Java Komponenten für ein verteiltes Echtzeitsystem

Jan Richling, Janek Schwarz, and Andreas Polze

Humboldt-Universität zu Berlin
Institut für Informatik
12489 Berlin
Rudower Chaussee 5
Email: {richling,schwarz,apolze}@informatik.hu-berlin.de

Zusammenfassung Die Programmiersprache Java mit ihrem architekturunabhängigem Abarbeitungsansatz bietet viele Vorteile, wie Sicherheit in verteilten Umgebungen, Wiederverwendbarkeit von Code und Portabilität. Diese Charakteristika führen zum Einsatz von Java in vielen neuartigen Umgebungen.
Obwohl Java viele Vorteile bietet, ist der Einsatz der Sprache in Fällen kritisch, in denen Echtzeitverhalten gefordert ist. Die existierenden Java-Ausführungsumgebungen und Sprachspezifikationen reichen für das Gewähren von Echtzeitgarantien nicht aus. Andererseits existieren auch in einem Echtzeitsystem Komponenten, an deren zeitliches Verhalten keine strengen Forderungen gestellt werden und für deren Implementation sich Java anbietet.
Unsere Forschung richtet sich auf Mechanismen zur Verknüpfung von Java-Komponenten mit (existierenden) Echtzeitsystemen. In diesem Artikel beschreiben wir die Verwendung des „Composite Objects"-Ansatzes zur zeitlichen Entkopplung und Interoperabilität zwischen Java-Komponenten und dem Echtzeitsystem. Wir diskutieren die Erweiterung eines bestehenden Echtzeitsystems um offene Schnittstellen.
Die „Unstoppable Robots" stellen eine verteilte, responsive (echtzeitfähige, fehlertolerante) Demo-Applikation auf Basis des Betriebssystems Mach dar. Wir demonstrieren die Erweiterung der Applikation um in Java geschriebene Komponenten. Diese Komponenten, eine graphische Benutzerschnittstelle und ein Controller, unterliegen unterschiedlich strengen Echtzeitanforderungen. Wir evaluieren die Leistung der erweiterten Demo-Applikation und zeigen die Brauchbarkeit unseres Ansatzes.

1 Einführung, Motivation

Java [1] wurde von Sun Microsystems entwickelt und ist in den vergangenen Jahren zunehmend populär geworden. Java ist eine objektorientierte Programmiersprache mit Ähnlichkeiten zu C++ und wurde für den Einsatz in weltweit verteilten Umgebungen, wie dem World Wide Web, entworfen.

Aufgrund ihrer Charakteristika wäre der Einsatz der Sprache Java auch in neuen Bereichen, wie Steuerungssystemen und Internet-Geräten wünschenswert.

Solche Systeme interagieren mit der wirklichen Welt. Typischerweise erfordern sie gewisse Echtzeitgarantien.

Weder der Sprachstandard noch gegenwärtige Implementationen von Java bieten Unterstützung für Echtzeitgarantien in Java-Programmen. Der Kern einer Echtzeitanwendung wird also auf einer anderen Plattform implementiert werden. Andererseits ist Java interessant, um eine Echtzeitanwendung mit offenen Schnittstellen ausstatten zu können. Weniger zeitkritische Komponenten können dann an eine existierende Echtzeitanwendung angebunden - und plattformunabhängig in Java programmiert werden.

Die „Unstoppable Robots“ [2] stellen eine verteilte, responsive (echtzeitfähige, fehlertolerante) Demo-Applikation dar, die vor einiger Zeit von uns entwickelt wurde. Die Applikation simuliert eine Menge von Robotern, die auf einer Platte im instabilen Gleichgewicht balancieren. Die Roboter verbrauchen Energie mit einer einstellbaren, zeitabhängigen Rate, sie müssen daher gelegentlich eine „Futterstelle“ anlaufen. Bei Nichteinhalten der Echtzeitanforderungen „verhungern“ die simulierten Roboter.

Die originale Version der „Unstoppable Robots“ basiert auf dem Betriebssystem Mach (NeXTSTEP 3.3) und benutzt replizierte Programmausführung zur Erzielung von Fehlertoleranz. Machs Interprozeß Kommunikation (IPC) wird zum Datenaustausch zwischen den Komponenten der Applikation benutzt.

Wir demonstrieren nun die Erweiterung der „Unstoppable Robots“ um in Java geschriebene Komponten. Diese Komponenten greifen in die Steuerschleife der Demo-Applikation ein. Zentraler Punkt ist dabei die Gewährleistung garantierten Zeitverhaltens – auch in der erweiterten Demo-Applikation dürfen Roboter nicht „verhungern“. Dazu müssen zwei gegensätzliche Anliegen - die Echtzeitsteuerung und die Ansteuerung der offenen Java Komponenten - voneinander entkoppelt werden. Wir benutzen dafür die „Composite Objects“-Technik. Schließlich evaluieren wir das zeitliche Verhalten der erweiterten „Unstoppable Robots“-Applikation unter variierenden Lastsituationen.

Die weiteren Abschnitte dieses Artikels gliedern sich wie folgt: Abschnitt 2 gibt zunächst einen Überblick über verwandte Arbeiten. In Abschnitt 3 diskutieren wir unseren Ansatz zur Integration von Java-Komponenten in ein existierendes Echtzeitsystem. Abschnitt 4 beschreibt die erweiterte „Unstoppable Robots“-Applikation als Fallstudie, während Abschnitt 5 Implementationsaspekte und die Evaluierung unseres Ansatzes präsentiert. Mit Schlußfolgerungen in Abschnitt 6 schließt der Artikel.

2 Verwandte Arbeiten

Unser „Composite Objects“-Ansatz [3] zeigt einen Weg zur vorhersagbaren Einbindung von Java-Komponenten in ein verteiltes Echtzeitsystem auf. Die Grundidee unseres Ansatzes ist die saubere Trennung verschiedener Anliegen: Java und das Echtzeitsystem sollen interagieren können, sich aber nicht gegenseitig in ihrem zeitlichen Verhalten beeinflussen.

