DIPLOMARBEIT

IEEE 1394-Kommunikation über das Socket-Interface in Java

ausgeführt am Institut für Computertechnik
der Technischen Universität Wien

unter der Anleitung von o.Univ. Prof. Dipl.-Ing. Dr. Dietmar Dietrich,
Dipl.-Ing. Gerhard Pratl
als verantwortlichen mitwirkenden Universitätsassistenten
durch

Klaus Salamonsberger
Johnstraße 56/15, 1150 Wien
Matr.Nr. 9425090

Wien, 13. September 2003

Unterschrift

Kurzfassung

Das Bussystem IEEE 1394 ist sehr gut für die digitale Verbindung von Multimedia-Geräte geeignet. Vor allem die unterschiedlichen Übertragungsarten, die zur Verfügung gestellt werden, sind dabei sehr hilfreich. Dabei wird zwischen der isochronen Datenübertragung, die Zustellung von Paketen in Echtzeit garantiert, und der asynchronen Datenübertragung, bei der der Sender eine Bestätigung erhält, sobald ein gesendetes Paket angekommen ist, unterschieden. Ein Socket stellt die Möglichkeit zur Verfügung, zwischen Prozessen auf unterschiedlichen Systemen zu kommunizieren. Da Java eine moderne objektorientierte und weit verbreitete Programmiersprache ist, liegt es nahe, in dieser Sprache mittels Sockets den Datenaustausch über ein IEEE 1394-Bussystem zu ermöglichen.

In dieser Diplomarbeit wurde eine Abbildung der Funktionalität des IEEE 1394-Bussystems auf einen Socket entwickelt und teilweise implementiert. Dabei wurde auf eine IEEE 1394-API zurückgegriffen, die den Zugriff auf einen IEEE 1394-Adapter ermöglicht. Da in Java Hardware nicht direkt angesprochen werden kann, musste dazu in der Programmiersprache C eine Bibliothek implementiert werden, die diese Aufgabe übernimmt. Diese kann in Java mittels JNI (Java Native Interface) verwendet werden. Die gesamte Implementierung der Socket-Funktionalität wurde somit in C durchgeführt. Damit ist nur mehr ein dünner Layer an Java-Klassen notwendig, um die volle Funktionalität auch in Java nutzen zu können. Außerdem kann die Implementierung in C unabhängig von Java als eigenständige Programmbibliothek genutzt werden.

Da die Funktionalität des IEEE 1394-Bussystems mit seinen unterschiedlichen Übertragungsarten sehr umfangreich ist und zudem die verwendete IEEE 1394-API zur Zeit der Fertigstellung dieser Arbeit noch nicht die gesamte benötigte Funktionalität zur Verfügung stellte, beschränkt sich diese Diplomarbeit auf die Implementierung des asynchronen Teils von IEEE 1394 und seiner Abbildung auf das Socket-Interface. Der isochrone Teil wird konzeptionell beschrieben und die vorgesehene Architektur erläutert.

Abstract

The bus system IEEE 1394 is well suited for the digital connection of multimedia devices. Above all the different available transmission modes are very helpful thereby. The isochronous data communication guarantees real time-communication for the sent packets, and the asynchronous data communication guarantees, that a packet will arrive by sending a confirmation as soon as the packet has arrived. A socket allows for communication between processes on different systems. Since Java is a modern object-oriented and wide-spread programming language, making socket-based communication available over the IEEE 1394-bussystem is reasonable for this language.

In this diploma-thesis a mapping of the functionality of the IEEE 1394-bussystems on a socket was developed and partly implemented. To have access to the IEEE 1394-adapter an IEEE 1394-API was used. Since Java programs cannot access hardware directly, an additional library had to be implemented in the programming language C, which takes over this task. This can be used in Java by means of the JNI (Java Native Interface). The entire implementation of the functionality of the socket was thus accomplished in C. So only a thin layer of Java classes is necessary in order to be able to use full functionality also in Java. In addition the implementation in C can be used by Java as independent program library.

Since the functionality of the IEEE 1394-bussystem with its different transmission modes is very extensive and the used IEEE 1394-API did not have the entire necessary functionality, this diploma-thesis is limited to the implementation of the asynchronous part of IEEE 1394 and its mapping to the socket-interface. The isochronous part is described conceptionally and the intended architecture is described.

Inhaltsverzeichnis

1 Einführung

Als erstes wird auf die Aufgabenstellung dieser Arbeit eingegangen. Danach werden die in dieser Arbeit benötigten Komponenten behandelt; in erster Linie handelt es sich dabei um den IEEE 1394-Bus, den Begriff Socket und die verwendeten Programmiersprachen, also Java und C. Außerdem wird auch auf das JNI (Java Native Interface) eingegangen, da dieses die Schlüsselstelle zwischen Java und C darstellt. Nachdem sozusagen die Grundlagen abgehandelt wurden, kann auf die Schnittstellendefinitionen und abschließend auf die Implementierung und deren Details eingegangen werden.

1.1 Aufgabenstellung

Ziel dieser Diplomarbeit ist es, die Verwendung des IEEE 1394-Busses mittels einer Socket-Schnittstelle in Java zu ermöglichen. Zuerst ist die Abbildung der IEEE 1394-API auf die Socket-Schnittstelle zu definieren und anschließend auch zu implementieren. Dabei ist zu bedenken, dass der IEEE 1394-Bus ein serieller Hochgeschwindigkeitsbus ist, der sowohl eine isochrone als auch eine asynchrone Übertragungsart kennt (*siehe Abschnitt 1.2*). Bei der isochronen Datenübertragung handelt es sich um eine Echtzeitübertragung. Sie ist z. B. für Audio- und Videodaten vorgesehen. Da es dabei nicht unbedingt auf Datensicherheit ankommt, ist diese Übertragungsart nicht gesichert[1]; die Echtzeit steht im Vordergrund. Diese wird vom IEEE 1394-Bus durch eine gegebene Bandbreite garantiert. Im Gegensatz dazu ist der asynchrone Datenaustausch gesichert, verfügt aber über keine garantierte Bandbreite. Dadurch kann nicht sichergestellt werden, dass asynchrone Daten in einer gewissen Zeit übertragen werden, nur dass sie ankommen. Da sich diese beiden Eigenschaften

[1] Datensicherheit bedeutet hier, dass der Sender weiß, ob der Empfänger die Daten korrekt gelesen hat. Es handelt sich nicht um Verschlüsselung oder andere kryptographische Maßnahmen.

stark unterscheiden, sowohl in der Datensicherheit als auch in der Echtzeitfähigkeit, sind unterschiedliche Arten von Sockets zu implementieren.

1.2 Der IEEE 1394 Bus

Beim IEEE 1394 Bus handelt es sich um einen seriellen Hochgeschwindigkeitsbus, der vom Institute of Electrical and Electronics Engineers (IEEE) unter der Bezeichnung „IEEE Standard for a High Performance Serial Bus“ (IEEE Std 1394-1995) erstmals im Jahr 1995 standardisiert wurde [1394St1]. Häufig ist dafür auch der Begriff `FireWire` oder `iLink` anzutreffen, dabei handelt es sich jedoch um firmeneigene Produktbezeichnungen; daher wird in diesem Dokument nur die Bezeichnung IEEE 1394 verwendet. Der genannte Standard ermöglicht einen Datentransfer von maximal 400 Mbit/s und eine maximale Anzahl von 63 Knoten pro Bus; Bridges zwischen einzelnen Bussen sind nicht definiert. Im Jahr 2000 wurde dieser Standard zum IEEE Std 1394a-2000 erweitert [1394St2]. Er kann als Anhang zu jenem vom Jahr 1995 gesehen werden; er nimmt einige Verbesserungen, Korrekturen und Erweiterungen vor. Dabei handelt es sich im speziellen um kurze Busresets, Verbesserung bei der Buszuteilung, asynchrone Streams und das Powermanagement. Im Jahr 2003 wurde auch dieser Standard zum IEEE Std 1394b-2003 erweitert. Dabei wird vor allem die Bandbreite auf 3,2 Gbit/s erweitert. Dafür ist jedoch eine massive Veränderung des physikalischen Layers vom IEEE 1394-Adapter erforderlich; für diese Übertragungsgeschwindigkeit wird auf das Übertragungsmedium Glasfaserkabel umgestellt. Weiters ermöglicht dieser Standard eine Übertragungsgeschwindigkeit von 100 Mbit/s über ein ungeschirmtes twisted pair CAT-5 und RJ-45 Kabel über eine Distanz von bis zu 100m. Diese Erweiterung wurde speziell für eine kostengünstige Umrüstung alter Systeme auf eine IEEE 1394-Verkabelung durchgeführt. Nachteil all dieser Standards ist jedoch immer noch, dass maximal 63 Adapter miteinander verbunden werden können. Daher wurde der Standard IEEE Std 1394.1 entwickelt. Dieser sieht die Verwendung von Bridges vor. Damit ist es möglich, mehrere Bussysteme miteinander zu verbinden [DISS1 pp31-33]. Für diese Diplomarbeit ist der ursprüngliche

Standard IEEE Std 1394-1995 relevant. Daher wird dieser nun etwas detaillierter beschrieben.

Der IEEE 1394 Bus ist „plug-and-play-fähig“, das heißt selbst-konfigurierend. Dadurch ist es möglich, Geräte an den Bus an- und wieder abzustecken, ohne dass er dazu speziell konfiguriert werden muss. IEEE 1394 besitzt auch die Eigenschaft des „hot-plugging“, wodurch Geräte während des Betriebs an- oder abgesteckt werden können, zudem reagiert das System automatisch auf alle Veränderungen. Aufgrund dieser beiden Eigenschaften wird am IEEE 1394-Bus automatisch ein Initialisierungsschritt eingeleitet, genannt Busreset, sobald ein Gerät an den Bus an- oder abgesteckt wird. Der Busreset veranlasst alle Adapter am Bus sich sozusagen neu anzumelden. Dadurch erhalten alle Knoten automatisch eine neue Adresse. Im Vergleich mit einem LAN (Local Area Network) kann festgestellt werden, dass auch Geräte während des Betriebes an und abgesteckt werden können, jedoch werden dort nicht automatisch neue Adressen vergeben. Damit am LAN automatisch neue Adressen vergeben werden können muss ein DHCP-Server (Dynamic Host Configuration Protocol) vorhanden sein. Dieser verteilt die Adressen an die am LAN vorhandenen Geräte. Dabei handelt es sich jedoch weder um einen vorgeschriebenen Dienst noch um eine Systemkomponente.

1.2.1 Die Adressierung der IEEE 1394 Knoten

Beim IEEE 1394-Bus handelt es sich zwar, wie der Name schon sagt, logisch um ein Bussystem, hardwaremäßig jedoch um einen Baum. Da Geräte, auch Knoten genannt, jederzeit an oder abgesteckt werden können, kann sich die Topologie des Netzes vollständig verändern (*siehe Abbildung 1-1*). Ist nur ein Port, dabei handelt es sich um einen Steckkontakt, des Knotens verbunden, handelt es sich um ein „Blatt“ (z. B. Knoten eins oder zwei aus *Abbildung 1-1*). Sind mehrere Ports verbunden, handelt es sich um einen Zweig (z. B. Knoten drei aus *Abbildung 1-1*). Der erste Knoten in einer solchen Baumstruktur wird Wurzel (engl. root) genannt.

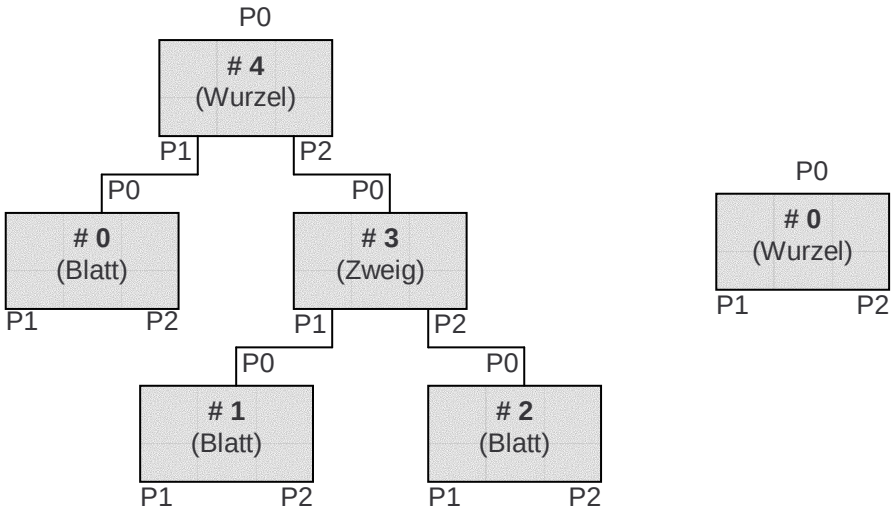

Abbildung 1-1 Physikalische Baumstruktur

Ein IEEE 1394-Netzwerk setzt sich aus bis zu 1023 IEEE 1394-Bussen zusammen. Jeder Bus kann wiederum bis zu 63 Knoten adressieren. Um einen Knoten eindeutig ansprechen zu können, ist sowohl eine Angabe über den Bus, `bus_ID` genannt, als auch über die Knotennummer (`physical_ID` genannt) notwendig. Zusammen ergeben sie die `node_ID`, die von der Bustopologie abhängig ist. Daher kann sie sich beim An- oder Abstecken eines Gerätes verändern; es wird auch von einer dynamischen Adresse gesprochen. Die Vergabe der `physical_ID` beginnt immer beim Wurzel-Knoten und sieht folgendermaßen aus:

1. Beginne mit dem Wurzel-Knoten und gib ihm die höchste Nummer (Anzahl der Knoten minus Eins)

2. Gib dem verbundenen, noch nicht bearbeiteten Knoten mit der höchsten Portnummer die nächst kleinere Nummer, und führe für diesen Knoten Punkt zwei aus.

3. Hat ein Knoten keine weiteren, nicht bearbeiteten Verbindungen, gehe wieder um einen Knoten zurück und fahre dort mit Punkt zwei fort.

Dieser Algorithmus lässt erkennen, dass sich bei einer Veränderung am Bus (durch Wegnehmen oder Hinzufügen eines Gerätes) die Adressen gravierend

verändern. Daher besitzt jeder IEEE 1394-Knoten eine eindeutige Kennung, die `GUID (Globally Unique Identifier)` [API1 pp21] oder auch `Node Unique Identifier` genannt. Sie ist 64 Bit lang und erlaubt es, jeden Adapter eindeutig zu identifizieren. Bei dieser Angabe handelt es sich jedoch nicht um eine Adresse im herkömmlichen Sinn, da mit ihr kein Adapter angesprochen werden kann. Ein IEEE 1394-Adapter kann nur über seine `node_ID` adressiert werden [API1 pp165].

Für die `bus_ID` wurde der spezielle Wert *3FFh* dafür reserviert, dass der aktuelle Bus angesprochen wird; der Wert *3Fh* für die `physical_ID` spricht alle Adapter des ausgewählten Busses an [API1 pp18] [1394St1 pp21]. Es handelt sich daher um eine so genannte Broadcast-Anforderung. Derzeit besteht noch keine Möglichkeit, mehrere Busse miteinander zu verbinden, da die erforderliche Hardware – so genannte Bridges – noch nicht existiert. Aus diesem Grund wird in dieser Arbeit nur der lokale Bus in Betracht gezogen.

Jeder Adapter besitzt weiters einen privaten Adressraum von 48 Bit. Daher ergibt sich folgende Adresse, um auf den Speicher eines Adapters zugreifen zu können (*siehe Abbildung 1-2*):

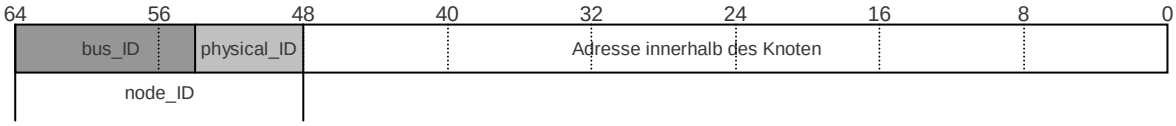

Abbildung 1-2 Aufbau der Adresse einer IEEE 1394-Speicherstelle

Die Adresse innerhalb eines Knotens teilt sich ebenfalls in mehrere Bereiche auf. Der größte Bereich ist der allgemeine Speicher. Er beginnt bei 0 und endet bei *FFF DFFF FFFFh*. Dieser Bereich ist für alle zugänglich. Anschließend folgt der private Speicherbereich. Darauf darf nur vom eigenen Adapter aus zugegriffen werden und endet bei *FFF EFFF FFFFh*. Der letzte Bereich wird als Config-Rom bezeichnet. Darin sind alle Register enthalten, die für die Konfiguration des IEEE 1394-Adapters benötigt werden. Darunter befindet sich z. B. die `GUID`. Dieser Bereich endet bei FFF FFFF FFFFh.

1.2.2 Der Busreset

Ein Busreset wird immer dann ausgelöst, wenn ein Gerät am Bussystem an- oder abgesteckt wird. Ab diesem Zeitpunkt können keine Benutzerdaten mehr über den Bus versendet werden. Wird ein Busreset ausgelöst, versenden alle Adapter so genannte `Self-ID-Pakete`. Sie enthalten unter anderem Daten über die Belegung der Ports oder welche Dienste und Datenrate die Knoten unterstützen. Aufgrund dieser Pakete kann ermittelt werden, wie das Bussystem hardwaremäßig aufgebaut ist. Damit kann auch die Vergabe der `node_ID` erfolgen. Die Information über alle vorhandenen Adapter und deren Adressen wird in der so genannten „topology map“ abgespeichert (*siehe Abschnitt 1.2.6*). Mit ihr kann jederzeit auf die Information über die Netzwerk-Topologie zugegriffen werden. Nachdem alle diese Pakete ausgewertet wurden, und die Adressen neu vergeben wurden, ist der Busreset abgeschlossen, und der Datenstrom kann wieder fließen.

Da sich durch einen Busreset die Adressen (`node_ID`) der IEEE 1394-Adapter ändern können, muss aus Sicherheitsgründen davon ausgegangen werden, dass sich alle Adressen nach einem Busreset verändert haben. Um sicherstellen zu können, dass auf einen bestimmten Adapter zugegriffen wird, muss dessen `GUID` überprüft werden. Tritt kein weiterer Busreset auf, bleiben die `node_IDs` unverändert. Ist die `GUID` einmal überprüft, kann somit davon ausgegangen werden, dass bei einem Zugriff auf diese `node_ID` der gewünschte Adapter angesprochen wird; ein Adapter wird immer über seine `GUID` eindeutig identifiziert. In der Zeit zwischen zwei Busresets wird der Knoten auch durch seine `node_ID` eindeutig identifiziert. Daher kann eine Identifizierung auch eindeutig über die beiden Informationen `node_ID` und Busresetcounter erfolgen. Verändert sich letzterer, ist die eindeutige Identifikation über die gegebene `node_ID` nicht mehr gewährleistet.

1.2.3 Datenübertragung

Die Datenübertragung erfolgt am IEEE 1394-Bus in Zyklen, die in Zeitscheiben unterteilt werden. Ein Zyklus dauert im Normalfall 125 µs, kann aber im Fall

einer laufenden Übertragung verlängert werden (*siehe Abbildung 1-3*) [1394St1 pp32-33]. Es gibt zwei unterschiedliche Möglichkeiten, solche Zeitscheiben zu erhalten: Belegt eine Übertragung fix eine gewisse Zeitscheibe, spricht man von `isochroner` Übertragung (*siehe Abschnitt 1.2.4*). In *Abbildung 1-3* wurde dies bei den Kanälen 1, 2 und 3 dargestellt. Für gelegentliches Versenden von Daten werden die verbleibenden Zeitscheiben verwendet; es wird von einer `asynchronen` Übertragung (*siehe Abschnitt 1.2.5*) gesprochen. In *Abbildung 1-3* ist dies durch die Pakete A, B und C dargestellt. Es wird garantiert, dass der isochrone Bereich maximal 80% vom gesamten Zyklus in Anspruch nimmt [1394St1 pp216]. Daher wird sichergestellt, dass auch asynchrone Datenpakete versendet werden können. Damit jedoch isochrone Datenpakete versendet werden können müssen am Bus mehrere Busmanager zur Verfügung stehen (*siehe Abschnitt 1.2.6*).

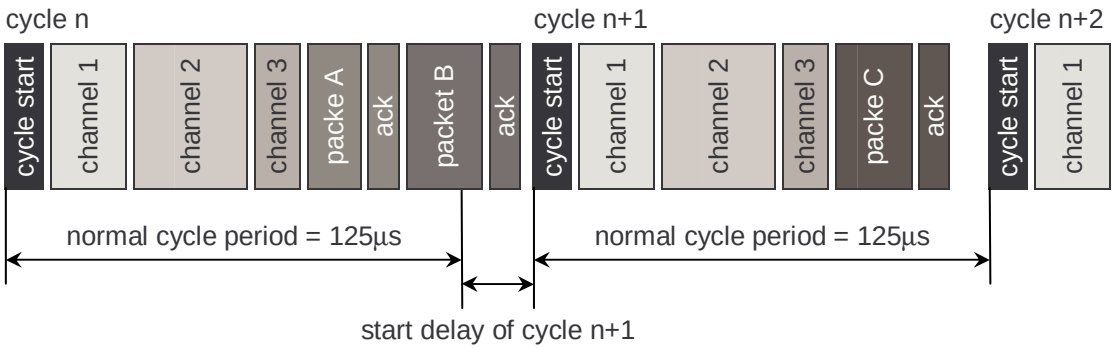

Abbildung 1-3 Zyklusstruktur

In *Abbildung 1-3* wurde ein Beispiel von mehreren Zyklen dargestellt. Es ist zu erkennen, dass die isochronen Übertragungen immer in denselben Abschnitten (channel 1, channel 2 und channel 3) erfolgen. Asynchrone Pakete werden an diese angeschlossen. Zwischen den einzelnen Paketen ist immer eine Pause (Gap), um die Pakete unterscheiden zu können. Im Zyklus n werden zu dem Zeitpunkt an dem die normale Periode zu ende wäre noch Daten übertragen. Daher wird diese Periode verlängert. Um genau diese Zeit muss der nächste Zyklus später beginnen. Sind hingegen zu wenig Daten vorhanden (Zyklus n+1), wird mit dem Beginn des Zyklus n+2 so lange gewartet, bis die Zykluszeit von 125 µs abgelaufen ist [1394St1 pp32-33].

1.2.4 Isochrone Datenübertragung

Das Wort isochronous kommt aus dem Griechischen, bestehend aus iso (=gleich) und chronous (=Zeit). Unter dem isochronen Datenverkehr wird ein Datenaustausch mit garantierter Bandbreite verstanden. Es wird garantiert, dass eine gewisse Datenmenge innerhalb einer gewissen Zeit beim Empfänger eintrifft. Es handelt sich somit um eine Echtzeit-Übertragung. Damit gewährleistet werden kann, dass isochrone Pakete übertragen werden können, muss beim Erstellen einer solchen Verbindung die gewünschte Bandbreite angegeben werden. Steht nicht genügend freie Bandbreite zur Verfügung, wird der Verbindungswunsch abgelehnt. Eine solche Verbindung wird auch als Kanal oder Channel bezeichnet; insgesamt stehen davon 64 pro Bus zur Verfügung. Auf einem Kanal kann nur ein Sender senden, es ist aber mehreren Empfängern gestattet, Daten von diesem Kanal zu empfangen. Nachteil der Echtzeit ist, dass die Datensicherheit nicht gewährleistet werden kann: Werden z. B. die Daten vom Empfänger nicht schnell genug verarbeitet, können dadurch Pakete verloren gehen.

Auf Grund der Echtzeitfähigkeit ist der IEEE 1394-Bus sehr gut für Multimedia-Applikationen geeignet. So ist er in modernen digitalen Multimedia-Geräten wie z. B. Videorecordern, SAT-Empfängern, Hifi-Geräten und Fernsehern zu finden. Für das menschliche Auge stellt es im Fall eines Fernsehfilmes z. B. kein Problem dar, wenn hin und wieder ein Bild verloren geht, da durch die schnelle Abfolge der Bilder das Fehlen eines einzelnen nicht bemerkt wird. Wenn der Film jedoch ins Stocken geraten würde, wäre der Fehler sofort erkennbar und würde als störend empfunden werden.

Für die Verwaltung von isochronen Datenverbindungen über den IEEE 1394-Bus wurde von der IEC (International Electrotechnical Commission) ein Standard mit der Kennzeichnung IEC 61883-1 entwickelt. Dieser definiert, wie die auf dem Bus vorhandene Bandbreite unter den Adaptern aufgeteilt wird, und die Verwaltung dazu auszusehen hat [IEC1].

1.2.5 Asynchrone Datenübertragung

Ebenso wie das Wort `isochronous` kommt auch `asynchronous` aus dem Griechischen und besteht aus `asyn` (=jeder) und `chronous` (=Zeit). Bei dieser Übertragung wird die restliche zur Verfügung stehende Bandbreite genutzt, um die Daten über den Bus zu versenden. Der IEEE 1394-Bus beinhaltet einen Transaction-Layer, der das korrekte Übertragen asynchroner Datenpakete garantiert. Zu diesem Zweck werden nach dem Empfang eines Datenpaketes so genannte Acknowledge-Pakete versendet [1394St1 pp173-198]. Aufgrund dieser kann der Sender ermitteln, ob das Paket korrekt angekommen ist oder nicht. Beim asynchronen Datenverkehr wird somit garantiert, dass ein versendetes Paket beim Empfänger eintrifft; andernfalls wird dem Empfänger mitgeteilt, dass es nicht angekommen ist. Daher ist er sehr gut für den Austausch der Steuerkommandos zwischen Multimedia-Geräten geeignet. Der genaue Zeitpunkt des Eintreffens kann jedoch nicht garantiert werden.

Für den asynchronen Datenaustausch gibt es drei mögliche Arten:

Read

`Read` ermöglicht es, Daten von einem anderen Adapter aus einem gegebenen Speicherbereich auszulesen. Dazu muss die Knotenadresse, der gewünschte Speicherbereich und ein Datenpuffer, in den die Daten geschrieben werden können, angegeben werden. Beim AL1394 (*siehe Abschnitt 1.4*) muss auch der Busresetcounter übergeben werden, wodurch garantiert wird, dass die Funktion nicht auf einen falschen Adapter zugreift. Zwischen dem Aufruf der `read`-Funktion und dem tatsächlichen Ausführen der Aktion kann nämlich aufgrund des verwendeten Multiprozesssystems eine Zeitverzögerung auftreten. Tritt genau in dieser Zeit ein Busreset auf und würde der Busresetcounter nicht berücksichtigt werden, würde die Aktion möglicherweise auf einem falschen Adapter ausgeführt werden, da sich die `node_ID` inzwischen geändert haben könnte.

Die Länge eines asynchronen Datenblockes der auf einmal über den IEEE 1394-Bus übertragen werden darf, hängt von der Übertragungsgeschwindigkeit

des Adapters ab. Bei einem System mit z. B. 400 Mbit/s ist die Blocklänge auf 2048 Bytes beschränkt. Halbiert sich die Übertragungsgeschwindigkeit des Adapters, so halbiert sich auch die maximale Blocklänge [1394St1 pp151].

Write

Mit diesem Befehl werden Daten im gegebenen Adapter in den gewünschten Speicherbereich geschrieben. Der Funktion muss dazu die Knotenadresse, der gewünschte Speicherbereich und ein Datenpuffer, aus dem die zu schreibenden Daten zu entnehmen sind, übergeben werden. Auch bei dieser Funktion gilt selbiges für den Busresetcounter und die Blocklänge wie für die Funktion `read`.

Lock

Die `Lock`-Operation stellt eine Spezialität des IEEE 1394-Bussystems dar. Sie bildet die Grundlage für Transaktionen, die über das Bussystem durchführt werden können. Bei dieser Funktion handelt es sich um ein atomares Lesen, Vergleichen und Schreiben der angegebenen Daten. Sie verändert den angegebenen Adressbereich am angegebenen Adapter nur dann, wenn der Sender weiß, was in diesem Adressbereich steht. Während diese komplexe Operation ausgeführt wird (es muss der Adressbereich ausgelesen, mit den vermuteten Daten verglichen und bei Übereinstimmung mit den neuen Daten beschrieben werden), kann keine andere Operation auf diesem Adressbereich erfolgen. Der Lock-Operation ist wie bei `read` und `write` die Knotenadresse und der Adressbereich zu übergeben. Weiters erwartet diese Funktion einen Datenpuffer mit den zu schreibenden Daten, mit den vermuteten Daten und einem weiteren Datenpuffer, in den die aktuellen Daten von dem gegebenen Adressbereich des Adapters retourniert werden. Durch die Angabe der vermuteten Daten kann sichergestellt werden, dass der Sender keine Transaktion übersehen hat. Eine `lock` kann im Gegensatz zu `read` und `write` jedoch nur mit einer Breite 32 Bit oder 64 Bit erfolgen.

1.2.6 Serielle Busmanager

Auf einem IEEE 1394-Bus können immer asynchrone Datenpakete ausgetauscht werden. Damit auch ein isochroner Datenverkehr möglich ist, müssen folgende Dienste zur Verfügung gestellt werden:

- Isochroner Ressource Manager (IRM),
- Bus-Manager (BM) und
- Cycle Master (CM).

Der IRM übernimmt die Verwaltung der isochronen Ressourcen. Dabei handelt es sich um die Zuteilung der isochronen Bandbreite, Zuteilung der Kanalnummern und Auswahl des Cycle Masters. Jeder Knoten, der die Funktionalität des IRM übernehmen könnte, gibt dies bei der Initialisierung des Busses in seinem Self-ID-Packet bekannt. Der IRM-fähige Knoten mit der höchsten node_ID wird dann tatsächlichen zum IRM. Ist keiner vorhanden, können auf diesem IEEE 1394 Bus nur asynchrone Datenpakete übertragen werden, da keine Verwaltung der isochronen Ressourcen möglich ist [1394St1 pp47].

Der BM stellt Managementfunktionen für alle Knoten zur Verfügung. Dabei handelt es sich z. B. um die Erzeugung der „topology map“ sowie der „speed map“, dem Energiemanagement und der Optimierung des Datenverkehrs [1394St1 pp47]. In der topology map sind alle am Bus vorhandenen Knoten und deren Eigenschaften gespeichert. Dadurch kann schnell ermittelt werden, welche Adapter verfügbar sind, welche Funktion sie haben und in welchem Zustand sie sich befinden. Die speed map beinhaltet ebenfalls alle am Bus vorhandenen Adapter, die dazu abgespeicherte Information betrifft jedoch die möglichen Übertragungsgeschwindigkeiten dieser Adapter.

Der Cycle Master (CM) sorgt dafür, dass das Paket „cycle start“ (*siehe Abbildung 1-3*) ausgesendet wird. Diese Nachricht zeigt an, dass ein neuer Zyklus begonnen hat. Am IEEE 1394 Bus übernimmt immer der Wurzel-Knoten die Aufgabe des CM. Um isochrone Daten übertragen zu können, ist es erforderlich, dass ein CM existiert, da ansonsten keine definierte Bandbreite gewährleistet werden kann [1394St1 pp32].

Am IEEE 1394 Bus ist höchstens ein `CM`, ein `IRM` und ein `BM` vorhanden, wobei ein IEEE 1394-Adapter auch mehrere Aufgaben auf einmal erfüllen kann. Damit isochrone Daten übertragen werden können, müssen alle drei Dienste zur Verfügung gestellt werden; asynchroner Datenverkehr ist auch mit keinem dieser Dienste möglich.

1.3 Der Socket

In der Literatur können für einen Socket unterschiedliche Definitionen gefunden werden. So handelt es sich laut [UNIX4 pp565] um eine Programmierschnittstelle zur Entwicklung netzwerkfähiger Applikationen, in [UNIX2 pp202] wird er als Programmierschnittstelle für Netzwerkanwendungen bezeichnet, und in [UNIX5 pp383] wird folgendes geschrieben:

> To provide common methods for interprocess communication and to allow use of sophisticated network protocols, the BSD system provides a mechanism known as sockets.

Es handelt sich somit um eine Prozesskommunikation, die auch den Zugriff auf Netzwerkprotokolle ermöglicht. Ursprünglich wurde der Socket als Netzwerkkommunikation für das Betriebssystem BSD-Unix von der University of California at Berkeley entwickelt und fand erstmals bei der Version Berkeley Unix 4.2bsd Verwendung [CNW1 pp351] [UNIX2 pp202]. In [UNIX2 pp203] wird ein Socket noch etwas genauer als die Verbindung einer Knotenadresse im Netzwerk mit der Schnittstellenadresse des Prozesses innerhalb eines Knotens definiert. In Java wird allgemein unter einem Socket jedoch etwas anderes verstanden, da es sich in diesem Fall um eine Implementierung für eine Übertragung mittels IP-basierter Protokolle handelt. Dies kann vor allem an der Adressierung mittels der IP-Adresse erkannt werden [JAVA2 pp443-521]. Auf dieses Problem wird vor allem in *Abschnitt 3.2.1* eingegangen.

Die Definition der Klasse `Socket` in Java weicht somit etwas vom ursprünglichen Gedanken vom Berkeley Unix ab; die Klasse `Socket` verwendet nur Verbindung über das Protokoll `TCP (Transmission Control Protocol)`

[RFC793] [JAVA2 pp451-469]. In Java gibt es weiters die Klasse `DatagramSocket`; sie wickelt den Datenaustausch mittels des Protokolls `UDP` (`User Datagram Protocol`) [RFC768] ab [JAVA2 pp509-521]. Beide basieren auf dem Internet Protocol (`IP`) [RFC791]. Ein Unterschied zwischen den beiden Protokollen und somit auch zwischen den Klassen `Socket` und `DatagramSocket` ist, dass `TCP` ein gesichertes Protokoll ist. `UDP` dagegen ist nicht gesichert, dafür einfacher aufgrund des geringeren Overheads [TIJ1 pp827] [JAVA2 pp508]. Eine weitere Unterscheidung findet sich darin, dass ein Socket im Gegensatz zum Datagram-Socket verbindungsorientiert ist. Somit muss bei einem Socket eine Verbindung aufgebaut werden, um Daten übertragen zu können. Datagram-Pakete können hingegen immer an eine beliebige Adresse gesendet werden, ohne zuvor eine Verbindung aufzubauen. Im „Ur-Socket“ von Berkeley sind beide Arten des Datenaustausches enthalten. Daher ist sowohl die Klasse `Socket` als auch die Klasse `DatagramSocket` als Socket im herkömmlichen Sinne zu verstehen.

