der *Link Driver* soll sichergestellt werden, dass hardware-

abhängiger Code nur in diesen vorliegt. Die Link Driver werden an die CSP-Channels (hardware-unabhängige Objekte) gebunden (siehe Abbildung 1 zur Verdeutlichung). So existiert eine klare Abgrenzung von hardware-abhängigem und hardware-unabhängigem Code, um die Eigenschaft der Widerverwendbarkeit, Erweiterbarkeit und Portierbarkeit zu stärken. Bei einer Portierung der Java-Anwendung auf ein anderes Betriebssystem oder ein anderes verteiltes System muss lediglich der entsprechende Link Driver ausgetauscht werden.

Der Transfer der Daten und die dazu benötigten Methoden wie `read()` und `write()`werden im Gegensatz zu dem in Abbildung 2 aufgeführten Beispiel auf den Link Driver übertragen. Um nun einen Channel zwischen Prozessen auf verschiedenen Rechnern zu realisieren, muss die Kommunikation über TCP ermöglicht werden. Dies geschieht über einen TCP/IP Link Driver, für den eine Implementierung unter [20] zu finden ist.

In der Praxis müssen aus der Klasse Main im Beispiel zwei Main-Programme entstehen. Eins auf der Seite des Senders und eins auf der Seite des Empfängers, wobei jedes nur seinen eigenen Prozess startet. Den beiden entstehenden Main-Klassen wird jeweils die *Link Driver*-Klassenbibliothek hinzugefügt

```
import linkdrivers.*;
```

Beim Erzeugen des Channels wird ein Port initialisiert, über den kommuniziert werden soll. Auf Seiten des Empfängers sieht dies folgendermaßen aus (z.B. Port 1701):

```
Channel channel = new Channel (new TCPIP(1701));
```

Zusätzlich muss der Sender die Adresse des Empfängers angeben (z.B. rtat421.rt.el.utwente.nl):

```
Channel channel = new Channel (new TCPIP(rtat421.rt.el.utwente.nl, 1701));
```

Werden beide Programme gestartet, so können über den Channel Daten zwischen den Prozessen auf entfernten Prozessoren übertragen werden. Auf diesem Wege ist die Integration von CSP-Channel-Konzept basierten Anwendungen in verteilte Umgebungen garantiert.

Das *Link Driver*-Konzept sorgt für die Unterstützung der Verteilung bei CSP basierten Anwendungen. Zu untersuchen bleibt noch, inwieweit das CSP-Konzept Echtzeit realisiert.

2.3 Echtzeit-Aspekt

Periodische und aperiodische Prozesse müssen in einer vom System festgelegten Reihenfolge abgearbeitet werden, so dass sie in der Lage sind, ihre Deadlines und Fristen - insbesondere bei harten Echtzeit-Bindungen - einzuhalten. Im CSP-Konzept werden alle Echtzeit-Eigenschaften und Zeitbindungen in Form von Prioritäten ausgedrückt, die als Indizes den Prozessen zugewiesen werden. Damit das System alle Zeitbindungen erfüllen kann, muss ein geeigneter Echtzeit-Scheduling Algorithmus gewählt und in das System integriert werden. Diskrepanzen aufgrund unterschiedlicher Prioritäten zweier Prozesse entstehen nur, wenn diese beiden Prozesse miteinander kommunizieren sollen: Der Prozess höherer Priorität wird solange geblockt, bis der Prozess niedriger Priorität zur Kommunikation bereit ist.
Eine solche Situation kann das sogenannte *Priority Inversion Problem* verursachen, da die Ausführung eines Prozesses, dessen Priorität zwischen den Prioritäten der beiden miteinander zu kommunizierenden Prozessen liegt, vor dem höher priorisierten (blockierten) Prozess stattfindet.
Der gewählte Scheduling-Algorithmus muss die Rechtzeitigkeit der Terminierung möglichst aller Prozesse beachten, aber auch Probleme, wie z.B. das *Priority Inversion Problem*, behandeln können. Da die Kommunikation zwischen den Prozessen über Channels stattfindet und es nur im Rahmen der Kommunikation Unstimmigkeiten in den Prozessprioritäten kommen kann, wird der Scheduler in den Kommunikationskanal integriert. Echtzeit-Priorisierung und Scheduling sind also nicht mehr an das Betriebssystem gebunden, sondern Teil der Anwendung.

Als Echtzeit-Scheduling Algorithmus wurde für die CSP-Channels der *Rate Monotonic Algorithmus* gewählt, da dieser in einer eingeschränkten Ausführung eine optimale CPU-Ausnutzung erlaubt und mit einer festen Prioritäten-Liste arbeitet. Andere mögliche Scheduling-Algorithmen, wie z.B. der *Deadline Driven Algorithmus*, der kontinuierlich seiner Prioritäten-Liste sortiert, verursachen zu viel überflüssige CPU-Auslastung und zusätzlich unnötigen Speicherverbrauch und sind damit für den Einsatz in verteilten Echtzeit-Systemen weniger geeignet. Wie genau das eingesetzte Scheduling-Verfahren funktioniert, kann vom geneigten Leser in [18] nachgelesen werden.

Mit Hilfe dieses in den Channels eingebetteten Schedulers und dem Link Driver Konzept lassen sich die Echtzeit-Zeitbindungen auf einer höheren Abstraktionsebene betrachten. Das CSP-Channel Konzept erlaubt, einen Scheduler als Objekt zur Laufzeit zu laden, wodurch in einer Anwendung auch unterschiedliche Scheduling-Verfahren eingesetzt werden können, welche situationsabhängig effizienter die Reihenfolge der Ausführung der einzelnen Prozesse bestimmen können.

Das CSP-Channel Konzept verwendet als Programmiermodell das Datenfluss-Modell (*data flow*, siehe Kapitel 1.2). Da die CSP-Channels für die zeitgebundene Datenübertragung verantwortlich sind, werden Nachrichten, Signale und Ereignisse in Form von Datenpaketen über die Channels transportiert. Zusätzlich wird an die CSP-Channels die Synchronisations- und Schedulingfunktionalität gebunden, welche dabei das Einhalten der Rechtzeitigkeit bei der Kommunikation sicher stellt, so dass sie innerhalb eines definierten Zeitraums abgewickelt wird. Hinzu kommt, dass lediglich die Kopie eines Objekts, welches über einen Channel übertragen werden soll, zur Zieladresse gesendet wird. Dadurch werden zeitliche Verluste, die beim Klonen oder Serialisieren von Objekten entstehen, verhindert und es wird auf den Einsatz des Garbage Collectors verzichtet, was das (zeitliche) Verhalten der Anwendung deterministischer und vorhersagbarer macht.

2.4 Fazit

Das CSP Channel Konzept bietet die Möglichkeit unter Java Echtzeit Anwendungen zu realisieren, welche über das Link Driver Konzept auch auf verteilten Systemen ablaufen können. Allerdings geht die Vielseitigkeit der Echtzeit-Eigenschaften (Deadline, erwartete Ausführungszeit, usw.) durch das CSP Konzept insofern verloren, dass diese nur über Prioritäten ausgedrückt werden können. Zwar wird über das CSP Channel Konzept das Datenfluss-Modell realisiert (*data flow*, siehe Kapitel 1.2), was die zeitgebundene Kommunikation und rechtzeitige Übertragung von Daten und Signalen sicherstellt, das Echtzeit-Verhalten wird jedoch auf die zeitabhängige Kommunikation reduziert. Es lassen sich keine Aussagen über die rechtzeitige Terminierung eines Prozesses treffen, der mit keinem anderen Prozess interagiert. Da das Datenfluss-Modell darauf reduziert ist, Daten zu übertragen, ist es nicht möglich, auf entfernten Objekten zu operieren und deren Methoden zu nutzen.