Andere Arbeiten beschäftigen sich damit, die Sprache Java zu erweitern und die zugehörige Abarbeitungsumgebung echtzeitfähig zu machen. Für die verteilte Programmierung in Java bietet sich CORBA (Common Object Request Broker Architecture) [4][5] an. Mit dem Projekt der „Java Beans“ existiert sogar die Idee, Javas eigenen Mechanisums zur entfernten Methodenausführung (RMI - Remote Method Invocation) standardmäßig auf CORBA abzubilden. Wir betrachten daher auch verwandte Arbeiten zu „Echtzeit-CORBA“.

2.1 Echtzeit-Java

In [6] wird die Implementation einer echtzeitfähigen virtuellen Maschine für Java diskutiert und evaluiert. Grundlage für die beschriebene Implementation ist das RT-Mach Betriebssystem. Es werden ein echtzeitfähiges thread-Modell für Java, Echtzeitsynchronisation in der virtuellen Maschine und ein Mechanismus für die Behandlung von Zeitfehlern (timing faults) vorgeschlagen. Allerdings bedeutet die Neuimplementation der virtuellen Java-Maschine und die Einführung eines neuen thread-Modells die teilweise Aufgabe der Portabilität der Programmiersprache Java.

Es gibt eine Reihe weiterer Arbeiten, die sich mit Erweiterungen verschiedener Programmiersprachen zum Ausdrücken von Echtzeitverhalten beschäftigen. Darunter fällt das ART Projekt an der Carnegie Mellon University, das sich mit der C++ Erweiterung RTC++ [7] beschäftigt. Real-Time Concurrent C [8] und FLEX [9] sind weitere Projekte, in denen Spracherweiterungen für C behandelt werden. Im Java-Kontext interessant ist die Arbeit von Nilsen[10].

Nilsen schlägt eine neue Programmierschnittstelle für Echtzeitanwendungen vor. Neue Anweisungen wie *timed* und *atomic* wurden eingeführt, um zeitliches Verhalten für die Ausführung von Code-Blöcken zu spezifizieren. Allerdings ist zur Programmausführung ein spezieller Compiler oder Präprozessor nötig.

Problematisch bei all diesen Ansätzen ist die Tatsache, daß die Idee der Spracherweiterung nur für homogene Systeme funktioniert, die völlig neu erstellt werden. Unser Ansatz konzentriert sich eher auf existierende Echtzeitsysteme, für deren Erweiterung offene, in Java geschriebene Komponenten eingesetzt werden sollen. Wir beschreiben hier, wie die Anbindung der Java-Komponenten ohne Störung des Echtzeitverhaltens des bestehenden Systems möglich ist.

2.2 Echtzeit-CORBA

Im Jahre 1995 wurde von der Object Managment Group (OMG) die „Realtime CORBA“ *special interest group* (SIG) gebildet. Etliche Dokumente zum Thema Echtzeit-CORBA werden derzeit bearbeitet, darunter ist ein *white paper* auf dem Stand eines *Initial Review Draft* (Nov., 15, 1996) [11]. Im September 1997 wurde ein *Request for Proposal* (RFP) für die *fixed priority*-Version von Realtime CORBA herausgegeben. Gesucht werden Technologien für einen *Realtime Object Request Broker*, die *fixed priority scheduling*, End-zu-End Ressourcereservierung und flexible Kommunikationsschnittstellen unterstützen. Allerdings ist nocht nicht klar, inwieweit RT-CORBA zur Standard-CORBA Spezifikation konform sein wird.

Unter dem Namen TAO [12][13] wurden wegweisende Arbeiten zum Thema RT-CORBA ausgeführt. TAO integriert *fixed priority real time scheduling*-Techniken in die CORBA Architektur. Hauptziel der Arbeiten ist es, End-zu-End Dienstgütegarantien für CORBA-basierte Anwendungen zu geben. Im TAO Projekt wurde ein Anforderungskatalog für *Object Request Broker*-Implementationen aufgestellt; dieser umfaßt Protokolle zur Ressourcereservierung, optimierte Echtzeitkommunikationsprotokolle und einen echtzeitfähigen *Object Adapter*. Allerdings ist das Hauptziel der TAO-Arbeiten die Anwendung von CORBA für den Aufbau neuer, verteilte Echtzeitsysteme.

Arbeiten an der University of Rhode Island und bei der MITRE Corporation behandeln Spracherweiterungen der CORBA Schnittstellensprache IDL zum Ausdruck von Zeitanforderungen [14] [15]. *Timed distributed method invocations* werden als notwendige Bedingung für verteiltes, CORBA-basiertes Echtzeitrechnen charakterisiert. Ein globaler Zeitdienst, Echtzeitscheduling von Diensten, ein globaler Prioritätsdienst und begrenzte Nachrichtenlaufzeiten werden als Vorbedingungen für den beschriebenen Ansatz aufgeführt. Auch in diesen Arbeiten wird die Interaktion von Echtzeit- und Standard CORBA-Komponenten nicht gezielt untersucht.

3 Java-Komponenten für ein Echtzeitsystem

In einer Reihe von Anwendungen ist die Kombination von Java-Technologie und Echtzeitsystemen sinnvoll und möglich. Als Grundlage der Kommunikation der Java-Komponenten mit dem Echtzeitsystem wird der plattformunabhängige Standard CORBA benutzt. Fehlende Echtzeitunterstützung auf Seiten von Java wie auch bei der Kommunikation mit Hilfe von CORBA müssen bei der Erweiterung eines Echtzeitsystemes um CORBA-Kommunikation und Java-Objekte berücksichtigt werden.

Grundsätzlich besteht ein Echtzeitsystem aus zeitkritischen Bereichen wie der inneren Kontrollschleife und peripheren Bereichen, wie beispielsweise Anzeigen, die sich außerhalb der zeitkritischen Bereiche befinden. In beiden Fällen gibt es verschiedene Auswirkungen der fehlenden Echtzeiteigenschaften der Java-Komponenten, die in ihrer Wirkung untersucht werden müssen.

