Diplomarbeit

Zur Erlangung des akademischen Grades
Diplom-Ingenieur

Entwicklung eines Netzwerk-Interface zur Steuerung der Datenkommunikation einer Netzwerkkarte

Angefertigt von
Martin Wodrich
bei
Prof. Dr.-Ing. Axel Hunger

Institut für Medientechnik und Software-Engineering (IMSE)
Abteilung Informatik, Informations- und Medientechnik
Universität Duisburg-Essen, Campus Duisburg

Duisburg, den 13. September 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Derzeitige Netzwerkumgebungen transportieren rein passiv, die ihnen übergebenen Daten. Dies bedeutet insbesondere das keinerlei Anpassung des Datenvolumens an die zur Verfügung stehende Bandbreite erfolgt. Im Gegensatz dazu bieten Application Layer Active Networks [1](ALAN) diese Möglichkeit. Multimedia Dienste, welche ihre Daten für Clients mit verschiedenen Bandbreiten (insbesondere auch an Clients mit relativ geringen Bandbreiten, wie z.B. Multimedia-Mobiltelefone) anbieten, müssen bei den herkömmlichen passiven Netzwerkumgebungen für jede zur Verfügung stehende Bandbreite angepasst werden. Der Ansatz der Active Networks stellt eine dynamische Anpassung der Daten durch das Netzwerk dar.
Bei ALAN werden die Daten auf dem Client an die Netzwerkverbindung angepasst. Hierbei ist es erforderlich die Kontrolle über alle ausgehenden Netzwerkpakete zu erlangen um ein Ressource-Management und damit auch eine effektive Ausnutzung der zur Verfügung stehenden Bandbreite des Systems zu handhaben.

In dieser Arbeit soll der Zugriff auf die Netzwerkschnittstelle derart modifiziert werden, so dass durch ein Management-System Netzwerkpakete modifiziert und koordiniert werden können. Das Betriebssystem Microsoft Windows stellt für den Zugriff auf die Netzwerkschnittstelle bereits verschiedene Schnittstellen bereit.
Dabei stellt jede der Schnittstellen des Betriebssystems eine andere Teilaufgabe innerhalb der Ansteuerung der Netzwerkschnittstelle dar.

Das Kapitel 2 wird nach kurzem Vergleich herkömmlicher Netzwerkumgebungen gegenüber Active Networks auf bestehende Active Network Lösungen eingehen.
Kapitel 3 wird neben der Darstellung des bestehenden Netzwerkstabels von Windows, analysieren welche Modifikationsmöglichkeiten bestehen um Active Network Funktionalität in den Windows-Netzwerkstabel einzubrigen. Insbesondere welche Komponenten hierzu modifiziert werden müssen. Schließlich erfolgt in Kapitel 4 die Vorstellung meines Gesamtkonzeptes einer Application Layer Active Network Systemarchitektur für Windows.
Zuletzt enthält Kapitel 5 eine kurze Zusammenfassung dieser Arbeit.
Dabei wird diese Arbeit primär nur auf die Version Windows 2000 und Windows XP des Windows Betriebssystems eingehen. Eine Betrachtung älterer Versionen wird nur am Rande erfolgen.

1 Application Layer Active Networks: aktive Netzwerke auf Anwendungsebene

2 Active Network Eigenschaften und Überblick

Dieses Kapitel stellt die wesentlichen Unterschiede zwischen einer herkömmlichen passiven Netzwerkumgebung und einem Active Network dar, bevor es auf bestehende Active Network Lösungen eingehen wird.

2.1 Vergleich Active Networks gegenüber herkömmlichen passiven Netzwerkumgebungen

Herkömmliche Netzwerkumgebungen transportieren Daten ohne sie an die zur Verfügung stehende Bandbreite anzupassen. Insbesondere ist es mit einem rein passiven Netzwerk nicht möglich Multimediadaten an Endgeräte mit geringer Bandbreite anzupassen.
Active Networks erlauben ein benutzerspezifische Transcodierung der Daten und damit eine Anpassung an die zur Verfügung stehende Bandbreite. Dies ist insbesondere bei mobilen Endgeräten notwendig und sinnvoll.

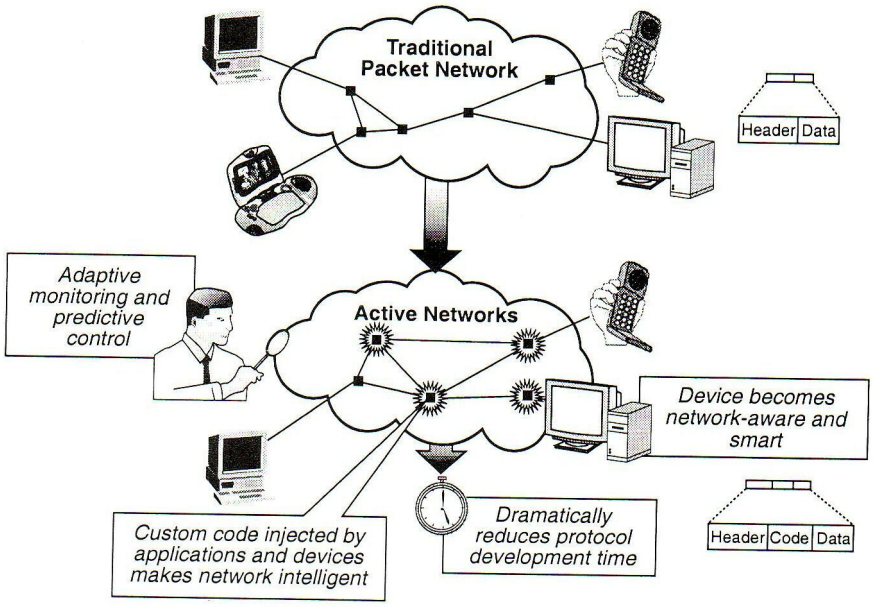

Abbildung 2.1 Überblick Active Networks
(aus [BU01] S.2)

Zusätzlich erlaubt ein Active Network eine reduzierte Entwicklungszeit für neue Netzwerkprotokolle, da die Geräte Code zusammen mit den Daten erhalten und die Daten mit Hilfe dieses Codes verarbeiten können. Außerdem ist ein adaptives Überwachen und vorausschauendes Kontrollieren möglich ([BU01] S.2) Siehe auch Abbildung 2.1.

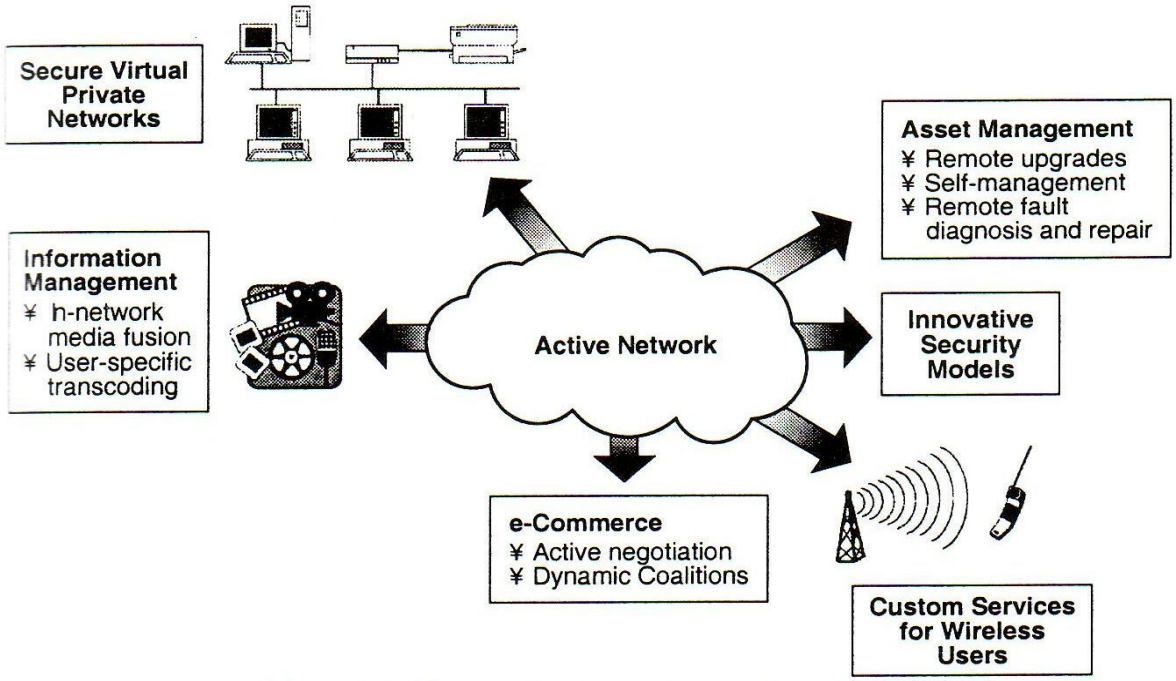

Abbildung 2.2 Möglichkeiten eines Active Network (aus [BU01] S. 21)

Weitere Vorteile eines Active Network

Als weitere Vorteile eines Active Networks gelten die Möglichkeit ein innovatives Sicherheitsmodel zu implementieren, ein virtuelles privates Netzwerk (VPN) abzusichern und die Möglichkeit die Netzwerkkomponenten aus der Ferne zu aktualisieren und auch bei Ausfällen diese aus der Ferne zu diagnostizieren und zu reparieren ([BU01] S.21). Siehe auch Abbildung 2.2.

2.2 Das Active Network Encapsulation Protocol (ANEP)

Das Active Network Encapsulation Protocol (ANEP) ist ein experimentelles Protokoll für die Übertragung von Active Network Informationen über verschiedene Transportmedien. ANEP erlaubt dabei sowohl die Übertragung über bestehende Infrastruktur mittels IPv4 oder IPv6 als auch die Übertragung über den Link-Layer. Der verwendete Mechanismus ist dabei so weit wie möglich erweiterbar ausgelegt. Auch erlaubt dieser Mechanismus die Verwendung mehrerer sogenannter Execution Enviroments unabhängig voneinander. Execution Enviroment nennt man dabei die Ausführungsumgebungen innerhalb des Betriebsystems eines Active Network Netzknotens. Ein solcher Netzknoten ist in der Lage dynamisch Programme nachzuladen und auszuführen. Diese Programme nennt man Active Applications. (siehe [BU01] S.12 bzw. Abbildung 2.3)

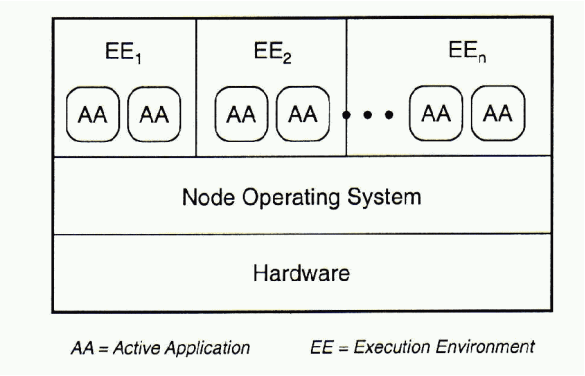

Abbildung 2.3 NodeOS

(aus [BU01] S.12)

Ein Active Network Packetheader erlaubt dabei die schnelle Auswahl einer geeigneten Umgebung zur Auswertung des Active Network Datenpaketes, sowie das Festlegen einer voreingestellten Aktion, wenn die Auswertungsumgebung nicht verfügbar ist.
In [UP04] ist folgender Packetheader beschrieben:

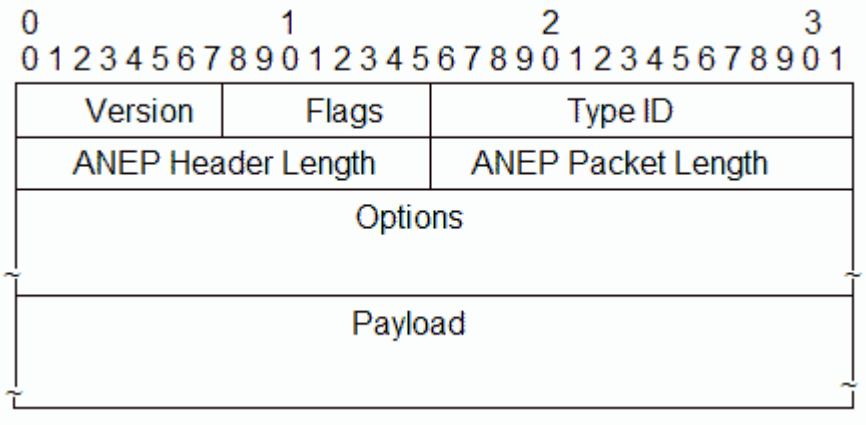

Abbildung 2.4 Aufbau eines ANEP-Pakets
(aus [UP04])

Dabei sind alle Headerfelder die größer als ein Oktet (=1 Byte = 8 Bit) in Big-Endian-Notation[2] zu verstehen. Dies gilt ebenfalls für alle Option-Header.
Die Versionnummer des Headers ist derzeit immer 1. Dieses Feld wird immer dann geändert, wenn sich der Aufbau des ANEP-Headers ändert.
Im Feld Flags wird derzeit nur das höchstwertige Bit benutzt. Dieses signalisiert was zu geschehen hat, wenn ein Netzknoten die TypeID eines ANEP-Packetes nicht kennt. Ist dieses Bit auf 0 gesetzt, so soll dieses Packet über eine Standardroutingregel weitergegeben werden, ansonsten soll es verworfen werden.

2 Big-Endian: höherwertiges Byte zuerst, im Gegensatz zu Little-Endian, welches zuerst das niedrigstwertige Byte überträgt

Das Feld ANEP Header Length spezifiziert die Länge des Header in 32 Bit Worten. Dies bedeutet, wenn keine Optionen verwendet werden, einen Wert von 2. Die TypeID bestimmt die Umgebung in der dieses Active Network Datenpacket ausgewertet werden soll (siehe [UP04]). Eine Liste der aktuell zugewiesenen TypeIds findet sich im Anhang 1.

2.3 Untersuchung bestehender Active Network Systeme

Bestehende Active Network Systeme enthalten intelligente Router, die nicht nur Datenpakete weiterleiten, sondern die enthaltenen Daten anpassen. Dies geschied mit Hilfe verschiedener Ansätze. Dieses Unterkapitel wird auf einige dieser Ansätze kurz eingehen.

2.3.1 Magician

Magician ist ein Active Network System, das in Java geschrieben ist (kompatibel mit JDK 1.1.6 und höher) und ein Execution Enviroment bereitstellt um in Java vorliegende SmartPackets (=Active Network Datenpakete) auszuführen (siehe [MA01]).
Ein Magician SmartPacket liegt dabei in einem genau definierten Format vor (siehe Abbildung 2.5).
Magician benutzt zur Übertragung der Java Klassendefinitionen und Objekten, aus denen die Active Network Datenpaket bei Magician bestehen, über das Netzwerk das ANEP-Format (siehe Kapitel 2.2).
Das Feld „Class Defintion TLV“ (siehe Abbildung 2.5) wird dabei dafür benutzt die Klassendefinitionen zu übertragen, welche zur Rekonstruktuion des als Bytestrom im Feld „Object Defintion TLV“ übertragenen Java Objekt dient.
Jedes SmartPacket enthält mit dem Feld „Type Identifier TLV“ eine eindeutige Nummer der mit Hilfe dieses SmartPackets übertragenen Active Application.
Eine Komponente eines Magician-Netzknotens ist dabei ein sogenannter Routing Manager der ein RIP ähnliches Routingprotokoll welches über das Senden von in SmartPackets enthaltenem Code Einträge in Routingtabellen ändern, löschen oder ergänzt. Dabei unterstützt dieser Routing Manager auch eine Standardroutingregel für alle SmartPackets. Diese sucht den Weg mit den geringsten Kosten. Ein SmartPacket kann diesen Standardmechanismus durch Implementierung eines eigenen Routingmechanismus überschreiben (siehe auch [BU00]). Magician überträgt Code und Daten im selben Active Network Paket (Capsule, siehe [BU01] S.13)

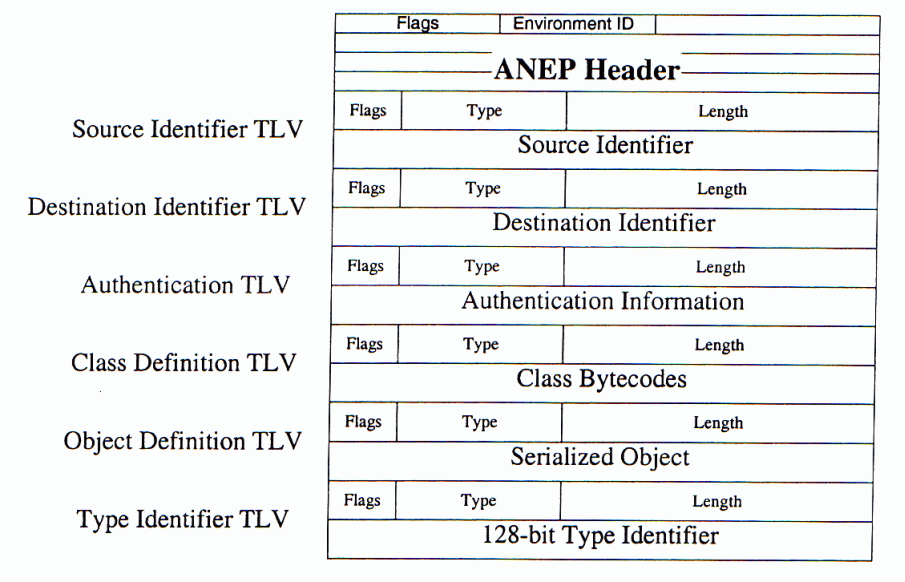

Abbildung 2.5 Magician SmartPacket
(aus [BU00] S.4)

2.3.2 ANTS

ANTS (Active Network Transport System; siehe [AN03]) ist ebenfalls ein Java basiertes Active Network System, welches ebenfalls Code und Daten im selben Paket überträgt.
Dabei wurde ANTS auch für asymetrisches und dynamisches Routing entwicklet.
Asymetrisches Routing bedeutet das der Weg von der Quelle zum Ziel eines Pakets nicht derselbe ist, wie zwischen dem Ziel und der Quelle. Damit ist also Hin- und Rückweg einer Verbindung verschieden (siehe [WE99] S.85).
Bei der Entwicklung von ANTS wird dabei auch berücksichtigt, das Daten eventuel von mehr als einem Empfänger benötigt werden. Dies wird in der TCP/IP-Protokollfamilie (siehe Anhang 2) durch das Verfahren Multicast erledigt (siehe [WE99] S.85).
ANTS kann dabei auch noch erweitert werden. Dies zeigt sich auch daran, das das aktuelle ANTS die Versionnummer 2.0 hat (1.0 gab es auch, hatte aber kaum etwas mit dem heutigen ANTS 2.0 gemeinsam). Mögliche Erweiterungen sind auch ab Seite 101 von [WE99] erläutert.

2.3.3 Switchware

Switchware ist ein hybrides Active Network System (siehe [BU01] S.13). Dies bedeutet das Code und Daten sowohl getrennt als auch gemeinsam übertragen werden können.
Bei der Entwicklung von Switchware wurde auch darauf geachtet, das die Active Applications möglichst leichtgewichtig sind. Dies ist notwendig, da in Active Networks auch der zeitkritischste Teil des Netzwerkes, die Datenübertragung selbst, durch Software geschied. Eben durch die Active Applications.
Switchware implementiert dabei nicht nur normale Active Network Datenpakete sondern auch sogenannte Active Extentions, welche neue Protokolle oder Funktionen implementieren können. Diese Active Extentions können dabei auf Router geladen werden und stellen die neuen Funktionen dann für sehr viele Active Network Datenpackete bereit (siehe [AL98]).

2.3.4 Netscript

Netscript übertragt in seinen Active Network Datenpacketen nur Daten (Discrete, siehe [BU01] S.13 bzw. [SI01]).
Dabei implementiert Netscript eine spezielle Programmiersprache zum Programmieren von Datenflußsteuerungen für Active Networks welche auf den Active Network Routern ausgeführt wird. Netscript-Programme führen dabei einfache Modifikationen an Datenpacketströmen durch. Dazu gehören z.B. Analyse und Klassifikation von Datenströmen, Analyse von Paketheadern, Multiplexen und Demultiplexen von Datenströmen und Routing (siehe [SI01]).
Netscript-Programme können dabei sowohl traditionelle Protokolle wie IP bzw. UDP als auch komplett neue Protokolle implementieren. Der Datenfluß innerhalb eines Netscript-Routers ist dabei durch Boxen mit verschiedener Funktion gegliedert (siehe [SI01] S.540 bzw. Abbildung 2.6).