Im `OSI-Referenzmodell` lässt sich eine sehr gute und anschauliche Darstellung für die Beschreibung eines Sockets finden [WSNP1 pp11-29]. Daher wird im folgenden Abschnitt kurz auf das OSI-Referenzmodell und die Beziehung des Sockets dazu eingegangen.

1.3.1 Sockets im OSI-Referenzmodell

Das `OSI-Referenzmodell` wurde 1983 von der ISO (International Organization for Standardization) entwickelt. Die Grundidee davon ist eine funktionelle Zerlegung eines Kommunikationsvorganges in eine Hierarchie von mehreren Funktionsschichten. Dabei baut jede Schicht auf den darunter liegenden auf. Eine Applikation ist nun in der Lage, die oberste Schicht zur Übertragung zu verwenden, die unterste bedient sich dann des eigentlichen Übertragungsmediums zum Austausch der Bitströme. Dazwischen liegt das `OSI-Referenzmodell`, das in sieben Schichten unterteilt ist. Die Anzahl der sieben Schichten kommt seltsamerweise jedoch eher aus der babylonischen Mythologie als aus einer technisch entstanden Notwendigkeit [UNIX1 pp164]. In *Abbildung 1-4* ist das Modell graphisch dargestellt [KOMM1 pp1-17].

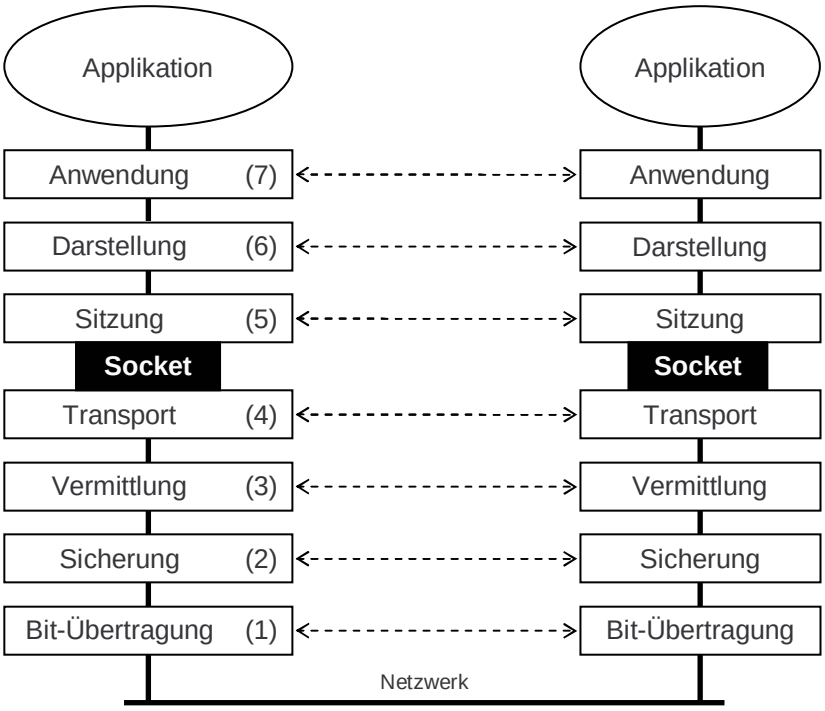

Abbildung 1-4 Socket im OSI-Referenzmodell

Da jede höher gelegene Schicht auf der darunter liegenden aufbaut, muss jede Schicht Dienste zur Verfügung stellen, die von der darüber liegenden verwendet werden können. Die Applikation verwendet nur Dienste der obersten Schicht, die in weiterer Folge jeweils auf die Dienste der darunter liegenden Schichten zurückgreifen, und zwar so lange, bis das Netzwerksystem erreicht ist. Dort werden die Bits real übertragen. Beim anderen Kommunikationspartner erfolgt der Datentransfer in umgekehrter Richtung. Virtuell kann jedoch jede Schicht so gesehen werden, dass sie mit der gleichen Schicht des Kommunikationspartners kommuniziert, da sie kein Wissen über die unter ihr liegenden Schichten (außer den ihr bekannten Diensten) besitzt. In *Abbildung 1-4* wird dies mittels der waagerechten, strichlierten Linien angedeutet.

Ein Socket stellt eine Verbindung zwischen zwei Prozessen dar. Daher kann er als Dienst der vierten Schicht (Transportschicht) angesehen werden [WSNP1 pp11-29]. Aus diesem Grund wird nun ein kurzer Überblick über die Aufgaben der vierten Schicht wiedergegeben:

Aufgabe der vierten Schicht ist es, Endsystemverbindungen zu einer Teilnehmerverbindung zu erweitern. Damit wird es möglich, über ein physikalisches

Netzwerk mehrere logische, voneinander unabhängige Verbindungen aufzubauen. Diese Schicht wurde speziell für den Anwendungsbereich der Dokumentenübertragung definiert und ist daher einfach gehalten. Ein simples Verwenden der Dienste der Schicht drei reicht jedoch nicht aus, da Funktionen zur Verbesserung der Dienstgüte und zur Optimierung der Nutzung von Endsystemverbindungen implementiert werden müssen. Dazu gehören Methoden der Fehlererkennung und Fehlerbehebung, aber auch Methoden für das Multiplexen von Verbindungen. Den Benutzern der Schicht vier soll somit ein Dienst unabhängig von der Netzrealisierung und frei von Überlegungen zur Wegewahl zur Verfügung gestellt werden. Die Transportschicht soll für diese Aufgabe die Dienste der Vermittlungsschicht optimal nützen, und auch eine Minimierung der Kosten ermöglichen [KOMM1 pp75].

In *Abbildung 1-5* kann die Abfolge einer Kommunikation zwischen zwei Benutzern der Schicht vier betrachtet werden. Als erstes sendet der anfordernde Benutzer eine Anforderung (Request) mittels eines Dienstes an die darunter liegende Schicht. Diese wird dem anderen Benutzer mittels einer Signalisierung (Indication) angezeigt, worauf sie von diesem bearbeitet wird. Nun kann eine Antwort (Response) an den auslösenden Dienst zurück gesendet werden. Dieser erhält anschließend eine Bestätigung (Confirm) [KOMM1 pp77-83].

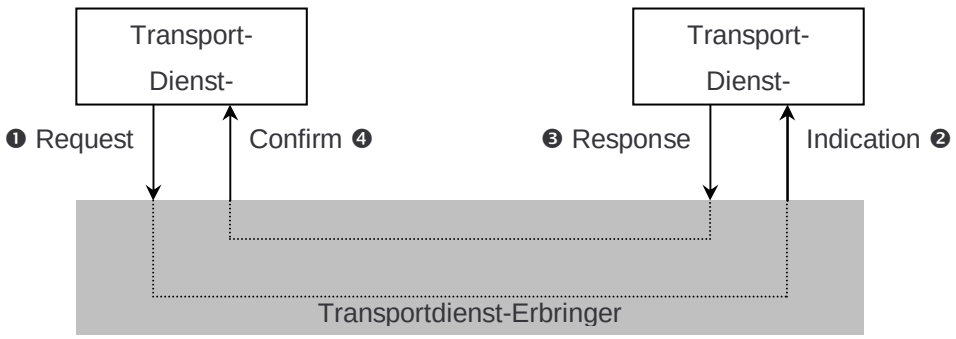

Abbildung 1-5 Dienstelemente

Eine Teilnehmerverbindung kann auch noch auf eine andere Art beschrieben werden (*siehe Abbildung 1-6*). In diesem Modell sind zwei Teilnehmer dargestellt, die über zwei Warteschlangen miteinander kommunizieren. Jeder Teilnehmer kann seine Daten in die jeweilige Warteschlange stellen, woraufhin

sie vom anderen Teilnehmer zu gegebenem Zeitpunkt wieder entnommen werden. Die Warteschlangen selbst besitzen eine begrenzte Kapazität. Daten werden in der Reihenfolge entnommen, wie sie in die Warteschlange gestellt wurden (FIFO - First In First Out); es sei denn, es handelt sich um Vorrang-Daten. An diesem Modell kann erkannt werden, dass es sich um eine transparente Duplex-Verbindung handelt, da je eine Warteschlange in jede Richtung existiert [KOMM1 pp77-83].

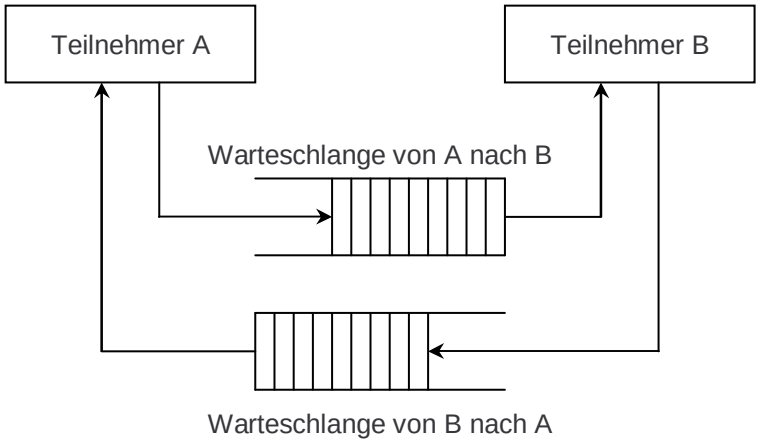

Abbildung 1-6 Modell mit Warteschlangen

Parameter für die `Dienstgüte` werden beim Erstellen einer Verbindung zwischen zwei Partnern vereinbart und gelten für die gesamte Dauer der Verbindung. Als Beispiel könnte eine Maximaldauer für den Aufbau oder eine Maximalanzahl von gescheiterten Versuchen einer Teilnehmerverbindung genannt werden. Ebenso zählen Merkmale für die Datensicherheit oder Prioritäten dazu. Es handelt sich somit um Parameter, die es ermöglichen, eine Verbindung etwas genauer in ihren Eigenschaften zu spezifizieren. Diese können, müssen aber nicht angegeben werden; eine Veränderung bei einer bestehenden Verbindung ist jedoch nicht möglich [KOMM1 pp79].

Der Aufbau einer Verbindung kann aktiv nur von einer Seite aus erfolgen, da die zweite bereits auf einen Verbindungsaufbau vorbereitet sein muss. Dies erfolgt durch einen Prozess, der auf eine aufzubauende Verbindung wartet. Andernfalls weiß das empfangende System nicht, welche Verbindung gewünscht wird. Kommt es zu einer Verbindung, retourniert dieser wartende Prozess den verbundenen Socket.

Die Beendigung einer bestehenden Verbindung kann von beiden Partnern zu jedem beliebigen Zeitpunkt initiiert werden, sowohl von der gerufenen Instanz aus, als auch von der aufrufenden. Der Abbruchwunsch kann vom anderen Partner nicht abgelehnt werden [KOMM1 pp75-84].

1.3.2 Socket in Java

In Java stellt die Klasse `Socket` einen verbindungsorientierten Socket dar. Dies bedeutet, dass es sich um eine Punkt-zu-Punkt-Verbindung handelt, es wird auch von Unicast-Verbindung gesprochen. Eine Verbindung zu mehreren Partnern (multicast) und somit auch eine Verbindung zu allen anderen Partnern (broadcast) ist nicht möglich. Daher müssen die Verbindungsdaten, z. B. Adressen und Ports, beim Verbindungsaufbau angegeben werden und bleiben für die gesamte Dauer der Verbindung unverändert. Um die Daten austauschen und die Verbindungen verwalten zu können, wird in Java standardmäßig das Protokoll TCP verwendet.

Für den Socket in Java sind zwei Klassen sehr wichtig: Die Klasse `ServerSocket` wird verwendet, um auf einen Verbindungsaufbau zu warten, die Klasse `Socket` ermöglicht den Verbindungsaufbau zu einem wartenden Server-Socket. Wurde eine Verbindung etabliert, retourniert die Instanz der Klasse `ServerSocket` eine bereits verbundene Instanz der Klasse `Socket`. Ab diesem Zeitpunkt sind beide Verbindungspartner gleich, da sie dieselbe Funktionalität besitzen und gleichberechtigt sind. Über diese beiden Instanzen der Klasse `Socket` kann nun die Kommunikation abgewickelt werden [JAVA2 pp451-469] [TIJ1 pp815-826].

Zum Datenaustausch selbst werden in Java Streams verwendet; der Socket ist nur für das Verbindungsmanagement zuständig. Es handelt sich dabei um die beiden Klassen `InputStream` (Daten können ausgelesen werden) und `OutputStream` (Daten können versendet werden). Jede Socket-Implementierung besitzt Methoden, die es ermöglichen, einen der genannten Streams von ihr zu erhalten. Erst die Streams enthalten die notwendigen Methoden zum Datenaustausch, wie z. B. `read` oder `write` [JAVA2 pp451-469] [TIJ1 pp815-826]. In

Java wurde somit eine klare Trennung zwischen dem Verbindungsmanagement und dem eigentlichen Datenaustausch vorgenommen. In der Klasse `Socket` und `ServerSocket` befindet sich der steuernde Teil des Sockets; in den Klassen `InputStream` und `OutputStream` jener Teil, der die Daten überträgt.

Das Verwenden eines Sockets in Java ist einfach. Bei einem Server ist eine Instanz der Klasse `ServerSocket` zu erstellen. Dabei ist die Adresse anzugeben, auf der auf eine Verbindung gewartet werden soll. Um tatsächlich auf einen Verbindungsaufbau zu warten, muss die Methode `accept` dieser Instanz aufgerufen werden. Nachdem eine Verbindung etabliert ist, retourniert diese Methode eine neue Instanz der Klasse `Socket`, die bereits mit dem rufenden Partner verbunden.

Ein Client erstellt die gewünschte Instanz der Klasse `Socket`. Dabei muss als Parameter die Adresse des wartenden Server-Sockets angegeben werden. Dadurch wird automatisch die Verbindung hergestellt. Konnte die Instanz fehlerfrei erstellt werden, kann diese für die Kommunikation verwendet werden. Andernfalls wird eine Exception ausgelöst.

In *Abbildung 1-7* wurde ein Kommunikationsprozess zwischen Server und Client schematisch dargestellt. Nachdem sowohl der Client als auch der Server die Instanz der Klasse `Socket` besitzen, können von diesen eine Instanz der Klasse `InputStream` und/oder der Klasse `OutputStream` erhalten werden. Mit Hilfe der Instanzen der Stream-Klassen erfolgt schließlich der Datenaustausch; über die Instanz der Klasse `InputStream` können mittels der Methode `read` Daten ausgelesen werden, über die Instanz der Klasse `OutputStream` könnten mittels der Methode `write` Daten versendet werden. Ist die Kommunikation beendet, werden die Instanzen der Klasse `Socket` mittels der Methode `close` geschlossen. Zuvor sollten jedoch auch die Instanzen der Stream-Klassen geschlossen werden. Die Instanz der Klasse `ServerSocket` kann nach dem `accept` zu einem beliebigen Zeitpunkt geschlossen werden, denn sie hat nichts mit der geöffneten Verbindung zu tun. Sie könnte aber auch wieder für ein weiteres `accept` aufgerufen werden. Dadurch würde sofort wieder auf einen weiteren Verbindungsaufbau gewartet werden.

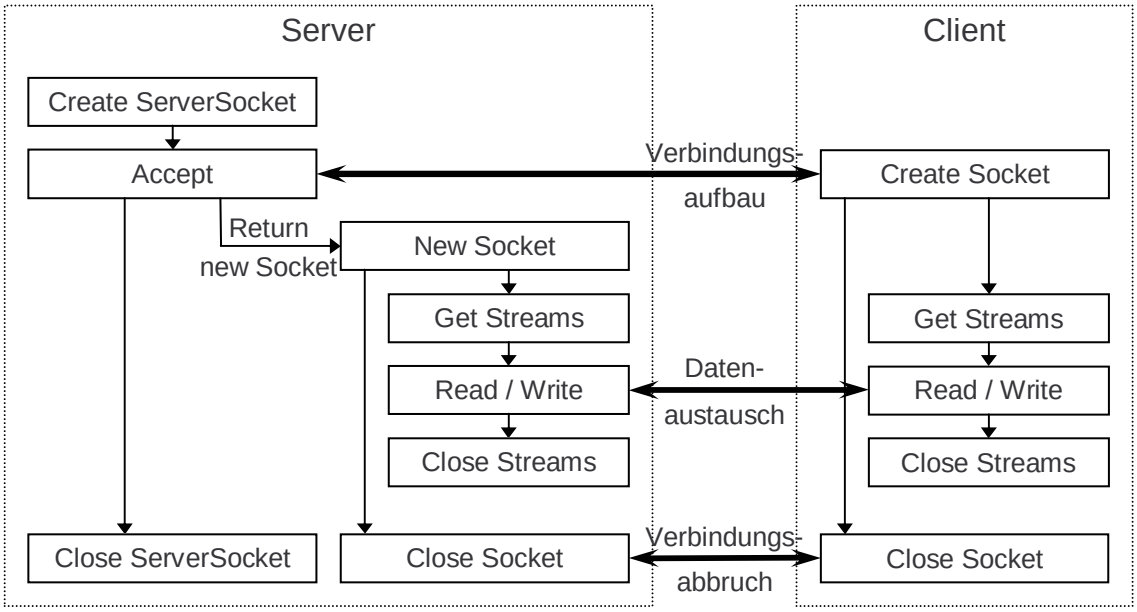

Abbildung 1-7 Verwendung von Sockets in Java

1.3.3 Datagram-Socket in Java

Da das TCP wegen seiner Datensicherung und Verbindungsorientiertheit einen großen Overhead besitzt, wurde ein weiteres, einfacher gestaltetes Protokoll entwickelt, das UDP (User Datagram Protocol). In TCP wird garantiert, dass gesendete Datenpakete in der richtigen Reihenfolge ankommen, keine Datenpakete verloren gehen und duplizierte Datenpakete kein Fehlverhalten hervorrufen. Beim UDP sind all diese Mechanismen nicht vorhanden [JAVA2 pp508] [TIJ1 pp827]. Weiters ist der Datagram-Socket nicht verbindungsorientiert, wodurch sich auch der Overhead verringert. Daher muss nicht zuerst eine Datenverbindung aufgebaut werden, um Daten austauschen zu können. Der Datagram-Socket ist in der Klasse `DatagramSocket` implementiert. Der Datentransfer erfolgt durch den Austausch von Datenpaketen; beim Socket werden im Gegensatz dazu Datenströme (Streams) übertragen. Daher sind die Methoden `send` und `receive` für das Senden und Empfangen von Datenpaketen bereits in der Klasse `DatagramSocket` implementiert. Da keine speziellen Datenverbindungen aufgebaut werden müssen, gibt es die Möglichkeit, Daten unicast, multicast und somit auch broadcast auszutauschen. Dies bedeutet, dass ein Datenpaket an einen einzigen, mehrere oder alle Knoten im Netz gesendet werden [JAVA2 pp508-521] [TIJ1 pp827-833].

Beim Datagram-Socket können im Gegensatz zum Socket nicht einfach nur Daten versendet und empfangen werden, da die Adressinformation bei jedem Schreiben angegeben werden muss und beim Lesen abgefragt werden sollte. Daher wird für diese Art der Übertragung ein so genanntes `DatagramPacket` verwendet; es enthält neben den Daten auch die notwendigen Adress-informationen [TIJ1 pp827-833].

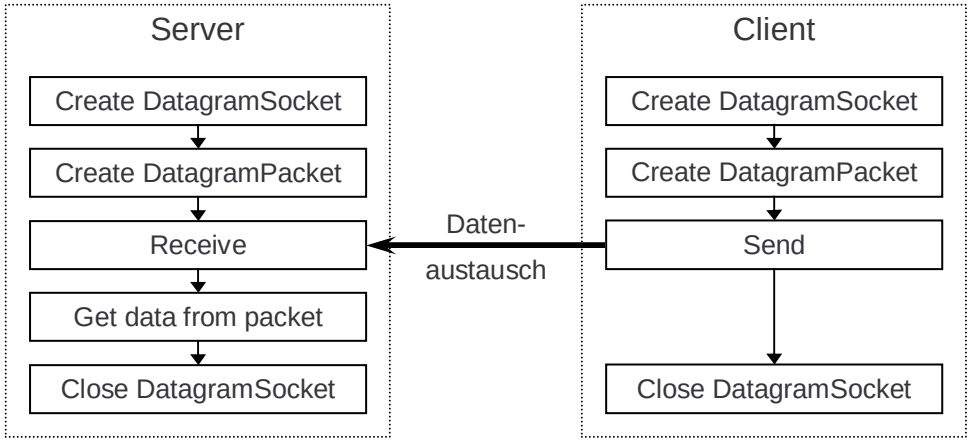

Abbildung 1-8 Verwendung von Datagram-Sockets in Java

Die Übertragung eines Datenpaketes mittels einem Datagram-Socket in Java ist besonders einfach (*siehe Abbildung 1-8*). Beim Server muss eine Instanz der Klasse `DatagramSocket` erstellt werden. Zusätzlich wird eine leere Instanz der Klasse `DatagramPaket` erstellt, welches das zu empfangende Paket aufnehmen kann. Anschließend wird mittels `receive` auf ein Datenpaket gewartet. Wurde eines empfangen, können die Daten und die Adresse des Senders der Instanz der Klasse `DatagramPaket` entnommen werden. Danach kann die Instanz der Klasse `DatagramSocket` wieder geschlossen, oder für eine andere Aufgabe verwendet werden. Der Client ist dem Server sehr ähnlich. Es wird ebenfalls eine Instanz der Klasse `DatagramSocket` erstellt. Weiters muss eine Instanz der Klasse `DatagramPacket` mit den zu versendenden Daten erstellt werden. Anschließend wird das Paket mittels `send` versendet. Auch diese Instanz der Klasse `DatagramSocket` kann nun entweder geschlossen oder für eine andere Aktion verwendet werden.

1.3.4 Sockets in C

Da das ursprüngliche Socket-Interface für BSD-Linux entwickelt wurde, wird nun auch im Detail auf diese Schnittstelle eingegangen [UNIX3] [CNW1 pp349-364]. Dies ist schon allein deshalb notwendig, da alle diese Funktionen auch in der Java-Implementierung Verwendung finden, wenn auch nicht für den Anwender ersichtlich. So werden z. B. in Java beim Erstellen einer Instanz der Klasse `Socket` automatisch die Funktionen `socket`, `bind` und `connect` aufgerufen.

In den Unix-Systemen ist der Socket vollständig in den übrigen Ein- und Ausgabefunktionen (wie z. B. auch der Zugriff auf eine Datei) integriert. Dies ist daran erkennbar, dass bei der Erstellung eines Socket-Objektes ein Deskriptor erzeugt und retourniert wird. Soll ein Datenaustausch über dieses Objekt erfolgen, können die Lese- und Schreibbefehle des Betriebssystems verwendet werden. Dabei ist diesen der Deskriptor des Objektes zu übergeben, anhand dessen das Betriebssystem erkennen kann, wie die Daten zu behandeln sind. Damit besteht beim Datenaustausch über einen Socket, oder bei der Bearbeitung einer Datei, kein Unterschied [CNW1 pp353].

Im Folgenden werden die wichtigsten Funktionen der Socket-Schnittstelle beschrieben. Da alle Funktionen einen negativen Wert retournieren wenn ein Fehler aufgetreten ist, wird dies nicht bei jeder Funktion gesondert beschrieben; einem positiven Rückgabewert kann entnommen werden, dass kein Fehler aufgetreten ist oder dieser entspricht der gewünschten Information (z. B. bei `getQueuesize`).

Socket

Mit `socket` wird ein neues Socket-Objekt erstellt.

```
int descriptor = socket (
    int           protofamily,
    int           type,
    int           protocol
);
```

Als Ergebnis liefert diese Funktion den Deskriptor für dieses Socket-Objekt; er wird in dieser Arbeit auch Socket-ID genannt. Damit kann dieses Objekt immer angesprochen werden. Der Parameter `protofamily` definiert, welches Protokoll verwendet werden soll (z. B. IP oder AppleTalk). Mit der Angabe `type` wird angegeben, ob es sich um einen RAW-Socket, um einen verbindungsorientierten Socket, um eine verbindungslosen Socket oder um einen sequenziellen Socket handeln soll. Für diese Diplomarbeit sind nur die Typen verbindungsorientiert und verbindungslos von Bedeutung, da nur diese in Java unterstützt werden; sie werden durch die Klassen `java.net.Socket` und `java.net.DatagramSocket` zur Verfügung gestellt. Mit dem Parameter `protocol` kann zusätzlich ein bestimmtes Transportprotokoll definiert werden [CNW1 pp354-355] [UNIX3 socket].

Close

Das durch den Parameter `descriptor` gegebene Socket-Objekt wird geschlossen.

```
int close (
    int           descriptor
);
```

Wenn nötig wird der Kommunikationspartner über diesen Vorgang informiert, damit auch dieser dementsprechende Vorgänge in die Wege leiten kann. Was genau beim Schließen geschieht hängt von dem Socket-Typ ab [UNIX3 close].

Bind

`Bind` weist dem durch den Parameter `descriptor` gegebnen Socket-Objekt eine lokale Adresse zu; damit kennt dieses seine eigene Adresse.

```
int bind (
    int           descriptor,
    sockaddr*     localaddress,
    int           addresslen
);
```

Mit dem Parameter `localaddress` wird dieser Funktion die lokale Adresse in Form einer `sockaddr`-Struktur [CNW1 pp356] übergeben; in `addresslen` muss weiters die Länge des reservierten Speicherplatzes für diese Adresse angeben werden. Diese Funktion muss immer als erstes aufgerufen werden, wenn ein erstelltes Socket-Objekt verwendet werden soll; ansonsten kennt dieses nicht seine eigene Adresse [UNIX3 bind].

Listen

Ein Socket-Objekt, das auf einen Verbindungsaufbau wartet, wird Server-Socket genannt. Damit dieses als ein solches verwendet werden kann, muss es mittels `listen` darauf vorbereitet werden.

```
int listen (
    descriptor,
    queuesize
);
```

Beim Aufruf dieser Funktion wird das durch den Parameter `descriptor` gegebene Socket-Objekt in den passiven Modus versetzt. Weiters wird die Queuegröße (Parameter `queuesize`) festgelegt. Nach dem Aufruf dieser Funktion, kann mittels der Funktion `accept` auf einen Verbindungsaufbau gewartet werden [UNIX3 listen].

GetQueuesize

Diese Funktion ermittelt und retourniert die Größe der Empfangsqueue für das gegebene Socket-Objekt (Parameter `descriptor`).

```
int queuesize = getQueuesize (
    int           descriptor
);
```

Accept

Ein Socket-Objekt, das durch `listen` in den passiven Modus versetzt wurde, kann mittels `accept` auf einen Verbindungsaufbau warten.

```
int newdescriptor = accept (
    int           descriptor,
    sockaddr*     connectedaddress,
    int           addresslen
);
```

Diese Funktion wir so lange blockiert, bis ein Client eine Verbindung zu diesem Objekt aufgebaut hat. Mit dem Parameter `descriptor` wird wie üblich das zu verwendende Socket-Objekt angegeben. Kann eine Verbindung erfolgreich etabliert werden, retourniert diese Funktion ein neues Socket-Objekt, dargestellt durch einen neuen Deskriptor (Rückgabewert `newdescriptor`). In dem Speicherbereich, angegeben durch den Parameter `connectedaddress`, wird zusätzlich die Adresse des Verbindungspartners abgelegt; die Größe des dafür reservierten Speichers muss durch den Parameter `addresslen` bekant gegeben werden. Das neu erstellte Socket-Objekt (`newdescriptor`) kann schließlich für die Kommunikation verwendet werden. Der weiterhin bestehende Server-Socket (`descriptor`) kann mittels eines erneuten Aufrufes von `accept` für einen weiteren Verbindungsaufbau verwendet werden. Sollte er nicht mehr benötigt werden, kann er aber auch mit `close` geschlossen werden [UNIX3 accept].

Connect

`Connect` ist das genaue Gegenstück zu `accept`; damit kann eine Verbindung zu einem wartenden Server-Socket aufgebaut werden.

```
int connect (
    int           descriptor,
    sockaddr*     destaddress,
    int           addresslen
);
```

Zu diesem Zweck muss mittels des Parameters `destaddress` die Adresse angegeben werden, zu der die Verbindung aufgebaut werden soll; der Parameter `addresslen` gibt deren Länge an. Wurde eine Verbindung etabliert,

kann das zugehörige Socket-Objekt, gegeben durch den Parameter `descriptor`, für den Datenaustausch verwendet werden [UNIX3 connect].

GetPeerName, GetHostName

Besteht Interesse an der Adresse des Verbindungspartners, kann diese mit der Funktion `getPeerName` ermittelt werden; die Funktion `getHostName` retourniert die lokale Adresse des gegebenen Socket-Objektes [CNW1 pp362].

```
int getHostName (
    int           descriptor,
    sockaddr*     address,
    int           addresslen
);

int getPeerName (
    int           descriptor,
    sockaddr*     address,
    int           addresslen
);
```

Mit dem Parameter `descriptor` muss das gewünschte Socket-Objekt angegeben werden. Im Speicherbereich von `address` wird beim Aufruf einer dieser Funktionen die gewünschte Adresse gespeichert. Mit dem Parameter `addresslen` muss angegeben werden, wie groß der für die Adresse zur Verfügung gestellte Speicherbereich ist.

GetHostByName, GetHostByAddress

Mit diesen Funktionen kann eine IP-Adresse zu einem Domainnamen erhalten werden oder umgekehrt zu einer IP-Adresse der Domainname [CNW1 pp362]. Die Domain `www.ict.tuwien.ac.at` hat z. B. die IP-Adresse 128.131.80.9.

```
struct hostent* gethostbyname (
    const char*   name
);
```

```
struct hostent* gethostbyaddr (
    const char*   addr,
    int len,
    int type
);
```

Die Rückgabewerte beider Funktionen sind vom Typ `hostent`. Dabei handelt es sich um eine Struktur, die in `netdb.h` definiert ist. Sie enthält den Namen der IP-Adresse, alle zusätzlichen Namen (so genannte alias) und auch alle IP-Adressen die dieser Name besitzt. Der Parameter `name` enthält den Namen für den die IP-Adresse gesucht werden soll. Beim Parameter `addr` handelt es sich um die Angabe der IP-Adresse, für die der Name gesucht werden soll. Weiters muss im Parameter `len` angegeben werden, wie lange die Adresse ist und mit `type` um welchen Typ von Adresse es sich handelt.

SetSockOpt, GetSockOpt

Die Funktion `setsockopt` ermöglicht es, Optionen einer Verbindung (z. B. Timeouts) zu verändern; `getsockopt` ermöglicht es, Optionen abzufragen.

```
int getsockopt (
    int           descriptor,
    int           level,
    int           optname,
    char*         optval,
    int*          optlen
);

int setsockopt (
    int           descriptor,
    int           level,
    int           optname,
    char*         optval,
    int           optlen
);
```

Mit dem Parameter `descriptor` ist jeweils anzugeben, welches Socket-Objekt zu bearbeiten ist. Da Optionen für unterschiedliche Protokolllevels existieren, ist dieser mittels des Parameters `level` anzugeben. Über den Parameter `optname` wird die zu verwendende Option ausgewählt, `optval` enthält den Wert und `optlen` gibt an, wie viele Bytes diese Information besitzt [CNW1 pp362] [UNIX3 getsockopt, setsockopt].

Read

Die Funktion `read` erlaubt es, Daten von dem gegebenen verbindungsorientierten Socket-Objekt (`descriptor`) zu lesen.

```
int bytesread = read (
    int           descriptor,
    char*         buffer,
    int           length
);
```

Es werden maximal so viele Bytes gelesen, wie im Parameter `length` angegeben wurde. Die gelesenen Bytes werden in das Bytearray `buffer` abgelegt. Sind weniger Bytes in der Queue vorhanden als übergeben werden könnten, werden nur die vorhandenen ausgelesen. Der Rückgabewert der Funktion (`bytesread`) gibt darüber Auskunft, wie viele Bytes tatsächlich gelesen wurden [UNIX3 read].

Weiters ist anzumerken, dass `read` eine synchrone Funktion ist. Dies bedeutet, dass sie so lange wartet, bis entweder Daten von dem Socket-Objekt gelesen werden können, bis die Verbindung unterbrochen wird oder bis ein Fehler aufgetreten ist. Erst nach einem dieser Vorgänge kehrt diese Funktion zum Aufrufer zurück.

Write

Mittels `write` können Daten an den Verbindungspartner eines verbindungsorientierten Socket-Objektes gesendet werden.

```
int byteswritten = write (
    int           descriptor,
    char*         data,
    int           length
);
```

Die zu sendenden Daten, mit der Anzahl von `length` Bytes, werden dem angegebenen Bytearray `data` entnommen. Auch diese Funktion ist, so wie `read`, eine synchrone Funktion. Es kann jedoch sein, dass die zu sendenden Daten aus Gründen der Geschwindigkeitsoptimierung zuerst in einer Queue gepuffert werden (*siehe die Funktion Flush*). In diesem Fall stellt `write` die zu schreibenden Daten in diese Queue und retourniert sofort. Der Rückgabewert (`byteswritten`) enthält die Anzahl der versendeten Bytes [UNIX3 write].

