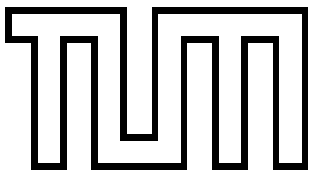
INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN



Diplomarbeit

Entwurf einer Managementschnittstelle für einen PC-basierten Switch

Peter Allgeyer

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Stephen Heilbronner, Norbert Wienold
Abgabedatum:	15. August 1998

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

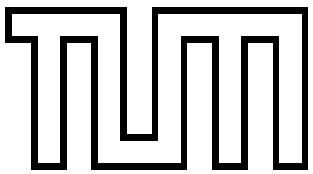
München, den 15. August 1998

.....

(Unterschrift des Kandidaten)

INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN



Diplomarbeit

Entwurf einer Managementschnittstelle für einen PC-basierten Switch

Peter Allgeyer

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Stephen Heilbronner, Norbert Wienold
Abgabedatum:	15. August 1998

Zusammenfassung

Der Switch ist ein immer häufiger anzutreffendes Element heutiger Rechnernetze. Dabei lassen sich an ihn Netzkomponenten mit unterschiedlichen Eigenschaften und Aufgabenbereichen anschließen. Dazu gehören Workstations, mobile Endgeräte (Notebooks), Drucker, Plotter u. a. Deren Aufgabenbereiche erstrecken sich von angebotenen Druck- oder Plottdiensten über Serverdienste, wie z. B. HTTP- oder FTP-Server, bis hin zur reinen Bereitstellung von Rechendiensten für Endanwender. Die Zuteilung der Rechte an bestimmten Diensten an die Endgeräte kann dabei recht komplex werden. Soll zusätzlich noch die Möglichkeit gegeben werden, fremden Endgeräten Zugang zu Netzressourcen wie beispielsweise den Daten einer öffentlichen Datenbank zu gewähren, braucht man ein effizientes Management der im Netz vorhandenen Ressourcen, das abhängig von den Eigenschaften der angeschlossenen Endsysteme ist. Ein Weg dies zu ermöglichen, ist die Einteilung der im Rechnernetz vorhandenen Endgeräte in VLAN's. Alle Mitglieder eines VLAN's sollen dabei nur die Ressourcen nutzen dürfen, die in ihrem VLAN angeboten werden.

In dieser Arbeit soll ein auf der PC-Architektur aufgebauter Switch die VLAN's realisieren. Die hauptsächliche Aufgabe besteht darin, einen vorhandenen Bridging-Code um VLAN Funktionalität zu erweitern und die Managebarkeit des Switch zu gewährleisten. Dazu soll ein CORBA-basiertes Management für den Switch entwickelt und prototypisch implementiert werden, das sich in ein bestehendes Agentensystem integrieren läßt. Die Entwicklung der Managementschnittstelle des Switch basiert auf der *Object Modeling Technique* (OMT), die einen objektorientierten Modellierungsansatz bietet, der unabhängig von einer bestimmten Managementarchitektur ist. Der Agent wird in Java mit Hilfe der CORBA-Entwicklungs- und Laufzeitumgebung *VisiBroker for Java* realisiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel gibt eine Einführung in die Thematik dieser Arbeit. Dabei wird anhand einer Zusammenfassung aktueller Gegebenheiten und der daraus resultierenden Probleme die Motivation hergeleitet. Weitergehend werden die Aufgabenstellung und die wichtigsten Ziele dieser Arbeit skizziert. Ein Überblick über den Aufbau dieser Arbeit schließt die Einleitung ab.

1.1 Motivation

Traditionell werden Rechnernetze aus organisatorischer Sicht, aufgrund infrastruktureller Anforderungen oder Performancegesichtspunkten in Broadcastdomänen unterteilt. Eine Aufteilung in mehrere kleine Broadcastdomänen verringert dabei die Anzahl der *Kollisionen* auf dem Netz. Vor allem die Trennung nach räumlichen Gesichtspunkten spielt oft eine große Rolle. Die Aufteilung der Rechnernetze bewerkstelligen sogenannte *Kopplungselemente*, wie sie Router, Bridges und Switches darstellen. Sobald aber *mobile* Endgeräte, im folgenden *nomadische Systeme* genannt, an einem solchen Netz beteiligt sind, läßt sich dieses statische Modell aufgrund der örtlichen Unabhängigkeit nomadischer Systeme nicht mehr so weiterverwenden.

In einem Rechnernetz werden im allgemeinen bestimmte *Serverdienste*, die *Ressourcen*, angeboten. Dazu gehören zum Beispiel Druck- und Dateidienste. Um diese nutzen zu können, müssen die nomadischen Systeme auf jedem Server einzeln mit einer Berechtigung versehen sein. Eine Rechtevergabe abhängig von dem Netzsegment, in dem sich das nomadische System gerade befindet, findet nicht statt. Statt einer statischen Unterteilung des Netzes in fest definierte Broadcastdomänen ist eine logische Trennung der Netzsegmente ein gangbarer Weg, die Ressourcen und angebotenen Dienste im Netz feiner freizugeben.

Geeignetes Mittel, eine logische Trennung der Netzsegmente zu erreichen ist der Einsatz von *Virtuellen LAN's* (VLAN's). Ein VLAN trennt ein herkömmliches Netz anhand gewisser Kriterien in mehrere logische Subnetze auf. Im kommerziellen Bereich sind VLAN's zunehmend Teil der Funktion eines Switch —

kaum noch ein namhafter Hersteller von Switchen, der nicht auch die VLAN-Funktionalität seiner Produkte bewirbt. Der Vorteil von VLAN's liegt darin, daß unabhängig vom physischen Standort des am Switch angeschlossenen Systems seine Zugehörigkeit zu einem oder mehreren logischen Subnetzen geregelt werden kann.

Die Trennung in logische Netzsegmente durch einen Switch mit VLAN-Funktionalität allein löst allerdings noch nicht das Problem der Ressourcenverteilung von angebotenen Serverdiensten. Es stellt lediglich ein Werkzeug zu dessen Verwirklichung bereit. Was fehlt ist ein Management, das in der Lage ist, nomadische Systeme mit der Hilfe von VLAN's so zu steuern, daß schon auf niedriger Ebene des *OSI-Schichtenmodells* [HA93] eine Art Zugangskontrolle und Ressourcenverteilung ermöglicht wird.

1.2 Aufgabenstellung

In dieser Arbeit soll insbesondere auf die Managementanforderungen eingegangen werden, die beim Einsatz mobiler Endgeräte entstehen. Die Trennung der Broadcastdomänen soll ein auf der PC-Architektur arbeitender Switch bewerkstelligen. Besondere Berücksichtigung soll die Eigenschaft des Switch finden, VLAN's zu bilden.

Es sollen Einsatzszenarien für die Nutzung von VLAN's unter Berücksichtigung nomadischer Systeme herausgearbeitet werden. Die sich daraus ergebenen Anforderungen an das Management des Switch sind ein wesentlicher Punkt dieser Arbeit. Probleme, die bei der Erkennung neuer Systeme auftreten, deren Identifizierung und Authentifizierung sowie die Zuordnung zu bestimmten VLAN's, sollen detailiert geschildert und diskutiert werden.

Insbesondere soll eine Managementschnittstelle für einen Switch mit VLAN Funktionalität spezifiziert werden. Wesentlich ist dabei die Betrachtung der Möglichkeiten, die sich aus dem Einsatz von nomadischen Systemen in einem solchen Rechnernetz ergeben.

Die prototypische Implementierung der VLAN-Funktionalität für einen auf PC-Basis arbeitenden Switch sowie dessen Management sollen ein zentraler Bestandteil dieser Arbeit sein.

1.3 Aufbau der Arbeit

Im Kapitel 2 werden die Grundlagen der wichtigsten vorkommenden Begriffe aus technischer Sicht erklärt. Darüberhinaus gibt es eine Einführung in die in dieser Arbeit verwendete objektorientierte Modellierungstechnik. Eine Einführung in CORBA schließt das Kapitel ab. Im Kapitel 3 werden mögliche Einsatzszenarien betrachtet und anhand eines Beispielszenarios die Problematik des Anschlusses nomadischer Systeme an ein Rechnernetz geschildert. Das Kapitel 4 arbeitet

die nötigen Anforderungen an die Managementfunktionalität heraus, die aus Kapitel 3 erwachsen. Insbesondere stellt dieses Kapitel ein Objektmodell für das Switch-Management vor. Kapitel 5 wird die Erweiterung der bestehenden Bridging-Funktionalität eines Linux Kernels um VLAN-Funktionalität betrachten. Darüberhinaus wird die Tragfähigkeit des in Kapitel 4 entworfenen Objektmodells im Kapitel 5 durch die prototypische Implementierung der Funktionalität der Managementschnittstelle nachgewiesen. Das letzte Kapitel gibt eine Zusammenfassung der Arbeit und einen Ausblick auf unzureichend behandelte Themen.

Kapitel 2

Grundlagen

Dieses Kapitel versteht sich als eine Einführung in die Grundlagen der in dieser Arbeit verwendeten Begriffe und Techniken. Neben den technischen Grundlagen zum Thema Bridging, Switching und DHCP sowie einer schichtspezifischen (im Sinne des OSI-Schichtenmodells) Betrachtung der Realisierung von VLAN's soll auch ein Überblick über die in dieser Arbeit verwendeten objektorientierten Modellierungstechniken gegeben werden. Am Ende dieses Kapitels steht eine Einführung in die Konzepte der *Common Object Request Broker Architecture* (CORBA).

2.1 Technische Grundlagen

Nachdem in den letzten 15 Jahren die Kopplung von Netzsegmenten durch Router und Bridgen geprägt war, werden vor allem die Bridgen zunehmend durch den Einsatz der zeitlich neueren Switchingtechnologie verdrängt. Dessen Technologie geht auf die Funktionsweise einer Bridge zurück. Neben der höheren Bandbreite bietet der Switch oftmals auch die Realisierung von VLAN's und eine erweiterte Managebarkeit gegenüber der Bridge an. Die Konfiguration der mobilen Endsysteme geschieht dabei zumeist mit dem *Dynamic Host Configuration Protocol* (DHCP). Da die Managebarkeit eines Switch zentrales Thema dieser Arbeit ist, sollen die vier Begriffe Bridge, Switch, DHCP und VLAN zunächst aus technischer Sicht erklärt werden.

2.1.1 Bridging

Die Technologie des Bridging gibt es seit Anfang der achtziger Jahre. Am Anfang verbanden Bridges homogene Netze miteinander und ermöglichten die Weiterleitung von Paketen zwischen den einzelnen Netzsegmenten. In letzter Zeit wurden auch Bridges für heterogene Netze definiert und standardisiert [ID97]. Damit ist es z. B. auch möglich, *Ethernet*-Netze mit *Token-Ring*-Netzen zu koppeln, eine Möglichkeit, die zuvor nur Router boten. Da sich in heutigen lokalen

Netzen überwiegend das Ethernet durchgesetzt hat, soll sich in der folgenden Beschreibung auf Ethernet-Netze bezogen werden, soweit nichts anderes explizit angegeben wird.

Zwei Hauptvarianten des Bridgings sind von Interesse: *Transparent Bridging* findet man vor allem in Ethernet Umgebungen. *Source-Route Bridging* wird dagegen vornehmlich in Token-Ring Umgebungen eingesetzt. Bridges erlauben es, ein Rechnernetz transparent zu segmentieren. Das bedeutet, daß die angeschlossenen Systeme nichts von der Bridge wissen müssen. Für sie erscheint das Netz vollkommen transparent und einheitlich. Die auf der OSI-Schicht 2 arbeitende Bridge kann eine Kollisionsdomäne auftrennen.

Eine Bridge besteht aus einer Anzahl von Anschlußvorrichtungen für Netzkomponenten, den sogenannten *Ports*. Die Ports einer Bridge müssen in der Lage sein, auf den Datenverkehr im Netz im sogenannten *promiscous mode* zu „lauschen“. Das bedeutet, daß die Ports alle Pakete akzeptieren, unabhängig davon, an wen die Pakete adressiert sind. Durch das Mithören auf dem Netz lernt die Bridge, welche Systeme wo angeschlossen sind und speichert diese Information in ihrer internen Tabelle. Erreicht ein Datenpaket die Bridge, so schaut diese in ihre intern aufgebaute Tabelle, ob die Zieladresse des Pakets bekannt ist. Wenn dem so ist, leitet sie das Paket nur an den Port weiter, an dem das Gerät mit der Zieladresse angeschlossen ist, oder verwirft das Paket, falls das System mit der Zieladresse an dem Port, an dem das Paket die Bridge erreichte (*Quellport*), angeschlossen ist. Kennt die Bridge das Zielsystem (oder technisch ausgedrückt dessen MAC-Adresse), noch nicht, so leitet sie das Paket an alle Ports bis auf den Quellport weiter. Gleiches gilt für empfangene Broadcast-Pakete. Durch einen Timeout (*aging time*) wird verhindert, daß längst entfernte Systeme in der Tabelle auftauchen und somit Speicherplatz verschwenden.

Ein Beispiel soll das verdeutlichen: Angenommen, ein Rechnernetz besteht aus einer Bridge mit vier Ports. Daran sind fünf Endgeräte angeschlossen (Abbildung 2.1). Die Ports der Bridge sind von eins bis vier durchnummeriert. An Port 1 sind die Endgeräte A und B angeschlossen, an Port 3 das Endgerät C und an Port 4 die beiden Systeme D und E. Port 2 bleibt frei.