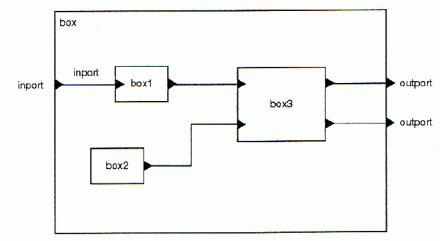

Abbildung 2.6 Netscript Box-Struktur (Beispiel) (aus [SI01] S.540)

2.3.5 Zusammenfassung und Übersicht weiterer Active Network Systeme

Das Active Network System SmartPackets der BBN ist ein Verfahren das wie Magician und ANTS Code und Daten innerhalb derselben Pakete überträgt. Auch CANeS ist ein solches System (siehe Tabelle 2.1).

EE Name	Institution	VM platform	Approach
Magician	University of Kansas	Java	Capsule
ANTS	Massachusetts Inst. of Technology	Java	Capsule
Switchware	University of Pennsylvania	Caml	Hybrid
Netscript	Columbia University	Netscript	Discrete
SmartPackets	BBN	Assembly code	Capsule
CANeS	Georgia Tech.	Unity	Capsule

Tabelle 2.1 Active Network Systeme und ihre Execution Enviroments (aus [BU01] S.13)

Zusamengefasst ist zu sagen, das alle bisherigen Active Network Systeme für Router verschiedene Aspekte und Ansätze für Active Networks ausprobieren. Dabei werden auch verschiedene Ideen bezüglich Verteilung des Codes umgesetzt. Einige dieser Systeme setzen auf Code und Daten in den gleichen Netzwerkpaketen, welches vorallem Vorteile bei relativ kleinen Codestücken hat, welche dann sehr flexibel einsetzbar sind. Andere Systeme setzen auf getrennte Verteilung von Code und Daten. Diese Systeme können sich dabei eine aufwendigere Sicherheitsstruktur leisten, ohne mit Problemen bei der Leistungsfähigkeit des Netzes zu kämpfen. Nachteilig ist bei gleichzeitiger Verteilung von Code und Daten, das aufwendigere Codestücke zuviel Platz verbrauchen würden, bei getrennter Verteilung von Code und Daten leidet dagegen die Flexibilität, da zu jedem Datenpacket immer auch ein dafür geeignetes Stück Active Network Code vorhanden sein muß. Dies kann bei getrennter Verteilung von Code und Daten nicht für jedes Datenpaket unterschiedlich gewählt werden.

Es sind somit Systeme vorteilhaft die beide Ansätze in sich vereinen, in dem sie sowohl Code und Daten getrennt, als auch gemeinsam übertragen können. Diese Systeme nennt man hybride Active Network Systeme.

3 Zugriffsmöglichkeiten auf die Netzwerkkarte unter Microsoft Windows

Dieses Kapitel beschreibt den Aufbau des Netzwerkstabels von Windows, sowie welche Komponenten daraus sich modifizieren lassen um Active Network Funktionalität bereitzustellen. Außerdem beschreibt es den Ablauf eines Netzwerkzugriffes durch eine Netzwerkanwendung mit anschließender Betrachtung der daraus ergebenen Konsequenzen auf tieferen Softwareebenen. Diese unterteilen sich in Usermode[3] und Kernelmode[4] auf die daher auch in den einleitenden Unterkapiteln die den Aufbau beschreiben getrennt eingegangen wird.

3.1 Netzwerk-Bestandteile des Usermode

Der Usermode besteht bezüglich Netzwerkzugriffen aus dem Winsock API, das sich wie folgt zusammensetzt (siehe auch Abbildung 3.1):

16 Bit Winsock-Anwendungen greifen über das Windows Sockets 1.1 API auf die Winsock.dll zu. 32 Bit Winsock-Anwendungen greifen entweder über das Windows Sockets 2.0 API auf die Ws2_32.dll, Mswsock.dll und Wshelp.dll zu, oder über das Windows Sockets 1.1 API auf die Wsock32.dll zu. Die Wsock.dll und Wsock32.dll sind dabei nur zwecks Kompatibilität zu alten Anwendungen vorhanden, da sie letztendlich nur das Windows Sockets 2.0 API in das Windows Sockets 1.1 API wandeln.

Die Ws2_32.dll greift über das Windows Sockets 2.0 Service Provider Interface auf die Layered Service Provider bzw. wenn keine Layered Service Provider vorhanden sind direkt auf die Base Service Provider. Diese unterteilen sich in Helper DLLs und Name Space DLLs. Dies ist allerdings für diese Arbeit weitgehend unwichtig. Ebenso wie sie oben genannten verschiedenen Varianten der Winsock-DLL. Sie werden hier nur der Vollständigkeit halber genannt und um klarzustellen das die in dieser Arbeit vorgestellte Application Layer Active Network Lösung unabhängig von der Art der Anwendung ist, die die Windows Sockets Schnittstellen benutzt. Daher wird im gesamten weiteren Text dieser Arbeit immer nur von der Winsock DLL die Rede sein. Gemeint ist dabei immer die hier genannten Bibliotheken als Ganzes.

3 Usermode: Privilegebene der Anwendungen bei den meisten modernen Betriebssystemen.
4 Kernelmode: Privilegebene des Betriebssystems selbst. Nur in diesem Modus/Privilegebene ist ein direkter Hardwarezugriff machbar.

Wichtig ist allerdings das Layered Service Provider auf andere Layered Service Provider oder den Base Service Providern aufsetzen können und damit erlauben den Zugriff auf die Base Service Provider abzufangen und zu modifizieren.
Mit Hilfe der Msafd.dll greifen die Helper DLLs auf die im Kernelmode vorhandenen Transportprotokolle zu ([JO02] , [MI02] , [ND04]).

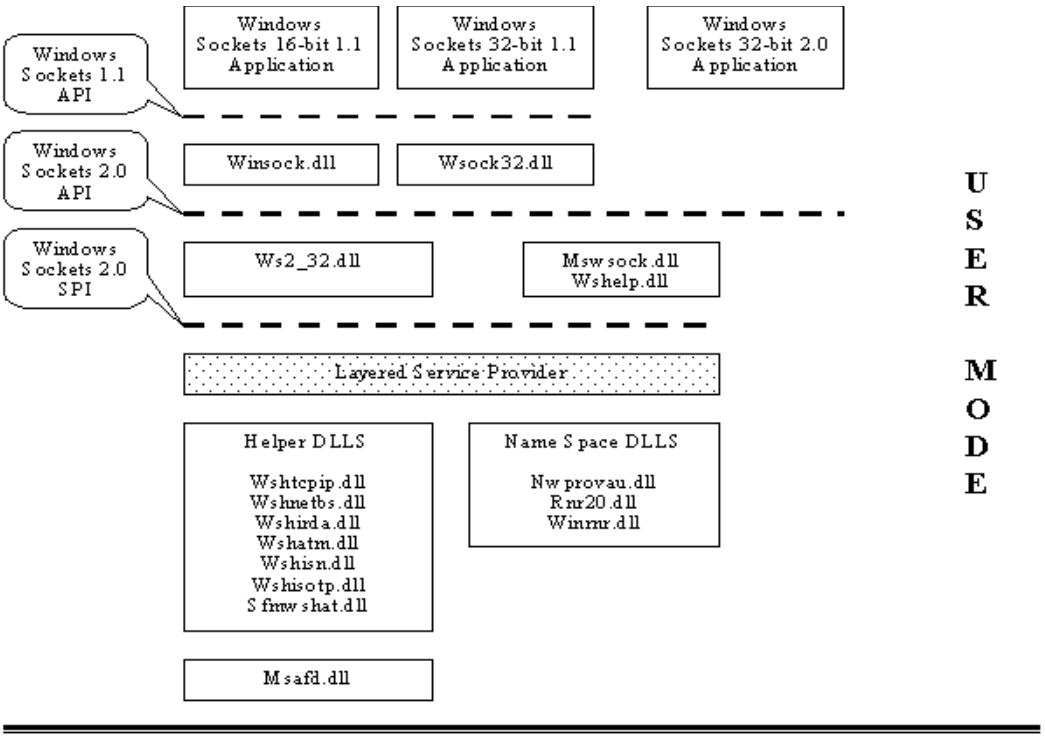

Abbildung 3.1 Windows Netzwerkstack (Usermode)
(aus: [ND04])

3.2 Netzwerk-Bestandteile des Kernel-Mode

Der Kernelmode besteht bezüglich Netzwerkzugriffen aus dem NDIS Interface (siehe Abbildung 3.2):
Ein NDIS-Miniport-Treiber steuert direkt die Hardware der Netzwerkkarte an. NDIS Intermediate Treiber können sich zwischen Miniport Treiber und Protokoll Treibern befinden. Sie lassen sich für verschiedene Zwecke einsetzen.
Diese sind Datenfilterung aus Sicherheitsgründen oder anderen Zwecken, Load Bancing, Erhöhung oder Realisierung von Ausfallsicherheit, Überwachung und Sammlung von Daten zur Erstellung von Statistiken über die Netzwerkzugriffe, sowie Anpassungen eines veralteten Protokoll/Transporttreibers an einen aktuellen Miniporttreiber. Ein NDIS Protokoll-Treiber stellt die oberste Treiberebene im Kernelmode dar ([JO02] , [MI02]).

Weitere Arten von Treibern des Kernelmode sind Firewall-Hook-Treiber (oft als NDIS Intermediate Treiber realisiert([MI02])) sowie Filter-Hook-Driver von denen nur genau einer installiert werden kann und nur die Entscheidung darüber realisiert werden kann, ob ein IP-Paket weitergeleitet,weggeworfen, oder durchgelassen wird ohne Möglichkeit einer Manipulation.

Tiefere Einblicke in den Kernelmode wie sie teilweise in Abbildung 3.2 gezeigt werden, sind für diese Arbeit nicht relevant und werden daher nicht weiter erläutert.

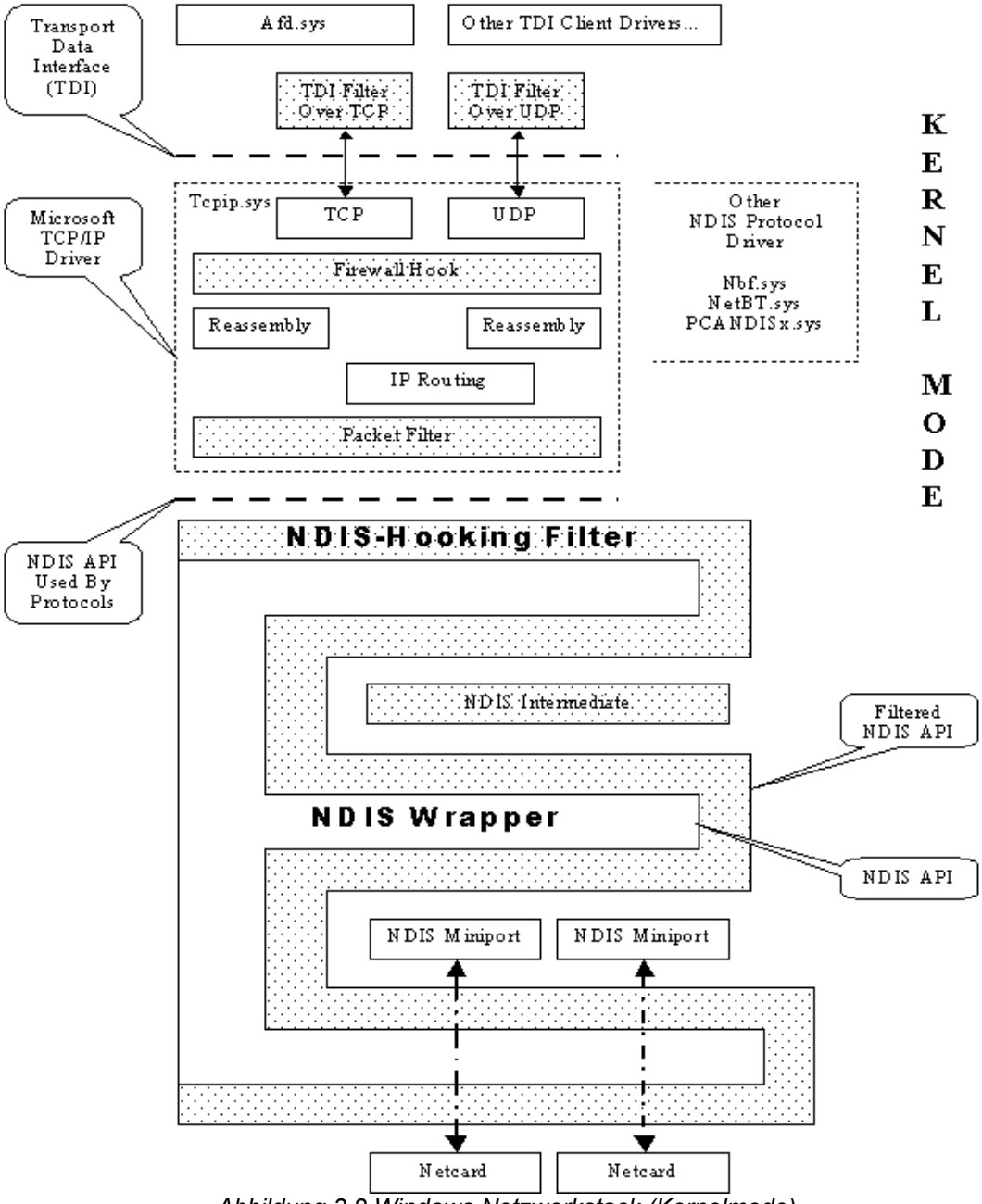

Abbildung 3.2 Windows Netzwerkstack (Kernelmode) (aus [ND04])

3.3 Eignung für Active Network Zwecke

Wie zum Teil bei der Vorstellung der verschiedenen Treiberarten in Kapitel 3.1 und 3.2 anklang, eignen sich verschiedene Treiber unterschiedlich gut für die Beeinflussung der zu sendenden Daten nach Active Network Gesichtspunkten. Daher folgt jetzt eine systematische Bewertung der entsprechenden Bestandteile.

Winsock-DLL

(siehe Kapitel 3.1)

Als fester Bestandteil des Windows-Netzwerkstabels lässt sich die Winsock-DLL nicht dazu verwenden, Active Network Datenbeeinflußung zu realisieren.

Layered Service Provider

(siehe Kapitel 3.1)

Diese Treiberart eignet sich hervorragend dazu Zugriffe auf den Base Provider abzufangen oder zu beeinflussen. Und damit ist dies auch eine der möglichen Teile des Netzwerkstabels von Windows um Active Network Datenbeeinflußung zu realisieren. Insbesondere da noch keinerlei Protokollheader an die Daten angefügt wurden, die nach Modifikation der Daten neu erzeugt wedren müssten.

Base Provider

(siehe Kapitel 3.1)

Diese Komponente ist wiederum fester Bestandteil des Netzwerkstabels von Windows. Er ist allerdings abhängig vom darunterliegenden Transportprotokolltreibers im Kernelmodus. Benutzt man einen anderen Transportprotokolltreiber so kann man auch einen neuen Base Provider benutzen.

Transportprotokolle

(siehe Kapitel 3.2)

Diese sind im allgemeinen zu komplex um sie extra für Active Network Belange neu zu schreiben. Zumal ein Eingriff an dieser Stelle auch bedeutet das man die Kontrolle über den gesamten Datenverkehr verliert, da Verkehr der über ein anderes bzw. über das herkömmliche Transportprotokoll abgewickelt wird, nicht durch das neu geschriebene Protokoll erfasst werden kann.

Intermediate Treiber

(siehe Kapitel 3.2)

Da sich diese Treiber genau zwischen Transportprotokolltreiber und NDIS Miniporttreiber befinden, eignen sie sich sehr gut dazu fertige Pakete zu filtern (durchlassen, weiterleiten oder wegwerfen) bevor sie mit Hilfe des Miniporttreibers auf die Reise gehen bzw. nachdem sie mit Hilfe des Miniporttreibers empfangen wurden. Eine Veränderung der enthaltenen Daten ist allerdings nicht mehr so einfach möglich, da bereits fast alle Protokollheader (Ausnahme ist z.B. der Ethernetheader wenn Ethernet als Hardwareebene verwendet wird. Bei Verwendung von anderen Hardwareprotokollen gilt Entsprechendes) angefügt sind und daher nach einer Modifikatiuon der Daten agepasst werden müssten. Außerdem erlaubt ein solcher Treiber es den Funktionsumfang eines Miniporttreibers zu erweitern.

NDIS Miniporttreiber

(siehe Kapitel 3.2)

Mit Hilfe dieses Treibers werden die fertigen Datenpakete über eine Netzwerkkarte gesendet bzw. empfangen. Da dieser Treiber direkt für die Ansteuerung der Hardware verantwortlich ist, ist er ebenfalls abhängig von der verwendeten Netzwerkkarte. Er ist daher nicht geeignet Active Network Datenbeeinflußung zu realisieren. NDIS Miniporttreiber werden in der Regel nur von Hardwareherstellern (also dem Hersteller der Netzwerkkarte geschrieben um die Netzwerkkarte auch tatsächlich ansteuern zu können).

Firewall-Hook Drivers

(siehe Kapitel 3.2)

Wie schon in Kapitel 3.2 geschildert handelt es sich hierbei oft nicht um eine eigne Treiberart, sondern nur um eine spezielle Verwendung des Intermediate Treibers.

Filter-Hook Driver

(siehe Kapitel 3.2)

Auch diese Treiberart unterstützt keine Datenbeeinflußung. Sondern erlaubt es nur Netzwerkpakete abzulehnen, durchzulassen oder weiterzuleiten. Und ist damit als Active Network Treiber ungeeignet.

Damit ist nur ein Layered Service Provider oder ein Intermediate Treiber überhaupt geeignet Active Network Funktionalität zu realisieren. Da es ein Layered Service Provider einfacher erlaubt diese Möglichkeiten zu realisieren, wird auch genau dieser dazu verwendet die Active Network Funktionalität in Kapitel 4 zu implementieren. Insbesondere da bei Verwendung eines Layered Service Provider nach Modifikation der Daten keine Header zusätzlich angepasst werden müssen.

Die nachfolgende Tabelle (Tabelle 3.1) fasst die ersten 3 Unterkapitel von Kapitel 3 zusammen um diese Entscheidung noch zu verdeutlichen:

Usermode	Winsock-DLL	- Bereitstellung von Sockets für Anwendungen	Ungeeignet, da fester Bestandteil des Windows-Netzstapels
	Layered Provider	- Beeinflußung und Abfangen von Anfragen an den Base Provider	Geeignet
	Base Provider	- Weiterleitung von Anfragen an im Kernelmode befindliche Transportprotokole	Ungeeignet, da fester Bestandteil des Windows-Netzstapels
Kernelmode	Transportprotokole	- Transportprotokole (stellen Netzwerkpakete zusammen)	Ungeeignet, da fester Bestandteil des Windows-Netzstapels
	Intermediate Treiber	- z.B. Filterung von fertig aufgebauten Netzwerkpaketen	Bedingt geeignet, da Netzwerkpakete ausgepackt bzw. eingepackt werden müssten
	Miniport Treiber	- Ansteuerung der Netzwerkkarte	Ungeeignet, da genaue Funktion abhängig von der Hardware der Netzwerkkarte
	Firewall-Hook	Paketfilterung	-/-
	Filter-Hook	Paketfilterung	Ungeeignet, da nur begrenzte Funktionalität möglich

Tabelle 3.1 Zusammenfassung Netzwerkbestandteile

3.4 Ablauf eines Netzwerkzugriffes

Eine Netzwerkanwendung muß mehrere wichtige Software-Schritte durchführen um tatsächlich auf das Netzwerk zuzugreifen. Abbildung 3.3 zeigt alle wesendlichen Schritte zusammenfassend auf die ich in den nachfolgenden Unterkapiteln näher eingehen werde.