Flush

Will der Sender sicherstellen, dass seine Daten sofort versendet werden, und nicht erst in einer Queue zwischengespeichert werden - dies kann für eine Optimierung der Übertragungsgeschwindigkeit sinnvoll sein - kann er dazu diese Funktion aufrufen. Dadurch wird bewirkt, dass alle Daten, die sich noch in der Queue befinden, sofort versendet werden.

```
int flush (
    int           descriptor
);
```

Ein Beispiel, wo es sinnvoll ist, eine Optimierung für die Übertragungsgeschwindigkeit vorzunehmen, ist eine Terminalverbindung. Dabei handelt es sich um eine Verbindung zu einem anderen Rechner, wobei die Tasteneingaben über eine Socket-Verbindung an den anderen Rechner gesendet werden, und die Bildschirmausgabe wieder über diese retourniert wird. Werden nun Tasteneingaben durchgeführt, würde jedes einzelne Zeichen einzeln versendet werden. Werden hingegen, z. B. 32 Tastenanschläge zusammengefasst und dann als Paket übertragen, ist die Belastung am Datenbus weit aus geringer, da jedes einzelne Datenpaket einen Overhead mit sich schleppt. Über

eine solche Terminalverbindung werden jedoch häufig Befehle für ein Betriebssystem übermittelt. Soll z. B. ein Directory gewechselt werden, wird dazu der Befehl „cd directory“ eingeben. Abgeschlossen wird eine solche Eingabe immer mit der Taste „Enter“. Würde nun jedoch die Optimierung der Übertragung feststellen, dass z. B. noch zu wenige Bytes in der Queue stehen um sie zu übertragen, würde dieser Befehl nicht ausgeführt werden, da er noch nicht an das andere System gesendet wurde. Daher wird in einem Terminalprogramm z. B. immer nach Betätigen der Taste „Enter“ ein `flush` auf das zugehörige Socket-Objekt durchgeführt. Damit wird sichergestellt, dass der eingegebene Befehl versendet wird.

Receive

Diese Funktion empfängt von dem gegebenen Socket-Objekt ein Datenpaket. Diese Funktion gibt es in zwei Formen. Bei der ersten (`recv`) können Datenpakete empfangen werden, es ist jedoch nicht explizit bekannt, wer dieses versendet hat. Daher scheint diese Form für verbindungsorientierte Sockets geeignet zu sein. Die zweite Form (`recvfrom`) ermöglicht auch das Übergeben der Sendeadresse, und ist somit gut für verbindungslose Sockets geeignet.

```
int recv (
    int           descriptor,
    char*         buffer,
    int           length,
    int           flags
);

int recvfrom (
    int           descriptor,
    char*         buffer,
    int           length,
    int           flags,
    sockaddr*     sendaddress,
    int           addresslen
);
```

Wie bei allen anderen Funktionen ist mit dem Parameter `descriptor` das zu verwendende Socket-Objekt anzugeben. Über den Parameter `buffer` wird der Datenspeicher, in den das Paket zu schreiben ist, übergeben; `length` definiert dabei wie groß dieser Speicherbereich ist. Mit dem Parameter `flags` können gewisse Details definiert werden. So ist es z. B. möglich, ein Datenpaket von dem Socket-Objekt nicht konsumierend zu lesen. Dies bedeutet, dass es erhalten wird, aber weiterhin auch im Socket-Objekt gespeichert bleibt. Über die Parameter `sendaddress` und `addresslen` kann die Sendeadresse erhalten werden; `sendaddress` ist ein Zeiger auf einen Speicherbereich, `addresslen` gibt an, wie groß dieser ist. Der Rückgabewert dieser Funktion gibt Auskunft darüber, wie viele Bytes vom Socket-Objekt gelesen wurden [UNIX3 recv] [CNW1 pp360-361].

Send

Send versendet ein Datenpaket über das gegebene Socket-Objekt.

```
int send (
    int           descriptor,
    char*         data,
    int           length,
    int           flags
);

sendto (
    int           descriptor,
    char*         data,
    int           length,
    int           flags,
    sockaddr*     destaddress,
    int           addresslen
);
```

Auch diese Funktion existiert so wie `recv` in zwei Varianten. Bei der ersten (`send`) kann der Empfänger nicht explizit angegeben werden; damit ist diese

Art für verbindungsorientierte Sockets geeignet. Bei der zweiten Version (`sendto`) muss immer ein Empfänger angegeben werden; sie ist daher für verbindungslose Sockets geeignet.

Die Beschreibung der Parameter kann der Funktionen `recv` und `recvfrom` entnommen werden. Der Unterschied besteht jedoch darin, dass hier Daten zur Verfügung gestellt werden müssen und nicht erhalten werden. Abschließend kann noch erwähnt werden, dass mittels der Funktion `connect` einem verbindungslosen Socket eine bevorzugte Zieladresse übergeben werden kann. In diesem Fall werden Daten mittels `send` immer an diese Adresse gesendet. Ein Aufruf von `send` bei einem verbindungslosen Socket, ohne vorheriger Definition der Zieladresse, würde hingegen eine Fehlermeldung zur Folge haben.

Verwenden der Sockets in C

Nachdem alle Funktionen beschrieben sind, sollte noch auf deren Verwendung eingegangen werden. In C ist das Verwenden der Sockets etwas komplizierter (*siehe Abbildung 1-9*) als in Java, da in Java bereits viele Teilschritte automatisiert sind.

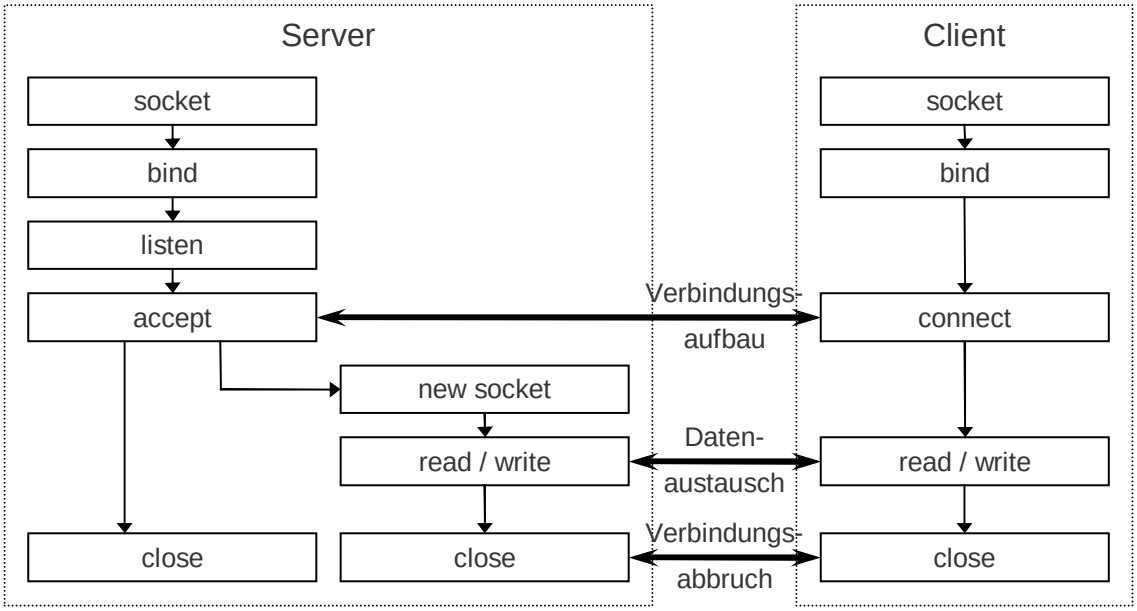

Abbildung 1-9 Verwendung von verbindungsorientierten Sockets in C

Als erstes Beispiel wird auf einen verbindungsorientierten Socket eingegangen. Dazu erstellt ein Server mittels `socket` ein Objekt eines Sockets. Diesem muss mittels der Funktion `bind` die eigene lokale Adresse mitgeteilt werden. Anschließend kann es mittels `listen` in den passiven Modus versetzt werden. Nun ist dieses Objekt als Server-Socket deklariert und kann mittels `accept` auf einen Verbindungsaufbau warten. Wird eine Verbindung erstellt, retourniert `accept` eine neue Socket-ID; sie kennzeichnet das neue verbundene Socket-Objekt.

Ein Client erstellt ebenfalls mit `socket` ein Objekt eines Sockets und teilt diesem mittels `bind` die lokale Adresse mit. Da es in diesem Fall häufig gleichgültig ist, wie die lokale Adresse lautet, kann eine beliebige frei Adresse verwendet werden (dies erfolgt durch Angabe der Portnummer 0). Nun kann dieses Socket-Objekt eine Verbindung mittels `connect` zu einem wartenden Server-Socket erstellen.

Nachdem die Verbindung aufgebaut wurde, kann mittels `read` und `write` und den Deskriptoren der beiden verbundenen Socket-Objekten der Datenstrom ausgetauscht werden. Soll die Verbindung abgebrochen werden, erfolgt dies mittels der Funktion `close`. Jenes Socket-Objekt, das für den Funktionsaufruf `accept` verwendet wurde (Server-Socket), besteht weiterhin. Es kann nach dem Aufruf von `accept` geschlossen werden, da es nichts mehr mit der aufgebauten Verbindung zu tun hat, könnte aber auch für einen weiteren Verbindungsaufbau genutzt werden, indem wieder die Funktion `accept` aufgerufen wird.

Bei einem verbindungslosen Socket (*siehe Abbildung 1-10*) erstellen sowohl der Server als auch der Client mittels `socket` eine Objekt des verbindungslosen Sockets. Für beide muss ebenfalls mittels `bind` die lokale Adresse definiert werden. Für diese Adresse kann so wie beim verbindungsorientierten Socket (siehe vorheriges Beispiel) eine beliebige freie Adresse ermittelt und verwendet werden. Um ein Datenpaket empfangen zu können, muss der Server auf ein solches warten. Dazu muss die Funktion `recvfrom` für das gewünschte Objekt des Sockets aufgerufen werden. Wird ein Datenpaket empfangen, werden die

empfangenen Daten retourniert und können verwendet werden. Anschließend kann das verwendete Socket-Objekt geschlossen, oder für andere Aktionen verwendet werden. Der Client kann nach dem `bind` mittels der Funktion `send` die gewünschten Daten versenden. Danach kann auch dieses Objekt geschlossen oder anderwärtig verwendet werden.

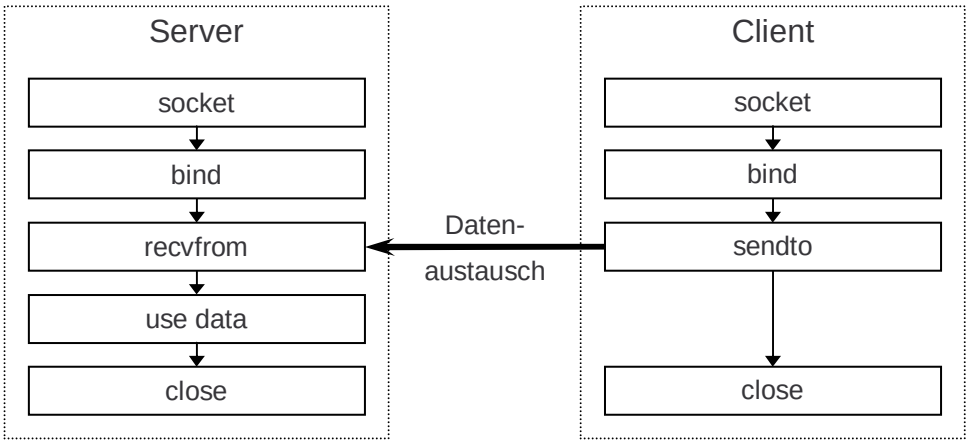

Abbildung 1-10 Verwendung von verbindungslosen Sockets in C

1.4 Der AL1394

Die Bezeichnung `AL1394` ist eine Abkürzung für `Abstraction Layer for IEEE 1394`. Er stellt eine `API (Application Programming Interface)` dar, die vom Institut für Computertechnik (ICT) an der TU-Wien entwickelt wurde [ALC1] [ALC2]. Diese API ist eine Abstraktionsebene für alle IEEE 1394-Interfaces. Da es derzeit für unterschiedliche Adapter verschiedene APIs gibt, ist jede Applikation an einen Adapter dieses Herstellers gebunden. Mit dem AL1394 wurde eine Vereinheitlichung aller Adapter erreicht und somit das Problem der Abhängigkeit gelöst. Um die Implementierung der IEEE 1394-Sockets so unabhängig von einem System wie nur möglich zu gestalten, baut sie auf dem AL1394 auf. Dieser bringt auch andere Vorteile mit sich, wie z. B. eine konzeptionelle Lösung für das Problem der Busresets (*siehe Abschnitt 1.4.4*).

1.4.1 Aktiver asynchroner Datenaustausch

Der AL1394 ermöglicht es, asynchrone Datenpakete mittels des Befehls `WriteNodeID` an eine Adresse in einem anderen Adapter zu senden. Als Gegenstück dazu können mittels `ReadNodeID` asynchrone Datenpakte von einer Adresse eines anderen Adapters gelesen werden. Die Operation `LockNodeID` führt eine atomare Lese- und Schreibfunktion aus. Dazu muss diesem Befehl der vermutete Dateninhalt, der überschrieben werden soll, und der neue Dateninhalt übergeben werden. Die Lockoperation liest nun den angegebenen Speicherbereich des angegebenen Adapters aus, vergleicht ihn mit dem vermuteten Dateninhalt und schreibt den neuen Dateninhalt nur dann, wenn sich der vermutete und der existierende Dateninhalt nicht unterscheiden. Die Applikation wird anschließend darüber informiert, ob diese Operation Erfolg hatte oder nicht.

1.4.2 Passiver asynchroner Datenaustausch

Als passiver Datenaustausch werden in dieser Arbeit jene Befehle bezeichnet, die für den Datenaustausch zwar notwendig sind, aber nicht selbst aktiv werden können. Diese Befehle warten auf ankommende Datenpakete und ermöglichen anschließend deren Verarbeitung. Beim AL1394 handelt es sich dabei um den Befehl `MapAddressRange`. Dadurch wird ein Speicherbereich des IEEE 1394-Adapters einer Applikation zugeordnet. Diese wird nun über die Vorgänge in diesem Speicherbereich benachrichtigt und kann die notwendigen Aktionen ausführen. Würde z. B. ein `WriteNodeID` Daten auf diesen Speicherbereich schreiben, würde der zugeordnete Prozess informiert werden, und könnte nun die empfangenen Daten aus diesem Speicherbereich auslesen und sie anschließend verarbeiten.

1.4.3 Isochroner Datenaustausch

Derzeit existiert für den `isochronen` Bereich des `AL1394` nur eine Spezifikation, jedoch noch keine Implementierung, weshalb auch nicht weiter

darauf eingegangen werden kann. Außerdem wurde in dieser Diplomarbeit die Implementierung des isochronen IEEE 1394-Sockets nicht durchgeführt.

1.4.4 Busreset

Der AL1394 behandelt im Gegensatz zu anderen firmeneigenen APIs bereits das Problem der auftretenden Busresets durch einen einfachen konstruktiven Eingriff. Beim AL1394 muss jeder Funktion der Wert des Busresetcounters mit übergeben werden. Damit wird sichergestellt, dass die Applikation alle Busresets erfasst hat. Ist dies nicht der Fall, wird dies am Wert des Busresetcounters erkannt und direkt der Applikation mit einem Fehlerstatus mitgeteilt. Bei einem Vergleich der Funktion `C1394ReadNode` [API1 pp125] der Unibrain FireAPI mit der Funktion `AL1394_ReadNode` [ALC1 pp15] vom AL1394 kann diese Aussage bestätigt werden.

Da außerdem der AL1394 alle Busresets behandelt, wurde die Abarbeitung an einer zentralen Stelle zusammengefasst, womit automatisch gewährleistet wird, dass jede Applikation über einen Busreset informiert werden kann. Bei einer für die Applikation selbst erstellten Lösung würde das Problem entstehen, dass vielleicht eine Applikation über den Busreset informiert wird, eine andere jedoch nicht, da die Information über einen Busreset bereits an einer anderen Stelle abgefangen wurde. Dieses Problem tritt speziell in Multiprozesssystemen auf. Die hier zu entwickelnden IEEE 1394-Sockets können nicht auf ein Single-prozesssystem beschränkt werden, da es nicht sinnvoll ist, die Sockets auf eine einzige Instanz zu beschränken. Daher hat diese Eigenschaft des AL1394 für diese Diplomarbeit eine enorm wichtige Bedeutung.

1.4.5 Acknowledgment

Dem asynchronen Datenaustausch liegt ein Transaction-Layer zugrunde. Dieser versendet Acknowledgment-Pakete, um den Empfang eines Datenpaketes bestätigen zu können. Der AL1394 ermöglicht es, diese Bestätigungen automatisiert oder auch manuell zu versenden. Dies kann beim `MapAddressRange` durch die Option `notificationoptions` eingestellt werden. Leider

besitzt die derzeit verfügbare Version des AL1394 nur die Möglichkeit, die Bestätigungen automatisiert zu versenden [ALC2 pp27-28]. Sobald also das empfangene Datenpaket behandelt wurde, wird ein Acknowledgment-Paket versendet.

1.5 Arbeits- und Entwicklungsumgebung

Da es Ziel dieser Diplomarbeit ist, eine IEEE 1394-Socket-Schnittstelle in Java zu erstellen, ist ein Teil der Arbeitsumgebung die Programmiersprache Java. Als Entwicklungsumgebung wird der `JBuilder 8` von Borland verwendet. Er bietet eine gute Unterstützung für den Programmierer und steht gleichzeitig für den privaten Gebrauch frei zur Verfügung. Um für Java entwickeln zu können, ist ein `Java Development Kit (JDK)` notwendig. Die zu entwickelnde Schnittstelle wird mit der Version JDK 1.3 erstellt. Aus Kompatibilitätsgründen wird jedoch nur auf den Funktionsumfang des JDK 1.1 zurückgegriffen. Dadurch wird sichergestellt, dass die implementierten IEEE 1394-Sockets auch auf älteren Versionen des JDK verwendet werden können.

Ein Nachteil an der Programmiersprache Java – zumindest für diese Arbeit – ist, dass sie keine Möglichkeit bietet, vorhandene Hardware direkt anzusprechen. Meist ist gerade dies ein großer Vorteil von Java, da dadurch Hardwareunabhängigkeit gewährleistet wird. Hier ist dies jedoch hinderlich, da es in Java noch keine Klassen zur Behandlung der IEEE 1394-Hardware gibt. Java stellt jedoch eine Schnittstelle zur Verfügung, die es dem Programmierer erlaubt, Funktionen aus anderen Programmiersprachen wie z. B. C zu verwenden. Diese Funktionen müssen in Bibliotheken als ausführbarer Code zur Verfügung gestellt werden. Java ermöglicht den Zugang zu diesen Bibliotheken mittels des so genannten Java Native Interface (JNI) [TIJ1 pp945-954] [JNI2]. Daher muss ein Teil der Implementierung in einer anderen Sprache als Java entwickelt werden. Dazu wurde C ausgewählt, da diese Sprache einerseits eine sehr breite Unterstützung hat und andererseits Java die Möglichkeit bietet, automatisch die Schnittstellendefinition für C zu generieren. Anfänglich wurde zur Erstellung des C-Codes die Entwicklungsumgebung `Visual C++ 6.0`

von Microsoft verwendet, da sie eine sehr gute Unterstützung für den Programmierer bietet. Augrund einer unvollständigen Implementierung des AL1394 wurde jedoch im Laufe dieser Diplomarbeit das Betriebssystem von Microsoft Windows™ auf Linux gewechselt, weshalb nun für den C-Code die Entwicklungsumgebung `K-Developer` verwendet wird. Auch sie wurde wegen seiner unterstützenden Hilfestellungen für den Programmierer ausgewählt, außerdem handelt es sich dabei um eine Freeware. Es musste zu einem Betriebssystemwechsel kommen, da anfänglich der AL1394 für Microsoft Windows™ entwickelt wurde. Die Entwicklung des AL1394 wurde während dieser Diplomarbeit jedoch auf Linux verlagert, und stellte schließlich nur in diesem Betriebssystem die benötigte Funktionalität zur Verfügung. Aus diesem Grund wurde zuerst die Bibliothek (der C-Code) für Microsoft Windows™ erstellt, und musste schließlich auf Linux umgestellt werden. Diese Veränderung betrifft jedoch nur die Bibliothek, der Java-Teil der IEEE 1394-Sockets ist, aufgrund der Programmiersprache Java, unabhängig vom Betriebssystem. In Java können die IEEE 1394-Sockets derzeit jedoch nur für Linux werden, da für Microsoft Windows™ oder auch andere Betriebssysteme diese Bibliothek nicht erstellt wurde.

1.5.1 Dynamic Link Library (DLL) – shared library

Eine `DLL (Dynamic Link Library)`, auch `shared library` genannt, ist eine Sammlung von kompilierten Funktionen (dabei wird auch von Bibliotheken gesprochen). Der Name DLL hat sich vor allem in Windows™ basierten Systemen etabliert, daher besitzen solche Dateien auch die Endung „.dll“. Unter Linux spricht man eher von shared libraries, es ist aber auch von DLLs die Rede. In Linux besitzt eine zugehörige Datei die Endung „.so“ (shared objects); zudem beginnt der Dateiname mit dem Präfix „lib“.

DLLs werden bei Gebrauch in den Speicher geladen. Ihr Vorteil ist, dass sie meist Funktionen enthalten, die von mehreren Programmen benötigt werden. Aus diesem Grund kann Speicher und Arbeitsaufwand gespart werden – Funktionen, die in einer DLL abgelegt wurden, müssen nur einmal auf einem Computer abgespeichert werden. Dies betrifft sowohl das Abspeichern auf der

Festplatte, als auch das Laden in den Arbeitsspeicher; eine DLL kann nämlich von mehreren Prozessen gleichzeitig genutzt werden. Da somit eine DLL nur einmal im Hauptspeicher gehalten wird, verbessert sich auch das Verhalten bezüglich Auslagerung (Swapping) des Betriebssystems. Sehr vorteilhaft ist auch die leichte Austauschbarkeit einer DLL, da Applikationen, die eine DLL verwenden, nicht neu kompiliert werden müssen, solange das Interface nicht verändert wird [WIN2 pp639-645] [WIN3 pp741-744].

Ein Nachteil einer DLL ist jedoch, dass Applikationen, die DLLs verwenden, nicht in sich abgeschlossen sind; sie sind zwangsläufig von deren Vorhandensein und Korrektheit abhängig [WIN2 pp639-645].

1.5.2 Das Java Native Interface (JNI)

Die Abkürzung JNI steht für Java Native Interface. Dabei handelt es sich um eine Schnittstelle, die es Java ermöglicht, nicht in Java implementierte Funktionen zu verwenden. Dies ist z. B. dann notwendig, wenn Java Plattformabhängige Funktionen nicht zur Verfügung stellt, wie es in dieser Arbeit der Fall ist. Es kann aber auch dann Verwendung finden, wenn Algorithmen bereits in einer anderen Sprache implementiert wurden und somit nur noch eingebunden werden müssen. Ein weiterer Punkt ist, dass zeitkritischer Code in einer hardwarenäheren Sprache implementiert werden sollte, da er performanter ausgeführt werden kann; hardwarenähere Sprachen besitzen aufgrund des geringeren Abstraktionslevels einen geringeren Overhead und bieten somit einen Geschwindigkeitsvorteil [JNI2 pp2].

In der Version 1.0 von Java wurde nicht das JNI, sondern das NMI (Native Method Interface) verwendet. Auch von Netscape und Microsoft wurde je eine Möglichkeit zum Verwenden von Code aus anderen Programmiersprachen entwickelt; das Java Runtime Interface (JRI) von Netscape und das Raw Native Interface (RNI) und das Java/COM Interface von Microsoft. Diese wurden jedoch alle aus Kompatibilitätsproblemen bei der Einbindung in die diversen Java Virtual Machines (JVM) in den

nachfolgenden Versionen von Java (beginnend bei Java 1.1) vom JNI abgelöst [TIJ1 pp945-954].

Damit in Java externe Funktionen aufgerufen werden können, müssen in den Klassen die zugehörigen Methodennamen mittels `native` deklariert werden. Soll eine externe Methode in Java verwendet werden, ist darauf zu achten, dass rechtzeitig die zugehörige Bibliothek in den Speicher geladen wird. Erfolgt dies nicht, kann sie nicht geladen werden oder enthält die geladene Library die angeforderte externe Methode nicht, wird ein `UnsatisfiedLinkError` ausgelöst. Das Laden selbst erfolgt mittels der Methode `loadLibrary` aus der Klasse `System`. Damit die zu ladende Bibliothek gefunden werden kann, muss sie sich entweder in einem Systempfad oder im Pfad der Anwendung befinden. Es ist auch zu berücksichtigen, dass nur der Name der Bibliothek ohne deren Erweiterung (in MS-Windows „.DLL“, oder in Solaris und Linux „.so“) und ohne deren etwaigen Präfix (in Linux „lib“) anzugeben ist. Dies ist deshalb notwendig, um eine Unabhängigkeit von dem verwendeten Betriebssystem zu erreichen (siehe die angegebenen Beispiele). Sollte ein Betriebssystem keine dynamischen Dateien unterstützen, müssen alle externen Methoden (`native`) mit der JVM (Java Virtual Machine) vorgelinkt werden. Damit beinhaltet bereits die JVM diese externen Methoden, und ein Laden einer zugehörigen Library wäre nicht mehr nötig. Aus Kompatibilitätsgründen sollte in Java jedoch der Befehl `loadLibrary` trotzdem implementiert werden, um eine Plattformunabhängigkeit im Java-Code gewährleisten zu können [JNI2 pp1-25].

Von der extern verwendeten Programmiersprache aus gesehen ist die Betrachtung des `JNI` interessanter und auch aufwändiger. Dabei ist unbedingt der Unterschied zwischen der Bezeichnung *externe Methode* und *JNI-Funktion* zu beachten. Man spricht von einer *externen Methode* immer dann, wenn es sich um den Code einer in Java als `native` deklarierten Methode handelt, sprich um die extern implementierte Funktion. Eine *JNI-Funktion* hingegen ist eine Funktion, die das `JNI` dem Programmierer in der externen Programmiersprache zur Verfügung stellt, um mit der JVM kommunizieren zu können. Es sind folgende Funktionalitäten möglich:

- Erstellen, Untersuchen und Bearbeiten von Java-Objekten
- Aufrufen von Java-Methoden
- Abfangen und Auslösen von Exceptions
- Laden und Anfordern von Klasseninformationen
- Untersuchen der Datentypen während der Laufzeit

Der extern verwendeten Sprache stehen viele Möglichkeiten offen, um mit dem Java-Programm aktiv agieren zu können. Es ist unbedingt zu beachten, dass aufgetretene Fehler in den JNI-Funktionen behandelt werden müssen. Ansonsten wird vom `JNI` per Definition nicht garantiert, dass es stabil weiter läuft. Speziell im Fall von Multiprozess-Applikationen ist darauf zu achten, dass alle aufgetretenen Fehler behandelt werden, ansonsten führt ein aufgetretener Fehler in einem Prozess zu einer Instabilität in einem anderen. Als Grund für dieses fehlerhaft scheinende Verhalten des `JNI` wird in der Dokumentation die bessere Performance genannt [JNI2 pp17-20] [TIJ1 pp952-953].

Ein sehr nützliches Werkzeug im Zusammenhang mit dem `JNI` ist das Programm `javah`, das dem JDK (Java Development Kit) beiliegt. Dieses erstellt automatisch aus den gegebenen Java-Klassen die notwendigen Header-Dateien für eine C-Applikation. Diese Datei enthält somit automatisch die Schnittstelle für die externe Programmiersprache anhand des bereits entwickelten Java-Codes. Durch diese Vorgehensweise können Fehler in der Schnittstellendefinition vermieden werden, zudem wird der Entwickler zu einem Top-Down-Design gezwungen.

2 Lösungsansätze

Um die zu entwickelnde Schnittstelle verwenden und testen zu können, muss diese dazu in einer Implementierung verwirklicht werden. Da es sich um ein Socket-Interface für den IEEE 1394-Bus handeln wird, wird diese Implementierung IEEE 1394-Socket genannt. Es stellt sich nun die Frage, wie dieser IEEE 1394-Socket aussehen kann oder auszusehen hat. Daher wird im Folgenden auf einige mögliche Lösungsansätze eingegangen. In weiterer Folge werden einige davon tatsächlich aufgegriffen und auch implementiert.

2.1 Die Adressierung

Bevor die ersten Lösungsansätze diskutiert werden, kann bereits eine Aussage über die Adressierung der IEEE 1394-Sockets getroffen werden. Jeder IEEE 1394-Adapter besitzt eine eindeutige `GUID`. Diese ist im eigentlichen Sinne keine Adresse, da der Adapter damit nicht direkt angesprochen, jedoch eindeutig identifiziert werden kann. Die Adresse eines Adapters ist die `node_ID`, die sich jedoch durch Busresets verändern kann. Daher ist es nicht sinnvoll, diese dynamische Adresse für die Applikation zur Verfügung zu stellen. Aus diesem Grund wird der IEEE 1394-Socket die `GUID` als Adresse akzeptieren und intern die dazu gültige `node_ID` ermitteln, um den zugehörigen Adapter ansprechen zu können. Es gäbe auch die die Möglichkeit, frei definierbare Adressen zu vergeben – man vergleiche das `IPv4`. Dort muss jedem Adapter z. B. vom Administrator eine Adresse (die IP-Adresse) zugeordnet werden. Dies scheint hier nicht sinnvoll zu sein, da jeder Adapter eindeutig durch die `GUID` identifiziert werden kann und somit keine Adressverwaltung auf den Knoten notwendig wird. Betrachtet man z. B. auch die aktuellere Version des `IP` (`IPv6` oder auch `IPng = IP next generation`), kann feststellt werden, dass auch bei dieser die `MAC-Adresse` der Netzwerkkarten (`MAC = Media Access Control`, dabei handelt es sich um eine eindeutige Nummer, die in der Karte gespeichert ist) als Teile der Adresse Verwendung finden [RFC1884 pp7].

Um diese Umrechnung effektiv durchführen zu können, ist es sinnvoll, eine Tabelle mit allen vorhandenen Adaptern am Bus, deren `node_ID` und `GUID` anzulegen. Will eine Applikation nun auf einen Adapter mittels einer `GUID` zugreifen, kann über diese Tabelle die zugehörige `node_ID` ermittelt werden. Es muss jedoch darauf geachtet werden, dass diese Tabelle nach jedem Busreset aktualisiert wird. Daher ist ein Listener zu implementieren, der auf einen auftretenden Busreset wartet. Beim Auftreten eines Busresets ist der Zugriff auf diese Tabelle so lange zu sperren, bis dieser beendet ist und die Tabelle mit den Daten aktualisiert wurde.

2.2 Mögliche Implementierungsansätze

Aufgrund der Komplexität des IEEE 1394-Busses und der in der Einführung aufgelisteten Eigenschaften (*siehe Abschnitt 1.2*) ist klar erkennbar, dass für die gesamte Funktionalität der IEEE 1394-API als Abbildung in einem Socket-Interface keine einheitliche Lösung gefunden werden kann. Im Folgenden werden daher mehrere mögliche Teil-Lösungen diskutiert.

2.2.1 IEEE 1394-API für Java

Als einfachste Lösung könnte ein Bereitstellen der Funktionalität der IEEE 1394-API für Java angeführt werden. Dazu müsste eine Bibliothek erstellt werden, die es mittels JNI ermöglicht, alle Funktionen der vorhandenen API in Java zu verwenden. Ein Vorteil dieser Implementierung wäre, dass der volle Funktionsumfang der IEEE 1394-API zur Verfügung stünde. Diese Implementierung hat aber nichts mit einer Socket-Schnittstelle zu tun und ist somit für diese Diplomarbeit zu verwerfen.

2.2.2 Asynchroner IEEE 1394-Datagram-Socket

Ein Datagram-Socket versendet Datenpakete. Da auch die asynchrone Übertragung beim IEEE 1394-Bus paketweise erfolgt, kann leicht eine Abbildung von der asynchronen Funktionalität des IEEE 1394 auf einen Datagram-Socket

gefunden werden. Bei dieser Implementierung ist ein IEEE 1394-Datagram-Paket zu entwerfen, das die notwendigen Adressinformationen und Daten enthält. Der asynchrone IEEE 1394-Datagram-Socket muss Methoden besitzen, um dieses Paket lesen oder schreiben zu können. Es sollte jedoch beachtet werden, dass der IEEE 1394 mehrere Zugriffsmöglichkeiten bietet: `read`, `write` und `lock`. Daher ist für jede Operation ein unterschiedliches IEEE 1394-Datagram-Paket zu entwickeln. Aufgrund des Pakettyps kann die richtige IEEE 1394-Operation ausgeführt werden. Für diese Implementierung ist kein Protokoll erforderlich, da es sich um eine Abbildung der asynchronen IEEE 1394-API-Funktionen auf die Methoden des asynchronen IEEE 1394-Datagram-Sockets handelt.

2.2.3 Isochroner IEEE 1394-Socket

Mit dem Bewusstsein, dass keine Datensicherheit im isochronen Bereich gegeben ist, ist eine Verwendung dieser Übertragungsart als Socket gut denkbar. Der isochrone Bereich bietet, wie auch ein Socket von sich aus die Fähigkeit, Datenströme auszutauschen. Wird dieser isochrone IEEE 1394-Socket für Multimedia-Anwendungen verwendet, ist die fehlende Datensicherheit nicht unbedingt von Relevanz. Ob er aber z. B. für die Übertragung einer Datei geeignet ist, sollte sich der Programmierer gut überlegen.