3.1 Außerhalb der zeitkritischen Bereiche

Komponenten, die sich außerhalb der zeitkritischen Bereiche eines Echtzeitsystems befinden, unterliegen keinen harten Echtzeitanforderungen. Für den Betrieb des Echtzeitsystems ist es meist keine Gefährdung, wenn von diesen Komponenten Zeitanforderungen gelegentlich nicht eingehalten werden. So ist es tolerierbar, wenn eine Anzeige einzelne Werte gar nicht oder mit geringer Verzögerung anzeigt (weiche Echtzeit).

Aus Sicht des Echtzeitsystems ist lediglich bedeutsam, daß die hinzugefügte Komponente unter keinen Umständen den Betrieb des Systems stört – also keine Einflüsse auf die zeitkritischen Bereiche haben darf. Prinzipiell gibt es zwei Möglichkeiten, Komponenten, wie beispielsweise eine Anzeige, via CORBA in ein System einzubinden. Aus Sicht von CORBA sind das Client- oder Serveransatz; aus Sicht der Kommunikation *poll*- und *push*-Technologie.

poll-Technologie. Bei diesem Ansatz ist das Java-Objekt CORBA-seitig ein Client, der Anfragen an einen CORBA-Server stellt. Dieser ist dann mit dem Echtzeitsystem verbunden. Vorteil ist, daß auf diese Weise problemlos viele Clients bedient werden können, Nachteile liegen in der Lastsituation auf der Echtzeitseite, die durch Anfragen mit zu hoher Frequenz entstehen kann. Ein Beispiel ist eine Java-Anzeige, die in bestimmten zeitlichen Abständen Werte anfordert. Das Vorhandensein zu vieler derartiger Anzeigen kann zu Überlastsituationen führen.

Essentiell bei der *poll*-Technologie ist ein *Firewall* zwischen dem Echtzeitsystem und der übrigen Welt. Jede CORBA-Anfrage wird vom *Firewall* bearbeitet, ehe Datenverkehr mit dem Echtzeitsystem stattfindet.

Auf dieser Basis ist es möglich, verschiedene Strategien zu implementieren:

- *Caching.* Ausgehend von der Natur der angeforderten Daten werden Anfragen nur dann an das Echtzeitsystem weitergegeben, wenn die letzte gleichartige Anfrage mehr als eine festgelegte Zeit zurückliegt. Problem ist, daß trotzdem eine sehr starke Belastung der äußeren Seite des *Firewalls* dazu führen kann, daß das System überlastet wird. (Auch abgewiesene Requests werden durch den Protokoll-Stack bearbeitet und erzeugen damit Last.)
- *Auslagerung auf ein anderes System.* Der *Firewall* implementiert caching, läuft selbst aber auf einem unabhängigen System, dessen Lastsituation keinen Einfluß auf das Verhalten des Echtzeitsystems hat. Die Trennung liegt hier in einer Rechnergrenze, über die mit Echtzeitkommunikation nur eine vom *Firewall* begrenzte Datenmenge weitergeleitet wird.
- *Call admission.* Die vom *Firewall* erzeugte Last wird mit Mitteln des Betriebssystems beschränkt. Das können entweder niedrigere Prioritäten sein, oder aber Partitionierung der Rechenleistung beispielsweise durch den Scheduling Server[16][17].

push-Technologie. Die Java-Komponente ist in diesem Fall CORBA-seitig ein Server, der vom Echtzeitsystem Werte zugeschickt bekommt. Es ist durch CORBA nicht gewährleistet, wie lange die Kommunikation dauert, und auch Java garantiert keine Ausführungszeiten. Es ist also nicht bekannt, wie lange der auf Seiten des Echtzeitsystems ausgelöste CORBA-Methodenaufruf benötigt.

Die Komponente, die im Echtzeitsystem das Versenden der Daten übernimmt, muß ebenfalls als *Firewall* ausgeführt sein, der Zugriffe auf die Daten des Echtzeitsystems und die CORBA-Kommunikation zeitlich entkoppelt.

Eine Überlastung des Echtzeitsystems verhindert der *Firewall*, da die Frequenz der CORBA-Aufrufe im System unter Beachtung der gegebenen Umgebung und der maximal möglichen Last festgelegt werden kann.

Ein Blockieren der CORBA-Kommunikation oder des Java-Objektes kann damit schlimmstenfalls zum Blockieren des *Firewalls* führen, hat aber keinen Einfluß auf das Verhalten des Echtzeitsystems, da der *Firewall* kein Teil der zeitkritischen Bereiche ist.

3.2 Innerhalb der zeitkritischen Bereiche

Für die innere Steuerschleife eines Echtzeitsystems existieren oftmals harte zeitliche Anforderungen. Hier muß es garantierte Obergrenzen für die Ausführungszeit aller beteiligten Komponenten geben. Diese Anforderung können derzeit weder Java noch die CORBA-Kommunikation erfüllen. Trotzdem gibt es sinnvolle Anwendungen, bei denen externe Komponenten wie beispielsweise Java-Objekte in Vorgänge der zeitkritischen Bereiche eingreifen sollen. Beispiele sind Fernsteuerungen, oder die Auslagerung von Funktionalität auf externe Systeme.

Die Benutzung von *Firewalls* nach dem „Composite Object"-Ansatz [3] bietet die Möglichkeit, Java-Objekte und zeitkritische Bereiche eines Echtzeitsystems so miteinander zu verbinden, daß sich beide nicht in ihre Zeitverhalten beeinflüssen. Das Hauptproblem liegt in der nicht bekannten Ausführungszeit von Methodenaufrufen über CORBA, die auch die ebenfalls nicht bekannte Ausführungzeit der Java-Methode enthält. Beispielsweise führt eine hohe Last auf der Maschine, auf der das Java-Objekt läuft, auch zu einer Verlängerung dieser Ausführungszeit.