Um die Implementierung des Scheduling zwischen zwei miteinander kommunizierenden Prozessen braucht der Systementwickler sich nicht explizit zu kümmern. Es müssen lediglich die relativen Prioritäten der Prozesse definiert werden, da Echtzeit-Scheduling Algorithmen bereits in das Channel Konzept integriert sind und während der Ausführung an die Channels gebunden werden. Jedoch erhöht die Nutzung proprietärer Klassenbibliotheken den Entwicklungsaufwand und kann Probleme in der Portierbarkeit der Anwendung auf ein anderes System verursachen, auf welchem diese Klassenbibliotheken nicht installiert wurden. Da dieses Konzept zur Umsetzung den Einsatz von proprietären Bibliotheken erfordert, sinkt die Praktikabilität für den Entwickler.

CSP ist ein bewährtes Konzept um verteilte Echtzeit-Anwendungen zu konstruieren. Es bietet zwar keine vollständige verteilte Echtzeit-Plattform um unter Java Echtzeit-Anwendungen zu designen, jedoch ist es zurzeit das einzige Verfahren, das diese Möglichkeit zur Verfügung stellt. Eine echte Java Echtzeit-Plattform wird derweil noch spezifiziert, ihr Konzept wird im folgenden Kapitel 3 dargestellt.

3 Distributed Real-Time Java

Die Java™ Plattform sowie die Java Technologie befinden sich in ständiger Entwicklung und werden permanent in Abhängigkeit von den Anforderungen und Bedürfnissen der Java-Nutzer und Entwickler um die entsprechenden Programmierschnittstellen (APIs) erweitert. In Bereichen der Industrie und Wissenschaft entstand der Bedarf Java in verteilten Echtzeit-Systemen einzusetzen. Java bietet dafür gute Voraussetzungen, da einerseits die Schnittstelle zur Verteilten Programmierung durch Konzepte wie RMI, JavaSpaces, JavaParty, Jini usw. eine der Stärken von Java geworden ist und andererseits die Spezifikation der Java Echtzeit-Plattform *Real-Time Specification for Java* (RTSJ) im Jahr 2001 fertig gestellt worden ist. Die Bestandteile des verteilten Systems und die Echtzeit-Umgebung müssen nun in ein Gesamtsystem integriert werden.
Die Entwicklung eines Standards für eine verteilte Java Echtzeit-Plattform *Distributed Real-Time Specification for Java* (DRTSJ) und das Konzept, an welchem sich die Spezifikation orientiert, werden in diesem Kapitel beschrieben.

Eine zentrale Charakteristik für verteilte Echtzeit-Systeme ist die *End-To-End Timeliness*. Zur Beurteilung der Qualität eines verteilten Systems werden zwei Dimensionen bezogen: „Optimality of Timeliness“ und „Predictability of Timeliness Optimality“. Das erstgenannte „Optimality of Timeliness“ umfasst die Einhaltung der Zeitbindungen, welche das System erfüllen muss. Zudem muss ein System die Fähigkeit besitzen, das Einhalten und vor allem das Nicht-Einhalten der Zeitbindungen im Vorfeld bestimmen zu können. `
Auch die sich aus einem möglichen Nicht-Einhalten der Zeitbindung ergebenden Konsequenzen müssen im Voraus vom System erkannt werden. Diese Besonderheit bildet die zweite Dimension der Qualität eines verteilten Systems, die Vorhersagbarkeit der Ersten.
Die DRTSJ-Spezifikation soll ein vollständiges verteiltes Echtzeitsystem erzielen, welches in der Lage ist, die kollektiven Zeitbindungen („collective timeliness“) der Anwendungen zu erfüllen und dessen Kontrollfluss durchgängig rechtzeitig agiert. Als eines der charakteristischen Merkmale von DRTSJ soll die Vorhersagbarkeit der Rechtzeitigkeit des Kontrollflusses mit einer anwendungsbezogenen hohen Sicherheit gewährleistet werden. Somit werden weitestgehend beide Kriterien der Qualität eines verteilten Echtzeit-Systems erfüllt.

Um den Gebrauch von Begrifflichkeiten zu vereinheitlichen und zu präzisieren, wurde während der Arbeit an der *Real-Time Specification for Java* (RTSJ) ein Lexikon zusammengestellt – zu finden in [9] - welches alle Definitionen von Begriffen enthält, die in einem Java Echtzeitsystem eine Rolle spielen (wie z.B. „real-time“, „timeliness“ usw.). Dieses Lexikon wurde für die Zwecke von DRTSJ genutzt und um die entsprechenden benötigten Begriffe erweitert, damit für diese Spezifikation Begriffe ebenfalls in einem eindeutigen Kontext verwendet werden.

3.1 Java Specification Request – 50

Für die Spezifikation einer neuen Java Technologie und ihrer Referenz Implementierung ist die Java Community [1] - eine offene Organisation von Java Entwicklern - verantwortlich. Diese Organisation existiert seit 1998 und sie wurde von Sun Microsystems gegründet, von woher seitdem unterstützt wird. Sie leitet den Prozess (*Java Community Process* – JCP [2]) der technischen Ausarbeitung einer Spezifikation und bewilligt ihre endgültige Fassung. Ihr kann jeder interessierte Entwickler beitreten.
Um einen Standard in Stabilität, Kompatibilität und Portierbarkeit zu gewährleisten geht die Java Community nach einem einheitlichen Ablaufprogramm, dem *Java Community Process*, vor. Entsteht unter den Entwicklern, Organisationen, die die Java Technologie benutzen, oder allgemein in der Industrie der Bedarf einer Erweiterung der Java™ Plattform, wird ein Antrag auf die entsprechende Erweiterung (in Form eines *Java Specification Request* – JSR [3]) verfasst. Für die Erweiterung von Java für verteilte Echtzeit-Anwendungen wurde JSR-50 [4] unter dem Namen *Distributed Real-Time Specification for Java* (DRTSJ) erstellt. MITRE Corporation [5] und IBM Corporation ergriffen überwiegend die Initiative für diesen Antrag.

Ein JSR wird in der Regel von einer Expertengruppe bearbeitet, so auch der Antrag JSR-50. Die Leitung dieser Expertengruppe für DRTSJ wurde von E. Douglas Jensen [6] übernommen. E. Douglas Jensen ist seit 1998 Leiter der Gesellschaft MITRE Corporation, die seit 1959 führend in der Forschung und Entwicklung für komplexe Informationssysteme, im Militärbereich (Befehls- und Kontrollsysteme), Luftfahrt-Kontrollsysteme und Sicherheit und Intelligenz in Verteidigungssystemen ist. Er hat bereits an der Entwicklung der Spezifikation für Echtzeit-Java (*Real-Time Specification for Java* – RTSJ [7]) im JSR-1 [8] mitgewirkt, sowie am Entwurf der Spezifikation von OMG Real-Time CORBA 1 (Fixed Priority Scheduling) und OMG Real-Time CORBA 2 (Dynamic Scheduling) und verfügt somit über ein sehr ausgeprägtes Wissen und langjährige Erfahrungen im Bereich der Verteilten Echtzeit-Systeme. Nahezu alle Informationen und Dokumente, die zu DRTSJ zu finden sind, wurden von E. Douglas Jensen verfasst.
Zu der Expertengruppe des JSR-50 zählen - abgesehen von E. Douglas Jensen - Personen, die ebenfalls als Experten für Verteilte Systeme und Echtzeit-Systeme gelten, die sich bereits am RTSJ - im JSR-1 - beteiligt haben: RMI Experten, Real-Time CORBA Experten, Fachleute aus der Industrie (NASA JPL, Boeing, IBM – um nur einige zu nennen), Akademiker und Wissenschaftler. Mit diesem großen Spektrum von Experten soll sichergestellt werden, dass DRTSJ sowohl für die Nutzung in der Industrie als auch für die Wissenschaft und Forschung geeignet ist und dass durch DRTSJ die erste für kommerzielle Zwecke brauchbare Echtzeit-Plattform für Verteilte Systeme entsteht.