Prinzipiell scheint sich diese Abbildung anzubieten, es sollten aber noch einige Punkte behandelt werden. Eine Socket stellt eine Verbindung zwischen genau zwei Prozessen dar; Daten auf einem isochronen Kanal können aber von mehreren Adaptern gelesen werden. Durch Einfügen eines Headers in den Datenstrom könnte dieses Problem umgangen werden, da dieser eine Identifizierung der Datenpakete ermöglichen würde. Es stellt sich aber die Frage, ob es bei einem isochronen IEEE 1394-Socket nicht Sinn machen würde, mehreren Prozessen den Zugriff auf einen Kanal zu gestatten. Ein Fernsehsender könnte so z. B. über einen Kanal „ausgestrahlt“ werden und gleichzeitig vom Fernseher und vom Videorecorder empfangen werden. Zur Verwaltung der isochronen Ressourcen, wie z. B. Kanalnummer oder Bandbreite, ist es sinnvoll, sich an die Spezifikation des Standards IEC 61883 [IEC1] zu halten.

Da über einen Socket Daten in beiden Richtungen ausgetauscht werden können, beim isochronen Datenaustausch am IEEE 1394-Bus jedoch nur in einer Richtung, muss abschließend das Problem des Rückkanals analysiert werden. Wie soll also der Rückkanal realisiert werden? Als Lösungsvorschläge kommen ein isochroner, asynchroner oder kein Rückkanal in Frage. Der isochrone Rückkanal scheint nicht praktikabel zu sein, da z. B. ein Fernsehbild nur in einer Richtung ausgestrahlt wird. Daher ist es sinnlos, einen isochronen Rückkanal zu belegen, der nicht verwendet wird. Dies würde einer Bandbreitenverschwendung gleich kommen. Wird kein Rückkanal implementiert, könnte theoretisch auch ein isochroner Datenaustausch mit einer isochronen IEEE 1394-Applikation erfolgen. Bei der Verwendung eines asynchronen Rückkanals wäre ein Datenaustausch in beiden Richtungen und ohne Bandbreitenverschwendung über das Socket-Interface möglich. Da der isochrone IEEE 1394-Socket aufgrund der Echtzeitfähigkeit bereits spezialisiert ist, sollte eher zugunsten der Eigenschaft des Adapters und nicht der des Sockets entschieden werden. Wird nun auch noch darauf verzichtet, einen Header jedem Datenpaket voranzusetzen, kann tatsächlich auch eine Kommunikation mit einer IEEE 1394-Applikation erfolgen. Aufgrund der Tatsache, dass diese Form des Sockets einen Spezialfall darstellt, ist diese Lösung zu bevorzugen.

Bei der Implementierung des isochronen IEEE 1394-Sockets sollte somit folgendes bedacht werden: Es ist der Standard IEC 61883 [IEC1] für das Management einzuhalten. Weiters besitzt dieser IEEE 1394-Socket-Typ nur einen Kanal in einer Richtung. Daher kann von einem IEEE 1394-Socket-Objekt, das mittels `connect` erstellt wurde, nur gelesen werden; auf eines das von `accept` retourniert wurde, nur geschrieben werden. Damit von einem Kanal mehrere Adapter Daten lesen können, muss es auch möglich sein, auf diesen Kanal mehrfach ein `connect` durchzuführen; ein `accept` darf jedoch nur einmal erfolgen, da nur ein Adapter auf einen Kanal schreiben darf. Zusätzlich ist anzumerken, dass für die Adressierung die Kanalnummer mit der `bus_ID` und nicht die `GUID` verwendet werden muss; es wird schließlich nicht mit einem einzelnen Adapter, sondern über gegebene Kanäle auf den einzelnen Bussen kommuniziert.

2.2.4 Asynchroner IEEE 1394-Socket

In den vorangegangenen Absätzen wurde die Abbildung der isochronen IEEE 1394-API auf einen Socket und der asynchronen IEEE 1394-API auf einen Datagram-Socket erfolgreich behandelt. Es stellt sich nun die Frage, ob es nicht möglich ist, den asynchronen Bereich auch für einen Socket zu verwenden. Das Hauptaugenmerk dieser Diplomarbeit orientiert sich an dieser Frage, da genau dieser Teil des IEEE 1394-Sockets tatsächlich implementiert wurde. Da der asynchrone Datenaustausch paketweise erfolgt, und zudem keine eindeutige Verbindung zwischen zwei Prozessen ermöglicht, kann die Implementierung nicht durch ein einfaches Abbilden erfolgen. Wird jedoch ein zusätzliches Protokoll implementiert, kann durch dieses eine Punkt-zu-Punkt-Verbindung und das Übertragen von Datenströmen ermöglicht werden. Vergleichend mit `TCP/IP` kann festgestellt werden, dass das unterlagerte `IP` [RFC791] paketorientiert ist und das darauf aufbauende `TCP` [RFC793] erst die Datenstromfähigkeit und Punkt-zu-Punkt-Verbindung zur Verfügung stellt. Daher könnte sich die Implementierung des asynchronen IEEE 1394-Sockets, der den asynchronen Modus nutzen soll, an diesem Protokoll orientieren, und eine ähnliche Lösung zum `TCP` anbieten. Bei `TCP/IP` handelt es sich um ein bereits gut erprobtes und funktionierendes Protokoll; es wird als Basisprotokoll für das Internet verwendet.

2.2.5 Isochroner IEEE 1394-Datagram-Socket

Abschließend wird noch die Abbildung der isochronen IEEE 1394-API auf einen Datagram-Socket behandelt. Es soll also versucht werden, einzelne Pakete isochron zu übertragen. Es kann jedoch nicht sinnvoll sein, einen echtzeit-fähigen Kanal zu öffnen, ein Paket zu versenden und diesen Kanal dann wieder zu schließen. Dieser Kanal ist dafür ausgelegt, Datenströme, nicht einzelne Pakete, in Echtzeit zu übertragen. Daher führt sich diese Möglichkeit der Realisierung ad absurdum, zudem der IEEE 1394-Bus für die paketorientierte Übertragung bereits das sehr gutes Konzept der asynchronen Übertragung zur Verfügung stellt.

2.2.6 Gewählte Lösung

In der Aufgabenstellung wurde gefordert, dass der Datenstrom der mittels der zu implementierenden Sockets übertragen werden soll, nicht verändert werden darf. Wenn jedoch verbindungsorientierte, gesicherte Datenübertragung auf den asynchronen Übertragungsmodus des IEEE 1394-Busses abgebildet werden soll, ist die Definition und Implementierung eines zusätzlichen Protokolls notwendig [OS1 pp544-550]. Dieses baut auf dem asynchronen Übertragungsmodus auf und liefert die notwendigen zusätzlichen Eigenschaften, bedingt jedoch die Veränderung des real zu übertragenden Datenstroms. Genau das widerspricht der Aufgabenstellung, da dadurch keine Kommunikation mit einer IEEE 1394-Applikation möglich ist. Warum aber ist es notwendig, ein zusätzliches Protokoll zu implementieren? Es muss mittels einer Socket-Verbindung möglich sein, zu einem anderen Adapter mehrere Verbindungen parallel zu etablieren. Beim IEEE 1394-Adapter könnte das mittels unterschiedlichen Speicherbereichen realisiert werden; jedem asynchronen IEEE 1394-Socket wird ein anderer Speicherbereich zugeordnet. Wie soll mit dieser Methode jedoch erkannt werden, ob ein Datenpaket auf den korrekten Speicherbereich geschrieben wurde? Das Paket selbst enthält ja keinerlei Information darüber. Weiters ist es bei einem Server-Socket möglich, auf einem Port mehrere Verbindungen parallel zu erstellen, die jedoch unterschiedlichen Socket-Verbindungen zugeordnet sind. Die Unterscheidung zwischen den einzelnen Socket-Verbindungen kann beim herkömmlichen Socket anhand der Adapteradressen und der zugehörigen Ports der Clients durchgeführt werden. Dazu betrachte man als einfaches Beispiel einen HTTP-Server (Hyper Text Transfer Protocol). Dieser ermöglicht es unter anderem, Internetseiten von einem Host im Internet zu downloaden. Dabei ist es möglich, nicht nur eine Seite, sondern viele Seiten parallel zu übertragen. Daher kann die Eigenschaft, dass auf einem Port mehrere Verbindungen geöffnet werden können, nicht außer Acht gelassen werden. Darf beim asynchronen IEEE 1394-Socket kein zusätzliches Protokoll verwendet werden, können Datenpakete, die auf einem Speicherbereich empfangen werden, nicht eindeutig einer Socket-Verbindung zugeordnet werden, da das IEEE 1394-Bussystem keine Ports kennt und somit auch keine Informationen darüber austauscht, und auch das Datenpakte per

Definition diese Information nicht enthält. Als Lösung käme in Frage, dass nur eine Verbindung auf einem Speicherbereich erlaubt werden dürfte. Dies widerspricht jedoch der soeben getätigten Aussage, dass Server-Sockets mehrere parallele Verbindungen auf einem Port zulassen. Aufgrund der eben genannten Probleme musste festgestellt werden, dass die Forderung, dass der Datenstrom nicht verändert werden darf, nicht sinnvoll ist. Sie wurde daher für die Implementierung des asynchronen IEEE 1394-Sockets fallen gelassen. Diese Einschränkung betrifft aber nicht den isochronen IEEE 1394-Socket und den asynchronen IEEE 1394-Datgram-Socket.

2.3 Alternativen für den Datenaustausch über IEEE 1394

Die Socket-Schnittstelle wurde speziell für den Datenaustausch zwischen unterschiedlichen Prozessen auf unterschiedlichen Systemen entwickelt. Es gibt aber auch andere Möglichkeiten für Datentransfer. An dieser Stelle soll auf einige davon eingegangen werden.

2.3.1 Direktes Ansprechen der verwendeten Hardware

Um Daten zwischen Systemen austauschen zu können, muss dazu eine Hardware verwendet werden. Diese transportiert die Information von einem System zum anderen. Damit Daten ausgetauscht werden, muss somit die dazu notwendige Hardware angesprochen werden. Da die verwendete Hardware im Allgemeinen jedoch sehr komplex ist und daher das Ansprechen sehr aufwändig ist, wird von den Herstellern ein eigenes Modul entwickelt, welches diese Aufgabe übernimmt. Dabei handelt es sich um die Betriebssystem-Treiber (*siehe Abschnitt 2.3.2*). Weiters ist es in modernen Betriebssystemen nicht mehr möglich, Hardware direkt anzusprechen, da die vorhandenen Ressourcen vom Betriebssystem verwaltet werden.

2.3.2 Betriebssystem-Treiber

Damit ein Betriebssystem Hardware verwenden und auch verwalten kann, ist es notwendig, Treiber zu installieren. Diese ermöglichen den Zugriff auf die

gewünschte Hardware für das zugehörige Betriebssystem. In Linux muss der benötigte Treiber dem Linux-Kernel zur Verfügung gestellt werden. Er kann entweder in den Kernel kompiliert werden, oder auch bei Bedarf als Modul geladen werden. Der zugehörige Modulname lautet für den IEEE 1394-Bus `ieee1394`. Ein Treiber stellt jedoch nur eine sehr einfache Form der Kommunikation mit der Hardware zur Verfügung. Außerdem ist jeder Treiber abhängig von der verwendeten Hardware. Daher gibt es in Linux das Modul `ohci1394`, welches den eigentlichen Zugriff auf einen OHCI-Kompatiblen IEEE 1394-Adapter ermöglicht. Diese Module sind bereits im Linux-Kernel der Version 2.4.19 enthalten und finden auch in dieser Diplomarbeit Verwendung.

Bei anderen Betriebssystemen wie z. B. Microsoft Windows™ ist es ähnlich. Bei der derzeit aktuellsten Version, Windows XP, sind bereits sehr viele Treiber inkludiert. Diese können bei Bedarf installiert werden. Windows XP besitzt die Möglichkeit, vorhandene Hardware selbst zu erkennen und die notwendigen Betriebssystem-Treiber zu installieren. Das Installieren erfolgt dabei mehr oder weniger automatisch; es ist im Normalfall nur mehr die Installations-CD nach vorheriger Aufforderung vom Betriebssystem einzulegen. Es wird auch Hardware, deren Treiber nicht inkludiert sind, erkannt. In einem solchen Fall muss dafür gesorgt werden, dass die notwendigen Treiber zur Verfügung stehen. Diese können entweder mit der Hardware mitgeliefert werden (z. B. auf Diskette oder CD), oder aber auch über das Internet downgeloadet werden. Im Gegensatz zum Betriebssystem Linux können bei Windows™-Betriebssystemen die Treiber nicht in den Kernel hinein kompiliert werden, da es nicht in Form von Quelldateien zur Verfügung steht.

2.3.3 Herstellerspezifische API

Dabei handelt es sich um ein Modul, welches der Hersteller zur Verfügung stellt, um mit seiner Hardware kommunizieren zu können. Für den IEEE 1394-Bus gibt es z. B. von dem Hersteller Unibrain eine eigene API, FireAPI genannt [API1]. Diese API baut auf dem Betriebssystemtreiber auf und stellt eine etwas komfortablere Schnittstelle für die Kommunikation zur Verfügung. Der Nachteil an einer speziellen API ist jedoch, dass diese nicht genormt oder standardisiert

ist und auch nur für Adapter dieses einen Herstellers verwendet werden kann. Wird die Hardware ausgetauscht, muss daher auch die Software abgeändert werden. Der Vorteil liegt jedoch darin, dass der Hersteller seine API speziell an seinen Adapter anpassen kann. Dies kann sich z. B. in einer guten Performance oder aber auch in einem sehr umfangreichen Funktionsumfang niederschlagen.

2.3.4 Andere Interfaces

Es besteht natürlich auch die Möglichkeit, ein Interface unabhängig von einem Hersteller zu definieren. In diesem Fall kann als Beispiel der AL1394 genannt werden [ALC1]. Dabei handelt es sich um ein Interface, welches vom Institut für Computertechnik an der TU-Wien entwickelt wurde. Es ermöglicht die Verwendung eines beliebigen IEEE 1394-Adapters mit ein und derselben Schnittstelle und demselben Funktionsumfang. Dadurch werden einerseits eine Hardwareunabhängigkeit und andererseits ein definierter Funktionsumfang garantiert, selbst unabhängig vom verwendeten Betriebssystem. Zusätzlich steht dieses Produkt auch kostenfrei zur Verfügung.

2.3.5 IPover1394

Dieses Protokoll ist in [RFC2734] definiert und ermöglicht eine IP-Verbindung zwischen mehreren IEEE 1394-Adaptern. Dieses Protokoll baut IP-Verbindungen über den IEEE 1394-Bus auf, genau so, als ob sie über ein LAN (Local Area Network) oder über ein Modem ins Internet aufgebaut werden. Ist dieses Protokoll, in welcher Form auch immer, in einem Betriebssystem implementiert, können mittels des darüber liegenden TCP Socket-Verbindungen erstellt werden, oder mittels dem UDP Datagram-Pakete ausgetauscht werden. Das Protokoll `IPover1394` stellt jedoch nur eine IP-Verbindung über den IEEE 1394-Bus her; in dieser Diplomarbeit werden im Gegensatz dazu die Funktionen des IEEE 1394 in der Socket-Schnittstelle abgebildet. Im Speziellen kommt es zu einer Abbildung der gesamten Funktionalität in drei Sockets; der isochrone IEEE 1394-Socket, der asynchrone IEEE 1394-Socket und der asynchrone IEEE 1394-Datagram-Socket. Beim isochronen IEEE 1394-Socket

und beim asynchronen IEEE 1394-Datagram-Socket handelt es sich tatsächlich um eine Abbildung. Aber auch der asynchrone IEEE 1394-Socket unterscheidet sich grundlegend von IPover1394. Es wird direkt eine Socket-Verbindung zwischen unterschiedlichen Prozessen aufgebaut; es muss nicht erst das überlagerte TCP dafür verwendet werden.

Ein Nachteil des IPover1394 gegenüber den IEEE 1394-Sockets liegt darin, dass IPover1394 den Standard IEEE 1394a, vor allem dessen asynchronen Datenstrom, benötigt [RFC2734 pp6]. Weiters besteht ein Unterschied in der Adressierung; die IEEE 1394-Sockets verwenden als Basisadresse die GUID, beim IPover1394 wird jedoch eine IP-Adresse verwendet, weswegen auch ein DHCP-Server (Dynamic Host Configuration Protocol) notwendig wird [RFC2734] [RFC2855].

3 Schnittstellen und Protokolle

Die wichtigste Schnittstelle und Ziel dieser Arbeit ist jene, die von Java aus genutzt werden soll, nämlich die IEEE 1394-Sockets in Java. Als unterste Schnittstelle wird die IEEE 1394-API (in diesem Fall der AL1394) genutzt, da sie den Zugriff auf den IEEE 1394-Adapter ermöglicht. Dazwischen wurden der Übersichtlichkeit, Modularität und Testbarkeit wegen weitere Schnittstellen eingeführt. Weiters müssen auch Protokolle definiert werden um einen fehlerfreien Datenaustausch garantieren zu können. In *Abbildung 3-1* sind alle Module der IEEE 1394-Sockets und deren Abhängigkeiten dargestellt. In den folgenden Abschnitten werden die einzelnen Schnittstellen und Komponenten im Detail definiert und beschrieben.

Grundsätzlich wurde versucht, die IEEE 1394-Sockets mittels Top-Down-Design zu entwickeln. Dabei musste jedoch immer bedacht werden, dass als unterste Schnittstelle der AL1394 zu verwenden ist. Dies hat die Unterteilung der IEEE 1394-Sockets in drei Teile stark beeinflusst (*siehe Abschnitt 2.2*). Als erstes wurde die Schnittstelle für Java entworfen, anschließend jene der IEEE 1394-Socket-API. Nun konnten die IEEE 1394-Sockets in Java implementiert und getestet werden. Zu diesem Zeitpunkt stellte die IEEE 1394-Socket-API nur Dummy-Funktionen zur Verfügung. Um eine etwas abstraktere Schnittstelle als den AL1394 verwenden zu können, wurde nun das Kommunikationsmodul implementiert. Diese Schnittstelle wurde durch den Wechsel des Betriebssystems und auch der verwendeten IEEE 1394-API während der Entwicklung der IEEE 1394-Sockets notwendig. Zuletzt wurde der IEEE 1394-Socket-Core entworfen und implementiert. Die Tools wurden nebenbei entwickelt, da es sich dabei um eine Sammlung häufig genutzter Funktionen handelt. Auch die Funktionen des Loggings wurden nebenbei entwickelt, da sich deren Funktionalität immer nach den gerade benötigten Funktionen richtete.

Bei der Betrachtung des Blockdiagramms (*siehe Abbildung 3-1*) kann festgestellt werden, dass die Funktionalität der IEEE 1394-Sockets und des IEEE 1394-Datagram-Sockets für Java bereits in C voll zur Verfügung gestellt wird;

sie kann bereits in dieser Ebene von Applikationen verwendet werden. Die IEEE 1394-Sockets in Java bauen nun auf den darunter liegenden auf. Der Vorteil dabei liegt darin, dass die IEEE 1394-Sockets und der IEEE 1394-Datagram-Socket nicht nur in Java verwendet werden können. Weiters kommt es dadurch zu keiner Vermischung von verschiedenen Technologien; Java-Code und C-Code sind vollständig und klar voneinander getrennt. Wäre dies nicht der Fall, könnten Probleme bei der Synchronisierung, Fehlerbehandlung und Parameterübergabe auftreten [JNI1].

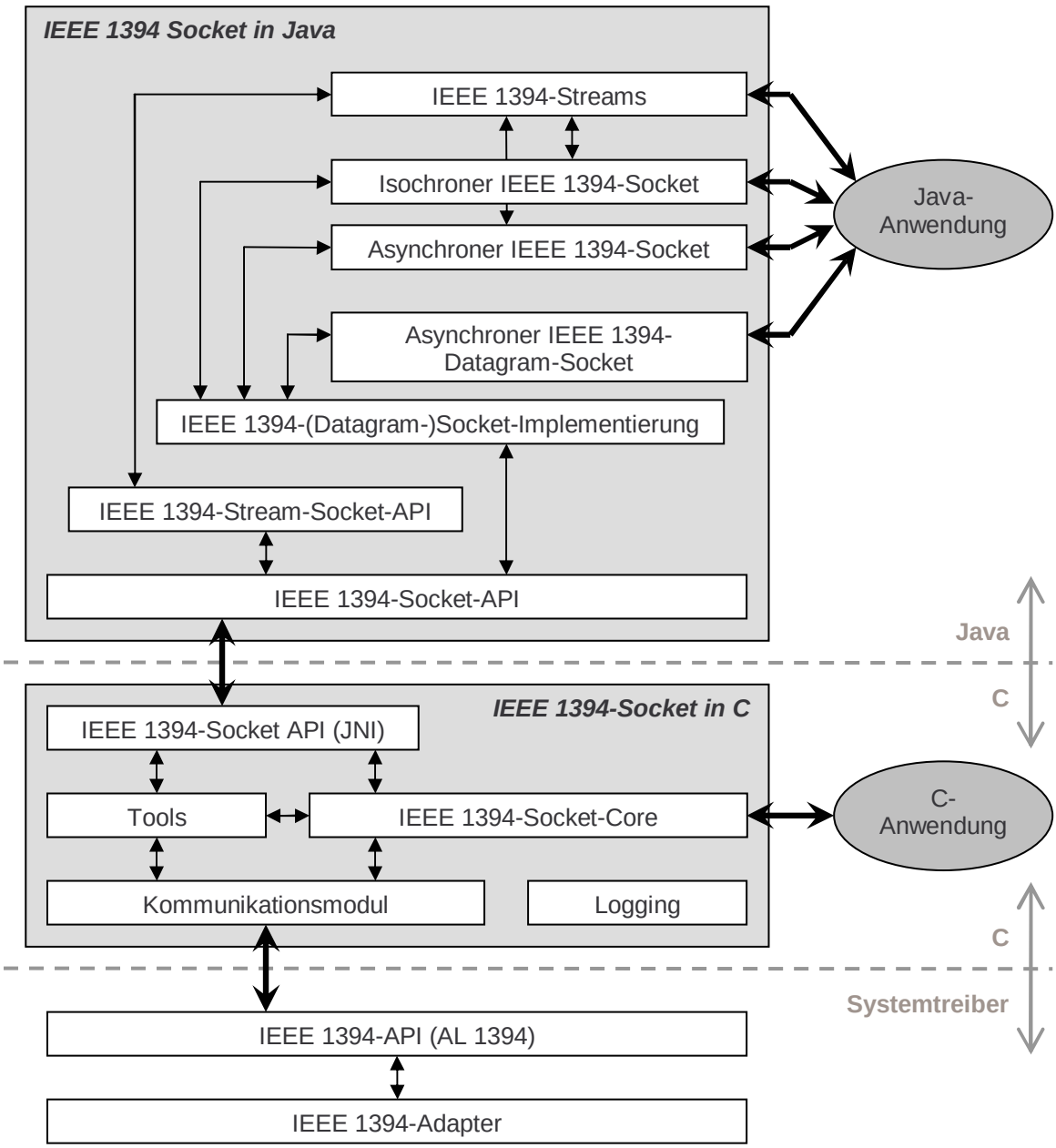

Abbildung 3-1 IEEE 1394-Socket-Schnittstellen

3.1 Adressierung der Sockets

Die Implementierung der IEEE 1394-Sockets teilt sich auf drei unterschiedliche Teile auf. Daher muss für jeden Teil die Adressierung behandelt werden.

3.1.1 Asynchrone IEEE 1394-Socket-Adresse

Der asynchrone IEEE 1394-Socket orientiert sich an dem Protokoll `TCP/IP`. Daher scheint es sinnvoll, so wie bei diesem Protokoll einen Port einzuführen, um mit mehreren Prozessen auf einem Adapter kommunizieren zu können. Die Adapter werden jedoch nicht wie beim `TCP/IP` mit einer IP-Adresse, sondern mittels der `GUID` angesprochen. Ein Port besitzt eine Breite von 16 Bit, die `GUID` eine Breite von 64 Bit.

3.1.2 Isochrone IEEE 1394-Socket-Adresse

Beim isochronen IEEE 1394-Socket wurde festgestellt, dass der Datenaustausch über einen isochronen Kanal erfolgt. Da diese Kanäle über den Standard IEC 61883 gesteuert werden, kann nur die Kanalnummer und die zugehörige `bus_ID` als Adresse eingesetzt werden. Eine Kanalnummer hat eine Breite von 6 Bit, und die `bus_ID` 10 Bit.

3.1.3 Asynchrone IEEE 1394-Datagram-Socket-Adresse

Der asynchrone IEEE 1394-Datagram-Socket muss in der Adresse alle Informationen zur Verfügung stellen können, die für eine asynchrone Operation am IEEE 1394-Bus erforderlich sind. Dabei handelt es sich um die `GUID` des gewünschten Adapters und um dessen Adressraum, der manipuliert werden soll. Der Adressraum setzt sich aus seiner Startadresse und Speicherlänge zusammen. Die `GUID` hat eine Länge von 64 Bit, die Startadresse und die Speicherlänge besitzen je eine Breite von 48 Bit.

3.1.4 Adressstruktur für die IEEE 1394-Sockets

Beim Protokoll `TCP` wird in C für die Adressangabe eine Struktur mit dem Namen `sockaddr` verwendet [CNW1 pp355-356]. Diese wird auch bei den IEEE 1394-Sockets eingesetzt. Dabei handelt es sich um eine Bytearray (*siehe Abbildung 3-2*) in der das erste Byte angibt, wie viele Bytes von dem Array mit der Adresse belegt sind (die beiden ersten Bytes werden nicht mitgezählt). Das zweite Byte kennzeichnet den Typ des zu verwendenden Socket. Die drei möglichen IEEE 1394-Socket-Typen wurden als Konstanten definiert und besitzen folgende Werte:

isochroner IEEE 1394-Socket	1
asynchroner IEEE 1394-Socket	2
asynchroner IEEE 1394-Datagram-Socket	3

Tabelle 3-1 IEEE 1394-Socket-Typen

Für den asynchronen IEEE 1394-Socket folgen nun zuerst 8 Bytes für die `GUID` und anschließend 2 Bytes für den Port. Beim isochronen IEEE 1394-Socket folgen 6 Bit (c) mit der Kanalnummer und weitere 10 Bit (`b_id`) mit der `bus_ID`. Der asynchrone IEEE 1394-Datagram-Socket besitzt zuerst wieder 8 Bytes für die `GUID`, 6 Bytes für die Adresse am Adapter und 6 Bytes für den Adressraum (*siehe Abbildung 3-2*).

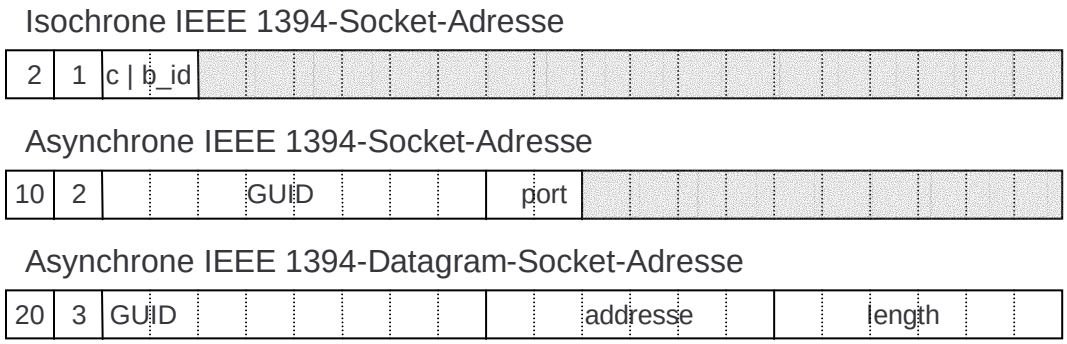

Abbildung 3-2 IEEE 1394-Addressstruktur

3.2 Externe Schnittstellen

Extern sind drei Schnittstellen zugänglich. Dabei handelt es sich um die IEEE 1394-Sockets in Java, die IEEE 1394-Sockets in C und die Zugriffsmöglichkeit auf die IEEE 1394-Sockets in C ausgehend von Java. Natürlich ist auch der AL1394 extern zugänglich, ist aber nicht Teil dieser Diplomarbeit.

3.2.1 IEEE 1394-Socket in Java

Diese Schnittstelle wird durch die Aufgabenstellung vorgegeben. Da es sich bei der Entwicklung der IEEE 1394-Sockets um eine Entwicklung des ICT der TU-Wien handelt, werden alle Klassen im Paket `at.ac.tuwien.ict` abgelegt. Diese Namensvergabe entspricht einer gängigen Konvention [JAVA2 p175]. Klassen für die IEEE 1394-Sockets selbst (IEEE 1394-Socket und IEEE 1394-Datagram-Socket) werden im Paket `at.ac.tuwien.ict.net` und Klassen für den Datenaustausch (Streams) im Paket `at.ac.tuwien.ict.io` abgelegt. Die Unterteilung in `io` und `net` wurde dem ursprünglichen Socket von Java entnommen. Klassen, die dem Testen der IEEE 1394-Sockets dienen, befinden sich im Paket `at.ac.tuwien.ict.testing`.

In den Java-Klassen wird prinzipiell nicht unterschieden, ob es sich um einen asynchronen oder isochronen Socket handelt, jedoch ob es sich um eine Socket-Klasse oder Datagram-Socket-Klasse handelt. Denn zwischen dem isochronen und asynchronen IEEE 1394-Socket besteht vom Interface her kein Unterschied (es handelt sich um einen Socket). Nur bei der Erstellung des Sockets muss die Typenbezeichnung des gewünschten IEEE 1394-Sockets angegeben werden. In weiterer Folge weiß die IEEE 1394-Socket-Instanz, wie die Daten zu versenden sind, das Interface bleibt jedoch gleich. Da sich der Datagram-Socket grundlegend von einem Socket unterscheidet, besitzt der asynchrone IEEE 1394-Datagram-Socket ein eigenes Interface; er verwendet außerdem anstelle von Streams Datenpakete für den Datenaustausch.

IEEE 1394-Socket

Da es sich beim isochronen IEEE 1394-Socket und beim asynchronen IEEE 1394-Socket um einen Socket handelt, sollten die zugehörigen Klassen in Java von der Klasse `java.net.Socket` abgeleitet werden. Ein Problem bei diesem Vorhaben stellt jedoch die Klasse `java.net.InetAddress` dar. Sie enthält die Adressangaben für den Java-Socket. Da alle Arten des IEEE 1394-Sockets eine unterschiedliche Adresse verwenden (*siehe Abschnitt 3.1*) und diese sich massiv von einer IP-Adresse unterscheiden, muss eine eigene Implementierung für die Adresse erstellt werden. Dies ist auch deshalb notwendig, da diese unterschiedlichen Adressangaben nicht kompatibel sind. Man betrachte nur einmal den Vergleich zwischen `GUID` und IP-Adresse; eine `GUID` besitzt 64 Bit und eine IP-Adresse 32 Bit. Weitere Unterschiede können in *Abschnitt 3.1* nachgelesen werden. Und genau darin liegt das Problem, denn die Klasse `InetAddress` ist `final` definiert; daher kann von dieser Klasse keine Ableitung vorgenommen werden. Genauer gesagt liegt das Problem bei der Klasse `java.net.SocketImpl`; sie stellt die eigentliche Implementierung des Sockets dar. In Java wäre es grundsätzlich so gedacht, dass für jede Art von Socket eine eigene Klasse von der Klasse `SocketImpl` abgeleitet wird. Diese wird anschließend von der Klasse `Socket` verwendet. Damit bleibt die Socket-Schnittstelle gleich, nur die Implementierung wird ausgetauscht. Aus diesem Grund ist die Klasse `SocketImpl` `abstract` definiert und enthält auch mehrere abstrakte Methoden wie z. B. `listen`, `accept`, `connect` und `available`. Diese Methoden muss jede abgeleitete Klasse von `SocketImpl` enthalten. Die Methode `connect` benötigt nun z. B. eine Instanz der Klasse `InetAddress`. Und genau deswegen ist es nicht möglich, die zu implementierenden IEEE 1394-Sockets in Java vom Java-Socket abzuleiten. Denn eine Implementierung für die IEEE 1394-Sockets benötigt eine eigene Adressangabe, muss aber an dieser Stelle eine Instanz der Klasse `Inet-Address` übergeben. Da von dieser jedoch keine eigene Klasse abgeleitet werden kann, dies aber erforderlich wäre, können die IEEE 1394-Sockets in Java nicht vom Java-Socket abgeleitet werden. Aber auch an der Klasse `java.net.Socket` kann bereits erkannt werden, dass es nicht sinnvoll ist, eine Ableitung davon vorzunehmen; der Haupt-Konstruktor (er wird von allen

anderen Konstruktoren aufgerufen) erfordert zwei Instanzen der Klasse `InetAddress` (lokale Adresse und Adresse des Verbindungspartners). Aber auch wenn z. B. die Adresse des Verbindungspartners mit der Methode `getInetAddress` abgefragt wird, erhält man eine Instanz der Klasse `InetAddress` als Antwort. Vor allem an der Methode `getInetAddress` ist ersichtlich, dass eine Ableitung nicht möglich ist, da es sich bei dieser um eine essenzielle Methode der Klasse `Socket` handelt. Würde diese Methode anders implementiert werden, müssten auch bei einer Ableitung von der Klasse `Socket` alle Applikationen die die IEEE 1394-Sockets verwenden umgeschrieben werden, da sich das Interface verändert hat. Aus den genannten Gründen werden deshalb Kopien der Klassendefinitionen verwendet. Der nennenswerte Unterschied ist, dass die eigens entwickelte Klasse `IEEE1394-Address` anstelle der Klasse `InetAddress` zum Einsatz kommt. Der Nachteil der daraus entsteht ist, dass bestehende Applikationen umgeschrieben werden müssen, wenn sie einen der IEEE 1394-Sockets in Java verwenden wollen. Könnte die Ableitung erfolgen, müsste nur eine andere Instanz verwendet werden.