Aufgabe des *Firewalls* ist es, die Auswirkungen dieses Verhaltens zu kapseln. Da der *Firewall* keine Möglichkeit hat, die entfernten Vorgänge zu beeinflussen (replizierter Aufruf kann die Ausführungszeit unter Umständen verkürzen, dies aber nicht in einer garantierten Art und Weise), muß es einen Fallback-Algorithmus geben, der die Funktionalität des externen Objektes in einer abgerüsteten Version auf dem *Firewall* für den Fall bereitstellt, daß das entfernte Java-Objekt nicht in der Lage ist, seine Ergebnisse zeitgerecht zu liefern. Abgerüstet bedeutet an dieser Stelle, daß der Algorithmus Ergebnisse liefert, die rechtzeitig kommen und dem System keinen Schaden zufügen. Somit existieren entsprechend dem Konzept des *Multi-Version-Programming* zwei Algorithmen, die sich in Komplexität, Ausführungszeit und Lokation unterscheiden. In Abhängigkeit vom benutzten Fallback-Algorithmus und den Eigenschaften der Applikation können Obergrenzen für die maximal mögliche Anzahl von aufeinanderfolgenden Ausführungen des Fallback-Algorithmuses angegeben werden.

Praktisch funktioniert diese Idee, indem der *Firewall* den CORBA-Methodenruf auslöst und danach wartet. Kehrt er rechtzeitig vor Ablauf einer Time-out-Zeit zurück, wird das Ergebnis benutzt, ansonsten wird innerhalb der zeitkritischen Bereiche das Ergebnis des Fallback-Algorithmus benutzt. Damit ist sichergestellt, daß die Zeitschranken der zeitkritischen Bereiche eingehalten werden. Die Time-out-Zeit muß dabei in Abhängigkeit von den zeitlichen Schranken der zeitkritischen Bereiche gewählt werden.

Um die Ausführungszeiten und deren Varianz bei Aufruf der Methoden des Java-Objektes zu verringern, gibt es Möglichkeiten auf Betriebssystemebene. Der von uns in einem früheren Projekt entwickelte Scheduling Server [16][17] erlaubt die Vergabe eines wohldefinierten Teils der CPU-Leistung an die virtuelle Java-Maschine. Ein anderer, weniger stabiler Weg ist die Erhöhung der Priorität der Java-Maschine.

4 Fallstudie: Unstoppable Robots

Basis unserer Untersuchungen zur Anbindung von Java-Objekten an ein bestehendes Echtzeitsystem sind die „Unstoppable Robots“, ein Beipiel für eine responsive Anwendung.

Die Idee der „Unstoppable Robots“ ist es, eine Reihe von Robotern zu simulieren, die sich auf einer beweglich gelagerten Platte bewegen. Jeder Roboter verbraucht Energie, wobei die Höhe des Energieverbrauches davon abhängig ist, ob sich ein Roboter bewegt oder nicht. Auf der Platte befindet sich eine Station, an der die Roboter ihre Energievorräte erneuern können. Die Aufgabe der Roboter ist es, die Balance der Platte zu sichern und selber am Leben zu bleiben, d. h. sie müssen vermeiden, ihre gesamte Energie aufzubrauchen. Das Ausbalancieren der Platte hat dabei Priorität. Die Simulation ist in der Lage, (derzeit) einen realen Roboter synchron zu den simulierten Robotern zu steuern.

Die Anwendung besteht aus zwei Teilen, der Anzeige, die diese Welt abbildet, und einer Menge von Controllern, die auf verschiedene Rechner verteilt sein können. Ein Roboter wird durch einen oder mehrere Controller gesteuert, wobei im zweiten Fall alle Controller einen Konsens für den nächsten Zug des kontrollierten Roboters finden. Damit kann die Anwendung verschiedene Fehler tolerieren:

- Ausfall eines Controllers
- Ausfall eines Roboters (bzw. Kommunikationsfehler zwischen Roboter und Controller)
- Berechnungsfehler eines Controllers

Die Anwendung wurde auf Basis von CORE[18] entwickelt, einem System, das es gestattet die Zuverlässigkeit einer Anwendung unter Einhaltung von Zeitgarantien zu erhöhen. Die derzeitige Implementation läuft auf Mach (NeXT-STEP 3.3) und nutzt zur Kommunikation zwischen den Komponenten der Applikation die Interprozeß Kommunikation von Mach.

Mit den „Unstoppable Robots“ haben wir eine abgeschlossene, auf proprietärer Hardware und Software laufende Echtzeitanwendung. Um Einsatzmöglichkeiten der Applikation zu erhöhen, haben wir Schnittstellen geschaffen, die das System öffnen und den Zugriff externer Programme auf Teile der Anwendung gestatten. Wir haben Java-Objekte implementiert, die diese Schnittstellen nutzen. Um sicherzustellen, daß bei der Anbindung der Java-Objekten an die „Unstoppable Robots“ die responsiven Eigenschaften der Anwendung erhalten

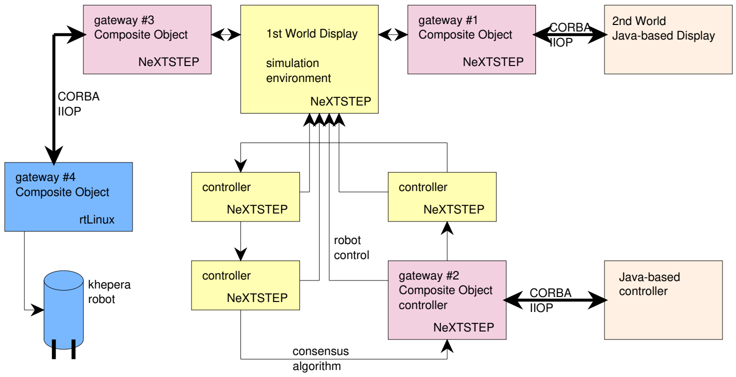

Abbildung1: Kommunikationstruktur der erweiterten Unstoppable Robots

bleiben, sind die Schnittstellen durch zwei Firewalls auf Basis des „Composite Object“-Ansatzes implementiert.

