Fachhochschule Bonn-Rhein-Sieg

University of Applied Sciences

Fachbereich Angewandte Informatik

Diplomarbeit

Entwurf und Implementierung eines broadcast-basierten Namensdienstes für IPv6

von

Sebastian Lederer

Erstgutachter: Prof. Dr. Martin Leischner

Zweitgutachter: Prof. Dr. Karl W. Neunast

21. Februar 2001

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bonn, den 21. Februar 2001

....
(Sebastian Lederer)

Zusammenfassung

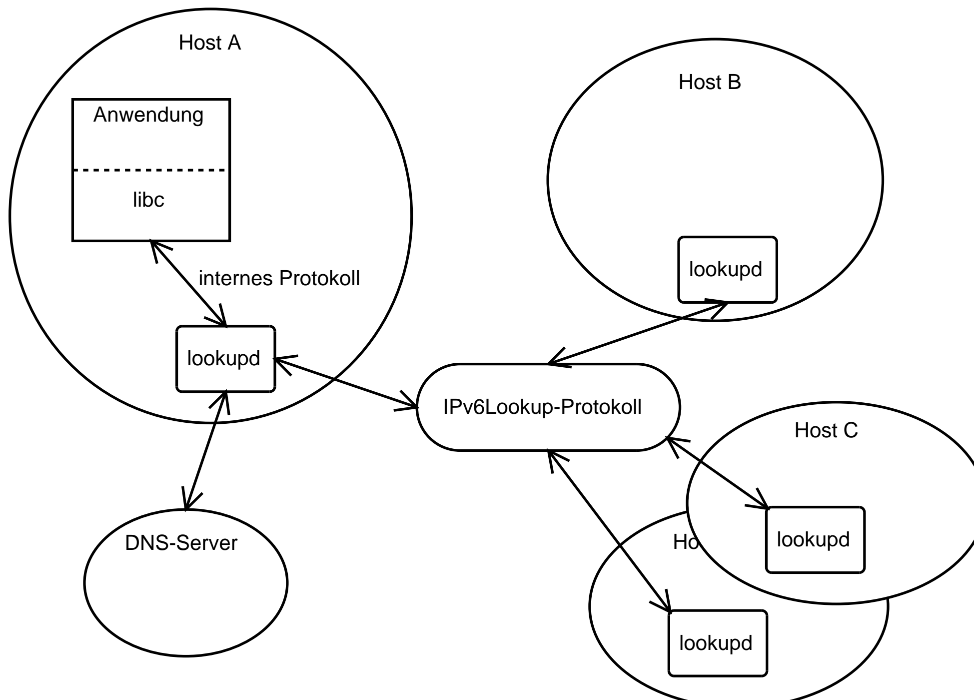
In der vorliegenden Arbeit wird ein Namensdienst entworfen und implementiert, der die Aufgabe hat, die inhärente Fähigkeit von IPv6 zu "Plug-And-Play-Networking" praktisch nutzbar zu machen.

Dazu müssen Hostnamen in IPv6-Adressen aufgelöst werden können, ohne dass eine Konfiguration durch den Benutzer erforderlich ist. Die Namensauflösung bei dem hier entworfenen Protokoll erfolgt durch ein Broadcast-Verfahren, wodurch die Notwendigkeit eines zentralen Servers und damit zentraler Administration des Namensdienstes entfällt. Im Zusammenhang mit diesem neuen Namensdienst wird für FreeBSD 4.0 eine Namensdienstarchitektur implementiert, bei der die Namensauflösung in einen Dæmon ausgelagert wird, der als Agent die Namensauflösung für die Anwendungen übernimmt.

IPv6 bietet die bisher wenig beachtete Möglichkeit, ohne jeglichen Konfigurationsaufwand mit benachbarten Netzwerkknoten zu kommunizieren. Dazu wird jedem Netzwerk-Interface automatisch eine IPv6-Adresse zugeteilt, die nur im lokalen Netzsegment gültig ist (*link local address*). Diese Adresse wird aus der MAC-Adresse des Netzwerk-Interfaces abgeleitet und ist damit immer eindeutig definiert. Kombiniert mit dem hier vorgestellten Broadcast-basierten Namensdienst ist damit ein konfigurationsloser Netzbetrieb mit IPv6 möglich. Die einzige Konfigurationsinformation, die jeder Netzwerkknoten erhalten muss, ist dessen Name. Der administrative Aufwand für das Aufsetzen von DNS-Servern und DHCP-Servern entfällt. Das entworfene Namensauflösungs-Protokoll, *IPv6Lookup* genannt, ist bewusst einfach gehalten, um die Implementierung einfach, effizient und resourcenschonend zu gestalten. Um dieses Protokoll für Anwendungen transparent nutzbar zu machen, musste in die bestehende Namensdienst-Architektur des Betriebssystems eingegriffen werden.

Die klassische Namensdienst-Architektur unter Unix sieht vor, verschiedene Namensdienste transparent zu nutzen. Üblicherweise sind als mögliche Informationsquellen mindestens lokale Dateien, DNS und NIS vorhanden. Die tatsächliche Namensauflösung wird dabei durch die Anwendung selbst durchgeführt. Der entsprechende Programmcode ist zwar in einer Bibliothek gekapselt, trotzdem findet die Ausführung in der Anwendung statt, die diese Bibliothek einbindet. Der hier vorgestellte Ansatz erweitert die Namensdienst-Architektur dahingehend, dass der Vorgang der Namensauflösung in einen speziell dafür vorgesehenen Dæmon ausgelagert wird. Anwendungen kommunizieren mit dem Dæmon über ein internes Protokoll, um ihn mit Namensdienst-Anfragen zu beauftragen. Die unterschiedlichen Namensdienst-Systeme sind

im Dæmon implementiert, der entsprechende Programmcode ist aus den Anwendungen ausgelagert worden. Dadurch wird eine höhere Flexibilität und eine reduzierte Komplexität der Namensdienst-Funktionen der C-Bibliothek erreicht. Außerdem ist es nun möglich, die Ergebnisse von Anfragen in einem systemweiten Cache zwischenzuspeichern, was weitere Anfragen beschleunigt und den Netzwerkverkehr für Namensauflösungen verringert. Bei Namensdiensten, die eine aufwendige Kommunikation mit einem Server benötigen, wird diese zentralisiert und muss nicht für jede Anwendung separat abgehandelt werden. Wird z.B. LDAP als Namensdienst verwendet, muss nicht jede Anwendung eine eigene Verbindung zum LDAP-Server aufbauen. Eine derartige Architektur findet sich in ähnlicher Form in NeXTSTEP - heute MacOS X- als *lookupd* (dieser Name wurde übernommen), oder als *nscd* (*Name Service Cache Dæmon*) in Solaris, Linux und anderen Unix-Systemen. Im Gegensatz zum in Linux vorhandenen (und damit als Quelltext verfügbaren) *nscd*-System, das ein starres, binär codiertes Protokoll verwendet, wird in der vorliegenden Arbeit ein System entworfen, das unspezifische, als Strings codierte Attribut-Mengen überträgt und damit flexibel und beliebig erweiterbar ist.



Die in dieser Arbeit vorgestellten Konzepte konnten aufgrund der Zeitbegrenzung nicht in vollem Umfang implementiert werden. Die derzeit implementierte Funktionalität umfasst die Auflösung von Hostnamen in IP-Adressen über IPv6Lookup (nur IPv6-Adressen) und DNS (IPv4 und IPv6) sowie die Auflösung von Adressen in Hostnamen. Nicht implementiert, aber vorgesehen, sind weitere Informationsarten wie Benutzer- und Gruppen-Informationen, Protokoll- und Portnummern und andere Informationen, die üblicherweise über NIS und ähnliche Dienste abgefragt werden können. Der Speicherverbrauch des Dæmons ist minimal (ca. 400KB Textsegmentgröße, ca. 220KB residente Speicherseiten), durch eine Event-basierte Programmstruktur können beliebig viele parallele Anfragen mit relativ geringer CPU-Last abgehandelt werden. Die C-Bibliothek

(*libc*) wurde so erweitert, dass die für die Namensauflösung vorgesehenen Funktionen Anfragen an den Dæmon der neuen Namensdienst-Architektur stellen können. Dies erfolgt zusätzlich zu den bisher in libc implementierten Methoden zur Namensauflösung (Files, DNS, NIS). Die geänderten Funktionen sind *gethostbyname()*, *gethostbyaddr()*, *getaddrinfo()*, *getipnodebyname()* und *getipnodebyaddr()*.

Als weiterführendes Projekt bietet sich nun an, die neue Namensdienst-Architektur so weit zu vervollständigen, dass alle (oder die meisten) Informationsarten abgefragt werden können. So sollten unter anderem die Funktionen *getnetbyname()*, *getnetbyaddr()*, *getprotoent()*, *getservent()*, *ether_ntohost()*, *getgroupent()* und *getpwent()* in der Lage sein, Informationen über den Namensdienst-Dæmon abzufragen. Außerdem sollten weitere Informationsdienste wie z.B. NIS in den Namensdienst-Dæmon integriert und die bisher in der C-Bibliothek vorhandene Funktionalität vollständig in den Dæmon ausgelagert werden. Damit würde FreeBSD eine flexible, leistungsfähige Namensdienst-Architektur erhalten, die auch in der Lage ist, zukünftige Weiterentwicklungen von Namensdiensten und Informationssystemen problemlos zu integrieren.

Danksagungen

Besonderer Dank gilt:

Prof. Dr. Martin Leischner und Prof. Dr. Karl W. Neunast für die gute Zusammenarbeit, die konstruktive Kritik und die Unterstützung im Vorfeld und während der Diplomarbeit,

Bernd Kolbeck für die kritische Evaluation meines Diplomarbeitsthemas,

Jan Hermanns, Steffen Kaiser, Martina Kannen, Christoph Neerfeld für die vielen anregenden Diskussionen,

Adriana Arghir für ihre Unterstützung bei der Herstellung der gedruckten Fassung,

Harald Deuer und Sascha Sertel für einige Hinweise zu LaTeX.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Problemstellung 5

1.3 Struktur der Arbeit 6

2 Evaluation vorhandener Namensdienste **7**

2.1 Der Internet-Namensdienst – DNS 7

2.2 Konfigurationslose Namensdienste 8

2.3 Verzeichnisdienste als Namensdienste 10

2.4 Ergebnis 11

3 Anforderungen an einen neuen Namensdienst **13**

4 Entwurf des Namensauflösungs-Protokolls **15**

4.1 Zielsetzung 15

4.2 Konzepte und Definitionen 15

4.3 Paketaufbau und Codierung 17

4.4 Ablauf des Protokolls 22

4.5 Mögliche Erweiterungen 24

5 Planung der Integration ins Betriebssystem **27**

5.1 Integration von Namensdiensten unter Unix 27

5.2 Entwicklung der Namensdienst-Architekturen unter Unix 28

5.3 Momentane Situation in FreeBSD 4.0 31

6 Entwurf einer universellen Namensdienst-Architektur **33**
6.1 Die Architektur im Überblick . . . 33
6.2 Das Informationsmodell . . . 34
6.3 Das *lookupd*-Protokoll . . . 37

7 Implementierung **41**
7.1 Ziele der Implementierung . . . 41
7.2 Design-Entscheidungen . . . 42
7.3 Einzelne Module . . . 47
7.3.1 Überblick . . . 47
7.3.2 Abstrakte Datentypen . . . 47
7.3.3 Request . . . 49
7.3.4 Result . . . 52
7.3.5 DB . . . 53
7.3.6 Cache . . . 54
7.3.7 Events . . . 55
7.3.8 Timeout . . . 56
7.3.9 IPv6Lookup . . . 57
7.3.10 DNSLookup . . . 61
7.3.11 lookupd . . . 63
7.4 Programmablauf . . . 64
7.5 Integration in die C-Bibliothek . . . 65

8 Tests und Leistungsmessungen **67**

9 Zusammenfassung und Ausblick **71**

Abbildungsverzeichnis

6.1 Die *lookupd*-Architektur 34

7.1 Parallelisierung durch Prozesse 44
7.2 Parallelisierung durch Threads 45
7.3 Parallelisierung durch Events 46
7.4 Module und deren Assoziationen 48
7.5 Programmablauf bei der Verarbeitung von Anfragen 64

Kapitel 1

Einleitung

1.1 Motivation

Die Idee für das Thema dieser Arbeit stammt aus einer näheren Untersuchung der besonderen Eigenschaften von IPv6, dem Internet-Protokoll der nächsten Generation, das in naher Zukunft das bisher verwendete IPv4 ablösen soll[1]. Eine detaillierte Beschreibung des IPv6-Protokolls findet sich unter anderem in [Dittler98] oder [HD99], die Reihe der relevanten Internet-Standards beginnt bei [RFC2460]. Auf die Einzelheiten soll hier generell nicht eingegangen werden.

Zunächst soll hier die Fragestellung beleuchtet werden, warum man überhaupt IPv6 einsetzen sollte, spricht doch der große Aufwand einer Umstellung sehr dagegen. Die IETF hat in dem in dem Papier "The Case for IPv6", siehe [KFP00], eine Reihe von Gründen dargelegt, die zur Migration nach IPv6 motivieren sollen. In dieser Einleitung sollen die dort vorgebrachten Argumente analysiert werden, wobei sich herausstellen wird, dass die meisten davon nicht wirklich relevant sind, wie z.B. der Verweis auf neue Eigenschaften in den Bereichen Sicherheit und Quality of Service. Dabei wird aber auch eine weitere, zunächst eher unwichtig scheinende Eigenschaft von IPv6 zur Adressenkonfiguration auffallen, die zu neuen Anwendungsmöglichkeiten führt, welche in dieser Form mit IPv4 nicht realisierbar sind. Aus diesen neuen Möglichkeiten leitet sich dann die Idee zu dieser Arbeit ab.

Zunächst aber eine vereinfachte Darstellung der besonderen Eigenschaften von IPv6, die grob aus [KFP00] entnommen wurde. Die Aufstellung ist nicht erschöpfend, umfasst aber alle wesentlichen Punkte. In dem IETF-Papier werden einige konkreten Probleme von IPv4 aufgeführt, und anschließend wird dargestellt, wie die neuen Eigenschaften von IPv6 diese Probleme beseitigen sollen. Der Intention des Papiers folgend werden die angesprochenen Probleme als mit

[1] Der Leser mag sich fragen, warum auf Version 4 die Version 6 folgt und ob ihm vielleicht eine ganze IP-Version entgangen ist. Er oder sie kann jedoch beruhigt sein: Die IETF hat sich entschlossen, eine Versionsnummer zu überspringen, um deutlich zu machen, welch einen großen Fortschritt die neue Version des IP-Protokolls mit sich bringt. Technisch gesehen ist die IP-Versionsnummer 5 dem ST-II-Protokoll zugewiesen, dass für Echtzeitübertragungen gedacht war.

IPv4 unlösbar und IPv6 als die einzig sinnvolle Lösung präsentiert, was bei näherer Betrachtung als nicht ganz realistisch einzustufen ist.

Vergrößerter Adressraum Durch das starke Wachstum des Internet werden die verfügbaren Adressen des IPv4-Adressraums knapp (wenn auch wesentlich langsamer als Mitte der 1990er-Jahre vorhergesagt). IPv6 verwendet 128 Bit zur Codierung einer Adresse, IPv4 dagegen nur 32 Bit. Dementsprechend gibt es theoretisch unter IPv4 etwa vier Milliarden IP-Adressen. IPv6 bietet dagegen 2^{128} theoretisch mögliche Adressen[2]. Ein Subnetz mit einem 64 Bit langen Präfix[3], welches die empfohlene Netz-Größe für einen einfachen Standort ist (in etwa vergleichbar zu einem Klasse-C-Netz in IPv4), hat etwa vier Milliarden mal mehr Adressen zur Verfügung als der gesamte theoretische Adressraum von IPv4. Auch bei großzügiger Vergabe von IPv6-Adressbereichen wird dieser enorme Adressraum also für lange Zeit ausreichen.

Zur Zeit wird mit IPv4 häufig das *NAT*-Verfahren (*Network Address Translation*) verwendet, um IP-Adressen einzusparen, so dass ein Standort mit mehreren tausend IP-Knoten in ihrem internen Netz aus Sicht des Internets nur eine (oder wenige) IP-Adressen belegt. Durch diese Maßnahme und die mittlerweile sehr restriktive Vergabe von IPv4-Adressen ist zu erwarten, dass der Vorrat an IPv4-Adressen noch einige Zeit ausreichen wird. Dies würde dagegen sprechen, dass die Einführung von IPv6 zwingend notwendig ist. Allerdings ist die Verwendung von *NAT* nur eine behelfsmäßige Lösung, die nicht die volle Funktionalität von IP garantiert.

Optimiertes Forwarding und Routing Eine Reihe von Vorgängen wurde in IPv6 einem "Streamlining" unterzogen: Der Aufbau des IP-Headers wurde vereinfacht, Ende-zu-Ende-Fragmentierung ist vorgeschrieben (*Path-MTU-Discovery*), *Classless Routing* (*CIDR*) wird verwendet.

In dem IETF-Papier wird argumentiert, das starke Wachstum des Internet würde dazu führen, dass die Backbone-Router an ihre Leistungsgrenzen gelangen, weil der IPv4-Header durch seine variable Länge komplizierter zu interpretieren sei. Da der IPv6-Header modularisiert wurde und in vielen Fällen nur der erste Bestandteil (mit fester Länge) vom Router betrachtet werden muss, soll IPv6 diese Überlastung der Router verhindern. Dieses Argument kann allerdings bei der sprunghaften Entwicklung der Leistung von Mikroprozessoren nicht besonders ernst genommen werden. Selbst wenn das Argument der Prozessorleistung zutreffen würde, könnte man eine Hardware-Lösung konzipieren, die den Großteil der vorkommenden IPv4-Header abdecken könnte und die seltener vorkommenden Varianten mit verschiedenen Optionen in Software abarbeiteten würde.

Außerdem wird argumentiert, dass durch das stark wachsende Internet die Routing-Tabellen der Backbone-Router derart an Größe zunehmen, dass die Performance einbricht oder sogar nicht genügend Speicher zur Verfügung steht. IPv6 soll hier durch die Verwendung von klassenlosem Routing (*CIDR*, *Classless Inter-Domain Routing*) Abhilfe schaffen. Das *CIDR*-Verfahren wird

[2] Die tatsächliche Zahl lautet 340282366920938463463374607431768211456.

[3] Das Netzpräfix ist die Bezeichnung des Netz-Anteils einer IPv6-Adresse, im Gegensatz zum Host-Anteil.

jedoch seit Jahren praktisch überall auch mit IPv4 verwendet, und stetig steigende Prozessorleistung und fallende Speicherpreise lassen dieses Argument wieder sehr schwach erscheinen. Allerdings hat IPv6 durch den wesentlich größeren Adressraum die Möglichkeit, die Adressbereiche günstiger aufzuteilen und verschiedene Netzbereiche besser zu aggregieren. So ist z.B. bei IPv6 vorgesehen, die Top-Level-Netzpräfixe unter anderem nach Providern zu vergeben, womit für das Routing zu einem bestimmten Provider (im optimalen Fall) nur ein einziger Routing-Eintrag notwendig ist. Ebenso kann z.B. für eine große Institution ein entsprechend dimensioniertes Netzpräfix vergeben werden, anstatt bei IPv4 eine Reihe von Klasse-C-Netzen zu vergeben.

Sicherheit Mit dem IPsec-Standard soll IPv6 transparente Sicherheit durch moderne Verschlüsselungsverfahren auf IP-Ebene bieten. Allerdings bietet noch längst nicht jede IPv6-Implementierung automatisch eine integrierte IPsec-Funktionalität, wenn auch die IPv6-Standards die Verfügbarkeit von IPsec vorschreiben. Genaugenommen ist jedoch nur die Unterstützung der ESP- und AH-Protokolle vorgeschrieben, die die Verschlüsselung und Authentifizierung realisieren. Das Management der Schlüssel ist ein ganz anderes Problem, das teilweise vom IKE-Protokoll (*Internet Key Exchange*) aufgriffen wird (siehe [RFC2409]). Die IKE-Spezifikation ist sehr umfangreich und bietet viel Raum für Inkompatibilitäten, außerdem ist die minimal vorgeschriebene Verschlüsselung (64 Bit DES) heutzutage nicht mehr ernst zu nehmen. Das Management von Zertifikaten, die zur abgesicherten Authentifizierung notwendig sind, ist kein Bestandteil von IKE. Mit ESP und AH abgesicherte IP-Kommunikation ist also von vielen anderen Faktoren abhängig, so dass die pauschale Aussage, "IPv6 bringt Sicherheit", nicht als zutreffend angesehen werden kann.

Zudem ist IPsec auch für IPv4 verfügbar. Dies ist also in Wirklichkeit keine besondere Eigenschaft von IPv6, wenn auch die IPsec-Entwicklung von IPv6 ausging.

Quality of Service (Dienstgüte) IPv6 soll Möglichkeiten zur Kontrolle der Dienstgüte bieten, indem im Header dafür bestimmte Felder vorgesehen werden, nämlich die Felder *traffic class* und *flow label*. Doch für eine tatsächliche Realisierung von Dienstgüte (bzw. von Dienstgüteklassen) sind andere Mechanismen wie *Integrated Services* oder *Differentiated Services* notwendig. Diese Ansätze lassen sich aber genauso mit IPv4 realisieren. So verwendet z.B. der Differentiated-Services-Ansatz ein Byte zur Spezifikation der Traffic-Klasse. Unter IPv4 wird dazu das *TOS*-Feld umdefiniert, bei IPv6 wird das *Traffic-Class*-Feld verwendet. Es gibt also keinen besonderen Vorteil bei der Verwendung von IPv6.

Erweiterte Möglichkeiten zur automatischen Konfiguration Durch *Link-local*-Adressen und *Stateless Autoconfiguration* wird die automatische Konfiguration von Netzwerkknoten erleichtert, ohne dass auf komplizierte Verfahren wie DHCP zurückgegriffen werden muss.

Jedem IPv6-Netzwerkinterface wird automatisch eine *Link-Local*-Adresse zugewiesen, die nur im lokalen Netzsegment gilt (also nicht routing-fähig ist). Für diese Adressen gibt es ein vordefiniertes Netz-Präfix (*fe80::/64*), das für alle Link-Local-Adressen gleich ist. Die unteren 64

Bits der Link-Local-Adresse werden mit dem EUI-64-Verfahren gebildet (siehe [EUI64]), effektiv wird dazu bei Ethernet-Interfaces die MAC-Adresse auf die unteren 64 Bit der IP-Adresse abgebildet. Damit können die Link-Local-Adressen eindeutig und vollautomatisch gebildet werden. Auf diese Weise ist, mit Einschränkung auf ein einzelnes, isoliertes Netzwerksegment, eine sofortige, konfigurationslose Verwendung möglich. Der Schlüssel zu dieser Möglichkeit ist natürlich der große Adressraum von IPv6. Unter IPv4 sind derartige Mechanismen grundsätzlich nicht möglich, da man (für Ethernet) zumindest die 48 Bit der MAC-Adresse auf die unteren 24 Bit eines privaten Klasse-A-Netzes (z.B. 10.0.0.0/24) abbilden müsste. Da eine solche Abbildung nicht eindeutig sein kann, müsste man den Fall einer Adressenkollision einplanen und zufällige Verfahren. Damit hätte man keine deterministische Adressenvergabe mehr.

Das *Stateless-Autoconfiguration*-Verfahren funktioniert ähnlich wie die Generierung der Link-Local-Adressen. Vorausgesetzt wird hier ein IPv6-Router, der mit per Multicast versendeten ICMP6-Paketen den Stationen in einem Netzsegment mitteilt, welches IPv6-Netzpräfix verwendet werden soll. Die Einzelheiten dieses Verfahrens lassen sich aus [RFC2462] entnehmen. Netzwerkknoten (Hosts), die das *Stateless-Autoconfiguration*-Verfahren verwenden, erhalten automatisch ihre IPv6-Adresse, in dem sie zu dem vom Router mitgeteilten Netzpräfix die EUI-64-Identifikation hinzufügen. Auf diese Weise erhält wieder jeder Host automatisch eine feste, im gesamten Internet eindeutige IPv6-Adresse. Dies ist nicht direkt vergleichbar mit der dynamischen Vergabe von IP-Adressen per DHCP (unter IPv6 oder auch IPv4), da bei DHCP die Adressen nur für einen bestimmten Zeitraum zugeteilt werden und nicht sichergestellt ist, dass ein Host immer die gleiche IP-Adresse erhält. Außerdem ist die Konfiguration und Administration eines DHCP-Servers wesentlich aufwendiger als das Aktivieren der für das Stateless-Autoconfiguration-Verfahren notwendigen ICMP6-Nachrichten in einem IPv6-Router.

Durch die Verwendung des Stateless-Autoconfiguration-Verfahrens erhält man also statische IP-Adressen, ohne den Aufwand für die manuelle Einrichtung der einzelnen Adressen treiben zu müssen. Nebenbei werden Netzparameter wie die Präfixlänge automatisch korrekt eingestellt.

Zusammenfassung und Bewertung Einige der aufgeführten Argumente, die zum Einsatz von IPv6 motivieren sollen, können bei realistischer Betrachtung nicht als stichhaltig angesehen werden, insbesondere die Themen Quality of Service und Sicherheit. Hier bietet IPv4 die gleichen Möglichkeiten, wenn auch diese Technologien erst in letzter Zeit für IPv4 nachgerüstet wurden und zum Zeitpunkt der Planung von IPv6 noch nicht vorhanden waren.

Die Argumente, die die Performance-Optimierung betreffen, sind zwar einleuchtend, aber durch die enormen Performance-Sprünge der Hardware in den letzten Jahren ist dieser Punkt nicht als fundamentales Problem für IPv4 zu sehen, das den Einsatz von IPv6 erzwänge.

So bleibt als letztes überzeugendes Argument die Vergrößerung des Adressraums übrig, und dieses Argument ist definitiv nicht von der Hand zu weisen. Zur Zeit reicht zwar der IPv4-Adressraum noch aus, indem einige “Sparmaßnahmen” ergriffen werden, aber langfristig kann das Problem der Adressenknappheit in IPv4 nicht gelöst werden. Betrachtet man die in naher Zukunft zu erwartende Unzahl von neuen Geräten, die Internet-Technologien verwenden, wie z.B.

PDAs, Mobiltelefone, Set-Top-Boxen oder sogar Haushaltsgeräte, oder etwa den zu erwartenden Einsatz von IP-Telefonie, so erscheint der Einsatz von IPv6 zwingend, um genügend große Adressbereiche zur Verfügung zu stellen.

Außerdem macht der stark vergrößerte Adressraum den Einsatz von Link-Local-Adressen und dem Stateless-Autoconfiguration-Verfahren erst möglich. Hier führt also eine Eigenschaft (bzw. die Haupteigenschaft) von IPv6 eher beiläufig zu einem Entwicklungsschritt, der bisher nicht möglich war, denn durch Link-Local-Adressen und Stateless Autoconfiguration wird ein Nachteil von IPv4 beseitigt: Bisher erforderte der Einsatz von TCP/IP immer eine explizite (manuell oder durch Aufsetzen von DHCP-, BOOTP- oder RARP-Servern) Konfiguration von IP-Adressen, was gerade bei kleinen Netzen wenig benutzerfreundlich ist.

Solange man nur innerhalb eines kleineren Netzes (mit nur einem Netzsegment) kommunizieren will, kann man also IPv6 völlig konfigurationslos betreiben. Dabei gibt es allerdings ein Problem: Man muss die IPv6-Adresse der Station wissen, die man ansprechen möchte. IPv6-Adressen sind zwangsweise sehr lang[4] und dementsprechend umständlich zu handhaben. Denkbar wäre nun, die Adressen aller Hosts zu sammeln und in Namenstabellen einzutragen (z.B. in der Datei */etc/hosts* auf Unix-Systemen), oder einen lokalen DNS-Server aufzusetzen. Damit wäre aber der Sinn des Szenarios verloren, ein IPv6-Netz ohne jeglichen Konfigurationsaufwand zu betreiben. Man benötigt also einen Namensdienst, der ohne Konfiguration, ohne Einsatz eines zentralen Servers funktioniert. Damit könnten die Anwender zum Identifizieren der anderen Hosts Namen verwenden statt der unhandlichen IPv6-Adressen, doch trotzdem wäre ein (nahezu) konfigurationsloser Betrieb möglich. Die einzig notwendige Konfiguration wäre die Vergabe eines Namens für jeden Host. Dies kann aber ohne besondere Kenntnisse durch den Anwender geschehen und ist sicher als einfacher und problemloser anzusehen als die korrekte Vergabe von IP-Adressen oder das Einrichten eines DNS-Servers.

Das Vorhandensein eines konfigurationslosen Namensdienstes ist also eine wichtige Bedingung, um Link-Local-Adressen praktisch nutzbar zu machen.

1.2 Problemstellung

Oft stellt man ein kleines Netzwerk aus nur wenigen Stationen zusammen, um zum Beispiel Dateien zu transferieren oder einen Drucker gemeinsam zu nutzen. Dies ist die typische Situation im SOHO-Bereich (Small Office/Home Office) oder auch im Heimbereich. In einer solchen Situation ist in der Regel auch kein Administrator oder eine Person mit entsprechenden Kenntnissen im Netzwerkbereich verfügbar, die Administration wird von normalen Anwendern vorgenommen. Hier ist es also besonders wichtig, dass das Netzwerk mit minimalem Konfigurationsaufwand betrieben werden kann, dass außer dem physikalischen Verbinden der Rechner mit Netzwerkkabeln oder etwa Wireless-LAN-Technologien kein weiterer Eingriff seitens des Anwenders notwendig ist, um das Netz zu nutzen ("Plug-and-Play-Networking").

[4] Ein Beispiel für eine Link-Local-Adresse ist `fe80::280:adff:fe19:a664`.

Bei dieser Arbeit wird im weiteren Verlauf von einem derartigen Szenario ausgegangen, bei dem eine kleine Anzahl von Rechnern als “Ad-Hoc-Netzwerk” zusammengeschaltet werden. Dieses “kleine Netz” soll vorerst nur isoliert auftreten, d.h. es findet kein Routing zu anderen Netzen statt, ebenso ist kein Internet-Anschluss vorhanden. Die Anzahl der Rechner soll sich im Rahmen von üblicherweise 10-20 bewegen, mit einer maximalen Obergrenze von 50 Rechnern. Sicherheitsaspekte sollen keine Rolle spielen, alle administrativen Arbeiten sollen von den Anwendern durchgeführt werden können, müssen also minimal gehalten werden.