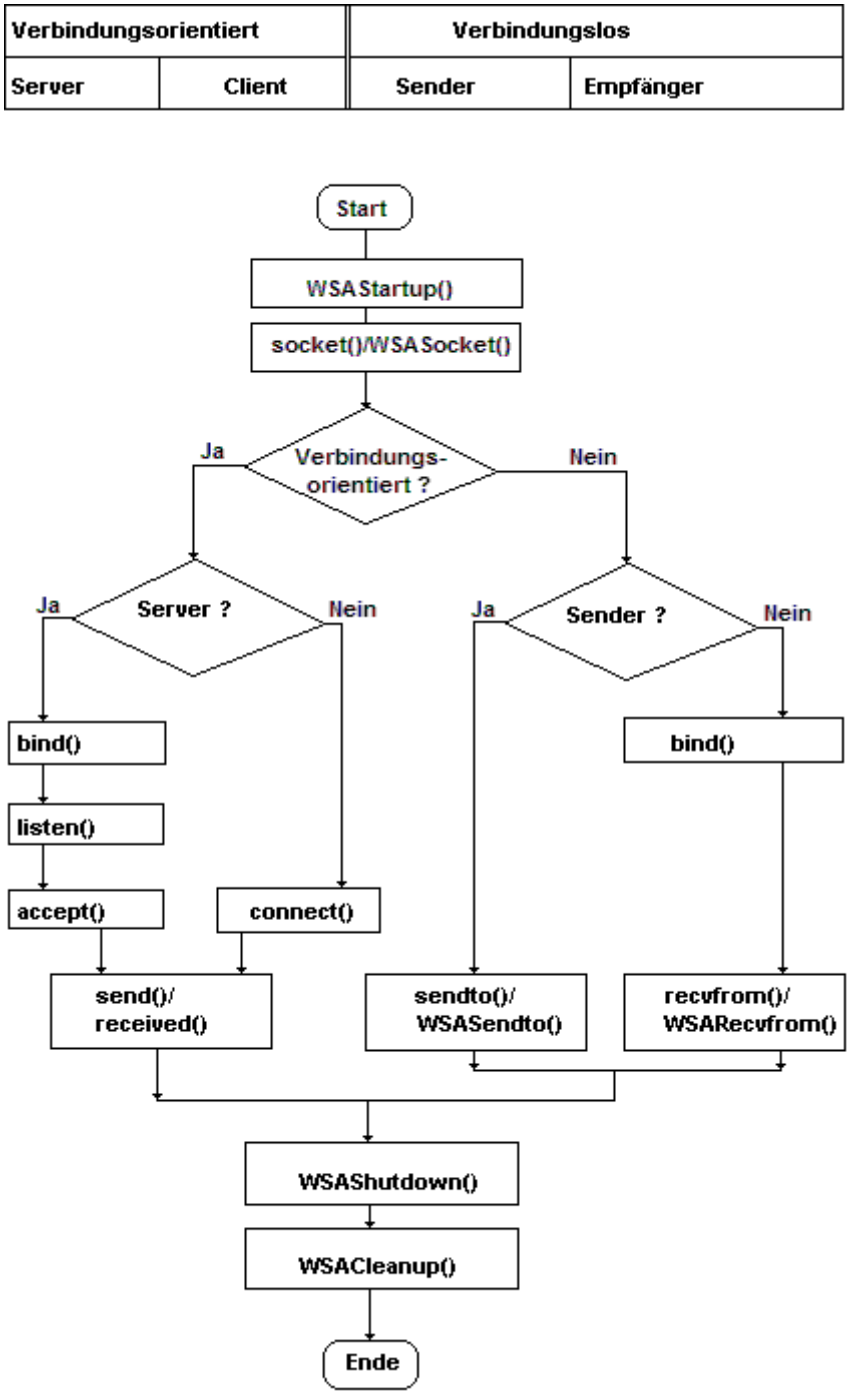

Abbildung 3.3 Ablauf eines Netzwerkzugriffes

3.4.1 Netzwerkzugriff vorbereiten

Um einen Netzwerkzugriff durchzuführen muß eine Anwendung zuerst die Funktions-Bibliothek für Netzwerkzugriffe öffnen und initialieren. (Diese Bibliothek ist die Winsock DLL und sie wird mit dem Aufruf der Funktion WSAStartup(), welche in der Winsock DLL enthalten ist für die Benutzung vorbereitet).
Anschließend muß für jede aufzubauende Verbindung ein Handle (Socket) angelegt werden. (Dies geschied über eine der Funktionen socket() oder WSASocket() der Winsock DLL).

Die weiteren Schritte hängen davon ab, ob ein verbindungsorientierter oder verbindungsloser Netzwerkzugriff erfolgen soll. Bei einem verbindungsorientiertem Netzwerkzugriff unterscheidet man zwischen Client und Server (Ein Client verbindet sich zu einem Server, welcher ständig auf eingehende Anfragen wartet), während bei verbindungslosen Netzwerkzugriffen zwischen Sender und Empfänger unterschieden werden muß. Jeder dieser Zugriffsarten bedingt unterschiedliche Software-Schritte.

3.4.2 Netzwerkzugriff eines Server (verbindungsorientiert)

Die für einen Server notwendigen Schritte sind:
Eine Serveranwendung muß für ihren Netzwerkzugriff den angelegten Handle (Socket) an eine Netzwerkadresse binden. (Dies geschied mit der Funktion bind() aus der Winsock DLL und bei TCP/IP (beide Versionen) ist damit die sogenannte IP-Adresse gemeint).
Anschließend den an eine Adresse gebundenen Handle in den Horchstatus bringen.
(Dies geschied mit der Funktion listen()).
Dann kann die Serveranwendung auf eintreffende Verbindungswünsche warten und diese dann annehmen. (Dies geschied mit der Funktion accept() und jeder Aufruf nimmt eine Verbindung an.
Während eine Verbindung besteht, können auch weitere Verbindungen angenommen werden und diese nachdem sie angenommen wurden auch zur Datenübertragung genutzt werden. Dabei wird für jede angenommene Verbindung ein eignes Handle (Socket) automatisch angelegt.
Aufgebaute Verbindungen können zum Daten senden und empfangen benutzt werden.
(Daten werden mit send() gesendet und mit received() empfangen).

Wird eine Verbindung nicht mehr zur Datenübertragung benötigt, so wird sie beendet (Hierfür gibt es mehr als eine Möglichkeit. Entweder senddisconnect() welches ein letztes Mal Daten sendet und anschließend die Verbindung beendet oder WSACloseSocket() welches des Handle direkt schließt.).
Soll die Serveranwendung beendet werden, oder vorübergehend keine Anfragen beantworten so ist der an die Netzwerkadresse gebundenen Handle freizugeben.
(Die Funktion WSAShutdown() der Winsock DLL wird hierzu verwendet).
Wird die Funktion der Winsock DLL endgültig nicht mehr benötigt so muß sie nach der Benutzung aufgeräumt werden. (Die Funktion WSACleanup() führt dies durch.)

3.4.3 Netzwerkzugriff eines Client (verbindungsorientierent)

Die für einen Client notwendigen Schritte sind dagegen nicht ganz so umfangreich.
Ein Client muß für seinen Netzwerkzugriff mit Hilfe des angelegten Handle eine Verbindung zu einem Server aufbauen. (Die Funktion connect() der Winsock DLL erledigt dies). Danach kann er Daten senden und empfangen (Daten werden mit send() gesendet und mit received() empfangen). Das Beenden der Verbindung und Aufräumen der Winsock DLL geschied auf die gleiche Weise wie bereits beim Netzwerkzugriff eines Servers beschrieben.

3.4.4 Netzwerkzugriff eines Sender (verbindungslos)

Der Ablauf des verbindungslosen Sendes ist sehr einfach, da nach Anlegen des Handle (Socket) Daten mit sendto() direkt losgesendet werden können. Nach Benutzung muß das Handle dann nur noch freigeben werden. (Die Funktion WSAShutdown() der Winsock DLL wird dazu auch bei verbindungslosen Netzwerkzugriffen benutzt).
Auch nach verbindungslosem Netzwerkzugriff muß die Winsock DLL nach Benutzung aufgeräumt werden. (Die Funktion WSACleanup() führt dies durch.)

3.4.5 Netzwerkzugriff eines Empfänger (verbindungslos)

Um verbindungslose Netzwerkzugriffe anzunehmen sind auch nur wenige Schritte notwendig. Die Netzwerkanwendung die dies durchführen will muß dazu einfach nur den angelegten Handle (Socket) an eine Netzwerkadresse binden. (Wie beim verbindungsorientierenten Server ist hierzu die Funktion bind() aufzurufen). Danach kann sie schon Daten mit recvfrom() empfangen.

Soll diese Netzwerkanwendung beendet werden, so ist der Handle nach Benutzung freizugeben. (Die Funktion WSAShutdown() der Winsock DLL wird dazu auch bei verbindungslosen Netzwerkzugriffen benutzt).

Und zuletzt ist wie immer die Winsock DLL nach der Benutzung aufzuräumen. (Die Funktion WSACleanup() führt dies durch.)

4 Application Layer Active Network Systemarchitektur für Windows

Dieses Kapitel beschreibt, erläutert und dokumentiert die von mir realisierte Application Layer Active Network Systemarchitektur für Windows wie sie Abbildung 4.1 zeigt.

4.1 Anforderungen an die Active Network Systemarchitektur

Die neuen Komponenten (fettgedruckte Beschriftung in Abbildung 4.1) des Netzwerkstabels müssen in der Lage sein, abhängig von der zur Verfügung stehenden Bandbreite, die zu sendenen Daten so anzupassen, das ein mit geringer Bandbreite angebundenener Empfänger die Multimediadaten weiterhin in Echtzeit erhalten kann.
Dabei ergeben sich folgene Teilprobleme:

Abgriff der Daten innerhalb des Netzwerkstabels von Windows
Um die durch den Netzwerkstabel durchfließenden Daten abzugreifen, bevor sie an die Netzwerkkarte gelangen, bietet sich wie bereits in Kapitel 3 geschildert, die Treiberart Layered Service Provider an. Diese erlaubt allerdings nur eine genau festgelegte Aufrufschnittstelle (Windows Sockets 2.0 SPI) welche nicht erweiterbar ist.

Transportprotokollabhängigkeit der Datenstrukturen innerhalb des Netzwerkstabels
Einige Datenstrukturen der Winsock 2.0 API/SPI sind abhängig vom verwendeten Transportprotokoll. Diese Strukturen enthalten Informationen darüber welches Protokoll auf Anwendungsebene verwendet wird.

Anwendungsabhängige Daten
Jede Anwendung kann unterschiedliche Daten über unterschiedliche Anwendungsprotokolle senden. Dies muß bei der Modifikation der Daten berücksichtigt werden können.

Datenvielfalt einer Anwendung
Eine einzige Anwendung kann mit Hilfe eines einzigen Anwendungsprotokolls viele verschiedene Daten senden. Diese Systemarchitektur muß also in der Lage sein, auf sich auf die tatsächlich gesendeten Daten anzupassen.

Erweiterbarkeit

Neuere Protokolle und Datentypen müssen einfach nachträglich implementierbar sein.

4.2 Überblick über die verschiedenen Komponenten

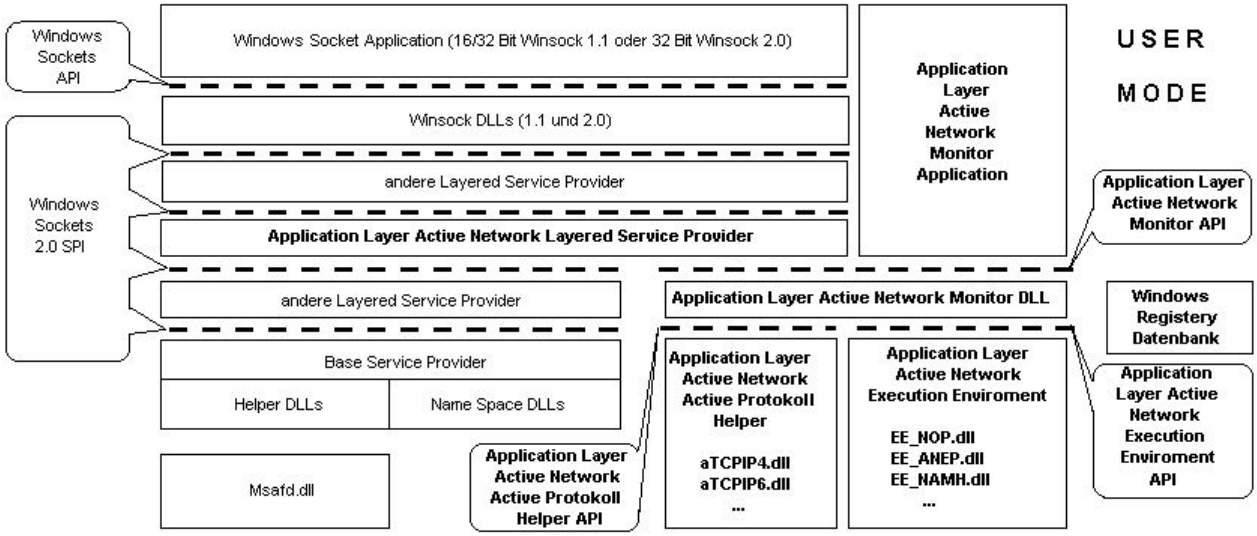

Abbildung 4.1 Application Layer Active Network Systemarchitektur für Windows (Überblick)

Application (Anwendung)

Sie braucht für den Zugriff auf die neue Systemarchitektur nicht extra angepasst werden, da die Schnittstelle zum System weiterhin das Windows Sockets API darstellt.
Allerdings ist die Verwendung eines gut für die Modifikation geeigneten Datenformats vorteilhaft. So stellen verschlüsselte Daten ein Problem dar, diese nicht analysierbar sind.

Winsock DLL

Die Winsock DLLs bleiben genauso erhalten, wie sie in aktuellen Windows Versionen enthalten sind.

Layered Service Provider (kurz auch: LSP)

LSPs können weiterhin in beliebiger Reihenfolge installiert werden, wie dies in einer normalen Windows Installation der Fall ist. Einer dieser LSPs ist dann der
Application Layer Active Network Layered Service Provider.

Application Layer Active Network Layered Service Provider

Dieser LSP stellt die Verbindung zwischen dem normalen Netzwerkstabel von Windows und den neu programmierten Komponenten der Active Network Systemarchitektur für Windows dar. Als Dateiname habe ich *ALAN_LSP.dll* ausgewählt.

Wie alle anderen LSPs bietet er das Windows Sockets Service Provider Interface 2.0 an die drüber liegenden Softwareebene (Winsock DLL oder anderer LSP) an und erwartet dasselbe Interface von der darunter liegenden Softwareebene.

Als Bindeglied zur den anderen Komponenten der Active Network Systemarchitektur greift er über das Application Layer Active Network Monitor API auf die Application Layer Active Network Monitor DLL zu um die durchlaufenden Daten durch die anderen Teile der Systemarchitektur durchzuleiten.

Application Layer Active Network Monitor DLL

Dies stellt die zentrale Komponente der hier vorgestellten neuen Systemarchitektur dar. Er hat den Dateinamen ALAN_MON.dll . Alle Daten kommen erstmal hier an und werden von hier aus durch weitere Teile der Systemarchitektur geleitet. Auch die Entscheidungen darüber welche Teile dies sind werden hier getroffen. Als dauerhafte Ablage für Konfigurationseinstellungen dient dabei die Windows Registery. Um protokollabhängige Entscheidungen treffen zu können greift die Application Layer Active Network Monitor DLL auf die Application Layer Active Network Active Protokoll Helper zu. Die tatsächlichen Datenmodifikationen werden durch die Execution Enviroments durchgeführt.

Application Layer Active Network Active Protokoll Helper

Diese DLLs helfen der Application Layer Active Network Monitor DLL dabei das korrekte Execution Enviroment auszuwählen. Dies geschied in der Form, das aus der von der Anwendung übermittelten Socketadressstruktur der Teil ausgelesen wird der auf das jeweilige Protokoll der Anwendungsebene hindeutet. Bei TCP/IP ist dies sowohl bei Version 4 (Dateiname des Helpers aTCPIP4.dll), als auch bei Version 6 (aTCPIP6.dll) die sogenannte Portnummer (z.B. HTTP verwendet normalerweise Port 80).

Die Socketadressstruktur (siehe auch Abbildung 4.2 und 4.3) ist jeweils abhängig davon welches Protokoll verwendet wird und daher ist für jedes Protokoll ein eigner Helper vorgesehen.

```
struct sockaddr_in
{
    short          sin_family;
    u_short        sin_port;
    struct in_addr  sin_addr;
    char           sin_zero[8];
};
```

Abbildung 4.2
Socketadressstruktur TCP/IPv4

```
struct sockaddr_in6 {
        short          sin6_family;
        u_short        sin6_port;
        u_long         sin6_flowinfo;
        struct in6_addr sin6_addr;
        u_long         sin6_scope_id;
};
```

Abbildung 4.3
Socketadressstruktur TCP/IPv6

Application Layer Active Network Execution Enviroment

Die Execution Enviroments führen die tatsächliche Modifikation der Daten durch. Sie können dabei mit Hilfe der Monitor DLL auch weitere Execution Enviroments aufrufen.
Dies ermöglicht einen sehr modularen Aufbau, der insbesondere bei komplexeren erweiterbaren Protokollen vorteilhaft ist. Das Execution Enviroment für ANEP (EE_ANEP.dll) demonstriert diesen Aufbau. ANEP (Active Network Encaplulation Protokoll) ist ein Standardformat für Active Network Datenpakete. Wie das Paket zu interpretieren ist erst durch die TYPE-Id feststellbar. Daher liest EE_ANEP.dll diese TYPE-Id aus und ruft mit Hilfe der Monitor DLL das für diese TYPE-Id zuständige Execution Enviroment auf. Insgesamt ist allerdings zu sagen, die genaue Funktion eines Execution Enviroments insgesamt beliebig ist. Fest ist nur die Parameter des Aufrufs und die Rückgabewerte an die Monitor DLL.
Möglich ist alles zwischen überhaupt keiner Funktion (EE_NOP.dll) und einer kompletten virtuellen Maschine (z.B. EE_NAMH.dll, die eine Nulladressmaschine in Havardarchitektur realisiert).

Base Service Provider

Die Base Service Provider bleiben so erhalten wie sie bei einem aktuellen Windows ausgeliefert werden, da jegliche Funktion durch die bereits genannten Komponenten realisiert sind.

4.3 Dateinamenskonventionen

Alle neuen Komponenten der hier vorgestellten Systemarchitektur sind mit eindeutigen Dateinamen versehen:

Der Application Layer Active Network Layered Service Provider heißt *ALAN_LSP.dll*.

Die Application Layer Active Network Monitor DLL heißt *ALAN_MON.dll*.

Die Application Layer Active Network Active Protokoll Helper haben Dateinamen die mit einem a beginnen und danach den Protokollnamen enthalten (*aTCPIP4.dll* (für TCP/IPv4) und *aTCPIP6.dll* (für TCP/IPv6)).

Application Layer Active Network Execution Enviroments haben Dateinamen die aus dem Kürzel EE (für Execution Enviroment) einem Unterstrich „_“ und einem Kürzel für das Protokoll oder die Funktion des Execution Enviroment haben. So heißt das funktionslose Dummy-Execution Enviroment EE_NOP.dll (NOP für No Operation vgl. mit dem Prozessorbefehl gleichen Namens der auch nichts weiter macht). Das Execution Enviroment für ANEP (Active Network Encaplulation Protokoll) heißt EE_ANEP.dll . Die virtuelle Nulladressmaschine mit Havardarchitektur dementsprechend EE_NAMH.dll.

Weitere Dateinamen werden in dieser Arbeit nicht verwendet da sie keine mit dieser Arbeit fertiggestellten Module betreffen, da aber diese Systemarchitektur modular aufgebaut ist, um leicht erweiterbar zu sein, sind weitere Dateinamen möglich. Insbesondere sind weitere Protokoll Helper und Execution Enviroment möglich. Daher kann es keine endgültige Liste aller Dateinamen geben, wohl aber eine Liste aller verwendeten Dateinamen. Dies ist Tabelle 4.1.

Layered Service Provider	
ALAN_LSP.dll	Layered Service Provider für Application Layer Active Networks
INSTLSP.exe	Installationsprogramm für ALAN_LSP.dll

Monitor	
ALAN_MON.dll	Application Layer Active Network Monitor DLL
MONTOR.exe	Kleines Testprogramm das die Datenzähler ausliest
INSTALL.exe	Installationsprogramm das die Registeryeinträge die ALAN_MON.dll für den Aufruf externer Module benötigt

Active Protokol Helper	
aTCPIP4.dll	Application Layer Active Network Active Protokoll Helper für TCP/IPv4
ATCPIP6.dll	Application Layer Active Network Active Protokoll Helper für TCP/IPv6

Execution Enviroments	
EE_NOP.dll	Execution Enviroment ohne Funktion außer das alle durchlaufenden Daten als modifiziert gelten
EE_ANEP.dll	Execution Enviroment für das Active Network Encaplulation Protokoll
EE_NAMH.dll	Virtuelle Nulladressmaschine

Tabelle 4.1 Dateinamen

4.4 Zusammenarbeit aller Komponenten bei Netzwerkzugriffen

Alle diese Komponenten arbeiten gemeinsam um die gewünschte Funktionalität bereitzustellen. Wie sich diese Zusammenarbeit darstellt findet sich in diesem Unterkapitel.