Ein „Composite Object“, Gateway #1 in Abbildung 1, befindet sich außerhalb der zeitkritischen Teile der Anwendung und kapselt die Funktionalität der Anzeige. Dazu wird das Polling-Verfahren verwendet, das es problemlos gestattet, weitere Displays an die „Unstoppable Robots“ anzubinden. Um Überlastsituationnen durch Clientanfragen zu vermeiden, werden die Ergebnisse älterer Anfragen eine gewisse Zeit gecached und nur ein Teil der Anfragen direkt an die Mach-Implementation weitergeleitet. Damit wird die Frequenz der an das Echtzeitsystem weitergereichten Anfragen beschränkt.

Das zweite „Composite Object“, Gateway #2 in Abbildung 1, agiert als Controller und ist, da es sich damit in der Controller-Kontrollschleife befindet, Teil der zeitkritischen Bereiche der Anwendung. Wir haben die Controllerfunktionalität in ein Java-Objekt ausgelagert, das, in dem es das „Composite Object“ nutzt, als Controller eines Roboters fungieren kann. Ist das Java-Objekt nicht in der Lage, die Zeitschranken der Controller-Kontrollschleife einzuhalten, übernimmt der Fallbackalgorithmus des „Composite Objects“ die Steuerung des Roboters. Dieser Algorithmus ist sehr einfach gehalten: Er sorgt dafür, daß der Roboter an der Stelle, an der er sich befindet, stehen bleibt. Solange die Anzahl der zeitlichen Ausfälle beschränkt ist, wird durch diesen Fallbackalgorithmus das Verhalten der Anwendung nicht gestört.

Die Abbildung 1 zeigt die Kommunikationsstruktur der erweiterten Applikation. Der „Composite Object“-Ansatz erlaubt auch das Ankoppeln gänzlich anderer Komponenten, wie beispielsweise der echten Khepera-Roboter [19].

5 Aspekte der Implementation und Messungen an der Applikation

5.1 Implementation

Die Beispielapplikation besteht aus Java-Komponenten und denen, die die Kommunikation mit dem Echtzeitsystem unter Verwendung der vorgestellten Techniken realisieren. Im folgenden soll auf die verschiedenen Teile kurz eingegangen werden.

Zur Implementation der „Composite Objects“ auf NeXTSTEP wurde ILU-CORBA[20] verwendet. Die Verbindung von Java und CORBA geschieht unter Benutzung des *Java Language Binding* der OmniBroker-CORBA-Implementation[21].

Passiver Wrapper. Gateway #1 agiert als passiver Wrapper. Es hat die Aufgabe, die Ankopplung eines Displays in der angesprochenen Weise zu ermöglichen. Es besteht aus zwei getrennten Prozessen, die lose über eine *Pipe* gekoppelt sind, über die Kommunikation in einer Richtung möglich ist. In der anderen Richtung werden UNIX-Signale benutzt.

Auf der Seite des Echtzeitsystems handelt es sich um einen Prozeß, der via MACH-IPC mit dem Echtzeitsystem kommunizieren kann. Er ist in der Lage, die für eine Anfrage relevanten Daten abzufragen und in eine Pipe zu schreiben. Ausgelöst wird die Abfrage durch ein UNIX-Signal.

Bei der Bearbeitung von Anfragen kommt die *Caching*strategie zum Tragen, die „alte“ Werte benutzt, solange sie weniger als 250 Millisekunden alt sind. Ansonsten wird ein Signal an den anderen Prozeß des „Composite Objects“ geschickt, dessen Werte von der Pipe gelesen und schließlich zurückgegeben.

Auf diese Weise ist sichergestellt, daß die Echtzeitanwendung nicht mit zu vielen Anfragen „überlastet“ werden kann. Jedoch ist eine Überlastung des ganzen Systems durch sehr viele CORBA-Anfragen immer noch möglich. Lösung für diesen Fall ist die Benutzung einer Ressourcenreservierungstechnik, wie dem Scheduling Server, zum Einschränken der Rechenleistung, die der CORBA-Seite des „Composite Objects“ zur Verfügung steht.

Aktiver Wrapper. Die Anforderungen für Gateway #2 sind strenger. Auch hier ist eine in zwei Prozesse zerlegte Implementation entstanden. Die Kopplung erfolgt über zwei Pipes, die einen zweiseitigen Datenaustausch ermöglichen. Zur Signalisierung werden wieder UNIX-Signale verwendet. Allerdings implementiert der aktive Wrapper zusätzlich den Fallback-Algorithmus.

Die Echtzeitseite des „Composite Object“ ist Bestandteil der inneren Kontrollschleife der Applikation und hat für deren Komponenten das „Aussehen“ eines Controllers. Der wesentliche Unterschied zu einem Controller besteht darin, daß keine eigene Berechnung stattfindet. Statt dessen werden die Ausgangsdaten an die CORBA-Seite des „Composite Objects“ übergeben, ein entfernter Methodenaufruf ausgeführt und dann gewartet. Kommen die Steuerdaten rechtzeitig, werden sie benutzt, ansonsten kommt der Fallback-Algorithmus zum Einsatz.

Abbildung 2 zeigt einen Screenshot der laufenden Applikation. Am unteren Rand sind die Statusmeldungen des aktiven Wrappers zu sehen.

Java-Display. Das externe Display, das ein Abbild der „Welt“ verkörpert, ist ein CORBA-Client, der seine Anfragen an die CORBA-Seite des passiven Wrappers richtet. Die Frequenz, mit der das geschieht, kann in Abhängigkeit von der gewünschten Aktualisierungsrate gewählt werden. Sinnvolle Obergrenze ist die durch das Caching im „Composite Object“ vorgegebene 4-Hz-Rate.

Java-Controller. Der Java-Controller läuft als CORBA-Server und ermöglicht dem Benutzer die Wahl zwischen manuellen und automatischen Betrieb. Erstgenannter Modus erlaubt die direkte Steuerung des Roboters über Steuerbuttons, letztgenannter implementiert den Algorithmus, den auch die Controller der Original-Applikation benutzen.

Abbildung 3 zeigt einen Screenshot der beiden Java-Komponenten. Auch hier sind im Hintergrund die Statusmeldungen des Java-Controllers und des aktiven Wrappers zu sehen.