Die Problemstellung besteht nun darin, in diesem “kleinen Netz” IP-Networking einzusetzen, ohne dass eine Konfiguration von IP-Adressen notwendig wäre. Dazu lassen sich, wie bereits beschrieben, die Link-Local-Adressen von IPv6 nutzen. Ebenfalls wurde bereits beschrieben, dass zur praktischen Nutzung der Link-Local-Adressen ein konfigurationsloser Namensdienst erforderlich ist. Ziel dieser Arbeit ist, dies konkret zu realisieren.

1.3 Struktur der Arbeit

In Kapitel 2 werden bereits vorhandene Namensdienste auf Tauglichkeit für das beschriebene Szenario des “kleinen Netzes” überprüft. Dabei wird festgestellt, dass kein geeigneter Namensdienst existiert, der die obigen Bedingungen gut erfüllt. Also muss ein eigener Namensdienst entworfen werden.

Dazu werden im Kapitel 3 technische Anforderungen an den zu entwerfenden Namensdienst formuliert, die sich aus den hier formulierten Bedingungen ableiten. Aus diesen Anforderungen heraus werden dann die weiteren Kapitel entwickelt.

In Kapitel 4 wird ein Protokoll entworfen, das die für den Namensdienst notwendige Netzwerkkommunikation spezifiziert.

Zudem wird eine Software implementiert, die das spezifizierte Protokoll nutzt, um eine Namensauflösung zu realisieren. Um diese Software aus Anwendungsprogrammen heraus nutzen zu können, wird zudem eine neue Architektur zur Integration von verschiedenen Namensdiensten ins System entworfen und implementiert. Die bisherige Architektur zur Verwendung von Namensdiensten muss dazu näher beleuchtet werden, dies geschieht in Kapitel 5. Anschließend kann der neue Entwurf in Kapitel 6 formuliert werden.

Nachdem nun die notwendigen Konzepte festgelegt wurden, wird in Kapitel 7 die Implementierung im Detail beschrieben.

Um die Funktionalität und die Leistungsfähigkeit der Implementation zu überprüfen, werden in Kapitel 8 die durchgeführten Tests beschrieben.

Abschließend wird in Kapitel 9 eine Bewertung zum Stand der Implementierung und ein Ausblick auf die Möglichkeiten zur Weiterentwicklung gegeben.

Kapitel 2

Evaluation vorhandener Namensdienste

2.1 Der Internet-Namensdienst – DNS

DNS (*Domain Name Service*) ist der Namensdienst, der praktisch immer im Zusammenhang mit IP-Netzen eingesetzt wird. Die genauen Spezifikationen für das Konzept und die Implementierung sind in [RFC1034] und [RFC1035] zu finden. DNS ist dafür ausgelegt, als Namensdienst für das gesamte Internet zu fungieren und passt daher von vornherein nicht zu dem Szenario des "kleinen Netzes". Aufgrund der praktisch vollständigen Verbreitung von DNS scheint es trotzdem angeraten, Überlegungen anzustellen, ob man DNS nicht doch für dieses Szenario nutzen kann, möglicherweise mit kleinen Anpassungen.

Wichtig hierfür ist die Art, wie die Verwaltung von Namen in DNS funktioniert. Die Namen, die im Namensraum von DNS vorkommen, werden in hierarchisch angeordnete *Domains* aufgeteilt, so dass jeder Name zu einer Domain gehört und auch den Namen der Domain enthält, zu der er gehört. So gehört der Hostname *pc-2n00.6lab.inf.fh-rhein-sieg.de* zu der Domain *6lab.inf.fh-rhein-sieg.de*, welche wiederum zu der Domain *inf.fh-rhein-sieg.de* gehört und so weiter. Interessant ist hier der Aspekt, dass diese hierarchische Anordnung auch eine administrative Bedeutung hat, denn für jede Domain (und Sub-Domain) gibt es in der Regel einen eigenen *Nameserver*. Jeder Nameserver ist dafür verantwortlich, die Namensinformationen für die von ihm zu verwaltende Domain zur Verfügung zu stellen. Fragen nach Namen von übergeordneten Domains werden an den Nameserver der übergeordneten Domain weitergereicht, Fragen nach Namen aus untergeordneten Domains werden an untergeordnete Nameserver delegiert[1]. So entsteht eine hierarchische verteilte Datenbank.

Dieses Konzept ist zunächst völlig ungeeignet als konfigurationsloser Namensdienst. Das Einrichten von Nameservern erfordert einen nicht unerheblichen administrativen Aufwand, außerdem muss jeder Host, der DNS nutzen will, die IP-Adresse mindestens eines Nameservers kennen.

[1] Tatsächlich ist die Aufteilung in der Realität nicht derart restriktiv festgelegt, ein Nameserver erhält die Verantwortung für *Zonen*, die mehrere Domain-Ebenen umfassen können. Außerdem kann ein Nameserver für mehrere Zonen verantwortlich sein, oder mehrere Nameserver für die gleiche Zone.

Da das DNS-Protokoll auch über UDP genutzt werden kann, wäre es denkbar, einen Nameserver über ein Broadcast-Paket (oder Multicast-Paket) anzusprechen. Unter Umständen müsste dazu sowohl die Client- als auch die Server-Seite der DNS-Software geringfügig modifiziert werden. Die Nutzung von Broadcast-Paketen würde die Notwendigkeit beseitigen, die IP-Adresse des Nameservers im voraus zu kennen. Da aber kein zentraler Server verwendet werden darf, müsste jeder Netzwerkknoten einen eigenen Nameserver starten, der nur den eigenen Namen und die eigene Adresse enthält. Bei einer Anfrage über ein Broadcast-Paket würden dann alle Nameserver eine Antwort senden, egal, ob der Name gefunden wurde (auf dem passenden Host), oder nicht. Damit würde die Anzahl der für jede Namensauflösung erzeugten Pakete im Quadrat zur Anzahl der Hosts ansteigen, was definitiv nicht wünschenswert ist. Sehr problematisch wäre auch ein Parallelbetrieb mit realen DNS-Servern, falls z.B. das "kleine Netz" zu einem späteren Zeitpunkt an das Internet angeschlossen werden soll.

2.2 Konfigurationslose Namensdienste

NetBIOS Name Service

Die NetBIOS-Spezifikation in [IBM84] definiert eine einfache API zur Netzwerkkommunikation in PC-Netzwerken. NetBIOS kann über verschiedene Protokolle realisiert werden, unter anderem auch über IP. In [RFC1001] und [RFC1002] wird definiert, wie NetBIOS über IP eingesetzt werden kann. Dazu gehört auch die Definition eines auf Broadcast-Paketen basierenden Namensdienstes, der NetBIOS-Namen in IP-Adressen umsetzen kann[2]. Durch die Verwendung von Broadcast-Paketen ist demnach ein konfigurationsloser Betrieb möglich. Allerdings sind die NetBIOS-Namen einigen Limitierungen unterworfen: Namen sind immer exakt 16 Zeichen lang, nicht benutzte Zeichen werden mit Leerzeichen aufgefüllt, außerdem sind nur Großbuchstaben erlaubt. Der NetBIOS-Namensraum ist "flach", eine Unterteilung in verschiedene Domains ist nicht möglich. Stattdessen existieren Namen für *Gruppen* (*Group NetBIOS Name*), wobei mehrere Netzwerkknoten sich einen Namen "teilen" können. Darüber kann dann indirekt eine für den Anwender sichtbare Unterteilung des Namensraums vorgenommen werden.

Zur Codierung verwendet der NetBIOS-Name-Service eine Variation des DNS-Paketformats. Demnach wäre es theoretisch relativ einfach, in einem solchen Paket auch eine IPv6-Adresse unterzubringen. Allerdings ist in [RFC1002] für den Typ der im Paket enthaltenen Adresse ausschließlich IPv4 definiert (ein *A-Record* in DNS-Terminologie). Dies ist auch nicht weiter verwunderlich, da die Veröffentlichung dieser RFCs lange vor der Entwicklung von IPv6 stattfand. Demnach ist NetBIOS und NetBIOS über IP also auch als relativ veraltete Technologie einzustufen.

Diese Betrachtungen führen zu dem Schluss, dass der NetBIOS-Name-Service für den gesuchten Einsatzzweck zwar nicht ideal, aber zumindest prinzipiell geeignet wäre. Anwendungs- und

[2] Tatsächlich ist die Verwendung von Broadcast-Paketen für den *NetBIOS Name Service* nur eine Möglichkeit, darüber hinaus kann auch ein spezieller Netzwerkknoten für die Namensauflösung verantwortlich gemacht werden, der dann direkt angesprochen wird.

Serverseite der Software müssten dazu angepasst werden, so dass auch IPv6-Adressen verarbeitet werden können. Damit würde allerdings auch gleich der eventuelle Vorteil zunichte gemacht, mit bestehenden Implementierungen kompatibel zu sein. Die Nachteile würden bestehen bleiben: Obwohl NetBIOS selbst nicht eingesetzt werden soll, muss für die Namensauflösung NetBIOS-Kommunikation implementiert werden, die einschränkende NetBIOS-Namensgebung muss übernommen werden. Diese Tatsachen und die Feststellung, dass NetBIOS in der Praxis ausschließlich im Windows- und PC-Bereich eingesetzt wird, lassen diesen Ansatz als nicht empfehlenswert erscheinen.

NBP – AppleTalk

AppleTalk ist eine Protokollfamilie, die von Apple für die Vernetzung von Macintosh-Rechnern entwickelt wurde[3]. Das Design ist speziell für konfigurationslosen Betrieb und generell hohe Benutzerfreundlichkeit ausgelegt worden. Die AppleTalk-Adressen werden dynamisch und automatisch vergeben, ebenso erfolgt die Namensauflösung über ein Broadcast-Verfahren ohne einen zentralen Server. In [Apple90] findet sich eine ausführliche Beschreibung der AppleTalk-Protokollfamilie. Eine AppleTalk-Vernetzung ist unter anderem über eine serielle RS422-Verbindung oder Ethernet möglich, wobei die serielle Verbindung heute natürlich praktisch keine Bedeutung mehr hat, früher jedoch als Low-Cost-Option zur Vernetzung durchaus eingesetzt wurde.

Ein AppleTalk-Netzwerk kann aus verschiedenen Netzen bestehen, wobei die einzelnen Netze Netzwerknummern erhalten und in Zonen aufgeteilt werden können. Zur Adressierung eines speziellen Dienstes wird eine AppleTalk-Adresse, die eine Netzwerknummer beinhaltet, und eine Socket-Nummer verwendet.

Von besonderem Interesse ist das Protokoll, mit dem Namen in AppleTalk-Adressen aufgelöst werden. Dieses Protokoll wird *Name Binding Protocol* (*NBP*) genannt und ist in [Apple90, Kapitel 7: Name Binding Protocol] detailliert beschrieben. Dabei haben einzelne Netzwerkknoten keinen expliziten Namen. Vielmehr werden sogenannte *Network Visible Entities* definiert, die einen Namen tragen können.

Diese *Entities* stellen tatsächliche Dienste dar, die über das Netz angesprochen werden können. So trägt ein Host, auf dem ein File-Server läuft, keinen Namen, sondern der File-Service selbst. Ein Host kann verschiedene Dienste anbieten, die als unterschiedliche Namen im Netzwerk sichtbar sind. Ein Name beinhaltet den Namen des Dienstes, einen String, der den Typ des Dienstes kennzeichnet und den Namen der Zone. Jedes dieser drei Elemente ist ein String mit einer maximalen Länge von 32 Zeichen. Über NBP werden diese Namen dann in eine AppleTalk-Adresse und eine Socket-Nummer übersetzt.

Daraus ergeben sich Eigenschaften, die für die Verwendung im Szenario aus Abschnitt 1.2 ideal sind. Leider sind die AppleTalk-Protokolle in keiner Weise mit IP kompatibel, und NBP ist speziell für die Namensauflösung nach AppleTalk-Adressen konzipiert. Eine Anpassung an

[3]AppleTalk und Macintosh sind eingetragene Warenzeichen von Apple Computer, Inc.

IPv6-Adressen wäre denkbar, käme aber einer Neuentwicklung des Protokolls gleich. Trotzdem sind die Eigenschaften von NBP in Hinsicht auf die Einfachheit der Benutzung vorbildlich, und sollten als Maßstab für den einzusetzenden Namensdienst angesehen werden.

2.3 Verzeichnisdienste als Namensdienste

LDAP

LDAP (*Lightweight Directory Access Protocol*) wird in Unternehmensnetzen häufig als Verzeichnisdienst für viele Arten von strukturierten Informationen genutzt. Es ist auch möglich, LDAP zur Namensauflösung oder zum Ablegen von Benutzerinformationen zu verwenden. LDAP ist jedoch für den Einsatz von zentralen Servern konzipiert und erfordert generell auch eine zentrale Administration der LDAP-Server. Damit ist ein konfigurationsloser Betrieb ausgeschlossen. Es wäre zwar möglich, einen LDAP-Server über ein separates Broadcast-Protokoll zu lokalisieren und anschließend direkt anzusprechen, aber damit ist noch nicht das Problem gelöst, dass automatisch ein zentraler Server ausgewählt werden muss. Zudem scheint LDAP für den Einsatz in einem kleinen Netz übermäßig komplex. LDAP lässt sich also für das gegebene Szenario mit vertretbarem Aufwand nicht verwenden.

SLP

SLP (*Service Location Protocol*) ist ein Verzeichnisdienst, der für den Einsatz in Unternehmensnetzen entworfen wurde. Wie aus dem Namen zu entnehmen ist, soll SLP zum Auffinden von Netzwerkdiensten dienen. Die relevanten Standards hierzu sind [RFC2608] und [RFC2609]. Netzwerkdienste können auf SLP-Servern registriert und anschließend von Anwendungen abgefragt werden. Im Gegensatz zu LDAP ist jedoch das automatische Auffinden von SLP-Servern und das automatische Erstellen von SLP-Verzeichnissen vorgesehen. Dabei werden Broadcast-Verfahren benutzt, um SLP-Server zu lokalisieren und danach mit einer direkten Verbindung anzusprechen, sowohl zur automatischen Registrierung von Diensten als auch zur Suche. Jeder Host kann seinen eigenen SLP-Server (*Service Agent*) starten, oder es können spezielle *Directory Agents* verwendet werden, die die Informationen sammeln und einheitlich zur Verfügung stellen. Dienste werden in SLP als URLs dargestellt, die zusätzlich noch verschiedene Attribute tragen können. Die Attribute dienen als Zusatzinformation für den Anwender und können auch dazu verwendet werden, um komplexe Suchanfragen zu formulieren[4].

Durch die Möglichkeit, ohne einen zentralen Server auszukommen, kommt SLP zunächst als mögliche Lösung für die Problemstellung in Frage. Bei näherer Betrachtung zeigen sich jedoch einige Probleme:

[4] Ein Beispiel für eine Suchanfrage in SLP wäre “Suche einen Laserdrucker, der sich im 3. Stock befindet und mindestens 10 Seiten pro Minute druckt.”

SLP ist zur Lokalisierung von Diensten konzipiert, eine Namensauflösung ist nicht direkt vorgesehen, vielmehr wird davon ausgegangen, dass ein Host, der SLP benutzt, bereits in der Lage ist, Namen aufzulösen.

Prinzipiell dürfen allerdings auch numerische Adressen in den in SLP-Verzeichnissen abgelegten URLs vorkommen. Man könnte nun einen Dienst definieren, der nur die Existenz eines Hosts anzeigt, also nicht mit einem realen Dienst assoziiert ist, und diesen zur Namensauflösung verwenden. Da ein SLP-Eintrag keinen expliziten Namen hat, müssten Attribute verwendet werden, um eine Zuordnung eines Namens zu der in der URL enthaltenen Adresse zu bilden. Eine andere Option wäre, dass man keine Namensauflösung im klassischen Sinne zulässt, sondern nur diejenigen Hosts sichtbar werden lässt, die auch einen (oder mehrere) Dienste anbieten, und die Namen von den Diensten abzuleiten.

Was jedoch SLP als Lösung für die Problemstellung am meisten ungeeignet erscheinen lässt, ist die Tatsache, dass SLP sehr komplex aufgebaut ist, da es für größere Unternehmensnetze entworfen wurde. Für ein Netz aus wenigen Rechnern scheint eine aufwendige SLP-Implementierung, die den vollen Standard implementiert, eher überdimensioniert. Hinzu kommt das Problem, dass die Verwendung von SLP zur reinen Namensauflösung bisher im Standard nicht definiert ist. Betrachtet man den zu erwartenden Aufwand der Implementierung und die zur Zeit geringe Verbreitung von SLP, so scheint dieser Lösungsansatz eher ungünstig zu sein.

2.4 Ergebnis

Es gibt bisher keinen Namensdienst für IPv6 oder IPv4, der für den Anwender einfach zu benutzen ist und der keine Konfiguration braucht. Die eben betrachteten Kandidaten lassen sich entweder gar nicht konfigurationslos betreiben, haben technische Limitierungen oder sind für kleine Netze überdimensioniert und nicht adäquat. Das NBP-Protokoll der AppleTalk-Protokollfamilie würde sich ideal eignen, ist aber nicht für IP verfügbar.

Also muss ein neuer Namensdienst mit den gewünschten Eigenschaften entworfen und implementiert werden, um die Möglichkeiten der Link-Local-Adressen von IPv6 nutzen zu können.

Kapitel 3

Anforderungen an einen neuen Namensdienst

Aus Anwendersicht ist das gewünschte Ziel, andere Rechner im lokalen Netz “zu sehen”, das heißt, dass die anderen Rechner mit Namen ansprechbar sind. Es sollte auch eine Liste aller eingeschalteten Rechner verfügbar sein, so dass sie z.B. visuell in einem Fenster dargestellt werden können. Anders gesagt sollen die Namen der Rechner im Namensraum aller Hostnamen automatisch auftauchen. Nicht unbedingt erforderlich, aber wünschenswert wäre, wenn auch die auf einem Rechner angebotenen Dienste wie z.B. Dateiserver- oder Printserver-Dienste direkt erkennbar wären. Durch die Unterscheidung von verschiedenen Diensten könnten dem Anwender dann nur diejenigen Netzwerkknoten in einer Auswahlmenge präsentiert werden, die ihn interessieren. So könnten z.B. bei der Auswahl eines Netzwerk-Druckers nur die Rechner angezeigt werden, die auch einen Netzwerk-Druckservice anbieten (oder Drucker mit integriertem Netzwerk-Interface).

Aus administrativer Sicht müssen diese Eigenschaften ohne einen zentralen Server und ohne manuelle Konfiguration erreicht werden, da es bei dem verwendeten Szenario des “kleinen Netzes” keinen Administrator gibt und nicht vorhersehbar ist, welcher Rechner zu welchem Zeitpunkt eingeschaltet ist. Daraus folgt, dass man für den Namensdienst einen Broadcast-Mechanismus verwenden muss, da nur auf diese Weise mit vorher unbekannten Rechnern kommuniziert werden kann.

Die Verwendung von Broadcast-Kommunikation ist jedoch netzwerktechnisch ungünstig, da bei jedem Senden eines Broadcast-Pakets alle angeschlossenen Stationen es verarbeiten müssen. Daher sollte die Broadcast-Kommunikation möglichst reduziert werden. Es sollte vermieden werden, dass jedesmal, wenn eine Anwendung auf einem einzelnen Rechner eine Namensauflösung anfordert, eine Broadcast-Kommunikation stattfinden muss. Um das zu erreichen, muss jeder Rechner selbst zum Namensdienst-Server werden und ein Verzeichnis aller anderen Rechner und derer Adressen anlegen. Dieses Verzeichnis muss automatisch “gelernt” werden, damit keine Konfiguration notwendig ist. Dadurch wird auch allgemein die zusätzliche Netzlast reduziert.

Als Konsequenz ergibt sich daraus also die Notwendigkeit, dass auf jedem Rechner ein Prozess ständig im Hintergrund laufen muss (ein Dæmon im Unix-Jargon), der Namen und Adressen anderer Rechner "sammelt" und sie auf Anfrage den Anwendungen mitteilt. Dieser Dæmon nimmt auch gleichzeitig die Aufgabe wahr, Namen und Adresse des eigenen Rechners den anderen Rechnern für ihre Sammlung mitzuteilen.

Sehr wichtig für einen neuen Namensdienst ist die Eigenschaft, dass er von den Anwendungen transparent genutzt werden kann. Nach Möglichkeit sollten bestehende Anwendungen den Namensdienst nutzen können, ohne neu compiliert werden zu müssen. Auf jeden Fall sollte der neue Namensdienst über die bestehenden Funktionen der Systembibliotheken genutzt werden können, so dass dafür keine Änderungen am Quelltext der Anwendungen, also Neu- oder Umprogrammierung, notwendig werden. Aus dieser Forderung leitet sich die Aufgabe ab, dass die (in C geschriebene) Systembibliothek entsprechend modifiziert werden muss, um den neuen Namensdienst zu nutzen. Dazu müssen eine Reihe von C-Funktionen aus der C-Bibliothek (*libc*) erweitert werden, so dass sie zusätzlich zu den bisher verfügbaren Namensdiensten (z.B. DNS) auch mit dem eben erwähnten Dæmon kommunizieren können.

Kapitel 4

Entwurf des Namensauflösungs-Protokolls

4.1 Zielsetzung

Vor dem Entwurf sollen zunächst die Zielsetzungen und die verschiedenen Anforderungen an das neue Protokoll gesammelt und konkretisiert werden. Das Protokoll soll die in Kapitel 3 geforderte Funktionalität bieten und Raum für spätere Erweiterungen lassen. Aus der Sicht des Software-Entwicklers sollte das Protokoll möglichst einfach zu programmieren sein und keine komplizierte, aufwendig zu implementierende Codierung der Daten verwenden. Aus Sicht des Netzmanagements sollte das Protokoll wenig Overhead (pro Paket) erzeugen und auch insgesamt möglichst wenig zusätzliche Netzlast erzeugen, gemessen sowohl an übertragenem Datenvolumen als auch an der Anzahl der gesendeten Pakete. Ist die Implementierung einfach zu programmieren, wird dies dazu beitragen, dass das Protokoll in der Praxis auch tatsächlich und möglichst oft implementiert wird. Eine geringe Netzlast durch das neue Protokoll wird die Bedenken eines Systemadministrators verringern, ein neues, unbekanntes Protokoll auch einzusetzen. Anders ausgedrückt, ist nicht nur die Funktionalität des Protokolls wichtig. Eine sehr große Rolle spielt auch der Aufwand, um ein neues Protokoll und die zugehörige Software einzusetzen, der natürlich so gering wie möglich sein soll. Dementsprechend wird bei diesem Entwurf großen Wert darauf gelegt, einen guten Kompromiss zwischen einfacher Implementierung und umfangreicher Funktionalität zu finden. Die Implementierung soll einfach und geradlinig sein, und das Protokoll und der dadurch erzeugte Netzverkehr soll “nicht auffallen”.

4.2 Konzepte und Definitionen

Das Protokoll erhält den Namen *IPv6Lookup*, obwohl seine Verwendung prinzipiell nicht auf das IPv6-Protokoll beschränkt ist. Das Protokoll dient generell dazu, *Namen* auf Netzwerkadressen (also hier IPv6-Adressen abzubilden). Zu einem Namen gehört auch eine *Klasse*, und die Kombination aus einem Namen und einer Klasse stellt einen *Dienst (Service)* dar (genaugenommen eine Instanz eines Dienstes). Die *Klasse* repräsentiert die Art des Dienstes, wobei verschiedene

Dienste die gleiche Adresse haben können, oder aber Dienste mit unterschiedlicher Klasse den gleichen Namen haben können. Die Port-Nummer eines Dienstes, bzw. allgemein der *Service Access Point* (SAP), wird über das *IPv6Lookup*-Protokoll nicht übertragen, er wird implizit über die Service-Klasse festgelegt. Die Definition von verschiedenen Dienstklassen ist für spätere Erweiterungen vorgesehen, der einfachste Dienst ist sozusagen die bloße Existenz eines Netzwerkknotens (*host*). Dies ist die einzige Dienstklasse, die von der derzeitigen Implementierung verwendet wird. Eine Anfrage nach diesem "minimalen Dienst" ist dabei äquivalent mit einer einfachen Namensauflösung. Spätere Implementierungen können dann die Dienstklasse verwenden, um Dienste im üblichen Sinn anzukündigen. Zu jedem Dienst gehört schließlich noch ein *Kommentar*, im Protokoll *Description* genannt, der vom Nutzer frei wählbar ist und dem Anwender eine Beschreibung oder andere weitergehende Informationen über den Dienst geben soll.

Zu beachten ist bei dieser Konzeption, dass eine Dienstinstanz aus Sicht des IPv6Lookup-Protokolls auf gleicher Stufe wie ein Netzwerkknoten steht. Es besteht kein expliziter Zusammenhang zwischen dem Dienst und dem Host, der diesen Dienst zur Verfügung stellt, bis auf die Tatsache, dass beide die gleiche Adresse haben. Die Namensauflösung geschieht für Dienste, nicht für Netzwerkknoten. Es wird also nicht zuerst festgestellt, welcher Host welchen Dienst anbietet und dann in einem zweiten Schritt die Adresse des Hosts erfragt, sondern es wird direkt nach der Adresse des Dienstes gefragt. Die Information, auf welchem Host sich welcher Dienst befindet, kann vom IPv6Lookup-Protokoll nicht dargestellt werden. Der Grund für diese Sichtweise ist, dass ein Benutzer beim Suchen eines Dienstes in der Regel zuerst nach Kategorien vorgeht, die für ihn leichter verständlich sind als detaillierte Informationen über den Aufbau der Netzressourcen. Bei der Suche nach einem speziellen Drucker, beispielsweise in einer visuellen Darstellung der verfügbaren Dienste, wird er zuerst eine Kategorie "Drucker" aufsuchen und dort nach einer speziellen Instanz suchen. Dabei wird der Benutzer sich nicht dafür interessieren, an welchen Host der gesuchte Drucker angeschlossen ist, sondern vielleicht eher, in welchem Büroraum sich der Drucker befindet. Auch wird es für den Benutzer wenig hilfreich sein, wenn Drucker den Namen des Hosts tragen, an den sie angeschlossen sind. Nützlicher ist, wenn die Drucker deskriptive Namen tragen, wie etwa den Modellnamen. Damit passt sich das Protokoll der "Weltsicht" des Benutzers an, nicht umgekehrt, und trägt damit zur Benutzerfreundlichkeit, zum "ease of use" bei.

Da das IPv6Lookup-Protokoll im Broadcast-Verfahren eingesetzt werden soll, ist dadurch klar, dass eine Kommunikation nur innerhalb eines Netzsegments stattfinden kann, also bei Ethernet innerhalb einer Broadcast-Domäne. Sollen Hosts über IPv6Lookup erreichbar sein, müssen sie sich also auf IP-Ebene in dem gleichen Netzwerk (gleiches Netz-Präfix) befinden und über einen Hub oder Switch direkt miteinander verbunden sein. Alle Hosts, die auf diese Weise erreichbar sind und das IPv6Lookup-Protokoll unterstützen, werden im folgenden als *Nachbarn* bezeichnet. Alle Nachbarn bilden wiederum eine *Nachbarschaft*. Die vom IPv6Lookup-Protokoll verwendeten Namen gelten jeweils nur in ihrer Nachbarschaft, auch wenn Namen verwendet werden können, deren Gültigkeit sich über weitere Bereiche erstreckt (wie z.B. DNS-Namen). Es gibt keine weitere Strukturierung einer Nachbarschaft auf Protokollebene. Es ist jedoch für Anwendungen möglich, durch geeignete Namenskonventionen eine Aufteilung einer Nachbarschaft in Zonen oder Domänen vorzunehmen.

4.3 Paketaufbau und Codierung

Um die oben beschriebene Zielsetzung zu erreichen, wurde ein textbasiertes Protokoll entworfen. Die Protokolldaten werden als einfache Strings codiert, so dass die Verarbeitung mit den üblichen Stringbehandlungsmethoden geschehen kann.

Eine String-Codierung vermeidet auch automatisch Probleme mit unterschiedlichen Prozessorarchitekturen durch unterschiedliche Wortlängen oder Big- und Little-Endian-Systeme. Die Kodierung eines Pakets als String ist nahezu trivial. Die Decodierung geschieht durch String-Parsing, was bei weniger Aufwand deutlich flexibler ist als eine binäre Codierung. So kann man z.B. bestimmte Teilstrings einfach weglassen, falls die entsprechende Information nicht benötigt wird. Bei einer binären Kodierung müsste auch das Fehlen einer Information im Paket codiert werden und im Programmcode dafür ein besonderer Fall vorgesehen werden (oder aber das Vorhandensein eines Protokollelements müsste durch ein eigenes Protokollelement dargestellt werden, mit ebenfalls höherem Programmieraufwand). Noch deutlicher wird diese Flexibilität, wenn es um Erweiterungen des Protokolls geht. Es ist wesentlich einfacher, an einen String noch einen weiteren String anzuhängen oder in einem Teilstring zusätzliche Informationen unterzubringen, als eine binär codierte Datenstruktur zu ändern. Binär codierte Protokolle enthalten für derartige Fälle meistens eine Versionsnummer, um bei Erweiterungen die Kompatibilität zu gewährleisten. Im Programmcode müssen dann Pakete mit verschiedenen Versionsnummern unterschiedlich behandelt werden. Bei der hier verwendeten String-Kodierung muss nur die Funktion zum Parsen des Strings geringfügig modifiziert werden, um weitere Protokollelemente einzubauen. Ein Beispiel dieser Flexibilität ist das Kommentar-Feld des IPv6Lookup-Protokolls, das weiter unten näher beschrieben wird. Dieses Feld ist optional und kann auch ganz fehlen. Trotzdem sieht die Implementierung dafür keinen Spezialfall vor. Es wird nicht speziell geprüft, ob dieses Feld existiert oder nicht. Eine andere einfache Erweiterungsmöglichkeit ist durch die Variation der bisher vorhandenen String-Bestandteile gegeben. Nachteile der String-Codierung gegenüber einer binären Codierung gibt es allerdings auch. Die String-Codierung nimmt in der Regel mehr Speicherplatz ein, und das Parsen der Strings verbraucht mehr Rechenzeit. Da die im IPv6Lookup-Protokoll übertragenen Informationsmengen jedoch sehr gering sind, fällt der erhöhte Platzbedarf nicht sehr ins Gewicht, in vielen Fällen bleibt die Paketgröße unterhalb der minimalen Ethernet-Paketgröße von 64 Bytes. Aus dem gleichen Grund ist der erhöhte Rechenzeitbedarf vernachlässigbar, das Bearbeiten von wenigen kurzen Textzeilen verursacht gerade bei den heute üblichen Prozessorleistungen keine messbaren Verzögerungen.