Die Arbeit an der Spezifikation hat zu Beginn des Jahres 2000 begonnen. Zurzeit befindet sich die Spezifikation für Distributed Real-Time Java immer noch in der Entwicklung, eine Fertigstellung wird im Herbst 2003 erwartet.

3.2 Konzept

Bereits beim Erstellen des Antrags JSR-50 wurden an die Spezifikation spezifische Anforderungen gestellt. Da Java über eine Vielzahl von verteilten Umgebungen verfügt und kürzlich die *Real-Time Specification for Java* (RTSJ) freigegeben wurde, wurde die Bedingung gestellt, ein verteiltes Java Echtzeit-System zu entwerfen, in dem die verteilte Umgebung und die Echtzeit-Plattform zu einem Gesamtsystem verknüpft werden. Als verteilte Umgebung soll hierfür Java's *Remote Method Invocation* (RMI) gewählt werden, da es bereits eine weitere Anforderung des JSR-50 – die Anwendung des Kontrollfluss-Programmiermodells (siehe Kapitel 1.2) – erfüllt. Das Konzept der Spezifikation soll die Kompatibilität zu Real-Time Java [8] bewahren. Da viele Fachleute aus der Expertengruppe bereits bei Entwicklung von Real-Time CORBA mitgewirkt haben, verfolgt DRTSJ ein dem Real-Time CORBA sehr ähnliches Konzept.

Die Realisierung des Kontrollfluss-Modells ist eine der fundamentalen Forderungen an DRTSJ. Die Transparenz der Position des Kontrollflusses und seine Bewegung innerhalb der entfernten Objekte soll mit Hilfe der Implementierung des *Distributed Thread* Modells (Kapitel 3.2.3) gewährleistet werden. Das Datenfluss-Modell wird als Programmiermodell in diesem Konzept nicht einbezogen, es kann später in einem weiterführenden JSR ausgearbeitet werden, da dessen Realisierung lediglich eine Erweiterung im Kontrollfluss-Modell erfordert.

Eine der wichtigsten Zielsetzungen bei der Integration von RMI in DRTSJ ist dessen Unterstützung von Echtzeit durch eine möglichst einfache und minimal aufwendige Erweiterung, welche den Echtzeit-Anforderungen genügt und eine durchgängige Rechtzeitigkeit und die Einhaltung der Ende-zu-Ende Bedingungen gewährleistet.
Auf der anderen Seite wird auch vom RTSJ eine Erweiterung verlangt: Das in RTSJ integrierte Scheduling-Verfahren muss den Anforderungen eines verteilten Echtzeit-Systems genügen. Der Scheduler muss die Echtzeit-Eigenschaften der Anwendung berücksichtigen und in Anbetracht dieser den Threads Ressourcen wie Prozessorzyklen, geteilte Ressource, Speicher und auch Kommunikationspfade entsprechend ihrer Notwendigkeit zur Verfügung stellen.
Damit das Gesamtsystem Kenntnis über die Abläufe der einzelnen Knoten besitzt und gegebenenfalls ein globaler Ablauf gesteuert werden kann, muss das verteilte Echtzeit-System über eine globale Systemzeit verfügen. Bei der Integration von RMI und RTSJ wird die Synchronisation der Systemuhren ebenfalls zu einer wichtigen Aufgabe. Um dieses zu gewährleisten muss eine globale Optimierung der

Zeiteigenschaften stattfinden, die mit Hilfe eines einheitlichen Verfahrens zu Ablaufsteuerung erreicht werden kann.
Es gibt drei Möglichkeiten Globalität innerhalb der Ablaufsteuerung zu erzielen: das einfachste Verfahren ist, dass jeder Knoten einfach das gleiche Scheduling-Verfahren benutzt und somit die Reihenfolge des Prozessablaufs auf den Knoten konsistent bleibt; einwenig komplexer wird es, wenn man eine globale Ablaufsteuerung einsetzt, die als Hilfsmittel jeweils ihre eigene Kopie auf jedem Knoten instanziert. Dies erhöht zwar die durchgängige Rechtzeitigkeit, erhöht aber gleichzeitig die Komplexität des Systems. Die dritte Möglichkeit bietet eine „Meta"-Ablaufsteuerung, die oberhalb der Scheduler der einzelnen Knoten operiert und diese global beeinflusst. Diese Art von Ablaufsteuerung ist jedoch die aufwendigste und verursacht unnötige Auslastung der lokalen Scheduler.
Scheduling kann entweder statisch oder dynamisch ablaufen. Statisches Scheduling legt die Reihenfolge der Ausführung der Prozesse vor dem Beginn der Ausführung fest. Dieses Verfahren setzt jedoch im Vorfeld die Kenntnis über alle möglich auftretenden Ereignisse und Umstände voraus, dies beinhaltet Latenzen, Systemauslastung, Betriebssystemabhängigkeiten und Kommunikation. DRTSJ soll erstmal nur das dynamisch Scheduling einbinden, das zur Ausführungszeit in Abhängigkeit der zeitlichen Bindungen der Prozesse die Reihenfolge der Ausführung bestimmt, da es im Allgemeinen nicht möglich ist, all die sich anhand der Infrastruktur ergebenden Eigenschaften im Voraus zu kennen und die auch im Voraus kontrollieren zu können.
Obwohl vorerst nur die Abdeckung der dynamischen Echtzeit-Systeme abgestrebt wird, wird erwartet, dass auch die Portierung zu einem statischen Echtzeit-System einwandfrei verlaufen sollte. Welches Echtzeit-Scheduling-Verfahren eingesetzt und nach welchem Prinzip die globale Ablaufsteuerung umgesetzt wird, wird in den zur Verfügung stehenden Informationen nicht erwähnt und bleibt deshalb noch offen.

Distributed Real-Time Java wird, den Anforderungen gemäß, RMI und RTSJ in einem System verknüpfen. Die folgenden beiden Kapitel geben einen kurzen Überblick über RMI und RTSJ. Es wird grob erläutert, welche Konzepte sich hinter den beiden Plattformen verbergen, welche Möglichkeiten durch RMI und RTSJ gegeben werden und wodurch womöglich Schwierigkeiten und Grenzen bei der Integration der beiden Komponenten zu einem Gesamtsystem entstehen könnten.

3.2.1 Remote Method Invocation – RMI

Die Unterstützung der verteilten Programmierung gilt als eine der Stärken der Programmiersprache Java, Modelle wie Jini, JavaSpaces, Voyager und JavaParty bieten ein weites Spektrum für verteilte Anwendungen, aber vor allem RMI schafft eine Grundlage um das verteilte System um Echtzeit-Verhalten zu erweitern.
Remote Method Invocation erlaubt einen ortsunabhängigen Zugriff auf die Methoden der auf verschiedene Rechnerknoten verteilten Objekte. RMI unterstützt durch ein *Distributed Object System (DOS)* das Modell des Kontrollflusses für entfernte Methodenaufrufe anhand der Definition abstrakter Interfaces. Es bietet einer Java™ Plattform die Möglichkeit mit einer anderen Java™ Plattform zu kommunizieren und serialisierte Objekte zu übertragen und den Zugriff auf Objekte anderer Plattformen anzufordern. Eine Erweiterung von RMI soll die Unterstützung der Echtzeit-Eigenschaften des Programmiermodells ermöglichen, da RMI ursprünglich nicht für Echtzeit-Anwendungen vorgesehen war.