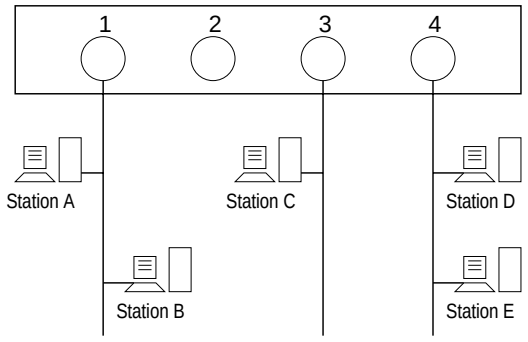


Abbildung 2.1: Einfaches Rechnernetz mit Bridge

Was passiert, wenn das Endgerät A ein Paket an das System C sendet? Die Bridge, die sich im *promiscous mode* befindet, fragt ihre interne Tabelle ab (dabei soll zunächst ungeklärt bleiben, wie diese Tabelle aufgebaut wird) und merkt, daß das System C an Port 3 angeschlossen ist. Daraufhin leitet sie das Paket an Port 3 weiter. Sendet während dieses Vorganges das Endgerät D ein Paket an das Endgerät E, so leitet die Bridge das Paket nicht weiter, da sie merkt, daß das Paket an den Quellport zurückgeschickt werden würde. Ohne eine Bridge in diesem Netz, würde es zu einer Kollison kommen und das Endgerät D könnte nicht senden, da das Endgerät A Zugriff auf das Netz hat (Abbildung 2.2).

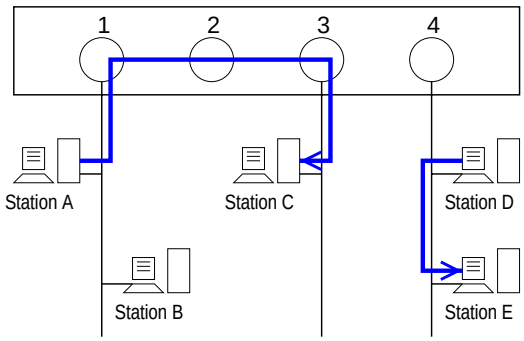


Abbildung 2.2: *Forwarding*

Wie baut die Bridge ihre Tabelle auf? Am Beispiel des obigen Rechnernetzes soll dies verdeutlicht werden. Die Annahme ist, daß die interne Tabelle der Bridge leer ist. Nun sendet wieder das Endgerät A ein Paket an das Endgerät C. Da die Bridge das Endgerät C nicht in ihrer Tabelle findet, leitet die das Paket an alle ihre Ports außer an Port 1, dem Port von wo das Paket kam, weiter (Abbildung 2.3). Dieses Vorgehen wird als *flooding* bezeichnet. Die Bridge weiß nun, daß das Paket von dem Endgerät A an Port 1 kam und trägt diese Information in ihre Tabelle ein. In Zukunft wird sie alle Pakete, die an das Endgerät A geschickt werden, nur an Port 1 weiterleiten.

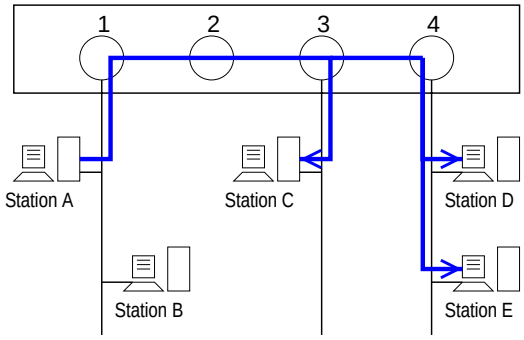


Abbildung 2.3: *Flooding*

Zusammengefaßt läßt sich feststellen: Bridging findet auf der Schicht 2 (*Link Layer*) des OSI-Referenzmodells statt. Ein Datenpaket auf dieser Schicht wird

als Frame bezeichnet. Bridges analysieren eingehende Frames, treffen Entscheidungen abhängig von der Information im betreffenden Frame und reichen den Frame an das entsprechende Ziel weiter. Hierbei spricht man von *Forwarding*. Manchmal, so im Fall des Source-Route Bridging, befindet sich der Pfad zum Ziel fest in jedem Frame. Da das aber selten, und wie erwähnt, nur in Token-Ring Umgebungen zum Einsatz kommt, soll das Source-Route Bridging hier nicht weiter betrachtet werden.

Bridges sind in der Lage, Frames anhand der Schicht 2 Felder zu filtern. So kann eine Bridge bspw. alle Frames, die von einem bestimmten Netz emfangen werden zurückweisen und damit nicht forwarden. Da Link-Layer Informationen oftmals Verweise auf auf höher gelegene Protokolle (Schicht 3) aufweisen, können Bridges für gewöhnlich auch aufgrund dieser Informationen filtern. Auch beim Umgang mit unnötigen *Broadcast*- und *Multicast*-Verkehr kann eine Bridge hilfreich sein.

Zum Thema Bridging findet sich viel Literatur. Einige ausgewählte Beispiele finden sich in [ID97, Hal92, Per93, BAS, BRIa, BRIb]. Darunter ist auch der Standard zum Thema *Media Access Control Bridges* der IEEE, der detailiert auf das Thema Bridging eingeht ([ID97]).

2.1.2 Switching

In den letzten Jahren verdrängt der Einsatz von Switchen zunehmend die Bridgen. Konzeptionell ist der Unterschied zwischen einer Bridge und einem Switch nur gering. Der Switch arbeitet im Prinzip genauso wie die Bridge. Im Gegensatz zur Bridge garantiert der Switch aber eine gewisse Bandbreite. Das bedeutet, daß auf jedem Port eine theoretisch garantierte Übertragungsrate möglich ist. Für den Switch heißt das, daß er hardwaretechnisch wesentlich aufwendiger ist als die Bridge. Will man bspw. an einem Switch mit vier Ports ein 100BaseT Netz anschließen, also je zwei gleichzeitige Verbindungen mit je 100 Mbit/s zulassen, so muß der Switch intern schon eine Bandbreite von 200 Mbit/s zur Verfügung stellen.

Ein weitere Unterschied zu Bridges ist das Angebot an Switches, die in der Lage sind, mit VLAN-Technologie umzugehen. Für den Switch bedeutet das erhöhten Aufwand bei seiner Managementsoftware. Es genügt nun nicht mehr, eine dynamisch erzeugte Abbildung aller MAC-Adressen zu deren Anschlußports vorzuhalten. Vielmehr müssen neue Abbildungen geschaffen werden, die die Ports bzw. MAC-Adressen einem oder mehreren VLAN's zuordnen. Das geschieht in der Regel nicht automatisch, sondern muß durch Managementeingriffe gesteuert werden. Die Einbindung der VLAN's in den Switching-Algorithmus ist eine weitere Neuerung. Hardwaretechnisch impliziert dies, daß der Switch aufgrund der erhöhten Anforderungen an seine Forwardingregeln leistungsstärker als die Bridge sein muß.

Im Regelfall wird pro Port eines Switch je ein Endsystem angeschlossen. Dadurch erreicht man die geringstmögliche Größe einer Kollisionsdomäne. Zusammen mit der Eigenschaft des Switch, eine garantierte Übertragungsrate pro Port zu ge-

währleisten, läßt sich so eine veraltete Netzstruktur mit relativ wenig Aufwand (ohne Neuverkabelung, Anschaffung neuer Netzkarten etc.) performancemäßig beschleunigen. Daher wird der Switch, trotz seines relativ hohen Preises, gegenüber der Bridge heute vorgezogen.

Aufgrund der geringen konzeptionellen Unterschiede werden die Begriffe Switch und Bridge in dieser Arbeit synonym verwendet.

2.1.3 DHCP

Das *Dynamic Host Configuration Protocol* (DHCP) [Dro93, Dro97] wird zur Konfiguration von Computersystemen eingesetzt, die an ein Rechnernetz angeschlossen werden. DHCP kommt v. a. dann zur Anwendung, wenn bestimmte Konfigurationsdaten (z. B. IP-Adressen) dynamisch beim Anschluß von Endgeräten an ein Rechnernetz an diese übertragen werden sollen. Daher eignet es sich besonders für die Konfiguration von mobilen Endgeräten. Die Konfigurationsdaten für die Computersysteme werden von einem *DHCP-Server* zur Verfügung gestellt. Dabei tauschen Server und Client beim Anmeldevorgang festdefinierte Protokolldaten, die sogenannten DHCP-PDU's, aus.

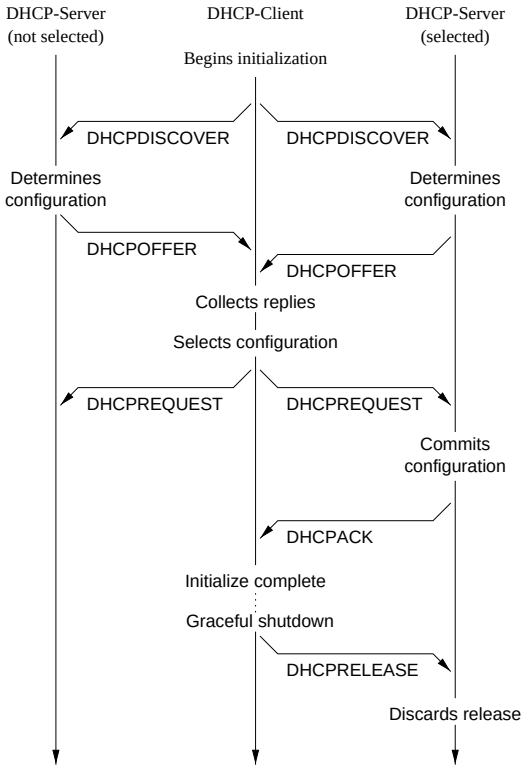


Abbildung 2.4: Client/Server-Dialog unter DHCP

Meldet sich ein Client beim Server an, so schickt der Client einen *DHCPDISCOVER* als UDP-Broadcast auf das Netz. Jeder im Netz vorhandener DHCP-Server antwortet mit einem *DHCPOFFER*, wenn er eine passende Konfiguration

anzubieten hat. Antwortet kein DHCP-Server, so wiederholt der Client nach einer gewissen Zeit seine Anfrage. Aus den empfangenen DHCPOFFER's wählt sich der Client eines aus und schickt dessen Konfiguration mittels eines *DHCPREQUEST* an den Server zurück. Im Normalfall bestätigt der Server den DHCPREQUEST mit einem *DHCPACK*. Der Client erhält so für einen gewissen Zeitraum, der sogenannten *lease time*, eine gültige Konfiguration. Mit dem Schicken eines *DHCPRELEASE* an den Server meldet sich der Client vor dem Verlassen des Netzes ab. Abbildung 2.4 verdeutlicht dieses Vorgehen.

2.1.4 Virtuelle LAN's

VLAN's bilden, wie schon angesprochen wurde, ein Mittel, Netze in logische Subnetze aufzuteilen. Diese pauschale Aussage soll im weiteren genauer definiert werden. Anhand des OSI-Referenzmodells wird eine klare Gliederung von VLAN's aufgezeigt.

Definition des Begriffs VLAN

Der Begriff VLAN wird in der Literatur unterschiedlich definiert [DvT97, Vuk96, IW98]. Im wesentlichen basieren die Erklärungen auf verschiedenen Sichtweisen.

Ein VLAN ist eine Limited Broadcast Domain.
Die Mitglieder eines virtuellen LAN's erhalten nur Broadcasts aus ihrem eigenen VLAN. Broadcasts aus anderen VLAN's erreichen sie nicht. Die Broadcast-Pakete werden intern in Multicast Pakete umgewandelt und so nur den DTE's zugeteilt, die in einem VLAN liegen. Der Begriff *limited* soll das verdeutlichen.

Ein VLAN trennt die physische von der logischen Netzsicht.
Durch die Gruppierung von DTE's in virtuelle LAN's bleibt die DTE auch bei einem physischen Umzug in der logischen Gruppe, der es zugeordnet wurde. Somit können die Mitglieder eines VLAN's physisch zerstreut über das gesamte Netz verteilt sein. Die logische Zuordnung geschieht transparent für diese in den Netzkomponenten.

Zwischen den Mitgliedern eines VLAN ist kein Routing notwendig.
Durch VLAN's auf Schicht 2 des OSI-Referenzmodells können alle Teilnehmer in einem VLAN ohne Verzögerung eines Routers miteinander kommunizieren. Router sind im allgemeinen langsamer als eine Bridge oder ein Switch, da sie auf der Schicht 3 des OSI-Referenzmodells arbeiten und dort umfangreichere Aufgaben übernehmen müssen. Hierzu gehören bspw. Protokollumsetzungen sowie das Auswerten von Routingtabellen. Teilnehmer, die sich nicht in einem gemeinsamen VLAN befinden brauchen jedoch weiterhin ein Kopplungselement auf Schicht 3.

Alle drei Aspekte sind für das Projekt, dessen Bestandteil diese Arbeit ist, von Interesse. Um aber genau zu verstehen, wie ein VLAN arbeitet, ist eine technische Sicht auf dieses notwendig.

Technischer Überblick über VLAN's

VLAN's werden nach der Schicht des OSI-Referenzmodells eingeteilt, auf der sie arbeiten. Daraus ergeben sich Unterschiede in den Möglichkeiten der Realisierung von VLAN's.