Bei IPv6Lookup handelt es sich um ein symmetrisches Protokoll, bei dem Anfrage- und Antwortpakete nach dem selben Schema aufgebaut sind. Als Transportprotokoll wird UDP verwendet, die derzeitige Implementierung verwendet die willkürlich gewählte Port-Nummer 28301. Im wesentlichen gibt es zwei unterschiedliche Pakettypen, *Report* und *Query*. Ein *Query*-Paket stellt eine Anfrage nach der einem bestimmten Dienst zugeordneten Adresse dar. Mit einem *Report*-Paket kann die einem Dienst zugehörige Adresse mitgeteilt werden. In einem Paket können auch mehrere Dienste angegeben sein, dies ist jedoch nur für *Report*-Pakete erlaubt. In diesem Fall enthält ein Paket mehrere einzelne *Reports*. Dementsprechend enthält ein *Report*-Paket mit einer einzelnen Dienst-Angabe einen einzigen *Report*. Ein *Query*-Paket enthält immer eine einzelne

Query.

Hier nun die Syntax einer einzelnen Dienst-Spezifikation:

Syntax

<Paket-Typ><Service-Klasse>/<Name>/<Adresse>[/*<Kommentar>*]

Eine Dienste-Spezifikation beginnt mit einem einzelnem Zeichen (ein Byte), das den Paket-Type repräsentiert. Es folgen mehrere Felder mit variabler Länge, die mit einem '/'-Zeichen (ASCII-Code 47) voneinander getrennt werden. Dementsprechend darf in den einzelnen Feldern das '/'-Zeichen nicht vorkommen oder es muss anders dargestellt werden. Befinden sich mehrere dieser Dienst-Spezifikationen in einem Paket, so werden sie durch Zeilenende-Zeichen (*line-feed*, in C-String-Notation *n*, ASCII-Code 10) voneinander getrennt. Ein Paket enthält damit also mehrere Text-Zeilen. Die letzte Zeile (oder die einzige) kann, muss aber kein Zeilenende-Zeichen enthalten. Das gesamte Paket (bzw. der Text) muss mit einem Null-Byte abgeschlossen werden, innerhalb der Felder dürfen keine Null-Bytes vorkommen (nach Definition ist das Paket beim ersten Null-Byte zu Ende). Die einzelnen Felder sollen nun genauer beschrieben werden.

Paket-Typ Der Paket-Typ besteht aus einem Zeichen (ein Byte). Gültige Pakettypen sind folgende:

"+": Dies ist ein *positiver Report*, mit dem ein bestimmter Dienst angekündigt wird.

"-": Dies ist ein *negativer Report*, mit dem angekündigt wird, dass der angegebene Dienst nicht mehr existiert.

"?": Dies ist ein *Query*-Paket, in dem nach der Adresse des angegebenen Dienstes gefragt wird.

Service-Klasse Die Service-Klasse ist ein String, der die Art des angebotenen oder angefragten Dienstes beschreibt. Die einzige von der derzeitigen Implementierung verwendete Klasse ist "**host**". Diese Klasse stellt die simple Existenz eines Netzwerkknotens dar und wird zur Namensauflösung verwendet. Andere Beispiele für Dienstklassen wären "**lpd**" für einen Netzwerk-Druck-Dienst über das LPD-Protokoll, oder "**nfs**" für einen NFS-Server.

Name Das *Name*-Feld enthält den Namen des entsprechenden Dienstes oder Hosts. Hier wird der normale Hostname verwendet, wie er z.B. von der *gethostname()*-Funktion zurückgeliefert wird. Der Name kann und sollte, falls vorhanden, auch den Domain-Namen enthalten (*fully qualified domain name, FQDN)*. Das Protokoll definiert aber keine besondere Semantik für Namen mit Domain-Anteil wie z.B. automatische Ergänzung der Domain für Anfragen. Das Protokoll spezifiziert auch keine besondere Codierung für die Zeichen des Strings. Es wird davon

ausgegangen, das alle beteiligten Anwendungen den gleichen Zeichensatz und die gleiche Zeichenkodierung verwenden. Es wird empfohlen, als Namen gültige DNS-Namen zu verwenden.

Bei Queries hat ein einzelnes '*'-Zeichen (ASCII-Code 42) als Inhalt des Namensfeldes eine besondere Bedeutung. Eine solche Query gilt für alle Dienstinstanzen mit der angegebenen Dienstklasse, demnach sollten alle Hosts, die den gewünschten Dienst anbieten, mit einem Report antworten. Dies wird auch als *Query-All* bezeichnet.

Adresse Dieses Feld enthält die Adresse des Hosts, auf dem der entsprechende Dienst erreichbar ist. Die Adresse bezieht sich implizit auf das Protokoll, mit dem das IPv6Lookup-Paket gesendet wurde. Bei der üblichen Verwendung mit dem IPv6-Protokoll steht also offensichtlich im Adressen-Feld die IPv6-Adresse. Demnach ist es bei der hier vorgestellten Fassung des IPv6Lookup-Protokolls nicht möglich, über IPv6 beispielsweise IPv4-Adressen mitzuteilen. Es wäre aber möglich, das IPv6Lookup-Protokoll über IPv4 zu senden und damit IPv4-Adressen mitzuteilen.

Die Kodierung der Adresse erfolgt in einer ASCII-Repräsentation, wie sie von der *inet_ntop()*-Funktion zurückgeliefert wird. Sie muss mit der *inet_pton()*-Funktion in eine binäre Adresse konvertiert werden können.

Wenn das Adressen-Feld ausschließlich aus dem Zeichen '.' besteht (ASCII-Code 46), hat dies eine besondere Bedeutung. In diesem Fall gilt die Quelladresse des Pakets als die Adresse des Dienstes. Ein '.' als Adresse ist nur bei Reports erlaubt.

Bei Queries ist das Adressenfeld leer und wird vom Empfänger ignoriert. Bei negativen Reports kann das Adressenfeld ebenfalls leer sein.

Kommentar Dieses Feld enthält einen beliebig wählbaren String, der eine nähere Beschreibung des jeweiligen Dienstes darstellt. Das Feld ist optional, kann also leer sein oder ganz fehlen. Fehlt das Kommentar-Feld ganz, wird auch das vorangehende '/'-Zeichen, das normalerweise das Adressen-Feld von dem Kommentar-Feld trennt, weggelassen. Das Protokoll spezifiziert keine Zeichenkodierung für den String. Wie beim Namens-Feld wird davon ausgegangen, dass alle beteiligten Anwendungen den gleichen Zeichensatz und die gleiche Zeichenkodierung verwenden. Empfohlen wird eine ISO-Latin-1-Codierung. Das Kommentar-Feld ist dazu vorgesehen, dem Benutzer als Hilfe bei der (visuellen) Auswahl eines Dienstes zu dienen. Es sollte vom System und von den Anwendungen nicht interpretiert werden, das Funktionieren sollte nicht von der Existenz oder bestimmten Inhalten des Kommentar-Feldes abhängen.

Beispiele

Hier einige Beispiele für den Inhalt von IPv6Lookup-Paketen in einfacher Textform:

?host/pc-2n00//

Eine Query nach einem Host mit dem Namen “pc-2n00”. Bei Queries ist das Adressen- und das Kommentar-Feld leer.

+host/pc-2n00/.

Ein Report eines Hosts mit dem Namen “pc-2n00”. Die Adresse ist implizit, da im Adressen-Feld das ’.’-Zeichen steht. Das Kommentar-Feld fehlt.

+host/pc-2n00/./This is the description

Ein Report mit impliziter Adresse und ausgefülltem Kommentar-Feld.

+host/pc-2n00/3ffe:400:200:0:260:8ff:fe29:5b85/This is the description

Ein Report mit expliziter Adresse und ausgefülltem Kommentar-Feld.

+host/pc-2n00.6lab.inf.fh-rhein-sieg.de/./6Bone-Router

Ein Report mit FQDN, impliziter Adresse und Kommentar-Feld.

-host/pc-2n00//

Ein negativer Report. Bei negativen Reports kann das Adressfeld wie bei einer Query leer sein oder ganz fehlen.

+lpd/Laserdrucker/./Laserdrucker in Raum C016

Dies ist ein Beispiel für eine weitere mögliche Anwendung des IPv6Lookup-Protokolls. Hier wird die Verfügbarkeit eines LPD-Drucker-Servers angekündigt. Der Name des Dienstes ist deskriptiv, es wird nicht der Name des Rechners verwendet, auf dem der LPD-Server läuft (obwohl dies auch möglich wäre). Der Dienst erhält einen eigenen Namen, der mit der IPv6-Adresse des Server-Rechners assoziiert wird.

+host/pc-2n00/3ffe:400:200:0:260:8ff:fe29:5b85/6bone-Router

+lpd/Deskjet/3ffe:400:200:0:260:8ff:fe29:5b85/Deskjet Farbtintendrucker in C015

Dies ist ein Beispiel für ein Paket, in dem mehrere Reports enthalten sind. Wenn ein Host verschiedene Dienste ankündigt, ist es effizient, wenn dies in einem einzigen Paket geschehen kann. Hier wird die Existenz des Hosts selbst und der Druckerdienst angekündigt. Beide Dienste haben einen unterschiedlichen Namen, wobei der Druckerdienst auch den Namen des Hosts haben könnte. Es ist jedoch benutzerfreundlicher, deskriptive Namen zu verwenden.

Zur Klarstellung folgt nun noch die Darstellung einiger Pakete als Byte-Sequenz. So wird der Report

+host/pc-2n00/./6bone-Router

als Byte-Sequenz wie folgt dargestellt:

Byte-Nummer	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	+	h	o	s	t	/	p	c	-	2	n	0	0	/	.	/
Hexadezimal	2b	68	6f	73	74	2f	70	63	2d	32	6e	30	30	2f	2e	2f
Dezimal	43	104	111	115	116	47	112	99	45	50	110	48	48	47	46	47
Byte-Nummer	16	17	18	19	20	21	22	23	24	25	26	27	28			
Zeichen	6	b	o	n	e	-	R	o	u	t	e	r	\0			
Hexadezimal	36	62	6f	6e	65	2d	52	6f	75	74	65	72	0			
Dezimal	54	98	111	110	101	45	82	111	117	116	101	114	0			

Nun ein Beispiel, bei dem zwei Reports in einem Paket enthalten sind. Die Text-Darstellung des Pakets ist folgende:

+host/pc-2n01/./
+lpd/Deskjet/./Tintendrucker

Nun die Darstellung als Byte-Sequenz:

Byte-Nummer	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeichen	+	h	o	s	t	/	p	c	-	2	n	0	1	/	.	/
Hexadezimal	2b	68	6f	73	74	2f	70	63	2d	32	6e	30	31	2f	2e	2f
Dezimal	43	104	111	115	116	47	112	99	45	50	110	48	49	47	46	47
Byte-Nummer	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Zeichen	\n	+	l	p	d	/	D	e	s	k	j	e	t	/	.	/
Hexadezimal	a	2b	6c	70	64	2f	44	65	73	6b	6a	65	74	2f	2e	2f
Dezimal	10	43	108	112	100	47	68	101	115	107	106	101	116	47	46	47
Byte-Nummer	32	33	34	35	36	37	38	39	40	41	42	43	44	45		
Zeichen	T	i	n	t	e	n	d	r	u	c	k	e	r	\0		
Hexadezimal	54	69	6e	74	65	6e	64	72	75	63	6b	65	72	0		
Dezimal	84	105	110	116	101	110	100	114	117	99	107	101	114	0		

Man beachte die Trennung der beiden Reports mit einem Zeilenende-Zeichen (Byte-Nummer 16) und den Abschluss des gesamten Pakets mit einem Null-Byte.

4.4 Ablauf des Protokolls

Nach der Definition des Paketformats können nun die relevanten Informationen übertragen werden. Der Ablauf des Protokolls, welche Pakete in welcher Situation ausgetauscht werden sollen und wie sich die Protokollteilnehmer verhalten sollen, wird in diesem Abschnitt beschrieben. Gemäß den Anforderungen hat jeder Host, der an der Kommunikation über das IPv6Lookup-Protokoll teilnimmt, die Aufgabe, eine möglichst vollständige Liste seiner Nachbarn und der von diesen angebotenen Dienste haben (die *Nachbarschaftsliste*). Außerdem soll möglichst wenig Netzverkehr erzeugt werden. Dies wird durch eine ständig vorgehaltene Liste der Nachbarschaft begünstigt. Im Normalfall soll die Anfrage einer Anwendung nicht in einer Anfrage auf dem Netz resultieren, sondern aus dieser Liste beantwortet werden. Da man die Vollständigkeit der Liste nicht garantieren kann, ist natürlich auch vorgesehen, ein Query-Paket zu senden, um die Anfrage einer Anwendung zu beantworten. Query-Pakete werden via Broadcast an alle anderen Hosts gesendet. Unter IPv6 wird das Senden per Broadcast nicht mehr unterstützt, auf technischer Ebene wird daher stattdessen ein Multicast an die spezielle Adresse *ff02::1* verwendet (gemäß [RFC2373] heißt diese Adresse “All Nodes, Local Scope”), die alle lokalen Knoten anspricht. Dies ist äquivalent mit dem klassischen Broadcast, daher wird hier auch weiterhin die Bezeichnung “Broadcast” verwendet. Auch Report-Pakete werden immer per Broadcast an alle Nachbarn gesendet, egal, ob der Report automatisch erzeugt wurde, oder als Antwort auf eine Anfrage gesendet wurde. Dadurch erhalten bei einer Anfrage auch alle anderen Nachbarn die nachgefragte Adressinformation. Dieses Vorgehen erhöht bei allen Hosts die Wahrscheinlichkeit, dass Anfragen aus der Nachbarschaftsliste beantwortet werden können, was insgesamt den Netzverkehr verringert.

Zunächst soll nun für einen einzelnen Host im Überblick beschrieben werden, wie welche Protokollelemente im zeitlichen Ablauf vom Systemstart bis zum Herunterfahren verwendet werden. Im Anschluss wird dann die Semantik der einzelnen Abläufe des Protokolls im Detail beschrieben.

1. Beim Start des Systems wird ein Report-Paket mit allen angebotenen Diensten gesendet. Falls mehrere Dienste angekündigt werden sollen, können dazu auch mehrere Report-Pakete verwendet werden, mit beliebig vielen Reports pro Paket (oder auch nur einem Report pro Paket).

2. Beim Start des Systems wird ein Query-All-Paket gesendet.

3. Wenn ein Report-Paket eintrifft, wird es verarbeitet und die entsprechenden Einträge in die Nachbarschaftsliste werden vorgenommen.

4. Wenn ein Query-Paket eintrifft, dessen Dienstespezifikation auf den Empfänger zutrifft, wird ein Report-Paket gesendet. Dieses Report-Paket muss einen zu der Query passenden Report enthalten. Es kann auch noch andere Reports enthalten.

5. Bei entsprechendem Bedarf kann selbständig ein Query-Paket gesendet werden. Dies wird durch die Anfrage einer Anwendung ausgelöst, die nicht durch die Liste der bereits bekannten Nachbarn beantwortet werden kann. Nach dem Senden der Query wird ein Timer gestartet (ein Query-Timeout)

6. Tritt ein Query-Timeout ein, kann der vorige Schritt wiederholt werden. Die Anzahl der Wiederholungen ist begrenzt. Wird die Grenze erreicht, sollte die Anfrage der Anwendung mit einem Fehlerstatus beantwortet werden.

7. Periodisch werden selbständig Report-Pakete wie in Schritt 1 gesendet.

8. Schritte 3 bis 7 werden solange ausgeführt, bis das System heruntergefahren wird.

9. Beim Herunterfahren des Systems wird für jeden von diesem Host angebotenen Dienst ein negativer Report gesendet. Dies kann in einem oder mehreren Paketen geschehen.

Anmerkungen

Reports enthalten immer die eigene Adresse (die Adresse des Senders), nur der Host, der den Dienst anbietet, darf einen entsprechenden Report senden.

Reports, die als Antwort auf ein Query-All-Paket gesendet werden, müssen zeitlich verzögert gesendet werden.

Query Der anfragende Host sendet ein Query-Paket mit dem gesuchten Dienst und wartet auf das Eintreffen eines entsprechenden Reports. Reports zu anderen Namen müssen in dieser Zeit auch verarbeitet werden. Trifft nach einer bestimmten Zeitspanne kein Report ein, darf die Query bis zu drei mal wiederholt werden. Der Abstand zwischen den Wiederholungen muss mindestens eine halbe Sekunde betragen.

Empfängt ein Host eine Query mit auf sich passender Dienste-Spezifikation, antwortet er darauf mit einem Report. Das Report-Paket kann dabei mehrere Reports enthalten, von denen nicht alle zu der vorangegangene Query passen müssen. Wurde z.B. eine Query nach der Dienstklasse "host" gesendet, kann das als Antwort gesendete Report-Paket einen entsprechenden Report mit "host"-Dienstklasse und weitere Reports mit anderen Dienstklassen (und Namen) enthalten.

Query-All Der anfragende Host sendet ein Query-Paket mit dem '*'-Zeichen im Namensfeld. Darauf eintreffende Reports werden normal verarbeitet. Ein Query-All darf *nicht* wiederholt werden, falls kein Report eintrifft.

Um das Auftreten von Spitzen bei der Netzwerklast zu vermeiden, muss ein Report, der als Antwort auf ein Query-All-Paket gesendet wird, mit einer zufälligen Zeitverzögerung innerhalb einer Spanne von null bis drei Sekunden gesendet werden.

Um die Liste der Nachbarn möglichst schnell zur Verfügung zu haben, sollte jeder Host beim Systemstart ein Query-All-Paket senden.

Report Generell darf ein Host jederzeit einen Report senden, nicht nur als Antwort auf eine Query. Auch müssen alle Hosts jederzeit beliebige Reports empfangen und verarbeiten können, egal ob entsprechende Anfragen ausstehen oder nicht. Ein Host ist jedoch verpflichtet, auf eine empfangene Query, deren Dienstespezifikation zu ihm passt, mit einem Report zu antworten.

Ist ein Dienst nicht mehr verfügbar, muss der jeweilige Host selbsttätig einen negativen Report senden. Dieser negative Report kann auch in einem Report-Paket mit anderen (negativen oder positiven) Reports enthalten sein.

Wird ein negativer Report empfangen, muss der jeweilige Dienst entweder aus der Nachbarschaftsliste gestrichen werden oder aber als inaktiv markiert werden.

Periodische Reports Ein aktiver Host sollte in gewissen Zeitabständen automatisch einen Report senden. Die Periodendauer sollte nicht zu kurz sein, um nicht zu viel Netzverkehr zu erzeugen. Empfohlen wird eine Periodendauer von 20 Minuten.

Inaktive Dienste Hosts, die eine gewisse Zeit keinen Report von sich gegeben haben, gelten als inaktiv. Dieser Mechanismus ist notwendig, damit Hosts, die auf "gewaltsame" Weise vom Netz getrennt wurden, ohne sich mit einem negativen Report abzumelden, nicht für immer in der Nachbarschaftsliste verbleiben. Hosts können auch durch einen negativen Report als inaktiv markiert werden. Ein inaktiver Dienst/Host kann noch eine Zeit lang in der Liste geführt werden und auch dann noch zur Namensauflösung herangezogen werden. Nach einer bestimmten Zeitspanne, die mindestens zweimal so groß wie die Periodendauer der weiter oben beschriebenen periodischen Reports sein sollte, müssen inaktive Dienste aus der Nachbarschaftsliste entfernt werden.

Dieser Mechanismus soll nicht als Status-Monitor für die verfügbaren Dienste dienen, die Information, ob ein Dienst aktiv oder inaktiv ist, muss nicht akkurat sein. Er soll nur sicherstellen, dass sich die Nachbarschaftsliste nicht im Laufe der Zeit mit einer immer größer werdenden Anzahl veralteten Einträgen füllt.

4.5 Mögliche Erweiterungen

Dienstelokalisierung

Bereits mehrfach angesprochen wurde die Möglichkeit, das IPv6Lookup-Protokoll dazu zu verwenden, um Dienste im lokalen Netz zu lokalisieren. Diese Möglichkeit ist in der hier ausgeführten Protokollspezifikation bereits enthalten, es fehlen allerdings konkrete Definitionen von Service-Klassen außer "host". Es wäre zu prüfen, ob eine eigene Liste mit gültigen Dienstklassen erstellt werden soll, oder eine Abbildung von anderweitig definierten Service-Bezeichnungen

und -zuordnungen vorgenommen werden soll. Beispielsweise wäre es möglich, eine direkte Abbildung mit Hilfe der Einträge der Datei */etc/services* vorzunehmen. Die in dieser Datei enthaltenen Informationen definieren zu in Textform dargestellten Service-Namen die jeweils zu verwendenden Port-Nummern. Der Service-Name ist dabei ein (kurzes) Schlüsselwort, das direkt für das Service-Klassen-Feld des IPv6Lookup-Protokolls verwendet werden könnte. Demnach würde ein LPD-Drucker-Service die Dienstklasse "printer" erhalten, ein NFS-Server die Klasse "nfs" und ein Webserver (falls es sinnvoll ist, einen Webserver über IPv6Lookup zu lokalisieren) die Klasse "http".

Diese Art der Benutzung des IPv6Lookup-Protokolls zur Lokalisierung von Diensten im Netz ist allerdings nur sinnvoll, wenn sie in speziell dafür entwickelten Anwendungen, möglichst mit entsprechender grafischer Benutzeroberfläche, verwendet wird. So könnte z.B. eine grafische Anwendung entwickelt werden, die alle im lokalen Netz verfügbaren Drucker und/oder NFS-Server anzeigt. Wählt der Benutzer dann z.B. einen NFS-Server aus, kann das entsprechende *mount*-Kommando mit der numerischen Adresse des Servers aufgerufen werden, da der Dienstname nicht gleich dem Hostnamen sein muss und daher nicht verwendet werden kann. Durch den Aufruf mit numerischen Adressen wird eine aufwendige und die Flexibilität einschränkende Modifikation des *mount*-Kommandos in Bezug auf die Namensauflösung vermieden.

Individuelle Port-Nummern

Das hier spezifizierte Protokoll erlaubt nicht, für einen angebotenen Dienst eine individuelle Port-Nummer festzulegen. Es wird davon ausgegangen, dass Server und Client die verwendete Port-Nummer für eine bestimmte Service-Klasse im voraus kennen. Dies ist für die meisten Anwendungsfälle sicherlich adäquat. Trotzdem kann es in bestimmten Anwendungsfällen wünschenswert sein, nicht standardisierte Portnummern zu verwenden. Um dies mit IPv6Lookup zu unterstützen, könnte man die Port-Nummer in die Service-Klasse einbauen und mit einem geeigneten Trennzeichen von dem Service-Schlüsselwort trennen (z.B. "printer-5400" für Service-Klasse "printer" und Port-Nummer 5400). Anwendungen, die diese Erweiterung nicht unterstützen, sehen dann nur eine weitere, für sie unbekannte Service-Klasse.

Anwendungsspezifische Informationen

Als andere Erweiterungsmöglichkeit könnten Anwendungen zusätzliche Informationen im Kommentar-Feld unterbringen. So könnte ein Drucker-Server Informationen über den Druckertyp in den Kommentar einbauen, möglichst so, dass der Kommentar für den Benutzer lesbar bleibt. Eine Software zur Einrichtung von Netzwerk-Druckern, die das IPv6Lookup-Protokoll benutzt, könnte diese Informationen aus dem Kommentar-Feld dann zur automatischen Einstellung des benötigten Druckertreibers verwenden. Fehlt die Zusatzinformation, muss diese Einstellung manuell vorgenommen werden können, da das Vorhandensein des Kommentar-Feldes von der Protokollspezifikation nicht garantiert wird. Auch muss das Kommentar-Feld vom Benutzer weiterhin frei wählbar sein, die anwendungsspezifischen Informationen dürfen den vom

Benutzer gewählten Text nur modifizieren (d.h. durch Anhängen oder Voranstellen), nicht ersetzen.

Unterstützung für größere Netze

In der hier vorgestellten Fassung ist das IPv6Lookup-Protokoll für kleinere Netzwerke gedacht. Eine weitere Überlegung wäre, das IPv6Lookup-Konzept so auszubauen, dass auch größere Netzwerke mit mehreren (oder vielen) hundert Netzwerkknoten unterstützt werden können. Für die Unterstützung von mehreren Netzsegmenten (verschiedene Netzadressen auf IP-Ebene) könnte man Multicast-Gruppen mit "*site-local scope*" verwenden oder eine Proxy-Funktionalität zwischen verschiedenen Netzsegmenten implementieren. Dabei sollten Queries, wenn möglich, von den Proxies beantwortet werden. Ein Query-All-Paket sollte nicht Reports von allen Hosts in allen Netzen auslösen, sondern der Proxy sollte in diesem Fall eine vollständige Nachbarschaftsliste in einem Report senden. Obwohl diese Maßnahmen technisch gesehen relativ einfach zu implementieren wären, bleibt die Frage, ob es bei einer derartigen Größe des Netzes noch sinnvoll ist, einen dezentralen Namensdienst einzusetzten. Vielmehr scheint es in dieser Situation angebracht zu sein, einen im Kontext des Netzmanagements geplanten und zentral administrierten Namensdienst wie DNS zu verwenden.

Kapitel 5

Planung der Integration ins Betriebssystem

5.1 Integration von Namensdiensten unter Unix

In diesem Abschnitt soll untersucht werden, auf welche Weise Informationsdienste in die Unix-Programmier- und Systemumgebung eingebettet sind. Daraus lässt sich dann ableiten, welche Anforderungen und Notwendigkeiten sich für die Implementierung ergeben.

Es gibt verschiedene Arten von Informationen, die unter Unix traditionell in einfachen Textdateien gespeichert sind und deren Format in den verschiedenen Unix-Standards festgeschrieben ist. Beispiele sind Hostnamen und IP-Adressen in der Datei */etc/hosts* oder Benutzerinformationen in den Dateien */etc/passwd* und */etc/group*. Man kann diese Informationen zwar direkt aus den Dateien lesen, aber es gibt auch in der C-Bibliothek eine Reihe von Funktionen, die diese Aufgabe übernehmen und geeignete Datenstrukturen zurückliefern. Zum Zugriff auf die Hostnamen-Informationen existieren z.B. die Funktionen *gethostbyname()*/*gethostbyaddr()*, für die Benutzerinformationen gibt es die Funktionen *getpwent()* und *getgroupent()*.

Es folgt eine Auflistung der Informationsarten sowie der beteiligten Dateien und C-Funktionen.

Passwort- und Benutzerinformationen /etc/passwd, /etc/master.passwd (BSD) oder /etc/shadow (SysVR4 Unix, Linux)

getpwent(), *getpwnam()*, *getpwuid()*

Gruppeninformationen /etc/group

getgrent(), *getgrnam()*, *getgrgid()*

Hostnamen und Adressen /etc/hosts

gethostbyname(), *gethostbyaddr()*, *gethostbyname2()*, *getipnodebyname()*, *getipnodebyaddr()*, *getaddrinfo()*, *getnameinfo()*

Netzadressen und -namen /etc/networks
getnetent(), *getnetbyname()*, *getnetbyaddr()*

Netgroups /etc/netgroup
getnetgrent(), *innetgr()*

Portnummern /etc/services
getservent(), *getservbyname()*, *getservbyport()*

Protokollnummern /etc/protocols
getprotoent(), *getprotobyname()*, *getprotobynumber()*

Ethernetadressen /etc/ethers
ether_notohost(), *ether_hostton()*

Druckerkonfiguration /etc/printcap
(NIS-Abfragen für Druckerkonfiguration werden *lpd*-intern unterstützt)

Mail-Aliasnamen /etc/aliases, /etc/mail/aliases
(NIS-Abfragen für Mail-Alias-Namen wird *sendmail*-intern unterstützt)

SunRPC-Nummern und Namen /etc/rpc
getrpcent(), *getrpcbyname()*, *getrpcbynumber()*

5.2 Entwicklung der Namensdienst-Architekturen unter Unix

Ursprünglich gab es unter Unix nur diese Textdateien als Informationsquelle. Mit der Entwicklung des Internet, die zu großen Teilen parallel zur Weiterentwicklung von Unix bzw. Berkeley-Unix (BSD) ablief, kam dann für Hostnamen und IP-Adressen als Informationsquelle der Internet-Namensdienst DNS hinzu. *BIND* (*Berkeley Internet Name Dæmon*), die Standard-Software für DNS, enthielt (und enthält noch immer) eine Bibliothek zum Abfragen eines Nameservers, die als Resolver-Library bezeichnet wird (*libresolv*). Dazu gehört z.B. die Funktion *res_query()*.

DNS ist allgemein als eine verteilte Datenbank konzipiert, in der man als Strings codierte Datensätze über hierarchisch organisierte Namen abfragen kann. Die hauptsächliche Verwendung ist bisher die Namensauflösung, obwohl auch andere Verwendungsformen existieren (siehe das *Hesiod*-System). Die *gethostbyname()*-Funktion und deren Verwandten wurden also modifiziert, um die mit Hilfe der Resolver-Library auch DNS zur Namensauflösung verwenden zu können. Die Resolver-Library hält sogar eine eigene vollwertige *gethostbyname()*-Funktion bereit.

Relativ bald kam eine weitere Informationsquelle hinzu, die sowohl von der Konzeption als auch vom praktischen Einsatz nicht nur als reiner Namensdienst, sondern als vielseitiger Informationsdienst vorgesehen war. *NIS*, der *Network Information Service*[1], entwickelt von der Firma SUN Microsystems, sollte eben die Inhalte der oben beschriebenen Textdateien über das Netz zur Verfügung stellen. NIS organisiert die Informationen in Tabellen, in denen als String codierte Datensätze über Schlüssel abgefragt werden können. Da SUN die Spezifikationen für NIS offenlegte, entwickelte sich NIS schnell als der Standard-Netzwerk-Informationsdienst für Unix, der bei allen kommerziellen Unix-Derivaten sowie bei Open-Source-Unix-Varianten wie Linux oder BSD unterstützt wird. Dementsprechend wurden die oben genannten C-Bibliotheksfunktionen modifiziert, um NIS-Tabellen anstelle der Textdateien abzufragen. In den meisten Fällen wurde die Implementierung so gestaltet, dass das Vorhandensein eines speziellen Eintrags in der entsprechenden Textdatei (ein '+'-Zeichen) signalisiert, dass (zusätzlich) die entsprechende NIS-Tabelle abgefragt werden soll. Die NIS-Tabellen stellen so eine Erweiterung der Textdatei dar, die ähnlich wie eine Include-Datei in einem C-Programm funktioniert.