Interfaces in Java stellen einen Mechanismus für die Schnittstelle zwischen einem Client und einem Server bereit. Durch die Implementierung eines Interfaces garantiert der Server die implizierte Funktionalität. RMI verlangt, dass alle Objekte, auf die über RMI zugegriffen werden sollen, das Interface `Remote` implementieren (siehe Abbildung 3) und von der Klasse `RemoteObject` abstammen. Jede Methode eines solchen Objekts muss einen Ausfall anhand einer Ausnahmebehandlung durch eine Deklaration von '`throw RemoteException`' abfangen. Eine solche Exception wird ausgelöst, wenn der Client einen entfernten Aufruf macht und dieser über RMI – aufgrund eines Serverausfalls – nicht ausgeführt werden kann. Um ein Objekt über RMI zu senden, muss es serialisierbar sein, also das Interface `Serializable` implementieren.

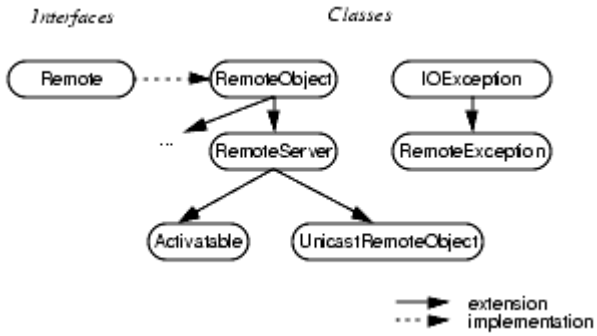

Abbildung 3: Klassenmodell RMI. Alle entferten Objekte auf die Zugriff über RMI erlaubt ist, müssen das Interface `Remote` implementieren. Objekte die über RMI übertragen werden sollen, müssen serialisierbar (`Serializable`) sein.

Vom Programmierer wird keine Kenntnis über den Ablageort des Objekts, auf welches er zugreifen möchte, verlangt – somit ist diese Eigenschaft transparent. Weil aber ein entferntes Objekt das Remote Interface implementieren muss, um über die RMI-Schnittstelle Zugriff darauf zu erhalten, muss der Programmierer explizit wissen, dass es sich um ein entferntes Objekt handelt. RMI bietet für den Entwickler also keine völlige Transparenz. Implementieren alle Objekte der Anwendung das Remote Interface, so ist der Zugriff auf die Methoden eines entfernten oder eines lokalen Objekts völlig transparent.

3.2.2 Real-Time Java – RTSJ

Real-Time Java wurde nur für Ein-Rechner-Systeme konzipiert, erfasst in seinem Konzept sowohl ‚hard real-time' als auch ‚soft real-time'. Der Unterschied zwischen hartem und weichem Echtzeitverhalten wurde in einem anderen Vortrag im Seminar [12] behandelt, und ist daher aus der dazugehörigen Ausarbeitung zu entnehmen. In [12] ist das Thema RTSJ separat bearbeitet worden, sodass Details über das zu Grunde liegende Konzept in der entsprechenden Ausarbeitung nachzulesen sind. In diesem Kapitel wird daher nur ein grober Überblick geschaffen.

Das Ergebnis der Arbeit an RTSJ ist eine Real-Time JVM (RT-JVM). Sie kümmert sich um das Memory Management und übernimmt die Synchronisation. Real Time Java arbeitet mit synchronen Ereignissen und Nachrichten, ermöglich aber auch Asynchronität und die asynchrone Übertragung der Steuerung. Durch zur Verfügung gestellte Mechanismen lassen sich in den Echtzeit-Anwendung die Scheduling- und Release-Parameter setzen und manipulieren.

Die sich hinter RTSJ versteckende Intention war die Schaffung der ersten kommerziell genutzten Echtzeit-Plattform.

3.2.3 Distributed Thread Modell

Distributed Thread Modell bildet einen Ansatz für die Abstraktion des verteilten Kontrollflusses. Dieses Modell tauchte im Zusammenhang mit *Alpha distributed real-time OS kernel's distributed thread model* [21] das erste Mal auf und wird für DRTSJ abgeleitet.
Bei einem Distributed Thread handelt es sich um einen Thread mit einer systemweiten eindeutigen ID, der sich selbst über beliebige lokale und entfernte Objekte erstreckt und wieder zurückzieht. Aus Sicht des Prinzips *Senden/Warten* ist ein Thread weder synchron noch asynchron. Der Aufruf einer entfernten Methode ist nicht mit Übertragung einer Nachricht gleichzusetzen, somit erfolgt auch keine Bestätigung an das aufrufende Objekt. Bei einem solchen Aufruf wird der Ausführungspunkt auf den Knoten der entfernten Methode verlagert. Nach Beendigung des Methodenaufrufs kehrt der Ausführungspunkt (head) des Threads wieder zum ursprünglichen Knoten zurück. Ein Thread überträgt nicht nur all seine Echtzeit-Eigenschaften und -Bindungen, sondern auch das ihm zugeteilte Recht auf Nutzung der Ressourcen, wenn der Ausführungspunkt die Grenzen des Objekts durchquert und teilt diese der aktuellen JVM und dem Scheduler mit.

Einen Thread bezeichnet man als Echt-Zeit Thread, wenn sich in diesem Thread ein Bereich befindet, der mit einer Zeitschranke versehen ist (siehe Abbildung). In der Regel führt ein System eine Vielzahl von zeitlich gebundenen Aktionen, sowie auch eine Vielzahl von zeitlich nicht gebundenen Aktionen aus. Der einfachste Ansatz für den Anwendungsprogrammierer ist, ihm ein Konstrukt zur Verfügung zu stellen, in welchem er die zeitgebundene Ausführeinheit direkt in der Applikation anhand Methoden wie z.B. `BeginTimeConstraint()` und `EndTimeConstraint()` spezifizieren kann. Beim Eintreten und Austreten in den mit einer Zeitschranke charakterisierten Bereich wird ein Scheduling-Ereignis ausgelöst und die Ablaufsteuerung zur Bestimmung der Ausführungsreihenfolge aller existierenden Threads gestartet

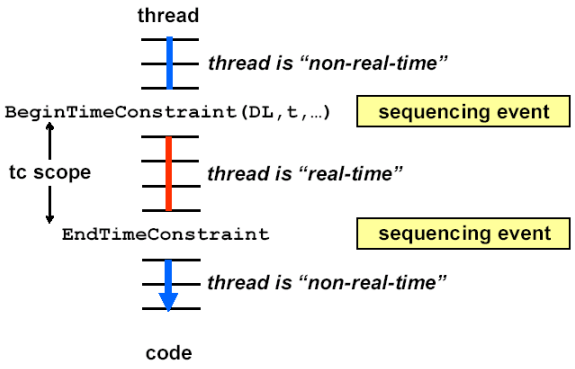

Abbildung 4: Real-Time Thread. In einem Standard Thread wird ein Bereich markiert, der über Echtzeit-Eigenschaften verfügt. Beim Eintreten in diesen Bereicht wird aus dem Standard Thread ein Real-Time Thread.

Ein Distributed Thread ist in der Lage seinen Ausführungspunkt frei über ein Verteiltes System durch Aufrufe entfernter Methoden zu bewegen. Er befindet sich permanent in einem zur Ausführung bereitstehendem Zustand, außer im Falle einer Zurückstellung. Der Ausführungspunkt befindet sich an einer Stelle des verteilten Systems, welche das entfernte Objekt speichert dessen Methode derzeit vom Thread aufgerufen wurde. In einem Verteilten System können mehrere Verteilte Prozesse gleichzeitig laufen.