In *Abbildung 3-3* wurde das Klassendiagramm des asynchronen und isochronen IEEE 1394-Sockets dargestellt. Es wurde nach dem `UML`-Standard (`Unified Modeling Language`) gezeichnet [UML1]. Eine Verbindung von einer Klasse A zur Klasse B mit einem geschlossenen Pfeil bedeutet, dass die Klasse A von B abgeleitet ist (die Klasse `IEEE1394InputStream` ist z. B. von der Klasse `IEEE1394AsyncInputStream` abgeleitet). Sind zwei Klassen mit einer Linie mit einem Kreis verbunden, bedeutet dies, dass eine Instanz der Klasse B in der Klasse A enthalten ist (die Klasse `IEEE1394Socket` enthält eine Instanz der Klasse `IEEE1394SocketImpl`). In der *Abbildung 3-4* ist weiters eine Verbindung mit einem geöffneten Pfeil zu finden. Diese bedeutet, dass in der Klasse B eine Instanz der Klasse A verwendet wird (die Klasse `IEEE1394AsyncDatagramSocket` verwendet Instanzen der Klasse `IEEE-1394AsyncDatagramPacket`). Der besseren Darstellung wegen wurden die Klassen unterschiedlich eingefärbt. Bei hellgrau hinterlegten Klassen handelt es sich um die eigentliche IEEE 1394-Socket-Schnittstelle. Weiße Klassen

beinhalten die Funktionalität, können jedoch nicht frei von außen verwendet werden. Abschließend wurden Klassen, die abstrakt definiert sind, dunkelgrau hinterlegt.

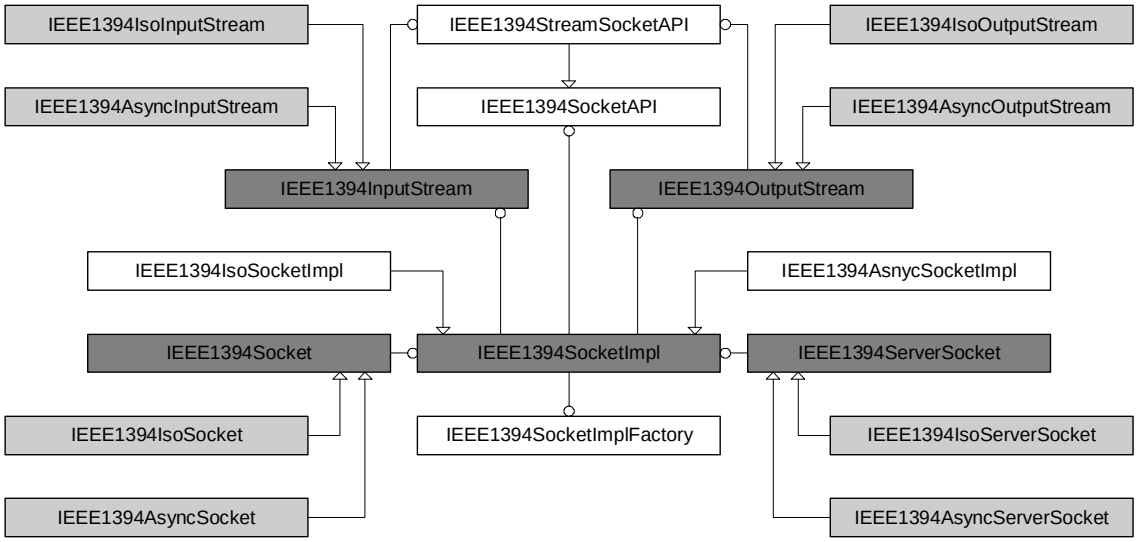

Abbildung 3-3 Klassendiagramm IEEE 1394-Socket

Bei den IEEE 1394-Sockets können nicht nur die Klassendefinitionen dem Java-Socket entnommen werden, sondern auch die gesamte Funktionsweise. Diese Vorgehensweise scheint sinnvoll, da diese Implementierung gut funktioniert und aufgrund der Aufgabenstellung ja auch gefordert ist. So gilt auch hier als zentrales Element die Klasse `IEEE1394SocketImpl`. Sie enthält die gesamte Funktionalität, denn die Klasse `SocketImpl` stellt eine Abbildung des Sockets vom Betriebssystem in Java dar. Dabei handelt es sich z. B. um Methoden wie `connect`, `bind`, `listen`, `accept` und `close`. An dieser Stelle wird darauf verzichtet, auf die Funktionalität dieser Methoden einzugehen, da sie bereits beim Socket in C beschrieben wurden (*siehe Abschnitt 1.3.4*). Weiters kann die Funktionsweise dieser Klasse auch in der Beschreibung des Java-Sockets nachgelesen werden. Um bei der IEEE 1394-Socket-Implementierung besser zwischen den unterschiedlichen IEEE 1394-Socket-Typen unterscheiden zu können, existieren die abgeleiteten Klassen `IEEE1394IsoSocketImpl` und `IEEE1394AsyncSocketImpl`. Diese enthalten keinen weiteren Code, ausgenommen den Konstruktoren, und dienen nur der besseren Unterscheidbarkeit für den Programmierer.

Damit die Klasse `IEEE1394SocketImpl` mit den IEEE 1394-Sockets in C kommunizieren kann, wurde die Klasse `IEEE1394SocketAPI` erstellt. Diese sorgt dafür, dass alle Parameter für den externen Funktionsaufruf korrekt zur Verfügung gestellt werden, und nach der Rückkehr dieser Funktion wieder verwendet werden können. Daher wird so weit wie möglich eine Kontrolle der Rückgabewerte vorgenommen und im Fehlerfall automatisch eine entsprechende Exception ausgelöst.

In den Klassen `IEEE1394Socket` und `IEEE1394ServerSocket` ist die Implementierung des Client-Sockets und des Server-Sockets enthalten. Dabei handelt es sich ebenfalls um Kopien vom Java-Socket. Diese Klassen übernehmen das Verbindungsmanagement, wie z. B. Aufbau und Beendigung einer Verbindung. Um dies bewerkstelligen zu können, nutzen sie die Funktionen der Socket-Implementierung von der Klasse `IEEE1394SocketImpl`. So wie auch bei der Socket-Implementierung gibt es die klare Trennung zwischen den IEEE 1394-Socket-Typen. Daher existiert je eine abgeleitete Klasse für den asynchronen und den isochronen IEEE 1394-Socket, die lediglich unterschiedliche Konstruktoren beinhalten. Unterschiedliche Konstruktoren sind notwendig, da z. B. für einen asynchronen IEEE 1394-Socket eine `GUID` und ein Port als Adresse erforderlich sind, beim isochronen IEEE 1394-Socket jedoch die Kanalnummer und die `bus_id`.

Streams für die IEEE 1394-Sockets

Da Datenströme in Java mittels Streams ausgetauscht werden, muss ein `IEEE1394InputStream` und `IEEE1394OutputStream` entwickelt werden. In *Abbildung 3-3* wurden neben den IEEE 1394-Socket-Klassen auch die Stream-Klassen und deren Abhängigkeit dargestellt. Eine Instanz der Klasse `Input-Stream` ermöglicht z. B. das Auslesen von Daten aus einem Socket. Dazu wird die Funktion `read` von dem zugehörigen IEEE 1394-Socket-Objekt in C aufgerufen und dabei die Socket-ID übergeben. Damit kann das zugehörige Objekt in C die Daten auslesen und an den Stream retournieren. Beim Schreiben auf eine Instanz der Klasse `OutputStream` wird das `write` von dem IEEE 1394-Socket-Objekt in C aufgerufen. Dabei muss ebenfalls die Socket-ID

übergeben werden. Durch diese Angabe können die Daten über das richtige Objekt versendet werden. Da das Interface von den Streams unabhängig davon ist, ob es sich um einen isochronen oder asynchronen IEEE 1394-Socket handelt, wird die Funktionalität der IEEE 1394-Streams in den Basisklassen `IEEE1394InputStraem` und `IEEE1394OutputStream` implementiert. Zur besseren Unterscheidbarkeit für den Programmierer wird jedoch von diesen Streams je eine isochrone und eine asynchrone Variante abgeleitet, die keine zusätzliche Funktionalität besitzen. Die Klassen `IEEE1394InputStream` und `IEEE1394OutputStream` können im Gegensatz zu den IEEE 1394-Socket-Klassen von den Java-Streams abgeleitet werden, da sie keinen Bezug zur Klasse `InetAddress` aufweisen. Dadurch kann gewährleistet werden, dass in Java diese Streams wie alle anderen verwendet werden können. Sprich es kann z. B. problemlos ein Writer auf einen `IEEE1394AsyncOutputStream` geöffnet werden.

Es stellt sich nun die Frage, wie von einer Stream-Klasse im Paket `at.ac.tuwien.ict.net` auf die Information der Socket-ID der Instanz der Klasse `IEEE1394SocketImpl`, befindlich im Paket `at.ac.tuwien.ict.io`, zugreifen werden kann. Der Zugriff auf diese Information ist mit `protected` geschützt. Da sich die Stream-Klassen jedoch im Paket `at.ac.tuwien.ict.net` befinden, kann daher diese Information nicht abgerufen werden. Nun stellt sich die Frage, da es sich bei der IEEE 1394-Socket-Implementierung um eine Kopie des Java-Sockets handelt, wie hat SUN dieses Problem gelöst? Denn auch diese müssen an die Information, welche Socket-Instanz bearbeitet werden soll, heran kommen. Dazu gibt es eine einfache Lösung: Das JNI berücksichtigt die Attribute für die Sichtbarkeit von Klassen nicht (`public`, `protected` und `private`), womit es möglich wird, auch auf `protected` Objekte und Felder in einem anderen Paket zuzugreifen. Dabei handelt es sich um einen Weg, der für diese Diplomarbeit nicht als akzeptabel befunden wurde, denn wozu gibt es ein Sicherheitskonzept, wenn es auf diese Weise ausgeschaltet wird. Es gibt auch andere „legale“ Wege, um an diese Information heran zu kommen. Die einfachste Variante wäre die gewünschte Information `public` zu deklarieren. Dies ist jedoch aus Sicherheitsgründen abzulehnen.

Wird aber eine Klasse `IEEE1394StreamSocketAPI` im Paket `at.ac.tuwien.ict.io` entwickelt, die von der Klasse `IEEE1394SocketAPI` aus dem Paket `at.ac.tuwien.ict.net` abgeleitet ist, können `protected` deklarierte Informationen vom Paket `at.ac.tuwien.ict.net` ausgelesen werden.

IEEE 1394-Datagram-Socket

Für den IEEE 1394-Datagram-Socket gilt ähnliches wie für die IEEE 1394-Sockets. Auch dieser stellt eine Kopie des Datagram-Sockets aus Java dar, da auch er wegen der Klasse `InetAddress` nicht abgeleitet werden kann. In *Abbildung 3-4* ist das zugehörige Klassendiagramm dargestellt.

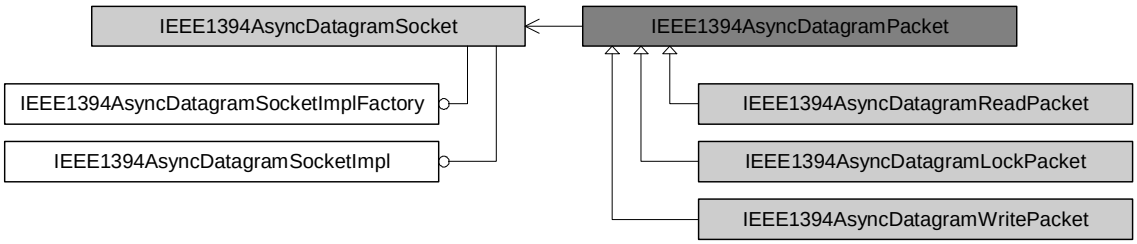

Abbildung 3-4 Klassendiagramm IEEE 1394-Datagram-Socket

In der Klasse `IEEE1394AsyncDatagramSocket` sind alle notwendigen Methoden enthalten um Datagram-Pakete versenden oder empfangen zu können. Da der IEEE 1394-Bus drei mögliche Kommunikationsarten besitzt, sind weiters drei IEEE 1394-Datagram-Pakete zu entwickeln. Es bietet sich an, eine Klasse `IEEE1394AsyncDatagramPaket` zu erstellen, die die Grundfunktionalität der Datagram-Pakete zur Verfügung stellt. Davon werden die drei möglichen Datagram-Pakete für `read`, `write` und `lock` abgeleitet. Wird ein `IEEE1394AsyncDatagramReadPaket` versendet, kommt es real am IEEE 1394-Bus zu einer Leseaktion vom gewünschten Adapter im gewünschten Adressbereich. Beim Empfangen eines solchen Datagram-Paketes wird darauf gewartet, dass eine andere Applikation auf dem angegebnen Speicherbereich eine Leseoperation durchführt. Gleiches gilt für `write` und `lock`. Eine asynchrone IEEE 1394-Datagram-Socket-Instanz muss ein eindeutiges Datagram-Paket versenden (der Sender sollte wissen was er tun will), kann aber auf

ein abstraktes Datagram-Paket warten. Daher besitzt bereits das abstrakte Datagram-Paket die volle Funktionalität. Wäre dies nicht der Fall, könnte es speziell beim Datagram-Paket vom Typ `lock` zu Problemen kommen, da dieses zwei zusätzliche Datenpuffer benötigt.

3.2.2 IEEE 1394-Socket in C

Diese Schnittstelle wurde an den „Ur-Socket“ von Unix [UNIX3] [CNW1 pp349-364] angelehnt. Diese Vorgehensweise wurde gewählt, da es sich bei dieser Schnittstelle um einen Socket handelt, der bereits anerkannt ist und weiträumig Verwendung findet.

Die Unterscheidung zwischen einem Socket und einem Datagram-Socket kann in C nicht wie in Java durch unterschiedliche Klassen dargestellt werden. Daher werden unterschiedliche Funktionen für die beiden verwendet. Für den Socket sind die Funktionen `read` und `write` und für den des Datagram-Socket die Funktionen `recv` und `send` für den Datenaustausch verantwortlich. Da bereits in der Einführung im *Abschnitt 1.3.4* auf die Funktionen des Sockets und Datagram-Sockets in C eingegangen wurde, wird an dieser Stelle nicht mehr im Detail auf jede Funktion eingegangen. Es werden nur jene Funktionen erwähnt, bei denen für die IEEE 1394-Sockets ein zusätzlicher Erklärungsbedarf besteht oder jene welche zusätzlich benötigt werden.

isNodeIDValid, isGUIDValid

Mit der Funktion `isNodeIDValid` kann überprüft werden, ob eine gegebene `node_ID` (Parameter `nodeID`) gültig ist, sprich ob sie am Bussystem verfügbar ist. Damit eine `node_ID` eindeutig ist, muss auch der Busresetcounter (Parameter `brc`) angegeben werden.

```
jint isNodeIDValid (
    jint          nodeID,
    jlong         brc
);
```

Die Funktion `isGUIDValid` ermöglicht es zu überprüfen, ob eine gegebene `GUID` (Parameter `guid`) angesprochen werden kann. Diese Funktion gibt es in zwei Versionen. Bei der ersten wird überprüft, ob die gegeben `GIUD` auf einem beliebigen Adapter verfügbar ist (in einem Computersystem können mehrere IEEE 1394-Adaper verfügbar sein). Die zweite Version hingegen führt diese Überprüfung für einen speziellen Adapter (Parameter `adapter`) durch.

```
jint isGUIDValid (
    jlong         guid
);

jint isGUIDValid (
    jlong         guid,
    jlong         adapter
);
```

GetAdapters

Diese Funktion stellt fest, wie viele IEEE 1394-Adatper in dem aktuellen System vorhanden sind. Ist der Rückgabewert positiv, handelt es sich um die Anzahl der installierten Adapter.

```
jint getAdapters (
    int           size,
    jlong*        adapters
);
```

Diese Funktion retourniert zusätzlich zu der Anzahl der verfügbaren Adapter auch die `GUID` dieser Adapter. Dazu muss mit dem Parameter `adapters` ein Speicherbereich übergeben werden, in dem diese Information abgelegt werden kann; der Parameter `size` gibt an, wie groß dieses Feld ist.

GetNodeID, GetGUID

Mit `getNodeID` kann die `node_ID` (Parameter `nodeID`) zu einer gegebnen `GUID` (Parameter `guid`) zu ermitteln.

```
jint getNodeID (
    jlong         guid,
    jint*         nodeID
);
```

Die Funktion `getGUID` ermittelt eine `GUID` (Parameter `guid`) zu einer gegebenen `node_ID`. (Parameter `nodeID`). Damit eine `node_ID` eindeutig angegeben werden kann, muss auch der Busresetcounter (`brc`) angegeben werden.

```
jint getGUID (
    jint          nodeID,
    jlong         brc,
    jlong*        guid
);
```

GetBusresetCounter

Diese Funktion retourniert den aktuellen Wert des Busresetcounters.

```
jlong getBusresetCounter (
    jlong         adapter
);
```

Da in einem Computersystem mehrere Adapter vorhanden sein können, muss mit dem Parameter `adapter` angegeben werden, von welchem der Busreset-counter abgefragt werden soll; jeder IEEE 1394-Bus weist i. a. einen anderen Wert vom Busresetcounter auf.

GetLast1394Status

Um im Fehlerfall einen genaueren Überblick über die Fehlerursache erhalten zu können, kann mit dieser Funktion der letzte fehlerhafte Status einer AL1394-Funktion abfragt werden. Diese Funktion dient mehr der Entwicklung als einer Verwendung im Gebrauch der IEEE 1394-Sockets.

```
jint getLastIEEE1394Status ();
```

GetHostByName, GetHostByAddress

Diese Methoden sind nur der Kompatibilität wegen zu den bestehenden Sockets eingefügt worden. Sie wurden derzeit jedoch nicht implementiert, da dazu so etwas wie ein `DNS-Server (Domain Name Server)` notwendig wäre.

GetAnyLocalAddress

Mit der Funktion `getAnyLocalAddress` kann eine beliebige Adresse generiert werden, um diese dann als Antwortadresse zu verwenden.

```
jint getAnyLocalPort (
    jlong         adapter
);
```

Es gibt Situationen, in denen der Benutzer einen Socket zu einem andern Prozess öffnen möchte, ihm aber die eigene Adresse gleichgültig ist. Um dies zu verdeutlichen folgendes Beispiel: Es wird ein Brief an eine Person versendet. Will diese Person eine Antwort senden, muss sie die Antwortadresse kennen. Der ersten Person kann es aber grundsätzlich gleichgültig sein, wie ihre eigene Adresse ist, solange die erwartete Antwort nur ankommt. Wenn das Postsystem weiß, wohin eine Antwort auf einen Brief zu senden ist, ist dies also völlig ausreichend. Selbiges gilt auch für die Sockets. Daher wurde diese Funktion implementiert. Als Parameter muss ihr der Adapter (Parameter `adapter`) übergeben werden, für den die Adresse zu generieren ist.

GetAvailable

Diese Funktion ermittelt die Anzahl der in der Lesequeue eines Socket-Objektes (Parameter `descriptor`) vorhandenen Bytes.

```
jint getAvailable (
    jlong         descriptor
);
```

Receive

Für den asynchronen IEEE 1394-Datagram-Socket muss diese Funktion, im Gegensatz zu der in der Einleitung beschriebenen, um zwei Parameter erweitert werden. Mit dem Parameter `allowedType` muss angegeben werden, welche Arten von Paketen empfangen werden dürfen. Dabei kann mittels einer Oder-Verknüpfung auch eine Kombination dieser drei Typen (`read`, `write` und `lock`) angegeben werden. Der Parameter `receivedType` gibt schließlich Auskunft darüber, welcher Paket-Type tatsächlich empfangen wurde.

```
jint recvfrom (
    jlong         descriptor,
    jint          allowedType,
    jint*         receivedType,
    jbyte*        buffer,
    jint          length
    jint          flags,
    sockaddr*     sendaddress,
    jint          addresslen
);
```

Send

Diese Funktion wurde beim Asynchronen IEEE 1394-Datagram-Socket auf zwei Arten implementiert, da unterschiedliche Typen von Datagram-Paketen zu behandeln sind. Die Typen `read` und `write` können mit derselben Funktion behandelt werden. In diesem Fall wurde die Funktion, im Gegensatz zu der Beschreibung in der Einführung, nur um den Parameter `packettype` erweitert. Damit wird angegeben, wie das Datenpaket zu versenden ist (`read` oder `write`). Für den Typ `lock` ist eine Funktion erforderlich, die noch zwei weitere Datenbereiche enthält; einerseits für den Vergleich (`comparedata` mit `comparelength`), andererseits für den aktuellen Bestand (`receivedata` mit `receivelength`). Diese Funktion besitzt aber auch die Fähigkeit, mit den Paketen vom Typ `read` und `write` umzugehen. In diesem Fall können die beiden erweiterten Datenbereiche mit dem Wert `NULL` angegeben werden.

Ansonsten kann die Beschreibung von der Einleitung (*siehe Abschnitt 1.3.4*) übernommen werden.

```
jint sendto (
    jlong         descriptor,
    jint          packettype,
    jbyte*        data,
    jint          length,
    jint          flags,
    sockaddr*     destaddress,
    jint          addresslen
);

jint sendto (
    jlong         descriptor,
    jint          packettype,
    jbyte*        data,
    jint          length,
    jbyte*        comparedata,
    jint          comparelength,
    jbyte*        receivedata,
    jint          receivelength,
    jint          flags,
    sockaddr*     destaddress,
    jint          addresslen
);
```

Seek

Mit der Funktion `seek` kann der Lesezeiger eines Socket-Objektes an eine angegebene Position gestellt werden. Als Parameter ist wie üblich die Socket-ID (`descriptor`) aber auch die Position, an die der Zeiger zu stellen ist (Parameter `position`), anzugeben. Bei der nächsten Leseoperation werden die Daten ab dieser neuen Position entnommen.

```
jint seek (
    jlong         descriptor,
    jlong         position
);
```

Diese Funktion ist nicht implementiert; sie wird für die prinzipielle Funktionalität eines Sockets nicht benötigt. Bei der Betrachtung der Klasse `InputStream` von Java, kann festgestellt werden, dass sie auch dort nicht vorhanden ist.

Skip

Will der Benutzer Daten aus der Lesequeue eines Socket-Objektes löschen, kann dies mit dieser Funktion erfolgen. Es wird die übergebene Anzahl von Bytes (Parameter `numbers`) ausgelesen und verworfen. Der zweite Parameter ist wiederum die Socket-ID (`descriptor`).

```
jlong skip (
    jlong         descriptor,
    jlong         numbers
);
```

Fehlerbehandlung

Alle Funktionen besitzen einen numerischen Rückgabewert. Ist dieser positiv oder 0, konnte die Funktion fehlerfrei ausgeführt werden. Bei einem negativen Rückgabewert ist hingegen ein Fehler bei der Ausführung des Befehles aufgetreten. Diesem negativen Zahlenwert kann eine genauere Fehlerursache entnommen werden. Die Fehlerwerte wurden der Lesbarkeit wegen mit Konstanten dargestellt:

`ALREADY_CONNECTED`	Das angegebene Socket-Objekt ist bereits mit einem anderen verbunden.
`API_ERROR`	Ein allgemeiner Fehler ist aufgetreten.
`API_STILL_IN_USE`	Die Gewünschte Funktion wird bereits ausgeführt und kann nicht ein zweites Mal gestartet werden.
`BRC_CHANGED`	Es ist während der Ausführung der gewünschten Aktion ein Busreset aufgetreten. Normalerweise versucht der IEEE 1394-Socket diesen Fehler selbst zu beseitigen, indem er diese Funktion nochmals ausführt. In Extremfällen (es treten nur noch Busresets auf) kann dieser Fehler trotzdem auftreten.
`CANNOT_ACCEPT`	Das warten auf einen Verbindungsaufbau ist fehlgeschlagen.

CANNOT_CONNECT	Es kann keine Verbindung zu dem gewünschten Partner aufgebaut werden.
CLOSED	Das Socket-Objekt wurde geschlossen.
INVALID_ADAPTER	Der angegebene IEEE1394-Adapter existiert nicht in diesem System.
INVALID_ADDRESS	Die angegebne Adresse kann nicht aufgelöst werden; sie ist nicht am IEEE 1394-Bus verfügbar.
INVALID_ARRAY_SIZE	Das übergebene Datenfeld hat eine falsche Größe.
INVALID_DATA_BUFFER	Der Angegebene Datenbuffer ist ungültig.
INVALID_GUID	Die angegebene GUID ist auf diesem IEEE 1394-Bus nicht verfügbar.
INVALID_NODEID	Die angegebene node_ID ist auf diesem IEEE 1394-Bus nicht verfügbar.
INVALID_PACKET_TYPE	Es wurde ein Paket mit einem ungültigen Format empfangen.
INVALID_SOCKET_STATE	Die gewünschte Funktion kann in dem aktuellen Zustand des Socket-Objektes nicht ausgeführt werden.
INVALID_SOCKET_TYPE	Es wurde ein ungültiger Socket-Typ angegeben.
INVALID_SOCKETID	Die angegebene Socket-ID ist nicht korrekt.
INVALID_STRING	Der übergebene String konnte nicht verarbeitet werden.
IO_ERROR	Es ist ein Fehler bei einer Ein- Ausgabeoperation aufgetreten.
IS_CLOSING	Das angegebene Socket-Objekt wird geschlossen. Daher kann die gewünschte Aktion nicht ausgeführt werden.
LOCK_ERROR	Der IEEE 1394-Befehl lock ist fehlgeschlagen.
NO_CLIENT_SOCKET	Es handelt sich nicht um einen Client-Socket.
NO_IEEE1394_ADAPTER	Es ist kein IEEE 1394-Adapter im System verfügbar.
NO_SERVER_SOCKET	Es handelt sich um keinen Server-Socket.
NOT_BOUND	Das angegebene Socket-Objekt hat noch keine lokale Adresse.
NOT_CONNECTED	Um die gewünschte Funktion ausführen zu können, müsste dieses Socket-Objekt mit einem anderen verbunden sein, was jedoch nicht der Fall ist.
NOT_FOUND	Die zu suchende Information konnte nicht gefunden werden.
NOT_IMPLEMENTED_YET	Diese Funktion ist noch nicht implementiert.
NOT_PASSIVE	Das angegebene Socket-Objekt ist nicht im passiven Modus. Es kann daher kein accept durchgeführt werden.
NOT_SUPPORTED	Die Angegebene Funktion wird (noch) nicht unterstützt.
READ_ERROR	Der IEEE 1394-Befehl read ist fehlgeschlagen.
SUCCESSFUL	Es ist kein Fehler aufgetreten.
TIMEOUT	Es wurde ein Zeitlimit überschritten.
WRITE_ERROR	Der IEEE 1394-Befehl write ist fehlgeschlagen.

Tabelle 3-2 Fehlercodes

3.2.3 IEEE 1394-API (AL1394)

Es wurde bereits in der Einführung darauf eingegangen, weswegen der AL1394 als IEEE 1394-API Verwendung findet (*siehe Abschnitt 1.4*). Anfänglich hat die Auswahl dieser Schnittstelle jedoch einige Probleme mit sich gebracht. Erst wurde an die Verwendung eines Adapters der Firma Unibrain gedacht. Doch durch die Entwicklung des AL1394 wurde schnell zu Gunsten dessen entschieden. Die erste Entwicklung des AL1394 war nur für das Betriebssystem Win2000 verfügbar, weshalb auch die Implementierung der IEEE 1394-Sockets

auf dieses Betriebssystem beschränkt wurde. Leider musste während der Arbeit festgestellt werden, dass diese Implementierung des AL1394 zu diesem Zeitpunkt noch nicht alle notwendigen Funktionen zur Verfügung stellen konnte. Es war auch nicht ersichtlich, wann die notwendigen Komponenten zur Verfügung stehen würden. Da die Entwicklung vom AL1394 auf das Betriebssystem Linux umgestellt wurde, konnte dieser die benötigte Funktionalität vorweisen. Aus diesem Grund wurde auch die Implementierung der IEEE 1394-Sockets auf das Betriebssystem Linux umgestellt.

3.3 Interne Schnittstellen

Bei den internen Schnittstellen handelt es sich um solche, die nicht von außen angesprochen werden können. Da die Implementierung der IEEE 1394-Sockets in mehrere Teile gegliedert wurde, muss auch zwischen diesen je eine Schnittstelle definiert werden.

3.3.1 Kommunikationsmodul

Das Kommunikationsmodul stellt die wichtigste interne Schnittstelle dar. Sie stellt alle benötigten Funktionen des AL1394 in der Form zur Verfügung, dass sie mit den internen, in der Implementierung dieser Diplomarbeit verwendeten Definitionen, verwendet werden können. So wird z. B. beim AL1394 die Definition `AL1394_OFFSET` als Adresse am Adapter verwendet. Da in Java dafür ein einfacher Integer Verwendung findet, wurde dieser auch für die gesamte Implementierung verwendet. Nur beim Aufruf der AL1394-Funktion muss dieser Wert umgewandelt werden. Diese Abstraktionsebene hat sich auch deshalb sehr gut bewährt, da durch den Tausch der IEEE 1394-API während dieser Diplomarbeit somit nur dieses Interface abgeändert werden musste.

Weiters ermöglicht dieses Interface, dass die Funktionen `read`, `write` und `lock` auf eine `GUID`, und nicht nur auf eine `node_id`, ausgeführt werden können. Aus diesem Grund besitzt dieses Modul eine Umrechnungstabelle, in der alle verfügbaren Adapter mit deren `node_id` und `GUID` eingetragen sind.

Soll ein Befehl auf eine `GUID` ausgeführt werden, sucht die Funktion zuerst die zugehörige `node_id` und gibt dann mit diesem Wert den Aufruf an den AL1394 weiter. Tritt während einer dieser Funktionen ein Busreset auf, sorgt sie auch dafür, dass die Operation nach Beendigung des Busresets automatisch wiederholt wird.

3.3.2 IEEE 1394-Socket API

Mit dieser Schnittstelle werden alle Funktionen der IEEE 1394-Sockets in C, auch der IEEE 1394-Socket-Core genannt, für Java zur Verfügung gestellt. Um dies zu ermöglichen, integriert diese Schnittstelle das JNI und sorgt dafür, dass die Parameteranpassung zwischen Java und C korrekt funktioniert. Die Header-Datei für C wird mittels des Tools `javah` vom JDK erstellt, und muss im Modul IEEE 1394-Socket-API inkludiert werden.

3.3.3 Tools

Während der Entwicklung hat sich herausgestellt, dass einige Funktionen häufig benötigt werden, die aber keinem Modul zuordenbar sind. Daher wurden sie im Modul Tools implementiert. Besonders zu nennen wären betriebssystem-abhängige Funktionen für die Synchronisation, der shared memory, die Signalisierung und die Lese-Queue. Dadurch ist sichergestellt, dass alle Funktionen vom Betriebssystem in diesem Modul gekapselt sind. Dies kommt einer Entwicklung der Bibliothek für die IEEE 1394-Sockets für ein anderes Betriebssystem zu gute. Weiters enthalten die Tools Funktionen zur Umrechnung von Zahlen in ein ByteArray oder von einer InetAddress in die einzelnen Werte der Adresse (z. B. `GUID` und Port bei einem asynchronen IEEE 1394-Socket).

3.3.4 Logging

Um Fehler leichter auffinden zu können, ist es sinnvoll, alle Vorgänge in den IEEE 1394-Sockets mitzuprotokollieren. Dies sollte einheitlich für die gesamte Diplomarbeit erfolgen. Dies erleichtert das Lesen und Auffinden von Einträgen in der Protokolldatei.

Bei der Protokollierung wurden mehrere Stufen eingeführt:

- LL_FATAL_ERROR
- LL_ERRO
- LL_WARN
- LL_INFO
- LL_DEBUG.

Beim Kompilieren des Moduls Logging kann angegeben werden, ab welcher Stufe die aufgetretenen Ereignisse mitprotokolliert werden sollen. Für die Entwicklung ist es wichtig, so viele Details wie möglich mitzuschreiben, um den Fehler besser eingrenzen zu können. Im Betrieb ist dies jedoch hinderlich, da die Flut an Informationen die Protokolldatei schnell sehr groß werden lassen (es entstehen einige MB pro Verbindung), und der Anwender nicht viel mit den detaillierten Informationen anfangen wird; für ihn sollte ein grober Überblick genügen, wie z. B. dass ein Fehler aufgetreten ist. Weiters wird auch der Name des Moduls aufgezeichnet, von dem der Protokolleintrag erfolgt. Da es sich bei dem verwendeten Betriebssystem um ein Multiprotzessystem handelt, darf auch nicht darauf vergessen werden, die Prozessnummer mitzuschreiben, ansonsten ist es nicht möglich, die korrekte Reihenfolge der Einträge zu rekonstruieren.

3.4 Protokolle

Beim asynchronen IEEE 1394-Datagram-Socket sollen direkt die Funktionen der IEEE 1394-API verwendet werden. Daher ist in diesem Fall kein Protokoll für den Datenfluss erforderlich. Anders sieht dies bereits beim isochronen IEEE 1394-Socket aus. Da jedoch die IEC den Standard IEC 61883 [IEC1] für den isochronen Datenaustausch entwickelt hat, scheint es sinnvoll, diesen zu verwenden. Damit bleibt nur mehr der asynchrone IEEE 1394-Socket. Wie bereits in den Lösungsvorschlägen (*siehe Abschnitt 2.2.4*) angedeutet, kann dieses Protokoll an das TCP angelehnt werden.

3.4.1 IEC 61883

Dieser Standard wurde ursprünglich auf einer HD Digital VCR-Konferenz entwickelt und 1997 erstmals spezifiziert. Er erfordert einige Erweiterungen der standardmäßigen isochronen IEEE 1394-Kommunikation. Isochrone Pakete werden im so genannten Common Isochronous Packet Format (CIP) übertragen. Aus diesem Grund wird dem Datenstrom ein CIP-Header vorangestellt [IEC1 pp15, fig 2]. Weiters definiert dieser Standard die Connection Management Procedures (CMP). Diese beschreiben, wie isochrone Datenströme aufgebaut, überlagert und wieder unterbrochen werden können [IEC1 pp31-37]. Um dies zu ermöglichen, gibt es das Function Control Protocol (FCP). Dabei handelt es sich um einen Adressraum am IEEE 1394-Adapter, der den Informationsaustausch für die Steuerung der Kanäle ermöglicht. Die Steuerbefehle werden asynchron übertragen und in command und response unterteilt [IEC1 pp37-39, fig 29].