Bei der Datei */etc/hosts* war dieses Verfahren nicht ausreichend. Die Resolver-Library wurde ebenfalls um NIS-Funktionalität ergänzt, womit jetzt zur Namensauflösung drei verschiedene Informationsquellen zur Verfügung standen (Dateien, DNS, NIS). Also wurde für die Resolver-Library eine Konfigurationsdatei eingeführt (*/etc/host.conf*), in der (unter anderem) festgelegt werden kann, welche Informationsquellen benutzt und in welcher Reihenfolge sie abgefragt werden sollen[2].

Bei allen anderen Informationsarten gab es nur Dateien und NIS, also genügt die Steuerungsmöglichkeit durch die '+'-Einträge.

Unter System V Release 4 (z.B. SunOS 5.x) wurde dann eine modulare Informationsdienst- Architektur in die Systembibliothek eingeführt, die als Name Service Switch (*nsswitch*) bezeichnet wurde. Hier existiert eine Konfigurationsdatei /etc/nsswitch.conf, die für jede Art von Information festlegt, aus welchen Quellen und in welcher Reihenfolge sie abgefragt werden soll. Damit werden die '+'-Einträge für NIS überflüssig, ebenso wie die Datei */etc/host.conf*. Für die Passwort-Informationen geht dadurch allerdings Funktionalität verloren, da mit den '+'-Einträgen hier selektiv Daten aus der NIS-Tabelle übernommen werden konnten. Also gibt es speziell hierfür einen Kompatibilitätsmodus, bei dem doch noch die '+'-Einträge in */etc/passwd* gelten (und dieser Modus wird auch sehr oft benutzt).

[1]Der ursprüngliche Name war “Yellow Pages” oder YP. Er wurde später in “NIS” geändert, um Rechtsstreitigkeiten mit British Telecom plc. aus dem Wege zu gehen.

[2]Die genaue Syntax dieser Konfigurationsdatei ist bei verschiedenen Systemen unterschiedlich.

Das NSSwitch-System besitzt auch die Fähigkeit, Module zum Zugriff auf Informationssysteme dynamisch zu laden. Diese Module werden als *shared objects/shared libraries* abgelegt und entsprechend der Konfigurationsdatei beim Programmstart oder bei der Initialisierung der NSSwitch-Funktionen dynamisch eingebunden. Damit besteht die Möglichkeit, beliebige neue Informationssysteme nachträglich einzubinden, ohne Programme neu compilieren zu müssen. Unter Solaris/SunOS 5.x müssen (zumindest in früheren Versionen, 5.0-5.2) alle Informationsdienste als NSSwitch-Module dynamisch geladen werden, was den Nachteil hat, dass man Programme, die *nsswitch* benutzen sollten, nicht mehr statisch linken kann, da für die NSSwitch-Module die Funktionalität des dynamischen Linkers benötigt wird. Unter Solaris befindet sich das ganze NSSwitch-System zusammen mit anderen Netzwerk-Funktionen in einer separaten Bibliothek (*libnsl*). Neuere Solaris-Versionen bieten auch eine statische Version der NSL-Library, um rein statisch gelinkte Programme zu ermöglichen.

Unter Linux existiert ebenfalls eine Nachimplementierung der NSSwitch-Library. Diese ist aber in der Standard-Systembibliothek (*libc* bzw. *glibc* unter Linux) integriert. Es existieren statische Module, die dynamisch konfiguriert werden können, sowie die Möglichkeit, dynamisch Module nachzuladen (sofern der dynamische Linker vorhanden ist). Im NetBSD-Betriebssystem existiert ebenfalls eine Reimplementierung des NSSwitch-Systems (mit kompatiblen Konfigurationsfiles), das zumindest derzeit nur statisch eingebundene Module, die dynamisch konfiguriert werden können, zur Verfügung stellt. Es existieren also in der Unix-Welt verschiedene Varianten und Interpretationen des NSSwitch-Systems. Ein Problem des NSSwitch-Systems besteht darin, dass die Resolver-Library der BIND-Distribution, die immerhin die Referenzimplementation für die Namensauflösungs-Funktionen darstellt, keine NSSwitch-Module bereitstellt. Außerdem unterstützen die *gethostbyname()*-Funktionen aus der Resolver-Library nicht die Verwendung des NSSwitch-Systems. Stattdessen enthält die BIND-Distribution ein eigenes, in der Funktion ähnliches System, welches *IRS* (*Information Retrieval System*) genannt wird und wie *nsswitch* eine dynamische Konfiguration der verschiedenen Informationsquellen erlaubt. Dynamisches Laden von *shared objects* ist hier aber nicht vorgesehen (vermutlich aus Gründen der Portabilität). Obwohl IRS in der Funktionalität *nsswitch* sehr ähnlich ist, ist dem Autor kein System bekannt, auf dem IRS benutzt würde. IRS ist wohl eher als Studie, als "Proof-of-Concept" zu betrachten. Der De-Facto-Standard unter Unix ist also das NSSwitch-System. Die neueste Version der BIND-Distribution (BIND 9), die größtenteils neu implementiert wurde, enthält dementsprechend IRS nicht mehr.

Der letzte Schritt der aktuellen Entwicklung besteht darin, die Funktionalität zum Abfragen der Informationsdienste auszulagern und einen Cache zu verwenden. Die NSSwitch-Library wird so erweitert, dass sie eine lokale Verbindung zu einem speziellen Dæmon aufbaut und die gesuchte Information dort anfragt. Der Dæmon nutzt nun seinerseits das NSSwitch-System, um die Informationen zu erlangen und sie an die Anwendung weiterzuleiten. Zusätzlich dient dieser Dæmon auch als Cache. Dementsprechend wird er als *nscd* (*Name Service Cache Dæmon*) bezeichnet. Ein derartiges System wird z.B. in SunOS 5.7 und unter Linux mit aktueller Glibc-Version verwendet. Die genaue Funktionsweise ist nur für den *nscd* unter Linux bekannt, da hier der komplette Sourcecode verfügbar ist. Die Verwendung des *nscd* unter Linux ist optional, d.h. die Systembibliothek enthält sowohl das bisherige NSSwitch-System als auch die Erweiterung,

die den *nscd* nutzen kann. Auch die neuesten BIND-Distributionen unterhalb Version 9 enthalten ein ähnliches System, das *IRP* (*Information Retrieval Protocol*) genannt wird und einen Dæmon namens *irpd* (*IRP Dæmon*) verwendet. Es setzt auf das bereits erwähnte IRS auf und verwendet *irpd* als Agent und als Cache für die Informationsdienste. Wie IRS wird jedoch IRP von keinem dem Autor bekannten System benutzt.

Ein weiteres System dieser Art wurde bereits Anfang der 90er Jahre, also deutlich vor Solaris oder Linux, in NeXTSTEP implementiert. Das dort verwendete Konzept war deutlich radikaler als das NSSwitch-System. Alle mit Informationsdiensten zusammenhängenden Funktionen benutzten einen Dæmon als Agenten, der als *lookupd* bezeichnet wurde. Dieser Dæmon verwendete dann in der Regel das ebenfalls von NeXT entwickelte netzwerkbasierte Informationssystem *NetInfo*, um die gesuchten Informationen zu erlangen, und zusätzlich DNS, falls Namensauflösung gefordert war. Der Zugriff auf die klassischen Konfigurationsdateien war nicht vorgesehen, außer für den Fall, dass das *lookupd*-System nicht lief, z.B. für den Systemstart. In der heutigen, in MacOS X und Darwin zu findenden Version von *lookupd* können verschiedene Informationssysteme wie NetInfo, NIS, DNS, Dateien oder LDAP verwendet werden. Insgesamt übertrifft dieses Konzept das NSSwitch-System deutlich an Eleganz, die Funktionalität ist mindestens ebenbürtig (obwohl das dynamische Nachladen von Modulen noch nicht vorgesehen ist). Mit der Kombination aus *lookupd*/NetInfo ist ein System auf einen Schlag in der Lage, alle Konfigurationsinformationen in einer gut gemanagten Weise über das Netzwerk zu beziehen. Insgesamt ist dieses System, auch wenn es ausschließlich unter MacOS X verfügbar ist, als richtungweisend anzusehen.

5.3 Momentane Situation in FreeBSD 4.0

FreeBSD enthält in der Version 4.0 kein NSSwitch-System. Zusätzlich zu den klassischen Konfigurationsdateien wird von allen in Abschnitt 5.1 beschriebenen Funktionen die Verwendung von NIS unterstützt. Dies wird, bis auf einige Ausnahmen[3], durch die entsprechenden NIS-Einträge in den Dateien aktiviert. Die Funktionen, die mit Namensauflösung und IP-Adressen befasst sind, unterstützen natürlich auch DNS, gesteuert durch die Datei */etc/host.conf*.

Um einen neuen Namens- und Informationsdienst ins System zu integrieren, muss also in jedem Fall die C-Bibliothek direkt modifiziert werden, da ohne das modulare NSSwitch-System ein Bereitstellen von externen Namensdienst-Modulen nicht möglich ist. Da für das IPv6Lookup-Protokoll ohnehin ein eigener Dæmon laufen muss, mit dem für eine Namensauflösung kommuniziert werden muss, liegt es nahe, den Schritt ganz zu vollziehen und ein System wie *nsswitch* und dem Dæmon *nscd* zu implementieren. Da die Linux-Version von *nsswitch* als Open Source Software verfügbar ist, scheint es ebenso nahe zu liegen, diese Software für FreeBSD zu portieren. Leider steht diesem Vorhaben die Lizenz des Glibc-Pakets, in dem das NSSwitch-System enthalten ist, im Wege. Diese Software unterliegt, wie unter Linux generell üblich, der

[3] Einige Funktionen (insbesondere die getrpc...-Funktionen) unterstützen immer NIS, falls eine NIS-Domain aktiviert ist. Bei der *gethostbyname()*-Funktion und deren Verwandten wird die Verwendung von NIS über die */etc/host.conf*-Datei gesteuert.

GNU General Public License (*GPL*) bzw. der *GNU Library General Public License* (*LGPL*). Diese Lizenzen erfordern, dass bei der Integration von GPL/LGPL-lizenziertem Sourcecode in anderen Sourcecode dieser andere Sourcecode ebenfalls GPL/LGPL-lizenziert (oder äquivalent lizenziert) sein muss. GPL-Kritiker sprechen hier von einer virusartigen Lizenz bzw. dem GPL-Virus[4]. Die Sourcecodes für FreeBSD unterliegen aber der BSD-Lizenz, die derartige Einschränkungen nicht macht. In die teilweise religiös geführte Debatte über Fluch und Segen der GPL, welche Lizenz nun die einzig wahre Open-Source-Lizenz ist und welche freier ist als die andere, soll hier nicht eingestiegen werden. Es bleibt jedoch die Tatsache, dass die Integration von GPL-Code in BSD-lizenzierten Code nicht möglich oder zumindest sehr problematisch ist und von der BSD-Entwickler-Community generell nicht akzeptiert wird. Daher müsste man das NSSwitch-System, wollte man es für FreeBSD implementieren und eine Integration in den offiziellen FreeBSD-Sourcecode nicht ausschließen, neu implementieren[5].

Bei näherer Betrachtung weist das NSSwitch/*nscd*-System aber auch einige Defizite auf, die eine Reimplementierung für FreeBSD als nicht erstrebenswert scheinen lassen[6]: Das Protokoll, das zur Kommunikation zwischen den Anwendungen und *nscd* verwendet wird, benutzt eine binäre Codierung der Daten. Die binären Datenstrukturen, die die entsprechenden Funktionen (z.B. *gethostbyname()*) zurückliefern, werden praktisch direkt über den Socket gesendet. Dieses Verfahren ist zwar sehr effizient, aber extrem unflexibel. Neue Arten von Informationen oder eine Erweiterung der Informationen, die übertragen werden, sind nur durch Änderung des binären Protokolls möglich, was sofort Inkompatibilitäten mit vorhandenen Anwendungen hervorruft. Diese könnten höchstens durch das Einführen von verschiedenen Protokollversionen beseitigt werden, was den Programmieraufwand beträchtlich erhöht, wenn man volle Auf- und Abwärtskompatibilität sicherstellen möchte. Anders gesagt, ist der Abstraktionsgrad der übertragenen Informationen sehr niedrig (praktisch null). Wünschenswert wäre aber ein hoher Abstraktionsgrad, um möglichst flexibel Informationen übertragen zu können. Einen höheren Abstraktionsgrad bietet das bereits erwähnte IRS/IRP-System. Hier wird ein textbasiertes Protokoll verwendet, das flexibel erweiterbar ist. Allerdings ist die Syntax dieses Protokolls für jede Informationsart zwar ähnlich, aber separat und starr festgelegt. Es existiert keine allgemeingültige Syntax, die für alle Informationsarten gilt. Außerdem spricht der geringe Verbreitungsgrad (null) nicht für den Einsatz dieses Systems. In der nächsten Version von BIND (Version 9) wird IRP/IRS auch nicht mehr enthalten sein. Also scheint es angeraten zu sein, ein eigenes System mit den gewünschten Eigenschaften zu entwerfen, da der Arbeitsaufwand dafür nur unwesentlich höher ist und man in diesem Fall die angesprochenen Probleme der anderen Lösungen umgehen kann[7].

[4]Ein Beispiel für eine kritische Sicht der GPL-Lizenz ist unter der URL *http://www.daemonnews.org/199905/gpl.html* zu finden.

[5]Unter NetBSD ist ein NSSwitch-System bereits teilweise implementiert worden, die Funktionalität im Gegensatz zum bestehenden FreeBSD-Code ist aber zum gegenwärtigen Zeitpunkt nicht wesentlich anders und daher uninteressant.

[6]Die Betrachtungen beziehen sich nur auf die Linux/Glibc-Variante des *nsswitch/nscd*-Systems, über andere Systeme (z.B. Solaris) kann wegen des nicht zugänglichen Sourcecodes keine Aussage gemacht werden.

[7]Es bliebe noch die Integration des *lookupd*-Systems von MacOS X/Darwin. Der Sourcecode oder sonstige Spezifikationen waren allerdings zum Zeitpunkt der Erstellung dieser Arbeit nicht verfügbar.

Kapitel 6

Entwurf einer universellen Namensdienst-Architektur

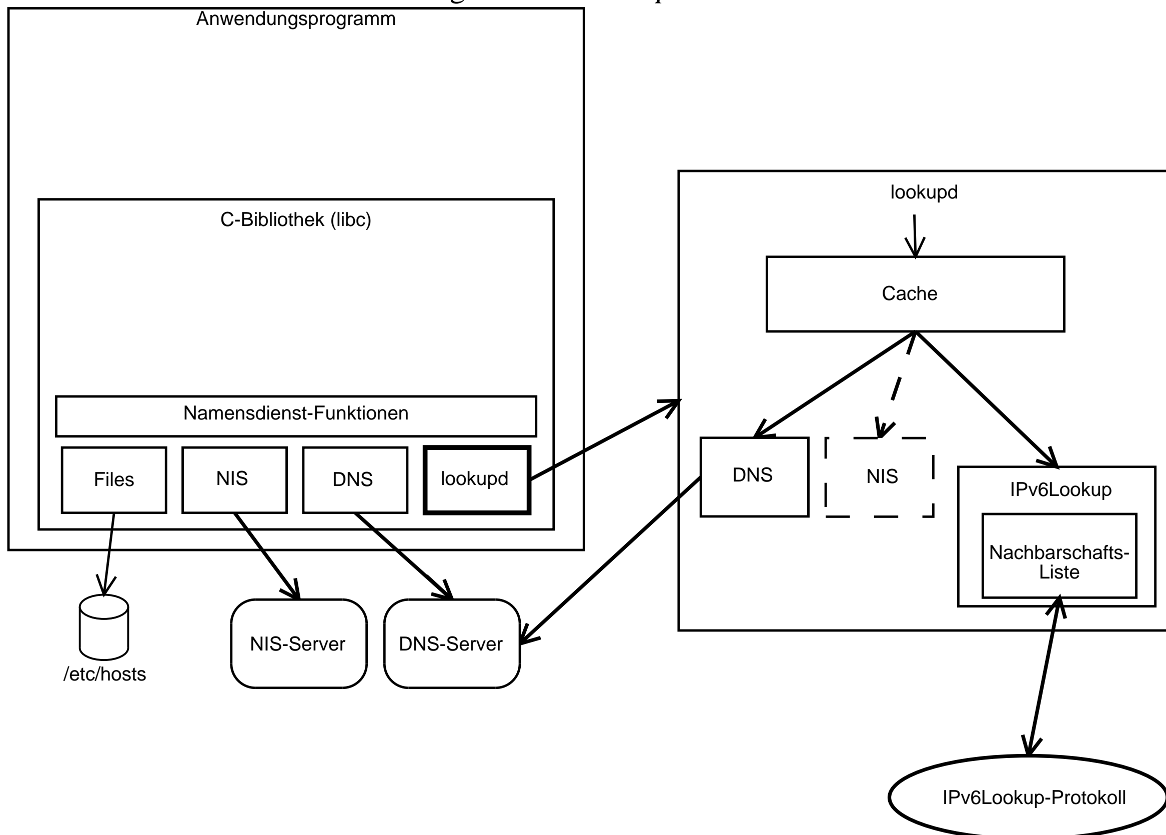
6.1 Die Architektur im Überblick

Die neue Namensdienst-Architektur verwendet einen Dæmon, der als Agent für Anwendungen die Aufgabe der Namensauflösung übernimmt. Der Dæmon soll darüber hinaus einen allgemeinen, systemweiten Informationsdienst zur Verfügung stellen. Da also einfach gesagt der Dæmon zum "Nachschauen" von Informationen dient, trägt er den Namen *lookupd*[1] (Dæmon-Prozesse tragen in Unix traditionell ein "d" am Ende ihres Namens). Es wird eine abstrakte Schnittstelle zur Namensauflösung und zur allgemeinen Abfrage von Informationen aufgebaut. Die Architektur integriert verschiedene Namens- und Informationsdienste, wie DNS, NIS oder auch das in dieser Arbeit entworfene IPv6Lookup-Protokoll. Außerdem werden verschiedene Arten von Informationen in die Architektur integriert, wie z.B. IP-Adressen und Passwort- und Benutzerinformationen. Im Gegensatz zu bisherigen Architekturen werden alle Informationsarten auf die gleiche Weise abgefragt und übertragen. Dabei ist es für die Anwendung völlig transparent, aus welcher Quelle die Information stammt, das heißt, mit welchem Namensdienst, von welchen Server, über welches Protokoll die Information erlangt wurde, ist für die Anwendung unsichtbar. Anwendungen kommunizieren mit dem Dæmon über einen lokalen Socket und verwenden dazu ein einfaches Protokoll (das *lookupd*-Protokoll), das weiter unten detailliert beschrieben wird. Anwendungen müssen nur noch das *lookupd*-Protokoll implementieren, der Programmcode zum Zugriff auf die verschiedenen tatsächlichen Namens- und Informationsdienste befindet sich nur noch im Dæmon. Das Protokoll ist textbasiert und kann Objekte mit beliebigen Attributen, codiert als Strings, übertragen. Es kann sehr flexibel erweitert werden, ohne dass mit Inkompatibilitäten mit älteren Programmen zu rechnen ist. Da die Anwendungen nichts von den unterliegenden Namensdiensten wissen, können beliebige neue Namensdienste integriert werden. Im Gegensatz zur *nsswitch*-Architektur können dann auch statisch gelinkte Programme, die

[1] Ein Daemon mit diesem Namen (und mit praktisch gleicher Funktion) existiert auch im NeXTSTEP/OpenSTEP/MacOS-X-Betriebssystem. Der hier verwendete Name wurde von dort "ausgeliehen".

keine Möglichkeit haben, dynamisch Module nachzuladen, die neuen Namensdienste nutzen. Der Dæmon dient außerdem noch als Cache, um die Namensauflösung generell zu beschleunigen (falls nicht nur auf lokale Dateien zugegriffen wird).

Abbildung 6.1: Die *lookupd*-Architektur



6.2 Das Informationsmodell

Die Informationen, die über den Dæmon abgefragt werden können, werden in *Tabellen* organisiert. In den Tabellen werden *Objekte* gespeichert, die wiederum *Attribute* besitzen. Jedes Objekt besitzt außerdem noch einen *Schlüssel*, über den es innerhalb einer Tabelle identifiziert wird. Dabei ist es möglich, dass mehrere Objekte den gleichen Schlüssel besitzen. Die Tabellen selbst werden über Namen (Strings) identifiziert. Der Namensraum der Tabellen ist flach, d.h. es gibt keine hierarchische Struktur. Dies ist so beabsichtigt, da an dieser Stelle nur die lokale Sicht des jeweiligen Rechners interessant ist und nicht die eventuell komplexere Organisation von Informationen, aus der die lokale Sicht zusammengesetzt wird. Attribute besitzen ebenfalls Namen, die wie die Werte der Attribute durch Strings repräsentiert werden. Um festzulegen, welche Attribute ein Objekt besitzen muss, werden *Objekt-Typen* definiert. In einer bestimmten

Tabelle sind dann typischerweise immer Objekte eines bestimmten Typs enthalten. Bei der Definition eines Typs können bestimmte Attribute optional sein, außerdem ist es explizit erlaubt, einem Objekt Attribute hinzuzufügen, die nicht in der Typdefinition enthalten sind.

Die Abfrage von Objekten aus den Tabellen geschieht, in dem ein Tabellenname und der gesuchte Schlüssel angegeben wird. Als Ergebnis erhält man alle Objekte, die den passenden Schlüssel haben. Außerdem erhält man zu jedem Objekt seinen Typ. Normalerweise ist dieser allerdings für eine Ergebnismenge immer gleich, da in einer Tabelle immer Objekte des gleichen Typs enthalten sind (zumindest in der hier beschriebenen Version).

Die hier beschriebene Version des Protokolls und der Software verwenden die Tabellen *hosts.byname* und *hosts.byaddr*, die jeweils Objekte des Typs *host* enthalten. Um ein weiteres Beispiel zu geben, wird im folgenden noch die Tabelle *passwd.byname* und *passwd.byuid* sowie der dazugehörige Typ *pwent* beschrieben[2].

Unter gewissen Umständen kann man als Suchschlüssel für eine Tabellenabfrage den speziellen String “*” verwenden. Als Ergebnismenge erhält man dann alle Objekte, die in der Tabelle enthalten sind. Allerdings ist nicht genau definiert, was “alle Objekte” sind. Die Tabelleninhalte können aus verschiedenen Quellen zusammengesetzt werden, verschiedene Informationsdienste können zusammengefasst werden. Bei vielen Informationsdiensten ist es unmöglich, eine komplette Zusammenstellung aller Einträge anzufertigen (z.B. DNS). In den meisten Fällen werden Informationen aus dem Informationsdienst nur dann eingeholt, wenn eine spezifische Suchanfrage an eine Tabelle gestellt wird. Im Normalfall existiert also keine Information darüber, welche Objekte in der Tabelle enthalten sind, es sei denn, man sucht (jeweils) über einen Schlüssel. Es kann jedoch Informationsdienste geben, die eine Liste aller deren Einträge haben, wie z.B. der IPv6Lookup-Dienst. In einem solchen Fall kann diese Liste durch Verwendung des “*”-Schlüssels abgefragt werden.

Definition des Typs *host*

Objekte des Typs *host* enthalten Informationen über einen Netzwerkknoten, einen Host, darunter hauptsächlich seinen Namen und die Adresse und die dazugehörige Adressfamilie. Hat ein Host mehrere Adressen, z.B. verschiedene IPv4-Adressen oder auch eine IPv4- und eine IPv6-Adresse, werden dafür mehrere Objekte angelegt. Hier nun die Definition der Attribute für ein Objekt mit dem Typ *host*.

Attribut	Beschreibung	
hostname	Hostname mit Domain-Anteil, falls vorhanden (FQDN)	
addr	Adresse in String-Repräsentation (s.u.)	
af	Adressfamilie (s.u.)	
descr	Ein beschreibender String für diesen Host	optional
state	Zustand des Hosts, 0 falls ausgeschaltet	optional

[2]Die hier verwendeten Tabellennamen sind an die Namen der entsprechenden NIS-Tabellen angelehnt.

Bemerkungen

- Die Adresse wird in der üblichen numerischen Form dargestellt, wie sie z.B. die Funktion *inet_ntop()* liefert. Die Darstellungsform muss eindeutig sein, da die String-Darstellung auch für die Rückwärtsauflösung verwendet wird.
- Die Adressfamilie wird als kurzes Schlüsselwort codiert, das den Konstantendefinitionen für Protokoll- und Adressfamilien unter Unix entspricht. So wird z.B. eine Adresse mit der Adressfamilie AF_INET mit dem String *inet* codiert, AF_INET6 als *inet6* und AF_APPLETALK als *appletalk*.

Definition des Typs *pwent*

Objekte vom Typ *pwent* stellen Benutzerinformationen dar, wie sie traditionell in der Datei */etc/passwd* gespeichert werden. Die Attribute und ihre Namen sind von der vom System definierten Datenstruktur *struct passwd* abgeleitet, wie sie z.B. von der Funktion *getpwent()* verwendet wird.

Attribut	Beschreibung	
name	Benutzername	
passwd	Das verschlüsselte Passwort (s.u.)	
uid	User-ID	
gid	Group-ID	
change	Zeitpunkt, zu dem das Passwort geändert werden muss (s.u.)	optional
class	Login-Klasse	optional
gecos	GECOS-Feld (s.u.)	
dir	Home-Directory	
shell	Die Shell des Benutzers	
expire	Zeitpunkt, zu dem der Account automatisch gesperrt wird (s.u.)	optional

Bemerkungen

- Das Passwort-Feld sollte nur dann ausgefüllt sein, wenn sich die anfragende Anwendung als privilegiert ausgewiesen hat.
- Die Felder *change* und *expire* werden in Sekunden seit dem 1. Januar 1970 (UTC) ausgedrückt. Es wird also eine String-Repräsentation des Standard-Typs *time_t* benutzt, wie er z.B. von der *time()*-Systemfunktion verwendet wird.
- Das GECOS-Feld enthält üblicherweise weitere Informationen über den Benutzer wie den realen Namen, Raumnummer, Telefonnummer etc. Der genaue Aufbau dieses Strings ist systemabhängig.

Definition der Tabelle *hosts.byname*

Die Tabelle *hosts.byname* enthält Objekte des Typs *host*. Als Schlüssel dient das Attribut *hostname*. Damit lassen sich in dieser Tabelle Objekte anhand des Hostnamens lokalisieren. Mehrfacheinträge sind möglich. Existieren für einen Host mehrere Adressen mit unterschiedlichen Adressfamilien, muss das Attribut *af* verwendet werden, um das Objekt mit der gesuchten Adressfamilie aus der Ergebnismenge herauszusuchen. Diese Tabelle kann den speziellen Schlüssel '*' unterstützen.

Definition der Tabelle *hosts.byaddr*

Die Tabelle *hosts.byaddr* enthält Objekte des Typs *host*. Als Schlüssel wird das Attribut *addr* benutzt. Diese Tabelle dient also dazu, um zu einer Adresse den passenden Hostnamen zu suchen. In der Regel wird es zu einem Schlüssel nur ein passendes Objekt geben, aber theoretisch ist es dennoch möglich, dass mehrere Objekte in der Ergebnismenge enthalten sind. Dies kann dann vorkommen, wenn es zu einem Host mehrere Adressen mit verschiedenen Adressfamilien gibt, deren String-Repräsentation gleich ist. Die Objekte unterscheiden sich dann nur in dem Attribut *af*. Daher sollte bei der Auswertung des Ergebnisses für jedes Objekt geprüft werden, ob das Attribut *af* den gewünschten Wert hat.

Definition der Tabelle *passwd.byname*

Die Tabellen *passwd.byname* und *passwd.byuid* dienen als Ersatz für die Datei */etc/passwd* und enthalten somit Objekte des Typs *pwent*. In der Tabelle *passwd.byname* dient der Benutzername (also der Wert des Attributs *name*) als Schlüssel. Diese Tabelle kann den speziellen Schlüssel '*' unterstützen.

Definition der Tabelle *passwd.byuid*

Diese Tabelle enthält ebenso Objekte des Typs *pwent*. Hier dient als Schlüssel die User-ID, der Wert des Attributs *uid*. Diese Tabelle dient dazu, die Informationen zu einem Benutzer anhand seiner User-ID schnell aufzufinden.

6.3 Das *lookupd*-Protokoll

Das *lookupd*-Protokoll ist verbindungsorientiert und textbasiert. Seine Hauptaufgabe ist es, Objekte und deren Attribute in Textform (als Strings) zu übertragen. Dazu können Anfragen gesendet werden, um bestimmte Objekte über einen Schlüssel anzufordern. Sowohl die Anfragen als auch die Objekte, die als Ergebnis zurückgeliefert werden, werden als eine Folge von Textzeilen übertragen. Als spätere Erweiterung ist vorgesehen, einen Befehl zum Eintragen eines

(oder mehrerer) Objekte in eine Tabelle zu senden (eine Schreiboperation). Dies geschieht analog zu einer Anfrage, mit dem Unterschied, dass kein Ergebnis zurückgesendet wird, sondern direkt eine Menge von Objekten als Parameter gesendet wird. Das *lookupd*-Protokoll läuft immer zwischen einem Client und einem Server ab. Der Client stellt Anfragen, auf die der Server mit einem Ergebnis (einer Objektmenge) antwortet. Der Client kann auch eine Schreiboperation an den Server senden. Dies sind die einzig möglichen Operationen (zumindest in der hier vorgestellten Version). Nachdem ein Client eine Verbindung zum Server hergestellt hat, kann er beliebig viele Operationen durchführen. Anschließend wird die Verbindung von Client beendet. Der Server beendet die Verbindung nur im Falle von Protokollfehlern (nicht bei fehlgeschlagenen Anfragen). Alle folgenden Protokollelemente bestehen aus einer oder mehreren Textzeilen, die jeweils von einem *LF*-Zeichen (ASCII-Code 10) abgeschlossen werden. Die jeweils maximal zulässige Länge einer Zeile ist inklusive Zeilenende-Zeichen auf 1023 Zeichen beschränkt.