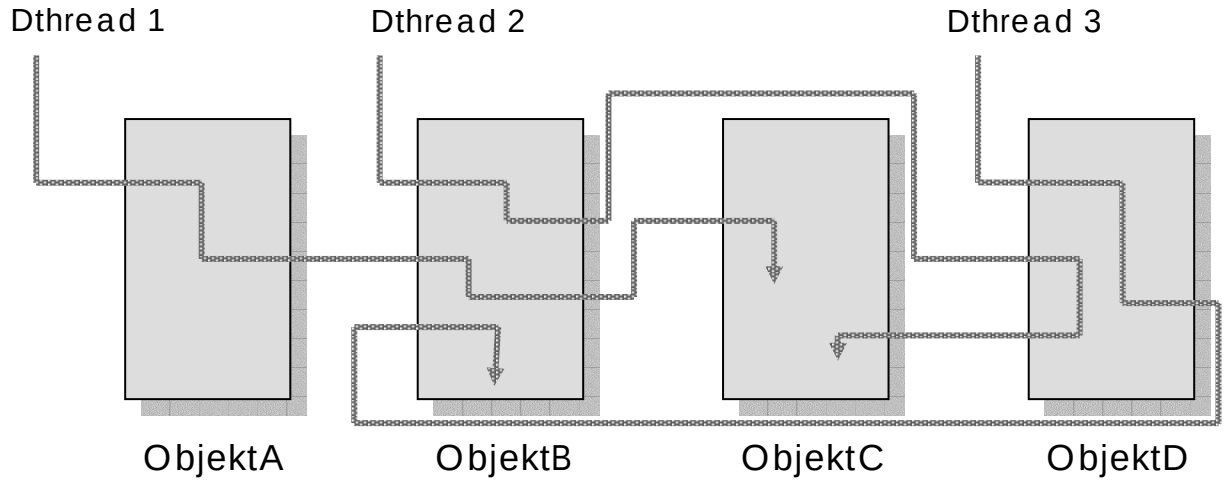

Abbildung 5: Distributed Thread Modell. Abtraktion des Kontrollflusses in einem verteilten System. Der Kopf einer Ausführsequenz erstreckt sich über die auszführenden Methoden der verteilten Objekte eines Threads.

Ein Distributed Thread wird erst dann zu einen Real-Time Thread, wenn er um entfernte Operationen und Eigenschaften erweitert wird, die sich in zwei Bereiche Klassifizieren lassen: 1. ein verteilter Echtzeit-Thread muss Operationen zur Verfügung stellen, mit denen sich das Scheduling des Threads, dessen Ursprung auf einem anderen Knoten liegt, durch einen anderen (lokalen) Thread manipulieren lassen und 2. muss der Status der Ausführung eines entfernten Threads beeinflussbar sein.

3.3 DRTSJ – Realisierung

Im Konzept der Spezifikation wird die Einbindung von RMI und RTSJ gefordert. In diesem Abschnitt werden drei mögliche Modelle zur Integration von RMI und RTSJ vorgestellt, auf deren Grundlage die eine Java™-Plattform für verteilte Echtzeitsysteme aufgebaut werden soll. Bei den drei Modellen handelt es sich um eine stufenweise Erweiterung der Einbindung von Verteilung und Echtzeit zu einem Verteilten

Echtzeit-System. In allen drei Stufen wird die Interoperabilität zwischen der Standard Java Virtual Machine, der Real-Time JVM und der Distributed Real-Time JVM gewährleistet.

Obwohl drei mögliche Vorschläge zur Realisierung von der Java Community innerhalb des JSR-50 erarbeitet wurden, wird im Endeffekt nur einer umgesetzt. Auf welches Modell die Wahl fällt, wurde bis zu diesem Zeitpunkt noch nicht preisgegeben. Ein Vorschlag für den Namen des Packages der API-Spezifikation ist `javax.realtime.distributed`.

3.3.1 Level 0 (Level 0,5) – Remote Interface

Dieser Ansatz – Level 0 – ist eine minimale Integration zwischen RTSJ und RMI. Bei dem Server handelt es sich um keinen Echtzeit-Server, der Client kann anwendungsabhängig muss aber nicht zwangsläufig ein Echtzeit-Client sein. Die Schnittstelle zwischen dem Client und dem Server bildet RMI, das zur Datenübertragung TCP benutzt. Der serverseitige Proxy-Thread wird als ein gewöhnlicher Java Thread betrachtet, der die vom Client benötigten Methoden ausführt, jedoch ohne jegliche Echtzeit-Anforderungen zu erfüllen. Da RMI und die zugrunde liegende Transportschicht ebenfalls keine Echtzeit-Komponenten besitzen, kann weder die Ausführung der Methoden, noch die Überlieferung des Ergebnisses an den Client, in Echtzeit stattfinden. Der Client selbst, ein Echtzeit-Thread, kann entfernte Methoden aufrufen, sich aber nicht auf die Zeitmäßigkeit des Servers verlassen, da dieser keine Kenntnis über die Zeitbindungen des Clients hat. Unter diesen Bedingungen muss ein Anwendungsprogrammierer durch explizite Verdrahtung von Scheduling- und Release-Parameter in der Anwendung Echtzeitverhalten nachbilden. Hierfür kann er die von Java bereitgestellte Klasse `Date` nutzen, da ihre Objekte serialisierbar sind und somit über RMI übertragen werden können.

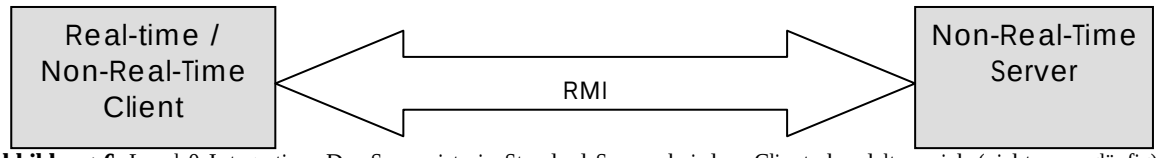

Abbildung 6: Level 0 Integration. Der Server ist ein Standard Server, bei dem Clients handelt es sich (nicht zwangläufig) um einen Real-Time Client. Die Kommunikation findet über die RMI-Schnittstelle statt. Der Server hat keine Kenntnis über die zeitlichen Bindungen des Client-Threads und garantiert somit keine zeitlich gebundene und rechtzeitige Berechnung und Lieferung des Ergebnisses. Die Übertragung über RMI ist ebenfalls nicht durch der Zeit-Konstanten des Clients gesichert.

Die Level 0 Integration verlangt weder eine Erweiterung von RMI noch eine von Seiten des RTSJ. Dadurch wird der zur Integration benötigte Aufwand minimalisiert, bildet aber gleichzeitig die Schwachstelle des Modells, da aus diesem Grunde keine Synchronisation der Systemuhren des Clients und des Servers stattfindet. Sie haben keine Kenntnis über die jeweils vorliegende Systemzeit. Echtzeit wird nur auf Seiten des Clients realisiert. Sobald Kommunikation stattfindet, ist eine rechtzeitige Terminierung des Prozesses nicht mehr garantiert. Im Falle einer Störung werden nur die von RMI zur Verfügung gestellten Maßnahmen ergriffen.

3.3.2 Level 1 – Real-Time RMI

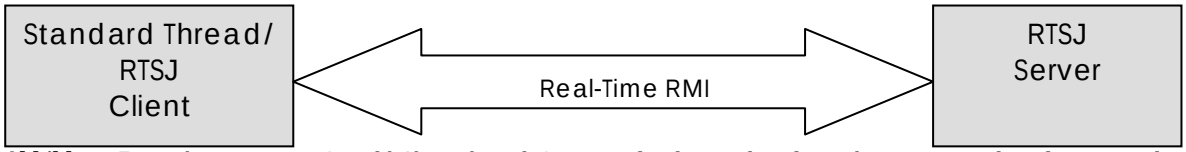

Abbildung 7: Level 1 Integration Sowohl Client als auch Server sind Echtzeit Threads. Real-Time RMI sichert die zeitgerechte Übertragung ab.