5.2 Messungen

Interessant für Messungen ist vor allem der Teil der Applikation, der in die zeitkritischen Bereiche eingreift. Außerhalb von diesen gilt ein „best-effort“-Ansatz, dessen zeitliches Verhalten weniger von den Java-Teilen, als viel mehr von den Kommunikationsstrukturen bestimmt wird. Demzufolge haben die hier vorgestellten Messungen auch den Java-Controller und die damit zusammenhängenden Komponenten zum Ziel.

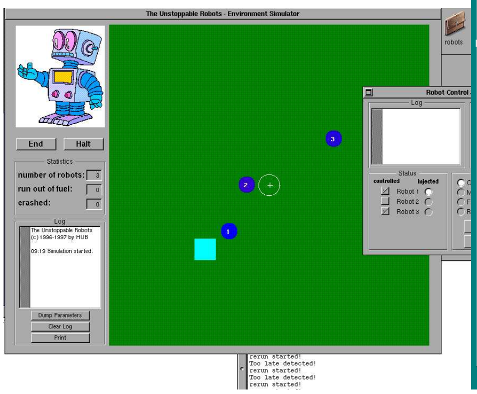

Abbildung2: Die Demoapplikation auf NeXTSTEP

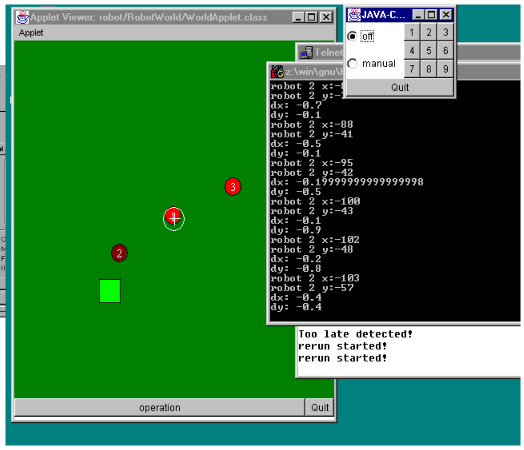

Abbildung3: Die Java-Komponenten auf Windows NT

Vorüberlegungen. Zwei Dinge müssen gelten, damit die Applikation in der beschriebenen Weise unter Benutzung des Java-Controllers funktioniert:

- Die Zeitschranke für die Berechnung auf der Echtzeitseite des „Composite Objects" muß eingehalten werden.
- Der Fallback-Algorithmus darf nicht so oft zur Anwendung kommen, daß die sekundäre Aufgabe der Roboter (am Leben bleiben) nicht erfüllt werden kann und der Roboter „verhungert". Da auch im Stillstand Energie verbraucht wird, führt Stillstand auf lange Sicht zum Ausfall des Roboters.

Zur Beantwortung der ersten Frage verfügt der aktive Wrapper über eine Instrumentierung, die Rundenzeiten auf beiden Seiten mißt. Die Vereinigung beider Meßreihen in einem Diagramm zeigt, ob es nichttolerierbare Schwankungen der „inneren" Kurve gibt, und ob diese von Schwankungen der „äußeren" abhängen.

Gleichzeitig ist jede Stelle, an der die äußere Kurve über der inneren liegt, ein Zeitpunkt, zu dem der Fallback-Algorithmus zur Anwendung kam.

Die zweite Forderung muß durch Langzeitbeobachtungen geklärt werden. Da die Roboter durch den Zwang, immer wieder zur Energiestation zurückkehren zu müssen, einem gewissen Zyklus unterliegen, genügt es, mehrere dieser Zyklen zu beobachten und das Ergebnis zu verallgemeinern.

Da im Normalfall solche Aussetzer (Benutzung des Fallback-Algorithmus) nur sehr selten auftreten, muß für diese Untersuchungen künstlich ein solcher Zustand hergestellt werden. Verursacht wird er im wesentlichen durch zwei Faktoren:

- Last im Netzwerk, die zu verringerter Bandbreite führt.
- Last auf dem Rechner, auf dem der Java-Controller läuft, die diesem soviel Rechenleistung entzieht, daß es nicht möglich ist, innerhalb der gegebenen Zeit das Ergebnis zu berechnen.

Der erstgenannten Problematik wurde bei diesen Versuchen nicht weiter nachgegangen, da die betrachtete Applikation nur sehr geringe Ansprüche an die verfügbare Bandbreite hat. Für die Untersuchung des zweiten Falls wurde auf dem Versuchsrechner künstlich eine sehr hohe Last generiert und deren Auswirkung auf die Meßwerte und das Verhalten der Roboter untersucht.

Im Anschluß an diesen Versuch wird ermittelt, inwieweit man unter Benutzung höherer Prioritäten, die man der virtuellen Java-Maschine gibt, trotz Vorhandensein von Last zu besseren Ergebnissen kommen kann. Im Versuch wird dies durch das Benutzen der Prioritätsstufe „hoch" auf dem verwendeten Windows-NT-System erreicht.

Umgebung. Folgende Hard- und Software wurde für die Versuche benutzt:

- Existierende Applikation und „Composite Objects":
  HP 715/50, 32 MB RAM, NeXTSTEP 3.3

– Java-Komponenten:
  PC Pentium 90, 40 MB RAM, Windows NT 4.0, JDK 1.1

Verbunden sind beide Testsysteme über ein 10 MBit Ethernet, zwischen ihnen befindet sich zwei Router, da sie zu verschiedenen Netzsegmenten gehören. Zur Erzeugung von Last auf dem Windows-NT-System steht ein Lastgenerator zur Verfügung, der 20 Prozesse erzeugt, die intensive Integer-Berechnungen durchführen.

Ergebnisse. Die Diagramme zeigen zwei Kurven, die die Abarbeitungszeiten der inneren Kontrollschleife (zeitkritischer Bereich; etwa bei 100 ms) und die Latenzzeit der CORBA-Kommunikation (CORBA-Bereich) darstellen. Abbildung 4 zeigt das zeitliche Verhalten für den Fall eines Java-Controllers auf einer unbelasteten Maschine.