VLAN's auf der Bitübertragungsschicht
Bei VLAN's, die auf der Schicht 1 des OSI-Referenzmodells arbeiten, werden eine Anzahl von physischen Ports zu einer Broadcastdomäne gruppiert. Für das Kopplungselement, meist einem Switch, sind ausschließlich die Ports von Bedeutung. Die physischen Endgeräte an einem derartigen Port arbeiten vollkommen transparent in Bezug auf den Switch, an dem sie hängen, in dem ihnen zugeordneten VLAN. Das an dem Port hängende Segment kann nur komplett einem bestimmten VLAN zugeordnet werden, da dem Switch Informationen über die Identität der angeschlossenen Systeme fehlt.

VLAN's auf der Sicherungsschicht
Durch das Arbeiten auf der Schicht 2 des OSI-Referenzmodells weiß der Switch, der die VLAN's realisiert, über die Identität der angeschlossenen Systeme Bescheid. Ihm dienen die weltweit eindeutigen MAC-Adressen der Endsysteme als Identifikator. Durch die Verwendung dieses Identifikators können die Endsysteme so auf ein oder mehrere VLAN's abgebildet werden. Der Switch muß jedem Endgerät einen Anschlußport zur Verfügung stellen. Würden mehrere Endgeräte an einem Port hängen, so ist es möglich, daß diese unter Umgehung des Switch intern kommunizieren. Dieses Szenario führt den Einsatz von VLAN's durch den Switch ad absurdum, da dieser keinen Einfluß mehr auf den Kommunikationsweg besitzt.

VLAN's auf der Vermittlungsschicht
Auf der Schicht 3 des OSI-Referenzmodells können Endsysteme auf der Basis von Subnetzen zu einer Broadcast-Domäne verbunden werden. Diese sogenannten virtuellen Subnetze basieren auf der Zuordnung der IP-Adressen der beteiligten Systeme zu verschiedenen VLAN's.

Der Vorteil von VLAN's liegt darin, daß sie es ermöglichen, Server und Clients räumlich getrennt voneinander aufzustellen, was die Verwaltung und Sicherheit[1] des Netzes erhöht, dabei den Clients jedoch nur eine bestimmte Untermenge der Serverdienste zuteil werden zu lassen. In Kapitel 3 wird in einem Beispiel auf diese Thematik näher eingegangen.

[1] Erhöhung der Sicherheit durch das Aufstellen der Server in einem abgeperrten Serverraum bspw.

Für das in dieser Arbeit vorgestellte Einsatzszenario sind vor allem VLAN's auf Schicht 1 und Schicht 2 von Bedeutung. In der Praxis ist es meist sinnvoll, VLAN's aus einer Kombination von Port- und MAC-Adressen zu bilden — vor allem, wenn nicht routbare Protokolle wie z. B. AppleTalk zum Einsatz kommen.

Der hier eingesetzte PC-basierte Switch auf Basis eines unter dem Betriebssystem Linux laufenden PC's soll VLAN's auf der Bitübertragungs- und Sicherungsschicht ermöglichen. Dazu ist es notwendig den vorhandenen Bridgingcode um VLAN-Funktionalität zu erweitern (siehe Kapitel 5.1.1).

2.2 Modellierungstechniken

Im folgenden sollen Techniken und Vorgehensweisen vorgestellt werden, die zur Erstellung eines *Objektmodells* für das Management eines Switch führen. Dabei wird zwischen den Begriffen *Informationsmodell* und *Objektmodell* unterschieden. Daran anschließend wird eine Einführung in die *Object Modeling Technique* (OMT) gegeben, die in dieser Arbeit zum Einsatz kommt.

2.2.1 Erstellen eines Informationsmodells

Das Informationsmodell stellt nach [HA93] die Abstraktion einer *Ressource* für Managementzwecke dar. Es definiert die Managementinformation und Managementfunktionalität der Ressource an ihrer *Managementschnittstelle*. In das Modell werden die für das Management der Ressource benötigten Eigenschaften aufgenommen.

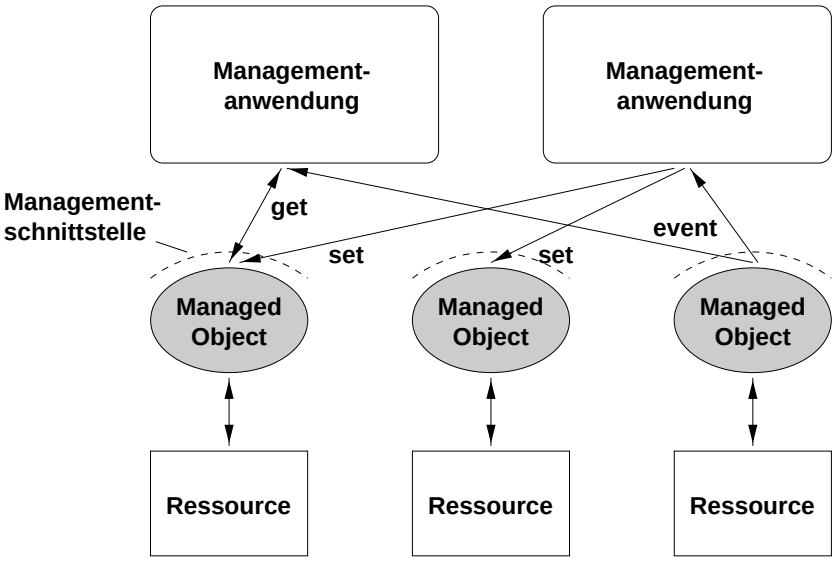


Abbildung 2.5: Managementschnittstelle zu einer Ressource

Eine *Managementanwendung* (Manager) operiert über eine Managementschnittstelle auf einem *Managed Object* (Agent). Das Managed Objekt abstrahiert die für das Management notwendigen Eigenschaften einer Ressource (siehe Abbildung 2.5).

Die Schnittstelle zwischen Managementanwendung und Ressource muß dabei folgendes leisten:

- Abfrage (polling) von Statusinformationen, Leistungsdaten, etc.,
- Aktives Einwirken auf die Ressource: Setzen von Konfigurationsparametern, Veranlassen einer Zustandsänderung (*start/stop*) und
- Senden von asynchronen Ereignismeldungen *notifications/events* an den Manager.

Um die Anforderungen, die sich aus dem Management dieser Ressource ergeben, zu erörtern, ist ein für das Management der Ressource relevantes Modell zu erstellen . Für den PC-basierten Switch wird in Kapitel 4 ein Modell seiner Ressourcen erstellt. Aus diesem Modell soll die spätere Entwicklung seiner Managementschnittstelle ersichtlich werden.

Zur Festlegung eines Informationsmodells muß eine Modellierungstechnik und eine geeignete Syntax zur Beschreibung der Managementinformation gewählt werden. Im Managementbereich werden v. a. folgende Techniken eingesetzt:

- Entity-Relationship Ansatz,
- Datentypansatz und
- objektorientierter Ansatz.

Am leistungsfähigsten hat sich dabei der objektorientierte Ansatz gezeigt. Hierbei werden die Ressourcen durch Managementobjektklassen repräsentiert. Die Managementanwendungen operieren in diesem konkreten Fall über die Managementschnittstelle auf Instanzen der Objektklassen, den Managed Objects.

Managementinformationen für die Objekte werden dabei über Attribute der Instanz einer Klasse modelliert. Ebenso wird die Steuerung der Managementfunktionalität über Methoden der Klasse realisiert. Attribute können durch den Manager abgefragt und unter Umständen geändert werden. Sie legen die Eigenschaften und damit das Verhalten einer Ressource fest. Die Methoden definieren Aktionen, die auf dem Objekt ausgeführt werden können. Asynchrone Ereignismeldungen (*Events/Notifications*) informieren über zur Laufzeit aufgetretene Ereignisse, die das Managed Object an den Manager senden kann. Tritt innerhalb des Managed Object ein für das Management relevantes Ereignis auf, so wird unaufgefordert eine Ereignismeldung (*Notification*) an den Manager geschickt.

Solche Meldungen lassen sich in Hinblick auf das Management folgendermaßen klassifizieren:

Statusmanagement: Ändert der Switch seinen Betriebszustand, so wird das dem Manager angezeigt. Ebenso ist es sinnvoll, den Betriebszustand einzelner Ports bekannt zu geben.

Schließt sich ein neues System an den Switch an, so wird diese Information an die Managementsoftware weitergeleitet.

Sicherheitsmanagement: Versucht sich ein System an einen für ihn unzugänglichen Port anzuschliessen, so kann das über die Eventbehandlung dem Manager signalisiert werden.

Ein objektorientiertes Modell unterstützt Enthaltenseins- und Vererbungshierarchien zwischen den Klassen. *Vererbung* bedeutet, daß eine Unterklasse einige, aber nicht unbedingt alle Attribute und Methoden ihrer Oberklasse übernimmt. Darüberhinaus kann die Unterklasse eigene Attribute und Methoden definieren und somit das Objekt spezifischer beschreiben. Ist ein Managed Object aus mehreren anderen Objekten aufgebaut, spricht man von *Enthaltensein*.

Der Top-Down Ansatz bei der Modellierung eines Objektmodells unterstützt die Vererbungshierarchie besonders gut. So ist es möglich, generische Klassen einzuführen, die Attribute und Methoden definieren, die sich besonders für eine Gruppe von Ressourcen eignen, dessen Eigenschaften sich ähneln. Konkrete Eigenschaften, die nur eine Ressource besitzt, werden durch Spezialisierung der generischen Basisklassen modelliert. So besitzt bspw. jeder Port eines Switch und jedes daran angeschlossene Endsystem einen eindeutigen Identifikator. Diesen Identifikator definiert man als ein Attribut einer generischen Basisklasse (siehe dazu auch Kapitel 4.3).

2.2.2 Quellen für ein Informationsmodell

Nach [HA93] gibt es sechs Quellen für Managementinformationen, die den Inhalt eines Informationsmodells beeinflussen:

- Managementanwendung,
- Netzbenutzer,
- Netzbetreiber,
- Kommunikationsprotokolle,
- Hersteller und
- Netzkomponenten.

Während die ersten drei Quellen einen sogenannten Top-Down Ansatz darstellen, handelt es sich bei letzteren um einen Bottom-Up Ansatz (Abbildung 2.6).

Beim Top-Down Ansatz steht die Frage nach den Informationen im Vordergrund, die das Informationsmodell beinhalten soll, damit das Management die

geforderte Leistung erfüllen kann. Hierzu zählen die Anforderungen, die die Managementanwendung, die Netzbenutzer und Netzbetreiber an das Management stellen. Im Gegensatz dazu steht der Bottom-Up Ansatz, der behandelt, welche Informationen das zu managende System überhaupt in der Lage ist zur Verfügung zu stellen. Das wiederum ist abhängig von den verwendeten Kommunikationsprotokollen und Netzkomponenten. Darüberhinaus muß auf eventuelle proprietäre Lösungen der Hersteller geachtet werden.

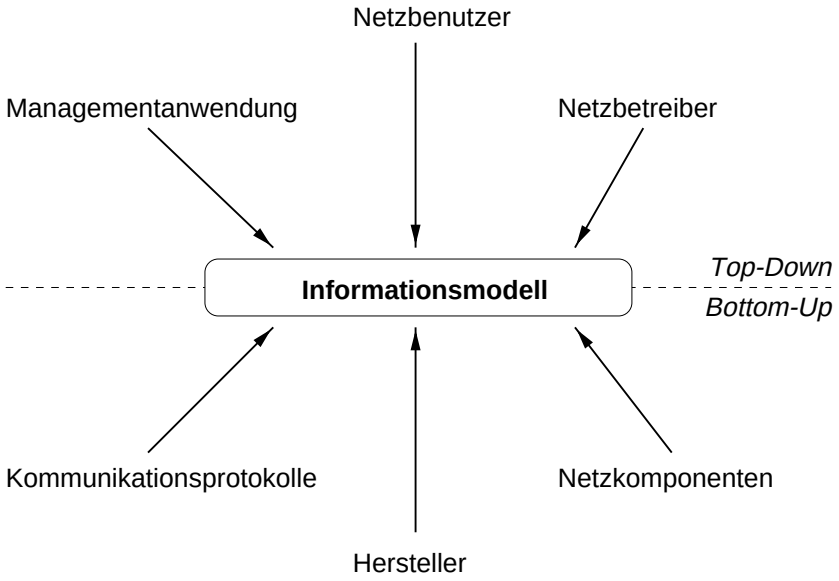


Abbildung 2.6: Quellen für ein Informationsmodell [HA93]

2.2.3 Erstellen des Objektmodells

Die Erstellung eines Objektmodells läßt sich in vier Schritte unterteilen. Zur Top-Down Modellierung gehören die Analyse, eine Art Stoffsammlung und die Modellierung. Betrachtet man den Bottom-Up Ansatz, so kommen im nächsten Schritt die Spezialisierung und die Abbildung auf die Managementarchitektur hinzu.

Analyse

Ziel der Analyse ist es, Objekte und Klassen zu finden, um ein geeignetes Modell der realen Welt unter Berücksichtigung der aktuellen Problemstellung entwickeln zu können. Dazu werden die Objkete durch Relationen in Beziehung zueinander gesetzt. Betrachtet man dieses Vorgehen unter dem Gesichtspunkt des Switch-Managements, so soll die Analyse geeignete Managementobjektklassen liefern, die den Anforderungen des Betreibers genügen. Hierzu ist es notwendig, die Eigenschaften und Funktionen eines PC-basierten Switch in Abhängigkeit

der für den Systemadministrator wichtigen Managementinformationen zu betrachten und diese in Beziehung zueinander zu setzen. Im Vordergrund steht hierbei die Frage nach dem, was wünschenswert ist und nicht, wie es durchsetzbar ist.