In dieser Stufe der Integration von RMI und RTSJ wird Echtzeit-Kommunikation durch die Einführung von Real-Time RMI erzielt. Client und Server arbeiten in Echtzeit unter der Einbindung von RTSJ und übertragen transparent die zeitlichen Bindungen über RMI. Der Proxy-Thread des Servers erbt die Scheduling und Release-Parameters des RTSJ-Client Threads, die über Real-Time RMI übertragen werden. Da der Proxy-Thread sich selbst wie einen gewöhnlichen Real-Time Thread sieht, dürfen seine Scheduling-Parameter von der Real-Time RMI Infrastruktur gesetzt und verändert werden. Ist der Client Thread ein Standard Thread, so werden vordefinierte Standard-Parameter zum Scheduling verwendet.

Real-Time RMI definiert ein `RealtimeRemote` Interface aus dem sich `RealtimeRemote` Objekte implementieren lassen. Sie können in einer Real-Time – Java Virtual Machine (RT-JVM) ausgeführt werden. Das `RealtimeRemote` Interface ist von RMI's Remote Interface abgeleitet.

```
public interface RealtimeRemote extends java.rmi.Remote{};
```

RMI muss für diese Integrationsstufe um Echtzeit-Eigenschaften ausgedehnt werden. Ein Vorteil ist aber, dass in diesem Modell keine Veränderungen an RTSJ und an der RT-JVM nötig sind. Server und Client kommunizieren unter gegebenen Zeitbindungen miteinander. Sie operieren aber zeitlich unabhängig voneinander, da RTSJ nur Ein-Rechner-Systeme betrachtet und es daher nicht möglich ist Beziehung zu den Echtzeit-Systemuhren anderer Knoten und deren Synchronisation herzustellen. Eine weitere Einschränkung, die sich durch diese Integration ergibt, ist die Tatsache, dass keine der RTSJ-Klassen das Remote Interface implementiert – die Objekte sind nicht serialisierbar und können nicht über die RMI Schnittstelle übertragen werden. Dieses Problem macht sich vor allem bei der Übertragung der Zeitbindungen oder der Systemzeit bemerkbar. Da die Objekte nicht zur Kategorie der `Serializable`-Objekte gehören, müssen sie daher vor die Übertragung in ein entsprechendes Format konvertiert werden und serverseitig nach dem Empfangen wieder in ihren Ursprungszustand überführt werden.

Ausfälle auf Seiten des Servers werden dem Client via `RemoteException` mitgeteilt. Ein Ausfall des Clients kann der Server einerseits ignorieren oder ihn in Forme einer RTSJ-seitigen `AsynchronousInterruptException (AIE)` oder eines `AsynchronuosEvent (AE)` behandeln.

3.3.2.1 Grenzen der Level 1 – Integration

Das Level 1 ermöglicht, dass sowohl Server und Client Echtzeit realisieren und die Kommunikation ebenfalls zeitgebunden stattfindet. Es deckt jedoch nicht die kritische Thread-Synchronisation in einem verteilten System ab. Beispielsweise kann sich folgende Situation ergeben:

Ein Client-Thread ruft eine synchronisierte Methode eines Objektes A auf, diese Methode ruft eine Methode eines entfernten Objektes B auf. In dieser Methode des Objektes B wird wiederum eine synchronisierte Methode des Objektes A aufgerufen. Da der JVM nicht bewusst ist, dass es sich dabei um ein und denselben Client-Thread handelt, wird an dieser Stelle ein Deadlock erzeugt, der vom System nicht behandelt wird. Bei solchen verkapselten entfernten Aufrufen von Methoden kann es auch innerhalb eines Objektes oder sogar eines Threads zur Dateninkonsistenz kommen.

Um solchen Problemen zu vorzubeugen, kann dem Thread eine globale ID zugewiesen werden, die bei jedem RMI-Aufruf übertragen wird. Auch eine entsprechende Erweiterung der RTJVM könnte darüber die Erzeugung einer Ausnahme auf einen Deadlock hinweisen.

Durch die bestehende RTSJ ergeben sich für das verteilte Echtzeit-System ebenfalls Einschränkungen im Hinblick auf Synchronisation der Systemuhren und der Übertragung von RTSJ Objekten (wie oben bereits beschrieben).

3.3.3 Level 2 – Distributed Real-Time Threads

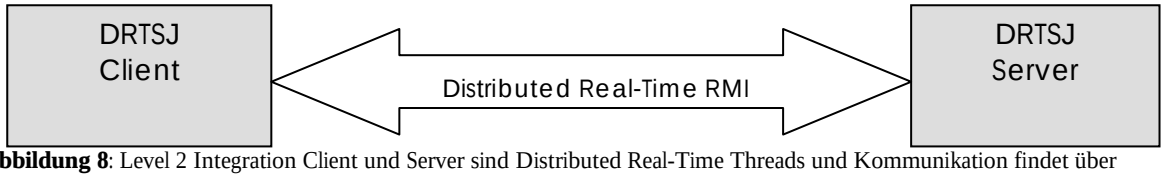

Abbildung 8: Level 2 Integration Client und Server sind Distributed Real-Time Threads und Kommunikation findet über Distributed Real-Time RMI statt. Bei dieser Integrationsstufe ist das Echtzeit-Verhalten der Verteilung und die Synchronisation der Systemuhren garantiert.

Das sich hinter der Level 2 Integration verbergende Konzept, erfordert eine Erweiterung von RMI (Real-Time RMI) zu Distributed Real-Time RMI und den Einsatz von verteilten Echtzeit-Threads (*Distributed Real-Time Threads*) für Server- und Client-Threads (siehe Abbildung 8). Die Level 2 Integration orientiert sich an dem (oben vorgestellten) Distributed Real-Time Thread Modell und setzt dieses zur Abstraktion

des Kontrollfluss-Programmiermodells um. In dieser Form der Integration sind die DRTJVM zweiter Knoten in der Lage, miteinander zu kommunizieren und über das DRT-RMI Objekte auszutauschen oder Methoden entfernter Objekte auszuführen.

Um einen Distributed (Real-Time) Thread unter Java zu formulieren, ist der Entwurf einer Klasse, abgeleitet von `RealtimeThread`, in der das `DistributedRealtimeRemote` Interface implementiert wird notwendig (Abbildung 9). Leider reichen die Fähigkeiten von RT-JVM und Real-Time RMI nicht aus, um einen verteilten Echtzeit-Thread zu realisieren – da dieser keine Kenntnis über die Eigenschaft des Verteilten Systems besitzt, sodass für diesen Zweck RT-JVM zur *Distributed RT-JVM* und RT-RMI zu *Distributed RT-RMI* konvertiert werden muss. Breitet ein Thread sich über eine Vielzahl von Knoten aus, muss jeder Knoten, oder zumindest ein lokaler Thread die Möglichkeit haben, diesen entfernten Thread manipulieren zu können. Um das zu ermöglichen, benötigt die Klasse `RealtimeThread` aus der Bibliothek RTSJ Operationen, durch die sich das Scheduling eines Distributed Real-Time Thread beeinflussen lässt und solche, die Abfrage und die Manipulation des Status der Ausführung eines Distributed Thread betreffen (`interrupt()`, `start()`). Operationen, die die Scheduling- und Release-Parameter betreffen, ermöglichen das Auslesen und das Verändern dieser (`getReleaseParameters()`, `setReleaseParameters()`, `getScheduler()`, `etc.`). Durch diese Methoden wird eine Umgebung erzeugt, die einem Thread erlaubt einen Echtzeit-Thread auf einem entfernten Knoten zu starten, zu unterbrechen oder seine Scheduling-Parameter entsprechend den Bedürfnissen anzupassen.

```
public interface RemoteThread
  extends DistributedRealTimeRemote
{

  public RemoteReleaseParameters getReleaseParameters()
    throws RemoteException;

  public void setReleaseParameters(RemoteReleaseParameters parameters)
    throws RemoteException;

  /* Similarly for SchedulingParameters */

  public RemoteScheduler getScheduler() throws RemoteException;

  public synchronized void interrupt() throws RemoteException;

  public void start() throws RemoteException, IllegalThreadStateException;
}

public class DistributedRealTimeThread extends RealtimeThread
  implements RemoteThread
{
  // Implementation of RemoteThread interface plus
  public static RemoteThread currentRemoteThread()
    throws NotARemoteThreadException;
}
```