4.4.1 Initialieren des Netzwerkstabels

Benötigt eine Anwendung Netzwerkzugriff so öffnet sie die Winsock DLL und initialiert diese mit Hilfe der Funktion WSAStartup derselben DLL. Die Winsock DLL ladet automatisch die jeweils benötigten Service Provider und initialiert diese mit dem Aufruf ihrer Funktion WSPStartup(). Dies geschied auch beim Application Layer Active Network Layered Service Provider. Da die Winsock DLL diesen Aufruf durchaus mehrfach durchführen kann (für jede Anwendung einmal) muß der ALAN_LSP auch erkennen, wie oft bereits seine Funktion WSPStartup() aufgerufen wurde (siehe Abbildung 4.4).

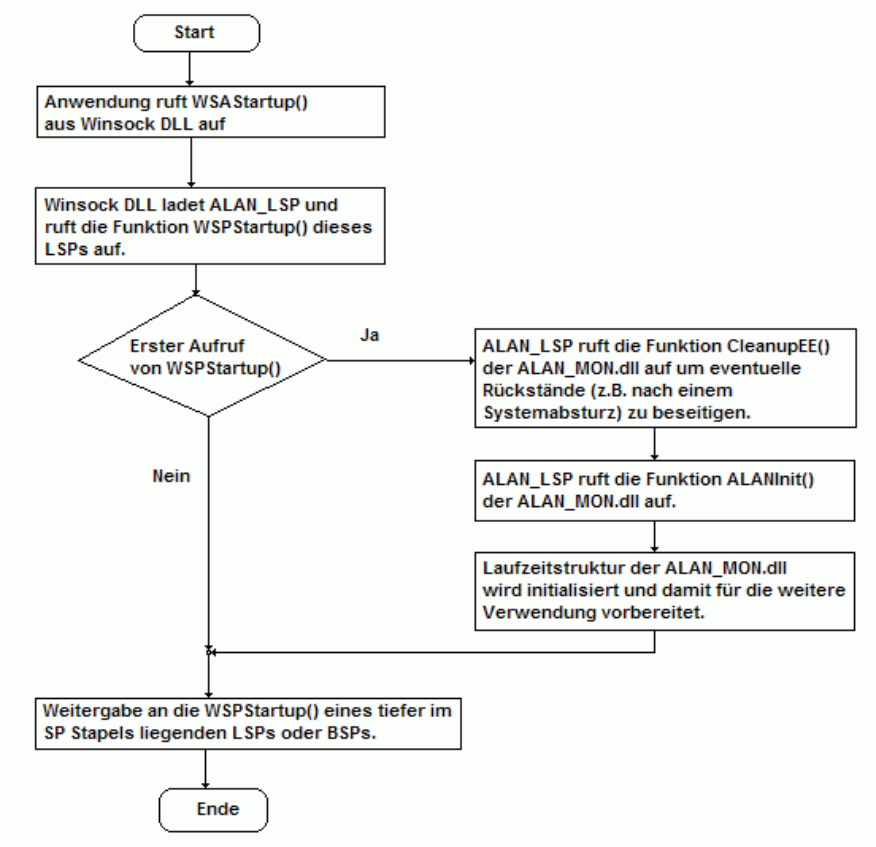

Abbildung 4.4 Initialisieren des Netzwerkstabels

Nur beim ersten Start darf er die Aufräumfunktion CleanupEE() aus der Monitor DLL aufrufen, da er sonst bestehende Netzwerk-Verbindungen stören würde.

Ebenfalls nur beim ersten Aufruf darf die Funktion ALANInit() der Monitor DLL aufgerufen

werden, da diese ebenfalls ansonsten bestehende Netzwerk-Verbindungen stören würde. Sie initialiert die Laufzeitstruktur der Monitor DLL (siehe Abbildung 4.5) in der die Verbindungsinformationen aller nach Active Network Gesichtspunkten beeinflußten Verbindungen gespeichert werden. Initialieren heißt hier das der Speicher für die Struktur geleert wird, um sicherzustellen das kein Datenmüll, von anderen Anwendungen, sich in diesem Speicher befindet, der zu Fehlinterpretationen führen würde.

```
// Laufzeitstruktur definieren
typedef struct _active_network_handler
{
	unsigned long bytes_in_send;	// Statistik
	unsigned long bytes_out_send;
	unsigned long bytes_in_rec;
	unsigned long bytes_out_rec;
	unsigned long sendrate;		// Letzte gemessene Senderate
	unsigned long lastclock;	// Zeit der letzten Daten von der Anwendung
	unsigned long lastsize;		// Größe der letzten Daten von der Anwendung
	char ee_name[13];			// Name des Execution Enviroments
	char start;				// Status 0=noch keine gültigen Informationen gespeichert
					//	1=ee_name enthält Protokolnummer+Sockettype
					//	2=ee_name enthät Name des zuständigen EE
	unsigned short id;		// Verbindungs-ID aka Socket
} active_network_handler;

// Laufzeitstruktur anlegen
active_network_handler active_network_handlers[65535];
```

Abbildung 4.5 Laufzeitstruktur der Monitor DLL

In dieser Laufzeitstruktur werden neben der Nummer des Sockets, Statusinformationen, Statistikinformationen vorallem auch der Name des Execution Enviroments gespeichert. Außerdem Informationen über tatsächlich aufgetrettene Datenraten beim Senden, sowie Zeitpunkt und Größe des letzten Datenbuffers.

Aus Sicherheitsgründen kann diese Struktur nur über eine extra zu diesem Zweck geschaffene Funktion durch eine Monitorapplication ausgelesen werden. Ein direkter Zugriff ist nicht möglich.

Der Speicherbedarf der Laufzeitstruktur, so wie sie von mir definiert wurde, beträgt für jeden möglichen Socket 44 Byte. Da 2^16-1 (65535) Sockets möglich sind, braucht diese Struktur insgesamt 2883540 Bytes (ca. 2,9 Mbyte).

Nach Ende dieser Initialierung der Monitor DLL, und ab dem zweiten Aufruf sofort, übergibt der Application Layer Active Network LSP die weitere Arbeit an den darunter liegenden Service Provider (wie es jeder LSP nach getaner Aufgabe machen sollte).

4.4.2 Anlegen eines Sockets

Um einen Socket anzulegen ruft eine Anwendung die Funktion Socket() oder WSASocket() der Winsock DLL auf. Die Winsock DLL leitet diesen Aufruf an die Funktion WSPSocket(), des für das gewählte Netzwerkprotokoll zuständigen Service Provider, weiter. Der ALAN_LSP ruft, wenn er diesen Aufruf erhält, die Funktion NewID() der Monitor DLL auf damit die Monitor diesen Socket als Verbindungsidentifikation speichern kann. Diese Speicherung erfolgt in der Laufzeitstruktur der Monitor DLL. Hierbei wird der Teil des Verbindungseintrags benutzt der später den Dateinamen des Execution Enviroments aufnimmt, da das Execution Enviroment zu diesem Zeitpunkt noch nicht bestimmbar ist (dazu fehlt noch die Nummer des Protokolls der Anwendungsebene).

Diese ist erst bestimmbar, wenn über diesen Socket tatsächlich eine Verbindung aufgebaut wird. Beim Anlegen eines Socket ist nur Socket, Protokollfamilie, Sockettype (Stream oder Datagramm) und Protokoll verfügbar (siehe auch Abbildung 4.6).

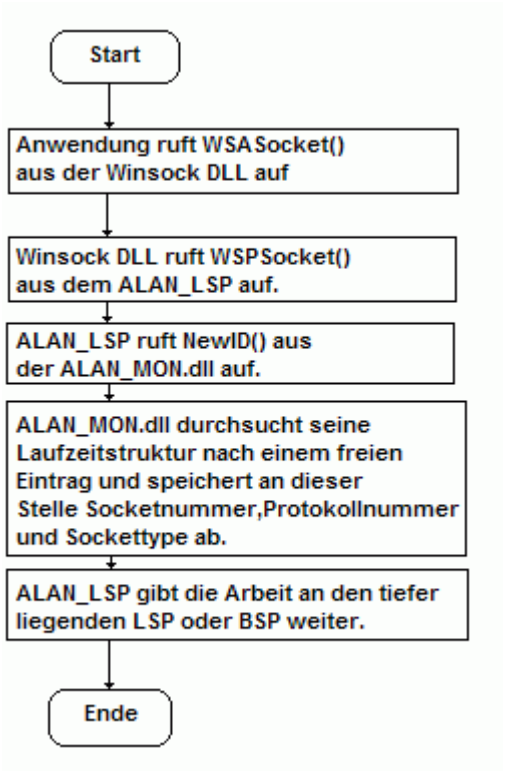

Abbildung 4.6 Anlegen eines Socket

Bei TCP/IP ist dies:

Protokollfamilie:	AF_INET (IPv4) oder
	AF_INET6 (IPv6)
Sockettype:	SOCK_STREAM (TCP) oder
	SOCK_DGRAM (UDP) oder
	SOCK_RAW (z.B. für ICMP)
Protokoll:	IPPROTO_TCP (TCP) oder
	IPPROTO_UDP (UDP)
	IPPROTO_ICMP (ICMP)

4.4.3 Binden eines Sockets an eine Netzwerkadresse

Um den Socket an eine eindeutige Netzwerkadresse zu binden, ruft die Netzwerkanwendung die Funktion Bind() aus der Winsock DLL auf. Diese leitet diesen Aufruf an den zuständigen LSP weiter, indem sie die Funktion WSPBind() aus diesem aufruft. Ist dies der ALAN_LSP so ruft dieser die Funktion OpenID() aus der Monitor DLL auf. Die Monitor DLL durchsucht daraufhin ihre Laufzeitstruktur nach den bei Anlegen des Sockets gespeicherten Informationen (siehe Kapitel 4.3.2) und sucht anschließend mit Hilfe dieser Informationen in der Windows Registery nach dem Dateinamen des Application Layer Active Network Active Protokoll Helpers der in der Lage ist die Socketadressstruktur zu analysieren (siehe Abbildung 4.5).

Wird kein passender Protokoll Helper gefunden, so werden alle in der Laufzeitstruktur zu dieser Verbindung gespeicherten Informationen verworfen, da eine Modifikation der Daten bei Verwendung von auf Anwendungsebene unbekannten Protokollen nicht möglich ist.

Wird dagegen ein passender Protokoll Helper gefunden, so wird diesem die Socketadressstruktur übergeben. Der Protokoll Helper sucht aus dieser Struktur nun die Nummer des auf Anwendungsebene verwendeten Protokolls heraus.

Bei TCP/IP (sowohl bei Version 4 als auch bei Version 6) handelt es sich hierbei um die Portnummer. Einige bekannte Protokolle der Anwendungsebene und ihre TCP-Portnummer finden sich in Tabelle 4.2 .

Mit dem Rückgabewert des Protokoll Helpers und den in der Laufzeitstruktur hinterlegten Informationen sucht nun die Monitor DLL das für diese Verbindung zu verwendene Execution Enviroment.

Findet die Monitor DLL kein Execution Enviroment das für die eingesetzen Protokolle verwendbar ist, so werden alle Informationen die in der Laufzeitstruktur hinterlegt sind verworfen, anderseits wird der Dateiname der Execution Enviroments in der Laufzeitstruktur für die weitere Verwendung hinterlegt.

ftp-data	20/tcp	File Transfer [Default Data]
ftp	21/tcp	File Transfer [Control]
ssh	22/tcp	SSH Remote Login Protocol
telnet	23/tcp	Telnet
domain	53/tcp	Domain Name Server
domain	53/udp	Domain Name Server
http	80/tcp	World Wide Web HTTP
pop3	110/tcp	Post Office Protocol - Version 3
nntp	119/tcp	Network News Transfer Protocol
imap	143/tcp	Internet Message Access Protocol
Imaps	993/tcp	imap4 protocol over TLS/SSL
pop3s	995/tcp	pop3 protocol over TLS/SSL (was spop3)
Active-net	3322-3325	Active Networks

Tabelle 4.2 Protokolle und Ports

(Auszüge aus [IA04])

Die Arbeit der Monitor DLL für diesen Teilschritt ist damit beendet und die Ausführung geht an den ALAN_LSP zurück, der zuletzt noch die Funktion WSPBind() des unter im liegenden Service Provider aufruft (siehe auch Abbildung 4.7).

4.4.4 Aufnahme einer Verbindung durch einen Client

Eine Netzwerkclient-Anwendung ruft um sich mit einem Server zu verbinden, die Funktion connect() der Winsock DLL auf. Die Winsock DLL leitet diesen Aufruf an die Funktion WSPConnect() des LSP weiter. Gelangt dieser Aufruf an den ALAN_LSP so ruft dieser die Funktion OpenID() der Monitor DLL auf. Die Monitor DLL sucht daraufhin in ihrer Laufzeitstruktur nach den beim Anlegen des Sockets (siehe Kapitel 4.3.2) hinterlegten Informationen und übergibt die Socketadressstruktur die ursprünglich der Connect()-Funktion der Winsock DLL übergeben wurde, an den für das verwendete Protokoll geeigneten Protokol Helper. Die Bestimmung desselben geschied genauso wie im Kapitel 4.3.3 bezüglich Serverzugriffe beschrieben. Der einzige Unterschied besteht darin, das der Funktion bind() die eigne Adresse inkl. Portnummer (bei TCP/IP) übergeben wird. connect() dagegen bekommt die Adresse inkl. Portnummer (TCP/IP) des entfernten Servers zu dem die Verbindung aufgebaut werden soll übergeben. Damit durchlaufen nur

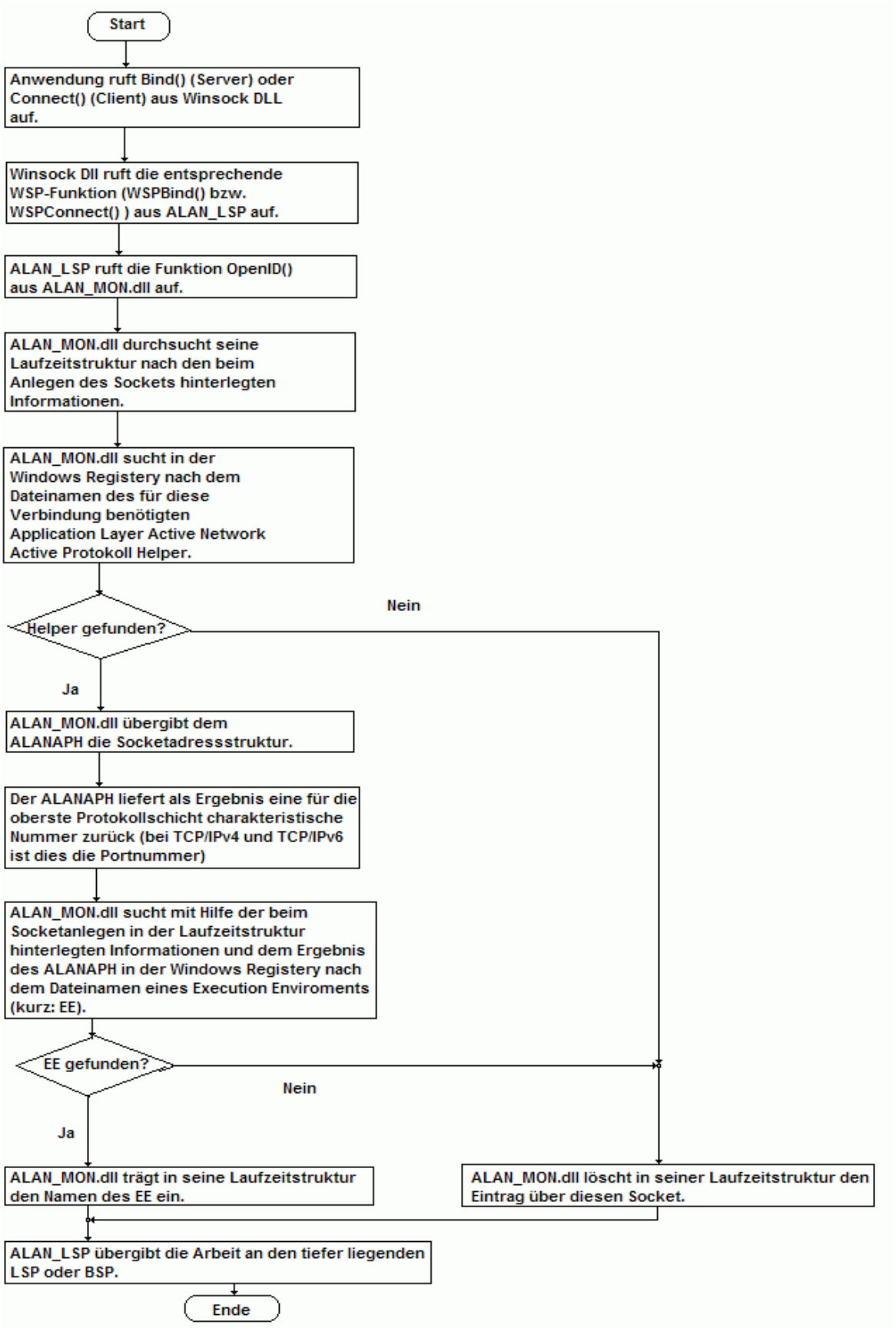

Abbildung 4.7 Ablaufdiagramm Bind und Connect

fest definierbare Adress-Informationen die Monitor DLL und alle weiteren Komponenten der Application Layer Active Network Systemarchitektur für Windows. Insbesondere bei Verwendung von TCP/IP wo Netzwerk-Clients dynamisch zugewiesene Portnummern einsetzen und nur Netzwerk-Serveranwendungen feste Portnummern verwenden. Die Monitor DLL sieht ausschließlich feste Portnummern der Server. Daher reicht es die bekannten Portnummern der Server in die Zweige der Windows Registery einzutragen die diese Systemarchitektur zu Konfigurationszwecken einsetzt. Eine Konfigurationsänderung dieser Systemarchitektur ist bei Netzwerk-Clientanwendungen nicht notwendig.
Für andere Protokolle als TCP/IP gilt das im letzen Abschnitt geschilderte nicht direkt, aber entsprechend. Daher verzichte ich auf weiteres Eingehen auf andere Protokolle, zumal ich keine Protokoll Helper für andere Protokolle als TCP/IPv4 und TCP/IPv6 erstellt habe.
Das weitere Vorgehen der Monitor besteht wie auch schon in Kapitel 4.3.3 geschildert darin das ein Execution Enviroment gesucht wird und der Dateiname desselben in der Laufzeitstruktur eingetragen wird und danach die Arbeit an den darunterliegenden Service Provider übergeben wird (siehe Abbildung 4.7).

4.4.5 Annehmen einer Netzwerkverbindung

Um eingehende Verbindungswünsche anzunehmen ruft eine Netzwerk-Serveranwendung die Funktion accept() der Winsock DLL auf.