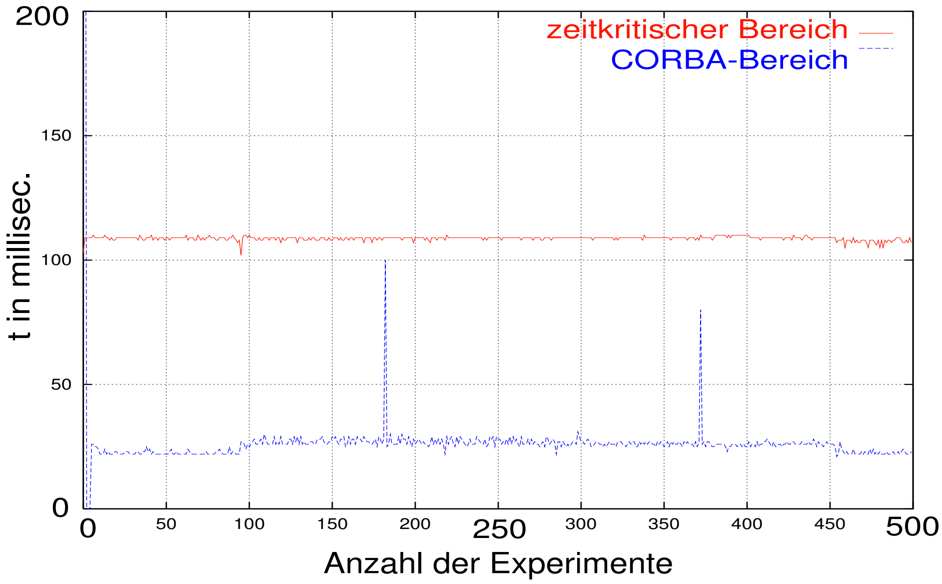

Abbildung4: keine Hintergrundlast

Abbildung 5 zeigt das gleiche Szenario, nur unter Benutzung des Lastgenerators, und Abbildung 6 zeigt die Auswirkungen einer höheren Prioritätsstufe bei gleicher Last.

Zu sehen ist, daß die Rundenzeiten der inneren Kontrollschleife vollkommen unabhängig von den externen Zeiten sind, die geforderten Bedingungen also eingehalten werden. Im Normalfall kommt es kaum zur Benutzung des Fallback-Algorithmus, eine hohe Last, die hohe Ausführungszeiten der Java-Algorithmen zur Folge hat, kann dies jedoch provozieren. Diesem Effekt kann man mit geeigneter Verwendung von Möglichkeiten des Betriebssystems entgegenwirken.

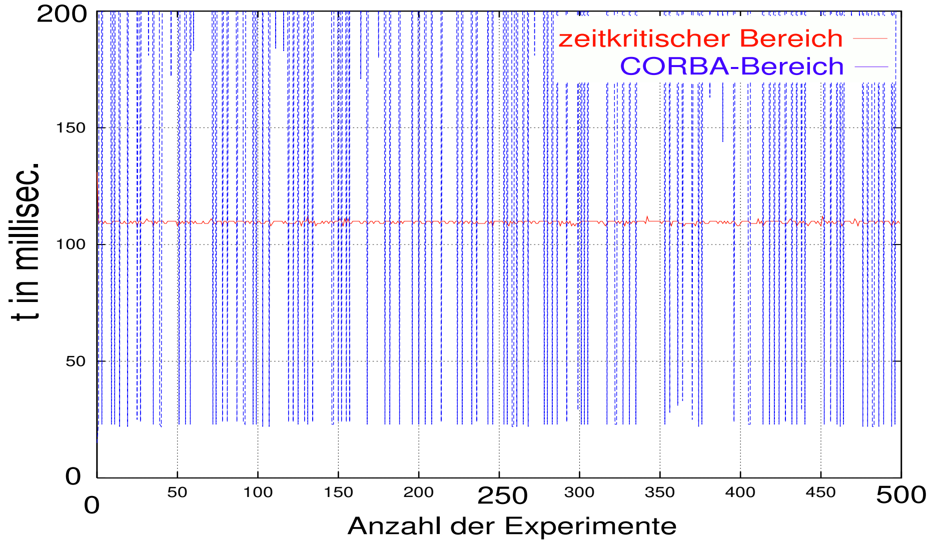

Abbildung5: Mit Hintergrundlast

Negative Auswirkungen auf die Bewegungen des Roboters gibt es in keinem der untersuchten Fälle. Selbst bei einer sehr hohen Rate der Anwendung des Fallback-Algorithmus erhält die Applikation ausreichend viele reguläre Steuerbefehle vom Java-Objekt, um dem Roboter eine Bewegung entsprechend seinen Aufgaben zu ermöglichen.

6 Schlußfolgerungen

In dieser Arbeit haben wir den Einsatz der Programmiersprache Java für Komponenten eines verteilten Echtzeitsystems untersucht. Unser Augenmerk richtet sich dabei auf Mechanismen zur Verknüpfung von Java mit existierenden Echtzeitsystemen. Wir gehen davon aus, daß für die in Java geschriebenen Komponenten nur schwache Echtzeitgarantien gegeben werden können und Techniken zur Kompensation verpaßter *deadlines* und verspätet eingehender Resultate benutzt werden müssen.

Der von uns entwickelte „Composite Objects"-Ansatz gewährleistet Interoperabilität zwischen Java-Komponenten und dem Echtzeitsystem bei gleichzeitiger zeitlicher Entkopplung beider Seiten. Verspätete Ausführung der Java-Komponenten beeinflußt damit das Verhalten des Echtzeitsystems kaum.