Abbildung 9: Schnittstellenbeschreibung: Distributed Thread(Beispiel entnommen aus [14]). Das Konzept der Klasse `DistributedRealTimeThread` erbt die Eigenschaften der `RealtimeThread` und implementiert das Inferface `RemoteThread`

Synchronisation der Systemuhren wird durch einen in die DRT-JVM integrierten Mechanismus übernommen. Eine dem Level 1 ähnliche Ausfallsicherung wird angestrebt, wobei verschiedene Fälle aufgrund der Verteilung betrachtet werden. Fällt gerade der Knoten aus, auf dem sich der aktuelle Ausführungspunkt des Threads befindet, so wird eine RemoteException ausgelöst. Beim Ausfall eines Segments, in das der Thread während der Ausführung wieder zurückkehren muss, wird wie in Level 1 reagiert: eine asynchrone Unterbrechung aufgrund eines Fehlers (AsynchronousInterruptException – AIE) oder ein asynchrones Ereignis (Asynchronous Event – AE) werden ausgelöst, oder der Ausfall wird einfach ignoriert. Dies liegt im Ermessen des Entwicklers.

Die Level 2 - Integration bildet ein vollständiges Verteiltes Echtzeit-System, welches sowohl Kommunikation als auch die Ausführung der Prozesse innerhalb der Echtzeit-Zeitbindungen ermöglich. Leider wird anhand der zur Verfügung stehenden Informationen nicht deutlich, wodurch ein Distributed Real-Time Thread einem entfernten Objekt seine ‚bis zur Terminierung noch zu verbleibende Zeit' signalisiert. Diese Fragestellung ergibt sich z.B. wenn ein Thread, welcher über eine Deadline beschränkt

ist, auf einem Knoten A startet, seinen Ausführungspunkt auf einen Knoten B verlagert, nach Beendigung der Ausführung der von ihm benötigten Methoden wieder zum Knoten A zurückkehrt und dort nach einigen weiteren Aufrufen terminiert. Es wird nicht erläutert, wie der Prozess dem Scheduler des Knotens B mitteilt, dass er auf dem Knoten B nur einen Bruchteil seiner ihm insgesamt zur Verfügung stehenden Ausführungszeit einnehmen darf (Gesamtausführungszeit – Ausführungszeit auf dem Knoten A (– Kommunikationszeit) = Höchstmögliche Ausführungszeit auf dem Knoten B), da er bis zum Eintreffen der Deadline auf dem Knoten A für Aufrufe weitere Ausführungszeit benötigt. Wird die Anpassung der Scheduling-Parameter implizit durch die Anwendung übernommen, sobald der Ausführungspunkt eines Threads auf einen Knoten übertragen wird, oder müssen diese explizit durch den Anwendungsprogrammierer geändert werden? Auch viele weitere Details dieses Modells zur Realisierung einer Integration von RMI und RTSJ, wie z.B. eine Implementierung und Benutzung der APIs aussehen könnte, werden von dem JSR-50 leider nicht preisgegeben.

3.3.3.1 Mögliche Beispiel-Implementierung

Da die Informationen über die mögliche Realisierung des Levels 2 wenig Auskunft darüber geben, wie die Verwendung der Referenz Implementierung tatsächlich aussehen soll, lässt sich über die Praktikabilität dieser Integration erstmal nur spekulieren.

```
import javax.realtime.distributed.*; //Einbindung der Klassenbiblithek
import …                             //weitere benötigte Bibliotheken

public class BeispielThread extends
{
  public static void main(String []args)
  {
    /* es wurden in der Schnittstellenbeschreibung keine Angaben über den Konstruktor
    * gemacht, in diesem Beispiel wird insofern angenommen, dass an den Konstruktor als
    * Parameter z.B. die Adresse des entfernten Knotens übergeben wird. Diese wird beim
    * Starten des main-Programms als erstes Argument angegeben. */

    MyDistributedRealTimeThread myThread = new MyDistributedRealTimeThread(args[0]);

    myThread.run(); // der Thread wird auf dem entfernten Knoten (Adresse in args[0])
                    gestartet
    ….  //Programmausführung

    if (bedingung)
      myThread.interrupt(); // falls eine vorgegebene Bedingung ‚bedingung' erfüllt
                            // ist, wird der Thread unterbrochen
  }

  class MyDistributedRealTimeThread extends DistributedRealTimeThread
  {
  /* diese Klasse ist von der Klasse DistributedRealTimeThread abgeleitet da
  * DistributedRealTimeThread nur über allgemeine Funktionalität verfügt, muss der
  * MyDistributedRealTimeThread durch Überschreiben die spezielle Funktionalität
  * zugewiesen werden */
….
  // hier werden die Methoden der Klasse DistributedRealTimeThread überschrieben
  }
}
```

Bei diesem Codeauszug handelt es sich um ein (sehr lückenhaftes) fiktives Beispiel. Ob die Realisierung eines verteilten Echtzeit-Threads wirklich dem im Beispiel aufgeführten Code entspricht, kann nur vermutet werden. In einem weiteren Beispiel werden die Scheduling-Parameter eines Threads entfernt beeinflusst:

```
// ein Auszug einer möglichen Methode innerhalb einer Klasse, die ein RealTimeRemoteObject
spezifiziert
public class MyRealTimeRemoteObject implements RealTimeRemote
{
… // hier die Implementierung der benötigten Methoden des Interfaces
    public void AendereEtwas() throws Remote Exception
    {
      // der aktuell ausführende Thread wird dem lokalen Objekt t zugewiesen
      DistributedRealTimeThread t = DistributedRealTimeThread.currentRemoteThread();

      // Zur Manipulation der Scheduling-Parameter mit Hilfe von
      //RemoteSchedulingParameters
      RemoteSchedulingParameters p = new RemoteSchedulingParameters(….);

      // Manupulation der lokalen Kopie des RemoteThreads
      t.setSchedulingParameters(p);
      // Manupulation des ursprünglichen Threads
      DistributedRealTimeThread.currentRemoteThread().setSchedulingParameters(p);
      ….
    }
}
```

3.4 Fazit

Stellt man die drei Modelle einander gegenüber, erhält man je Integrationsstufe ein höherwertiges verteiltes Echtzeit-System, vom minimalen verteilten Echtzeit-System (Level 0), wo entfernte Objekte zwar miteinander kommunizieren können aber gar keine Synchronisation der Systemuhren stattfindet, bis zu einer vollständigen verteilten Echtzeit-Plattform (Level 2). Die Einbindung von RMI und RTSJ erfordert in Abhängigkeit von der Integrationsstufe einen höheren Aufwand in der Erweiterung und Anpassung der beiden Plattformen. Pro Integrationsstufe nimmt die Komplexität der Erweitung und die Komplexität des Systems zu. Welches dieser drei Integrationsmodelle letztendlich von der JSR-50 umgesetzt wird, ist noch nicht bekannt gegeben worden.
Aus [15] geht hervor, dass eine Realisierung des Real-Time RMI Modells in Kooperation mit einem Ansatz des *Distributed Real-Time Scheduling* (verteiltes Echtzeit-Scheduling) angestrebt wird. Als Idealfall sollte aber die Level 2-Integration in Betracht gezogen werden, da durch dieses Modell eine vollständige Plattform erzeugt wird – auch wenn das vorgestellte Modell einwenig lückenhaft ist. Durch DRT-RMI wird auf der einen Seite die zeitgebundene Kommunikation zwischen verteilen Objekten sichergestellt und auf der anderen Seite wird die Einhaltung des Echtzeitverhaltens des Gesamtsystems aufgrund des Distributed Real-Time Thread Modells gewährleistet, also vollständiges verteiltes Echtzeit-Verhalten.