Zusätzlich soll sich das Modell an die Anforderungen, die sich durch das Einsatzszenario ergeben, anpassen. So ist z. B. wichtig, neben der Konfiguration des Switch auch die Möglichkeit zu bieten, dessen Betrieb und Durchsatz zu überwachen. Ebenso muß der Switch asynchrone Meldungen über bestimmte Ereignisse an einer spezifizierten Schnittstelle zu Verfügung stellen.

Am Ende der Analysephase soll eine Menge von Objektklassen stehen, die obigen Anforderungen möglichst umfassend genügen.

Modellierung

Die ungeordneten Basisklassen aus der Analysephase sollen in diesem Schritt zu einem Objektmodell zusammengefügt werden. Eine übliche Methode dies zu realisieren, ist die Verwendung der *Object Modeling Technique* (OMT). Die OMT bietet unter anderem Relationen an, um die einzelnen Objektklassen zueinander in einem Objektmodell in Beziehung zu setzen. Hierzu gehören Assoziation, Enthaltenssein- und Vererbungshierachien.

Die OMT ist weit verbreitet und entsprechend etabliert. Eine gute Dokumentation [R+93] und die rechnerbasierte Unterstützung durch das CASE-Tool *Software through Pictures* (StP) machen die OMT zum Mittel der Wahl. Darüberhinaus sind schon einige Arbeiten an diesem Lehrstuhl auf Basis von der OMT entwickelt worden, so daß für diese Technik genügend Wissen zur Verfügung steht.

StP unterstützt die Erstellung von OMT-Modellen und kann aus diesen Codegerüste für diverse objektorientierte Programmiersprachen erstellen. Auf StP wird im Kapitel 5.2.1 noch näher eingegangen.

Ein wichtiger Punkt der Modellierung ist die Ordnung und Bereitstellung von wenigen Basisklassen. Diese sollen als Wurzel des Vererbungsbaumes dienen. Daher kommt es wesentlich auf den Inhalt dieser Klassen an. Sie sollen soviel Information bereitstellen, um eine möglichst große Anzahl verschiedener, abgeleiteter Ressourcen gleichartig behandeln zu können. Die Anwendbarkeit auf unterschiedliche Objekte soll hierdurch aber nicht eingeschränkt werden.

Spezialisierung

In diesem Schritt sollen die generischen Basisklassen verfeinert werden. War die bisherige Modellierung noch weitestgehend systemunabhängig, kommen nun architektur- und systemabhängige Aspekte hinzu. Die Spezialisierung soll zu einem Modell für den hier verwendeten PC-basierten Switch führen.

Von den entwickelten Basisklassen werden hierzu Subklassen abgeleitet und um systemspezifische Attribute und Methoden ergänzt. Hierzu gehört die Analyse konkret benötigter Ressourcen in einem Bottom-Up Ansatz. Die für das Management des konkreten Einsatzszenarios benötigten Informationen und Funktionalitäten werden zusammengetragen und in Form von Attributen und Methoden in einer Subklasse den ererbten Attributen und Methoden der Basisklasse hinzugefügt.

Die speziellen Subklassen sollen die Konstruktion von Werkzeugen erlauben, die auf den konkreten Anwendungsfall abgestimmt sind.

Abbildung auf die Managementarchitektur

Das Objektmodel soll sich unabhängig von der Managementarchitektur umsetzen lassen. Nach [Mül98] sind die drei wichtigsten Managementarchitekturen das *OSI-Management*, das *Internet-Management*, besser bekannt unter dem Namen *SNMP*, und das CORBA-basierte Management.

Allen gemeinsam ist die Aufteilung des Managements in Manager und Agenten (siehe auch Kapitel 2.2.1). Der Agent befindet sich bei diesem Modell auf der zu managenden Ressource und führt Operationen auf dessen Attributen aus. Im allgemeinen gibt es für jede Ressource einen eigenen Agenten. Der Manager überwacht und steuert die Operationen der Agenten.

Wie bereits erwähnt steht diese Arbeit im Rahmen eines größeren Management-Projekts. Zentraler Bestandteil dieses Projekts ist die Entwicklung von Managementanwendungen auf objektorientierter Basis mittels verteilter Agenten. CORBA stellt hierbei die Infrastruktur zur Kommunikation verteilter Objekte durch den ORB bereit (siehe 2.3). Die Sprache IDL beschreibt die Schnittstellen. Das CASE-Tool StP kann aus dem entwickelten OMT-Objektmodell direkt IDL-Code generieren. Ein spezielles Werkzeug erzeugt im nächsten Schritt aus der IDL-Beschreibung generische Java-Klassen, die zur Implemetierung des Objektmodells dann noch mit geeigneten Methoden versehen werden müssen. Im Kapitel 5 wird dieser Vorgang und die Einordnung der entwickelten Klassen in die Rollen von Manager und Agent näher beschrieben.

2.2.4 OMT

Die OMT (*Object Modeling Technique*) nach James Rumbaugh [R+93], Michael Blaha, William Premerlani, Frederick Eddy und William Lorensen ist eine der gebräuchlichsten Methoden zum objektorientierten Modellieren. Dieses Verfahren knüpft an bewährte Entwurfstechniken an, wie Entity-Relationship, Zustands- und Datenflußdiagramme, die auf das objektorientierte Paradigma zugeschnitten werden. Dieser Ansatz — Erweiterung von bereits Angewandtem um moderne Verfahrensweisen — ist wohl auch der Grund für den Erfolg dieser Methode.

Die OMT definiert drei unterschiedliche Modelle. Zur Darstellung der statischen, strukturierten Daten wird das sogenannte *Objektmodell* verwendet. Weiter wird zum Beschreiben des zeitlichen Verhaltens, der Kontrollaspekte, das *Dynamikmodell* und für die transformatorischen Funktionsaspekte das *Funktionsmodell* verwendet.

Das Objektmodell beschreibt, **was** implementiert werden soll, das Dynamikmodell zeigt an, **wann** etwas geschieht, also die Zustände des Systems, und das Funktionsmodell stellt dar, **wie** etwas geschieht. Diese drei Diagramme werden dabei für die Dokumentation der Analyse, des Design und der Implementierung verwendet.

Das Objektmodell

Im Objektmodell werden die Objekte und Klassen des zu modellierenden Anwendungsgebietes dargestellt. Dabei wird zwischen Instanzen- und Klassendiagrammen unterschieden. Einige der Möglichkeiten, die OMT bietet, diese und deren Beziehungen zueinander in einem Modell darzustellen, werden im folgenden betrachtet. Darüberhinaus gibt es noch einige sehr spezielle Konstrukte, die jedoch hier nicht beschrieben werden sollen. Um einen näheren Überblick über die OMT und ihre Notationsmöglichkeiten zu bekommen, soll auf [R+93] verwiesen werden.

a) Klassen und Instanzen

Klassen werden im Objektmodell durch ein Rechteck dargestellt (siehe linke Seite in Abbildung 2.7). Dieses Rechteck wird in drei Bereiche unterteilt. Der obere Bereich enthält zentriert den Klassennamen in Fettschrift. Im mittleren Teil werden die Attribute der Klasse notiert, wobei optional ein Datentyp und ein Start- respektive Initwert angegeben werden kann.

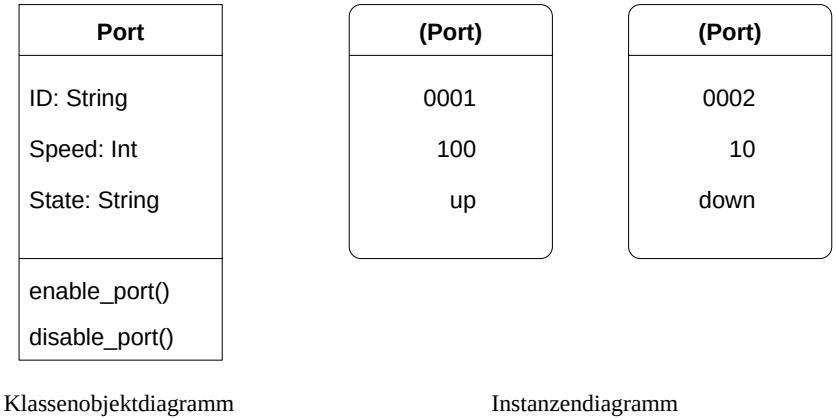


Abbildung 2.7: Notation für die Modellierung von Objektklassen und ihre Instanzen

Mit diesem Startwert wird das Attribut einer Instanz nach dem Erzeugen derselben vorbelegt und erhält damit einen vordefinierten Zustand. Das untere Feld des Klassensymbols enthält die Methoden der Klasse. Hierbei können neben dem Namen der Methode auch eine Parameterliste und der Typ des Rückgabewertes angegeben werden. Die Bereiche für Attribute und Methoden sind optional.

Instanzen werden durch Rechtecke mit abgerundeten Ecken notiert (siehe rechte Seite, Abb. 2.7). Auch hier wird der Name der Instanz zentriert und fett am oberen Rand angegeben; im Falle der Instanz jedoch in runden Klammern. Auf diese Angabe folgen die Werte der Attribute. Die Angabe des Attributnamens vor seinem Wert kann auch entfallen. In diesem Fall bestimmt die Reihenfolge der Werte deren Bedeutung. Insbesondere in größeren Projekten führt dies jedoch zu unübersichtlichen Modellen.

b) Vererbung

Um die Vererbung von Objekten anzuzeigen, wird in der OMT eine Linie zwischen der Kind- und der Elternklasse gezogen. Auf dieser Linie befindet sich ein Dreieck, welches mit der Spitze auf die Elternklasse zeigt. Existieren mehrere Kindklassen, so wird entlang der zu den Kindklassen weisenden Kante des Dreiecks eine waagerechte Linie gezeichnet, von der die jeweiligen Linien zu den Kindklassen führen (Abbildung 2.8).

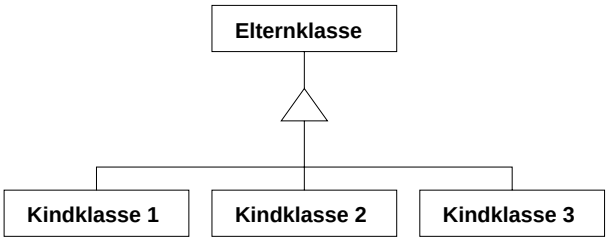


Abbildung 2.8: Notation für die Vererbung von Objektklassen

c) Aggregation

Bei der *Aggregation* handelt es sich um eine Beziehung zwischen Klassen, die sich als eine *Whole/Part-Struktur* darstellen lassen.

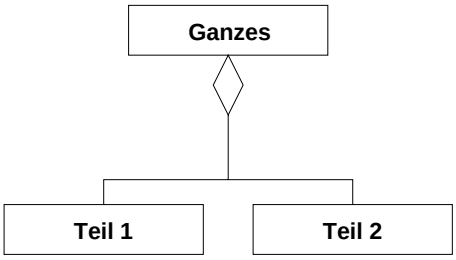


Abbildung 2.9: Darstellung der Aggregation

So ist zum Beispiel ein Port ein Bestandteil eines Switch. Die Aggregation wird bei Rumbaugh durch eine Linie, an deren Ende sich eine Raute befindet, ausgedrückt. Die Raute *hängt* dabei an der Klasse, die das *Ganze* definiert. Bei mehreren Aggregationen können entweder jeweils Einzellinien zu den Teilklassen gezeichnet werden oder ähnlich wie bei der Vererbung alle Teilklassen an einer waagerechten Linie aufgelistet werden (siehe Abbildung 2.9).

d) Assoziation

Ähnlich wie in einem Entity-Relationship Diagramm kann auch in der OMT die Zahl der Teile, die zu einem Ganzen gehören, angegeben werden. Diese *Kardinalität* wird durch gefüllte oder ungefüllte Kreise an den Enden der Relation gekennzeichnet. Hat die Relation keinen Kreis, so handelt es sich um eine 1:1 Beziehung. Während der leere Kreis andeutet, daß das Objekt genau einmal oder gar nicht vorhanden sein muß, steht der gefüllte Kreis für keine oder mehr Instanziierungen des Objekts. Die Notation für n:m Beziehungen geschieht durch das Anschreiben der konkreten Zahlen an das betreffende Ende der Relation (Abbildung 2.10).