Die „Unstoppable Robots" stellen eine Fallstudie für ein verteiltes, fehlertolerantes Echtzeitsystem auf Basis des Betriebssystems Mach dar. Wir haben die Demo-Applikation um zwei Java-Komponenten, eine graphische Benutzerschnittstelle und einen Controller, erweitert. Die hier präsentierten Messungen

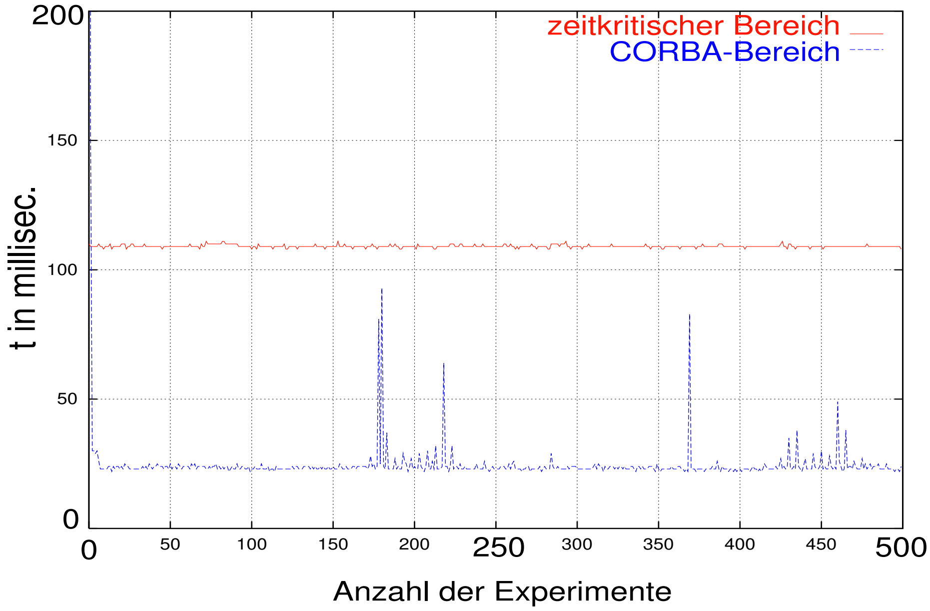

Abbildung6: Mit Hintergrundlast und höherer Priorität

belegen ein stabiles Verhalten des erweiterten Echtzeitsystems unter variierenden Lastsituationen.

Der Entwurf eines Echtzeitsystems stellt eine nicht-triviale ingenieurtechnische Leistung dar. Solche Systeme beruhen oftmals auf Heuristiken, ihre Erstellung ist langwierig und teuer und ihre Wartung schwierig. Ebenso wie in den hier präsentierten Fallstudien läßt sich unsere „Composite Objects"-Technik zur Erweiterung solcher legacy-Systeme um offene Schnittstellen einsetzen. Die Programmiersprache Java bietet sich aufgrund ihrer Charakteristika für solche Erweiterungen an.

Literatur

[1] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[2] M. Werner and M. Malek. The Unstoppables - Responsiveness by Consensus. *HUB Informatik-Berichte*, (90):19, Dec 1996.

[3] A. Polze and L. Sha. Composite Objects: Real-Time Programming with CORBA. In *to appear in Proceedings of 24th Euromicro Conference, Network Computing Workshop*, 1998.

[4] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Inc., Framingham, MA, USA, 1995.

[5] R. M. Soley, editor. *Object Management Architecture Guide*. John Wiley & Sons, New York, third edition, 1995.

[6] A. Miyoshi and T. Kitayama. Implementation and Evaluation of Real-Time Java Threads. In *Proceedings of 18th Real-Time Systems Symposium*, 1997.

[7] Y. Ishikawa, H. Tokuda, and C. Mercer. An Object-Oriented Real-Time Programming Language. *IEEE Computer*, 25(10), 1992.

[8] N. Gehani and K. Ramamritham. Real-Time Concurrent C: A Language for Programming Dynamic Real-Time Systems. *Real-Time Systems Journal*, 3(4), 1991.

[9] K. B. Kenny and K.-J. Lin. Building Flexible Real-Time Systems Using the Flex Language. *IEEE Computer*, 24(5), 1991.

[10] K. Nilsen. Java for Real-Time. *Real-Time Systems Journal*, 11(2), 1996.

[11] J. McGoogan, editor. *Realtime CORBA - A White Paper - Issue 1.0.* OMG Realtime Platform SIG, 1996.

[12] D. C. Schmidt, A. Gokhale, T. H. Harrison, D. Levine, and C. Cleeland. *TAO: a High-performance Endsystem Architecture for Real-Time CORBA.* RFI response to the OMG Special Interest Group on Real-Time CORBA, 1997.

[13] D. C. Schmidt, A. Gokhale, T. H. Harrison, and G. Parulkar. A High-performance Endsystem Architecture for Real-Time CORBA. *IEEE Communications Magazine*, 14(2), 1997.

[14] B. Thuraisingham, P. Krupp, and V. Wolfe. Position Paper: On Real-Time Extensions to Object Request Brokers. In *Proceedings of Second Workshop on Object-Oriented Real-Time Dependable Systems (WORDS).* IEEE Comp. Soc. Press, Februar 1996.

[15] V. F. Wolfe, J. K. Black, B. Thuraisingham, and P. Krupp. Real-Time Method Invocations in Distributed Environments. In *Proceedings of the International High Performance Computing Conference*, 1995.

[16] A. Polze, G. Fohler, and M. Werner. Predictable Network Computing. In *Proceedings of 17th International Conference on Distributed Computing Systems (ICDCS'97)*, pages 423–431, Baltimore, USA, 1997. IEEE Computer Society Press.

[17] J. Richling and A. Polze. Scheduling Server for Predictable Computing: an Experimental Evaluation. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, held in conjunction with Real-Time Systems Symposium*, pages 130–137, San Francisco, USA, Dec 1997.

[18] M. Malek, A. Polze, and M. Werner. A Framework for Responsive Parallel Computing in Network-based Systems. In *Proceedings of International Workshop on Advanced Parallel Processing Technologies*, Bejing, China, September 1995.

[19] K-Team SA. *Khepera User Manual, Version 4.06.* K-Team SA, Ch.du Vuasset, CP 111, 1028 Preverenges, Lausanne, Switzerland, 1995.

[20] Xerox Parc. Ilu. *ftp://ftp.parc.xerox.com/pub/ilu/ilu.html*, 1998.

[21] M. Laukien and U. Seimet. Omnibroker. *http://www.ooc.de/*, 1998.