Syntax einer Anfrage (Query)

Eine Anfrage hat folgende Syntax:

?<*Tabelle*><*Leerzeichen*><*Schlüssel*>

Eine Anfrage beginnt mit einem '?'-Zeichen (ASCII-Code 63). Dann folgt der Name der Tabelle, auf den sich die Anfrage bezieht. Als Trennzeichen folgt ein einzelnes Leerzeichen (ASCII-Code 32). Abschließend folgt der Schlüssel, nach dem gesucht wird. Die gesamte Zeile wird mit einem LF-Zeichen abgeschlossen (ASCII-Code 10). Der Name der Tabelle darf kein Leerzeichen enthalten. Der Schlüssel darf außer dem Zeilenendezeichen beliebige Zeichen enthalten. Null-Bytes sind an keiner Stelle erlaubt.

Beispiele:

```
?host.byname www.inf.fh-rhein-sieg.de
?passwd.byuid 1000
```

Syntax einer Schreiboperation (Set)

Eine Schreiboperation hat folgende Syntax:

=<*Tabelle*><*Leerzeichen*><*Schlüssel*>

Die Syntax ist bis auf das erste Zeichen identisch mit der einer Anfrage. Eine Schreiboperation beginnt mit einem '='-Zeichen (ASCII-Code 61). Es folgt der Name der Tabelle, in die ein Objekt eingefügt werden soll. Dann folgt ein Leerzeichen und der Schlüssel, unter dem das Objekt eingefügt werden soll. Ansonsten gelten die gleichen Regeln wie für die oben beschriebene Anfrage.

Übertragung von Objektmengen

Ein Objekt wird als eine Typangabe und eine Menge von Attributen übertragen. Die Typangabe wird wie folgt codiert:

+<*Typ-Name*>

Ein einzelnes Attribut wird wie folgt übertragen:

<*Attribut-Name*><*Leerzeichen*><*Attribut-Wert*>

Ein Attribut-Name muss mit einem Buchstaben beginnen und darf kein Leerzeichen enthalten. Der Attribut-Wert darf bis auf das Zeilenende-Zeichen und das Nullbyte beliebige Zeichen enthalten.

Die Attribute einer Attribut-Menge werden wie oben beschrieben nacheinander übertragen. Das Ende einer Attribut-Menge wird nicht explizit codiert. Entweder kann das Ende durch ein '.'-Zeichen (ASCII-Code 46) erkannt werden, welches auch das Ende der übertragenen Objektmenge anzeigt, oder durch die Typangabe des nächsten Objekts der Objektmenge.

Die Objekte einer Objektmenge werden wie oben beschrieben nacheinander übertragen, auf das letzte Objekt folgt eine Zeile mit einem einzelnen '.'-Zeichen.

Ein Beispiel für eine Objektmenge mit zwei Objekten:

```
+host
name localhost
af inet
addr 127.0.0.1
+host
name localhost
af inet6
addr ::1
.
```

Ein Beispiel für eine Objektmenge mit einem Objekt:

```
+pwent
name sleder2s
passwd *
uid 1000
gid 1000
gecos Sebastian Lederer
dir /home/sleder2s
shell /bin/sh
.
```

Ein Beispiel für eine leere Objektmenge:

.

Kapitel 7

Implementierung

7.1 Ziele der Implementierung

Gemäß dem obigen Entwurf soll der Namensdienst-Dæmon *lookupd* für mehr oder weniger alle Anfragen an Namensdienste oder Verzeichnisdienste zuständig sein. Das bedeutet, dass sehr viele im System laufenden Prozesse diverse Anfragen an *lookupd* stellen werden. Dazu gehören nicht nur Kommandos und Programme wie *ssh* oder *ping*, die im Netzwerk aktiv sind und daher Hostnamen auflösen müssen. Auch an unvermuteten Stellen wie z.B. beim Aufruf des *ls*-Befehls (der einen Verzeichnisinhalt auflistet) werden Namens- bzw. Verzeichnisdienste in Anspruch genommen, um z.B. *user-ids* (Benutzernummern) in lesbare Benutzernamen umzuwandeln. Das bedeutet, dass zu jeder Zeit von verschiedensten Programmen *lookupd*-Anfragen zu erwarten sind, und dass diese Anfragen in beliebiger Anzahl auch parallel auftreten können und werden.

Ebenso bedeutet dies, dass sich die Performance und die Verfügbarkeit des Dæmons direkt und für den Anwender deutlich sichtbar auf die Performance des Gesamtsystems auswirkt. Wäre z.B. die Antwortzeit auf eine Anfrage nach einem Benutzernamen durch ein *ls*-Kommando schlecht, weil möglicherweise gerade viele andere Anfragen bearbeitet werden, würde der Anwender direkt bemerken, dass die Reaktion des Systems auf die Eingabe seines Kommandos träge wird. Noch kritischer als die Performance ist die Verfügbarkeit. Sollte der Dæmon einmal nicht laufen, wäre die Funktionalität des Systems stark beeinträchtigt. Da auch die Benutzerinformationen über *lookupd* abgefragt werden, wäre eine Identifikation und Authentifizierung von Benutzern nicht mehr möglich. Das für den Anwender sichtbare Verhalten wäre dann, dass er sich nicht mehr beim System anmelden kann. Sollte der Dæmon zwar laufen, aber nicht korrekt funktionieren und keine Anfragen mehr bearbeiten, wäre eine ähnliche Situation erreicht. Unter diesen Umständen würden alle Prozesse, die eine lookupd-Anfrage gestellt haben, blockieren, weil sie (vergeblich) auf eine Antwort warten. Aus Anwendersicht würde dann das System teilweise oder ganz "hängen" und damit unbenutzbar werden.

Der *lookupd*-Dæmon wird also zu einem wichtigen Systembestandteil, ohne den das Funktionieren des Gesamtsystems nicht mehr gewährleistet ist und dessen Performance sich auch auf

den Rest des Systems auswirkt. Das bedeutet wiederum, das der Dæmon schon sehr früh beim Booten des Systems gestartet werden und ab dann permanent laufen muss, solange das System läuft. Die Tatsache, dass der Dæmon ständig im System präsent sein muss, lässt es auch wünschenswert erscheinen, dass er so wenig zusätzliche Systemressourcen wie möglich verbraucht. Hauptsächlich sollte möglichst wenig Speicher verbraucht werden und nur wenige zusätzliche Prozesse (wenn überhaupt mehr als einer) erzeugt werden. Zudem sollte der Dæmon möglichst effizient programmiert sein und wenig CPU-Zeit verbrauchen. Er sollte in der Lage sein, theoretisch beliebig viele gleichzeitig oder nahezu gleichzeitig gestellte Anfragen zu beantworten. Die bloße Fähigkeit, viele parallele Anfragen zu beantworten, reicht jedoch allein noch nicht aus, auch die Antwortzeiten dürfen bei hoher Belastung nicht überproportional steigen, der Dæmon muss also auch unter hoher Belastung noch in akzeptabler Zeit Antworten liefern.

Wichtig ist auch die Robustheit des Programmcodes. Der Dæmon muss immun sein gegen Anwendungen, die das interne Protokoll nicht korrekt implementieren oder sogar mit bösartiger Absicht versuchen, die Funktion des Dæmons zu stören. Es darf beispielsweise nicht möglich sein, den Dæmon durch das Senden sehr langer Strings, unsinniger Anfragen oder einfach Datenmülls zum Absturz zu bringen. Auch sollten unprivilegierte Benutzer nicht in der Lage sein, sicherheitsrelevante Daten wie Passwort-Hashes zu lesen.

Derartige Fähigkeiten sind wichtig für die Akzeptanz einer solchen neuartigen Namensdienst-Architektur. Ein Dæmon, der viel Speicher verbraucht, mehrere Prozesse erzeugt und mäßige Performance zeigt, lässt im praktischen Einsatz die Vorteile gegenüber der bisherigen Architektur nicht gerade besonders hervortreten. Die oben genannten Design-Ziele stellen jedoch auch sehr hohe Ansprüche dar. Dementsprechend muss der Entwurf und auch die Programmierung sehr sorgfältig unter Berücksichtigung dieser Kriterien geschehen, was, wie sich herausstellen wird, auch vermehrten Arbeitsaufwand verursacht.

7.2 Design-Entscheidungen

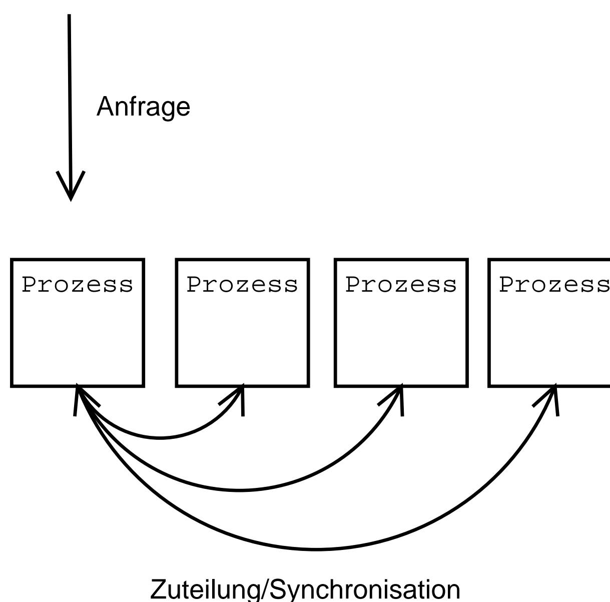
Der Namensdienst-Dæmon muss in der Lage sein, mehrere Verbindungen zu Anwendungen aufrecht zu erhalten und auf diesen Verbindungen verschiedene Anfragen parallel zu bearbeiten. Das Programm muss also mehrere Sockets verwalten, von diesen Sockets Daten lesen (die Anfragen), und die Antworten zurücksenden. Die übliche Methode für eine solche Situation wäre, in einer Schleife mit dem *select()*- oder *poll()*-Systemaufruf alle entsprechenden File-Deskriptoren auf anliegende Daten zu überprüfen und dann alle Anfragen nacheinander abzuarbeiten. Diese Methode ist jedoch hier nicht akzeptabel, da mehrere Anfragen, die von verschiedenen Anwendungen parallel gestellt werden, nur sequentiell abgearbeitet werden. Dies wäre unproblematisch, wenn alle Anfragen ohne Verzögerungen von außen sofort abgearbeitet werden könnten. Wird jedoch durch eine Anfrage z.B. eine DNS-Namensauflösung ausgelöst, die mehrere Sekunden dauert, müssten alle anderen Anfragen solange auf Bearbeitung warten, auch wenn sie möglicherweise sofort aus dem Cache beantwortet werden könnten. Die Performance wäre dadurch also teilweise beträchtlich reduziert. Dies ist für einen Dienst, der als übergreifendes Informationssystem für das ganze System zur Verfügung stehen soll, nicht ausreichend. Das Programm

muss also nicht nur mehrere Verbindungen gleichzeitig handhaben können, auch der Programmablauf muss parallelisiert werden können. Hierzu gibt es nun verschiedene übliche Ansätze: Mehrere Prozesse, Multithreading und Event-basierte Programmierung.

Prozesse Die erste Möglichkeit besteht darin, zur Parallelisierung mehrere Prozesse zu verwenden. Dazu wird entweder für jede neue Verbindung ein eigener Prozess erzeugt, oder aber man erzeugt im voraus mehrere Prozesse, denen dann durch einen besonderen Zuteilungs-Prozess bestimmte Verbindungen oder einzelne Anfragen zugewiesen werden (*pre-forked server*). Beide Varianten haben für den hier vorgesehenen Einsatzzweck Nachteile. Erzeugt man für jede Verbindung einen eigenen Prozess, so muss man mit größerem Overhead für den Aufbau einer Verbindung rechnen. Da in der Regel für jede Anfrage eine eigene Verbindung aufgebaut und nach Erhalt der Ergebnisses wieder abgebaut wird, würde dies zu einer großen Anzahl von neu erzeugten und terminierenden Prozessen führen. Das Erzeugen eines neuen Prozesses durch den *fork*-Systemaufruf ist zwar nicht sehr aufwendig, da nur eine Kopie eines existierenden Prozesses geschaffen wird und durch moderne Virtual-Memory-Technologien (*Copy-On-Write*-Verfahren) nur sehr wenige Speicherseiten für den neuen Prozess angelegt werden müssen. Trotzdem ist dieses Verfahren durch die zu erwartende hohe Anzahl der *fork*-Aufrufe nicht optimal. Beim Pre-forked-Modell besteht dieses Problem nicht, dafür werden jedoch ständig mehrere Plätze der Prozesstabelle belegt, was dem Design-Prinzip des minimalen Ressourcenverbrauchs widerspricht. Außerdem verursacht das Verwalten der verschiedenen Prozesse und das Zuteilen von eingehenden Verbindungen an einen der Prozesse erhöhten Aufwand. Problematisch wäre bei diesem Modell auch die Notwendigkeit zur Realisierung eines eines gemeinsamen Caches, der entweder durch gemeinsam genutzten Speicher oder durch zusätzliche Kommunikation der Prozesse untereinander realisiert werden müsste.

Threads Ein weiterer Ansatz verwendet Multithreading zur Parallelisierung. Er ist ähnlich zum vorhergehenden Ansatz, verwendet aber statt mehrerer Prozesse Threads, die innerhalb eines einzigen Prozesses ablaufen. Analog dazu existieren hier die Varianten, für jede neue Verbindung einen eigenen Thread zu erzeugen, oder aber eine mehr oder weniger feste Anzahl von Threads ständig bereitzuhalten, denen dann einzelne Anfragen zugeteilt werden (*thread pool*). Da Threads im Vergleich zu ausgewachsenen Prozessen wesentlich leichtgewichtigere Objekte sind, fällt der Overhead zum Erzeugen und Terminieren von (vielen) Threads nicht so drastisch aus wie bei Prozessen, ist aber trotzdem vorhanden. Hinzu kommt der programmtechnische Aufwand zur Synchronisation der parallel laufenden Threads, besonders bei dem Zugriff auf gemeinsam genutzte Datenstrukturen wie vor allem den Cache. Verwendet man einen Thread-Pool, erhöht sich der Programmieraufwand für dessen Verwaltung, denn nun müssen nicht nur Zugriffe auf gemeinsam genutzte Daten synchronisiert werden, sondern auch ein Management von wartenden und arbeitenden Threads durchgeführt werden. Dazu muss auch eine Verwaltung von noch nicht abgearbeiteten Anfragen und eine Zuteilung zu wartenden Threads geschaffen werden. Ein ganz anderer gravierender Nachteil einer Multithreaded-Lösung kommt allerdings aus einer anderen Richtung: Sowohl unter dem hier verwendeten Betriebssystem

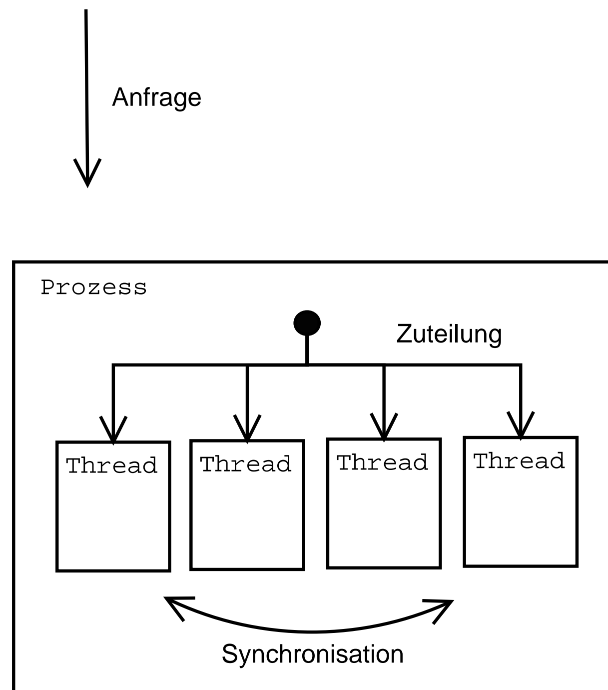
Abbildung 7.1: Parallelisierung durch Prozesse



FreeBSD 4.0 als auch unter der anderen Open-Source-Alternative Linux existieren keine qualitativ hochwertigen Threads-Implementierungen. Unter FreeBSD 4.0 werden Threads mit Hilfe einer speziellen Version der C-Bibliothek ohne Hilfe des Betriebssystems realisiert (*user space threads*). Dies hat zur Konsequenz, dass die Programmgröße sich durch die zusätzlich enthaltene Threading-Funktionalität erhöht. Außerdem lassen sich auf diese Weise nicht mehrere Prozessoren ausnutzen, was aber für den hier vorgesehenen Zweck nicht ins Gewicht fällt. Die Threads-Implementierung unter Linux hingegen verwendet mehrere Prozesse mit gemeinsam genutztem Speicher, um Threads zu simulieren. Damit würden die Nachteile der Multi-Prozess-Lösung und die oben genannten Nachteile der Multithreading-Lösung kombiniert werden.

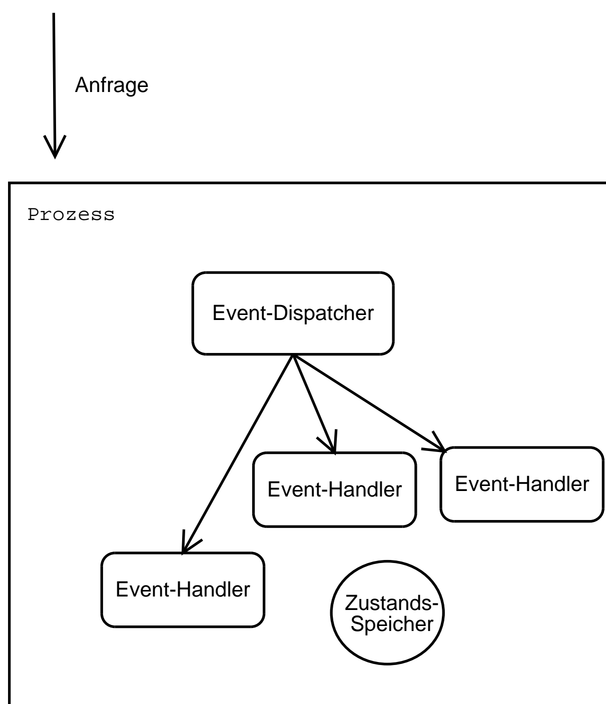
Events Der letzte Ansatz schließlich verwendet eine ereignisgesteuerte Programmstruktur, um die Abarbeitung von Anfragen zu parallelisieren. Tatsächlich gibt es bei dieser Methode keine parallelen Abläufe im Programm. Stattdessen werden alle Abläufe von externen Ereignissen (*Events*) angestoßen. Dabei darf es keine einzelnen Abläufe geben, die längere Zeit andauern oder Stellen, wo eine gewisse Zeit gewartet wird. Eine Funktion muss "schnell" beendet sein, hauptsächlich darf nicht auf Eingaben oder das Verstreichen eine Zeitspanne gewartet werden. Derartige Programmstrukturen werden durch Ereignisse ersetzt, die z.B. durch Ein-/Ausgabe oder zeitgesteuert ausgelöst werden. Das Programm kann sich dabei in unterschiedlichen Zuständen befinden, wobei bestimmte Ereignisse einen Übergang in einen anderen Zustand auslösen, ähnlich wie bei einem endlichen Automaten. Unterschiedliche Zustände bedeuten hier unterschiedliche Reaktionen auf Ereignisse. So würde zum Beispiel beim Einlesen einer Text-Zeile nicht so lange in einer Schleife Zeichen gelesen, bis ein Zeilen-Ende-Zeichen gefunden wurde, sondern das Eintreffen der Zeichen wären Ereignisse, die den Aufruf einer Funktion auslösen.

Abbildung 7.2: Parallelisierung durch Threads



Diese Funktion sichert die Zeichen in einem Puffer und kehrt sofort zum Aufrufer zurück. Wurde das Zeilen-Ende-Zeichen gefunden, geht das Programm in einen anderen Zustand über, der eine weitere Verarbeitung der eingelesenen Zeile anstößt. Während die Zeile nach und nach eingelesen wird, können nebenbei beliebige andere Ereignisse, die zu völlig anderen Abläufen gehören, verarbeitet werden. Ein anderes Beispiel wäre eine Anfrage an einen DNS-Server. Hier würde man nicht wie üblich die Anfrage senden und (mit einem Timeout) auf die Antwort warten. Stattdessen wird die Verarbeitung nach dem Senden der Anfrage vorerst beendet und eventuell andere anstehende Aufgaben bearbeitet. Das Eintreffen der Antwort vom DNS-Server und das Timeout bedeuten Ereignisse, die den Aufruf von entsprechenden Funktionen zur Folge haben. Der Empfang des Antwort-Pakets des DNS-Servers würde eine Funktion aufrufen, die das Paket auswertet und das Ergebnis an den ursprünglichen Aufrufer weiterleitet. Würde das Timeout ausgelöst, würde eine andere Funktion aufgerufen werden, die zum Beispiel eine Fehlermeldung an den ursprünglichen Aufrufer schickt. Der offensichtliche Nachteil dieses Ansatzes ist, dass man alle Programmabläufe an das ereignisorientierte Konzept anpassen muss und oft nicht wie gewohnt programmieren kann. Außerdem muss man darauf achten, dass viele Bibliotheksfunktionen nicht oder nur eingeschränkt verwendet werden können, weil sie die oben aufgeführten Bedingungen nicht erfüllen, wie zum Beispiel *fgets()* oder *gethostbyname()*. Beide Funktionen können unter Umständen (keine Eingabe vorhanden bzw. keine Antwort vom DNS-Server) über längere Zeit blockieren, was die Verarbeitung von anstehenden Events verhindert.

Abbildung 7.3: Parallelisierung durch Events



Der verwendete Ansatz Für die tatsächliche Implementierung wurde der Event-Ansatz verwendet, wobei zusätzlich in bestimmten Situationen mehrere Prozesse verwendet werden. Der Event-Ansatz schien aus verschiedenen Gründen besonders geeignet. Das resultierende Programm hat eine geringe Größe und hohe Effizienz. Es ist in der Lage, sehr viele parallele Anfragen zu bearbeiten, ohne das System stark zu belasten. Es ist keine leistungsfähige Threads-Implementierung im Betriebssystem oder die aufwendige Programmierung von gemeinsam genutztem Speicher und Synchronisationsmechanismen erforderlich. Die Problematik der besonderen eventbasierten Programmierung wird teilweise gemildert, indem zusätzliche Prozesse verwendet werden. Dies ist zum Beispiel beim aufruf der Resolver-Funktionen notwendig, die DNS-Anfragen realisieren. Für den Aufruf dieser Funktion wird mit dem *fork*-Systemaufruf ein neuer Prozess erzeugt, der über lokale Prozesskommunikation (eine *pipe*) mit dem ursprünglichen Prozess in Verbindung steht. Wenn die Resolver-Funktion zurückkehrt, wird das Ergebnis über die pipe zurückgesendet, was beim Empfänger ein Ereignis darstellt. Damit ist die ereignisorientierte Struktur gewahrt und die vorhandenen Resolver-Funktionen können trotzdem genutzt werden. Aus Sicht der Performance ist das Erzeugen eines neuen Prozesses nicht optimal, dies ist aber nur dann notwendig, wenn sich das Ergebnis der Anfrage noch nicht im Cache befindet. In diesem Fall muss eine Anfrage über das Netzwerk an den DNS-Server (oder einen anderen Namensdienst-Server) gestellt werden, was mit großer Wahrscheinlichkeit ohnehin wesentlich länger dauert als das Erzeugen eines neuen Prozesses mit *fork*. Es bleibt das Problem, dass bei vielen parallelen (unterschiedlichen) DNS-Anfragen viele neue Prozesse erzeugt werden, was eine Belastung für das System darstellen kann. Dieses Problem wird aber durch den Cache ge-

mildert und könnte durch eine Begrenzung der maximal gleichzeitig erzeugten Prozesse behoben werden. Dies wurde jedoch (noch) nicht implementiert, da sich im Betrieb in der Praxis keine Probleme durch zu viele Prozesse zeigten.

7.3 Einzelne Module

7.3.1 Überblick

Die Implementierung wurde in verschiedene Quelltext-Module aufgeteilt, die nach Funktionalität und nach Art der zu verarbeitenden Daten gruppiert sind. Dabei wurde versucht, so weit es in einer prozeduralen Sprache möglich ist, objektorientiert vorzugehen. Ein Modul soll jeweils eine bestimmte Klasse implementieren. Dazu enthält es die Datenstrukturen und alle Funktionen, die auf diesen Strukturen operieren. Dabei wurde die Namenskonvention verwendet, den Namen der C-Funktionen jeweils am Anfang den Namen der Klasse voranzustellen. Üblicherweise umfasst ein Modul eine C-Quelltext-Datei und eine Header-Datei, die den Namen der Klasse tragen. Dies konnte allerdings nicht immer strikt einzuhalten werden. In bestimmten Fällen enthält eine Quelltextdatei mehrere Klassen, die eng zusammenarbeiten, oder aber es wurden Funktionen aus verschiedenen Klassen in einer Datei zusammengefasst, weil diese Funktionen den Anteil der Module darstellen, die ausschließlich von Anwendungsprogrammen genutzt werden.

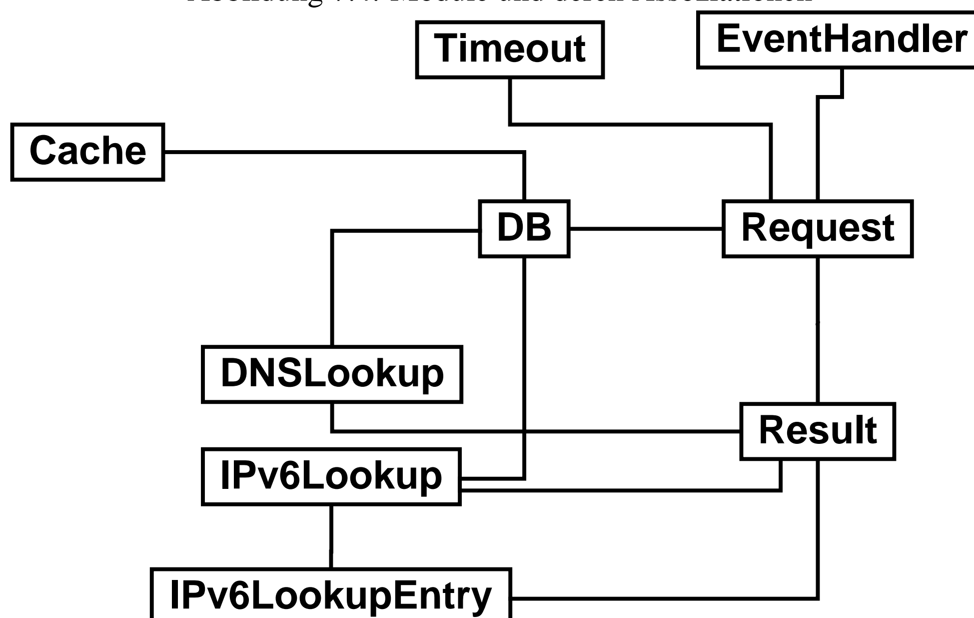
Die Module *list*, *StringMap* und *HashTable* stellen abstrakte Datentypen zur Verfügung, die von anderen Modulen zur Handhabung von verschiedenen Daten verwendet werden. Die Module *Events* und *Timeout* liefern die Grundfunktionalität zur Realisierung der asynchronen, eventorientierten Programmstruktur. Die Module *Request* und *Result* repräsentieren die interne Darstellung und die Codierung der Protokollobjekte sowie die Funktionalität zum Speichern und Bearbeiten von Anfrageergebnissen. Das Modul *Cache* verwaltet die verschiedenen Caches zum Zwischenspeichern von Anfrageergebnissen. Das Modul *DB* stellt eine abstrahierte Schnittstelle zu Informationssystemen dar, die unter anderem die Aufgabe hat, nacheinander verschiedene konkrete Informationssysteme abzufragen. Die Module *IPv6Lookup* und *DNSLookup* sind zwei Implementierungen für eine solche Abfrage von Informationssystemen. Schließlich existiert noch das Haupt-Modul *lookupd*, was die Programminitialisierung und den Aufruf der Haupt-Event-Schleife übernimmt sowie für den Aufbau und Annahme der Socket-Verbindungen sorgt.

Die einzelnen Module sollen nun im folgenden näher betrachtet werden.

7.3.2 Abstrakte Datentypen

List Das Modul *List* implementiert eine einfach verkettete Liste. Die zugehörigen Quelltext-Dateien sind *list.c* und *list.h*. Die Elemente der Liste sind generische Zeiger (*void*-pointer), so dass beliebige Objekte in der Liste gespeichert werden können. An Funktionalität wird das Einfügen von Objekten, das Anhängen eines Objektes am Ende der Liste, das Durchlaufen der

Abbildung 7.4: Module und deren Assoziationen



Liste und das Löschen eines bestimmten Objektes geboten. Die Implementierung ist an dem Beispiel im Einführungskapitel aus [Sedgewick98] angelehnt und verwendet einen besonderen Listen-Knoten, der das Ende und den Anfang der Liste markiert, um nicht die Spezialfälle "Ende der Liste" und "Anfang der Liste" gesondert abzufragen und implementieren zu müssen. Das *List*-Modul wird von vielen anderen Modulen verwendet, wobei der Zugriff auf die Liste oft in eine kleine zusätzliche Funktion eingepackt wird, um die Konvertierung der verwendeten Datentypen in die von der Liste verwendeten *void*-pointer vor dem Aufrufer zu verbergen.

StringMap Dieses Modul ist in den Dateien *StringMap.c* und *StringMap.h* implementiert. Es realisiert eine Datenstruktur, die mehrere Key/Value-Paare enthält. Eine solche Datenstruktur wird gewöhnlich als *Map* (z.B. im Java-Collection-Framework) oder als *Dictionary* bezeichnet. Diese Implementierung bildet je einen String auf einen anderen String ab, daher wurde der Name *StringMap* gewählt. Die StringMap bietet im wesentlichen Funktionen zum Abfragen eines Wertes zu einem Schlüssel (*StringMapGet()*) und zum Eintragen eines Key/Value-Paares. Ist ein Schlüssel bereits vorhanden, wird der alte Eintrag überschrieben, mehrere Werte zu einem Schlüssel sind nicht möglich. Strings, die als Parameter zum Eintrag in die StringMap übergeben werden, werden intern kopiert. Das kostet in einigen Fällen zwar mehr Speicherplatz, befreit den Aufrufer aber von dem Aufwand, die Speicherplatzbelegung von Strings, die in eine StringMap eingetragen wurden, genau zu verfolgen. Die StringMap greift intern zum Speichern der Daten auf das *list*-Modul zurück, die String-Paare werden in einer Liste abgelegt. Dieses Implementierungsdetail bleibt dem Aufrufer verborgen, so dass bei Bedarf auch eine andere Implementierung (z.B. mit der Hash-Tabelle) eingesetzt werden könnte. Es zeigt sich jedoch, dass die einfache Listen-Implementierung für den hier vorgesehenen Einsatz ausreicht.