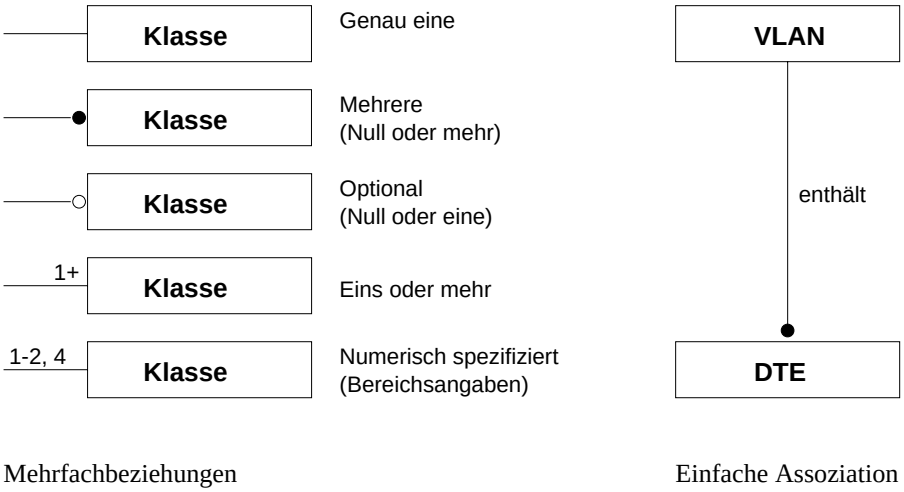


Abbildung 2.10: Darstellung von Mehrfachbeziehungen und Assoziation

Treten mehrere Klassen in Relation zueinander, so gruppiert OMT diese um eine Raute und verbindet die Klassen mit dieser. An den Enden der Verbindungslinien können wieder die Kardinalitäten angegeben werden. Es ist ratsam, darauf zu achten, alles was mehr Beziehungen als eine ternäre Relation eingeht, aufzuspalten und, wenn möglich, in binären Relationen darzustellen, damit das Objektmodell nicht zu unübersichtlich wird.

e) Link-Attribute

Diese sind eine spezielle Art der Assoziation. Da diese Attribute sich nicht einer speziellen Klasse zuordnen lassen, werden sie als *link* zwischen diese Klassen gesetzt. Ein Beispiel hierfür ist der Datendurchsatz auf einer Leitung

zwischen einem Switch und einer DTE. Der Durchsatz läßt sich beiden Seiten zuordnen.

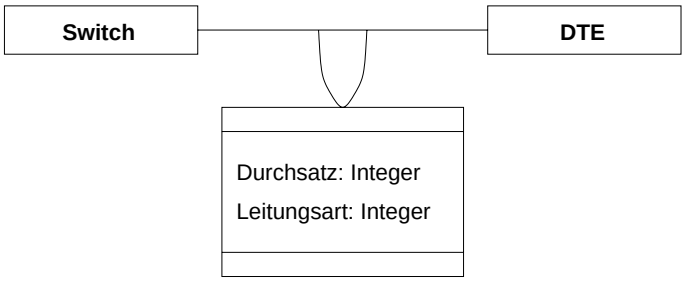


Abbildung 2.11: Darstellung von Link-Attributen

Eine Beziehung durch ein Link-Attribut wird bei einer binären Relation durch eine einfache Linie dargestellt, an der, verbunden durch eine halbe Ellipse, das Link-Attribut eingerahmt durch ein Rechteck angebunden wird. Bei der ternären Relation wird das Link-Attribut an die Raute gehängt. Die Darstellung von Link-Attributen ähnelt der von Objektklassen. Ein Beispiel gibt die Abbildung 2.11.

Dynamik- und Funktionsmodell

Während im Objektmodell die statischen Aspekte des Anwendungsgebietes festgehalten werden, geht es im Dynamikmodell darum, festzustellen, wann sich etwas ereignet. Hier sollen Kontrollstrukturen entwickelt werden, die beschreiben, welche Sequenz von Zustandswechseln auftreten, wenn ein Ereignis auftritt. Ein Ereignis bewirkt in einem Objekt eine Zustandsänderung. Die auftretenden Ereignisse in einem System können hierbei in einem Ereignisdiagramm festgehalten werden.

Darüberhinaus gibt es die Zustandsdiagramme, die ein Mittel zur Darstellung von Zustandsänderungen, die durch die Ereignisse hervorgerufen werden, sind. In einem solchen Diagramm werden alle Zustände einer Klasse eingetragen und durch die Ereignisse, die die Übergänge zwischen diesen Zuständen auslösen, verbunden.

Im Gegensatz dazu beschreibt das Funktionsmodell die transformatorischen Aspekte, also die Änderung der Daten des Systems. Während das Dynamikmodell beschreibt, *wann* etwas geschieht und das Objektmodell zeigt, mit *was* dies geschieht, zeigt das Funktionsmodell, *wie* dieses etwas geschieht. Die Darstellung der funktionalen Sicht wird in Datenflußdiagrammen festgehalten.

Näheres zum Dynamik- und Funktionsmodell läßt sich in der Dokumentation zu OMT [R+93] nachlesen.

2.3 CORBA

Die *Common Object Request Broker Architecture* (CORBA) ist das Produkt eines Konsortiums, der *Object Management Group* (OMG) [OMG], der über 800 Firmen angehören. CORBA definiert einen offenen Standard für verteilte Objekte. Es stellt eine Technik bereit, mit deren Hilfe verteilte Objekte unabhängig von der zugrundeliegenden Netztechnologie, dem verwendeten Betriebssystem und der benützten Programmiersprache Interaktionen in Form von Methodenaufrufen eingehen können. Die OMG gibt dabei lediglich Schnittstellen-Spezifikationen heraus, keinen Code. Die Spezifikationen sind in einer neutralen Sprache, der *Interface Definition Language* (IDL), geschrieben.

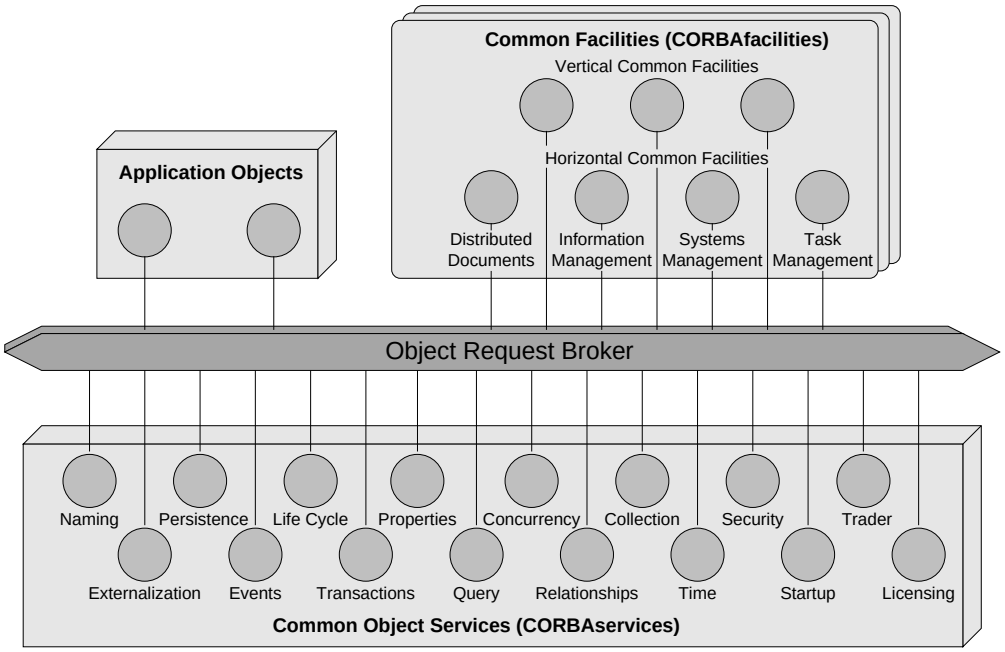


Abbildung 2.12: Object Management Architecture (OMA)

CORBA beschreibt den Standard der *Object Management Architecture* (OMA) (siehe Abbildung 2.12), dessen wichtigste Komponente der *Object Request Broker* (ORB) darstellt. Der Broker ist ein Vermittlungsmechanismus zum Aufruf von Methoden auf entfernten Objekten. Zentraler Bestandteil des ORB ist der *Object Bus*. Er leitet Methodenaufrufe eines Clients einschließlich Parameter an ein Server-Objekt weiter und gibt umgekehrt Ergebnisse oder Fehler zurück. Als ein weiteres Merkmal übernimmt er die Lokalisierung der Server-Objekte. Auf dem ORB bauen sämtliche Komponenten der OMA auf. Die *Object Services* stellen eine Zusammenstellung von fundamentalen Diensten dar, die für die Entwicklung von verteilten, objektorientierten Anwendungen benötigt werden. Insgesamt hat die OMG Standards für 16 Dienste veröffentlicht, darunter Funktionen für das Auffinden von Objekten (*Naming Service*), das Kreieren, Kopieren, Verschieben oder Löschen von Objekten (*Life Cicle Service*), dem Versand von asynchronen Ereignismeldungen (*Event Service*) oder der persistenten Spei-

cherung von Objekten (*Persistence Service*). Die *Common Facilities* standardisieren unabhängig von einem bestimmten Anwendungsbereich infrastrukturelle Dienste wie dem *Information* und *System Management*. An den Standards für die *Common Facilities* wird nach wie vor gearbeitet [OMG, OH97].

Der ORB vermittelt Nachrichten zum Transport von Methodenaufrufen eines Clients an ein Server-Objekt. Im Gegenzug übermittelt er Ergebnisse oder Fehlermeldungen. Für den Client ist es völlig transparent, wo sich das Server-Objekt physisch befindet. Der Client benötigt lediglich eine Referenz auf den Server, anhand derer der ORB die Objekte lokalisiert. Das impliziert, daß die Server-Objekte ihre Verfügbarkeit beim ORB anmelden. In einem großen Netz können mehrere ORB´s an der Nachrichtenvermittlung beteiligt sein. Zur Kooperation der ORB's untereinander müssen sie sich eines gemeinsamen, standardisierten Protokolls bedienen. Seit der Version 2.0 wird diese Interoperabilität zwischen den ORB's verschiedener Hersteller durch das *Internet Inter-ORB Protocol* (IIOP) gewährleistet.

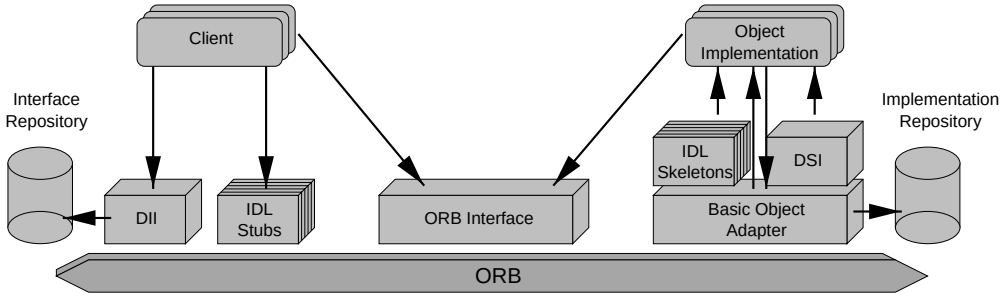


Abbildung 2.13: Struktur des CORBA 2.0 Object Request Brokers (ORB)

Abbildung 2.13 zeigt die Client- und Server-Seiten eines CORBA ORB's. Auf den Kern des ORB's setzen die Objekte, sogenannte *Stubs* auf Client-Seite und *Skeletons* auf Server-Seite, auf. Mit IDL definiert CORBA eine eigene Sprache, die unabhängig von einer bestimmten Programmiersprache ist. Sie ähnelt allerdings bekannten Programmiersprachen wie bspw. C oder C++. IDL bietet ausschließlich Konstrukte zur Beschreibung von Attributen, Methoden und Datentypen von Objekten. Die OMG legt über standardisierte *language mappings* die Abbildung von IDL auf die Implementierungsprache wie Java [Mic, Fla97] oder C++ fest. Mittels dieser Abbildungsvorschrift erzeugt ein IDL-Compiler den *Stub* oder *Skeleton* in der jeweiligen Implementierungssprache.

Die Client Stubs sowie die Server Skeletons bilden eine statische Schnittstelle zur Nutzung von Diensten. Aus Sicht des Clients agiert der Stub als ein lokaler Aufruf. Es stellt einen lokalen *Proxy* für ein verteiltes Server-Objekt dar. Im Gegensatz dazu definiert CORBA das *Dynamic Invocation Interface* (DII), das es Anwendungen erlaubt, dynamisch zur Laufzeit Dienste vom Server-Objekt in Anspruch zu nehmen. Zu allen registrierten Server-Objekten enthält das *Interface Repository* (IR) die IDL-Beschreibungen ihrer Schnittstellen. Mit Hilfe dieser Beschreibungen kann ein Client dynamisch über das DII Methodenaufrufe an den Server senden. Das *ORB Interface* besteht aus einigen wenigen

für die Anwendungen interessanten Funktionen. Zum Beispiel stellt CORBA Funktionen zur Verfügung, die Objekt-Referenzen in Strings umwandeln und umgekehrt.

Auf Server Seite ist das *Dynamic Skeleton Interface* (DSI) das Äquivalent zum DII. Nützlich ist dieses Interface v. a. für die Bildung von generischen Brücken zwischen den ORB's. Das DSI kann entweder statische oder dynamische Aufrufe des Clients empfangen. Der *Basic Object Adapter* (BOA) hat seinen Platz an der Spitze der zentralen Kommunikationsdienste des ORB. Er nimmt Anfragen für Dienste im Auftrag der Server-Objekte entgegen. Der BOA bietet eine Laufzeitumgebung für die Instantiierung der Server-Objekte, stellt Anfragen an sie und ordnet ihnen Objekt ID's, sogenannte *Object References* in CORBA, zu. Ebenso registriert der BOA alle von ihm unterstützten Klassen und ihre Instanzen im sogenannten *Implementation Repository*. CORBA verlangt, daß jeder ORB einen BOA zur Verfügung stellt.