Diesen Aufruf leitet die Winsock DLL an die Funktion WSPAccept() eines Service Providers weiter. Ist dies der ALAN_LSP so erstellt dieser einen neuen Socket um die eingehende Verbindung zu verwalten. Dieser Socket wird nicht nur an die Winsock DLL oder einen darüberliegenden Service Provider zurückgeben, sondern auch der Funktion CopyID() der Monitor DLL übergeben.
Die Monitor DLL durchsucht nun ihre Laufzeitstruktur nach den Socketinformationen des Sockets der mit bind() an eine Adresse gebunden wurde. Accept() wurde dieser Socket als Parameter übergeben und ALAN_LSP hat ihn als Parameter von CopyID() an die Monitor DLL weitergegeben.
Findet die Monitor DLL die Socketinformationen so kann auch die Daten der anzunehmende Verbindung modifiziert werden. Daher legt die Monitor DLL nun eine Kopie der Socketinformation an. Dabei wird als Socket allerdings der gerade neu erzeugte

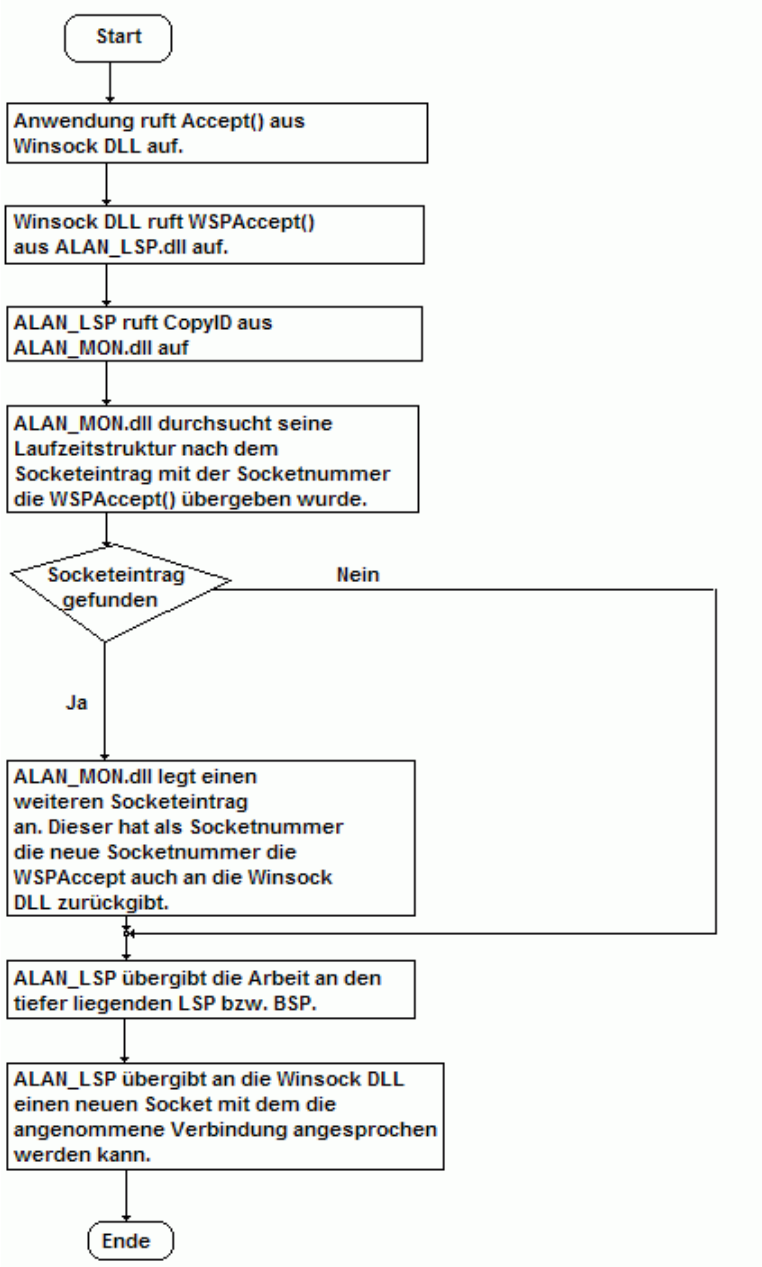

Abbildung 4.8 Annahme einer Verbindung

Socket eingetragen. Damit ist auch der neuen Verbindung dasselbe Execution Enviroment zugeordnet wie allen anderen Verbindungen dieser Serveranwendung.

Findet die Monitor DLL keine Socketinformationen so können die Daten der anzunehmenden Verbindung nicht modifiziert werden. Diesen Sachverhalt hat die Monitor DLL bereits beim Binden des ursprünglichen Sockets an die Netzwerkadresse erkannt und daher die Socketinformationen in der Laufzeitstruktur verworfen. Die Konsequenz daraus ist, das für diese Verbindung sich die Active Network Systemarchitektur genauso verhält, wie als wenn sie nicht vorhanden wäre, und nur die normalen Bestandteile des Windows Netzwerkstabels im Einsatz kommen würden.

In jedem Fall aber übergibt der ALAN_LSP die Aufgaben an die darunter liegenden Service Provider und liefert am die Winsock DLL einen Socket zurück unter dem die angenommene Verbindung angesprochen werden kann.

4.4.6 Daten senden

Um nach dem Aufbau der Verbindung Daten zu senden, ruft eine Netzwerkanwendung die Funktion send() oder WSASend() aus der Winsock DLL. Beide Funktionen leiten ihren Aufruf an die Funktion WSPSend() des zuständigen Service Providers weiter. Ist dies der ALAN_LSP so ruft dieser die Funktion Active_Network_Send() der Monitor DLL auf.
Daraufhin durchsucht die Monitor DLL ihre Laufzeitstruktur nach dem Dateinamen des für diese Verbindung zu verwendenen Execution Enviroment.
Wird kein Eintrag zu dieser Verbindung in der Laufzeitstruktur gefunden, so übergibt die Monitor DLL die Sendebuffer direkt unverändert zurück an den ALAN_LSP. Vorher wird die Gesamtgröße der Sendebuffer, in der Statistik der Monitor DLL, als nicht veränderte Bytes hochgezählt.
Wird dagegen ein Eintrag zu dieser Verbindung gefunden, so wird die Größe der Sendebuffer zu den rausgehend zu modifizierenden Bytes sowohl in der Statistik der Monitor DLL als auch im Statistikteil des Eintrags in der Laufzeitstruktur (siehe Abbildung 4.5). Ist die aktuelle Sendebufferstruktur die allererste zu dieser Verbindung gehörende Sendebufferstruktur so bedeutet dies das noch keine Messung erreichbarer Datenübertragungsraten existiert und auch keine Messung der Datenrate mit der die Anwendung versucht, Daten zu senden. Daher wird notgedrungen angenommen das mit beliebig hoher Datenrate gesendet werden kann. Es wird daher die Gesamtgröße der Sendebuffer auch als neue Gesamtgröße der Sendebuffer angenommen.
Wenn die aktuelle Sendebufferstruktur nicht die allererste ist, so existiert zumindest eine Zeitangabe darüber wann die letzte Sendebufferstruktur mit welcher Größe gesendet wurde. Über die aktuelle Zeit lässt sich damit die Rate bestimmen mit der die Anwendung versucht zu senden (siehe Abbildung 4.9).

$$\frac{\text{Größe\_des\_letzten\_Sendebuffers}}{\text{aktuelle\_Zeit - letzte\_gespeicherte\_Zeit}} = \text{Senderate\_Anwendung}$$

Abbildung 4.9 Berechnung der Rate mit der eine Anwendung versucht zu senden

Leider liegt nicht unbedingt schon beim Senden der zweiten Sendebufferstruktur ein Meßwert der tatsächlich erreichbaren Datenrate vor, den ein Aufruf der Winsock DLL und auch eines Service Providers um Daten zu senden oder zu empfangen kann sowohl als sogenannter Blocking Call (dann kommt die Anwendung erst wieder nach Ende der

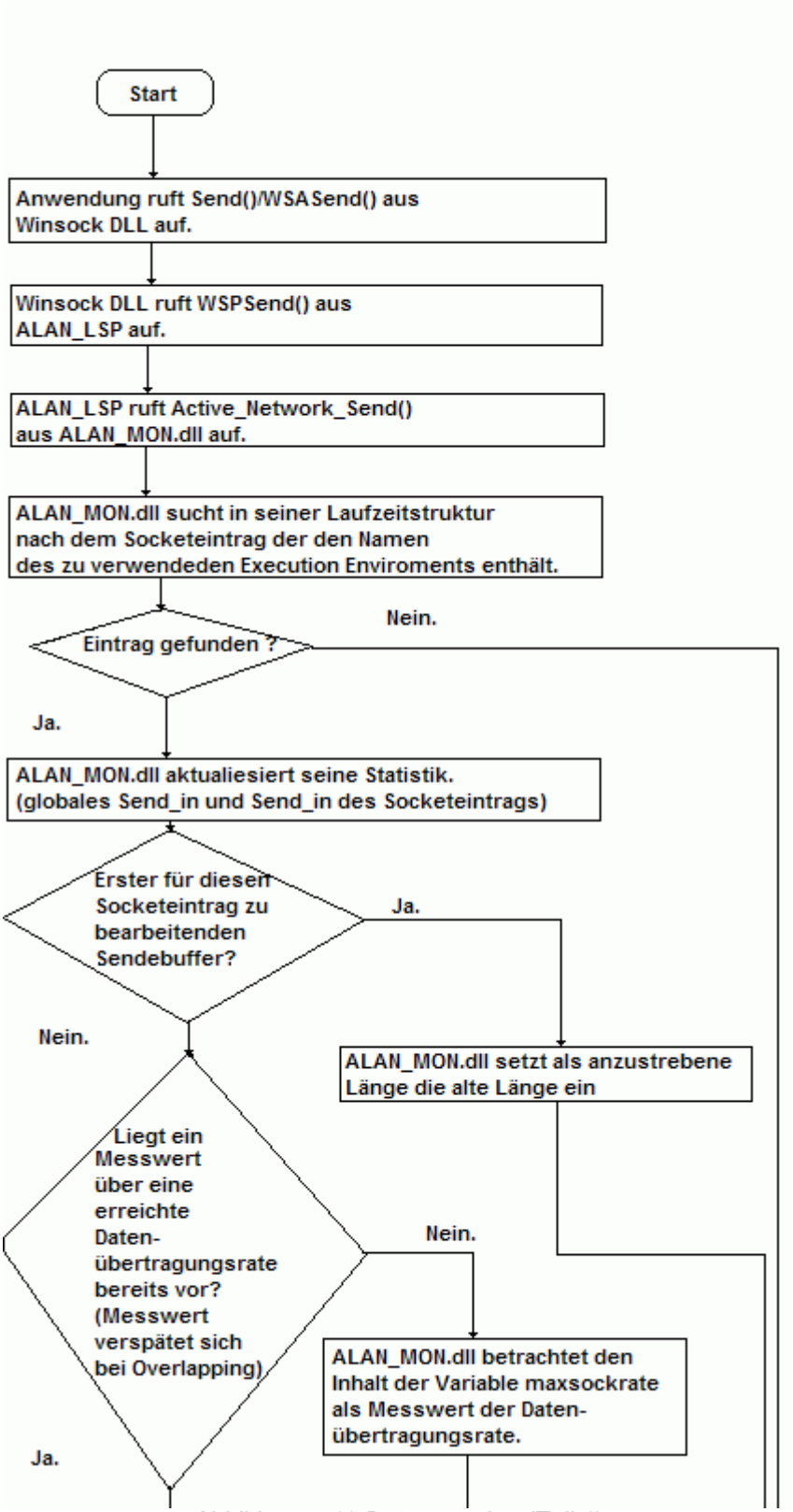

Abbildung 4.10 Daten senden (Teil 1)

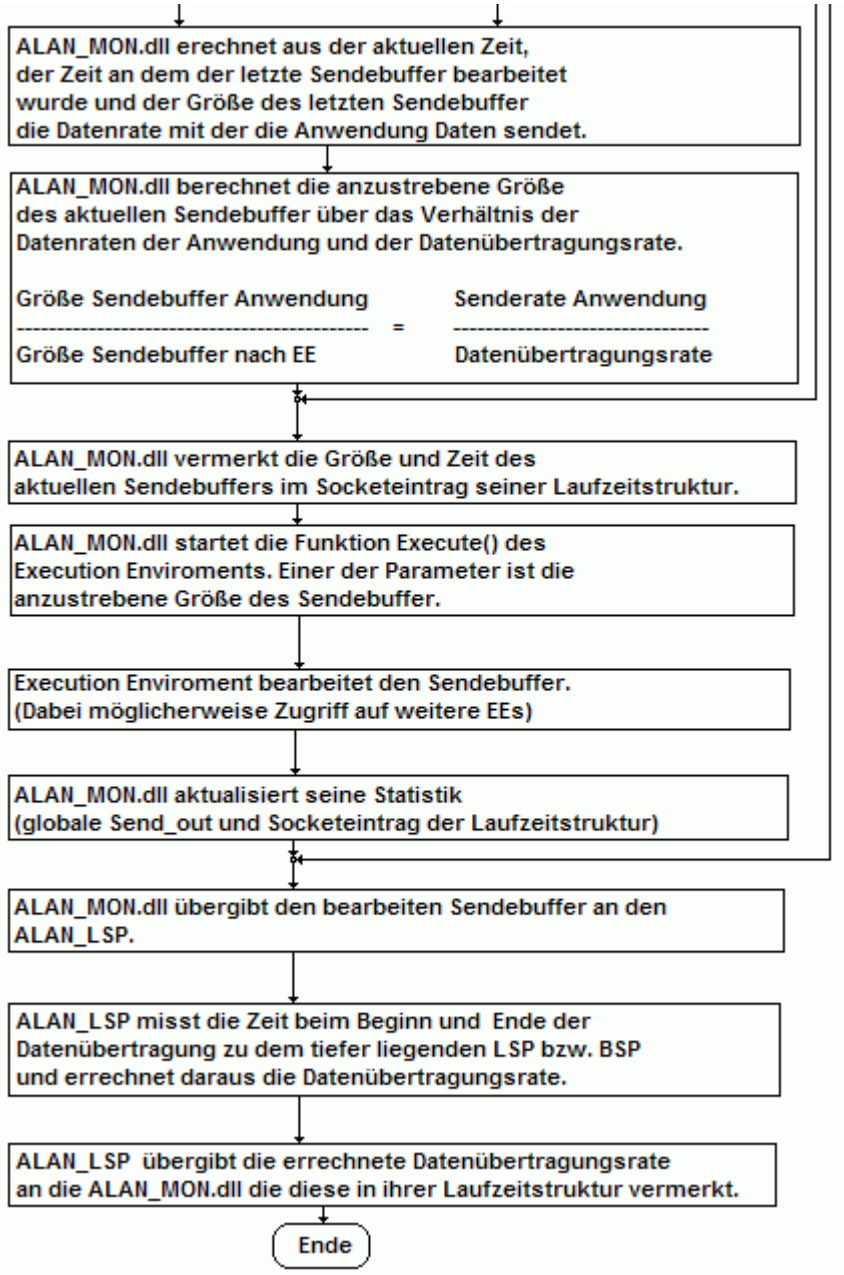

Abbildung 4.11 Daten senden (Teil 2)

Übertragung wieder zum Zuge und kann daher auch weitere Datenbuffer erst danach auf den Weg bringen) oder als sogenannter Overlapping Call (dann wird die Abwendung direkt weiter ausgeführt, die Datenübertragung erfolgt im Hintergrund).

Liegt kein Meßwert der tatsächlich erreichbaren Datenrate vor, weil Overlapping Calls verwendet wurden und noch keiner beendet wurde, so benutzt die Monitor DLL ersatzweise den Konfigurationswert maxsockrate. Dieser kann durch eine Monitor Anwendung modifiziert werden. Ohne Monitor Anwendung ist dieser Wert so gesetzt das er eine Datenleitung mit 128 kbit/sec Senderate (z.B. ISDN mit Kanalbündelung von 2 Basiskanälen oder T-DSL 1000[5] ohne erhöhten Upstream) annimmt.

5 Die Produktnamen der T-Com bezüglich T-DSL bezeichnen den Downstream, also die Datenrate in

Aus dem Meßwert der erreichten Übertragungsrate, der Datenrate mit der die Anwendung versucht zu senden, sowie der Gesamtgröße der aktuellen Sendebuffer errechnet die Monitor DLL die anzustrebene neue Gesamtgröße der Sendebuffer (siehe Abbildung 4.11).

Für weitere Sendevorgänge wird die Größe und Zeit der aktuellen Sendebuffer in die Laufzeitstruktur eingetragen. Anschließend ruft die Monitor DLL die Funktion Execute() des in der Laufzeitstruktur vermerkten Execution Enviroment auf. Die Funktionsweise dieser Funktion wird in einem eignen Unterkapitel behandelt.

Nachdem das Execution Enviroment die Sendebuffer (die vorher zu einem großen Buffer zusamengefasst wurden) bearbeitet hat, addiert die Monitor DLL in die Größe des resultierenden Buffers zur Größe aller zum Senden modifizierten Buffer. Dies erfolgt sowohl in der Laufzeitstruktur als auch in der Gesamtstatistik der Monitor DLL.

Bevor die Monitor DLL den modifizierten Buffer an den ALAN_LSP zurückgibt, wird dieser wieder in die einzelnen Sendebuffer aufgeteilt. Dabei muß berücksichtigt werden, das die Gesamtgröße aller Sendebuffer üblicherweise kleiner geworden ist. Dies bedeutet insbesondere das Sendebuffer nicht mehr voll ausgenutzt werden oder sogar ganz leer sein können. Die Größen der Sendebuffer müssen daher angepasst werden.

Da der Funktion WSPSend() bzw. WSASend()/send() die Sendebuffer als Area von WSABUF-Strukturen übergeben wird, bedeutet das Anpassen der Sendebuffer-Größen, das Anpassen der WSABUF-Strukturen (siehe Abbildung 4.12).

```
typedef struct _WSABUF {
  u_long  len;
  char FAR  *buf;
} WSABUF, FAR * LPWSABUF;
```

Abbildung 4.12 WSABUF-Struktur (aus [MI02])

Eine WSABUF-Struktur (die vom ALAN_LSP auch genauso an die Monitor DLL weitergegeben wird, gesteht aus der Größe des Buffers und einem Zeiger auf den Buffer, also der Adresse des Buffers (siehe [MI02]).

Zuletzt sendet der ALAN_LSP die modifizierten Sendebuffer an den darunterliegenden Serviceprovider und misst dabei die erzielbare Datenrate, welche an die Monitor DLL übergeben wird, um sie für die als nächstes zu übertragenden Daten in der Laufzeitstruktur zu speichern.

Empfangsrichtung, hier geht es aber um die Datenrate in Senderichtung, auch Upstream genannt.

4.4.7 Daten empfangen

Um Daten zu senden ruft eine Netzwerkanwendung die Funktion WSARecv() oder Received() aus der Winsock DLL auf. Dieser Aufruf wird an die Funktion WSPRecv() weitergeleitet. Der ALAN_LSP leitet diesen Aufruf an den darunter liegenden Service Provider weiter um den Datenempfang durchzuführen. Nachdem der eigendliche Datenempfang vollzogen ist ruft der ALAN_LSP die Funktion Active_Network_Receive() auf. Diese Funktion ist für die reine Funktion der Application Layer Active Networks nicht notwendig, erlaubt allerdings weitere Einsatzbereiche, die auf der Modifikation von Daten basieren. Dazu kommt wie auch beim Senden von Daten ein Execution Enviroment zum Einsatz. Allerdings wird diesem im Gegensatz zum Senden von Daten keine errechnete neue Buffergröße übergeben, sondern als neue Buffergröße die alte Buffergröße. Dies geschied deshalb, da beim Empfang von Daten eine Verringerung der Datenmenge nicht notwendig ist, ein Execution Enviroment allerdings immer als Parameter eine neue Buffergröße erwartet.

Die Funktionalität der Monitor DLL hinter der Funktion Active_Network_Receive() ansonsten nahezu identisch mit der Funktion Active_Network_Send() welche beim Senden eingesetzt wird. Da keine neue Buffergröße errechnet werden muß, erfolgt allerdings auch keine Berechnung von Datenübertragungsraten oder -zeiten. Die Statistikwerte werden dagegen einfach in einem eignen Satz von Werten gespeichert. (Siehe Tabelle). Die Datenbuffer liegen auch beim Empfang von Daten in der Form von WSABUF-Strukturen (siehe Abbildung 4.12) vor und müssen daher auch beim Empfang für den Aufruf eines Execution Enviroment zusammengefasst werden.

4.5 Möglichkeiten eines Execution Enviroment

Ein Execution Enviroment bekommt als Parameter einen Zeiger auf einen Datenbuffer, die Größe des Datenbuffers, die anzustrebene neue Größe des Datenbuffers und die Datenübertragungsrichtung (als ob die Daten gerade zum Senden bereitgestellt werden oder gerade empfangen wurden und an die Netzwerkanwendung weitergeleitet werden sollen). Als Ergebnis liefert eine Execution Enviroment immer die neue Größe des Datenbuffers zurück. Der Datenbuffer selbst wird direkt an der ursprünglichen Stelle modifiziert, da er als Zeiger übergeben wurde. Daher braucht ein Execution Enviroment keinen Datenbuffer als Ausgabeparameter, sondern nur als Eingabeparameter.

Die Möglichkeiten eines Execution Enviroment sind dabei vielfätig, wie es auch die 3 Beispiel Execution Enviroments zeigen:

EE_NOP.dll (NOP = No Operation => Keine Funktion) enthält nur die Aufrufschnittstelle und gibt ansonsten einfach die alte Größe des Datenbuffers, als neue Größe des Datenbuffers zurück. Eine Beeinflußung des Datenbuffers geschied nicht. Dieses Execution Enviroment eignet sich daher vorallem zur Darstellung der Aufrufschnittstelle eines Execution Enviroments (das heißt also des Application Layer Active Network Execution Enviroment API; siehe Überblick in Abbildung 4.1), aber auch zu Testzwecken, da es keine Beeinflußungen des Datenbuffers durchführt. Für den normalen Betrieb der hier vorgestellten Systemarchitektur ist ein Execution Enviroment, das nie eine Datenmodifikation durchführt nicht notwendig, da Daten auch dann nicht modifiziert werden, wenn gar kein Execution Enviroment für eine betreffende Verbindung gefunden werden kann.