3.4.2 Protokoll des asynchronen IEEE 1394-Sockets

Das Protokoll für den asynchronen IEEE 1394-Socket wird an das TCP angelehnt (*siehe Abschnitt 2.2.4*). Bei der gewählten Implementierung handelt es um ein dem TCP sehr ähnliches Protokoll; TCP geht davon aus, dass das darunter liegende IP keinerlei Art von Datensicherung besitzt. Der IEEE 1394-Bus besitzt bereits einen Transaction-Layer für asynchrone Datenpakete. Dieser garantiert, dass die Datenpakete beim Empfänger ankommen, oder im Fehlerfall der Sender über das Fehlverhalten informiert wird.

Datenpakete

Ein für den asynchronen IEEE 1394-Socket über den IEEE 1394-Bus zu übertragendes Datenpaket besteht aus einem Protokoll-Header und den eigentlich zu versendenden Daten. Es besitzt eine maximale Länge von 1024 Bytes (*siehe Abbildung 3-5*); der Header muss immer versendet werden und besitzt eine Länge von 24 Bytes. Er ist notwendig, um das Datenpaket eindeutig identifizieren zu können. Außerdem können damit auch Steuerbefehle übertragen werden. Für den effektiven Datentransfer bleiben somit 1000 Bytes

pro Paket übrig. Ein Datenpaket kann jedoch bis zu dieser Grenze eine beliebige Anzahl von Bytes übertragen; in Abbildung 3-5 ist dies durch den grauen Bereich dargestellt. Bei einem Steuerpaket (z. B. für das Verbindungsmanagement) werden normalerweise keine Daten mit versendet. In diesem Fall beträgt die Datenlänge daher 0 Bytes.

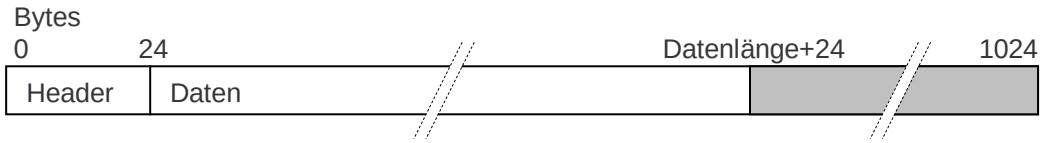

Abbildung 3-5 Datenpaket

Protokoll-Header

Jedes Datenpaket besitzt einen Protokoll-Header. Dieser enthält Daten, die das Paket genauer spezifizieren. *Abbildung 3-6* stellt alle Teile des Headers und ihre Anordnung graphisch dar. Der in dieser Implementierung verwendete Header unterscheidet sich in keiner Weise von jenem vom TCP [RFC793 pp15].

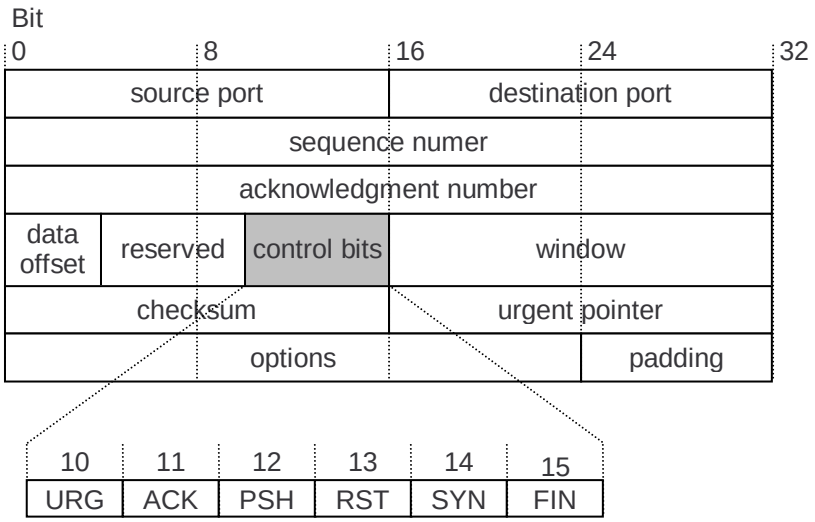

Abbildung 3-6 Protokoll-Header

Im Folgenden wird kurz auf die einzelnen Felder eingegangen. Dabei ist zu beachten, dass in dieser Diplomarbeit zwar alle Felder vorhanden, aber nur einige davon verwendet werden. Dies ist deshalb der Fall, um bei einer eventuell folgenden Erweiterung der aktuellen Implementierung alle Möglichkeiten zur Verfügung zu haben. Für eine Demonstration ob der asynchrone

IEEE 1394-Socket prinzipiell funktioniert, ist es jedoch nicht notwendig, die gesamte Leistungsfähigkeit zu implementieren.

- `source port` und `destination port`

Diese Informationen müssen in diesem Protokoll untergebracht sein, da es sich bei den Ports um zusätzliche Adressdaten handelt, die durch dieses Protokoll eingeführt werden. Würden die Ports nicht enthalten sein, könnten die Pakete nicht eindeutig identifiziert werden.

- `sequence number`

Sie stellt sicher, dass der Datenstrom, der in Pakete zerstückelt werden muss, wieder in der richtigen Reihenfolge zusammengesetzt werden kann. In dieser Implementierung wird jedes Datenpaket mit einer aufsteigenden Nummer versehen. Damit kann überprüft werden, ob die Datenpakete in der korrekten Reihenfolge empfangen werden. Aufgrund des Transaction-Layers sollte jedoch sichergestellt sein, dass die Pakete nicht umgereiht werden.

- `acknowledgment number`

Damit kann dem Sender die nächste zu erwartende `sequence number` mitgeteilt werden. Da aufgrund des Transaction-Layers sichergestellt ist, dass Datenpakete korrekt ankommen, wird dieses Feld nur beim Verbindungsaufbau mit einem Wert belegt.

- `data offset`

Zeigt an, ab wo der Datenstrom in dem aktuellen Datenpaket beginnt. Anders ausgedrückt handelt es sich um die Länge des Protokoll-Headers. Diese Angabe erfolgt nicht in Bytes, sonder in Längen von 32 Bits; die tatsächliche Offsetposition beginnt somit beim Vierfachen dieses Wertes. Die Implementierung in dieser Diplomarbeit unterstützt keine Optionen, daher kann sich die Länge des Protokoll-Headers nicht verändern; dieses Feld enthält somit immer den Wert 24.

- reserved

Dieser Bereich wird nicht verwendet, wurde aber für zukünftige Erweiterungen reserviert.

- `control bits`

Diese dienen der Steuerung des Sockets. Es stehen folgende sechs Kontrollbits zur Verfügung:

- `URG` Das Feld `urgent pointer` enthält Daten.
- `ACK` Das Feld acknowledgment besitzt einen Wert.
- `PSH` Es handelt sich um eine push-function.
- `RST` Es tritt ein Reset der Verbindung ein.
- `SYN` Es erfolg eine Synchronisierung.
- `FIN` Es folgen keine weiteren Daten vom Sender.

Die Kontrollbits sind beim Öffnen und Schließen einer Socket-Verbindung sehr wichtig. Das Bit `SYN` zeigt an, dass eine Verbindung geöffnet werden soll. Ein gesetztes `FIN`-Bit informiert darüber, dass die Verbindung geschlossen wird. Beim Bit `ACK` handelt es sich immer um eine Bestätigungsmeldung. Das Bit `URG` zeigt an, dass es sich um dringliche Daten handelt. Sie werden meist bevorzugt behandelt. Das Flag `PSH` signalisiert, dass ein `flush` ausgeführt wurde, und bei `RST` wird eine Verbindung zurückgesetzt.

Da keine zusätzliche Datenpufferung für eine Performancesteigerung implementiert ist, werden immer alle Daten direkt über den Bus versendet. Daher ist die Funktion `flush` ohne Funktion, was auch bedeutet, dass kein Flag `PSH` versendet wird. Weiters wird auch das Flag URG nicht verwendet, da es nicht vorgesehen ist, Daten bevorzugt zu übertragen; aufgrund des Transaction-Layers ist dies nicht notwendig, da dieser dafür sorgt, dass die Daten direkt übertragen werden. Es stellt sich daher nicht die Frage, ob z. B. ein Router Datenpakete bevorzugen soll oder nicht.

- `window`

Damit gibt der Sender das Ausmaß des Datenvolumens an, das er empfangen kann. Bei dieser Angabe handelt es sich ebenfalls um eine Anzahl der 32-Bit Werte. Diese Angabe enthält bei der Implementierung für den asynchronen IEEE 1394-Socket immer den Wert 0, da alle asynchronen IEEE 1394-Sockets gleicht implementiert sind und jeder genau einen Speicherbereich von 1024 Bytes reserviert.

- `checksum`

Wie der Name schon sagt, handelt es sich dabei um eine Prüfsumme. Sie kann verwendet werden, um die Datenpakete auf ihre Korrektheit zu überprüfen. Da Aufgrund des Transaction-Layers sicher gestellt ist, dass Datenpakete korrekt empfangen werden, wird dieses Feld im Fall des asynchronen IEEE 1394-Sockets nicht befüllt.

- `urgent pointer`

Dieses Feld ist nur dann gültig, wenn das `control bit URG` gesetzt ist. Der enthaltene Wert ist jene `sequence number`, bis zu der die Daten bevorzugt werden. Da das Flag `URG` nicht verwendet wird, kommt auch dieses Feld nie zum Einsatz (siehe `control bits`).

- `options`

Dieses Feld spezifiziert die Übergabe von Optionen. Da die Implementierung des asynchronen IEEE 1394-Sockets zeigen soll, dass eine Socket-Übertragung möglich ist, wird darauf Verzichtet das Feature für die Einstellung der Optionen zu implementieren.

- padding

Dieser Bereich ist ungenutzt, er dient dazu, den Header auf eine Länge zu erweitern, die durch 32 Bit teilbar ist. Da in dieser Implementierung keine Optionen übertragen werden, kann sich der Header nicht in seiner Länge verändern. Daher besitzt dieser Bereich in diesem Fall immer die Länge 1 und ist mit dem Wert 0 befüllt.

Verbindungsmanagement

Das Verbindungsmanagement stellt eine Zustandsmaschine dar, und kann somit auch sehr anschaulichsten graphisch dargestellt werden (*siehe Abbildung 3-7 und Abbildung 3-8*). Es besteht aus mehreren Zuständen, die aufgrund von Eingaben aus der Applikation oder des Verbindungspartners geändert werden. Da es sich bei den IEEE 1394-Sockets um ein Kommunikationssystem handelt, wurde eine Darstellung mittels SDL gewählt. SDL bedeutet „Specification and Description Language“ und wurde erstmals 1972 von der CCITT (Comitè Consultatif International Telegraphique et Telephonique) standardisiert. Es wurde für Echtzeitanwendungen, interaktive Anwendungen und Verteilte Systeme entwickelt [SDL1 pp2]. In *Abbildung 3-7* ist das SDL-Diagramm für das Öffnen einer Socket-Verbindung dargestellt; *Abbildung 3-8* enthält das SDL-Diagramm für das Schließen. Beide Zustandsdiagramme sind vom `TCP` entnommen [RFC793 pp23] und wurden beim Asynchronen IEEE 1394-Socket genau so implementiert. Zur besseren Übersichtlichkeit sind in den Abbildungen die Zustände dunkelgrau und die Eingaben von der Applikation hellgrau hinterlegt.

Verbindungsaufbau

Beim Öffnen einer Verbindung wird ein `Three-Way-Handshake` verwendet [RFC793 pp30-37]. Damit kann vermieden werden, dass der Verbindungsaufbau gestört wird. Im Fall einer Störung kann dadurch der Verbindungsaufbau abgebrochen werden, wodurch es zu keiner falschen Verbindung kommen kann. In *Abbildung 3-7* ist sowohl der aktive Teil, Abzweigung `connect`, als auch der passive Teil, Abzweigung `listen`, des Verbindungsaufbaus enthalten. Zu Beginn muss jedoch ein Socket-Objekt erstellt werden. Zu diesem Zeitpunkt befindet sich dieses im Zustand `Closed`. Danach muss diesem Objekt seine eigene lokale Adresse mit der Funktion `bind` mitgeteilt werden. Das Socket-Objekt wird damit in den Zustand `Bound` versetzt. Ab diesem Zeitpunkt kann dieses entweder aktiv oder passiv verwendet werden.

Ein passives Socket-Objekt, in Java auch Server-Socket genannt, wartet auf einen Verbindungsaufbau, ein aktives Socket-Objekt hingegen initiiert einen Verbindungsaufbau. Damit ein Socket-Objekt jedoch auf eine Verbindung

warten kann, muss es darauf vorbereitet werden. Dies erfolgt mittels der Funktion `listen`. Dabei wird die Queuegröße festgesetzt und das Objekt wechselt in den Zustand `Passive`. Nun kann das Objekt des Server-Sockets mittels `accept` in den Wartezustand `Listen` versetzt werden. Hier verharrt dieses so lange, bis ein anderer Partner eine Verbindung aufbauen will.

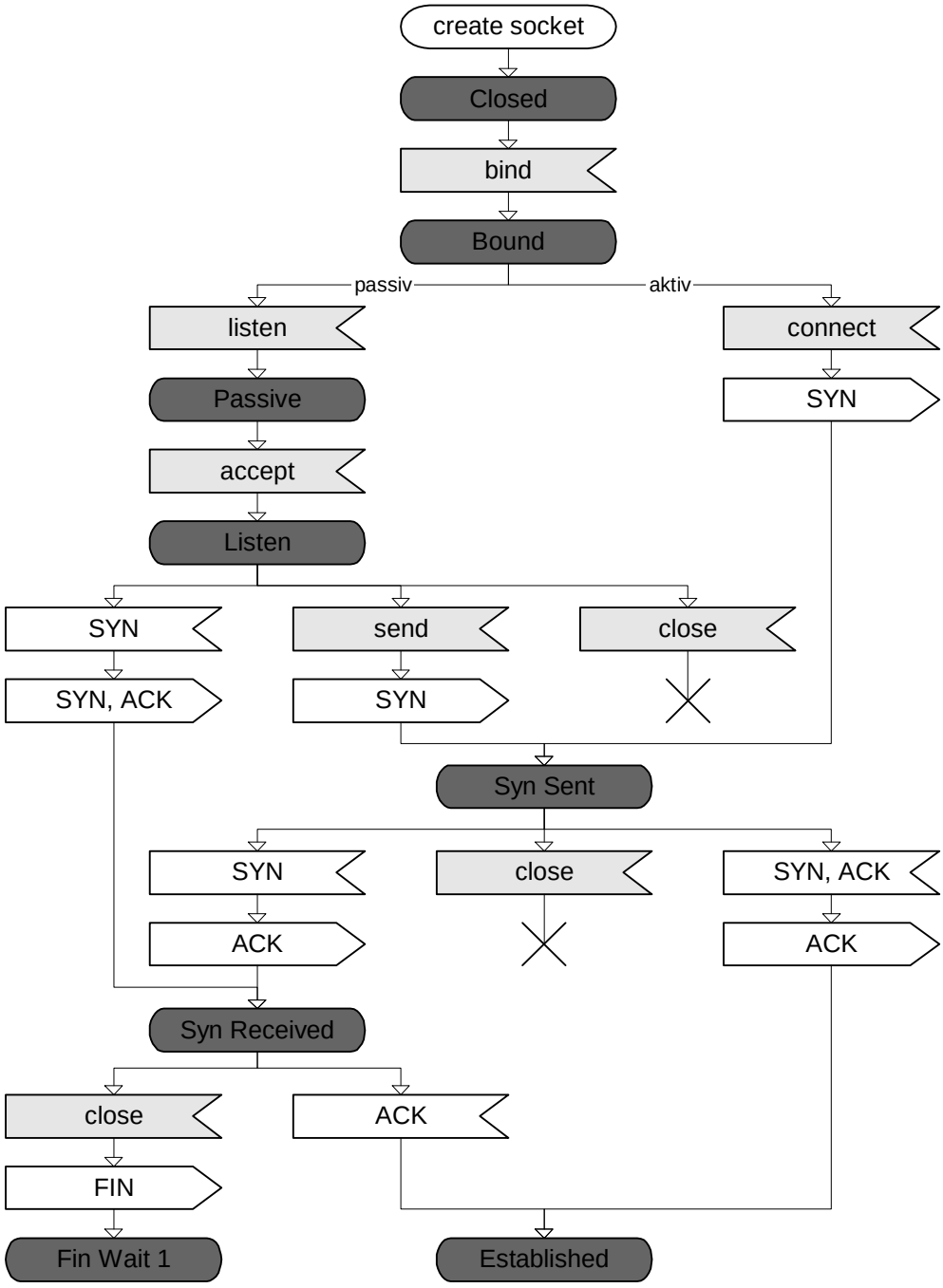

Abbildung 3-7 Verbindungsaufbau

Ein aktives Socket-Objekt, kann mittels `connect` aktiv einen Verbindungsaufbau initiieren. Dadurch wird ein SYN-Paket versendet, und das zugehörige Objekt wird in den Zustand `Syn Sent` versetzt. Das Objekt des Server-Sockets empfängt dieses Paket und beantwortet dieses mit einem SYN-ACK-Paket. Danach verharrt dieses in dem Zustand `Syn Received`. Nachdem das Objekt des aktiven Sockets dieses Paket empfangen hat, antwortet dieses nochmals mit einem ACK-Paket und geht in den Zustand `Established` über. Das Objekt des Server-Sockets empfängt wiederum dieses Paket; damit ist auch von dieser Seite der Verbindungsaufbau komplettiert, und das zugehörige Objekt wechselt ebenfalls in den Zustand `Established`. Bei dieser Beschreibung handelt es sich um die Darstellung eines normalen Vorganges des Verbindungsaufbaus.

Natürlich muss zu jedem Zeitpunkt behandelt werden, was geschehen soll, wenn der Wunsch nach einem Abbruch des Verbindungsaufbaus auftritt, oder wenn ein falsches Paket empfangen wird. Weiters kann ein passives Socket-Objekt in ein aktiv öffnendes überwechseln. Dies erfolgt in diesem Zustandsdiagramm mittels der Eingabe `send` aus dem Zustand `Listen`. Es ist leicht zu erkennen, dass in diesem Fall ebenfalls ein SYN-Paket wie beim `connect` versendet wird. Selbiges gilt auch für den aktiven Teil, dieser kann vom Zustand `Syn Sent` mittels eines SYN-Paketes in den passiven Teil überwechseln. Diese Zweige werden jedoch so gut wie nie genutzt.

Würde während eines laufenden Verbindungsaufbaus ein anderes Socket-Objekt versuchen, ebenenfalls eine Verbindung aufzubauen, würde dies abgelehnt werden, da sich keine Objekt eines Socket in einem entsprechenden Zustand befände. Würde, aus welchem Grund auch immer, ein falsches Paket empfangen werden, könnte dieses anhand der Adressdaten und der `sequence number` als falsch erkannt werden. Da es sich bei der Implementierung des asynchronen IEEE 1394-Sockets um keine Sicherheitsverbindung handelt, werden solche Pakete verworfen. Bei Secure-Sockets, dabei handelt es sich um gesicherte Verbindungen, müssten solche Pakete als „Bedrohung“ angesehen werden, und der Verbindungsaufbau würde abgebrochen oder neu gestartet werden. Ein Abbruch erfolgt durch das Versenden eines `FIN`-Paketes, ein Neustart mit einem RST-Paket.

Verbindungsabbruch

Das Schließen einer Socket-Verbindung, dargestellt in *Abbildung 3-8*, kann sowohl von der aktuellen Applikation, siehe den Zweig `close`, als auch von dem anderen Verbindungspartner mittels eines `FIN`-Paketes initiiert werden. Wurde das Schließen einer Verbindung durch den anderen Partner eingeleitet, wird dies durch das Versenden eines `ACK`-Paketes bestätigt, und das Socket-Objekt wechselt in den Zustand `Close Wait`. Darin verharrt es solange, bis die Applikation die halb geschlossene Socket-Verbindung ganz schließt. Dies wird dem anderen Partner durch Versenden eines `FIN`-Paketes mitgeteilt, der dann das Schließen endgültig mit einem `ACK`-Paket bestätigt. Daher wird noch im Zustand `Last Ack` auf ein solches Paket gewartet.

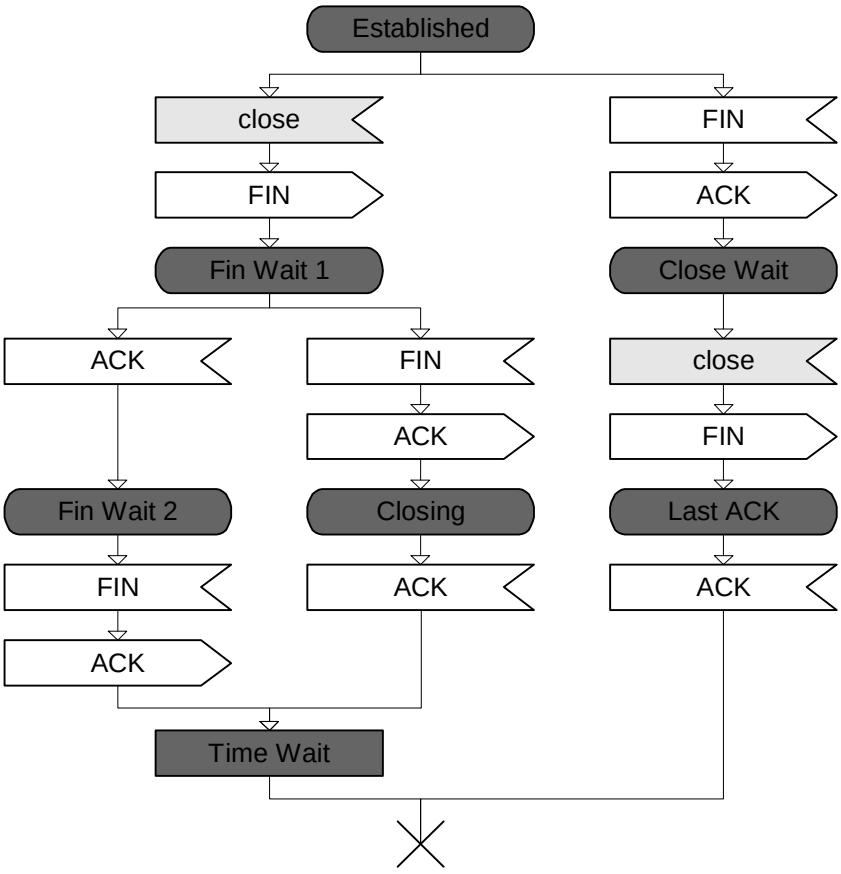

Abbildung 3-8 Verbindungsabbruch

Wird hingegen die Verbindung von der Applikation geschlossen, wird zuerst der gewünschte Abbruch durch Versenden eines `FIN`-Paketes angezeigt. Danach müssen zwei Möglichkeiten unterschieden werden: Entweder der andere

Partner geht nach dem oben beschriebene Vorgang vor, oder wünscht gleichzeitig einen Verbindungsabbruch. Diese Entscheidung wird im Zustand `Fin Wait 1` abgewartet. Akzeptiert der andere Partner den Abbruch, versendet dieser ein ACK-Paket. Dadurch wechselt das Socket-Objekt in den Zustand `Fin Wait 2`. Dort verharrt dieses Objekt so lange, bis der andere Partner mit `close` geschlossen wird, denn dadurch versendet er ein `FIN`-Paket. Dieses Paket ermöglicht den Wechsel des Socket-Objektes in den Zustand `Time Wait`, wobei dieser Übergang nochmals mit dem Versenden eines ACK-Paketes bestätigt wird. Schließt der andere Partner die Verbindung gleichzeitig, versendet er ebenfalls ein `FIN`-Paket. Dadurch kann das Socket-Objekt in den Zustand `Closing` wechseln, was es dem anderen Partner durch das Versenden eines ACK-Paketes mitteilt. In diesem Zustand wird schließlich noch auf die endgültige Bestätigung des anderen Partners gewartet. Nachdem die aktuelle Applikation das Socket-Objekt geschlossen hat, wird im Zustand `Time Wait` eine Zeit von 2 MSL (Maximum Segment Lifetime) gewartet. Dadurch wird gewährleistet, dass kein verspätetes Pakete für dieses Socket-Objekt eintreffen kann. Diese Zeit könnte beim asynchronen IEEE 1394-Socket aufgrund des Transaction-Layers vernachlässigt werden. Um jedoch der Applikation Zeit zu geben, notwendige Aktionen, wie z. B. das Freigeben benötigter Ressourcen, durchführen zu können, wird diese Wartezeit implementiert, wenn auch nur sehr kurz. Aus diesen Abläufen kann erkannt werden, dass es sich auch beim Schließen einer Verbindung um einen Three-Way-Handshake handelt.

Dem aufmerksamen Leser dürfte nicht entgangen sein, dass ein Zustandsübergang, dargestellt in *Abbildung 3-7*, nicht behandelt wurde. Wurde nämlich bereits ein Verbindungsaufbau vom Gegenüber initiiert, aber noch nicht vollends bestätigt, und will die Applikation diese halb offene Verbindung bereits wieder schließen (Zustand `Syn Received`), kann dies direkt mit der Funktion `close` erfolgen. Dabei wird ein `FIN`-Paket versendet und das zugehörige Objekt wechselt in den Zustand `Fin Wait 1` (*Abbildung 3-8*). Anschließend wird das Schließen der Verbindung wie oben beschrieben fortgesetzt.

Wann können Daten übertragen werden?

Daten können über ein Socket-Objekt nur im Zustand `Established` ausgetauscht werden. Eine Ausnahme bildet der Zustand `Close Wait`; dabei handelt es sich um eine halb geschlossene Socket-Verbindung. Von einer solchen wird deswegen gesprochen, da eine Applikation den Abbruch durch `close` initiiert hat, die andere diesen aber noch nicht bestätigt hat. Zu diesem Zeitpunkt dürfen daher keine Daten mehr über eine halb geschlossene Socket-Verbindung versendet werden, jedoch kann die Applikation, die den Abbruch nicht eingeleitet hat, noch alle vorhandenen Daten aus einem solchen Socket-Objekt auslesen. Dies ist notwendig, da z. B. eine Applikation Daten über ein Socket-Objekt versendet und die Verbindung abbricht, sobald sie keine Daten mehr senden will. Kann die andere Applikation nicht alle Daten schnell genug von dieser Verbindung auslesen, kann es sein, dass die Verbindung bereites geschlossen wird, obwohl immer noch nicht ausgelesene Daten vorhanden sind. Daher muss gewährleistet sein, dass diese trotzdem ausgelesen werden können. Es dürfen jedoch keine weiteren Daten versendet werden, da nicht sicher gestellt ist, dass diese noch ausgelesen werden können.

Timeouts

Abschließend sollte noch auf die Timeouts der Zustände eingegangen werden. Dabei handelt es sich um Zeiten, nach denen automatisch der Zustand geändert wird. Dies kann notwendig sein, um z. B. einen Deadlock zu verhindern. Zustände, die von der Applikation verändert werden (z. B. `Close Wait`), besitzen keine Timeouts; es könnte sein, dass die Applikation absichtlich keine Daten übertragen will. Die anderen Zustände besitzen beim asynchronen IEEE 1394-Socket auch keine expliziten Timeouts, da ein Datenpaket aufgrund des Transaction-Layers entweder sein Ziel erreicht, oder der Sender verständigt wird, dass es nicht angekommen ist. In diesem Fall wird der Timeout, der vom IEEE 1394-Bussystem vorgegeben ist, ausgenutzt. Tritt ein Busreset auf, wird eine Überprüfung aller vorhandenen IEEE 1394-Socket-Objekte vorgenommen. Sollte eines davon seinen Verbindungspartner verloren haben, wird dies bei der nächsten Aktion, die auf dieses Objekt ausgeführt wird, erkannt und der Applikation mit einem Fehlerstatus mitgeteilt.

4 Implementierung

Im Folgenden wird auf die durchgeführte Implementierung eingegangen. Da die Teile von Java und C bis auf die dazwischen liegende Schnittstelle unabhängig voneinander sind, werden sie auch unabhängig voneinander beschrieben. Weiters kann angemerkt werden, dass die Implementierung in Java komplett ist. Sie enthält alle drei vorgeschlagenen Arten der IEEE 1394-Sockets und ist auch für alle Betriebssysteme verfügbar. Bei der entwickelten Bibliothek in C wurde nur der asynchrone IEEE 1394-Socket implementiert und dieser auch nur für das Betriebssystem Linux. Durch die Modularisierung kann gewährleistet werden, dass für eine Implementierung auch für andere Betriebssysteme, nur wenige Teile zu verändern sind, da die betriebssystemabhängigen Teile speziell gekapselt wurden.

4.1 Implementierung in Java

In *Abschnitt 3.2.1* wurde dargelegt, warum es nicht möglich ist, die Klassen für die IEEE 1394-Sockets und den IEEE 1394-Datagram-Socket vom Java-Socket abzuleiten. Daher werden Kopien dieser Klassen verwendet. Es wurde darauf geachtete, dass der Aufbau vom Java-Socket und die dahinter liegende Strategie übernommen wurden. Das Interface nach außen wird durch die im Paket `at.ac.tuwien.ict.net` befindlichen Klassen `IEEE1394AsyncDatagramSocket`, `IEEE1394AsyncSocket`, `IEEE1394IsoSocket`, `IEEE1394AsyncServerSocket` sowie `IEEE1394IsoServerSocket` zur Verfügung gestellt. Die eigentliche Funktionalität der IEEE 1394-Sockets wird erst durch die jeweilige Socket-Implementierung in den Klassen `IEEE1394AsyncSocketImpl`, `IEEE1394IsoSocketImpl` und `IEEE1394AsyncDatagramSocketImpl` zur Verfügung gestellt. Da die gesamte Funktionalität der IEEE 1394-Sockets bereits in der C-Bibliothek implementiert ist, müssen die genannten Socket-Implementierungen die Funktionsaufrufe nur an diese Bibliothek, mittels der Klasse `IEEE1394SocketAPI`, weiter reichen.

4.1.1 IEEE 1394-Socket

Die externe Schnittstelle der IEEE 1394-Sockets wurde bereits in der Schnittstellendefinition behandelt (*siehe Abschnitt 3.2.1*). Daher wird hier nur mehr auf die internen Klassen eingegangen.

IEEE1394SocketImpl

Die Implementierung der IEEE 1394-Socket-Implementierung in der Klasse `IEEE1394SocketImpl` unterscheidet sich wesentlich von der Klasse `SocketImpl` von Java-Socket. Sie enthält Methoden wie z. B. `bind`, `listen`, `accept`, `connect`, `available` und `close`. Da all diese Funktionen in der Definition des Sockets in C vorhanden sind, wird jeder Methodenaufruf direkt an die Instanz der Klasse `IEEE1394SocketAPI` weiter gegeben. Wird eine Instanz der Klasse `IEEE1394SocketImpl` erstellt, erfolgt gleichzeitig ein Aufruf der Funktion `socket`. Dadurch wird in C ein Objekt des gewünschten IEEE 1394-Sockets erstellt und die zugehörige Socket-ID retourniert. Diese wird in der Instanz der erstellten Klasse `IEEE1394SocketImpl` abgelegt. Dadurch kann auch ausgehend von Java, das zugehörige Socket-Objekt von C aufgerufen werden. Daher muss für alle Aufrufe einer Methode der Instanz der Klasse `IEEE1394SocketAPI` diese Socket-ID mit übergeben werden.

Die beiden Klassen `IEEE1394AsyncSocketImpl` und `IEEE1394IsoSocketImpl` sind eine Ableitung der Klasse `IEEE1394SocketImpl`. In ihnen sind nur die Konstruktoren und die Methoden für das Erhalten der Streams implementiert. Dies ist notwendig, da für die unterschiedlichen Typen von IEEE 1394-Sockets unterschiedliche Streams verwendet werden. Im Detail handelt es sich dabei um die Methoden `getInputStream` und `getOutputStream`.

IEEE1394SocketAPI

Die Klasse `IEEE1394SocketAPI` wurde bereits in der Schnittstellendefinition beschrieben. Sie stellt die Schnittstelle zwischen den IEEE 1394-Sockets in Java und jenen in C dar. Daher enthält sie all jene Methoden, die von den IEEE 1394-Sockets in C zur Verfügung gestellt werden. Damit Methoden dieser

Klasse aufgerufen werden können, muss eine Instanz dieser Klasse existieren. Diese kann mit der statischen Methode `getInstance` erhalten werden. Genau genommen wird eine Instanz der Klasse `IEEE1394StreamSocketAPI` erzeugt. Das stellt jedoch keinen Unterschied dar, da diese von der ersten Klasse abgeleitet ist, außer dass diese Instanz auch im Paket `at.ac.tuwien.ict.io` verwendet werden kann. Da für das gesamte System eine Instanz ausreichend ist, wird diese in einer statischen Variable in der Klasse `IEEE-1394StreamSocketAPI` abgelegt und bei jedem Aufruf retourniert.