Die Motivation, CORBA statt SNMP für das Management einzusetzen, ergibt sich aus den Vorteilen von CORBA. So beschränkt sich das Management von SNMP auf das Lesen und Schreiben einfacher Variablen. SNMP bietet keine komplexen Datentypen. Die Realisierung von Beziehungen zwischen Objekten kann nur über Indexvariablen bzw. Indextabellen verwirklicht werden. CORBA hingegen erlaubt durch die Sprache IDL eine objektorientierte Modellierung der Ressourcen. Mit IDL lassen sich auch komplexere Datenstrukturen modellieren. Ein weiterer Vorteil ist die Vererbung. Mit ihr lassen sich definierte Informationen und Funktionalitäten wiederverwenden. Im Gegensatz zu SNMP, das Tabellen nur sequentiell auslesen kann, gewährt CORBA einen wahlfreien Zugriff auf die Attribute eines Objekts. Die Beschreibung von CORBA-Objekten in IDL ist unanhängig von der Programmiersprache. Dadurch ermöglicht der ORB den Austausch von Objekten, die in unterschiedlichen Programmiersprachen implementiert worden sind. Die Verwendung von Java als Implementierungssprache erleichtert weitergehend die Entwicklung von portablen Managementanwendungen und deren grafischer Oberfläche. Nicht zuletzt bietet CORBA mit seinem *Event Service*, der im Kapitel 2.3.1 behandelt wird, ein leistungsstarkes Kommunikationsmechanismus für den Versand von asynchronen Meldungen.

2.3.1 Eventbehandlung

CORBA bietet für die asynchrone Kommunikation zwischen Manager und Managed Object den *Event Service* an. Dieser entkoppelt die Kommunikation zwischen Objekten durch die Einführung eines *Event Channels*. Es stellt ein *Supplier-Consumer* Kommunikationsmodell zur Verfügung, das mehreren Sendern (*Suppliers*) das Verschicken asynchroner Meldungen an mehrere Empfänger (*Consumers*) erlaubt.

Abbildung 2.14 verdeutlicht dieses Verfahren. Die auf der linken Seite abgebildeten drei Supplier Objects kommunizieren über den Event Channel mit den

beiden Consumer Objects auf der rechten Seite. Der Event Channel wird von den Supplier Objects mit Daten versorgt, die die Consumer Objects verarbeiten.

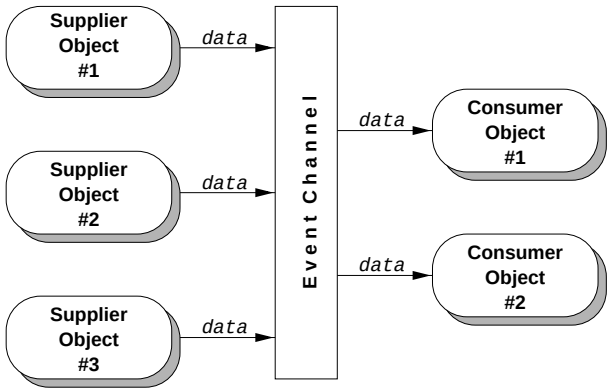


Abbildung 2.14: Das Supplier-Consumer Kommunikationsmodell [Vis97c]

Der Event Channel ist sowohl Sender als auch Empfänger von Nachrichten. Für die Kommunikation zwischen Sender und Empfänger steht einzig die CORBA *Any*-Klasse zur Verfügung. Die Kommunikation über den Event Channel geschieht über CORBA-Aufrufe (siehe Kapitel 5.1).

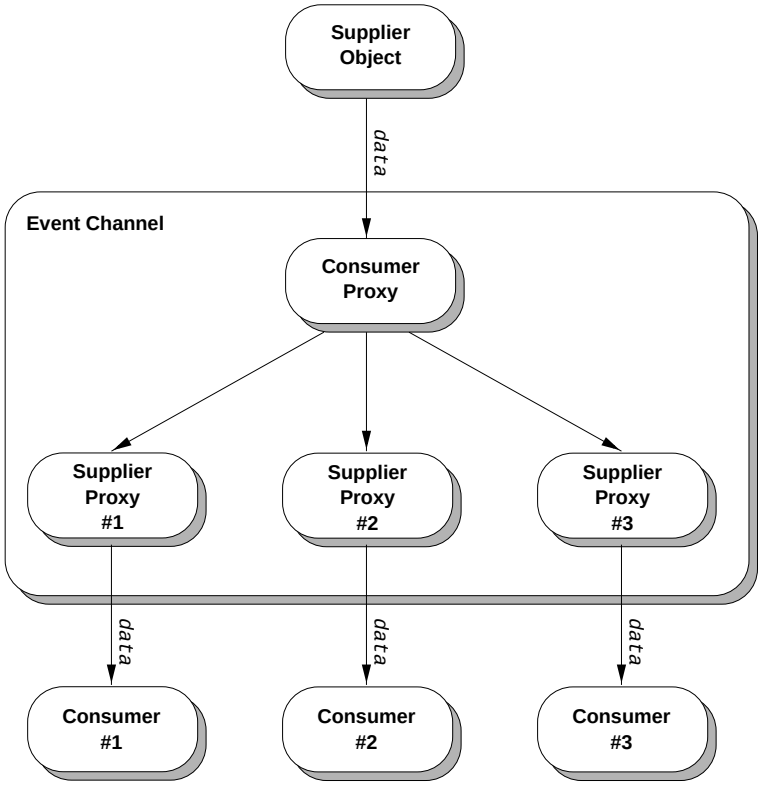


Abbildung 2.15: Sender-Empfänger Proxy Objekte

Die Entkopplung von Sender und Empfänger durch den Event Channel basiert auf der Einführung von *Proxy Objects*. Statt direkt mit dem Gegenüber zu interagieren, erhalten beide ein Proxy Object vom Event Channel und kommunizieren mit diesem. Jedes Sender-Objekt erhält einen Empfänger-Proxy und jedes Empfänger-Objekt einen Sender-Proxy. Der Event Channel kümmert sich um den internen Datenaustauch zwischen den Proxy-Objekten. Abbildung 2.15 zeigt, wie ein Sender mehrere Empfänger mit Daten versorgt.

Der Event Service stellt zwei Kommunikationsmodelle zur Verfügung: das *Push-* und das *Pull-Modell*. Beim Push-Modell initiiert der Sender die Kommunikation, indem er die Daten an den Event Channel sendet. Dieser schickt die Daten unaufgefordert an die Empfänger. Im Gegensatz dazu fragt der Event Channel beim Pull-Modell regelmäßig bei den Agenten nach, ob Nachrichten vorliegen. Der Event Channel puffert diese Daten und gibt diese bei Anfrage an den Manager weiter. Nach [Vis97c] ist das Push-Modell das üblichere Verfahren der beiden.

2.3.2 Der Notification Service

Der Notification Service ist eine Erweiterung des Event Service, auf den im vorigen Kapitel eingegangen wurde. An dieser Erweiterung, die noch nicht standardisiert ist, haben sich eine Reihe von Firmen beteiligt. Einige der bekannteren sind unter anderem Borland International, Fujitsu Limited, NEC Corporation und IBM. Die vorliegende *Joint Revised Revision* [OMG98] sieht folgende Erweiterungen vor:

- Die Möglichkeit, Ereignisse in Form einer wohldefinierten, festgelegten Datenstruktur zu senden. (Der Event Channel bietet im Gegensatz dazu nur den CORBA *Any*-Datentyp an.)
- Die Clients können selbst genau festlegen, welche Ereignisse sie empfangen wollen. Ermöglicht wird dies durch Filtereinstellungen am Proxy Supplier.
- Die Supplier sind nun in der Lage festzustellen, welche Consumer sich an den Event Channel gebunden haben und welche Event-Typen von diesen angefordert werden. Supplier können somit Events auf Abfrage liefern oder das Verschicken von Events unterlassen, an denen kein Consumer Interesse hat.
- Die Consumer ihrerseits können feststellen, welche Event-Typen von den Suppliern an den Event Channel offeriert werden und so bei Bedarf neue Events „abonnieren“.
- Die Möglichkeit, sogenannte *Quality of Service*-Parameter (QoS) auf Basis des Event Channels, der Proxy-Objekte als auch auf Basis der Events zu definieren. Ein QoS-Paramter ist bspw. die Verlässlichkeit (reliability) der Verbindung, die den Transport der Ereignismeldungen zwischen Client und Server garantiert.

- Das Bereitstellen eines optionalen *Event Type Repository* mit dessen Hilfe es dem Benutzer erleichtert wird, Filtereinstellungen auf Basis der zur Verfügung gestellten Informationen über die Ereignisstruktur zu tätigen.

Im Rahmen dieser Arbeit soll die Betrachtung der wohldefinierten Datenstruktur der Ereignisse des Notification Service im Mittelpunkt stehen. Grund hierfür ist, daß es einerseits noch keine Implementierung des Notification Service gibt[2] und andererseits der Notification Service dem Event Service sehr ähnlich ist. Weitergehende Unterschiede und nähere Information zu diesen kann in der Spezifikation zum Notification Service [OMG98] nachgeschlagen werden.

Der CORBA Event Service unterstützt zwei Arten der Eventkommunikation: typisierte und untypisierte Events. Letzterer wurden schon im vorigen Kapitel angesprochen und stellen die sogennannten Any-Datentypen dar, die leicht zu implementieren sind. Viele Anwendungen brauchen hingegen strenger typisierte Ereignismeldungen. Der OMG Event Service definiert gewisse Schnittstellen und Konventionen für typisierte Ereignismeldungen.

Aus diesem Grund bietet der Notification Service eine wohldefinierte Event-Datenstruktur, der als *Structured Event* bezeichnet wird. Im Gegensatz zum Event Service ist es nicht mehr nötig, dieses Konstrukt in eine Any-Datenstruktur umzuwandeln, bevor es an den Event Channel geschickt wird. Leider zwingt uns jedoch das Fehlen einer Implementation des Notification Service noch zu dieser Datenkonvertierung.

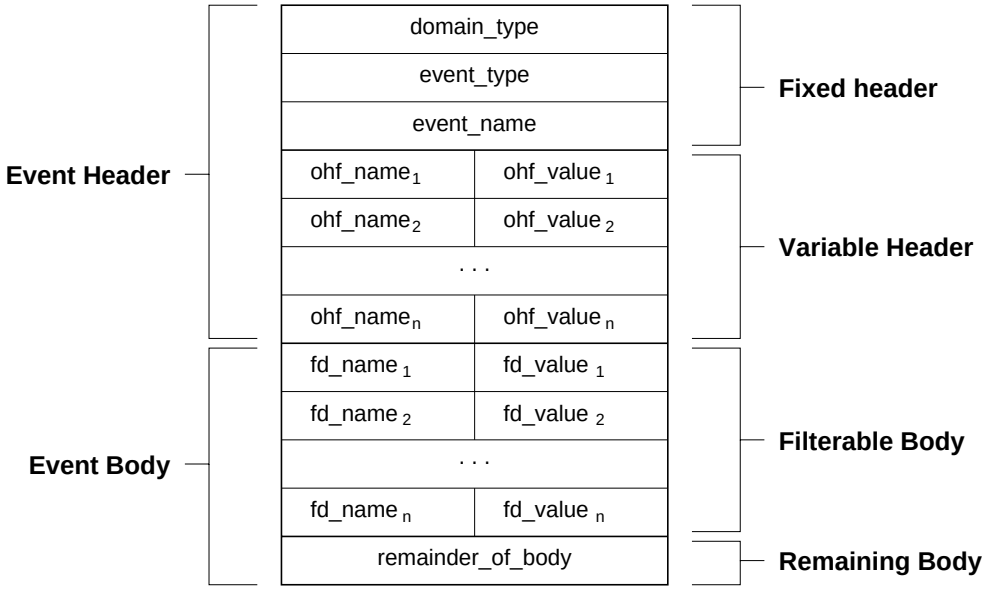


Abbildung 2.16: Aufbau des Structured Events

Der Aufbau des Structured Event, wie er in Abbildung 2.16 zu betrachten ist, besteht aus zwei Hauptbestandteilen:

[2] In [Mau98] wird der Notification Service implementiert.

Der Event Header wird aufgeteilt in einen festen (*fixed header*) und einen variablen (*variable header*) Teil. Der Vorteil dieser Zerlegung liegt in der Minimierung der Headerinformation, die bei jedem Structured Event benötigt wird und somit der Vermeidung von überflüssigem Overhead.

Der Fixed Header besteht aus drei Feldern vom Typ String: dem *domain_type*, *event_type* und *event_name*. Der domain_type bezeichnet die vertikale Industriezugehörigkeit und kann Werte wie beispielsweise *Telekommunikation*, *Finanz-* oder *Gesundheitswesen* annehmen. Der event_type spezifiziert den Typ der Ereignismeldung und wird somit sinnvollerweise mit Werten wie etwa *AgentUp* oder *AgentDown* belegt[3]. Das event_name-Feld identifiziert die Instanz des Events eindeutig.

Der variable Teil setzt sich aus Null oder mehreren Bezeichnern (*ohf_name*) und deren Wertebelegung (*ohf_value*) zusammen. Die Bezeichner sind hierbei vom Datentyp String und die Wertebelegung vom CORBA-Datentyp Any[4].