EE_ANEP.dll (ANEP = Active Network Encapsulation Protokoll) demonstriert die Möglichkeit eines Execution Enviroment ein anderes auszusuchen und indirekt über die Monitor DLL aufzurufen. Dies erlaubt es, das ein Execution Enviroment eine Voranalyse der Daten durchführt. Und mit Hilfe des Ergebnis dieser Voranalyse ein für die tatsächlich angetroffenen Daten spezialisiertes Execution Enviroment aufruft. Dies wird auch genau bei der EE_ANEP.dll durchgeführt. (ANEP siehe Kapitel 2.2). Die EE_ANEP.dll sucht die Type-ID aus dem Datenbuffer raus und sucht mit Hilfe dieser Type-ID ein für diese Type-ID spezialisiertes EE aus. Die eigentliche Datenmodifikation geschied durch das spezialisierte EE. Existiert kein passendes spezialisiertes EE so erfolgt keine Datenmodifikation.

EE_NAMH.dll (NAMH = Null Adress Maschine in Havard-Architektur) ist eine einfache virtuelle Maschine die eine Nulladress Maschine implementiert, ähnlich derjenigen wie sie im Skript Datenverarbeitung I im Kapitel Mikroprogrammierung [GE98] beschrieben ist.
Allerdings benutzt die hier benutzte Nulladress Maschine die Havardarchitektur, dies bedeutet das alle Speicherbereiche getrennt voneinander sind (Code,Daten und Stack) im Gegensatz zur von Neumann Architektur welche für Code, Daten und Stack nur einen gemeinsamen Speicher vorsieht. Außerdem mußten einige Befehle(Opcodes) zusätzlich eingeführt werden um Übergabewerte in die virtuelle Maschine einzugeben und Rückgabewerte auszugeben. Dies ist ein Befehl um die Länge des Datenbuffers auf den

Stack zu schreiben (PUSH len), einen um die gewünschte neue Länge auf den Stack zu schreiben (PUSH newlen) und einen um die tatsächliche neue Länge vom Stack zu laden und die virtuelle Maschine zu beenden (POP ret). Dabei sind len,newlen und ret wie Prozessorregister für besondere Zwecke aufzufassen. Sie verändern oder lesen dabei direkt die entsprechenden Übergabe- bzw. Rückgabewerte der EE_NAMH.dll.

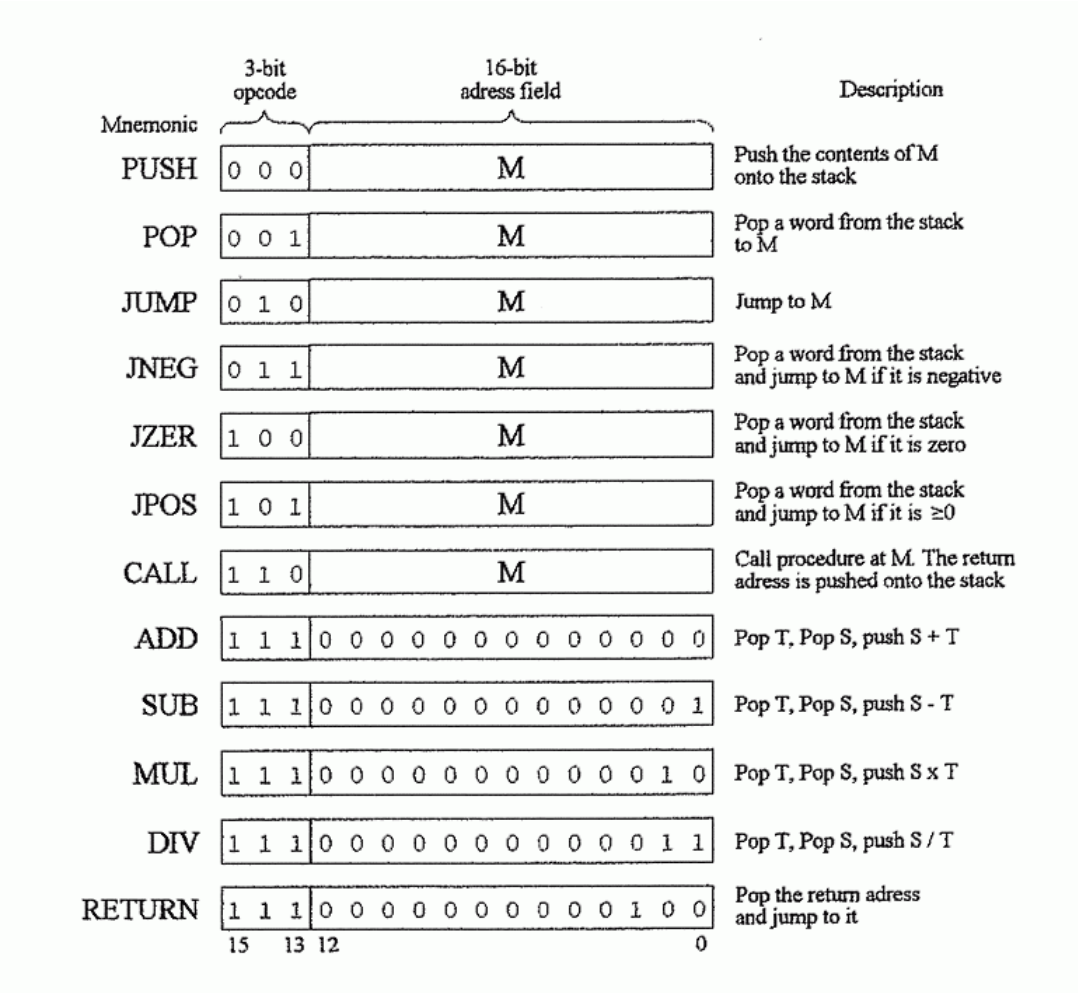

Abbildung 4.13 Befehle der Nulladress-Maschine (aus [GE98] S.47)

Zusätzliche Befehle der NAMH:

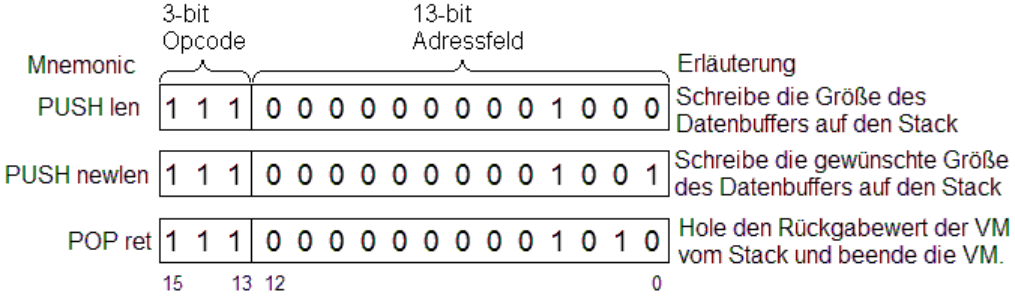

Abbildung 4.14 Zusatzbefehle der NAMH

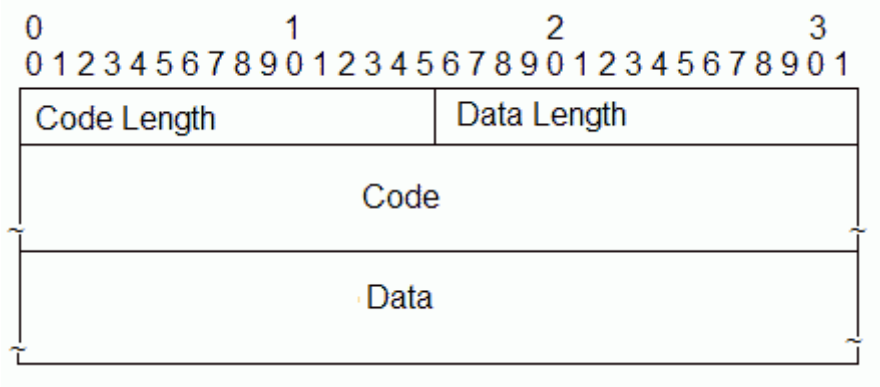

Abbildung 4.15 Paketformat der NAMH

Das Execution Enviroment EE_NAMH.dll erwartet als Datenbuffer ein ANEP-Packet (siehe Kapitel 2.2) dessen Nutzinhalt den in Abbildung 4.15 abgebildeten Aufbau entspricht. Länge des Code als vorzeichenlose Ganzzahl in Big Endian Notation, gefolgt von der Länge der Daten in Big Endian Notation. Anschließend der Code (sofern Codelänge >0 ist) und zuletzt die Daten (sofern Datenlänge >0). Dabei muß mindestens eine der beiden Längen größer 0 sein, um sinnvoll bearbeitet zu werden. Das Execution Enviroment fängt allerdings fehlerhafte Datenbuffer ab und versucht sie nicht zu bearbeiten. Wird nur Code gesendet, so wird dieser für die spätere Verwendung mit Hilfe der zu diesem Zweck eingerichteten Funktion in der Monitor DLL gespeichert. Wird dagegen nur Daten gesendet, so wird zum Bearbeiten dieser Daten der vorher gespeicherte Code verwendet. Da die NAMH nur 13 Bit für die Adressierung bereitstellt, ist die Größe eines damit bearbeitbaren Datenbuffers auf 8192 Bytes beschränkt. Der Code der zur Bearbeitung verwendet wird, kann dabei ebenfalls maximal 8192 Bytes groß sein. Diese Größen sind aufgrund der Havard-Architektur unabhängig voneinander zu sehen und die Beschränkung besteht nicht darin das für Code+Daten nur 8192 Bytes zur Verfügung stehen. Dies gilt nur für dieses spezielle EE. Die Schnittstelle zwischen Monitor DLL und Execution Enviroment erlaubt grundsätzlich Datenbuffer bis zu 2^32 Bytes (=4 GByte). Dies muß allerdings nicht für alle Execution Enviroments intern gelten. Kann ein Execution Enviroment keine 4 GByte bearbeiten, so muß es dennoch in der Lage sein mit einer übergebenen Datenbuffergröße von bis zu 4 GByte fehlerfrei umgehen zu können. Dies bedeutet das die Größe des Datenbuffers getestet werden muß und bei einem zu großen Datenbuffer das Execution Enviroment einfach nur als neue Länge die alte Länge ausgibt und ansonsten keine Modifikation durchführt. Dies ist notwendig um sicherzustellen, das unter keinen Umständen Daten einfach zerstört werden oder die Systemstabilität durch Bufferüberläufe beeinträchtigt wird.

4.6 Schnittstellenbeschreibungen

In diesem Unterkapitel werden alle internen und externen Schnittstellen der Application Layer Active Network Systemarchitektur für Windows beschrieben. Teilweise wurden sie bereits in früheren Kapiteln angerissen, hier finden sich allerdings systematisch alle Schnittstellen erkläutert.

4.6.1 Windows Sockets Application Programming Interface (API)

Das Windows Sockets API ist die Schnittstelle mit der eine Netzwerkanwendung auf die Winsock DLL zugreift. Die in dieser Arbeit beschriebene Systemarchitektur verändert diese Schnittstelle nicht. Da sie aber in dieser Arbeit noch nicht detaliert beschrieben wurde , aber nicht unwichtig ist , hier ihre Funktionen (siehe [MI03] Windows Sockets Functions) in einem kurzen Überblick:

Funktionskatalog:

accept()	Nimmt einen reinkommenden Verbindungswunsch an einen Socket an
AcceptEx()	Nimmt den reinkommende Verbindungswunsch an und liefert dabei sowohl die Adresse des lokalen und des entfernten Systems zurück. Außerdem empfängt diese Funktion den ersten Block von Daten die, der Client gesendet hatte.
bind()	Verbindet einen Socket mit einer lokalen Netzwerkadresse
closesocket()	Schließt einen vorhandenen Socket.
connect()	Stellt über eine bestimmten Socket eine Verbindung her
ConnectEx()	Stellt eine Verbindung her und sendet einen ersten Block an Daten
DisconnectEx()	Beendet eine Verbindung, so das der verwendete Socket weiter verwendet werden kann.
freeaddrinfo()	Gibt eine Adressinformation die mit getaddrinfo() zugewiesen wurde wieder frei.
gai_strerror()	Hilft bei der Ausgabe von Fehlermeldungen

GetAcceptExSockaddrs()	Analysiert die Ausgabe von AcceptEx() und liefert dabei die Adressinformationen des lokalen und entfernten Systems zurück.
GetAddressByName()	Liefert Netzwerkadressinformationen zurück.
getaddrinfo()	Protokollunabhängiges Wandeln eines Hostnames in eine Netzwerkadresse
gethostbyaddr()	Liefert den Hostnamen zu einer Netzwerkadresse.
gethostbyname()	Liefert den Hostnamen aus einer Hostdatenbank zurück
gethostname()	Liefert den Hostnamen des lokalen Computers zurück
getnameinfo()	Namesauflösung von einer Netzwerkadresse zu einem Hostnamen
getpeername()	Liefert zu einem Socket die Adressinformationen des entfernten Endes
getprotobyname()	Liefert die Protokollinformationen zu einem Protokollnamen zurück
getprotobynumber()	Liefert die Protokollinformationen zu einer Protokollnumber zurück
getservbyname()	Liefert Serviceinformationen zu einem Servicenamen und Protokoll
getservbyport()	dito zu einem Port und Protokoll
getsockname()	Liefert zu einem Socket den lokalen Hostnamen
getsockopt()	Liefert die zu einem Socket gesetzten Optionen zurück
htonl()	Wandelt eine Zahl vom Typ vorzeichenlose lange Ganzzahl (u_long) von der Bytereihenfolge des lokalen Computers (bei x86 ist dies Little Endian) in die Bytereihenfolge von TCP/IP (Network Byte Order oder auch Big Endian genannt)
htons()	dito für eine Zahl vom Typ vorzeichenlose kurze Ganzzahl (u_short)
inet_addr()	Wandelt eine Zeichenkette (String) der eine IPv4 in Punktschreibweise (z.B. 127.0.0.1) enthält in die Binärform um, wie sie für diverse andere Funktionen der Winsock DLL nötig ist.
inet_ntoa()	Wandelt eine IP-Adresse im Binärformat, in die Punktschreibweise um

ioctlsocket()	Steuert den Ein/Ausgabemodus eines Sockets
listen()	Bringt einen Socket in den Horchstatus
ntohl()	Wandelt eine Zahl vom Typ vorzeichenlose lange Ganzzahl in die Bytereihenfolge des lokalen Computers
ntohs()	dito für eine Zahl vom Typ kurze Ganzzahl
recv()	Empfängt Daten über einen Socket
recvfrom()	Empfängt ein Datagram und speichert die Quelladresse
select()	Stellt den Status eines oder mehrerer Sockets fest
send()	Sendet Daten über einen Socket
sendto()	Sendet ein Datagram
setsockopt()	Setzt die Optionen zu einem Socket
shutdown()	Beendet die Datenübertragung eines Sockets
socket()	Erzeugt einen Socket der an einen bestimmten Service Provider gebunden ist
TransmitFile()	Überträgt eine Datei über einen Socket
TransmitPackets()	Überträgt im Speicher befindliche Daten über einen Socket
WSAAccept()	wie Accept, aber mit Zusatzfunktionen wie z.B. QoS
WSAAddressToString()	wandelt alle Komponenten einer SOCKADDR-Struktur in ein menschenlesbares Zeichenketten-Format
WSAAsyncGetHostByAddr()	wie GetHostbyaddr, aber asynchron
WSAAsyncGetHostByName()	wie GetHostbyName, aber asynchron
WSAAsyncGetProtoByName()	wie GetProtobyname, aber asynchron
WSAAsyncGetProtoByNumber()	wie GetProtobynumber, aber asynchron
WSAAsyncGetServByName()	wie GetServbyname, aber asynchron
WSAAsyncGetServByPort()	wie GetServbyport, aber asynchron
WSAAsyncSelect()	Fordert Windows nachrichtenbasierte Benachrichtigungen zu Netzwerkereignissen für einen Socket an
WSACancelAsyncRequest()	Bricht einen noch nicht vollendeten asynchronen Netzwerkzugriff ab.
WSACleanup()	Beendet die Nutzung der Winsock DLL durch eine Netzwerkanwendung
WSACloseEvent()	Beendet einen Ereignis Objekt Handler

WSAConnect()	wie connect(), aber mit QoS-Optionen
WSACreateEvent()	Erzeugt einen Ereignis Objekt Handler
WSADuplicateSocket()	Erzeugt eine Struktur, mit der ein neuer Socket Beschreiber für ein teilbaren Socket erzeugt werden kann.
WSAEnumNameSpace Providers()	Liefert Informationen über vorhandene Namesräume
WSAEnumNetworkEvents()	Durchsucht zu einem Socket die aufgetrettenen Netzwerkereignisse und löscht alle aufgezeichneten Netzwerkereignisse.
WSAEnumProtocols()	Liefert Informationen zu allen verfügbaren Transportprotokollen
WSAEventSelect()	Spezifiziert eine Ereignisobjekt mit einem Satz von Netzwerk Ereignissen
__WSAFDIsSet()	Spezifiziert ob ein Socket in einem Satz von Socket Beschreibern enthalten ist.
WSAGetLastError()	Liefert den Fehlerstatus zum letzten fehlgeschlagenen Zugriff
WSAGetOverlappedResult()	Liefert das Ergebnis eines Overlapping Zugriffs
WSAGetQOSByName()	Eine QOS-Struktur initialisieren
WSAGetServiceClassInfo()	Liefert Classeninformationen zu einem speziellen Service eines Namespace Providers
WSAGetServiceClassName ByClassId()	Liefert den Namen eines Service
WSAHtonl()	Wie Htonl()
WSAHtons()	Wie Htons()
WSAInstallServiceClass()	Registriert eine Service Klasse in einem Namensraum
WSAIoctl()	Kontrolliert den Modus eines Socket
WSAJoinLeaf()	Erklärt die Teilnahme an einer Multipoint Sitzung (Multicast!)
WSALookupServiceBegin()	Initialisiert eine Clientsuche
WSALookupServiceEnd()	Gibt den bei WSALookupServiceBegin() und WSALookupServiceNext() benutzten Handler frei
WSALookupServiceNext()	Liefert die angeforderten Service Informationen

WSANSPIoctl()	Entwickelt für Ein/Ausgabekontrollaufrufe zu einem registrierten Namensraum
WSANtohl()	wie Ntohl()
WSANtohs()	wie Ntohs()
WSAProviderConfigChange()	Benachrichtigt die Anwendung, wenn sich sie Providereinstellung verändert hat
WSARecv()	wie Recv()
WSARecvDisconnect()	beendet eine Netzwerkverbindung und enpfängt dabei noch letzte Daten
WSARecvEx()	ähnlich wie Recv()
WSARecvFrom()	wie Recvfrom()
WSARecvMsg()	Empfängt Daten und Steuerinformationen
WSARemoveServiceClass()	Entfernt eine Serviceklasse
WSAResetEvent()	Setzt den Status eines spezifischen Ereignisobjekts zurück
WSASend()	wie Send()
WSASendDisconnect()	Sendet letzte Daten und beendet dann die Verbindung
WSASendTo()	wie Sendto()
WSASetEvent()	Setzt den Status eines spezifischen Ereignisobjekts
WSASetLastError()	Setzt den Fehlercode
WSASetService()	Registriert oder entfernt aus der Windows Registery einen Service aus einem oder mehreren Namesräumen
WSASocket()	wie Socket()
WSAStartup()	Beginn die Benutzung der Winsock DLL durch eine Anwendung
WSAStringToAddress()	Konvertiert einen nummerischen String in eine SOCKADDR-Struktur
WSAWaitForMultipleEvents()	Liefert einen oder alle spezifizierten Ereignisobjekte zurück die sich im gesetzten Zustand befinden

Folgene Funktionen sind veraltete (aber noch vorhandene) Winsock 1.1 Funktionen:
(sie werden daher hier nur der Vollständigkeit halber genannt, aber nicht erläutert)
EnumProtocols()
GetAddressByName()
GetNameByType()
GetService()
GetTypeByName()
SetService()

Diese Funktionen waren in der Winsock 1.1 enthalten, sind aber in der aktuellen Version 2.2 nicht mehr enthalten:
(sie sind daher auch nur über das Winsock 1.1 API erreichbar)
WSACancelBlockingCall()
WSAIsBlocking()
WSASetBlockingHook()
WSAUnhookBlockingHook()

4.6.2 Windows Sockets 2.0 Service Provider Interface(SPI)

Das Windows Sockets 2.0 Service Provider Interface (siehe [MI02] und [MI03]) wird von jedem Service Provider als Aufrufschnittstelle nach oben angeboten. Der oberste Service Provider bietet es somit der Winsock DLL an. Jeder andere Service Provider seinem Vorgänger in der Kette. Damit ergibt sich die Schichtung der Service Provider. Ein Base Service Provider ist dabei immer der unterste Service Provider in der Kette. Layered Service Provider heißen alle Service Provider die über dem Base Service Provider in der Kette angeordnet sind. (Siehe Abbildung 4.1). Service Provider sind dabei Dynamic Link Librarys (DLL) mit nur einem einzigen direkt exportierten Einsprungpunkt (das heißt nur eine einzige Funktion ist direkt aufrufbar). Dieser Einsprungpunkt gehört immer zur Funktion WSPStartup(). Alle im untenstehenden Funktionskatalog stehenden Funktionen stehen erst nach Aufruf von WSPStartupex() über eine Funktionstabelle zur Verfügung.
Die Aufgabe jeder Funktion ergibt sich meist aus der Tatsache das die meisten Funktionen der Winsock DLL nur den benötigten Service Provider suchen und dann dessen fast gleichnamige Funktion aufrufen. Dabei eine Funktion die in der Winsock DLL xy() oder WSAxy() heißt mit der Funktion WSPxy() eines Service Providers. Daher findet

sich in untenstehendem Funktionskatalog auch keine Funktionsbeschreibung. Sie lässt sich beim Winsock API (Kapitel 4.5.1) entnehmen.
Als Layered Service Provider erwartet der Application Layer Active Network Layered Service Provider somit mindestens einen weiteren Service Provider unter sich in der Kette der Service Provider. Die meisten Funktionen reicht der ALAN_LSP dabei nur an den drunterliegenden Service Provider durch. Einige wenige Funktionen rufen vor dem Durchreichen an den drunterliegenden Service Provider Funktionen der Monitor DLL auf um die Active Network Funktionalität bereitzustellen. Der ALAN_LSP benutzt dabei das mit dieser Systemarchitektur neu eingeführte Application Layer Active Monitor API.