HashTable Dieses Modul befindet sich in den Dateien *HashTable.c* und *HashTable.h*. Darin wird eine generische Hash-Tabelle implementiert, die Kollisionen durch Listen (Buckets) auffängt und sich bei zunehmendem Füllstand selbst vergrößert. Die Hash-Tabelle verwaltet selbst nur generische Zeiger auf beliebige Daten, wobei ein Element auch gleichzeitig der Schlüssel ist. Beim Erzeugen einer neuen Hash-Tabelle kann eine Hash-Funktion (in Form eines Funktionszeigers) und eine Vergleichsfunktion angegeben werden. Dadurch ist das Modul in der Lage, beliebige Objekte in der Hash-Tabelle zu speichern. Ein weiteres Merkmal dieser Implementierung ist die Fähigkeit, mehrere Objekte mit gleichem Schlüssel (also mehrere für die Hash-Tabelle äquivalente Objekte) zu speichern. Existieren mehrere gleiche Objekte, werden diese eindeutig, aber in nicht definierter Reihenfolge durchnummeriert. Mit einer entsprechenden Funktion kann dann ein Objekt mit einem bestimmten Index abgefragt werden (*HashTableGetN()*). Verwendet man in einem solchen Fall die normale Funktion zum Abfragen der Hash-Tabelle (*HashTableGet()*), so wird das erste Objekt geliefert. Löscht man in einem solchen Fall ein Objekt (*HashTableRemove()*), so wird jeweils immer nur das erste äquivalente Objekt gelöscht. Will man also alle äquivalenten Objekte löschen, muss *HashRemove()* mehrfach aufgerufen werden. Es gibt keine Möglichkeit, ein bestimmtes Objekt aus einer Menge von mehreren äquivalenten Objekten zu löschen (da diese für die Hash-Tabelle ja alle gleich sind). In der vorliegenden Implementierung wurde als Parameter für den maximalen Füllstand ein Wert von $1\frac{1}{3}$ gewählt. Ist die Anzahl der Elemente in der Hash-Tabelle um diesen Faktor größer als die Anzahl der Buckets, wird die Größe der Hash-Tabelle verdoppelt (nach der Formel $neu = 2 \cdot alt + 1$).

Eine weitere Besonderheit bietet die Funktion *HashTableFilter()*. Ihre Aufgabe ist das "Herausfiltern" von mehreren Objekten aus der Hash-Tabelle. Die herausgefilterten Objekte werden dabei aus der Hash-Tabelle entfernt. Zum Filtern wird zunächst ein Vergleichsobjekt (also ein Schlüssel) angegeben. Für alle Objekte, die zu diesem Vergleichsobjekt passen, wird dann eine beim Aufruf von *HashTableFilter()* als Parameter angegebene Funktion aufgerufen (die Filterfunktion). Diese Filterfunktion kann nun durch ihren Rückgabewert signalisieren, ob das jeweilige Objekt gelöscht werden soll oder nicht. Der Vorteil dieser Methode liegt darin, dass mit einem Durchlauf durch die Hash-Tabelle viele Objekte auf einmal gelöscht werden können. Besonders beim Löschen von Objekten aus großen Hash-Tabellen lässt sich so die Performance gegenüber einzelnen Löschoperationen stark verbessern. Diese Funktionalität wird vom Cache-Modul genutzt.

Die Module, die die Hash-Tabelle verwenden, verwenden sozusagen immer eine spezialisierte Fassung, indem sie eine spezielle Hash-Funktion bei der Erzeugung angeben und die Konvertierung der zu speichernden Objekte in generische Zeiger mit Hilfe von Wrapper-Funktionen übernehmen.

7.3.3 Request

Das *Request*-Modul, das sich in den Dateien *Request.h* und *Request.c* befindet, ist eine der zentralen Stellen des Programms. Hier wird das Empfangen und Verarbeiten von Anwendungs-Anfragen gesteuert und alle weiteren Verarbeitungsschritte eingeleitet. Die *Re-*

quest-Datenstruktur bzw. die *Request*-Objekte dienen als Container, von denen alle weiteren Informationen, die eine Anfrage betreffen, abhängen.

Am Anfang der Verarbeitung einer Anfrage steht die Funktion *RequestReadHandler()*. Für jede Verbindung wird diese Funktion als Event-Handler für ein Lese-Ereignis eingerichtet. Wenn nun auf dem Socket Daten eintreffen, werden sie in *RequestReadHandler()* verarbeitet. Hier wird zunächst eine *Request*-Datenstruktur erzeugt, die von nun an alle weitere Informationen aufnehmen wird, bis der Socket wieder geschlossen wird. Ein Zeiger auf dieses *Request*-Objekt wird in der dazugehörigen *EventHandler*-Struktur gespeichert, um sie bei weiteren Aufrufen von *RequestReadHandler()* zur Verfügung zu haben. Da die Daten in beliebig großen (oder kleinen) Portionen ankommen können, werden sie zunächst in einem Puffer, der zum *Request*-Objekt gehört, gespeichert. Dann wird überprüft, ob eine vollständige Zeile eingelesen wurde und, wenn dies nicht der Fall ist, die Kontrolle an den Aufrufer zurückgegeben, damit andere Events bearbeitet werden können. Es können also mehrere Aufrufe von *RequestReadHandler()* notwendig sein, bis eine Zeile vollständig gelesen wurde. Dies kann parallel für verschiedene Verbindungen geschehen. Zu jeder Verbindung existiert dann ein *EventHandler*-Objekt und ein *Request*-Objekt, und jedes *Request*-Objekt enthält einen eigenen Puffer mit den bisher gelesenen Daten. Somit können beliebig viele parallele Anfragen verarbeitet werden, wobei zu jedem Zeitpunkt jeweils für jede Verbindung nur eine Anfrage existieren kann.

Beim Lesen von Daten kann eine besondere Situation eintreten: Der Sender hat die Möglichkeit, Authentifizierungsdaten senden. Dies geschieht mit dem *sendmsg()*-Systemaufruf unter der Verwendung sogenannter *Control Messages*. Eine Art dieser *Control Messages* dient dazu, dem Empfänger Authentifizierungsinformationen (*credentials*) des sendenden Prozesses zu übermitteln. Der Typ dieser *Control Message* wird als *SCM_CREDS* bezeichnet und kann nur für lokale Sockets verwendet werden. Dabei wird die *user-id* und die *group-ids* übertragen. Der Sender gibt dazu eine entsprechende Datenstruktur an, die aber beim Senden vom Kernel ausgefüllt wird und dann vom Empfänger gelesen werden kann. Der empfangende Prozess kann somit feststellen, von welchem Benutzer der sendende Prozess ausgeführt wird. Dieser Mechanismus ist spezifisch für FreeBSD, wobei alle anderen BSD-Varianten ähnliche oder identische Möglichkeiten bieten. Die *RequestReadHandler()*-Funktion ist in der Lage, eine *SCM_CREDS-Control-Message* zu empfangen und auszuwerten. Dabei wird überprüft, ob die gesendete *user-id* gleich null ist. Wenn dies der Fall ist, wird das *Request*-Objekt als *privilegiert* markiert. In der derzeitigen Implementierung wird diese Markierung nicht weiter genutzt, sie kann aber in Zukunft genutzt werden, um den Zugriff auf einzelne Tabellen zu regeln. So sollte z.B. der Zugriff auf Passwort-Informationen nur dem Superuser (mit der *user-id* null) erlaubt sein.

Wenn nun eine vollständige Zeile eingelesen wurde, wird in der *RequestReadHandler()*-Funktion die weitere Verarbeitung eingeleitet. Dazu wird die *RequestProcess()*-Funktion aufgerufen. Hier wird die eingelesene Zeile nach der Spezifikation des internen Protokolls in die einzelnen Teile aufgespalten. Dann wird, abhängig davon, ob es sich um ein *Get-Request* oder ein *Set-Request* handelt, die Funktion *RequestQuery()* oder *RequestSet()* ausgeführt, wo die spezifische Verarbeitung der Requests abläuft. Bei der *RequestSet()*-Funktion ist diese relativ kurz, diese Funktionalität ist (noch) nicht implementiert. Hier wird nur die Anfrage mit einem Fehler beendet. Die Funktion *RequestQuery()* dient zum tatsächlichen Abarbeiten einer Anfrage. Die wesentli-

che Arbeit dazu wird von dem *DB*-Modul geleistet, daher wird nach einigen Vorbereitungen in *RequestQuery()* nur die *DBQuery*-Funktion aufgerufen. Das Ergebnis der Anfrage wird dabei in Form einer Referenz auf ein *Result*-Objekt in dem *Request*-Objekt gespeichert.

Die Funktion *RequestReply()* dient schließlich dazu, ein Ergebnis auf eine Anfrage an die Anwendung zurückzusenden. Das Ergebnis muss zu diesem Zeitpunkt bereits in dem *Result*-Objekt, das von dem *Request*-Objekt referenziert wird, gespeichert sein. Mit Hilfe der Funktion *ResultSend()* aus dem *Result*-Modul wird dann das *Result*-Objekt über den Filedeskriptor der Verbindung gesendet. Anschließend wird das *Request*-Objekt als abgearbeitet gekennzeichnet.

Der Lebenszyklus eines *Request*-Objekts unterliegt besonderen Bedingungen. Er ist nicht direkt an eine einzelne Anfrage gekoppelt, statt dessen kann ein *Request*-Objekt für mehrere aufeinanderfolgende Anfragen verwendet werden. Beim Annehmen einer Verbindung wird jeweils ein *Request*-Objekt erzeugt. Ist eine Anfrage abgearbeitet, wird nur der Inhalt des *Request*-Objekts gelöscht (z.B. das *Result*-Objekt). Anschließend kann eine weitere Anfrage erfolgen, für die das selbe *Request*-Objekt verwendet wird. Erst wenn der Socket einer Verbindung geschlossen wird, entweder durch die Anwendung oder den Dæmon selbst, wird das *Request*-Objekt und der zugehörige Event-Handler gelöscht.

Das Löschen eines *Request*-Objekts ist ein besonderes Problem, da es vorkommen kann, dass das Objekt an anderen Stellen des Programms noch referenziert wird. Will man z.B. in der *RequestReadHandler()*-Funktion das Objekt löschen, weil der Socket geschlossen wurde, kann es sein, dass in Timeout-Handlern zu einem späteren Zeitpunkt noch auf dieses Objekt zugegriffen wird. Diese Problematik führt zu einem etwas komplizierten Management der *Request*-Objekte. Generell darf ein *Request*-Objekt überhaupt nicht explizit gelöscht werden. Statt dessen muss eine Funktion, die mit *Request*-Objekten arbeitet (z.B. die Funktionen aus den *IPv6Lookup*- und *DNSLookup*-Modulen), die Funktion *RequestDone()* aufrufen, wenn das *Request*-Objekt nicht mehr benötigt wird. Alternativ kann auch die Funktion *RequestReply()* verwendet werden, hier wird implizit *RequestDone()* aufgerufen. Wenn die Bearbeitung einer Anfrage gestartet wurde, wird das *Request-Objekt* als in Bearbeitung befindlich markiert. In der *RequestDone()*-Funktion wird diese Markierung wieder aufgehoben. Es kann nun vorkommen, das ein Socket von einer Anwendung geschlossen wird, bevor das Ergebnis der Anfrage an die Anwendung gesendet wurde, also während das *Request*-Objekt noch aktiv ist. Das Schließen eines Sockets wird in der *RequestReadHandler()*-Funktion festgestellt. Hier müsste nun eigentlich das *Request*-Objekt gelöscht werden, was aber nicht möglich ist, wenn es noch aktiv ist. Daher wird das *Request*-Objekt als abgebrochen markiert. Dies geschieht mit der Funktion *RequestAbort()*. Außerdem wird mit der Funktion *RequestClose()* der zugehörige Socket geschlossen, um den Filedeskriptor wieder freizugeben. Zu einem späteren Zeitpunkt wird nun die Bearbeitung des *Request*-Objekts fortgesetzt (sonst wäre es nicht als aktiv markiert worden) und zu deren Abschluss die Funktion *RequestDone()* aufgerufen. Hier wird nun erkannt, dass die Anfrage abgebrochen wurde, und das *Request*-Objekt wird tatsächlich gelöscht. Falls beim Schließen eines Sockets auf Anwendungsseite in *RequestReadHandler()* das zugehörige *Request*-Objekt nicht als aktiv markiert ist, z.B. nach dem Abschließen einer Anfrage oder während des Einlesens einer Zeile, ist dieser Aufwand nicht nötig. In diesem Fall wird das *Request*-Objekt sofort gelöscht.

7.3.4 Result

Das *Result*-Modul dient zur Verwaltung von Anfrageergebnissen. Dies ist eine nicht völlig triviale Aufgabe, da das *Result*-Objekt mehrere Ergebnisse zu einer Anfrage aufnehmen können muss. Der Großteil der Funktionalität ist in den Dateien *Result.h* und *Result.c* implementiert. Ein einzelnes Ergebnis, hier als ein *Result*-Eintrag bezeichnet, wird in einer *StringMap* gespeichert. Für das *Result*-Objekt und damit für alle Einträge kann ein Typ festgelegt werden. Für jede Tabelle, aus der das Anfrageergebnis stammt, existiert eine Typdefinition und entsprechende Menge von Attributen. Diese Zuordnung entspricht der Definition des internen Protokolls in Abschnitt 6.2. Die Festlegung des Typs geschieht durch die Funktion *ResultSetType()*.

Der Zugriff auf *Result*-Objekte erfolgt meistens über einzelne Einträge, ein *Result*-Objekt dient gewissermaßen nur als Container für Einträge. Das Füllen eines *Result*-Objektes erfolgt durch das Hinzufügen von Einträgen, der einzige Weg, um Informationen von dem *Result*-Objekt zu erhalten, ist das Durchlaufen aller Einträge.

Zum Hinzufügen von Einträgen dient die Funktion *ResultAddEntry()*, der man als Parameter außer dem *Result*-Objekt selbst eine *StringMap*-Datenstruktur mit den Attributen des Ergebnisses angibt. Dabei wird im *Result*-Objekt nur eine Referenz auf die *StringMap*-Struktur gespeichert, die *StringMap* selbst wird nicht kopiert, darf also später vom Aufrufer nicht wieder freigegeben werden. Dieses Verhalten ist dadurch begründet, dass in den meisten Fällen die *StringMaps*, die im *Result* eingetragen werden sollen, nur für diesen Zweck angelegt wurden und anschließend nicht mehr verwendet werden. Aus Performance-Gründen ist es hier also zweckmäßig, auf eine weitere Kopie zu verzichten. Das *StringMap*-Objekt, das bei *ResultAddEntry()* übergeben wird, geht in den Besitz des *Result*-Objekts über. Dementsprechend wird es auch beim Zerstören des *Result*-Objekts automatisch wieder freigegeben.

Das Abfragen von Informationen aus einem *Result*-Objekt geschieht über einen Iterator-Mechanismus, der in den Funktionen *ResultFirstEntry()* und *ResultNextEntry()* enthalten ist. *ResultFirstEntry()* liefert immer den ersten Eintrag zurück. Außerdem wird ein Zeiger auf die aktuelle Position in der Liste aller Einträge zurückgeliefert. Dieser Zeiger muss bei allen darauffolgenden Aufrufen von *ResultNextEntry()* wieder als Parameter übergeben werden. Dadurch kann die Liste aller Einträge durchlaufen werden. Da dieser Zeiger von *ResultFirstEntry()* und *ResultNextEntry()* wieder geändert werden muss, ist der Parameter tatsächlich ein Zeiger auf den Zeiger. Der Eintrag selbst wird als Rückgabewert der Funktion zurückgeliefert. Wenn es keine weiteren (oder überhaupt keine) Einträge gibt, wird hier NULL zurückgeliefert.

Weitere Funktionalität des *Result*-Moduls bezieht sich auf das Senden und Empfangen von *Result*-Objekten über Sockets oder Pipes (technisch gesehen allgemein über File-Deskriptoren). Zum Senden eines *Result*-Objektes dient die Funktion *ResultSend()*, die als Parameter einen File-Deskriptor und ein *Result*-Objekt erhält. Der File-Deskriptor muss zum Schreiben geöffnet worden sein. Die *ResultSend()*-Funktion versetzt den File-Deskriptor immer in den nichtblockierenden Modus, damit die schnelle Beendigung der Funktion garantiert ist. Andernfalls wäre es denkbar, das eine Anwendungen fehlerhaft oder böswillig das gesendete Ergebnis nicht ausliest und damit nach Erreichen einer internen Puffergröße den Dæmon blockiert. Das Empfangen eines *Result*-Objekts geschieht nur in der Anwendung, daher existiert dieses Problem bei

ResultReceive() nicht. Dies *ResultReceive()*-Funktion befindet sich mit anderen Funktionen, die für die Anwendungsseite relevant sind, in der Datei LookupQuery.c. Durch diese zunächst unsinnig erscheinende Aufteilung wird zwar das Modul auf verschiedene nicht zusammengehörige Dateien verteilt. Eine Anwendung braucht dadurch jedoch nicht den in diesem Fall unbenutzten Code des Dæmons (aus Result.c) zu enthalten.

7.3.5 DB

Das Modul *DB* (für DataBase) bietet die Schnittstelle, um eine Anfrage, die durch ein *Request*-Objekt dargestellt wird, tatsächlich durch verschiedene Namensdienst-Module abarbeiten zu lassen. Die beteiligten Quelltext-Dateien sind *DB.c* und *DB.h*. Zum Bearbeiten einer Anfrage wird die Funktion *DBQuery()* verwendet, die als Parameter ein *Request*-Objekt erhält. Die unterschiedlichen Namensdienst-Module stellen jeweils eine eigene Abfrage-Funktion zur Verfügung. Im *DB*-Modul ist eine Liste der verfügbaren Abfrage-Funktionen vorhanden, die sequentiell durchlaufen wird. Die Reihenfolge kann über eine Konfigurationsdatei festgelegt werden (in lookupd.h definiert als */usr/local/etc/lookupd.conf*). Problematisch ist nun der Fall, wenn eine Anfrage-Funktion aufgerufen wird, die auf ein (meist zeitgesteuertes) Ereignis wartet. In diesem Fall kehrt die Abfrage-Funktion, nachdem sie entsprechende Event-Handler installiert hat, zur *DBQuery()*-Funktion zurück. An dieser Stelle besteht keine Möglichkeit, zu bestimmen, ob die Anfrage Erfolg hatte oder nicht, und ob die nächste Abfrage-Funktion aus der Liste aufgerufen werden muss. Dies kann erst geschehen, wenn die zuletzt angestoßene Abfrage (bzw. die gerade laufende) beendet wurde. Dies geschieht jedoch außerhalb des Einflussbereichs der *DBQuery()*-Funktion. Deshalb ist die jeweilige Abfrage-Funktion selbst dafür verantwortlich, eine fehlgeschlagene Anfrage beim nächsten Namensdienst-Modul fortzusetzen. Dies geschieht mit der Funktion *DBResumeQuery()*. Sie funktioniert genau wie *DBQuery()*, mit dem Unterschied, dass die nächste Abfrage-Funktion aufgerufen wird anstatt der ersten aus der Liste. Die Information, welche Listenposition gerade aktuell ist, wird dabei im *Request*-Objekt gespeichert. Es ist auch möglich, dass eine Abfragefunktion sofort feststellt, dass die Anfrage fehlgeschlagen ist (z.B. weil der gewünschte Namensdienst nicht aktiviert ist). In diesem Fall kann die Abfrage-Funktion einen Fehlercode zurückgeben, und *DBQuery()* oder *DBResumeQuery()* rufen direkt die nächste Abfrage-Funktion aus der Liste aus.

Das Modul *DB* realisiert in der *DBQuery()*-Funktion auch die Abfrage der Caches. Bevor eine tatsächliche Namensdienst-Anfrage gestellt wird, wird überprüft, ob ein passender Cache-Eintrag existiert (mit der Funktion *CacheGet()* aus dem *Cache*-Modul). Ist ein Eintrag im Cache vorhanden, wird sofort eine entsprechende Antwort gesendet (in der Funktion *DBQueryCache()*) und die Bearbeitung der Anfrage abgeschlossen.

Die Funktion *DBInit()* muss einmal aufgerufen werden, bevor andere Funktionen aus dem *DB*-Modul angesprochen werden können. Bei der derzeitigen Implementierung wird hier die Konfigurationsdatei gelesen und die Liste der Abfrage-Funktionen initialisiert.

7.3.6 Cache

Das Modul *Cache* erledigt die Verwaltung von mehreren Caches. Es befindet sich in den Dateien *Cache.c* und *Cache.h*. Der Cache dient dazu, um die Ergebnisse aus vorangegangenen Anfragen zu speichern und anhand eines Schlüssels schnell wiederzufinden. Gespeichert werden jeweils die Attribut-Mengen eines Tabelleneintrags in Form einer *StringMap*, zusammen mit dem zugehörigen Schlüssel. Diese bilden einen Cache-Eintrag, der durch die Datenstruktur *CacheEntry* repräsentiert wird. Um ein schnelles Auffinden der Einträge zu gewährleisten, werden zum Speichern Hash-Tabellen aus dem Modul *HashTable* verwendet. Wenn der Inhalt eines Result-Objekts im Cache gespeichert werden soll, werden unter Umständen mehrere Cache-Einträge vorgenommen, da ein Result ja auch mehrere Einträge haben kann. Demnach muss der Cache also mehrere Einträge unter dem gleichen Schlüssel verwalten können. Dazu wird die speziell für diesen Zweck implementierte Funktionalität der Hash-Tabelle benutzt, mehrere gleiche Einträge aufnehmen zu können.

Für jede Tabelle existiert ein eigener Cache, die einzelnen Caches werden über den Namen der Tabelle angesprochen. Es existiert kein Zusammenhang zwischen den einzelnen Caches. Die Funktionen zum Zugriff auf den Cache haben jeweils einen String-Parameter, mit dem ein bestimmter Cache ausgewählt werden kann. Die wichtigsten Funktionen zum Zugriff auf den Cache sind *CacheGet()* und *CachePut()*, die zum Abfragen und Einstellen von Cache-Einträgen dienen. *CacheGet()* erhält als Parameter den Tabellen-Namen und den Schlüssel, zurückgeliefert wird ein Zeiger auf eine *CacheEntry*-Struktur. Falls zu einem Schlüssel mehrere Cache-Einträge existieren, werden diese sequentiell nummeriert, beginnend bei null. Mit der Funktion *CacheGetN()* lässt sich dann ein bestimmter Cache-Eintrag aus dieser Sequenz abfragen. Dabei ist die Ordnung innerhalb der Sequenz nicht definiert, aber jeder Eintrag kommt in der Sequenz genau einmal vor. Die bei *CacheGet()* und *CacheGetN()* zurückgelieferten Zeiger auf die *CacheEntry*-Datenstruktur zeigen auf dynamisch allozierten Speicher, der nach der Benutzung vom Aufrufer wieder freigegeben werden muss. Dazu dient die Funktion *CacheEntryFree()*. Die Menge und die Namen der Caches sind nicht im voraus festgelegt, vielmehr werden die Caches nach Bedarf dynamisch angelegt. Ein neuer Cache wird immer dann erzeugt, wenn ein Tabellen-Name angegeben wird, zu dem noch kein Cache existiert.

Damit ein Cache im Laufe der Zeit nicht immer weiter wächst, wird die Anzahl der Einträge pro Cache begrenzt. Ist die maximale Anzahl erreicht, werden keine neuen Einträge mehr angenommen. Die maximale Anzahl wird am Anfang der Datei Cache.c mit dem Makro CACHE_MAX_ENTRIES definiert. Außerdem erhalten Cache-Einträge eine bestimmte Lebensdauer, die als Attribut mit dem Namen "ttl" (*time to live*) gespeichert wird. Dieses Attribut zeigt, als String kodiert, wieviele Sekunden der Eintrag noch gültig ist. In regelmäßigen zeitlichen Abständen werden nun alle Cache-Einträge durchlaufen und der Lebensdauer-Wert verringert. Sinkt der Wert auf oder unter null, wird der Eintrag entfernt. Wenn die Anzahl der Cache-Einträge über einen bestimmten Schwellenwert steigt, definiert als CACHE_HIGH_WATER_MARK, wird diese Alterung der Cache-Einträge künstlich beschleunigt, um wieder Platz im Cache zu schaffen, da es als günstiger angesehen wird, wenn Cache-Einträge, deren Lebensdauer schon relativ niedrig ist, aus dem Cache entfernt werden, als wenn neue Cache-Einträge nicht mehr an-

genommen werden. Die für diese Prozedur verantwortlichen Funktionen sind *CacheCleanup()*, mit der ein einzelner Cache "aufgeräumt" wird, und *CacheCleanupAll()*, mit denen alle Caches entsprechend bearbeitet werden.

Um Einträge aus dem Cache zu entfernen, stehen die zwei Funktionen *CacheRemove()* und *CacheRemoveAll()* zur Verfügung. Die Funktion *CacheRemove()* entfernt einen Cache-Eintrag anhand des angegebenen Schlüssels. Existieren mehrere Einträge mit dem gleichen Schlüssel, wird nur einer (ein beliebiger) der Einträge entfernt. Die Funktion *CacheRemoveAll()* ist komplexer. Hier kann nicht nur ein Schlüssel als Parameter angegeben werden, sondern auch ein Attribut und ein Attribut-Wert. Dabei werden zunächst alle Cache-Einträge gesucht, die den angegebenen Schlüssel haben. Dann werden davon diejenigen Einträge gelöscht, die das angegebene Attribut mit dem angegebenen Wert enthalten. Diese Funktionalität wird z.B. dazu genutzt, um alle Cache-Einträge, die von einem bestimmten Namensdienst stammen, zu entfernen, um die Einträge mit neuen Informationen aus diesem Namendienst zu aktualisieren.

7.3.7 Events

Dieses Modul, das sich in den Dateien *Events.c* und *Events.h* befindet, bildet den Grundstein zur eventorientierten Programmierung. Hier werden Ein-/Ausgabe-Operationen auf Ereignisse abgebildet. Wenn die Möglichkeit besteht, von einem File-Deskriptor Daten zu lesen (ohne zu blockieren), wird dafür ein Event erzeugt. Ebenso kann man ein Event erzeugen lassen, wenn ein File-Deskriptor bereit ist, Daten zum Schreiben aufzunehmen. Zur Verarbeitung von Events kann ein Aufrufer eine Funktion (*event handler*) registrieren lassen, die beim Eintreten der Ereignisses auf dem gewünschten File-Deskriptor aufgerufen wird. Event-Handler können für jeden File-Deskriptor und für Lese- und Schreiboperationen separat definiert werden. Dazu wird die Funktion *EventHandlerAdd()* verwendet. Als Parameter gibt man hier zunächst eine der vordefinierten Event-Handler-Listen *EventsReadable* für Leseoperationen oder *EventsWritable* für Schreiboperationen an. Der zweite Parameter ist ein Zeiger auf eine Datenstruktur vom Typ *EventHandler*, die den gewünschten File-Deskriptor und die aufzurufende Funktion angibt. Außerdem kann hier noch ein generischer Zeiger angegeben werden, der der Event-Handler-Funktion als Parameter übergeben wird. Dies kann vom Aufrufer genutzt werden, um kontextbezogene Daten für den Event-Handler zu speichern. Dies wird zum Beispiel genutzt, um dem Event-Handler des *Request*-Moduls mitzuteilen, welchem *Request*-Objekt er zugeordnet ist. Der Inhalt der *EventHandler*-Datenstruktur wird von der *EventHandlerAdd*()-Funktion kopiert, das heißt, die Variable, die vom Aufrufer als Argument verwendet wird, kann nach dem Aufruf verworfen werden. Die Funktion *EventHandlerRemove()* dienst analog zum Entfernen eines Event-Handlers.

Von zentraler Bedeutung ist schließlich die *EventLoop()*-Funktion. Dies ist die Event-Schleife, die das gesamte Programm steuert. Hier wird mit dem *select()*-Systemaufruf festgestellt, welche File-Deskriptoren lese- und schreibbereit sind und die entsprechenden Event-Handler aufgerufen. Außerdem werden mit Hilfe des *Timeout*-Moduls zeitgesteuerte Ereignisse abgearbeitet. Die *EventLoop()*-Funktion läuft so lange, bis die *EventLookBreak()*-Funktion aufgerufen wird.

In diesem Fall wird die Event-Verarbeitung bei der nächsten Gelegenheit beendet und die *EventLoop()*-Funktion verlassen.

7.3.8 Timeout

Das *Timeout*-Modul realisiert zeitgesteuerte Ereignisse. Die Implementierung befindet sich in den Dateien *Timeout.h* und *Timeout.c*. Ähnlich wie im *Events*-Modul werden verschiedene Timeouts verwaltet, wobei nach dem Ablauf der entsprechenden Zeitspanne eine vom Aufrufer angegebene Funktion aufgerufen wird. Timeouts sind periodisch, das heißt, nach dem Eintreten eines Timeouts wird dieser nicht gelöscht, sondern tritt nach erneutem Ablauf der Zeitspanne wieder auf. Dies geschieht so lange, bis der Timeout explizit gelöscht wird. Zum Aktivieren eines Timeouts wird die Funktion *TimeoutAdd()* verwendet. Hier müssen als Parameter die aufzurufende Funktion, die gewünschte Zeitspanne in Sekunden (auch Bruchteile von Sekunden sind möglich) und wieder ein beliebiger generischer Zeiger angegeben werden. Die *TimeoutAdd()*-Funktion liefert einen Zeiger auf ein Objekt des Typs *Timeout* zurück, welches zur Identifizierung dieses Timeouts dient. Die beim Timeout aufzurufende Funktion bekommt dieses *Timeout*-Objekt als Parameter, in dem dann auch der generische Zeiger enthalten ist. Wie im *Request*-Modul kann dieser generische Zeiger vom Aufrufer dazu verwendet werden, um speziell auf diesen Timeout bezogene Daten zu speichern. Das *Timeout*-Objekt dient schließlich auch dazu, um einen Timeout mit der *TimeoutRemove()*-Funktion wieder zu entfernen. Die tatsächliche Realisierung der Timeouts geschieht in Kooperation mit der *EventLoop()*-Funktion durch den Intervall-Timer des Systems und die Systemfunktion *setitimer()*. Dabei wird der Realzeit-Timer (*ITIMER_REAL*) verwendet, dementsprechend wird vom System beim Ablauf des Timers ein *SIGALRM*-Signal erzeugt, was in der *EventLoop()*-Funktion entsprechend behandelt wird.