4 Zusammenfassung

Java gewinnt in Wissenschaft und Industrie immer mehr Bedeutung und Anwendung. Die Entwicklungen in diesen beiden Wirtschaftsektoren fordern den Einsatz von Echtzeit-Systemen, um deren Performance zu erhöhen und Ressourcen auf kostengünstigerem Wege (als dem Einsatz von Hochleistungsrechnern) zu vermehren. Es wird dazu tendiert, Anwendungen unter Echtzeit auf Verteilte Systeme zu übertragen. Damit Java dem Trend der Entwicklung in der Wirtschaft folgen kann, war die Planung eines verteilten Java-Echtzeitsystems in jedem Fall erforderlich. Das Prinzip CSP-basierter verteilter Anwendungen hat keine Popularität gewonnen. Es stellt zwar Möglichkeiten zur Verfügung verteilte Echtzeit-Anwendugung unter Nutzung der CSP-Bibliotheken zu implementieren, jedoch bildet es keine allgemeine Plattform für ein verteiltes Echtzeit-System. Die Spezifikation einer solchen Plattform war somit unumgänglich.

Die Ausarbeitung der Spezifikation für Distributed Real-Time Java startete um etwa anderthalb Java später, als die Arbeit an der Spezifikation von Real-Time Java, sodass die Spezifikation von Real-Time Java sich zu diesem Zeitpunkt noch in Entwicklung befand, und dennoch sollte Distributed Real-Time Java auf der noch nicht vollständigen Spezifikation aufbauen. In der Zwischenzeit ist die Arbeit an Real-Time Java und ihrer Referenz Implementierung fertig gestellt und befindet sich zurzeit in der Erprobungsphase. An Distributed Real-Time Java wird derweil noch gearbeitet, eine Fertigstellung wird im Herbst 2003 erwartet. Erst wenn die Referenz Implementierung freigegeben ist, besteht die Möglichkeit Aussagen über die Praktikabilität und Effizienz von Distributed Real-Time Java zu treffen. Ohne deren Referenz Implementierung fällt es schwer vorauszusagen, inwieweit sich die Real-Time Java Technologie und die Distributed Real-Time Java Technologie in der Wissenschaft, Forschung und Industrie einführen lassen und ob sich diese auch etablieren werden.
Da eine zeitintensive und lang andauernde Testphase notwendig ist, wird es mit Sicherheit einige Jahre dauern, bis eine stabile und vollständige verteilte Echtzeit-Java Implementierung entsteht welche sich unter den verteilen Echtzeit-Plattformen durchsetzt und eine feste Stellung auf dem Markt einnimmt.

Literatur

[1] The Java Community Process, http://jcp.org/en/home/index

[2] The Java Community Process, http://web1.jcp.org/en/introduction/overview

[3] The Java Specification Request, http://web1.jcp.org/en/jsr/overview

[4] Java Specification Request, JSR-50, *Distributed Real-Time Specification for Java (DRTSJ)*
http://web1.jcp.org/en/jsr/detail?id=50

[5] MITRE Corporation, http://www.mitre.org

[6] E. Douglas Jensen, http://www.real-time.org/aboutme.htm

[7] Offizielle Internet-Seite der Real-Time Specification for Java RTSJ, http://www.rtj.org

[8] Java Specification Request, JSR-01, *Real-Time Specification for Java (RTSJ)*
http://web1.jcp.org/en/jsr/detail?id=1

[9] Spezifikation RTSJ auf der offiziellen Webseite von Real-Time Java
http://www.rtj.org/rtsj-V1.0.pdf

[10] Offizielle Seite der Distributed Real-Time Specification for Java, http://www.drtsj.org; http://pitfall.mitre.org/

[11] RMI – Remote Method Invocation, http://java.sun.com/products/jdk/rmi/

[12] Homepage des Seminars „Trends in der Softwaretechnik von Echtzeit-Systemen“;
http://www.uni-paderborn.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Seminare/SWTfEZ/

[13] E. Douglas Jensen, *The Distributed Specification for Java – An Initial Proposal*;
http://citeseer.nj.nec.com/cache/papers/cs/26160/http:zSzzSzwww.real-time.orgzSzpreviewzSz..zSzdocszSzcsse.pdf/the-distributed-real-time.pdf;
http://pitfall.mitre.org/pdfcontent/csse.pdf

[14] Andy Wellings, Ray Clark, Doug Jensen, Doug Wells; *The Distributed Specification for Java- A Status Report*;
http://www.it.uc3m.es/mvalls/asignaturas/ad/trabajos/esc02paper.pdf

[15] Andy Wellings, Ray Clark, Doug Jensen, Doug Wells; *A Framework for Integrating the Real-Time Specification for Java and Java's Remote Invocation*; http://citeseer.nj.nec.com/wellings02framework.html;
http://citeseer.nj.nec.com/cache/papers/cs/26160/http:zSzzSzwww.real-time.orgzSzpreviewzSz..zSzdocszSzisorc02_v41.pdf/wellings02framework.pdf

[16] DRTSJ, *Overview and Status*, September 2002, http://citeseer.nj.nec.com/543008.html

[17] André Bakkers, Gerald Hilderink und Jan Broenink; *A Distributed Real-Time Java System Based on CSP*, The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2000,
www.ce.utwente.nl/javapp/information/CTJ/DRTJSBCSP.pdf

[18] André Bakkers, Gerald Hilderink und Jan Broenink; *A Distributed Real-Time Java System Based on CSP*, Architectures, Languages and Techniques, B.M. Cook (Ed.), IOS Press 1999,
http://www.ce.utwente.nl/bnk/papers/RTjava-wotug22.pdf

[19] CSP for Java, http://www.ce.utwente.nl/javapp/

[20] TCP/IP Link Driver http://www.ce.utwente.nl/javapp/Plug-and-Play/TCPIP_linkdriver/

[21] Raymond K. Clark, E. Douglas Jensen, Franklin D. Raynolds, *An Architectural Overview of the Alpha Real-Time Distributed Kernel*, http://www.isis.vanderbilt.edu/projects/core/bibliography/core/AlphaRtDistributedKernel.pdf

Archiv:

RTjava-wotug22.pdf	-	Artikel [18], André Bakkers, Gerald Hilderink und Jan Broenink; *A Distributed Real-Time Java System Based on CSP*,
csse.pdf	-	Artikel [13], E. Douglas Jensen, *The Distributed Specification for Java – An Initial Proposal*
esc02paper.pdf	-	Artikel [14], Andy Wellings, Ray Clark, Doug Jensen, Doug Wells; *The Distributed Specification for Java- A Status Report*
15580013.pdf	-	Artikel [15], Andy Wellings, Ray Clark, Doug Jensen, Doug Wells; *A Framework for Integrating the Real-Time Specification for Java and Java's Remote Invocation*
clark93architectural.pdf	-	Artikel [21], Raymond K. Clark, E. Douglas Jensen, Franklin D. Raynolds, *An Architectural Overview of the Alpha Real-Time Distributed Kernel*