Funktionskatalog eines Windows Sockets 2.0 Service Providers:

WSPAccept()	WSPGetQOSByName()
WSPAddressToString()	WSPIoctl()
WSPAsyncSelect()	WSPJoinLeaf()
WSPBind()	WSPListen()
WSPCancelBlockingCall()	WSPRecv()
WSPCleanup()	WSPRecvDisconnect()
WSPCloseSocket()	WSPRecvFrom()
WSPConnect()	WSPSelect()
WSPDuplicateSocket()	WSPSend()
WSPEnumNetworkEvents()	WSPSendDisconnect()
WSPEventSelect()	WSPSendTo()
WSPGetOverlappedResult()	WSPSetSockOpt()
WSPGetPeerName()	WSPShutdown()
WSPGetSockOpt()	WSPSocket()
WSPGetSockName()	WSPStringToAddress()

WSPAccept() und WSPJoinLeaf() ruft CopyID() aus der Monitor DLL auf. WSPBind() und WSPConnect() ruft dagegen OpenID() auf. WSPCleanup() ruft wenn es die letzte Instanz des ALAN_LSP ist ALANCleanup() auf. WSPCloseSocket() und WSPShutdown() rufen immer CloseID() aus der Monitor DLL auf. WSPSocket() erstellt über den Aufruf von NewID() eine neue Verbindungs-ID. Das Initialisieren des ALAN_LSP mit WSPStartupex() initialisiert über den Aufruf von ALANInit() auch die Monitor DLL. Dies allerdings nur beim ersten Aufruf des ALAN_LSP.

Die Daten sendenen Funktionen WSPSend(), WSPSendto() und WSPSenddisconnect() lassen ihre Daten alle über die Funktion Active_Network_Send() vor dem Senden modifizieren. Ebenso lassen alle Daten empfangenden Funktionen (WSPRecv(), WSPRecvFrom() und WSPRecvDisconnect()) ihre Daten vor dem Weitergeben an die darüberliegenden Softwareschichten erstmal noch durch die Funktion Active_Network_Receive() bearbeiten.

4.6.3 Application Layer Active Network Monitor API

Das Monitor API gliedert sich in 3 Teile, die alle 3 von der Monitor DLL stammen:
Funktionen die dafür bestimmt sind vom Application Layer Active Network Layered Service Provider aufgerufen zu werden, Funktionen die von einer Monitor Anwendung (dem Mangmentsystem) aufgerufen werden können, Hilfsfunktionen die von Execution Enviroments genutzt werden können sowie Funktionen die die Strukturen in der Windows Registery anlegen bzw. dabei helfen.

Funktionen für den Layered Service Provider:

ALANInit()	Initialisiert die Laufzeitstruktur und breitet sie zur ersten Benutzung vor
ALANCleanup()	Räumt die Laufzeitstruktur auf.
NewID()	Erzeugt einen neuen Eintrag in der Laufzeitstruktur
OpenID()	Bestimmt das zu einer Verbindung benötigte Execution Enviroment und trägt es in den mit NewID() erzeugten Eintrag in die Laufzeitstruktur ein.
CloseID()	Entfernt einen nicht mehr benötigten Eintrag aus der Laufzeitstruktur
CopyID()	Kopiert einen Eintrag, der unter einer weiteren Verbindungsnummer benötigt wird (wird bei Servern benötigt die für jeden verbundenen Client einen eignen Socket offen halten, der vom Socket abgeleitet wird, der beim Starten des Servers angelegt wurde)
CleanupAA()	Löscht alle in der Windows Registery hinterlegten Active Applications (dies sind kleine Programme die ein Execution Enviroment hinterlegen kann, um damit für später durchfließende Daten zu verwenden).

Active_Network_Send()	Bearbeitet die übergebenen Datenbuffer vor dem Senden nach Active Network Gesichtspunkten.
Active_Network_Receive()	dito, für empfangene Datenbuffer
Active_Send_Rate()	Funktion um die gemessene Senderate in die Laufzeitstruktur einzutragen und damit bei späteren Sendevorgängen zu berücksichtigen
calcrate()	Berechnet Datenraten und berücksichtigt dabei, das der Millisekundenzäher von Windows alle 49,7 Tage einen Überlauf hat. Dies bedeutet bei Datenübertragungen die kurz vor dem Überlauf beginnen und kurz nach demselben beendet werden, das die Zeitdifferenz nicht einfach durch zeit2-zeit1 berechnet werden kann. Diese Funktion berücksichtigt genau dies.

Funktionen die von einem Management System benutzt werden können:

Interne Variablen:

bytes_send	Gesendete Bytes (über alle Verbindungen)
bytes_pass_send	Gesendete Bytes die nicht modifiziert werden konnten
bytes_in_send	Gesendete Bytes vor dem Modifizieren
bytes_out_send	Gesendete Bytes nach dem Modifizieren
bytes_received	Empfangende Bytes (über alle Verbindungen)
bytes_pass_received	Empfangende Bytes die nicht modifiziert wurden
bytes_in_received	Empfangende Bytes vor dem Modifizieren
bytes_out_received	Empfangende Bytes nach dem Modifizieren
maxnicrate	Maximal mögliche Datenrate, ist die Datenrate mit der die Netzwerkkarte maximal arbeiten kann. Dieser Wert wird für einen Plausibilitätstest verwendet. Alle Messwerte der Datenrate die über diesen Wert liegen, werden als Fehlmessungen gewertet und deshalb durch diesen Wert ersetzt. Standardwert ist 16000 (Bytes/sec) was für der Upstreamrate eines T-DSL 1000 Anschlußes der T-Com entspricht bzw. einer Verbidnung über 2 ISDN-Basiskanäle.

defaultsockrate	Standard-Datenrate mit der gesendet wird bevor bei overlapping ein Messwert existiert. Wird verwendet da bei Overlapping der Sendevorgang unabhängig von der Ausführung der Netzwerkanwendung im Hintergrund erfolgt.

Funktionskatalog:

Die Registerfunktionen erlauben das Hochzählen der Zähler, ohne das diese Zähler durch gleichzeitige Datenübertragungen inkonsitent werden. Obige Variablen sollten daher niemals direkt verändert werden, sondern nur ausgelesen werden. Zum Ändern immer nur die folgenden Funktionen nutzen:

RegisterSend()	Zähler für gesendete Bytes hochzählen.
RegisterReceive()	Zähler für empfangende Bytes hochzählen
RegisterSend_pass()	Zähler für nicht modifizierbare Bytes hochzählen (Senden)
RegisterReceive_pass()	Zähler für nicht modifizierbare Bytes hochzählen (Empfang)
RegisterSend_in()	Zähler für zu modifizierende Bytes hochzählen (Senden)
RegisterReceive_in()	Zähler für zu modifizierende Bytes hochzählen (Empfang)
RegisterSend_out()	Zähler für modifizierte Bytes hochzählen (Senden)
RegisterReceive_out()	Zähler für modifizierte Bytes hochzählen (Empfang)
ALANSuspend()	Netzwerkzugriffe vorübergehend aussetzen
ALANResume()	Netzwerkzugriffe wieder zulassen
ResetStat()	Alle Zähler zurücksetzen
SetStat()	Die Zähler auf bestimmte Werte setzen
SaveStat()	Alle Zähler gleichzeitig und damit konsistent auslesen
ALANEnumStruct()	Laufzeitstruktur auslesen. Der Rückgabewert dieser Funktion ist die Nummer des nächsten Eintrags in der Struktur. Ist kein weiterer Eintrag mehr vorhanden ist der Rückgabewert 0. Damit lässt sich leicht die Laufzeitstruktur zyclisch auslesen.

Funktionen für Execution Enviroments:

SubExecute()	Aufruf eines Execution Enviroment durch ein anderes Execution Enviroment (siehe Kapitel 4.4)
nbothbo16()	Wandeln von mehreren kurzen Ganzzahlen von Netzwerk Bytereihenfolge (Big Endian) in Little Endian
hbotnbo16()	Wandelt mehrere kurze Ganzzahlen in Big Edian um.
SaveAA()	Speichert eine Active Application für die spätere Verwendung zwischen.
RestoreAA()	Ladet eine Active Application in ein Execution Enviroment zwecks Ausführung zur Modifikation aktueller Eingabedaten
mkALANStr()	Wandelt eine ganze Zahl in eine Zeichenkette die die ganze Zahl in Hexadezimalschreibweise enthält.

Funktion zur Einrichtung von wichtigen Strukturen in der Windows Registery:

SetupALAN()	Erzeugt die durch diese Systemarchitektur verwendendete Struktur in der Registery
SetupALANAPH()	Trägt einen neuen ALAN Protokoll Helper in die Registery ein. Damit wird ein neues Transportprotokoll der Systemarchitektur bekannt gegeben.
SetupALAN_EE()	Trägt ein neues Execution Enviroment ein, welches durch das Ergebnis der Analyse der Adressdaten durch einen ALAN Protokoll Helper bestimmt werden soll.
SetupALAN_Chain()	Trägt ein neues Execution Enviroment ein, welches durch ein anderes Execution Enviroment aufgerufen werden soll.

4.6.4 Application Layer Active Network Protokoll Helper API

Das ALAN Protokoll Helper API besteht ausschließlich aus der Funktion Subprotokoll() welche von einem ALAN Protokoll Helper zur Verfügung gestellt wird. Diese Funktion erwartet als Eingabe eine SOCKADDR-Struktur und liefert daraus die Nummer des Protokolls welches auf Anwendungsebene benutzt werden soll zurück.
Bei TCP/IP ist dies sowohl bei TCP/IPv4 als auch bei TCP/IPv6 die Portnummer.

(Portnummern diese Tabelle 4.2). Dabei ist zu beachten das der Protokoll Helper für TCP/IPv4 nicht für TCP/IPv6 benutzt werden kann, sondern ein anderer ist. Siehe Tabelle 4.1, SOCKADDR-Strukturen von TCP/IPv4 und TCP/IPv6 siehe Abbildung 4.2 und 4.3).

4.6.5 Application Layer Active Network Execution Enviroment API

Ein Execution Enviroment bietet über das Application Layer Active Network Execution Enviroment API nur die Funktion Execute() an.

Dieser Funktion wird als Argumente einen Zeiger auf den zusammengefügte Datenbuffer, seine Länge, die gewünschte neue Länge, die ID der dazu gehörigen Verbindung, sowie ob es sich um einen zu versendenen Datenbuffer handelt oder um einen gerade empfangenen Datenbuffer. Bei Datenbuffern die gerade empfangen wurden ist die gewünschte neue Länge immer gleich der aktuellen Länge des Datenbuffers. Das Argument neue Länge ist hier nur um des einheitlichen Aufrufs wegen vorhanden. Ein Verkleinern eines Empfangsbuffers ist eher nicht notwendig. Wohl aber auch möglich, wenn die gewünschte Größe doch kleiner als die aktuelle Größe gesetzt würde. Die Monitor DLL führt dies aber nicht durch. Als Rückgabewert liefert ein Execution Enviroment immer die neue Länge des Datenbuffers zurück.

Der veränderte Datenbuffer wird stehts an der alten Stelle belassen.

4.7 Bedienung der Application Layer Active Network Systemarchitektur

Die Komponenten der Application Layer Active Network Systemarchitektur sind so aufgebaut, das sie normalerweise nach einmaliger Konfiguration alles alleine und automatisch machen können.

Konfiguriert werden müssen alle Zuordnungen zwischen Protokollen und Protokollhelpern , sowie zwischen Protokollen und Execution Enviroments. Außerdem muß die Monitor Dll, die maximal mögliche Sende-Datenübertragungsrate kennen, um Fehlmessungen zu erkennen, sowie einen Defaultwert als Ersatzwert, für die Zeit wo noch kein Meßwert der realen Übertragungsrate existiert.

Außerdem stellt die ALAN Systemarchitektur auch einige statistische Daten, über die bearbeitetemn Daten, bereit.

Diese Statistikdaten auszuwerten und die Konfiguationswerte einzustellen sind die Aufgaben des Management-Systems. Um diese Einstellungen aufzunehmen besitzt die Monitor DLL einen speziellen Satz von Einsprungpunkten und exportierten Variablen (siehe auch Kapitel 4.6.3).

maxnicrate ist dabei auf die Senderate in Byte pro Sekunde zu setzen.

Beispiel: T-DSL 1000 (Senderate 128 KBit/sec was 16 KByte/sec entspricht)
Die Variable maxnicrate ist also auf den Wert 16000 zu setzen.
Dies ist allerdings auch gleichzeitig der Defaultwert, den diese Variable auch ohne gezielte Einstellen hat.

Defaultsockrate ist die angenommene Senderate, bevor eine echte Messung der tatsächlichen Datenrate existiert. Der Wert ist ebenfalls in Byte pro Sekunde anzugeben. Standardmäßig steht er ebenfalls auf 16000.

Einmalig beim Installieren der hier beschriebenen Systemkomponenten ist die Struktur der Windows Registery Zweige für Active Network über den Aufruf von SetupALAN einzurichten. Diese Funktion kennt keine Parameter und braucht nur einmal aufgerufen werden.
Das Registrieren eines neuen Protokollhelpers um ein neues Transportprotokoll zu unterstützen, wird mit der Funktion SetupALANAPH durchgeführt. Dieser Funktion sind als Zahl zu übergeben: Die Protokollfamilie, der Sockettype sowie das zu benutzende Transportprotokoll. Als weiterer Parameter kennt diese Funktion nur einen Dateinamen als 13 Zeichen umfassendes Feld von Zeichen. Das letzte Zeichen ist dabei das Zeilenende das als 0-Byte zu übergeben ist.
Execution Enviroments werden je nach dem ob sie als direkt durch die Wahl eines Anwendungsprotokolls angesprochen werden sollen oder über ein anderes Executions Enviroment mit der Funktion SetupALAN_EE oder mit der Funktion SetupALAN_Chain registiert. Die Funktion SetupALAN_EE registriert dabei für die Auswahl über ein Anwendungsprotokoll und erwartet als Parameter die Protokollfamilie, den Sockettype, das zu benutzende Transportprotokoll, sowie das zu benutzende Anwendungsprotokoll, gefolgt vom Dateinamen des Execution Enviroments. SetupALAN_Chain erwartet dagegen einen internen Namen für das Execution Enviroment das ein anderes aufrufen soll als Feld von Zeichen, ein Zeichen das als Nummer des aufzurufenden Execution Enviroments aufzufassen ist, sowie dem Dateinamen des aufzurufenden Execution Enviroments.

4.8 Grenzen der Application Layer Active Network Systemarchitektur

Nicht in jedem Fall ist die hier vorgestellte Systemarchitektur in der Lage eine Modifikation der zu versendenen Daten wie gewünscht durchzuführen. In diesem Unterkapitel werden alle Fälle aufgezählt und erläutert bei der dies der Fall ist.

Unbekannte Transportprotokolle

Da die Monitor DLL über die Protokoll Helper die SOCKADDR-Struktur des verwendeten Transportprotokoll kennen muß, den Verbindungen das jeweils passende Execution Enviroment zuzuordnen, ist eine Modifikation nicht möglich, wenn das Transportprotokoll unbekannt ist.

Unbekanntes Protokoll auf Anwendungsebene

Auch, wenn zwar ein passender Protokoll Helper vorhanden ist, aber kein für das Protokoll das auf Anwendungsebene eingesetzt wird geeignetes Execution Enviroment vorhanden ist gelingt die Modifikation nicht. Das Execution Enviroment hat die Aufgabe die Modifikationen durchzuführen, welche durch die Monitor DLL und weitere Komponenten dieser Systemarchitektur vorbereitet wurden, daher ist ohne EE keine Modifikation möglich.

Kein passendes Unter-Execution Enviroment

Benutzt ein Execution Enviroment, die Möglichkeit je nach Art der Daten ein weiteres Execution Enviroment zu starten, so muß für jede Art von Daten die durch das Execution Enviroment laufen, auch ein Execution Enviroment existierten, welches die Daten wirklich bearbeiten kann. Ist dies nicht der Fall, so gibt es Daten die durch kein Execution Enviroment bearbeitet werden können, und damit unbearbeitet gesendet bzw. empfangen werden.

Verbindungslose Netzwerkzugriffe

Der ALAN_LSP misst die Zeitdauer der Datenübertragung, um daraus zu berechnen, wie viel Datendurchsatz erreicht werden kann. Dies passiert darauf das verbindungsbezogene Netzwerkzugriffe erst als gelungen gelten, wenn die Übertragung wirklich durchgeführt ist. Verbindungslose Netzwerkzugriffe gestalten sich derart, das einfach drauf los gesendet wird. Dabei existiert keinerlei Sicherheit ob ein Datenpaket überhaupt ankommt, geschweige den wann. Daher ist die Datenübertragung bei verbindungslosen

Netzwerkzugriffen beendet, sobald der Datenbuffer an die unter dem ALAN_LSP befindliche Softwareschicht übergeben wurde. Die gemessene Zeitdauer ist daher praktisch immer sehr klein, eventuell sogar fast immer kleiner als 1 ms, was der minimalen Zeitauflösung der bei Windows vorhandenen Zeitzähler entspricht.
Datenbuffer von verbindungslosen Netzwerkzugriffen können daher praktisch nicht sinnvoll modifiziert werden.

Broadcast, Multicast und Anycast

Broadcasts (siehe [NE02]) sind Netzwerkzugriffe die vorallem innerhalb eines lokalen Netzwerkes alle erreichbaren Netzwerkstationen ansprechen. Dies geschied im Wessendlichen durch Drauflossenden der Datenpakete ins Netz. Damit ist wie bereits bei verbindungslosen Netzwerkzugriffen keine effektive Modifikation der Daten möglich. Zumal Broadcasts üblicherweise auch verbindungslos sind.
Multicasts (siehe [NE03] sind Netzwerkzugriffe die eine spezielle Gruppe von Netzwerkstationen anspricht, die sich dazu anmelden. Der eigentliche Sendevorgang geschied allerdings ähnlich wie bei Broadcasts durch drauflossenden der Datenpakete. Multicast ist ein eine modifizierte Form von Broadcast.
Anycast (siehe [NE01]) ist eine bei TCP/IPv6 neu eingeführte Form von Netzwerkzugriffen, wo ein Netzwerkrechner aus einer Gruppe von gleichartigen Rechnern benutzt wird, ohne das der anforderne Rechner dazu selbst ein spezielles Mitglied dieser Gruppe auswählen muß. Die Auswahl geschied automatisch durch Komponenten des Netzwerkes. Dies beeinträchtigt die Funktion dieser Systemarchitektur nicht, soweit nicht bereits ein anderer Grund vorliegt, der eine Modifikation von Daten erschwert oder ausschließt.