Der Event Body ist ebenfalls zweigeteilt. Der *filterable body* soll die wichtigeren Felder des Ereignisses enthalten. Er ist so angeordnet, daß er es den Consumern erlaubt, Filtereinstellungen vorzunehmen. Gleich dem *optional header field* teilt sich der *filterable body* in ein Attributfeld vom Datentyp String (*fd_name*) und einer Wertbelegung vom Datentyp Any (*fd_value*) auf.

Der letzte Teil des Body's eines *Structured Events*, der remainder_of_body, ist vom Datentyp Any. Vorgesehen ist dieses Feld für Daten, die sich aufgrund ihrer Größe nicht zum Filtern eignen oder nicht wichtig genug sind, um sie wohlstrukturiert auszugeben, dem Consumer aber nicht vorenthalten werden sollen. Denkbar ist hier die Ausgabe von Fehlermeldungen bzw. *core*-Files, die Informationen über einen möglichen Programmabsturz liefern.

Um die Vorteile der wohldefinierten Datenstrukturen nutzen zu können und spätere Portierungen vom Event Channel-Konzept zum Notification Service-Konzept möglichst einfach zu gestalten, werden alle Events in ein Structured Event verpackt. So kann ein später hinzukommender Notification Service diese wohldefinierte Datenstruktur weiter verwenden, ohne Supplier und Consumer noch an den Notification Service anpassen zu müssen. Gleichwohl können dann die Filtermöglichkeiten des Notification Service sofort genutzt werden.

[3] Diese beiden Meldungen werden beim Starten sowie Beenden des Switch-Agenten in den *AgentInitChannel* verschickt (siehe Kapitel 5.1).

[4] Das Präfix *ohf_* steht für *optional header field*.

Kapitel 3

Einsatzszenarien

Diese Kapitel befaßt sich mit der Nutzung von VLAN's unter Berücksichtigung von nomadischen Systemen. Hierzu werden verschiedene Einsatzszenarien unter logischen und physischen Gesichtspunkten betrachtet. Dabei sollen Probleme beim Umgang mit nomadischen System in einem offen zugänglichen Netz herausgearbeitet werden.

3.1 Einsatz nomadischer Systeme

Der Einsatz nomadischer Systeme in einer Netzumgebung bringt verschiedene neue Gesichtspunkte im Rahmen des Netzmanagements mit sich:

- Bildung logischer Subnetze auf offenen Netzen zwischen nomadischen Systemen und Servern für spezielle Dienste anhand ausgewählter Kriterien,
- Authentifizierung nomadischer Systeme,
- Autorisierung derselben,
- Konfiguration des nomadischen Systems in Hinblick auf IP-Adressen, Nameserver, Defaultroute etc.

3.1.1 Subnetzbildung

In einem Rechnernetz stehen für gewöhnlich mehrere Dienste zur Verfügung, die von Servern angeboten werden. Neben der Datenhaltung sind dies im allgemeinen Druckdienste, Zugangsvermittler zum WAN, Backupdienste u.s.w. Der Zugang zu diesen Diensten steht allen stationären Systemen frei, soweit die Server ihre Dienste für die stationären Systeme freigeschaltet haben. Dieses System basiert darauf, daß die stationären Systeme im Netz fest bekannt sind. Durch intelligente Subnetzbildung auf der Schicht 3 können die Systeme zu Gruppen zusammengenommen werden, denen jeweils nur festdefinierte Server zur Verfügung stehen. Eine andere Möglichkeit besteht darin, für jeden angebotenen

Dienst eine Liste von Endgeräten zu erstellen, die diesen in Anspruch nehmen dürfen. Die Zuteilung der Dienste zu bestimmten Endgeräten ist dabei abhängig von einer bestimmten *Policy*.

Anders sieht es da bei nomadischen Systemen aus. Durch ihre Möglichkeit, sich überall an einem freien Zugang im Netz anzuschließen, können sie an der physischen Subnetzbildung vorbei Dienste des jeweiligen Netzsegments in Anspruch nehmen oder gar zusätzliche anbieten. Eine Möglichkeit, dies zu unterbinden, ist der Weg weg von einer physischen Aufteilung des Netzes hin zu einer logischen Segmentierung. Erreichen läßt sich das — wie schon Eingangs erwähnt — durch die Einführung von virtuellen LAN's.

3.1.2 Authentifizierung

Die Identifikation eines nomadische Systems beruht zunächst nur auf dessen MAC-Adresse und dem Port, an dem es sich an den Switch angeschlossen hat. Ein Managementsystem muß nun in der Lage sein, aufgrund dieser Information nach gewissen Regeln, den sogenannten *Policies*, das nomadische System einem VLAN zuzuordnen, es abzuweisen oder nach einer stärkeren Authentifizierung zu verlangen[1] Die starke Authentifizierung des nomadischen Systems kann z. B. ein DHCP-Server übernehmen. Leider beschränkt sich dieses Verfahren zur Zeit auf einen theoretischen Ansatz [Dro96]. In diesem Vorschlag bietet DHCP die Möglichkeit, die Authentifizierung von DHCP-Nachrichten mit Hilfe eines Verschlüsselungsverfahrens zu gewährleisten. Dadurch kann die Berechtigung des nomadischen Systems für die Benutzung oder Anbietung bestimmter Dienste feiner granuliert werden.

3.1.3 Autorisierung

Ist das System nun eindeutig authentifziert worden, muß es das Managementsystem einem geeigneten logischen Subnetz zuordnen, in dem es die Dienste in Anspruch nehmen kann, die ihm das Managementsystem zugesteht. Umgekehrt werden einem Server auf diese Weise logische Subnetze zugeordnet, in denen er seine Dienste anbieten kann. Ein Switch bietet die Möglichkeit, virtuelle LAN's auf Schicht 1 oder 2 des OSI-Schichtenmodells zu bilden. Für die Einteilung des nomadischen Systems in ein VLAN auf Schicht 1 oder 2 ist die Information über den Anschlußport bzw. seine MAC-Adresse ausreichend (siehe Kapitel 2.1.4). Beide Informationen liegen beim Anschluß des Systems an einen Port vor. Durch die Zuweisung des Systems an ein bestimmtes VLAN autorisiert es das Managementsystem, bestimmte Dienste in Anspruch zu nehmen oder diese anzubieten.

Eine Alternative wäre es, wie schon erwähnt wurde, für jeden Dienst, den ein Server anbietet, dessen Konfigurationsdateien anzupassen. Dazu muß jeder Cli-

[1] Igor Radisc beschreibt in seiner Arbeit über ein policy-basiertes Konfigurationsmanagement für nomadische Systeme [Rad98] ausführlich diese Regeln.

ent, oder alternativ eine ganze *Domain*, in die Konfigurationsdatei eingetragen werden. Eine weitergehende Gruppierung von Clients bietet das *Network Information Service* (NIS) der Firma SUN Microsystems. Dazu kann man die Clients in sogenannte *netgroups* einteilen. Kann der Server, der für einen Dienst konfiguriert werden muß, mit *netgroups* umgehen, dann können diese Gruppen durch einen Eintrag in seine Konfigurationsdatei(en) für die Benutzung des Dienstes eingerichtet werden. NIS hat allerdings den Nachteil, als nicht besonders sicher zu gelten.

3.1.4 Konfiguration

Um diese Dienste in Anspruch zu nehmen, braucht das nomadische System im allgemeinen noch ein paar Informationen, um in das Netz integriert zu werden. Hierzu gehören seine IP-Adresse, die Adressen der zuständigen Nameserver, Broadcastadresse, Defaultgateway u. a. Die Bereitstellung dieser Informationen kann von einem DHCP-Server übernommen werden, da DHCP im Gegensatz zu anderen Konfigurationsprotokollen wie bspw. *RARP* oder *BOOTP* besser für den Einsatz mit mobilen Systemen geeignet ist [Rie96].

3.2 Beispielszenario

Für das nachstehende Beispiel sind zwei Betrachtungsweisen relevant: die physische und die logische Netzsicht. Erstere beschreibt den tatsächlichen Netzaufbau in seiner physischen Struktur, letztere dagegen die logische Netzaufteilung in Gruppen. Weitere Beispiele liefert [Hei98b] und [Hei98a].

3.2.1 Physische Netzsicht

Ausgangspunkt ist das Szenario eines *Office Parks*, wie er in [GHW98] ausführlich beschrieben wird (Abb. 3.1). An Infrastruktur steht ein zentraler Switch bereit, der über sogenannte *öffentliche* und *nicht öffentliche* Ports verfügt. An diesen Switch werden Netzkomponenten unterschiedlichster Art angeschlossen. Das können bspw. Drucker, Plotter, Datenbankserver oder Notebooks sein.

Das Szenario ist in vier Bereiche, (A – E), aufgeteilt. Bereich A sieht die physische Verkabelung (physical layer) sowie den Betrieb des Switch (data link layer) vor, der den Backbone für die Verbraucher stellt. In Bereich B werden Dienste wie Druckdienste, Backupdienste, Dateidienste sowie Proxydienste für den Zugang zum Internet angeboten. Den Bereich C nimmt eine in sich geschlossene Gruppe von Endgeräten ein, die die Dienste von B in Anspruch nehmen können. Diese Gruppe kann z. B. ein Lehrstuhl oder auch, wie in [GHW98] beschrieben, ein Softwareunternehmen sein, das nur die Dienste von B beanspruchen möchte, ihre eigenen Softwareprojekte aber lokal gekapselt von der Außenwelt und den anderen Bereichen sehen will.

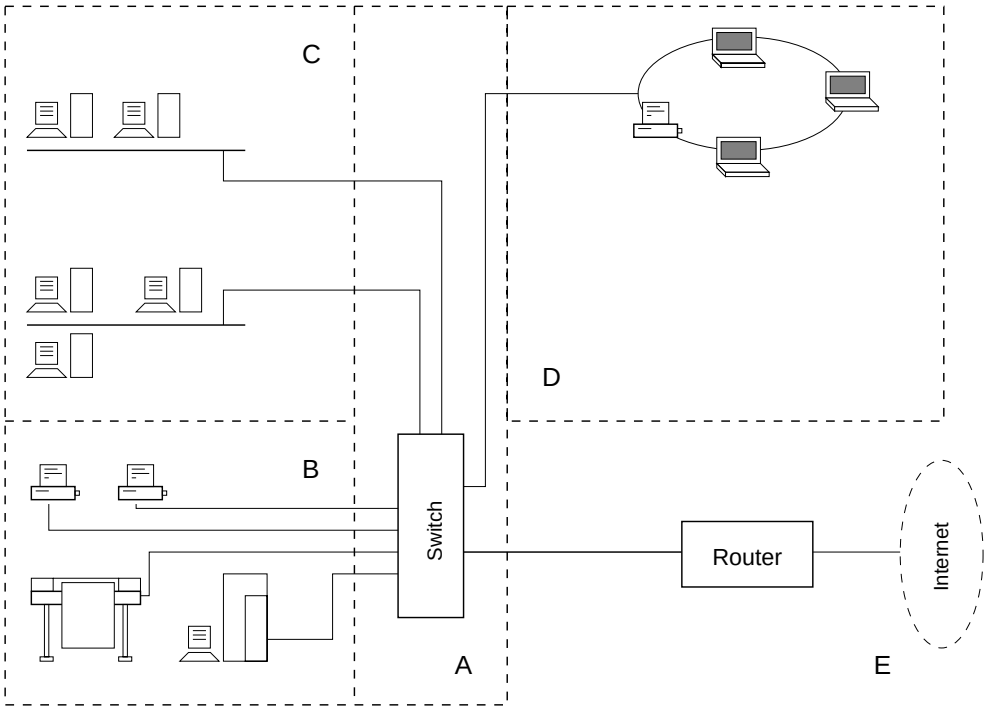


Abbildung 3.1: Office Park-Szenario (vereinfacht nach [GHW98])

In Bereich D dürfen nomadische Systeme angeschlossen werden. Hier müssen zwei unterschiedliche Szenarien unterschieden werden: (a) Nomadische Systeme, die dem Office Park angehören und daher diesem bekannt sind sowie (b) nomadische Systeme der Außenwelt, die innerhalb des Office Park nicht bekannt sind, trotzdem aber (eingeschränkten) Nutzen der angebotenen Dienste in Anspruch nehmen können. Beide Gruppen sollen sich in Bereich D, den man sich als eine Art öffentlicher Arbeitsraum vorstellen kann, an die frei zugänglichen öffentlichen Ports anschließen dürfen. Die Interoperabilität zwischen den Systemen soll in beiden Fällen gewährleistet werden.

Der Router in Bereich E stellt die physische Verbindung zur Außenwelt, bspw. dem Internet, dar. Die räumliche Aufteilung der Endsysteme (DTE), wie sie in der Abbildung 3.1 gezeigt wird, ist keineswegs ein Muß. Durch die logische Subnetzbildung ist der Switch in der Lage, unabhängig von der räumlichen Anordnung der Endsysteme, diese zu Arbeitsgruppen zu gruppieren.

3.2.2 Logische Netzsicht

Gruppiert man die DTE's nun zu logischen Gruppen, denen der Einfachheit halber wieder die Bereiche A – E zugeordnet werden, dann lassen sich für jede dieser Gruppen eine Anzahl von Diensten definieren, die diese in Anspruch nehmen dürfen (Abb. 3.2). Die Einteilung in Gruppen geschieht hierbei mit Hilfe von VLAN's, die der Switch implementiert.