Da diese Klasse das Bindeglied zwischen Java und C darstellt, ist sie auch dafür verantwortlich, dass die Bibliothek in den Speicher geladen wird. Dazu wird beim Erstellen der Instanz die Methode `loadLibrary` aus der Klasse `java.lang.System` aufgerufen, um die Bibliothek `IEEE1394Socket` zu laden. Würde das Laden dieser Bibliothek fehlschlagen, kommt es zu einer Exception. Es kann aber weiterhin versucht werden, eine Instanz dieser Klasse zu erstellen. Es gäbe sicher auch die Möglichkeit, alle Methoden dieser Klasse statisch zu definieren. Dann müsste keine Instanz davon erstellt werden. Es würde aber, wenn das Laden der Bibliothek fehlschlägt, keine Möglichkeit geben, diese nochmals zu laden, da der statische Code bereits ausgeführt wurde. Wird nämlich z. B. der aufgetretene Fehler behoben, und die Bibliothek kann nun doch geladen werden, kann bei der Variante mit der Instanz diese erstellt werden. In diesem Fall können schließlich die IEEE 1394-Sockets verwendet werden. Bei der statischen Implementierung wäre dies jedoch nicht der Fall.

In der Klasse `IEEE1394SocketAPI` sind weiters alle notwendigen Konstanten-definitionen enthalten. Dadurch kann sichergestellt werden, dass sie sowohl in Java als auch in C korrekt verwendet werden. Dabei hilft die Benutzung des Tools `javah`. Dadurch wird automatisch die zugehörige Header-Datei mit den benötigten Konstanten für C erzeugt. Weiters nehmen alle Methodenaufrufe eine Kontrolle des Rückgabewertes vor und lösen bei einem Fehler automatisch eine entsprechende Exception aus.

IEEE1394Address

Die Implementierung der Klasse `IEEE1394Address` wurde so gewählt, dass eine Instanz über einzelne Parameter (z. B. `GUID` und Port) erstellt werden kann, aber auch ausgehend von einer `socketaddr`-Struktur. Ebenso kann aus der IEEE1394Address wieder eine `socketaddr`-Struktur gewonnen werden. Diese Funktionalität ist beim Übergang in ihn die Bibliothek von C sehr wertvoll.

IEEE1394GUID

Eine Instanz dieser Klasse enthält nur den Wert einer `GUID`. Dieser Wert könnte sicherlich auch als long-Zahl abgespeichert werden. So bietet sie jedoch den Komfort, dass eine `GUID` von beliebiger Herkunft, wie z. B. aus einer `node_ID` oder einer `GUID` als long erstellt werden kann. Ebenso kann diese `GUID` wieder in alle Formen zurück konvertiert werden. Außerdem übernimmt diese Klasse automatisch die Überprüfung der Gültigkeit der beinhaltenden `GUID`. Ist die gewünschte `GUID` nicht am IEEE 1394-Bus verfügbar, wird eine `IEEE1394-InvalidGUIDException` ausgelöst.

IEEE1394Status

Eine Instanz dieser Klasse wandelt die Fehlercodes des AL1394 in den zugehörigen Text um. Tritt ein Fehler in einer AL1394-Funktion auf, wird dieser Wert im Speicher abgelegt. Mittels der Methode `getLast1394Status` der Klasse `IEEE1394SocketAPI` kann dieser Fehlerwert abgefragt werden. Wird eine Instanz dieser Klasse ohne Parameter erstellt, wird automatisch der letzte Fehlerwert geladen. Diese Klasse dient speziell der Entwicklung.

4.1.2 IEEE 1394-Datagram-Socket

Der asynchrone IEEE 1394-Datagram-Socket wurde ebenfalls bereits in der Schnittstellendefinition behandelt. Die Klasse `IEEE1394AsyncDatagram-Socket` ist eine Kopie der Klasse `DatagramSocket` von Java, ausgenommen den Konstruktoren. Ebenso wie bei den IEEE 1394-Sockets gilt auch hier, dass die Funktionalität in der IEEE 1394-Datagram-Socket-Implementierung ent-

halten ist. Dabei handelt es sich um die Klasse `IEEE1394AsyncDatagram-SocketImpl`. Sie gibt die Funktionsaufrufe direkt an die Instanz der Klasse `IEEE1394SocketAPI` weiter. Um Pakte übertragen zu können, muss ein Datagram-Paket implementiert werden. Dabei handelt es sich um die Klasse `IEEE1394AsyncDatagramPacket`. Sie enthält bereits die Funktionalität für alle IEEE 1394-Paket-Typen. Von dieser Klasse werden drei weitere für die unterschiedlichen Übertragungsarten abgeleitet (`read`, `write` und `lock`). Diese schränken den vorhandenen Funktionsumfang auf den jeweils benötigten ein. Diese Vorgehensweise wurde gewählt, da somit für das Empfangen eines Paketes das allgemeine angegeben werden kann. Damit steht die Möglichkeit offen, dass z. B. auf alle drei Arten von Pakete gewartet werden kann. Würde die allgemeine Klasse jedoch nicht den gesamten Funktionsumfang enthalten, würde es schwierig werden, darin z. B. ein `lock`-Paket unter zu bringen, da dieses drei Datenbereiche im Gegensatz zu read und write benötigt. Bezüglich der Klasse `IEEE1394SocketAPI` wird auf die Implementierung der IEEE 1394-Sockets in *Abschnitt 4.1.1* verwiesen, da die API alle Funktionen für die IEEE 1394-Sockets wie auch für den IEEE 1394-Datagram-Socket abdeckt.

4.1.3 Fehlerbehandlung

Bei den IEEE 1394-Sockets werden alle Exceptions von der `java.io.SocketException` abgeleitet. Da diese wiederum von der `java.io.IOException` abgeleitet ist, kann diese auch bei den Streams ausgelöst und weitergegeben werden. Als Basis-Exception für die IEEE 1394-Sockets wird die Klasse `IEEE1394SocketAPIException` verwendet, die sich wiederum von der `SocketException` ableitet. Damit wird angedeutet, dass jeder aufgetretene Fehler in der IEEE 1394-Socket API entstanden ist, da dort die gesamte Funktionalität der IEEE 1394-Sockets implementiert ist. Um jedoch den Fehler besser einschränken zu können, wurden von dieser weitere abgeleitet, die spezifisch auf einen Fehler hinweisen. Bei der `IEEE1394IOException` handelt es sich um einen Fehler bei einer Lese- oder Schreiboperation am IEEE 1394-Adapter. Die `IEEE1394AddressException` tritt auf, wenn eine falsche Adresse angegeben wurde und eine `IEEE1394NameException` wenn eine

Adressumwandlung fehlschlägt. Die `IEEE1394BusresetCounterChanged-Exception` wird ausgelöst, wenn ein Busreset während einer Operation ausgelöst wurde. Normalerweise sollte diese Exception nicht auftreten, da die Implementierung in C bei einem Busreset die Funktion zu wiederholen versucht. Treten sozusagen jedoch nur mehr Busresets auf, kann es trotzdem zu dieser Exception kommen. Bei den Exceptions `IEEE1394InvalidChannelNumber-Exception`, `IEEE1394InvalidGUIDException`, `IEEE1394InvalidNode-IDException` oder `IEEE1394InvalidSocketIDExcpetion` ist der jeweilige Wert (Kanalnummer, `GUID`, `node_ID` oder Socket-ID) ungültig. Sollte kein IEEE 1394-Adapter angesprochen werden können, kommt es zu einer `IEEE1394NoAdapterPresentException`. Dieser Fehler kann auftreten, wenn kein Adapter im System vorhanden ist, oder der Adapter nicht angesprochen werden kann, da z. B. der Driver nicht geladen ist, oder der AL1394 nicht installiert ist.

In der Klasse `IEEE1394SocketAPI` gibt es die Methode `checkStatus`. Sie untersucht den Rückgabewert einer externen Funktion von C auf einen Fehler; diese sind durch einen negativen Wert gekennzeichnet. Ist ein Fehler aufgetreten, löst diese Methode eine entsprechende Exception aus. Durch diese Methode kann auf keinen Fall darauf vergessen werden, einen Fehler abzufragen. Somit kann sich ein eigentlich bereits bekannter Fehler nicht unerkannt weiter verbreiten.

4.2 Implementierung in C

Die Implementierung in C ist zumindest teilweise betriebssystemabhängig. Dies lässt sich nicht umgehen, da auf Funktionen des Betriebssystems zurückgegriffen werden muss. Vor allem bei der Synchronisation oder beim Datenaustausch zwischen parallelen Prozessen muss auf das Betriebssystem zurückgegriffen werden. Um die Implementierung trotzdem möglichst leicht auch an andere Betriebssysteme anpassen zu können, wurden solche Aufrufe in einzelnen Modulen gekapselt. Dabei handelt es sich um die Module Kommunikationsmodul und Tools. Das Kommunikationsmodul ist von den

Funktionen der IEEE 1394-API abhängig, welche wiederum von System zu System unterschiedlich sein können. Das Modul Tools enthält betriebssystem-abhängige Aufrufe für:

- Gemeinsame Speicher für parallele Prozesse durch das shared memory,
- Synchronisation durch Semaphore,
- Signalisieren von Zuständen mittels message queue und
- Lese-Queues mittels named pipe.

4.2.1 IEEE 1394-Socket-Core

Im IEEE 1394-Socket-Core wird bereits die gesamte Funktionalität für alle Arten der IEEE 1394-Sockets zur Verfügung gestellt. Er bedient sich dabei vor allem des Kommunikationsmoduls, das ihm den Zugriff über den AL1394 auf den IEEE 1394-Adapter ermöglicht. Die Informationen über die IEEE 1394-Sockets werden in einer verketteten Liste abgelegt. Die Listenelenelemente sind durch die C-Struktur mit dem Namen `SocketData` definiert. Sie wurde speziell für die Implementierung der IEEE 1394-Sockets zusammengestellt und enthält z. B. die Socket-ID, die lokale Adresse, die verbundene Adresse und den Zustand in dem sich das zugehörige Socket-Objekt befindet. Es ist notwendig, all diese Informationen abzulegen, um gegebenenfalls darauf zurückgreifen zu können. Soll z. B. ein Datenpaket versendet werden, muss die Information verfügbar sein, wohin dieses Paket zu senden ist. Es muss aber auch die Möglichkeit geben, nachzusehen, wie viele Bytes in der Lesequeue stehen. Daher ist auch diese Information in der Struktur `SocketData` enthalten. Da diese verkettete Liste in einem shared memory abgelegt ist (*siehe Abschnitt 4.2.3*), wird ein Speicherbereich als Array mit 1024 Einträgen vom Typ `SocketData` angelegt. Die Verkettung der Liste wird über den Indexeintrag in der Liste erstellt. Dieser Weg wurde gewählt, da er einfacher zu implementieren ist, vor allem aufgrund des shared memory. Dadurch wird diese Implementierung der IEEE 1394-Sockets jedoch auf maximal 1024 gleichzeitige Verbindungen pro Knoten beschränkt. Weiters muss darauf geachtet werden, dass die Zugriffe auf die Daten der IEEE 1394-Socket-Objekte synchronisiert werden (*siehe Abschnitt 4.2.3*).

In C wurde nur der asynchrone IEEE 1394-Socket implementiert, daher wird nun im Detail darauf eingegangen. Die Kommunikation mit einem anderen Adapter erfolgt über einen definierten Speicherbereich. Der Adressbereich beginnt an der Adresse 0xabcd00000000h die in der Konstante `LISTENER_ADDRESS` abgelegt ist und umfasst eine Länge von 1024 Bytes, die in der Konstante `MAXPACKAGESIZE` definiert ist. Ein Paket hat somit nach den obigen Angaben eine maximale Größe von 1024 Bytes. Damit Pakete identifiziert werden können, müssen sie einen Protokoll-Header besitzen (*siehe Abbildung 3-6*). Anhand dieses Headers kann auch die Art des Paketes unterschieden werden; es gibt Verwaltungspakete und Datenpakete. Für den Datentransfer pro Paket bleibt aufgrund des Headers eine Größe von 1024 Bytes minus 24 Bytes also 1000 Bytes übrig. Will ein Objekt eines asynchronen IEEE 1394-Sockets ein Paket an einen anderen Partner senden, schreibt es dieses mittels der Funktion `AL1394_WriteNodeID` in den genannten Adressraum des zugehörigen Adapters. Die Adresse kann bei einem verbundenen IEEE 1394-Socket der Datenstruktur des Objektes dieses IEEE 1394-Sockets entnommen werden. Beim Verbindungsaufbau muss die Adresse des Partners dem initiierenden Datenpaket entnommen werden.

Damit empfangenen Pakete bearbeitet werden können, muss auf jedem Adapter, der die IEEE 1394-Sockets verwenden will, ein Socket-Listener installiert werden. Dieser Listener sorgt dafür, dass alle, in dem oben genannten Speicherbereich, empfangenen Pakete ausgelesen, ausgewertet und bearbeitet werden. Einem Datenpaket für einen verbundenen IEEE 1394-Socket werden die enthaltenen Daten entnommen und in die Lesequeue des zugehörigen Socket-Objektes geschrieben. Als Lesequeue werden named pipes verwendet (*siehe Abschnitt 4.2.3*). Bei einem Verwaltungspaket muss der Listener nach dem passenden Socket-Objekt suchen und dieses dementsprechend manipulieren. Dabei handelt es sich vor allem um die Zustandsübergänge vom Verbindungsmanagement aus *Abschnitt 3.4.2*.

Bevor der Socket-Listener gestartet wird, muss der oben genannte Adressbereich mittels der Funktion `AL1394_MapAddressRange` in den Besitz der IEEE 1394-Socket-API gebracht werden. Der Listener ist nun nichts anderes,

als ein Prozess, der darauf wartet, dass ein Packet auf diesen Speicherbereich geschrieben wird. Daher wird mittels des Systemaufrufes `fork` ein eigener Prozess generiert, der diese Wartefunktion durchführt. Das warten auf ein entsprechendes Ereignis erfolgt mittels der Funktion `AL1394_WaitAsyncEvent`. Tritt eines auf, retourniert diese Funktion und der Event kann mittels `AL1394_ExecAsyncEvent` behandelt werden. Durch diesen Aufruf wird die Behandlungsfunktion `handleListener` aufgerufen, die beim `MapAddressRange` mit übergeben wurde. Da in einem Computersystem jedoch mehrere IEEE 1394-Adapter vorhanden sein können, muss dieser Listener für jeden davon installiert werden. Um alle Listener verwalten zu können, müssen die zugehörigen Daten gespeichert werden. Da mehrere Prozesse darauf zugreifen, werden diese im shared memory (*siehe Abschnitt 4.2.3*) abgelegt. Der Einfachheit wegen wurde für diesen Speicher ein Array angelegt. Diese Vorgehensweise beschränkt die IEEE 1394-Sockets auf die Verwendung von maximal 8 Adaptern in einem System. Dieser Wert wurde in der Konstante `MAX_ADAPTERS` festgelegt. Das Installieren der einzelnen Listener wird bei der Initialisierung automatisch durchgeführt (*siehe Abschnitt 4.2.4*).

Write

Werden Daten auf ein IEEE 1394-Socket-Objekt geschrieben, kann aufgrund der gegebenen Socket-ID die zugehörige `SocketData` gefunden werden. Nun wird der Zustand dieses Objektes überprüft; nur im Zustand `Established` kann der gegebene Datenstrom versendet werden. Der `SocketData` können die `GUID` des Zieladapters und die für den Header notwendigen Werte für den lokalen Port und den Zielport entnommen werden. Nun wird der gegebene Datenstrom in einzelne Pakete der Größe 1000 Bytes zerlegt und mittels des Kommunikationsmoduls auf den gegebenen Adapter geschrieben. Am Ziel-Adapter wartet der Socket-Listener auf dem definierten Speicherbereich auf ankommende Datenpakete. Dieser liest das erhaltene Paket aus und bewertet den Protokoll-Header. Wird festgestellt, dass es sich um ein Datenpaket handelt, kann ein entsprechendes verbundenes IEEE 1394-Socket-Objekt aufgrund der Ports und der `GUID` des Senders gesucht werden. Die Ports können

dem Header entnommen werden, die `node_ID` den Informationen vom IEEE 1394-Bus des empfangenen Paketes. Da ein IEEE 1394-Socket jedoch über die `GUID` angesprochen wird, muss die `GUID` zu dieser `node_ID` ermittelt werden, was vom Kommunikationsmodul erledigt wird. Konnte ein zugehöriges Socket-Objekt gefunden werden, werden die Daten dem Paket entnommen und in die entsprechende Lesequeue gestellt. Die Applikation ist so aufgebaut, dass ein aufgetretener Fehler (z. B. das Socket-Objekt wird nicht gefunden oder die Queue ist voll) über das Acknowledge-Paket an den Sender zurückgemeldet wird. Da leider die derzeitig verfügbare Version des AL1394 nur die automatisierte Bestätigung unterstützt, bleibt ein solcher Fehler unerkannt! Die aktuelle Implementierung ist somit fehlerhaft, doch wird dieser Fehler toleriert, da er durch einen neuern AL1394 behoben wird. Es bestünde die Möglichkeit, diesen Fehler z. B. mit dem Versenden eines eigenen Bestätigungspaketes zu eliminieren. Doch würde dadurch der Datentransfer am Bus unnötigerweise steigen. Der Fehler manifestiert sich dadurch, dass diese Datenpakete verloren gehen.

Read

Die Implementierung vom `read` ist wesentlich einfacher als jene vom `write`. Beim Aufruf von `read` wird aufgrund der Socket-ID das zugehörige Socket-Objekt und somit deren `SocketData` gesucht. Dieser kann die Lesequeue entnommen werden. Darauf wird schließlich ein Read-Befehl ausgeführt, der erst zurückkehrt, wenn Daten von der Queue gelesen werden können. Bevor dieser Befehl ausgeführt werden kann, muss jedoch überprüft werden, ob sich das Objekt des IEEE 1394-Sockets im Zustand `Established` befindet; in diesem Fall dürfen die Daten ausgelesen werden. Befindet sich dieses Objekt im Zustand `Close Wait`, es handelt sich somit um eine halb geschlossene Socket-Verbindung, muss überprüft werden, ob Daten in der Lese-Queue verfügbar sind. Auch in diesem Fall können vorhandene Daten ausgelesen werden, andernfalls muss der Anwender darauf hingewiesen werden, dass die Socket-Verbindung geschlossen wird. Befindet sich das Socket-Objekt in keinem dieser beiden Zustände, darf kein Lese-Befehl ausgeführt werden.

Accept

Beim Aufruf der Funktion `accept` ist nichts anderes zu tun, außer darauf zu warten, dass ein anderer Partner versucht eine Verbindung zu dem gegebenen Socket-Objekt aufzubauen. Daher wird nur der Zustand dieses Objektes auf `Listen` verändert, und es wird auf eine Nachricht über die Signalisierung gewartet (*siehe Abschnitt 4.2.3*). Diese zeigt entweder an, dass eine Verbindung aufgebaut wurde, oder dass ein Fehler aufgetreten ist.

Versucht ein anderer IEEE 1394-Socket eine Verbindung aufzubauen, wird das Protokoll für den Verbindungsaufbau (*siehe Abbildung 3-7*) abgewickelt; dafür sorgt der Socket-Listener. Konnte die Verbindung etabliert werden, wird mittels der Signalisierung eine Nachricht an den wartenden Prozess gesendet, womit die Funktion `accept` fortgesetzt wird. Diese erstellt ein neues Objekt eines IEEE 1394-Sockets, mit den Daten des verbundenen Partners. Das IEEE 1394-Socket-Objekt, auf das die Funktion `accept` durchgeführt wurde, wird in den Zustand `Passive` zurückgesetzt. Die Funktion `accept` retourniert abschließend die Socket-ID des neuen verbundenen Socket-Objektes, das anschließend für die Kommunikation verwendet werden kann.

Connect

Beim Aufruf der Funktion `connect` wird ein `SYN`-Paket an den gewünschten Verbindungspartner gesendet. Danach wird darauf gewartet, ob die Verbindung etabliert werden konnte oder nicht. Daher wird diese Funktion in eine Warteposition versetzt. Die Antwort auf diesen Verbindungswunsch empfängt der Socket-Listener und sorgt dafür, dass das Protokoll für den Verbindungsaufbau (*siehe Abbildung 3-7*) abgehandelt wird. Ist dieser beendet, wird der wartende Prozess mittels einer Nachricht über die Signalisierung (*siehe Abschnitt 4.2.3*) darüber informiert, dass der Verbindungsaufbau abgeschlossen ist. Aufgrund dieser Nachricht wird die Funktion `connect` weiter fortgeführt. Anhand des Zustandes, in dem sich das Socket-Objekt befindet, kann überprüft werden, ob der Verbindungsaufbau erfolgreich war oder nicht. Das Resultat wird schließlich dem Anwender mitgeteilt.

Close

Die Funktion `close` ist im Gegensatz zu `connect` und `accept` nicht blockierend. Will ein Anwender eine Verbindung schließen, teilt er dies dem Socket-Objekt mittels des Aufrufs dieser Funktion mit. Er will aber nicht damit aufgehalten werden, ob der andere Verbindungspartner tatsächlich den Abbruch akzeptiert. Sollte dieser z. B. in einem Deadlock verharren, würde dies auch bedeuten, dass der schließende Prozess endlos warten würde. Beim Aufrufen der Funktion `close`, wird daher ein `FIN`-Paket an den Verbindungspartner gesendet. Danach retourniert diese Funktion. Der Prozess des Schließens wird dann im Hintergrund vom Socket-Listener nach dem in *Abschnitt 3.4.2* definierten Protokoll (*siehe Abbildung 3-8*) abgewickelt. Eine Objekt eines Socket, das sich im Prozess des Schließens befindet, kann erst dann wieder verwendet werden, wenn dieser Prozess beendet ist, und es sich somit wieder im Zustand `Closed` befindet.

4.2.2 Kommunikationsmodul

Das Kommunikationsmodul stellt eine Erweiterung des AL1394 dar. Auf diese Funktionen wird daher an dieser Stelle nicht eingegangen. Es sollte jedoch auf die Umrechnung zwischen `node_ID` und `GUID` im Detail eingegangen werden. Dazu wird eine Tabelle mit 64 Einträgen (abgelegt in der Konstante `MAX_RESOLVE_ITEMS`) erstellt. Dieser Wert wurde deshalb gewählt, da maximal 64 Adapter an einem IEEE 1394-Bus angeschlossen werden können, und derzeit keine Bridges existieren. Jeder Eintrag enthält eine `node_ID`, die zugehörige `GUID` und die Nummer des Adapters, auf dem dieser Knoten zu finden ist. Diese Tabelle wird in einem shared memory (*siehe Abschnitt 4.2.3*) abgelegt, da mehrere Prozesse diese Information auslesen müssen. Weiters wird auch die Anzahl der belegten Felder, sprich die Anzahl der angeschlossenen Adapter, gespeichert. Soll die `node_ID` zu einer `GUID` ermittelt werden, wird die `GUID` in der Tabelle gesucht und die zugehörige `node_ID` ausgelesen. Außerdem wird auch die Nummer jenes Adapters mitgeteilt, an dem dieser Knoten angeschlossen ist. Selbiges kann auch umgegehrt erfolgen: man suche eine `node_ID` in der Tabelle und lese die zugehörige `GUID` aus.

Damit diese Tabelle immer aktuell ist, muss sie nach jedem Busreset aktualisiert werden. Aus diesem Grund wird ein Busreset-Listener installiert. Dies erfolgt automatisch beim Laden der Bibliothek (*siehe Abschnitt 4.2.4*). Dabei wird mittels `AL1394_RegisterNotification` das Interesse an einem Busreset dem IEEE 1394-Bus mitgeteilt. Damit der Listener immer aktiv ist, wird mittels `fork` ein neuer Prozess erstellt, der durch Aufruf des Befehls `AL1394_WaitAsyncEvent` auf das Auftreten eines Busresets wartet. Ist dies der Fall, wird durch den Aufrufen der Funktion `AL1394_ExecAsyncEvent` der aufgetretene Event behandelt. Aufgrund dessen wird schlussendlich die Funktion `handleBusresetEvent` aufgerufen, die beim Aufruf von `AL1394_-RegisterNotification` übergeben wurde. Da der AL1394 derzeit nur die Beendigung eines Busreset mitteilt, kann nur dieses Event berücksichtigt werde [ALC2 pp23]. Bei einer späteren Implementierung des AL1394 sollte es auch möglich sein, ein Event für den Beginn eines Busresets zu erhalten. Dies wäre vorteilhaft für die IEEE 1394-Socket-Implementierung; damit wäre es möglich, den Zugriff auf die Umrechnungstabelle während des gesamten Busreset zu sperren. Der Zugriff würde erst wieder nach Beendigung des Busresets und der Aktualisierung der Tabelle erlaubt werden. Da dies der aktuelle AL1394 jedoch nicht unterstützt, muss es genügen, die Tabelle nach Beendigung des Busresets zu sperren, sie zu aktualisieren und dann wieder frei zu geben. Nachteil dieser Variante ist, dass bei einem Busreset eine IEEE 1394-Aktion (z. B. `WriteNodeID`) mit einer Fehlermeldung zurückkehrt und somit wiederholt werden muss. Ist jedoch der Zugriff auf einen Adapter während eines Busresets gesperrt, könnte es nicht zu diesem Fehler kommen.

Die Daten für die Umrechnungs-Tabelle werden der topology map (*siehe Abschnitt 1.2.2*) entnommen. An den Einträgen dieser Liste kann erkannt werden, ob ein Adapter angeschlossen ist, aktiv ist und wie seine `node_ID` lautet. Mit der somit gegebenen `node_ID` kann der zugehörige Adapter angesprochen und seine `GUID` aus dessen Config-Rom mittels der Funktion `ReadNodeID` an der Stelle `0xFFFFF000040Ch` ausgelesen werden. Diese Daten können nun in die Umrechnungstabelle eingefügt werden. Da nicht alle IEEE 1394-Adapter eine topology map unterstützen, wurden für diesen Fall zwei andere Möglichkeit implementiert, um die Daten für die Tabelle zu

erhalten. Bei der ersten wird versucht, diese topology map vom IRM (Isochronous Resource Manager) zu laden (*siehe Abschnitt 1.2.6*). Dieser Knoten muss unter anderem eine topology map besitzen. Kann der IRM gefunden werden und auch diese Liste gelesen werden, wird diese als Ermittlungsbasis für die Umrechnungstabelle herangezogen. Bei der zweiten Möglichkeit wird versucht, für jede mögliche physical_ID am lokalen Bus, die GUID des Adapters auszulesen. Ist das Auslesen erfolgreich, wird die Wertekombination in die Tabelle eingefügt. Diese Vorgehensweise dauert jedoch wesentlich länger, da ein Fehlzugriff vom IEEE 1394 erst nach einem Timeout erkannt werden kann. Mit dieser Methode ist jedoch gewährleistet, dass die Umrechnungstabelle in jedem Fall gefüllt ist. Der letzte Fall kann z. B. dann vorkommen, wenn kein IRM am Bus verfügbar ist.

4.2.3 Tools

Im Folgenden wird auf die wichtigsten Funktionen der Tools eingegangen. Funktionen des Betriebssystems werden in der IEEE 1394-Socket-Implementierung nicht direkt verwendet, sondern im Modul Tools gekapselt. Dadurch muss z. B. bei einem Wechsel des Betriebssystems nicht die gesamte Applikation umschreiben werden. Weiters werden einige Funktionen auch in einer einfacheren Form zur Verfügung gestellt. Oft besitzen die Betriebssystemfunktionen einen Umfang, der bei dieser Implementierung nicht benötigt wird, und somit eingeschränkt werden kann. Dadurch ist die Verwendung dieser Funktionen weniger kompliziert. Im Folgenden wird auf die wichtigsten Teile der Tools eingegangen:

Shared memory

Die Implementierung der IEEE 1394-Sockets soll und wird in einem Multiprozesssystem eingesetzt. Daher ist es erforderlich, dass gewisse Daten allen Prozessen zur Verfügung stehen. Dabei handelt es sich um die gesamte Umrechnungstabelle zwischen node_ID und GUID, aber auch um die notwendigen Daten aller vorhandenen IEEE 1394-Socket-Objekte, wie z. B. deren Zustände oder Adressen. Da jeder Prozess jedoch in seinem eigenen

Speicherbereich abgewickelt wird, kann nicht so einfach auf die Daten eines anderen zugegriffen werden. Würde dies versucht werden, würde das Betriebssystem mit einem Speicherzugriffsfehler reagieren. Daher müssen diese Bereiche in allgemein zugängliche Bereiche, shared memory, ausgelagert werden. Bei der Implementierung der IEEE 1394-Sockets wurde somit für die Umrechnungstabelle ein shared memory, aber auch für die Daten der IEEE 1394-Socket-Objekte angelegt. Eine Unterteilung musste erfolgen, da beide Datenbereiche in unterschiedlichen Modulen Verwendung finden und somit autark sein sollten; damit kann garantiert werden, dass jedes Modul unabhängig vom anderen implementiert, getestet, verbessert und verwendet werden kann. Mit der Funktion `shmget` kann ein neuer shared memory generiert, oder ein bestehender geöffnet werden. Dieser Speicher muss, um verwendet werden zu können, mit dem Befehl `shmat` in den Speicherbereich des aktuellen Prozesses gelinkt werden. Diese Funktion retourniert einen Zeiger, mit dem dieser Speicher angesprochen werden kann. Wird der Zugriff nicht mehr benötigt, kann er wieder mit dem Befehl `shmdt` vom Prozess getrennt werden. Um einen shared memory löschen zu können, wird der Befehl `shmctl` verwendet.

Synchronisation

Da es bei der IEEE 1394-Socket-Implementierung Datenbereiche gibt, auf die parallele Prozesse zugreifen können, muss der Zugriff darauf geregelt werden; sprich es muss eine Synchronisation implementiert werden. Dies gilt vor allem für die Umrechnungstabelle zwischen `node_ID` und `GUID`, aber auch für die gesamten Daten der IEEE 1394-Socket-Objekte (z. B. Zustand, Adressen).

Prinzipiell werden zur Synchronisation paralleler Prozesse Semaphore verwendet. Da jedoch auf den gemeinsamen Speicherbereichen sowohl geschrieben als auch gelesen werden kann, muss die beste Möglichkeit zur Synchronisation ausgewählt werden. In der Literatur kann dazu im Grunde nur eine Lösung gefunden werden, nämlich das „Readers/Writers Problem“ [OS1 pp237-242]. Es wird dabei mehreren Prozessen gleichzeitig erlaubt, den gemeinsamen Speicherbereich zu lesen, aber jeweils nur einem Prozess, diesen Bereich exklusiv zu beschreiben (zu diesem Zeitpunkt hat nur dieser

eine Prozess Zugriff auf diesen Bereich). In dieser Arbeit wurde weiters eine Priorisierung des Writers vorgenommen. Dies bedeutet, dass die schreibenden Prozesse bevorzugt werden. Sobald sich ein schreibender Prozess anmeldet, erhält kein weiterer Prozess Zugriff auf diesen Bereich. Der Schreiber muss jedoch solange warten, bis alle aktiven Leser diesen Bereich freigegeben haben. Prinzipiell wäre es auch denkbar, dass solange ein lesender Prozess aktiv ist, andere lesende Prozesse erlaubt werden würden, und ein schreibender nur dann, wenn kein lesender zugreifen würde. Dabei würde von einer Priorisierung der Reader gesprochen werden, was aber in diesem Fall nicht Verwendung findet. Vor allem beim Auftreten eines Busresets ist es wichtig, dass unbedingt die neuen Adressdaten (`node_ID`) in der Umrechnungstabelle erneuert werden. Daher muss unbedingt der schreibende Prozess vor allen anderen lesenden ausgeführt werden. Ein anderes Beispiel ist, wenn eine IEEE 1394-Socket-Objekt geschlossen wird, muss das Schließen dieses Objektes Vorrang gegenüber dem Auslesen von Daten aus den IEEE 1394-Socket-Objekten haben. Ansonsten könnte es z. B. sein, dass ein sich im Schließen befindliches IEEE 1394-Socket-Objekt nicht als solche erkannt wird.

Die Synchronisation ist in den Funktionen `PReader`, `VReader`, `PWriter` und `VWriter` untergebracht. Die Funktionen unterscheiden sich in P- und V-Operation. Weiters gibt es jede Funktion für das Auslesen (Reader) und das Beschreiben (Writer). Will ein Prozess einen Zugriff durchführen, ruft er eine P-Funktion auf. Diese wartet so lange, bis der Zugriff gestattet ist. Ist der Prozess mit seinem Zugriff fertig, teilt er dies dem System mit dem Aufruf einer V-Funktion mit. Der Algorithmus für diese Funktionen kann [OS1 pp237-242] entnommen werden.

Signalisierung

Es ist notwendig, einen Prozess anzuhalten und erst dann wieder fortzufahren, wenn ein anderer Prozess seine Arbeit erledigt hat. Dies ist vor allem beim Verbindungsaufbau notwendig. Z. B. versendet ein `connect` direkt ein `FIN`-Paket und wartet dann so lange, bis der Verbindungsaufbau durchgeführt ist. Ein `accept` ändert den Zustand des Socket-Objektes auf `Passive` und wartet ebenfalls bis die Verbindung aufgebaut ist. Der Aufbau der Verbindung wird

jedoch von dem Prozess des Socket-Listeners durchgeführt, da dieser alle Pakete über den IEEE 1394-Bus erhält. Daher muss eine Möglichkeit geschaffen werden, einen Prozess anhalten zu können und erst dann wieder fortzusetzen, nachdem ein anderer Prozess ihm dies mitteilt. Für diese Aufgabe wurden die message queues von Linux verwendet. Diese ermöglichen, dass kurze Informationen zwischen Prozessen ausgetauscht werden können. Mittels der Funktion `msgget` kann ein Zugriff auf eine message queue erhalten, mittels der Funktion `msgrcv` auf eine Nachricht gewartet und mittels der Funktion `msgsnd` eine Nachricht versendet werden. Eine Nachricht muss mit einer `message type` vom Typ `long` beginnen und kann danach beliebig definiert werden. In dieser Implementierung folgt ein weiterer Wert, der über die Art der Nachricht Auskunft gibt (z. B. Abbruch, Verbindung hergestellt, es ist ein Fehler aufgetreten). Die `message type` wird dazu verwendet, um eine Nachricht an einen speziellen anderen Prozess senden zu können. Beim `msgrcv` muss eine Zahl angegeben werden; ist diese positiv, wird so lange gewartet, bis eine Nachricht mit genau dieser `message type` versendet wurde. Bei negativen Zahlen werden alle Nachrichten mit einer `message type` bis zu diesem Wert empfangen. In dieser Implementierung werden nur positive Zahlen verwendet, da ein Prozess darauf wartet, dass genau ein anderer, nämlich der Socket-Listener, seine Aufgabe abgeschlossen hat. Muss ein Socket-Objekt auf ein Ereignis warten, kann dieses mittels `getFreeMessageWaitID` eine freie `message type` erhalten. Diese wird in dem zugehörigen IEEE 1394-Socket-Objekt gespeichert, anschließend wird auf dessen Auftreten gewartet. Hat der Socket-Listener seine Arbeit abgeschlossen, kann er dem zugehörigen IEEE 1394-Socket-Objekt die zu sendende `message type` entnehmen und damit den wartenden Prozess wieder aktivieren.