Da nur ein System-Timer verwendet wird, hat das Timeout-Modul auch die Aufgabe, die verschiedenen Timeouts auf einen einzigen Timer abzubilden. Zum Ausrechnen des jeweils kürzesten Timeouts und zur Bearbeitung aller Timeouts nach einer bestimmten verstrichenen Zeitspanne dienen die Funktionen *TimeoutNext()* und *TimeoutProcess()*. Die Funktion *TimeoutStart()* wird schließlich verwendet, um den System-Timer mit dem jeweils aktuellen Timeout zu starten. Der Ablauf geschieht dabei wie folgt: Zuerst wird die Funktion *TimeoutStart()* aufgerufen. Hier wird wiederum mit *TimeoutNext()* bestimmt, welches das nächste (zeitlich kürzeste) zu verarbeitende Timeout ist. Dann wird der System-Timer gestartet und *TimeoutStart()* kehrt zum Aufrufer zurück. Der Aufrufer ist nun selbst dafür verantwortlich, dass nach dem Eintreffen des *SIGALRM*-Signals die Funktion TimeoutProcess() aufgerufen wird, in der dann alle Timeout-Funktionen aufgerufen werden, deren Zeitspanne abgelaufen ist. Tatsächlich kann der Aufrufer die *TimeoutProcess()*-Funktion auch zu anderen Zeitpunkten aufrufen. Entscheidend ist dabei, dass dabei als Parameter die seit dem Aufruf von *TimeoutStart()* verstrichende Zeit angegeben wird, damit alle Timeout-Zähler korrekt aktualisiert werden können. Die *EventLoop()*-Funktion installiert einen Signal-Handler für das *SIGALRM*-Signal und konfiguriert diesen so, das Systemaufrufe, also hier der *select()*-Aufruf in *EventLoop()*) durch das Signal unterbrochen werden. Dementsprechend wird *TimeoutProcess()* immer nach Ablauf der in *TimeoutNext()* berechneten Zeitspanne aufgerufen.

Der Einfachheit halber und Aufgrund der Tatsache, dass hier keine besondere Genauigkeit erforderlich ist, existieren zwei Stufen der Genauigkeit. Ist die eingestellte Timeout-Spanne größer als eine Sekunde, wird das aktuelle Timeout nur mit einer Genauigkeit von einer Sekunde berechnet. Ist die Spanne kleiner als eine Sekunde, wird das aktuelle Timeout in Mikrosekunden berechnet. Dieses Verfahren ist einfach und effizient und deckt sowohl kurze Timeouts von z.B. einer zehntel Sekunde als auch längere Timeouts von mehreren Sekunden bis zu mehreren Stunden ab.

7.3.9 IPv6Lookup

Das Modul *IPv6Lookup* realisiert die Namensauflösung über das IPv6Lookup-Protokoll. Es ist in den Dateien *IPv6Lookup.h* und *IPv6Lookup.c* implementiert. Hier werden die Protokolloperationen durchgeführt und die Nachbarschaftsliste verwaltet. Die Schnittstelle zum Rest des Programms besteht hauptsächlich aus der Funktion *IPv6LookupQuery()*, die bei einer Anfrage aus der *DBQuery()*-Funktion heraus aufgerufen wird. Dabei werden Anfragen für die Tabellen *hosts.byname* (Namensauflösung) und *hosts.byaddr* (Rückwärtsauflösung) unterstützt. Außerdem muss die Funktion *IPv6LookupInit()* bei Programmstart einmal aufgerufen werden. Bei Programmende muss die Funktion *IPv6LookupShutdown()* einmal aufgerufen werden.

Die übrigen Bestandteile des IPv6Lookup-Moduls sind in drei Teile strukturiert: Die Verwaltung der Nachbarschaftsliste, das Senden und Empfangen von *Query*- und *Report*-Paketen sowie die Verwaltung und Bearbeitung der ausstehenden Anfragen.

Die Nachbarschaftsliste Die Elemente der Nachbarschaftsliste werden in der Struktur *IPv6-LookupEntry* gespeichert. Hier sind Name, Service-Klasse, Adresse und Kommentar einer Dienstinstanz in Textform gespeichert. Zusätzlich enthält ein *IPv6LookupEntry*-Objekt einen Zeitstempel und eine Variable *state*, die den Zustand des Dienstes, also aktiv oder inaktiv, kennzeichnet. Mit der Funktion *IPv6LookupAddEntry()* kann ein Eintrag zur Nachbarschaftsliste hinzugefügt werden. Dabei wird von der als Parameter übergebenen *IPv6LookupEntry*-Struktur eine Kopie angelegt und diese Kopie in die Liste eingetragen. Das als Parameter übergebene Objekt wird dann nicht mehr benötigt und kann wieder freigegeben werden. Zum Durchlaufen der Nachbarschaftsliste können die Funktionen *IPv6LookupFirstEntry()* und *IPv6LookupNextEntry()* genutzt werden. Diese Funktionen erhalten einen Zeiger auf einen Listenzeiger als Parameter. Dieser Listenzeiger, der vom Aufrufer bereitgestellt werden muss, dient dazu, die aktuelle Position in der Liste zu speichern. *IPv6LookupFirstEntry()* liefert dann den ersten Eintrag der Liste. *IPv6LookupNextEntry()* liefert dementsprechend den jeweils nächsten Eintrag der Liste. Beide Funktionen liefern NULL zurück, wenn es keine weiteren Einträge oder überhaupt keinen Eintrag gibt. Während die Liste durchlaufen wird, dürfen keine Elemente entfernt oder hinzugefügt werden, insbesondere ist ein Aufruf von *IPv6LookupRemove()* nicht erlaubt. Mit der Funktion *IPv6LookupFindEntry()* kann ein einzelner Eintrag anhand von Name und Service-Klasse in der Liste gesucht werden. Die gesuchte Klasse und der gesuchte Name werden dabei als String-Parameter übergeben. Einer der beiden Parameter darf auch ein NULL-Zeiger sein, in diesem Fall wird nur nach dem passenden Namen oder Service-Klasse gesucht. Man kann also einen

Namen mit beliebiger Service-Klasse oder einen Eintrag mit einer bestimmten Service-Klasse und beliebigem Namen suchen. Falls es mehrere passende Einträge gibt, wird ein beliebiger davon als Ergebnis zurückgeliefert (tatsächlich handelt es sich im den ersten passenden Eintrag). Das Löschen eines Eintrags erledigt schließlich die Funktion *IPv6LookupEntry()*. Sie erhält als Parameter einen Zeiger auf ein *IPv6LookupEntry*-Objekt. Diesen Zeiger muss der Aufrufer vorher durch den Aufruf von *IPv6LookupFirstEntry()*, *IPv6LookupNextEntry()* oder *IPv6LookupFindEntry()* erhalten haben, es ist also ein Zeiger auf ein tatsächliches Element der Nachbarschaftsliste erforderlich.

Der Zeitstempel im *IPv6LookupEntry*-Objekt wird dazu verwendet, um die Lebensdauer der Objekte zu begrenzen. In der Funktion *IPv6LookupCleanEntries()* werden regelmäßig alle Einträge der Nachbarschaftsliste auf das Alter ihres Zeitstempels überprüft. Ab einem gewissen Alter wird ein Eintrag als inaktiv markiert. Dieses Alter wird über das Makro *CLEANING_TIME* bestimmt und ist als $CLEANING_TIME \cdot 2$ definiert. Wenn dann ein inaktiver Eintrag ein größeres Alter als den Wert *MAX_AGE* besitzt, wird er aus der Nachbarschaftsliste gelöscht. Damit werden Hosts, die längere Zeit keinen periodischen Report mehr senden, als inaktiv markiert und inaktive Hosts nach einiger Zeit gelöscht. Ein Host muss also periodische Reports senden, um den Zeitstempel seines Nachbarschaftslisten-Eintrags aufzufrischen und in der Liste zu verbleiben. Durch diese Mechanismen ist gesichert, dass die Nachbarschaftsliste sich nicht im Laufe der Zeit mit veralteten Einträgen füllt. Es muss nur noch dafür gesorgt werden, dass die *IPv6LookupCleanEntries()*-Funktion regelmäßig aufgerufen wird. Deshalb wird ein Timeout mit der Zeitspanne *CLEANING_TIME* eingerichtet, das die Funktion *cleanTimeout()* als Timeout-Handler erhält. Hier wird dann *IPv6LookupCleanEntries()* aufgerufen. Außerdem wird dieser Timeout-Handler noch dazu benutzt, um periodische Reports zu senden. Dazu wird die weiter unten beschriebene Funktion *sendReportToAll()* verwendet.

Empfangen und Senden von IPv6Lookup-Paketen Der nächste Teil des *IPv6Lookup*-Moduls befasst sich mit der Kommunikation über das IPv6Lookup-Protokoll selbst. Dabei gibt es aus technischer Sicht drei Funktionsblöcke: Das Empfangen von IPv6Lookup-Paketen, das Senden von Reports und das Senden von Queries.

Empfang von IPv6Lookup-Paketen Der Empfang von IPv6Lookup-Paketen beinhaltet die Verarbeitung von Report- und Query-Paketen. Dies geschieht zunächst in der Funktion *receive-Handler()*, die als Event-Handler für Lese-Events auf dem IPv6Lookup-Socket eingerichtet wird. Hier wird das eingetroffene Paket gelesen und festgestellt, ob es sich um ein Query-Paket oder ein Report-Paket handelt. Für Query-Pakete wird dann die Funktion *queryReceived()* aufgerufen. Bei Report-Paketen werden hier auch die einzelnen Zeilen extrahiert und für jeden einzelnen Report die Funktion *reportReceived()* aufgerufen. In der Funktion *queryReceived()* läuft nun die weitere Verarbeitung von empfangenen Queries ab. Zunächst wird das Paket analysiert und die einzelnen Felder der Dienstespezifikation geparst. Da die gegenwärtige Implementierung nur die Service-Klasse “host” unterstützt, werden Queries mit einer anderen Service-Klasse ignoriert. Dann wird überprüft, ob der gesuchte Name mit dem eigenen übereinstimmt und im positiven

Fall ein Report-Paket als Antwort gesendet. Für den Empfang eines Query-All-Pakets ist hier ein Spezialfall vorgesehen. Die Antwort auf ein Query-All-Paket soll laut Protokollspezifikation verzögert gesendet werden, also kann der Antwort-Report nicht in dieser Funktion gesendet werden. Statt dessen wird ein Timeout eingestellt, bei dessen Eintreten die Funktion *deferredResponse()* aufgerufen wird. Die Zeitspanne wird dabei zufällig gewählt, der mögliche Maximalwert wird durch die globale Variable *reportDelay* angeben (in Sekunden). Standardmäßig ist hier der Wert 2 eingestellt. Die Funktion *reportReceived()* handelt die Verarbeitung von einzelnen Reports ab, was vor allem das Aktualisieren der Nachbarschaftsliste beinhaltet. Auch hier wird zunächst der Report-String in seine Bestandteile zerlegt. Dann wird in der Nachbarschaftsliste nach einem Eintrag mit demselben Namen und derselben Service-Klasse gesucht. Falls ein solcher Eintrag gefunden wurde, wird er gelöscht, damit keine mehrfachen Einträge entstehen. Dann wird ein neuer Eintrag in die Nachbarschaftsliste mit den gerade erhaltenen Informationen vorgenommen. Abschließend muss noch die Liste der ausstehenden Anfragen überprüft werden, die weiter unten genauer beschrieben wird. Dies ist für Anfragen notwendig, die nicht aus der Nachbarschaftsliste beantwortet werden können und für die ein Query-Paket gesendet wurde.

Senden von Reports Das Senden von Reports wird von den Funktionen *sendReport()* und *sendReportToAll()* implementiert. Das Codieren eines Report-Pakets ist prinzipiell sehr einfach und geschieht mit Hilfe der *sprintf()*-Funktion aus der Standard-C-Bibliothek. Das ist jedoch nur ein Teil der Aufgaben der *sendReport()*-Funktion. Es besteht ein gewisses Problem darin, den abzusendenden Report mit der korrekten Absenderadresse zu versehen. Zwei Gesichtspunkte sind dabei zu beachten. Erstens kann ein Host verschiedene Netzwerk-Interfaces haben, die jeweils unterschiedliche Adressen besitzen. Es muss also festgestellt werden, auf welches Interface der Report gesendet werden soll (d.h. auf das gleiche Interface, von dem die entsprechende Query empfangen wurde). Zweitens ist es wünschenswert, dass ein Host in dem Report nicht seine Link-Local-Adresse sendet, falls er eine weitere IPv6-Adresse besitzt, die durch manuelle Konfiguration, *Stateless Autoconfiguration* oder andere Mechanismen eingerichtet wurde. Da Reports an eine Link-Local-Multicast-Adresse gesendet werden, ist die Quelladresse des Pakets die Link-Local-Adresse des sendenden Hosts. Würde man als Adressangabe im Report einfach das "."-Zeichen angeben, würde der Empfänger also die Link-Local-Adresse erhalten. Wenn der Host aber eine weitere Adresse außer der Link-Local-Adresse besitzt, sollte er diese explizit in seinem Report verwenden, damit in den Nachbarschaftslisten die "echten" IPv6-Adressen anstelle der Link-Local-Adressen gespeichert werden. Beide Teilprobleme werden dadurch gelöst, indem der *sendReport()*-Funktion eine Socket-Adressen-Struktur als Parameter übergeben werden kann, die die Adresse anzeigt, die als Quelladresse des zu sendenden Report-Pakets verwendet werden soll. Da die verschiedenen Netzwerk-Interfaces unterschiedliche Adressen haben, kann man damit darstellen, welches Netzwerk-Interface zum Senden verwendet werden soll. Diese Information wird dem Kernel vor dem Senden mit dem *bind()*-Systemaufruf mitgeteilt. Indem man einen Datagramm-Socket, der zum Schreiben geöffnet ist, an eine spezielle lokale Adresse bindet, kann man implizit das Netzwerk-Interface wählen, über das gesendet werden soll. Die Zieladresse kann hier nicht zur Bestimmung des Interfaces verwendet werden, da es sich um eine allgemeingültige (auf allen Interfaces gleiche) Multicast-Adresse handelt. Schon in der

receiveHandler()-Funktion wird die Quelladresse eines eintreffenden Query-Paketes überprüft. Handelt es sich um eine link-lokale Adresse, wird über die in der Adressen-Struktur enthaltenen *scope-id* und die *if_indextoname()*-Funktion der Name des Netzwerk-Interfaces bestimmt, auf dem das Paket empfangen wurde. Dann wird die Konfigurationstabelle aller Netzwerkinterfaces vom Kernel abgefragt[1] und darin eine IPv6-Adresse gesucht, die für das entsprechende Interface konfiguriert ist und keine link-lokale Adresse ist. Wurde eine solche Adresse gefunden, wird sie als Parameter für die *sendReport()*-Funktion verwendet, womit sowohl im Report-Paket die reale IPv6-Adresse steht, als auch gleichzeitig das korrekte Netzwerk-Interface zum Senden des Pakets verwendet wird.

Wenn ein Report-Paket automatisch gesendet werden soll, entweder beim Systemstart oder periodisch, existiert jedoch kein zugehöriges Query-Paket, aus dem die Adresse abgeleitet werden könnte. Zudem müssen hier auch Report-Pakete auf *alle* vorhandenen Interfaces geschickt werden[2], wobei für jedes Interface eine andere Quelladresse verwendet werden muss. Diese Aufgabe erledigt die Funktion *sendReportToAll()*. Hier wird wie oben die Interface-Konfigurationstabelle festgestellt und für jedes Interface (mit Ausnahme von Loopback- und Point-to-Point-Interfaces) die eigene Adresse festgestellt. Mit dieser Adresse wird dann *sendReport()* aufgerufen. Dadurch wird an alle angeschlossenen Netze ein Report gesendet.

Senden von Queries Query-Pakete müssen dann gesendet werden, wenn auf Anfrage einer Anwendung in der *IPv6LookupQuery()*-Funktion kein passender Eintrag in der Nachbarschaftsliste gefunden wurde. Dies übernimmt die Funktion *sendQuery()*, die als Parameter die gesuchte Service-Klasse und den gesuchten Namen erhält. Die *sendQuery()*-Funktion ist eher einfach und kurz. Das Codieren des Query-Pakets ist wie bei Reports sehr einfach und geschieht mit einem simplen *sprintf()*-Aufruf. Anschließend wird das Paket an die Multicastadresse für alle lokalen Knoten (ff02::1) geschickt. Der Aufruf von *sendQuery()* geschieht dabei nicht in *IPv6LookupQuery()* selbst, sondern beim Hinzufügen einer Anfrage zur Liste der ausstehenden Anfragen, wie sie weiter unten beschrieben wird. Weiterhin wird *sendQuery()* in *IPv6LookupInit()* benutzt, um das anfängliche Query-All-Paket zu senden. Dies wird erreicht, indem als Name der String "*" übergeben wird.

Ausstehende Anfragen Wenn eine Anwendung eine Anfrage stellt, wird im *IPv6Lookup*-Modul die Funktion *IPv6LookupQuery()* aufgerufen. Falls die Anfrage über die Nachbarschaftsliste beantwortet werden kann, wird mit Hilfe der Funktion *IPv6LookupReply()* der passende Eintrag herausgesucht und eine Antwort gesendet. Für Rückwärtsauflösung wird die Funktion *reverseLookupReply()* verwendet, die grundsätzlich nur die Nachbarschaftsliste konsultiert.

[1] Die derzeitige Implementierung verwendet dazu einen *SIOCGIFCONF-ioctl*-Aufruf, wie er in [LFJL93] aufgeführt ist. Die Verwendung von IPv6-Adressen verursacht dabei einige Unannehmlichkeiten bei der Programmierung. Nach Angaben der FreeBSD-Entwickler auf einer Developer-Mailingliste sollte stattdessen ein *sysctl()*-Aufruf oder die *getifaddrs()*-Funktion verwendet werden.

[2] In den meisten Fällen wird ein Host außer den "uninteressanten" Interfaces wie Loopback- oder Point-To-Point-Interfaces nur ein einziges Netzwerkinterface haben.

Wenn in *IPv6LookupReply()* kein passender Eintrag gefunden wurde, wird dies durch einen bestimmten Rückgabewert (null) angezeigt. In diesem Fall muss ein Query-Paket gesendet und ein passender Report abgewartet werden.

Da die Funktion *IPv6LookupQuery()* nicht auf die Antwort eines gesendeten Query-Pakets warten darf, müssen noch nicht beantwortete Anfragen außerhalb von *IPv6LookupQuery()* gesichert werden und zu einem späteren Zeitpunkt in Event- oder Timeout-Handlern weiterverarbeitet werden. Dazu wird eine Liste von noch nicht bearbeiteten *Request*-Objekten angelegt. Mit der Funktion *IPv6LookupAddRequest()* wird ein Zeiger auf ein *Request*-Objekt zu dieser Liste hinzugefügt. Dabei wird dann auch ein entsprechendes Query-Paket gesendet. Mit der Funktion *IPv6LookupRemoveRequest()* kann ein *Request*-Objekt wieder aus der Liste entfernt werden. Die Funktion *IPv6LookupFinishRequests()* dient nun dazu, diese Liste der ausstehenden Anfragen zu bearbeiten. Dabei wird für jedes *Request*-Objekt aus der Liste geprüft, ob der gesuchte Eintrag in der Nachbarschaftsliste enthalten ist. Wenn dies der Fall ist, wird die entsprechende Antwort gesendet (mit *IPv6LookupReply()*) und das *Request*-Objekt aus der Liste der ausstehenden Anfragen entfernt. Diese Überprüfung der ausstehenden Anfragen muss jedesmal beim Empfangen eines Reports ausgeführt werden, da bei jedem neu eingetroffenen Report die Möglichkeit besteht, dass er den in einer ausstehenden Anfrage gesuchten Namen enthält. Im Normalfall wird ein derartiger Report natürlich durch eine entsprechende vorher gesendete Query ausgelöst worden sein, was aber nicht unbedingt der Fall sein muss. Dementsprechend wird in der *reportReceived()*-Funktion auch *IPv6LookupFinishRequests()* aufgerufen.

Für den Fall, dass auf ein gesendetes Query-Paket nicht in annehmbarer Zeit mit einem Report-Paket geantwortet wird, wird vor dem Senden des Query-Pakets ein Timeout-Handler eingerichtet. In dem Timeout-Handler wird das Query-Paket erneut gesendet und nach zwei Wiederholungen, also insgesamt drei Query-Paketen, die Anfrage beendet. Da der für einen Timeout-Handler frei wählbare Parameter schon für das zugehörige *Request*-Objekt verwendet wird, gibt es keine einfache Möglichkeit, einen Zähler für die Wiederholungen zu benutzen. Daher werden nacheinander mehrere verschiedene Funktionen als Timeout-Handler verwendet, nämlich *IPv6LookupTimeout()*, *IPv6LookupSecondTimeout()* und *IPv6LookupFinalTimeout()*. Die ersten beiden Funktionen setzen ein neues Timeout fest und senden das Query-Paket erneut, die letzte Funktion übergibt dann die Kontrolle über das *Request*-Objekt wieder an das *DB*-Modul, indem es die Funktion *DBResumeQuery()* aufruft.

Zu erwähnen ist noch eine besondere Funktion von *IPv6LookupReply()*. Wird als Suchschlüssel der String “*” übergeben (nur für die Tabelle *hosts.byname*), wird als Antwort die komplette aktuelle Nachbarschaftsliste gesendet. Dazu dient die Funktion *IPv6LookupSendAll()*. Dies ist für Anwendungen gedacht, die dem Benutzer eine Liste der im Netz vorhandenen Hosts oder Dienste präsentieren wollen.

7.3.10 DNSLookup

Das Modul *DNSLookup*, das sich in den Dateien *DNSLookup.c* und *DNSLookup.h* befindet, implementiert Namensdienst-Anfragen an DNS. Die einzige nach außen sichtbare Funktion dieses

Moduls ist *DNSLookupQuery()*, die aus dem *DB*-Modul aufgerufen wird. Bei der Implementierung dieses Moduls waren zwei Besonderheiten zu beachten.

Erstens war es nicht möglich, die üblichen Funktionen zur Namensauflösung, wie *gethostbyname()* oder *getaddrinfo()* zu verwenden, da diese ja unter der neuen Namensdienst-Architektur dann wiederum eine Anfrage an den *lookupd*-Dæmon starten würden, was zu einer endlosen Rekursion führen würde. Also musste eine eigene Version dieser Funktionen implementiert werden. Dies ist in den Funktionen *dnsSimpleLookup()* und *dnsSimpleRevLookup()* geschehen. Der Code dieser Funktionen wurde größtenteils aus dem Quelltext der Originalfunktionen aus der C-Bibliothek übernommen.

Zweitens besteht das Problem, dass die Funktionen, die von der Systembibliothek zum direkten Senden von DNS-Anfragen zur Verfügung gestellt werden (z.B. *res_query()*), nicht für die eventorientierte Programmierung geeignet sind, da sie auf die Antwort des DNS-Servers (bis zu einer gewissen Zeitspanne) warten. Wie in Abschnitt 7.2 erläutert, darf ein solches Verhalten nicht vorkommen. Da die vollständige Neuimplementierung der DNS-Resolver-Bibliotheken nicht anzustreben ist[3], wurde das Problem umgangen, indem für den Aufruf der Resolver-Funktionen ein eigener Prozess gestartet wird. Dies geschieht in der Funktion *dnsFind()*. Dazu wird zuerst eine *Pipe* eingerichtet und dann mit *fork()* der Dæmon-Prozess dupliziert. Der neue Prozess (*Child*-Prozess*)* führt nun die Namensauflösung mit *dnsSimpleLookup()* oder *dnsSimpleRevLookup()* durch, während der ursprüngliche Prozess (der *Parent*-Prozess) zum Aufrufer und damit letztendlich zur Event-Schleife des Hauptprogramms zurückkehrt, damit weitere Events verarbeitet werden können. Vorher wurde aber im Parent-Prozess ein Event-Handler für die Pipe eingerichtet (siehe Funktion *pipeReadHandler()*). Wenn nun im Child-Prozess die Resolver-Funktion zurückkehrt, entweder mit einem Ergebnis oder mit einem Fehler, wird das Ergebnis über die Pipe an den Parent-Prozess gesendet, was dort ein Ereignis und den Aufruf von *pipeReadHandler()* auslöst. In der *pipeReadHandler()*-Funktion wird das Ergebnis dann gelesen, in den Cache eingetragen und dann die Anfrage der Anwendung mit *RequestQueryReply()* beantwortet. Lag kein Ergebnis vor (d.h. das Ergebnis war leer), so wird hier auch die *DBResumeQuery()*-Funktion aufgerufen, um die anderen Namensdienst-Module zum Zuge kommen zu lassen. Zum Übertragen des Ergebnisses über die Pipe werden die Funktionen *ResultSend()* und *ResultReceive()* verwendet, die auch zur Kommunikation mit den Anwendungen dienen. Die Funktion *ResultReceive()* wird normalerweise nur auf der Anwendungsseite verwendet und ist eigentlich nicht geeignet, um im *lookupd*-Dæmon selbst verwendet zu werden, weil sie so lange blockiert, bis das Ergebnis vollständig empfangen wurde, was der eventorientierten Programmierung widerspricht. In diesem speziellen Fall führt das jedoch nicht zu Problemen, weil bekanntermaßen in dem Child-Prozess immer das komplette Ergebnis gesendet wird. Da der Child-Prozess ja durch den Dæmon-Prozess erzeugt wurde, er also gewissermaßen “mit sich selbst” kommuniziert, kann hier dem Kommunikationspartner am anderen Ende der Pipe vertraut werden, dass er sich korrekt verhält.

[3] Eine andere Möglichkeit wäre die Verwendung einer asynchronen DNS-Bibliothek. Mit *adns* von Ian Jackson ist eine derartige Bibliothek auch verfügbar (siehe `http://www.chiark.greenend.org.uk/~ian/adns/`). Leider konnte sie nicht eingesetzt werden, da sie nur IPv4 unterstützt und unter der GPL-Lizenz veröffentlicht wird.

Zu bemerken ist noch, dass das *DNSLookup*-Modul das Abfragen der Tabellen *hosts.byname* für die Auflösung von Hostnamen in Adressen sowie *hosts.byaddr* für die Rückwärtsauflösung implementiert. DNS kann auch noch andere Informationen als Hostnamen und IP-Adressen speichern, wie z.B. im *Hesiod*-System, bei dem Benutzerinformationen über DNS verwaltet werden. Eine derartige Verwendung ist jedoch nicht sehr verbreitet, daher wurde hier nur die Namensauflösung für IP-Adressen berücksichtigt. Bei einer Anfrage an das *DNSLookup*-Modul können als Ergebnisse sowohl IPv4- als auch IPv6-Adressen zurückgeliefert werden. Es wird immer nach Einträgen für beide Adressfamilien gesucht. In diesem Zusammenhang stellt die hier reimplementierte Funktionalität von *dnsSimpleLookup()* eine kleine Verbesserung zu der entsprechenden in der C-Bibliothek vorhandenen Funktion dar. In *dnsSimpleLookup()* wird eine DNS-Anfrage mit dem Query-Typ *ANY* gesendet, die mit einer einzigen Antwort (falls vorhanden) IPv4- und IPv6-Adressen liefert. Die Funktionen der C-Bibliothek senden zuerst eine DNS-Anfrage nach *AAAA-Records* (also IPv6-Adressen), und erst nachdem diese Anfrage beantwortet wurde, wird eine erneute DNS-Anfrage nach *A-Records* (also IPv4-Adressen) gesendet. Dies dauert, insbesondere bei einem langsamen DNS-Server, erheblich länger als eine einzelne Anfrage mit dem *ANY*-Typ. Ein kleiner Nachteil einer ANY-Anfrage ist, dass der DNS-Server bei einem Alias-Namen (Typ *CNAME*) nicht den echten Adressen-Record mitliefert, was bei einer Anfrage nach *A*- oder *AAAA*-Records automatisch geschieht. Demnach muss die *dnsSimpleLookup()*-Funktion beim Entdecken eines *CNAME*-Records eine zweite DNS-Anfrage senden.

7.3.11 lookupd

Das Modul *lookupd* ist das Hauptmodul des *lookupd*-Dæmons, das auch die *main()*-Funktion enthält. Es ist in der Datei *lookupd.c* implementiert. Hier werden viele Initialisierungsaufgaben übernommen und die Event-Hauptschleife gestartet. Das Erzeugen der Server-Sockets, das Annehmen der Client-Sockets und das Installieren von Signal-Handlern wird von diesem Modul realisiert. Das Annehmen von Verbindungen wird von der Funktion *AcceptHandler()* übernommen, die als Event-Handler für den lokalen Server-Socket eingerichtet wird. Bei Annahme einer Verbindung wird dann für den neuen Socket ein Lese-Event-Handler mit der *RequestReadHandler()*-Funktion aus dem *Request*-Modul eingerichtet.

Die *main()*-Funktion selbst hat zunächst die Aufgabe, eventuell über die Kommandozeile übergebene Parameter zu überprüfen. Insbesondere gibt es die Möglichkeit, mit dem Parameter "-d" einen Debug-Modus zu aktivieren, der eine Reihe von verschiedenen Status- und Debug-Ausgaben beinhaltet. Dazu muss allerdings der Debugging-Code schon beim Compilieren mit dem Makro *DEBUG* aktiviert worden sein.