Unicast

Unicast (siehe [NE04]) ist die normale Art eines Netzwerkzugriffs, wo eine Netzwerkstation eine andere anspricht. Dies ist daher auch die Art von Netzwerkzugriffen wofür diese Systemarchitektur entworfen wurde. Sofern keine oben genannten Gründe vorliegen, spricht daher nichts gegen die Modifikation der zu sendenen Daten nach Active Network Gesichtspunkten.

4.9 Neuere und zukünftige Entwicklungen

Dieses Unterkapitel stellt kurz frische und zukünftige Windows Versionen vor und erläutert kurz, welche Änderungen für das hier vorgestellte Application Layer Active Network System relevant sind, die diese Windows Versionen mit sich bringen.

Windows XP – Servicepack 2

Am 9.08.2004 hat Microsoft das Servicepack 2 zu Windows XP als Netzwerkinstallationspaket für IT-Spezialisten und Entwickler veröffentlicht. Mit diesem Servicepack haben sehr viele Änderungen in Windows Einzug gehalten. Insbesondere auch im Bereich des Netzwerkstapels. Diese Arbeit kann sich nicht im einzelnen mit jeder dieser Änderungen beschäftigen, da zum Zeitpunkt der Veröffentlichung dieses Servicepacks diese Arbeit bereits sehr weit fortgeschritten war.

Die bei Microsoft zum Servicepack 2 dokumentierten Änderungen umfassen allerdings nicht das Windows Sockets Service Provider Interface (siehe [MI04]). Da dieses die einzige für die Funktion der in dieser Arbeit beschriebenen Systemarchitektur von Windows benötigte Schnittstelle ist, ist davon auszugehen das die Funktion der Application Layer Active Network Systemarchitektur durch das Servicepack 2 von Windows XP nicht beeinträchtigt wird.

Windows 2003 Server

Microsoft hat auch bei Windows 2003 Server keine Änderung des Windows Sockets Service Provider Interface durchgeführt. Damit ist diese Systemarchitektur auch bei Windows 2003 Server einsetzbar.

Windows Longhorn

Für Windows Longhorn, die nächste große Windows Version sind sehr einschreitende Änderungen angekündigt [CT03]. Allerdings sind bisher nur wenige Hinweise darauf zu finden, welche Auswirkungen diese Änderungen tatsächlich haben. Bekannt ist allerdings das die Ausführungsschicht von Windows sich in mindestens zwei Teile teilt und diese beiden Teile mehr oder weniger unanhängig voneinander arbeiten werden [CT03].

Dies deutet zumindest auf größe Änderungen hin, welche auch den Netzwerkstabel betreffen können. Insbesondere ist damit eine Änderung an den für die in dieser Arbeit vorgestellte Systemarchitektur wichtigen Systemschnittstellen nicht undenkbar.

Windows 64-Bit Edition

Die in dieser Arbeit vorgestellte Systemarchitektur wurde auf einem aktuellen Windows XP entwickelt. Also einem 32-Bit-Windows. Dies zeigt sich insbesondere durch viele Speicherlimits, die darauf zurückzuführen sind, das ein 32-Bit-System 4 GByte Gesamtspeicher direkt adressieren kann. Davon sind allerdings nur 2 GB für eine Anwendung direkt verfügbar.

64-Bit-Systeme wie der Intel Itanium und die AMD64-CPUs können dagegen sehr viel mehr Speicher bereitstellen. Davon würde unter Umständen auch ein Application Layer Active Network 64 System profitieren können. Dies ginge allerdings nur nach Anpassung der Limits im Quellcode der hier beschriebenen Module der Systemarchitektur.

Aufgrund der Tatsache das, das Windows Sockets Service Provider Interface beim Schritt zu 64-Bit nicht verändert wurde, wäre eine einfache 64-Bit Version auch durch einfaches neucompilieren denkbar. Eine genauere Betrachtung habe ich allerdings nicht durchgeführt.

4.10 Andere Windows Versionen

Bisher hat diese Arbeit nur aktuelle und zukünftige Windows Versionen betrachtet. Dieses Unterkapitel wird auch noch kurz frühere und weniger bekannte Windows-Versionen betrachten. (Winsock Versionen außer für NT3.x siehe auch [JO02] Tabelle 1-1).

Windows 1.0 bis Windows 3.11

Windows war bis einschließlich Version 3.11 von Haus aus nicht netzwerkfähig. Die Benutzung von Netzwerkanwendungen setze daher propritäre Treiber von anderen Herstellern voraus. Daher ist eine Veränderung der Treiberarchitektur zum Realisierung von Active Network Funktionalistät eher nicht realisierbar. Diese Windows Versionen sind allerdings auch schon so alt, das sie praktisch nicht mehr relevant sind.

Windows 95

Mit Windows 95 zog erstmals ein Netzwerkstabel in Windows ein. Dies war Windows Sockets 1.1. Dies war daher noch der Windows Netzwerkstabel der in dieser Arbeit betrachtet und modifiziert wurde.

Später war allerdings ein Update der Winsock DLL erhältlich. Dieses realisiert erstmal die Winsock 2.0 Schnittstelle und dabei auch das Service Provider Interface, welches in dieser Arbeit benutzt wird um sich in den Netzwerkstabel einzuklinken, und Active Network Funktionalität zu implementieren. Es ist daher anzunehmen das nach dem Update der Winsock auch die Module dieser Arbeit einsetzbar sind.

Windows 98 und Windows ME

Windows 98 und Windows ME wurde bereits mit der neuen Winsock DLL ausgeliefert und haben daher bereits den modernen Netzwerkstabel, der Vorausetzung für die in dieser Arbeit beschriebene Active Network Systemarchitektur ist.

Windows NT 3.x

Alle Windows NT Versionen der Versionslinie 3 enthalten nur einen Netzwerkstabel nach der Winsock Spezifikation 1.1 (siehe [TA04]) und sind daher nicht für die Realisierung dieser Systemarchitektur geeignet .

Windows NT 4.0

Windows NT in der Version 4.0 enthält bereits den für diese Arbeit vorausgesetzen Netzwerkstabel nach Winsock Spezifikation 2.2 und ist daher auch geeignet die Active Network Datenmodifikation durchzuführen.

Windows CE

Das Windows für PDAs enthält leider nur einen Netzwerkstack nach Winsock 1.1 Spezifikation und ist daher nicht geeignet die hier vorgestellten Softwaremodule zu nutzen. PDAs können also nur indirekt von dieser Systemarchitektur profitieren, in dem sie einen Server ansteuern welcher mit einem der anderen modernen Windows-Versionen arbeitet.

Windows 2000/XP/2003

Diese Systemarchitektur ist für Windows 2000 und Windows XP entworfen worden. Daher ist es selbstverständlich für die Benutzung mit Windows 2000 und Windows XP geeignet.
Windows 2003 als neuere Windows-Version war bereits im letzten Unterkapitel Thema und ist auch geeignet.

4.11 Portierbarkeit der hier beschriebenen Ideen auf andere Platformen

Zuletzt noch einige Gedanken zur Umsetzung der Ideen, hinter der hier beschriebenen Systemarchitektur, auf andere Platformen als aktuelle Windows Versionen.

Winsock 1.1

Die Winsock 1.1 Spezifikation enthält keine dokumentierte Schnittstelle zur Erweiterung der Fähigkeiten der Winsock DLL. Insbesondere existiert das Konzept eines Layered Service Provider nicht (siehe [TA04]). Dies ist erst in der Spezifikation Winsock 2.2 enthalten. Es wären allenfalls Lösungen denkbar die die Winsock DLL durch eine eigne Version ersetzen.

Dies gilt nicht für Winsock 1.1 Anwendungen auf einem Windows mit Winsock 2.2 DLL. In diesem Falle wird das Winsock 1.1 API nur zu Kompatibiltätszwecken genutzt. Die Winsock 1.1 DLL auf einem System mit Winsock 2.2 übersetzt Winsock 1.1 Aufrufe nur in Winsock 2.2 API-Aufrufe. Es wird also auf Systemen mit Winsock 2.2 DLL auch bei Verwendung des Winsock 1.1 API ein Layered Service Provider benutzt. Siehe auch Abbildung 4.1. Der oben beschriebene Sachverhalt gilt nur für Windows Versionen die ausschließlich die Winsock 1.1 DLL installiert haben.

MSDOS und Windows bis Version 3.11

Für MSDOS und Windows bis einschließlich Version 3.11 existiert kein standardisierter Netzwerkstapel. Dies ist allerdings für das Finden eines wirklich zuverlässigen Weges nützlich, um in diesen Netzwerkstabel die Active Network Funktionalistät einzubauen.

Die vorhandenen propritären Netzwerkstabel für diese Betriebsysteme sind primär dafür gedacht Netzwerkclients auch mit diesen alten Betriebsystemen zu betreiben.

Betriebsysteme, die nicht von Microsoft stammen

Die Ideen hinter dieser Arbeit lassen sich am leichtesten auf Betriebsysteme portieren, die bereits eine standardisierte Schnittstelle zur Erweiterung des Netzwerkstabels haben.

Betriebsysteme mit einem monolytischen Kernel (z.B. Linux), der auch die Netzwerkfunktionalität enthält, würden wohl einen Kernelpatch benötigen, um Schnittstellen zur Realisierung von Active Network Funktionalistät bereitstellen zu können. Da dies nicht Gegenstand dieser Arbeit ist, werde ich daher nicht weiter eingehen.

5 Zusammenfassung

Active Networks erlauben im Gegensatz zu herkömmlichen passiven Netzwerkumgebungen die Anpassung der Daten an die zur Verfügung stehende Bandbreite. Dies bietet insbesondere bei der Übertragung von Multimediadaten Vorteile. Dies gilt besonders bei Verwendung von Netzwerkclients mit geringer Bandbreite z.B. Multimedia-Mobiltelefone.

Gewöhnliche Active Networks setzen dabei auf mehr oder weniger intelligente Router, welche die Active Network Datenpakete anpassen. Da hierbei die Router unter Umständen fremden und damit nicht vertrauenswürdigen Programmcode ausführen müssen gibt es erhebliche Vorbehalte gegenüber dieser Idee. Als Alternative zu intelligenten Routern gibt es die Idee der Application Layer Active Networks. Diese beinhalten keine intelligenten Router sondern realisieren ihre Active Network Funktionalität durch Anpassung der zu sendenen Datenpakete auf dem Netzwerkserver.

In dieser vorliegenden Arbeit wurde ein solches System für Windows 2000 und Windows XP entwickelt. Dieses setzt auf das für diesen Zweck sehr gut geeignete Konzept der Layered Service Provider auf. Dies ist ein wichtiger Teil der Windows Sockets 2.0 Architektur. Diese Architektur erlaubt fast beliebige Erweiterungen der Funktionalität der Winsock DLL mit Hilfe der genannten Layered Service Provider.

Als Vorteil dieser Treiberart gegenüber weiteren zu Erweiterungszwecken benutzbaren Treibern ist zu sehen, das Layered Service Provider keine Neuerstellung von Netzwerkpaketheadern erforderlich machen. Dies erlaubte die Realisierung eines Systems das fast unabhängig von verwendeten Netzwerkprotokollen die gewünschte Funktionalität bereitstellt. Einzig die später in den Netzwerkpaketheadern unterzubringenden Informationen müssen gelesen werden um die eigentlichen Daten ädaquat anpassen zu können. Um dies umzusetzen ist das in dieser Arbeit entwickelte und vorgestellte Active Network System modular aufgebaut.

Alle protokoll- oder datenabhängigen Teile können durch andere Teile ausgetauscht oder mit weiteren Modulen ergänzt werden. Dies erlaubte eine fast unbegrenzte Erweiterbarkeit auf neue oder schon vorhandene, aber in dieser Arbeit noch nicht realisierte Protokolle.

6 Anhänge

Hier noch einige zusätzliche Materialien die für den Gesamtzusammenhang der Active Networks nützlich sind.

6.1 Active Network Encapsulation Protocol (ANEP) TypeID

Das Active Network Encapsulation Protocol (ANEP) verwendet zur Unterscheidung der verschiedenen Active Network Verfahren sogenannte TypeIDs. Diese sind bei der Active Networks Assigned Number Authority (ANANA, http://www.isi.edu/~braden/anana/) registriert. Unter [AN04] findet sich die folgene Liste mit bereits registrierten TypeIDs:

```
Assigned TypeIds:
        #       Org             Purpose

        0       Reserved for future use
        1-10    (Reserved)

        11      UKans           User packets
        12      UKans           Management packets
        13      UKans           Test TypeId 1
        14      UKans           Test TypeId 2

        15      ISI             Active signaling tests
        16      ISI             Active signaling tests
        17      ISI             Active signaling tests

        18      MIT             ANTS

        19      UPenn           PLAN
        20      UPenn           Reserved-1
        21      UPenn           Reserved-2
        22      UPenn           Reserved-3

        23      Reserved

        24      BBN             Spanner
        25      BBN             CENCOMM
        26      BBN             CENCOMM
        27      BBN             CENCOMM
        28      BBN             CENCOMM

        29-30   Reserved

        31      Columbia        NetScript program deployment
                                packets
        32      Columbia        NetScript packet traffic
        33      Columbia        NetScript management traffic
        34      Columbia        Reserved-1
        35      Columbia        Reserved-2

        36      U of Washington ANTS
        37      U of Washington ANTS
        38      U of Washington ANTS
        39      U of Washington ANTS
        40      U of Washington ANTS
```

```
41      TASC/UMass      Multicast data packets
42      TASC/UMass      Multicast control packets
43      TASC/UMass      Multicast session packets
44      TASC/UMass      Reserved

45      Reserved
46      Reserved

47      MIT DS

48-50   Reserved

51      SRI             ANCORS daemon
52      SRI             Reserved-1
53      SRI             Reserved-2
54      SRI             Reserved-3
55      SRI             Reserved-4

56-60   Reserved

61-64   Lucent

65-70   Reserved

71      LORIA           ANAIS - Management1
72      LORIA           ANAIS - Management2
73      LORI            ANAIS - Data
74      LORIA           ANAIS - Multicast
75      LORIA           ANAIS - reserved

75      Reserved

77      ICSI Berkeley   MO messengers

78-80   Reserved

81      ETH Zurich      Anet
82      ETH Zurich      Anet simulation

83-85   Reserved

86      Wash U, StL     DAN

87-90   Reserved

91      3Com            3Com Experimental 1
92      3Com            3Com Experimental 1
93      3Com            3Com Experimental 1
94      3Com            3Com Experimental 1

94-100  Reserved

101     UPenn           SANE/OS
102     UPenn           SANE/OS
103     UPenn           SANE/OS
104     UPenn           SANE/OS

105-110 Reserved

111     ICU Korea       ANGLE testbed
112     ICU Korea       ANGLE testbed
113     ICU Korea       ANGLE testbedd

114-115  Reserved
116-120  CMU            CS.CMU (ref. Christopher Lin)
```

```
121-125  Utah           antsr

126-130  Reserved

131-139  ISI            ASP EE

140      ISI/TASC       Team 1 Demo 2000

141-145  Technion       DIAN system
```

6.2 Die Protokolle der TCP/IP Protokollfamilie

Abbildung 6.1 zeigt die verschiedenen Schichten der TCP/IP-Protokollfamilie.
Schicht 1 und 2 sind dabei die hardwäreabhängigen Schichten die eigentlich nicht zu TCP/IP gehören, da TCP/IP hier auf diese aufsetzt.
Schicht 3 beinhaltet in erster Linie das Internetprotokoll IP, auf welches die Schicht 4 Protokolle (TCP und UDP) aufsetzen. Im Gegensatz zum OSI-Reverenzmodell beinhaltet die TCP/IP-Protokollfamilie keine Schicht 5 und 6. Die Anwendungsprotokolle auf Schicht 7 setzen direkt auf die Schicht 4 Protokolle auf.

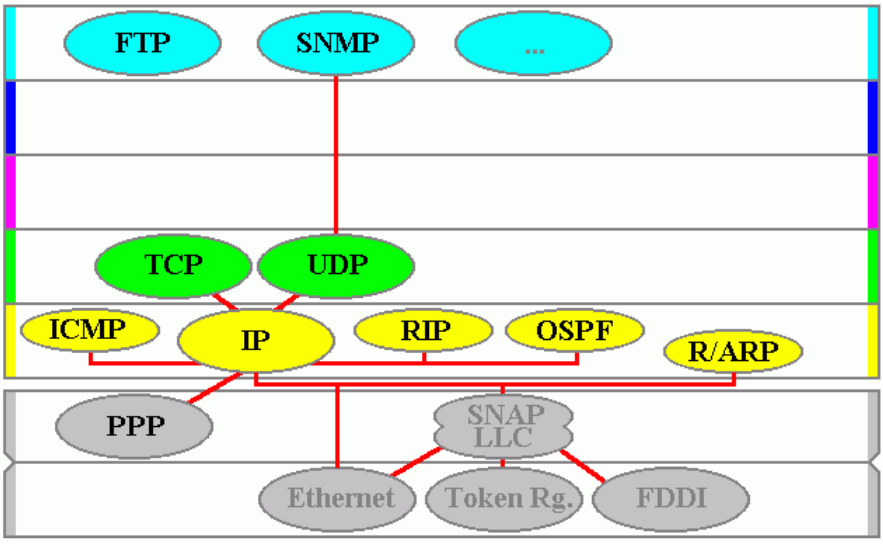

Abbildung 6.1 Die TCP/IP Protokollfamilie
(aus [SY04])

Literaturverzeichnis

[AL98]: D.Scott Alexander, William A. Arbaugh, Michael W. Hicks u.a,
The Switchware Active Network Architekture, 1998,
http://www.cis.upenn.edu/~switchware/papers/switchware.ps

[AN03]: ANTS, ANTS 2.0, 2004,
http://www.cs.washington.edu/research/networking/ants/manual.html

[AN04]: ANANA, Assigned TypeIds, 2004, http://www.isi.edu/~braden/anana/info/typeid.txt

[BU00]: A.Kulkarni, S.Bush, G.Minden, Implementation and Experiencs on the
Magician Active Network Toolkit, 2000, http://www.crd.ge.com/cooltechnologies/pdf/2000crd127.pdf

[BU01]: Stephen F. Bush, Active Network and Active Network Management, 2001, ISBN: 0306465604

[CT03]: Siering,Schulz,Withopf,Himmelein,Longhorns tragende Teile,Technische
Unterschiede zu den Vorläufern, c´t 24/2003, S.118ff

[GE98]: Prof. Dr.-Ing. W.Geisselhardt, Skript zur Vorlesung Datenverarbeitung I, 1998/99

[IA04]: IANA, PORT NUMBERS, 2004, http://www.iana.org/assignments/port-numbers

[JO02]: Jones,Ohlund, Network Programming for Microsoft Windows, 2002, ISBN:0735615799

[MA01]: Kulkarn, Magician - A Toolkit for Building Active Networks, 2004, http://www.geocities.com/akulkarn/

[MI02]: Microsoft, Microsoft Windows XP SP1 DDK (us), 2002

[MI03]: Microsoft, Platform SDK Documentation, 2003

[MI04]: Microsoft, Diese Funktionsänderungen bringt Microsoft Windows XP Service Pack 2,2004,
http://www.microsoft.com/germany/ms/technetdatenbank/showArticle.asp?siteid=600337

[ND04]: NDIS, Packet Filtering, 2004, http://www.ndis.com/papers/winpktfilter.htm

[NE01]: LANLINE, Anycast, 2004, http://www.lanline.de/html/lanline/lexikon/lex/anycast.htm

[NE02]: LANLINE, Broadcast, 2004, http://www.lanline.de/html/lanline/lexikon/lex/broadcast.htm

[NE03]: LANLINE, Multicast, 2004, http://www.lanline.de/html/lanline/lexikon/lex/multicast.htm

[NE04]: LANLINE, Unicast, 2004, http://www.lanline.de/html/lanline/lexikon/lex/unicast.htm

[SI01]: Sushil da Silva u.a., The NetScript Active Network System, IEEE,2001

[SY04]: SYNAPSE, Inhalt: TCP/IP-Protokolle, 2004,
http://www.synapse.de/ban/HTML/P_TCP_IP/Ger/P_tcp_ip.html

[TA04]: Tangentsoft, Winsock Programmer's FAQ, 2004, http://tangentsoft.net/wskfaq/general.html

[UP04]: D.Scott Alexander u.a., Active Network Encapsulation Protocol, 1997,
http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt

[WE99]: David J. Wetherall, Service Introduction in an Active Network,Thesis, 1999

Alle Internetdokumente wurden am 7.09.2004 letztmalig überprüft, ob sie weiterhin an den genannten Stellen existieren.