Lese-Queue

Datenpakete, die der Socket-Listener empfängt, stellt er in eine Queue, die dem zugehörigen IEEE 1394-Socket-Objekt zugeordnet ist (*siehe Abbildung 1-6*). Aus dieser werden durch den Aufruf von `read` die Daten ausgelesen. Für die Implementierung wurde eine named pipe gewählt, da sie die benötigte Funktionalität zur Verfügung stellt: Ein Prozess kann darauf schreiben, und ein

anderer diese Daten davon wieder auslesen. Das Lesen und Schreiben erfolgt bereits synchronisiert und muss daher nicht zusätzlich durch Semaphore abgesichert werden. In Linux ist es sinnvoll, named pipes im Verzeichnis `/tmp` abzulegen, da dort jeder Prozess vollen Zugriff hat. Weiters erhält sie einen Namen mit dem Präfix `.ieee1394socket_` und anschließend folgt die Kennung des zugehörigen IEEE 1394-Socket-Objektes (Socket-ID). Damit kann jede Lesequeue eindeutig einem IEEE 1394-Socket-Objekt zugeordnet werden.

Die named pipe wird erstellt, wenn der Verbindungsaufbau des zugehörigen IEEE 1394-Socket-Objektes erfolgt ist, und wird nach dem Aufrufen des Befehls `close` wieder gelöscht. Um auf eine named pipe zugreifen zu können, muss sowohl ein Leser, als auch ein Schreiber darauf geöffnet werden. Das Öffnen eines Lesers und Schreibers retourniert jedoch immer erst dann, wenn gleichzeitig ein Leser und ein Schreiber auf der named pipe verfügbar sind. Dies könnte Probleme verursachen, wenn Daten auf eine named pipe geschrieben werden sollen, aber noch kein Read-Befehl darauf ausgeführt wurde. Daher wird beim Erstellen ein Leser und ein Schreiber parallel, durch Zuhilfenahme eines weiteren Prozesses mit dem Befehl `fork,` geöffnet. Der Leser kann danach wieder geschlossen werden, der Schreiber bleibt jedoch immer aktiv. Beim Schreiben auf die named pipe wird genau dieser Schreiber verwendet, da das Öffnen so implementiert ist, dass es durch den Prozess des Socket-Listeners erfolgt. Dies ist erforderlich, da es nicht möglich ist, auf einen Filedescriptor zuzugreifen, der in einem andern Prozess erstellt wurde. Das Lesen erfolgt jedoch immer von einem anderen Prozess. Dies ist weiter nicht störend, da beim Aufruf der Funktion `read` jeweils ein neuer Leser auf die named pipe geöffnet wird, dieser die Daten ausliest, und dann wieder geschlossen wird. Soll ein IEEE 1394-Socket-Objekt geschlossen werden, muss zuerst der permanente Schreiber geschlossen werden. Dadurch wird ein möglicher aktiver Leser mit einer Fehlermeldung unterbrochen, da das Ende der named pipe erreicht wurde. Dadurch kann der Applikation mitgeteilt werden, dass das zugehörige IEEE 1394-Socket-Objekt geschlossen wurde.

Von einem IEEE 1394-Socket-Objekt kann mittels des Aufrufs der Funktion `getavailable` abgefragt werden, wie viele Bytes in der Lese-Queue stehen.

Es ist jedoch nicht möglich, von einer named pipe zu erfahren, wie viele Bytes in ihr enthalten sind. Daher muss bei jedem Schreiben auf eine named pipe und bei jedem Lesen mitgezählt werden, wie viele Bytes bearbeitet wurden. In dem Objekt des zugehörigen IEEE 1394-Sockets wird dann der Wert der aktuell verfügbaren Bytes abgelegt. Wird versucht, Daten von einem IEEE 1394-Socket-Objekt im Zustand `Close Wait` zu lesen, muss überprüft werden, ob Daten in der zugehörigen named pipe verfügbar sind, was anhand des Zählers ermittelt werden kann. Würde dies nicht erfolgen, würde das Öffnen eines Lesers so lange blockiert werden, bis wieder ein Schreiber aktiv wird. Ein schließendes IEEE 1394-Socket-Objekt würde jedoch keinen Schreiber mehr erstellen, und der eine geöffnete Schreiber wurde bereits beim Aufruf von `close` geschlossen. Daher käme es zu einem Deadlock. Sind Daten Verfügbar werden sie einfach ausgelesen. Im dem Fall, dass keine Daten verfügbar sind, wird die Fehlermeldung, dass dieses IEEE 1394-Socket-Objekt geschlossen wird, retourniert.

4.2.4 Initialisierung

Wird eine Bibliothek in den Speicher geladen, wird dabei eine spezielle Funktion aufgerufen. Dabei handelt es sich in Linux um die Funktion `_init`. Diese kann dazu verwendet werden, um benötigte Ressourcen zu initialisieren. Es muss beachtet werden, dass diese Funktion immer dann aufgerufen wird, wenn ein Prozess diese Bibliothek verwenden will. Dies bedeutet, wenn drei Prozesse diese Bibliothek verwenden, dass diese Funktion auch dreimal aufgerufen wird. Ob die Bibliothek bereits einmal geladen wurde, lässt sich leicht daran erkennen, ob der shared memory bereits existiert. Existiert dieser kann in diesem mitgezählt werden, wie oft die Bibliothek bereits in Verwendung ist. Wird eine Bibliothek nicht mehr benötigt, wird vom System die Funktion `_fini` aufgerufen. In dieser wird der Wert der aktiven Bibliotheken korrigiert. Daran kann erkannt werden, ob noch mindestens eine Bibliothek in Verwendung ist. Ist dies nicht der Fall, werden alle initialisierten Ressourcen frei gegeben, einschließlich der shared memory. Bei diesen beiden Funktionen ist zu beachten, dass sie in einem Bereich `extern "C"` zu implementieren sind; nur dadurch kann Linux diese Funktionen als solche erkennen.

Ressourcen die initialisiert werden sind:

- AL1394
- Shared memory für Socket-Objekte und Kommunikationsmodul
- Semaphore für die Synchronisation der Socket-Objekte und des Kommunikationsmoduls
- Message queue für die Signalisierung
- Umrechnungstabelle für `node_ID` zu `GUID`
- Socket-Listener
- Busreset-Listener

Alle Ressourcen werden beim ersten Laden der Bibliothek initialisiert, und wenn keine mehr aktiv ist, wieder frei gegeben. Eine Ausnahme davon bildet der AL1394. Dieser muss in jedem Prozess initialisiert und am Ende wieder freigegeben werden.

4.3 Test und Installation des IEEE1394-Socket

Einführend ist zu den Tests der erstellten Software im Rahmen dieser Diplomarbeit zu sagen, dass diese parallel zur Entwicklung durchgeführt wurden. Da die Software als Top-Down-Design entwickelt wurde, konnte diese Schritt für Schritt und immer parallel zur Entwicklung getestet werden. Beim Testen von Software ist jedoch immer im Hinterkopf zu behalten, dass keine Software auf Korrektheit hin getestet werden kann, sondern immer nur Fehler nachgewiesen werden können. Daher wird im Folgenden nur von der Funktionalität und nicht von Korrektheit oder Fehlerfreiheit gesprochen.

Als erstes wurden die Klassen der IEEE 1394-Sockets in Java erstellt. Die Klasse `IEEE1394SocketAPI` wurde jedoch nur soweit erstellt, dass sie Methodenaufrufe protokollierte und fiktive Werte für die Rückgabe erzeugte. Somit konnte bereits in diesem Stadium die Funktionalität der IEEE 1394-Sockets in Java nachgewiesen werden. Anschließend wurde die IEEE 1394-SocketAPI des Socket-Cores erstellt. Jeder Aufruf einer IEEE 1394-Socket-Funktion wurde mitprotokolliert, und es wurden ebenfalls fiktive Rückgabewerte

retourniert. In diesem Schritt war es möglich, die Funktionalität der erstellten JNI-Schnittstelle zu testen. Dabei wurde auf die korrekte Parameterumwandlung und Kontrolle der übergebenen Werte geachtet. Nun konnte die eigentliche Funktionalität der IEEE 1394-Sockets implementiert werden. Dabei wurden jedoch alle Aufrufe, die an den IEEE 1394-Adapter gingen wiederum durch fiktive Funktionen ersetzt, die deren Aufruf protokollierten und ebenfalls fiktive Rückgabewerte kreierten. Damit konnte gezeigt werden, dass die Logik der IEEE 1394-Sockets, im Speziellen die Verwaltung, funktionstauglich ist. Abschließend wurden die Funktionsaufrufe der AL1394-Funktionen integriert. Dabei wurden jedoch wiederum alle Funktionsaufrufe und wichtige Parameter mitprotokolliert, um etwaige Fehler erkennen und analysieren zu können.

Die Implementierung wurde so erstellt, dass es möglich ist, die Funktionen für die Protokollierung mitzukompilieren oder auch nicht. Es ist auch möglich, nur Teile der Protokollierung zu aktivieren. Dies ist speziell bei der Fehlersuche sehr vorteilhaft, da ansonsten das Protokoll sehr unübersichtlich werden kann. Vor allem da es sich bei der Implementierung der IEEE 1394-Sockets um eine Multiprozessapplikation handelt; einerseits werden bereits bei der Initialisierung der IEEE 1394-Socket-API zwei Prozesse pro IEEE 1394-Adapter gestartet (die Listener), andererseits können die Funktionen von mehreren Prozessen verwendet werden. In der Protokolldatei werden zwar das Datum, die Uhrzeit und die Prozessnummer mitgeschrieben, aber bei der Fülle von Protokollinformationen und Funktionsaufrufen, kann das Protokoll eine beträchtliche Länge erhalten.

4.3.1 Installation

Damit die Implementierung der IEEE 1394-Sockets gestestet werden kann, muss sie zuerst installiert werden. Zur Erstellung des C-Codes existiert ein `makefile`. Damit kann unter Linux mittels des Befehls `make` der Code kompiliert, gelinkt und installiert werden. Weiters kann im `makefile` die Protokolldatei angegeben werden, standardmäßig handelt es sich dabei um `/var/log/ieee1394socket.log`. Die Bibliothek wird in zwei Varianten erstellt. Bei der einen Variante ist die Protokollierung aktiv (`libIEEE1394-`

SocketDebug.so), bei der anderen nicht (libIEEE1394Socket.so). Beim Testen ist es sicherlich vorteilhaft, die Protokollierung zu aktivieren, um aufgetretene Fehler besser erkennen zu können. Da für das Protokollieren jedoch zusätzlicher Code erforderlich ist, und auch zwischendurch immer wieder auf das Dateisystem zugegriffen werden muss, ist diese Version der IEEE 1394-Sockets langsamer. Damit die IEEE 1394-Sockets für C erstellt werden können, muss außerdem der AL1394 installiert sein. Dieser benötigt zusätzlich das Paket raw1394, das wiederum einen passenden IEEE 1394-Treiber benötigt. Bei der Installation der IEEE 1394-Sockets in C wird die Bibliothek in das Verzeichnis /usr/local/lib kopiert. Bei der Installation sollte darauf geachtet werden, dass sie von einem Benutzer durchgeführt wird, der ausreichend Berechtigungen dafür besitzt; ein Benutzer mit zu geringen Berechtigungen wird nicht die Möglichkeit besitzen die entsprechenden Treiber und Module zu installieren und auch keine Schreibberechtigung im Verzeichnis /usr/local/lib besitzen.

Damit die IEEE 1394-Sockets in Java funktionieren, muss zuerst jener für C installiert werden. Es muss darauf geachtet werden, dass die benötigte Bibliothek von Java gefunden werden kann. Um die IEEE 1394-Sockets in Java zu verwenden kann entweder der gesamte Code der IEEE 1394-Sockets in Java in das Projekt inkludiert werden, oder es wird eine JAR-Datei der IEEE 1394-Sockets inkludiert. Dabei handelt es sich um eine komprimierte Datei im ZIP-Format, die all kompilierten Java-Dateien der IEEE 1394-Sockets enthält.

4.3.2 IEEE 1394-Socket-Commander

Diese Applikation wurde für Linux in C entwickelt, um die Zustände aller vorhandenen IEEE 1394-Socket-Objekte überwachen zu können. Sie gibt eine Liste aller aktiven Objekte und deren jeweiligen Zustand auf der Standard-Ausgabe stdout aus. Diese Applikation erleichtert das Auffinden von Fehlern in der Implementierung der IEEE 1394-Sockets, vor allem beim Verbindungs-aufbau und -abbau. Aber auch für die Entwicklung von Applikation, die die IEEE 1394-Sockets verwenden wollen, kann dieser IEEE 1394-Socket-Commander hilfreich sein, Fehler bei der Implementierung der Applikation zu erkennen. Er

ist nützlich, um erkennen zu können, ob z. B. ein IEEE 1394-Socket-Objekt richtig geschlossen wurde. Der IEEE 1394-Socket-Commander wird ebenfalls mittels dem `makefile` erstellt, und besitzt den Dateinamen `ieee1394SocketCommander`.

```
IEEE 1394-Socket Commander 1.0

socketID        state
4               ESTABLISHED
3               FIN_WAIT_2
2               CLOSE_WAIT
1               LISTEN
```

Abbildung 4-1 Beispielausgabe vom IEEE 1394-Socket-Commander

Der IEEE 1394-Socket-Commander ist eine textbasierte Applikation, die von einer Kommandozeile aufgerufen wird. Da er keine zusätzlichen Parameter erwartet, kann das Programm mittels des Namens `ieee1394Socket-Commander` gestartet werden. Danach wird die Liste aller IEEE 1394-Socket-Objekte mit deren Zuständen retourniert (*siehe ein Beispiel in Abbildung 4-1)*. Anhand dieser Liste kann z. B. festgestellt werden, welche Objekte auf einen Verbindungsaufbau warten, welche erstellt und welche geschlossen wurden.

4.3.3 Testapplikationen

Zusätzlich zur Protokollierung der Funktionen in einer Log-Datei wurden zwei Testapplikationen implementiert. Während der Implementierung wurden auch noch weitere Programme erstellt, diese dienten jedoch rein der Entwicklung und können nicht als Testapplikationen gewertet werden. So war es z. B. notwendig, Programme zu schreiben, die eine Verbindung öffnen um sie dann wieder zu schließen, oder ein Programm, das absichtlich eine Verbindung öffnt, die nicht geöffnet werden kann. Anhand dieser Programme, konnten gezielt einzelne Funktionen des asynchronen IEEE 1394-Sockets getestet und ausgebessert werden. Als Testapplikationen wurden zwei Beispiele aufgegriffen, die tatsächlich in der Praxis Verwendung finden. Anhand dieser konnte ein praktischer Test des asynchronen IEEE 1394-Sockets durchgeführt werden kann.

Versenden einer Datei

Es wird eine beliebige Datei, ihr Dateiname kann als Aufrufparameter übergeben werden, von einem Client auf einen Server versendet. Die Datenübertragung erfolgt dabei über den asynchronen IEEE 1394-Socket. Damit der Server weiß, wie die zu übertragende Datei heißt, wird zuerst der Name über die geöffnete asynchrone IEEE 1394-Socket-Instanz versendet. Der Server versucht nun, diese Datei zu erstellen; ist dies möglich, sendet er dem Client ein Paket mit dem Inhalt „OK“, ist es nicht möglich, wird eine Fehlermeldung wie z. B. „ungültiger Dateiname“ oder „Datei konnte nicht erstellt werden“ zurückgesendet. Ist ein Fehler aufgetreten, kann der Client die empfangene Fehlermeldung ausgeben und anschließend das Programm beenden. Hat er den Text „OK“ empfangen, beginnt er mit dem paketweisen Einlesen und paketweisen Versenden dieser Datei. Der Server liest die empfangenen Daten aus der asynchronen IEEE 1394-Socket-Instanz aus und hängt diese an die von ihm erstellte Datei an. Hat der Client das Ende der Datei erreicht, schließt er die Instanz. Damit kann auch der Server seine Instanz und auch die erstellte Datei schließen. In *Abbildung 4-2* ist die Applikation als Ablaufdiagramm dargestellt.

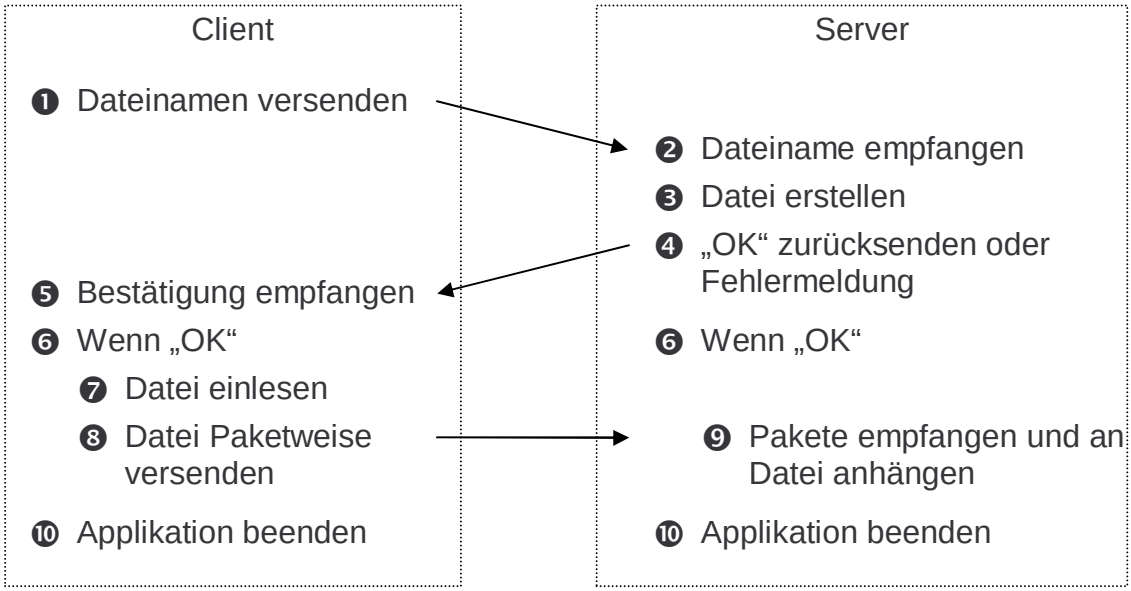

Abbildung 4-2 Datei versenden

Da die Implementierung des asynchronen IEEE 1394-Sockets Datenblöcke von 1 KB über den Bus versendet (wobei zu beachten ist, dass ein versendetes Datenpaket auch den Header enthält) wurde die Blockgröße im Client mit 2 KB

und im Server mit 555 Bytes gewählt. Durch diese unterschiedlichen Blockgrößen kann gezeigt werden, dass die aktuelle Implementierung unabhängig von den definierten Blockgrößen funktioniert, und es auch zu keinen Problemen mit blockübergreifenden Datenpaketen kommt. Zum Programmstart ist noch anzumerken, dass der Server vor dem Client zu starten ist, da der Server auf den Verbindungsaufbau wartet.

Kommunikationsprogramm

Als zweites Testprogramm wurde ein Kommunikationsprogramm ausgewählt. Es ermöglicht, Texte zwischen zwei Programminstanzen über den asynchronen IEEE 1394-Socket auszutauschen. Zuerst muss der Server gestartet werden. Dieser wartet darauf, dass ein Client die Verbindung zu ihm aufnimmt. Wurde mit einem Client eine Verbindung etabliert, verwenden beide das gleiche Interface, um Texte eingeben und an den anderen Partner versenden zu können. Wird eine der beiden Applikationen beendet, wird auch automatisch die Applikation des anderen Partners beendet. In *Abbildung 4-3* ist die Applikation als Ablaufdiagramm dargestellt.

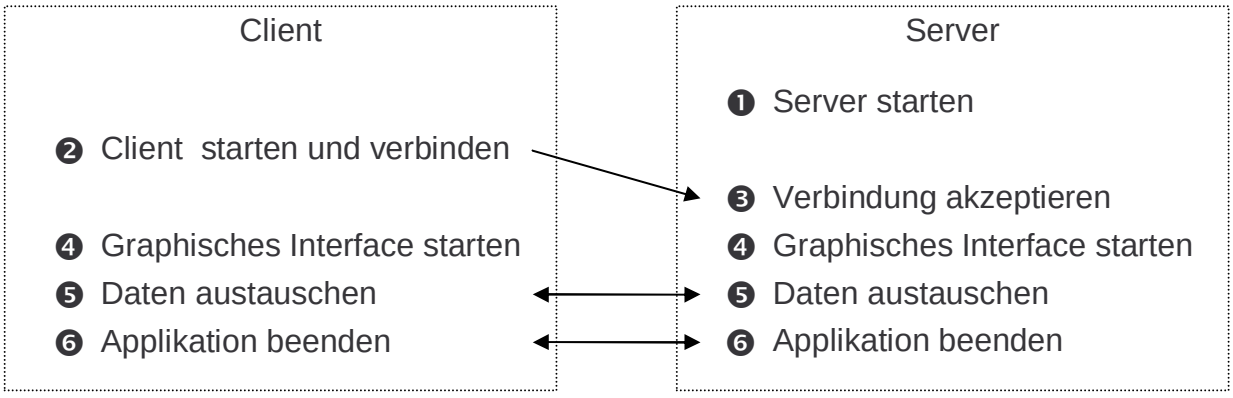

Abbildung 4-3 Testprogramm Kommunikation

5 Resümee

Zunächst wird auf die gewonnen Ergebnisse dieser Diplomarbeit eingegangen. Abschließend wird die Frage beantwortet, wozu die erfolgte Implementierung verwendet werden kann und was weiter geschehen sollte.

5.1 Ergebnisse der Diplomarbeit

Anfangs hat die Diplomarbeit einige Probleme bereitet, vor allem die Abbildung der Funktionalität der IEEE 1394-API auf das Socket-Interface. Durch längere Diskussionen konnte jedoch geklärt werden, ob die Eigenschaften des Sockets oder jene der IEEE 1394-API im Vordergrund stehen sollten. Mit der gewählten Implementierung wurde ein guter Kompromiss zwischen den Eigenschaften des Socket-Interfaces und jenen der IEEE 1394-API gefunden: Der asynchrone IEEE 1394-Datagram-Socket und der isochrone IEEE 1394-Socket bieten die Möglichkeit, auch mit einer IEEE 1394-Applikation asynchron und isochron zu kommunizieren. Bei diesen beiden Sockets wurde jeweils zugunsten der Eigenschaften der IEEE 1394-API entschieden. Beim asynchronen IEEE 1394-Socket wurden die Eigenschaften der Sockets in den Vordergrund gestellt, womit auch die ursprüngliche Funktionalität eines Sockets gewährleistet werden kann.

Durch die klare Trennung zwischen Java und C mittels der vollständigen Implementierung der Funktionalität der IEEE 1394-Sockets in C, kann die erstellte Implementierung sowohl in Java wie auch in C zur Verfügung gestellt werden. Diese Trennung hat sich auch positiv auf die Implementierungsarbeit und das Testen ausgewirkt. Weiters wurde die Implementierung der IEEE 1394-Sockets in mehrere Module unterteilt. Dadurch konnte ein modulares, gut strukturiertes und übersichtliches Design erreicht werden.

Die Implementierung in Java enthält alle drei vorgeschlagenen Arten von IEEE 1394-Sockets. In C wurde jedoch nur die Implementierung des asynchronen

IEEE 1394-Sockets durchgeführt. Dies ist auf die fehlenden Teile des AL1394 zurückzuführen. Die durchgeführte Implementierung (Java und C) umfasst 63 Dateien mit Source-Code, die in Summe ca. 23.300 Zeilen Code beinhalten.

Als Konkurrenz zu dem in dieser Arbeit implementierten Socket-Design könnte `IPover1394` genannt werden [RFC2734]. In Abschnitt 2.3.5 wurden bereits die wesentlichen Unterschiede zu den IEEE 1394-Sockets erläutert und damit auch eine Begründung für das Bestehen dieser Diplomarbeit geliefert.

5.2 Verbesserungsvorschläge

Durch umfangreiche Tests konnte gezeigt werden, dass die vorhandene Implementierung gut funktioniert. Als Ausnahme muss jedoch erwähnt werden, dass das Überlaufen einer named pipe nicht erkannt werden kann, da der AL1394 derzeit kein manuelles Acknowledgment unterstützt. Die Implementierung eines work around wurde als nicht sinnvoll erachtet, da dies bedeuten würde, zusätzlich Datenpakete zu versenden, die eine zusätzliche unnötige Belastung des Bussystems bedeuten würden. Das genannte Problem kann dann auftreten, wenn eine Applikation zu langsam die Daten aus einem asynchronem IEEE 1394-Socket-Objekt entnimmt. Der Fehler manifestiert sich dahingehend, dass die Datenpakete, die somit überlaufen, verloren gehen. Weiters stellen die IEEE 1394-Sockets in C nicht den gesamten Funktionsumfang zur Verfügung. Dabei handelt es sich einerseits um den isochronen IEEE 1394-Socket und den asynchronen IEEE 1394-Datagram-Socket in C und andererseits um die Funktionen `flush`, `seek` und die Socket-Optionen beim asynchronen IEEE 1394-Socket in C. Abschließend sollten noch einige Details verbessert werden: Ist von Seiten des AL1394 auch eine Unterstützung des Events bei Beginn eines Busresets verfügbar, sollte dieses Event beachtet werden, um den Zugriff auf die Umrechnungstabelle während des gesamten Busresets sperren zu können (*siehe Abschnitt 4.2.2*). Schlussendlich sollte auch bei einer Erweiterung des AL1394 bezüglich des manuellen Acknowledgment dieses bei den Datenpaketen berücksichtigt werden.

5.3 Ausblick

Abschließend wird noch darauf eingegangen, wozu die durchgeführte Implementierung verwendet werden kann. Durch die Testprogramme konnte gezeigt werden, dass der asynchrone IEEE 1394-Socket die geforderte Funktionalität zur Verfügung stellt. Er könnte somit für den Datenaustausch zwischen Systemen, die mit einem IEEE 1394-Bussystem verbunden sind, verwendet werden. Dies wiederum würde bedeuten, dass Applikationen zu entwickeln sind, oder bestehende umgeschrieben werden müssten, damit die erfolgte asynchrone IEEE 1394-Socket-Implementierung auch tatsächlich zum Einsatz kommt.

Abschließend sollte noch angemerkt werden, was basierend auf dieser Diplomarbeit weiter durchgeführt werden könnte. Damit die volle Funktionalität der IEEE 1394-Sockets genutzt werden kann (vor allem der isochrone Datenaustausch), sollten auch die beiden nicht implementierten IEEE 1394-Socket-Typen implementiert werden; dabei handelt es sich um den isochronen IEEE 1394-Socket und den asynchronen IEEE 1394-Datagram-Socket. Aber auch am asynchronen IEEE 1394-Socket könnten einige Erweiterungen vorgenommen werden. Einige Punkte wurden bereits im Abschnitt 5.2 behandelt. Es wäre aber auch denkbar, eine Optimierung für die Übertragungsgeschwindigkeit, wie bei der Funktion `flush` in Abschnitt 1.3.4 erwähnt, durchzuführen. Dabei würde es sich um ein Zwischenspeichern der Daten handeln, um die Belastung des IEEE 1394-Bussystems so gering wie möglich zu halten. Dieser Puffer sollte dabei vor allem an die maximale asynchrone Paketgröße angepasst werden. Dabei sollte jedoch bedacht werden, dass diese von der maximalen Übertragungsgeschwindigkeit pro IEEE 1394-Adapter abhängig ist.

Abkürzungen

AL1394	Abstraction Layer 1394
CIP	Common Isochronous Package
DHCP	Dynamic Host Configuration Protocol
DLL	Dynamic Link Library
DNS	Domain Name Server
FIFO	First In First Out
GUID	Globally Unique Identifier
ID	Identifier
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IRM	Isochronous Resource Manager
ISO	International Organization for Standardization
JNI	Java Native Interface
JRI	Java Runtime Interface
JVM	Java Virtual Machine
MSL	Maximum Segment Lifetime
NMI	Native Method Interface
OSI	Open Systems Interconnection
RNI	Raw Native Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VM	Virtual Machine

Literaturverzeichnis

1394St1 IEEE Computer Society: IEEE Standard for a High Performance Serial Bus, 1995

1394St2 IEEE Computer Society: IEEE Standard for a High Performance Serial Bus—Amendment 1, 2000

ALC1 M. Ziehensack, T. Grafendorfer, S. Smejkal, M. Haas: AL1394Core, Institute of Computer Technology, Vienna University of Technology, 2001

ALC2 T. Grafendorfer, C. Tögel, M. Weihs, M. Ziehensack: AL1394Core, Abstraction Layer 1394 Version 1.2, Institute of Computer Technology, Vienna University of Technology, 2002

API1 Unibrain 1394 API Documentation: FireAPI User Mode Interface, 04.04.2000

CNW1 D. Comer: Computernetzwerke und Internets, Seite 349-364, Prentice Hall, 1998

DISS1 N. Stampfl: IEEE1394 as Control Network and High-Speed Backbone for Home and Industrial Automation, Dissertation an der TU Wien (www.ub.tuwien.ac.at), 2000

IEC1 International Electrotechnical Commission: IEC 61883, Consumer audio/video equipment – Digital interface, 1998

JAVA1 Java™ 2 Platform, Standard Edition, v1.2.2 API Specification

JAVA2 P. Heller, S. Roberts: Java 1.2, Developer's Handbook, Seiten 283-356 und 411-521, Sybex Inc., 1999

JNI1 Sun: Java Native Interface - Tutorial, http://java.sun.com/docs/books/tutorial/native1.1/index.html am 6.3.2000

JNI2 JavaSoft: Java Native Interface Specification Release 1.1, http://sunsite.sut.ac.jp/java/docs/jdk1.1/jni.pdf am 6.3.2000, 1997

KOMM1 E. Giese, K. Görgen, E. Hinsch, G. Schulze, K.Truöl: Dienste und Protokolle in Kommunikationssystemen, Die Dienst- und Protokollschnitte der ISO-Architektur, Seite 2-16 und 74-96, Springer-Verlag, 1985

OS1 W. Stallings: Operating Systems, Internals and Design Principles, third edition, Seite 187-285 und 543-584, Prentice-Hall, 1998

RFC1884 R. Hinden, S. Deering: IP Version 6 Addressing Architecture, Network Working Group, ftp://ftp.nic.de/pub/rfc/rfc1883.txt am 15.12.2002, 1995

RFC2734 P. Johansson: IPv4 over IEEE 1394, ftp://ftp.nic.de/pub/rfc/rfc2734.txt am 11.03.2003, 1999

RFC2855 K. Fujisawa: DHCP for IEEE 1394, ftp://ftp.nic.de/pub/rfc/rfc2855.txt am 11.03.2003, 2000

RFC768 J. Postel: User Datagram Protocol, ftp://ftp.nic.de/pub/rfc/rfc768.txt am 11.03.2003, 1981

RFC791 Information Sciences Institute, University of Southern California: Internet Protocol, ftp://ftp.nic.de/pub/rfc/rfc791.txt am 11.03.2003, 1981

RFC793 Information Sciences Institute, University of Southern California: Transmission Control Protocol, ftp://ftp.nic.de/pub/rfc/rfc793.txt am 11.03.2003, 1981

SDL1 F. Belina, D. Hogrefe, A. Sarma: SDL with Applications from Protocol Specification, Prentice Hall Inc. (UK) Ltd, 1991

TIJ1 B. Eckel: Thinking in Java, 2nd edition, rev 3, http://www.codecuts.com/ads-cgi/viewer.pl/codecuts/pdfs/bruceeckel/TIJ2R3.pdf am 6.3.2000, 1999

UML1 B. Oestereich: Die UML-Kurzreferenz für die Praxis, kurz bündig ballastfrei, Oldenbourg Wissenschaftsverlag GmbH, 2001

UNIX1 W. Alex, G. Bernör: UNIX, C und Internet, Moderne Datenverarbeitung in der Wissenschaft und Technik, Seite 162-166, Springer-Verlag, 1994

UNIX2 P. K. Andleigh: Unix Systemarchitektur, Seite 202-294, Carl Hanser Verlag, 1995

UNIX3 Berkeley Software Distribution: UNIX Programmer's Reference Manual, 1986

UNIX4 K. Rosen, R. Rosinski, J. Faber: Unix System V, Grundlagen und Praxis, 4. Auflage, Seite 563-566, te-wi Verlag Gmbh, 1991

UNIX5 M. J. Bach: The design of the Unix operating system, Seite 382-389, Prentice/Hall-International Inc., 1986

WIN1 J. Richter: Microsoft Windows, Programmierung für Experten, 3. Auflage, Seite 925-993, Microsoft Press

WIN2 B. Ezzell: 32Bit-Programmierung für Insider, SAMS, 1996

WIN3 B. Marshall: Win32 System Services, The Heart of Windows NT, Prentice-Hall, 1994

WSNP1 B. Quinn, D. Shute: Windows Sockets Network Programming, Addison-Wesley Publishing Company, 1996

Abbildungsverzeichnis

Tabellenverzeichnis