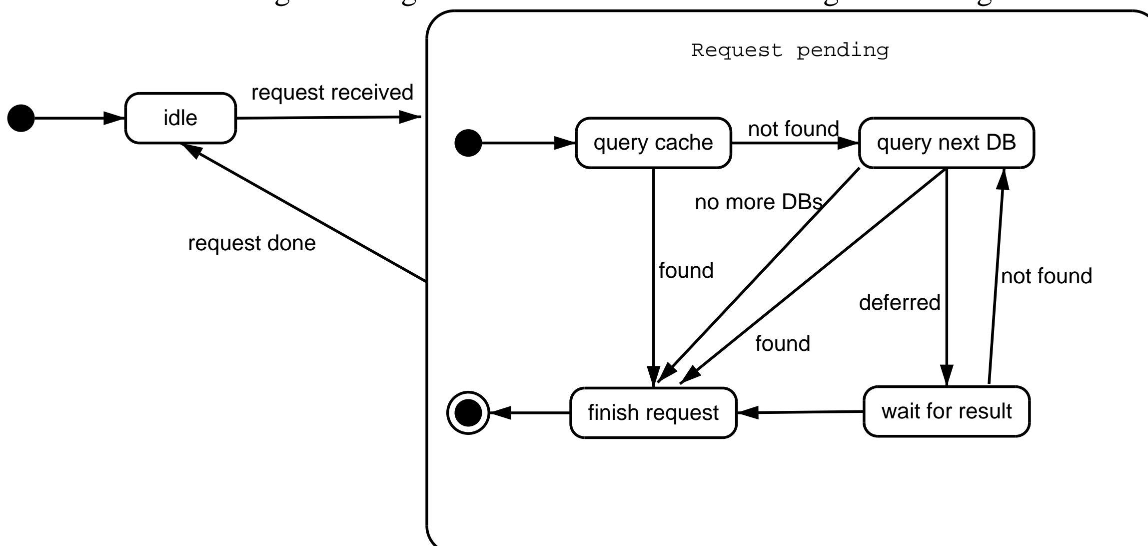
Dann wird sichergestellt, dass der Dæmon nicht bereits läuft, indem die Existenz einer besonderen Datei überprüft wird, in die ein laufender Dæmon seine Prozess-ID (*pid*) schreiben soll. Ist dies nicht der Fall, steht der weiteren Ausführung nichts mehr im Wege. Dann werden Signal-Handler für die Signale *SIGINT* und *SIGTERM* installiert, um die Beendigung des Dæmons von außen (z.B. beim Herunterfahren des Systems) sauber abzufangen. Anschließend wird der lokale Socket, über den die Anwendungen mit dem Dæmon kommunizieren sollen, erzeugt und

der oben beschriebene Event-Handler *AcceptHandler()* eingerichtet. Nun muss sich der Dæmon noch in den Hintergrund versetzen (durch den *Dæmon()*-Systemaufruf) und seine Existenz durch Schreiben des PID-Files anzeigen. Nachdem noch verschiedene Initialisierungsfunktionen von anderen Modulen aufgerufen werden, sind die Vorbereitungen abgeschlossen. Jetzt wird die Event-Hauptschleife mit *EventLoop()* aufgerufen, die so lange läuft, bis der Dæmon durch ein Signal zur Terminierung aufgefordert wird.

7.4 Programmablauf

Nachdem nun die Funktion der einzelnen Module beschrieben wurde, sollen nun die Zusammenhänge zwischen den Modulen näher betrachtet werden. Die Abläufe (*code paths*) im Dæmon sollen in diesem Abschnitt anhand von Beispielen demonstriert werden. Dabei wird beobachtet, wann wo welche Objekte erzeugt werden, durch welche Module die Ausführung läuft und wann die Objekte wieder gelöscht werden.

Abbildung 7.5: Programmablauf bei der Verarbeitung von Anfragen



1. In der *main()*-Funktion wird ein Event-Handler für das Annehmen von Verbindungen auf dem lokalen Sockets eingerichtet.

2. Die *IPv6LookupInit()*-Funktion richtet einen Event-Handler für eintreffende IPv6Lookup-Pakete ein (*readHandler()* in *IPv6Lookup.c*).

3. Bei einer ankommenden lokalen Verbindung: Ein *Request*-Objekt wird erzeugt und für den neuen Socket ein Event-Handler eingerichtet (*RequestReadHandler()* in *Request.c*).

4. Über *RequestReadHandler()* wird eine Protokoll-Request eingelesen, möglicherweise in mehreren Schritten.

5. *DBQuery()* aufrufen (Modul DB)
   (a) Cache abfragen in *DBQuery()*, *DB.c*
   (b) Reply senden (RequestReply(), *Request.c*), *Request*-Objekt “säubern”, *Result*-Objekt löschen
   (c) Abfragemodule aufrufen (*IPv6LookupQuery()* in IPv6Lookup.c, *DNSLookupQuery()* in *DNSLookup.c*)
   (d) *DBResumeQuery()* wird von den einzelnen Abfragemodulen aufgerufen
6. Abbruch der lokalen Verbindung: *Request*-Objekt wird als abgebrochen markiert (in *RequestReadHandler()*)
7. Bei einer ungültigen Request (*RequestProcess()* in *Request.c*) : Verbindung wird geschlossen, *RequestAbort()* wird aufgerufen
8. Shutdown: *IPv6Shutdown()* aufrufen, negativen Report senden, Programmende

7.5 Integration in die C-Bibliothek

Nachdem nun die Server-Seite der Implementierung beschrieben wurde, muss noch die Anbindung an die Anwendungsseite (oder Client-Seite) geschehen. Dazu wurde eine Funktion *LookupdQuery()* implementiert, die eine Anfrage über das lookupd-Protokoll sendet und das Ergebnis als ein *Result*-Objekt präsentiert. Die Implementierung dieser Funktion befindet sich in der Datei *LookupdQuery.c*. Hier befindet sich auch die Funktion *ResultReceive()*, die das Gegenstück zur Funktion *ResultSend()* aus dem Result-Modul darstellt und zum Empfangen eines kompletten *Result*-Objektes dient.

Außerdem müssen die im Abschnitt 5.1 aufgeführten Funktionen der C-Bibliothek so modifiziert werden, dass sie das lookupd-Protokoll verwenden können. Für diese Arbeit wurden aus Zeitgründen nur die Funktionen *gethostbyname2()*, *gethostbyaddr()* , *getipnodebyname()* und *getipnodebyaddr()* entsprechend modifiziert. Die *gethostbyname()*-Funktion ruft intern *gethostbyname2()* auf, ebenso wird *getipnodebyname()* von *getaddrinfo()* und *getipnodebyaddr()* von *getnameinfo()* aufgerufen. Damit ist die Namensauflösung für den Großteil aller Anwendungen abgedeckt. Die wichtigsten Änderungen der C-Bibliothek wurden in der Datei *name6.c* aus dem Verzeichnis */usr/src/sys/lib/libc/net* vorgenommen. Hier wurde eine Funktion eingefügt, die eine Namensauflösung mit Hilfe der *LookupdQuery()*-Funktion vornehmen kann und das Ergebnis in die entsprechenden Datenstrukturen umwandelt. Diese Funktion wurde *_lookupd_ghbyname()* genannt[4] und ist nur für internen Gebrauch innerhalb der C-Bibliothek bestimmt. Eine analoge Funktion *_lookupd_ghbyaddr()* wurde für die Rückwärtsauflösung implementiert. Die Funktionen *getipnodebyname()* und *getipnodebyaddr()* mussten entsprechend

[4] Funktionen in der C-Bibliothek, die nicht für Anwendungen sichtbar sein sollen, haben einen Unterstrich als erstes Zeichen im Funktionsnamen, da diese Namensgebung laut ANSI-C eben nur für solche Bibliotheksfunktionen erlaubt ist.

angepasst werden, um die eben beschriebenen neuen Funktionen zur Namensauflösung auch zu verwenden. In der Datei *name6.c*, die die Funktionen *getipnodebyname()* und *getipnodebyaddr()* enthält, existiert ein Mechanismus, über den verschiedene Funktionen zur Namensauflösung nacheinander aufgerufen werden können (ein Array von Funktionszeigern, das *_hostconf* genannt wird). Es existieren Funktionen für Dateien (also */etc/hosts*), für DNS und für NIS, die entsprechend *_files_ghbyname()*, *_dns_ghbyname()*, *_nis_ghbyname()* für die Vorwärtsauflösung und *_files_ghbyaddr()*, *_dns_ghbyaddr()* und *_nis_ghbyaddr()* für die Rückwärtsauflösung heißen. Die Reihenfolge der aufgerufenen Funktionen wird über die Konfigurationsdatei */etc/host.conf* gesteuert. Dort steht eine Sequenz von Schlüsselwörtern, die die Reihenfolge darstellen, z.B. "files dns nis"[5]. Durch die Änderungen an *name6.c* wird den Schlüsselwörtern für die Konfigurationsdatei das Schlüsselwort "*lookupd*" hinzugefügt und die Tabelle der Funktionszeiger um die Funktionen *_lookupd_ghbyname()* und *_lookupd_ghbyaddr()* erweitert. Damit sind *getipnodebyname()* und *getipnodebyaddr()* mit der neuen Funktionalität ausgerüstet worden.

Die gleiche Prozedur muss nun für die Funktionen *gethostbyname2()* und *getaddrinfo()* durchgeführt werden. Diesmal ist die Datei *gethostnamadr.c* aus dem Verzeichnis */usr/src/lib/libc/net* betroffen. Auch hier existiert wieder eine Tabelle mit aufzurufenden Funktionen, deren Reihenfolge durch die Datei */etc/hosts* bestimmt wird[6]. Allerdings wird hier in dem Array *service_order* nur eine Reihenfolge von symbolischen Konstanten festgelegt, die Wahl der tatsächlichen Funktion erfolgt später über ein *switch*-Statement. Wieder gibt es eine Reihe von Funktionen, die dann die tatsächliche Namensauflösung erledigen (*_gethostbyhtname()* und *_gethostbyhtaddr()* für Dateien, *_gethostbydnsname()* und *_gethostbydnsaddr()* für DNS, *_gethostbynisname()* und *_gethostbynisaddr()* für NIS). Um diese Reihe zu ergänzen, wurden wurden die Funktionen *_gethostbylkupdname()* und *_gethostbylkupdaddr()* implementiert. Diese Funktionen dienen nur als Wrapper für die Funktionen *_lookupd_ghbyname()* und *_lookupd_ghbyaddr()*, da die Funktionalität praktisch identisch ist, aber die Parameter sich unterscheiden. Die Implementierung von *gethostbylkupdname()* und *_gethostbylkupdaddr()* befindet sich in der (neuen) Datei *gethostbylkupd.c*, die sich im Verzeichnis */usr/src/lib/libc/net* befinden sollte.

Die eben beschriebenen *switch*-Statements in *gethostbyname2()* und *getaddrinfo()* wurden also um eine weitere Möglichkeit erweitert, bei der dann die neuen Funktionen *_gethostbylkupdname()* und *_gethostbylkupdaddr()* aufgerufen werden. Außerdem wurde dem Code zum Lesen des Konfigurationsfiles das neue Schlüsselwort "lookupd" hinzugefügt. Damit ist auch die Familie der *gethostbyname*-Funktionen mit der neuen Funktionalität ausgerüstet.

[5]Diese Schlüsselwörter können auch in verschiedenen Zeilen stehen, außerdem sind Kommentarzeilen mit '#' am Zeilenanfang erlaubt.

[6]Die Tatsache, dass hier zweimal der gleiche Mechanismus an verschiedenen Stellen implementiert wurde, hat historische Gründe. Die *getaddrinfo()/getipnodebyname()*-Funktionsfamilie stammt aus der IPv6-Implementierung, die als separates Paket zur offiziellen FreeBSD-Version entwickelt wurde. Um die bisherigen Quelltexte nicht zu modifizieren und damit eine leichtere Integration der neuen Quelltexte zu ermöglichen, wurden die neuen Funktionen unabhängig von den alten entwickelt. Da IPv6 mittlerweile Bestandteil der offiziellen FreeBSD-Version ist, ist zu erwarten, dass dieser Zustand sich ändern wird.

Kapitel 8

Tests und Leistungsmessungen

Da der *lookupd*-Dæmon eine wichtige Aufgabe im System übernimmt und permanent laufen muss, ist es wichtig, hinreichende Tests durchzuführen. Diese Tests sollen dabei nicht nur das korrekte Funktionieren nachweisen, sondern auch das Verhalten unter hoher Belastung demonstrieren.

Wichtig ist auch eine Untersuchung des IPv6Lookup-Protokolls im Netz. Besonders interessant ist hier eine Situation mit vielen aktiven Hosts, bei der das Zusammenwirken der einzelnen *lookupd*-Instanzen beobachtet werden kann. Dabei sollte auf Anomalitäten wie Lastspitzen oder Synchronisationseffekte geachtet werden. In der verfügbaren Testumgebung konnten leider nur maximal drei verschiedene Rechner verwendet werden, so dass derartige Tests nicht möglich waren. Es konnte nur die grundsätzliche Funktionalität erfolgreich überprüft werden.

Für die Tests der Software wurde ein Programm geschrieben, das in einer Schleife die Funktionen des *lookupd*-Dæmons aufruft. Dieses Testprogramm ist in der Datei *tester.c* implementiert. Es enthält zwei wesentliche Funktionen, die eine Namensauflösung bewirken. Die erste Möglichkeit besteht darin, zur Namensauflösung die *LookupdQuery()*-Funktion zu verwenden. Die zweite Möglichkeit verwendet die *gethostbyname()*-Funktion. Damit wird zusätzlich die Anbindung and den *lookupd*-Dæmon über die üblichen Funktionen der C-Bibliothek getestet.

Ferner wurde ein Programm verwendet, das nicht zum Umfang dieser Diplomarbeit gehört, mit dem direkt über die Standard-Ein- und -Ausgabe mit dem *lookupd*-Dæmon kommuniziert werden kann. Dies wurde unter anderem dazu verwendet, um zufällige Daten an den *lookupd*-Dæmon zu senden.

Die Tests wurden auf einem PC mit Pentium-MMX-Prozessor (233 MHz) und 64 MB RAM durchgeführt.

Funktionstest Um die Funktion des *lookupd*-Dæmons und des IPv6Lookup-Protokolls zu testen, wurden mit dem *tester*-Programm verschiedene Namensauflösungs-Anfragen gestellt und die Korrektheit der Antworten überprüft. Zudem wurde die C-Systembibliothek als *shared li-*

brary mit der neuen Funktionalität compiliert, so dass jedes dynamisch gelinkte Programm zur Namensauflösung die *lookupd*-Architektur benutzte.

Belastungstest Die Belastungstests bestanden darin, das *tester*-Programm in einer Endlosschleife laufen zu lassen und das Verhalten des *lookupd*-Programms zu beobachten. Ebenso wurde das Verhalten bei mehreren parallel laufenden *tester*-Prozessen untersucht. Dies ist wichtig für die Bewertung des in Kapitel 7 ausgewählten Verfahrens zur Parallelisierung der Programmabläufe. Es stellte sich heraus, dass auch dann, wenn der *lookupd*-Prozess durch mehrere parallel laufende *tester*-Prozesse stark belastet war, bei einer zusätzlichen einzelnen Namensauflösungs-Anfrage keine merkliche Verzögerung auftrat. Das lässt den Schluss zu, dass die gewählte Methode der eventbasierten Programmierung gut funktioniert.

Ein weiterer Belastungstest beinhaltete den Aufbau einer Socket-Verbindung zum *lookupd*-Prozess und dem anschließenden Senden großer Mengen zufälliger Daten. Damit sollte die Robustheit des Programmcodes getestet werden. Es zeigten sich keine Anomalitäten bei der Programmausführung.

Rechenzeit -und Speicherverbrauch Der Verbrauch an Rechenzeit und Speicher wurde während der Tests mit dem *top*-Programm gemessen. Generell ist der Speicherverbrauch des *lookupd*-Prozesses mit etwa 450KB Programmgröße und 280KB residenten Speicherseiten minimal, dabei ist zu beachten, dass das Programm statisch gelinkt wurde. Auch der Verbrauch an Rechenzeit liegt im normalen Betrieb unter einem Prozent. Der Speicherverbrauch bleibt auch während der Belastungstests stabil, so dass davon ausgegangen werden kann, dass keine Speicherlecks im Programmcode vorhanden sind. Es folgen nun einige Bildschirmdarstellungen des *top*-Programms bei verschiedenen Testphasen. Besonders interessant ist dabei die Spalte mit der Bezeichnung *CPU*, die den Anteil des jeweiligen Prozesses (in der Spalte *COMMAND*) an der insgesamt verbrauchten Rechenzeit anzeigt.

gethostbyname() über lookupd, Quelle DNS/Cache

```
last pid: 69916;  load averages:  1.45,  1.08,  0.68   up 100+00:20:21 14:11:16
52 processes:  3 running, 49 sleeping
CPU states: 40.5% user,  0.0% nice, 59.1% system,  0.4% interrupt,  0.0% idle
Mem: 34M Active, 6496K Inact, 16M Wired, 2792K Cache, 14M Buf, 1412K Free
Swap: 261M Total, 20M Used, 241M Free, 7% Inuse

  PID USERNAME PRI NICE  SIZE    RES STATE    TIME   WCPU    CPU COMMAND
55336 root      51   0   476K   284K RUN      3:40 43.95% 43.95% lookupd
69908 sleder2s  51   0   344K   208K RUN      0:48 41.42% 41.21% tester
66406 root       2   0 20528K 11760K select 191:38  4.25%  4.25% XF86_S3
69788 sleder2s   2   0  2596K  1612K select   0:02  2.49%  2.49% xterm
```

Bei diesem Testlauf wurde das *tester*-Programm so gestartet, dass es über einen *gethostbyname()*-Aufruf einen Namen auflösen sollte, der nur in der DNS-Datenbank vorhanden war. Zu beachten ist dabei, dass nach der ersten Anfrage alle weiteren Anfragen aus dem Cache des *lookupd*-Dæmons beantwortet werden. Es ist deutlich zu sehen, dass die Rechenzeit zu jeweils etwa

40% vom *lookupd*-Prozess und vom *tester*-Prozess verbraucht wird. Da die Hauptschleife des *tester*-Programms sehr einfach ist und die Anteile an CPU-Zeit trotzdem beinahe gleich sind, kann man daraus ableiten, dass die Bearbeitung der Anfrage im *lookupd* effizient ist. Ebenso ist anzumerken, dass die insgesamt verbrauchte CPU-Zeit zu etwa 60% auf den Kernel fällt (dritte Zeile, *CPU-States*). Das bedeutet, dass ein relativ hoher Anteil für die Socket-Operationen, also das Kopieren der Daten zwischen den Prozessen, aufgewendet wird, und die eigentliche Abfrageoperationen sowie das Codieren und Decodieren der Daten weniger aufwendig sind.

gethostbyname() über files

```
last pid: 71912;  load averages:  0.75,  0.38,  0.16   up 101+01:15:38 15:06:36
52 processes:  2 running, 50 sleeping
CPU states: 40.9% user,  0.0% nice, 58.8% system,  0.4% interrupt,  0.0% idle
Mem: 35M Active, 6320K Inact, 16M Wired, 2508K Cache, 14M Buf, 1360K Free
Swap: 261M Total, 14M Used, 247M Free, 5% Inuse

  PID USERNAME PRI NICE  SIZE    RES STATE    TIME   WCPU    CPU COMMAND
71912 sleder2s  55   0   336K   188K RUN      0:16 95.87% 52.83% tester
70256 root       2   0 14068K  9808K select   7:59  1.37%  1.37% XF86_S3
71897 sleder2s  28   0  1924K  1004K RUN      0:02  0.00%  0.00% top
```

Hier als Vergleich ein Testlauf mit einem Hostnamen, der direkt aus der */etc/hosts*-Datei entnommen werden konnte. Der *lookupd*-Prozess ist dementsprechend nicht beteiligt, und der *tester*-Prozess verbraucht nahezu die gesamte CPU-Zeit. Was aus dieser Darstellung nicht erkennbar ist, ist die Tatsache, dass die Zeit für eine einzelne Namensauflösung um den Faktor 20 kürzer ist, da Dateioperationen innerhalb eines Prozesses wesentlich schneller sind als eine Interaktion zwischen zwei Prozessen.

LookupdQuery(), Quelle DNS/Cache

```
last pid: 69926;  load averages:  1.08,  0.96,  0.74   up 100+00:25:25 14:16:20
52 processes:  3 running, 49 sleeping
CPU states: 32.3% user,  0.0% nice, 67.7% system,  0.0% interrupt,  0.0% idle
Mem: 34M Active, 6768K Inact, 16M Wired, 2636K Cache, 14M Buf, 1128K Free
Swap: 261M Total, 20M Used, 241M Free, 7% Inuse

  PID USERNAME PRI NICE  SIZE    RES STATE    TIME   WCPU    CPU COMMAND
55336 root      46   0   476K   284K RUN      4:22 48.68% 48.68% lookupd
69926 sleder2s   2   0   340K   200K RUN      0:26 44.22% 41.80% tester
66406 root       2   0 20528K 11772K select 191:47  0.15%  0.15% XF86_S3
69792 sleder2s  28   0  1928K   976K RUN      0:11  0.00%  0.00% top
```

Bei diesem Testlauf wurde direkt die LookupdQuery()-Funktion genutzt, um eine Namensauflösung aus dem DNS-Dienst zu erhalten. Das Ergebnis weicht kaum von dem ersten Testlauf ab. Allenfalls ist eine leichte Senkung des *lookupd*-Anteils an der Rechenzeit zu beobachten. Dies könnte darauf hindeuten, dass der *tester*-Prozess bei diesem Testlauf weniger Arbeit zu verrichten hat, weil er das Ergebnis nicht erst noch in das für die *gethostbyname()*-Funktion geeignete Format umwandeln muss.

LookupdQuery, Quelle IPv6Lookup/Cache

```
last pid: 69928;  load averages:  1.45,  1.12,  0.83   up 100+00:27:34 14:18:29
53 processes:  4 running, 49 sleeping
CPU states: 35.0% user,  0.0% nice, 64.6% system,  0.4% interrupt,  0.0% idle
Mem: 35M Active, 6976K Inact, 16M Wired, 2544K Cache, 14M Buf, 480K Free
Swap: 261M Total, 20M Used, 241M Free, 7% Inuse

  PID USERNAME PRI NICE  SIZE    RES STATE    TIME   WCPU    CPU COMMAND
55336 root      56   0   476K   284K RUN      5:14 48.49% 48.49% lookupd
69927 sleder2s  51   0   340K   200K RUN      0:24 42.44% 39.99% tester
66406 root       2   0 20528K 11816K select 191:51  0.98%  0.98% XF86_S3
```

Hier wurde zur Namensauflösung ein Name nachgefragt, der nur in der IPv6Lookup-Nachbarschafts-Tabelle vorhanden war. Das Ergebnis entspricht dem des vorigen Testlaufs.

Kapitel 9

Zusammenfassung und Ausblick

Die Implementierung des *lookupd*-Dæmons und der Anbindung der C-Bibliothek wurde erfolgreich abgeschlossen. In Tests und bei täglicher Nutzung auf einigen Testsystemen hat sich die Software bereits bewährt. Die Performance ist, soweit es feststellbar war, mehr als akzeptabel. Der Speicherverbrauch ist minimal. Auch unter hoher Last werden Anfragen schnell verarbeitet. Der gewählte Ansatz zur Parallelisierung von Anfragen, der auf Events basiert, hat sich damit als gut geeignet erwiesen.

Das IPv6Lookup-Protokoll wurde erfolgreich in den *lookupd*-Dæmon integriert. Ebenso wurde parallel dazu die Namensauflösung über DNS integriert. Damit wurde die Tragfähigkeit der hier entworfenen Namensdienst-Architektur gezeigt.

Die anfangs gestellte Forderung nach dem konfigurationslosen Betrieb eines IPv6-Netzwerks kann nun mit Hilfe des IPv6Lookup-Protokolls erfüllt werden. Durch die hier entworfene Namensdienst-Architektur kann die neue Methode der Namensauflösung transparent von allen Anwendungen genutzt werden.

Das Konzept des *lookupd*-Dæmons ist sehr umfassend ausgelegt, daher konnte aus Zeitgründen nur eine Untermenge implementiert werden. Zunächst wurde nur die Namensauflösung implementiert, obwohl das Konzept alle in Kapitel 5.1 aufgeführten Informationsarten einschließt. Als Informationsquellen wurden nur DNS und IPv6Lookup implementiert. Es fehlen Module für den Einsatz von NIS, LDAP oder anderen Informationssystemen. Ebenso muss die C-Bibliothek noch weiter modifiziert werden, um eine Anbindung an *lookupd* für andere Informationsarten zu erreichen. Alle diese Elemente können aber problemlos nachträglich in die vorhandene Software eingebaut werden.

Für FreeBSD 5.0 ist eine Integration einer Implementierung des NSSwitch-Systems (siehe Kapitel 5.1) geplant. Dies hat einige Konsequenzen für eine Weiterentwicklung der *lookupd*-Software. Eine Möglichkeit der Weiterentwicklung besteht darin, *lookupd* so zu erweitern, dass das NSSwitch-System in den Dæmon integriert wird. Eine andere Möglichkeit wäre, die Anbindung an den *lookupd*-Dæmon über ein NSSwitch-Modul zu realisieren. Dabei würden zwar Teile der *lookupd*-Funktionalität ungenutzt bleiben, aber eine Modifikation der C-Bibliothek wä-

re dann nicht mehr notwendig. Denkbar wäre auch eine reduzierte Version des *lookupd*-Dæmons, die nur das IPv6Lookup-Modul enthält.

Ein anderes Feld zur Weiterentwicklung betrifft das IPv6Lookup-Protokoll. Hier wäre eine Möglichkeit für Anwendungen von Vorteil, um Dienste registrieren zu lassen und Dienste gezielt zu suchen. Dazu müsste eine Bibliothek geschaffen werden, die diese Funktionen für Anwendungen zur Verfügung stellt. Außerdem müsste eine Schnittstelle zu dem sich im *lookupd*-Dæmon befindlichen IPv6Lookup-Modul entworfen werden. Dazu kann die bereits vorgesehene Schreiboperation des *lookupd*-Protokolls genutzt werden.

Eine Archivdatei der Software ist unter der URL `http://pc-2n00.6lab.inf.fh-rhein-sieg.de/~sleder2s/lookupd.html` abrufbar. Hier werden voraussichtlich auch die Weiterentwicklungen veröffentlicht.

Abkürzungsverzeichnis

AH	Authentication Header
BIND	Berkeley Internet Name Dæmon
BOOTP	Boot Protocol
BSD	Berkeley Software Distribution
CIDR	Classless Inter-Domain Routing
DES	Data Encryption Standard
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Service
ESP	Encapsulating Security Payload
EUI	Equipment Unique Identifier
FQDN	Fully Qualified Domain Name
GPL	GNU General Public License
ICMP	Internet Control Message Protocol
IKE	Internet Key Exchange
IP	Internet Protocol
IRP	Information Retrieval Protocol
IRPD	Information Retrieval Protocol Dæmon
IRS	Information Retrieval System
LDAP	Lightweight Directory Access Protocol
LGPL	GNU Library General Public License
LPD	Line Printer Daemon
MAC	Medium Access Control
MTU	Maximum Transfer Unit
NAT	Network Address Translation
NBP	Name Binding Protocol
NFS	Network File System
NIS	Network Information System
NSL	Network Services Library
NSSwitch	Name Service Switch
RARP	Reverse Address Resolution Protocol
SLP	Service Location Protocol
TCP	Transmission Control Protocol
TOS	Type Of Service
UDP	User Datagram Protocol
URL	Uniform Resource Locator
YP	Yellow Pages

Literaturverzeichnis

[Sedgewick98] SEDGEWICK, ROBERT: “Algorithmen in C”, Addison-Wesley, Bonn, 3. Auflage 1998.

[Stevens00] STEVENS, W. RICHARD: “Programmieren von UNIX-Netzwerken”, Hanser, München, 2. Auflage 2000.

[Dittler98] DITTLER, HANS PETER: “IPv6. Das neue Internet- Protokoll. Technik, Anwendung, Migration.”, dpunkt-Verlag, Heidelberg, 1998

[HD99] HINDEN, T. M. und S. DEERING: “IPv6 protocols”, Addison-Wesley, Reading, 1999

[LFJL93] LEFFLER, SAMUEL J., ROBERT S. FABRY, WILLIAM N. JOY , PHIL LAPSLEY et al.: “An Advanced 4.4BSD Interprocess Communication Tutorial”, 4.4BSD Programmer’s Supplementary Documents PSD:21, 1986, 1993

[EUI64] IEEE, "Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority", `http://standards.ieee.org/regauth/oui/tutorials/EUI64.html`, März 1997.

[Apple90] APPLE COMPUTER, INC.: “Inside AppleTalk”, Addison-Wesley, New York, 2. Auflage, 1990, `http://devworld.apple.com/macos/opentransport/docs/dev/Inside_AppleTalk.pdf`

[KFP00] KING, STEVE, ROBERT FINK, CHARLES E. PERKINS et al.: “The Case for IPv6”, Internet Architecture Board, Internet Draft, Juni 2000

[RFC1034] MOCKAPETRIS, P. : “Domain Names – Concepts And Facilities”, IETF, RFC 1034, November 1987

[RFC1035] MOCKAPETRIS, P. : “Domain Names – Implementation And Specification”, IETF, RFC 1035, November 1987

[RFC2460] DEERING, S. und R. HINDEN: “Internet Protocol, Version 6 (IPv6) Specification”, IETF, RFC 2460, Dezember 1998.

[RFC2461] NARTEN, T. , E. NORDMARK und W. SIMPSON: “Neighbor Discovery for IP Version 6 (IPv6)”, IETF, RFC 2461, Dezember 1998

[RFC2462] THOMSON, S. und T. NARTEN: “IPv6 Stateless Address Autoconfiguration”, IETF, RFC 2462, Dezember 1998

[RFC2463] CONTA, A. und S.DEERING: “Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification”, IETF, RFC2463, Dezember 1998

[RFC2373] HINDEN, R. und S. DEERING: “IP Version 6 Addressing Architecture”, IETF, RFC 2373, Juli 1998.

[RFC2133] GILLIGAN, R., S. THOMSON, J. BOUND und W. STEVENS: “Basic Socket Interface Extensions for IPv6”, IETF, RFC2133, April 1997

[RFC2292] STEVENS, W. und M. THOMAS: “Advanced Sockets API for IPv6”, IETF, RFC2292, Februar 1998

[RFC2409] HARKINS, D. und D. CARREL : “The Internet Key Exchange (IKE)”, IETF, RFC2409, November 1998

[IBM84] IBM CORP., "IBM PC Network Technical Reference Manual", No. 6322916, September 1984.

[RFC1001] "Protocol Standard For a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods", IETF, RFC 1001, März 1987.

[RFC1002] "Protocol Standard For a NetBIOS Service on a TCP/UDP Transport: Detailed Specifications", IETF, RFC 1001, März 1987.

[RFC2608] GUTTMAN, E., C. PERKINS, J. VEIZADES und M. DAY: "Service Location Protocol Version 2", IETF, RFC 2608, Juni 1999.

[RFC2609] GUTTMAN, E., C. PERKINS und J. KEMPF: "Service Templates and Service: Schemes", IETF, RFC 2609, Juni 1999.