Deutlich wird die Aufteilung der Dienste an die einzelnen Gruppen. Auf der vertikalen Achse sind die angebotenen Dienste aufgereiht, auf der horizontalen Achse die fünf Gruppen. Darf eine Gruppe einen Dienst in Anspruch nehmen, so ist in der betreffenden Spalte und Zeile ein Kästchen eingezeichnet. Der Buchstabe im Kästchen zeigt an, welcher Gruppe die Diensteanbieter angehören, die den benötigten Dienst zur Verfügung stellen.

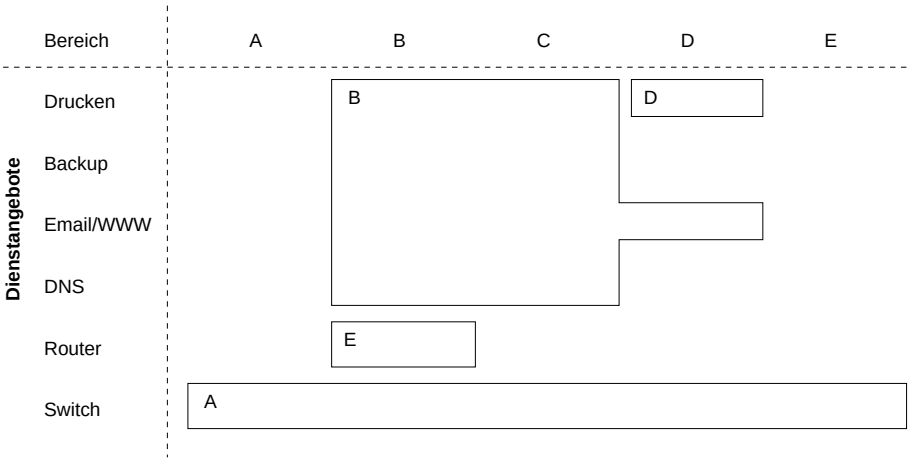


Abbildung 3.2: Dienstangebote

Jedes Gerät hängt am beteiligten Switch und wird von diesem einem oder mehreren VLAN's zugeordnet. Das VLAN bestimmt, welche Dienste in Anspruch genommen werden dürfen. So steht den Geräten des Bereichs D z. B. nur der Email- und Proxydienst für das Internet zur Verfügung. Drucken darf der Bereich D nur auf den Drucker, der sich innerhalb seines eigenen Bereichs befindet. Dem Bereich C stehen ebenso wie dem Bereich B alle Dienste offen. Zusätzlich ist dem Bereich B noch die direkte Anbindung über den Router an das Internet erlaubt, was dem Bereich C nicht gestattet wird.

Bei Betrachtung dieser Hierarchien fällt sofort die Notwendigkeit auf, die Gruppierung der einzelnen Bereiche in VLAN's regelbasiert anzubieten. Für jedes managebare Objekt wie Switch, nomadisches System oder Server müssen sogenannte *Policies* aufgestellt werden, anhand derer die Objekte virtuellen LAN's zugeordnet werden können[2].

Kriterien für den Zusammenschluß in Gruppen von Systemen gibt [GHW98]. Für das Management eines Switches ist der erste Punkt dieser Arbeit entscheidend: die Einteilung von Systemen zu Gruppen aufgrund ihrer MAC-Adresse oder ihres Anschlußports an den Switch.

[2]Die Untersuchung von Policies ist nicht zentraler Bestandteil dieser Arbeit. Siehe dazu [Rad98].

3.2.3 Anschluß von Systemen

In diesem Abschnitt soll der Vorgang beim Anschluß eines Systems an das vorgestellte Netz betrachtet werden. Eine Managementsoftware verwaltet unter anderem den Switch und die von diesem unterstützten VLAN's. Wir unterscheiden zwischen einem nomadischen System und einem statischen System. Notebooks charakterisieren typischerweise erstere Gruppe, während Workstations, Drucker u. a. zweitere Gruppe repräsentieren.

Eine Möglichkeit der Identifikation eines neuen Systems bietet die Authentifizierung über seine IP-Adresse. Von dieser Möglichkeit machen im allgemeinen Serverdienste Gebrauch. Sie halten eine Liste von Systemnamen (engl. *hostnames*), die über den Domain Naming Service (DNS) ihrer IP-Adresse zugeordnet werden, für jeden angebotenen Dienst vor. Nur wenn der Systemname in der Liste steht, kann das zugehörige Endgerät den Dienst in Anspruch nehmen. Ein Nachteil dieser Methode ist, daß sich die IP-Adresse eines Endgeräts beliebig ändern läßt. Dazu sind im allgemeinen nur Superuserrechte auf dem betreffenden Gerät nötig. Durch das Vortäuschen einer falschen Identität kann das Endgerät so an einem Dienst gelangen, den es eigentlich nicht benutzen darf.

Eine andere Möglichkeit, die Identifikation des Systems zu gewährleisten, ist die schon angesprochene starke Authentifizierung über einen DHCP-Server (siehe Kapitel 3.1.2). Dadurch, daß bei diesem Vorgang die Zugangsdaten verschlüsselt übertragen werden, handelt es sich um ein relativ sicheres Verfahren.

Die dritte Möglichkeit ein neues System zu authentifizieren besteht darin, es über seine MAC-Adresse zu identifizieren. Dabei handelt es sich um eine relativ sichere Methode die Identität eines Systems festzustellen, da die MAC-Adresse einer Netzkarte weltweit eindeutig vergeben wird [Hal92]. Aber auch in diesem Fall kann die MAC-Adresse gefälscht werden, wenn auch mit mehr Aufwand als bei der IP-Adresse, weil die MAC-Adresse i. a. von der Netzkarte übernommen wird und es nicht vorgesehen ist, in diesen Algorithmus einzugreifen.

Das dritte Verfahren zur Identifikation von neuen Systemen eignet sich für diese Arbeit am besten, weil ein Switch Frames auf Schicht 2 des OSI-Schichtenmodells forwardet, somit also Kenntnis über die angeschlossenen Systeme anhand ihrer MAC-Adresse hat. Es wird daher näher betrachtet.

Der Switch erkennt anhand der MAC-Adresse, die er mit dem ersten Frame bekommt, das ein neu angeschlossenens System verschickt, daß sich ein neues System an einen seiner Ports angeschlossen hat. Um diese Information auswerten zu können, bedarf es eines Managementsystems, das die gewonnenen Information annimmt, auswertet und anhand gewisser Policies Aktionen tätigt. Der Switch teilt also die Information über den Anschlußport und die MAC-Adresse des neuen Systems einem Managementsystem mit. Dieses kann anhand der gelieferten MAC-Adresse entscheiden, ob das System dem Office Park bekannt ist oder es sich um ein unbekanntes, außenstehendes System handelt (Auswertung). Um das feststellen zu können, braucht das Managementsystem eine Liste aller dem Office Park bekannten MAC-Adressen. Soll eine Zuordnung von

MAC-Adresse zu VLAN bzw. Port zu VLAN stattfinden, muß die Managementsoftware diese Tabellen ebenfalls beinhalten.

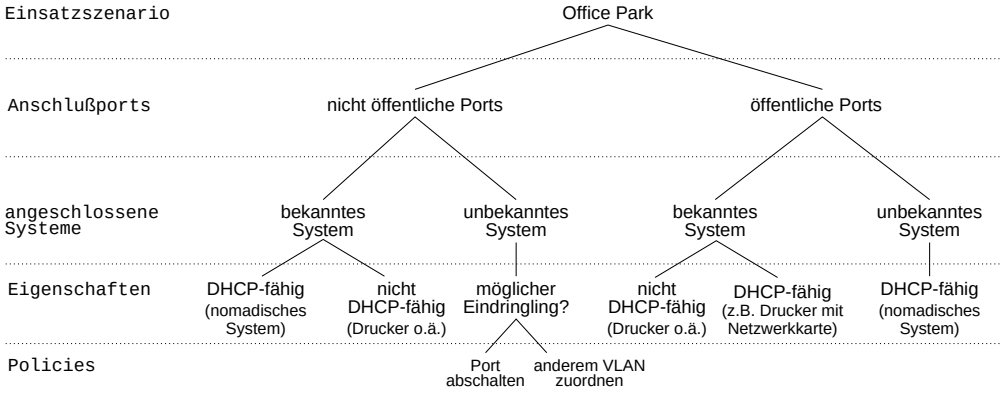


Abbildung 3.3: Anschlußszenario

In unserem Beispielszenario (Kapitel 3.2.1) gibt es zwei unterschiedliche, logische Arten von Ports eines Switch, an denen Endgeräte angeschlossen werden können: *öffentliche* und *nichtöffentliche* Ports. An die öffentlichen Ports darf sich im Prinzip jedes Endgerät anschließen, während die nichtöffentlichen Ports den im Office Park bekannten Systemen vorbehalten sind. Aus den verschiedenen Möglichkeiten der Permutationen von bekannten und unbekannten Systemen an öffentliche und nichtöffentliche Ports gibt es mehrere Fälle zu unterscheiden, die in Abbildung 3.3 grafisch dargestellt sind. Diese Fälle beschreiben unterschiedliche Policies, die sich aus dem Beispielszenario ergeben. Der Switch soll in der Lage sein, die verschiedenen Policies zu unterstützen. Bei der Ausarbeitung der Policies sollen die Fälle der Identifikation eines neuen Systems über seine MAC-Adresse und über starke Authentifizierung mittels DHCP betrachtet werden:

1. Ein bekanntes System schließt sich an einen nichtöffentlichen Port an. Um die Zuordnung zu einem VLAN bestimmen zu können, gibt es zwei Fälle: das System ist DHCP-fähig oder nicht. Im ersten Fall kann es sich über die vom DHCP-Protokoll angebotene starke Authentifizierung beim Managementsystem anmelden. Dieses ordnet das Endgerät anhand seiner Authentifizierung und seinen Policies einem VLAN zu. Ist das System nicht DHCP-fähig, was häufig bei Druckern der Fall ist, muß das Managementsystem aufgrund der fehlenden starken Authentifizierung das System anhand der Information über seine MAC-Adresse und den Anschlußport identifizieren und einem VLAN zuordnen.

2. Ein unbekanntes System schließt sich an einen nichtöffentlichen Port an. Aufgrund des fehlenden MAC-Adressen Eintrags in der Tabelle der Managementsoftware erkennt diese, daß die angeschlossene DTE an diesem Port unerwünscht ist. Möglicherweise handelt es sich sogar um einen Eindringling, der unberechtigterweise Zugriff auf für ihn gesperrte Dienste zu erlangen sucht. Die Managementsoftware muß nun entscheiden, was mit

dem System zu tun ist. Beispielweise könnte sie den Port sperren, oder das unbekannte System einem öffentliche VLAN zuordnen und den Status des Ports entsprechend ändern.

3. Ein bekanntes System schließt sich an einen öffentlichen Port an. Hier kann ähnlich wie bei einem Anschluß an einen nichtöffentlichen Port verfahren werden. Dieser Fall ist v. a. dort interessant, wo bestimmte Dienste einem öffentlichen VLAN zur Verfügung gestellt werden sollen. Ein Beispiel hierfür ist die Bereitstellung eines Drucker für den öffentlichen Arbeitsraum D in unserem Office Park-Szenario.

4. Ein unbekanntes System schließt sich an einen öffentlichen Port an. Dies stellt den Normalfall dar. Voraussetzung für die korrekte Anbindung des Systems an das Netz ist, daß die DTE das DHCP-Protokoll beherrscht, so daß ihr Informationen über ihre IP-Adresse, Nameserver, Broadcast-Adresse u. ä. zugewiesen werden können. Die Managementsoftware teilt die Maschine dann einem vorher festgelegten öffentlichen VLAN zu.

3.3 Das Managementszenario

In Abbildung 3.4 wird die Interaktion zwischen dem Managementsystem und seinen Agenten verdeutlicht. Betrachtet wird der Anschluß einer neuen Komponente an einen Switch. Das Managementsystem besteht aus einem Manager, seinen Agenten und den *Managed Objects*, also dem Switch und die an ihm angeschlossenen Systeme. Zusätzlich steht der Event Service für asynchrone Meldungen der Agenten an den Manager zur Verfügung.

Schließt sich ein neues nomadisches System an den *Switch* an, so versucht es zuerst durch das Senden eines DHCPDISCOVER einen zuständigen DHCP-Server zu erreichen. Der Switch empfängt das Frame und gibt über seinen Agenten (*SwitchAgent*) eine Ereignismeldung an den *Event Channel* ab. Das Ereignis enthält die MAC-Adresse und den Port, an den sich das neue System angeschlossen hat. Der *Manager*, der auf neue Ereignismeldungen vom Event Channel wartet, empfängt die Information und versucht anhand bestimmter Regeln, den sogenannten *Policies* ([Rad98]), das System

a) einem VLAN zuzuordnen und

b) ihm gültige Konfigurationsparameter zu übermitteln.

Kann der Manager das System einem oder mehreren VLAN's zuordnen, so wird der Switch durch einen Methodenaufruf des Managers auf dem Agenten dahingehend konfiguriert. Im nächsten Schritt geht der normale Client/Server-Dialog unter DHCP weiter (siehe [Rie96, Dro93, Dro97]). Die Konfiguration des DHCP-Servers wird dabei von seinem Agenten, dem *DHCP-Agent*, bewerkstelligt. Dieser wiederum bekommt Vorgaben in Form von Methodenaufrufen vom Manager. Die Interaktion zwischen Manager, DHCP-Agent und DHCP-Server wird in [Rad98] und [Dem98] näher beschrieben.

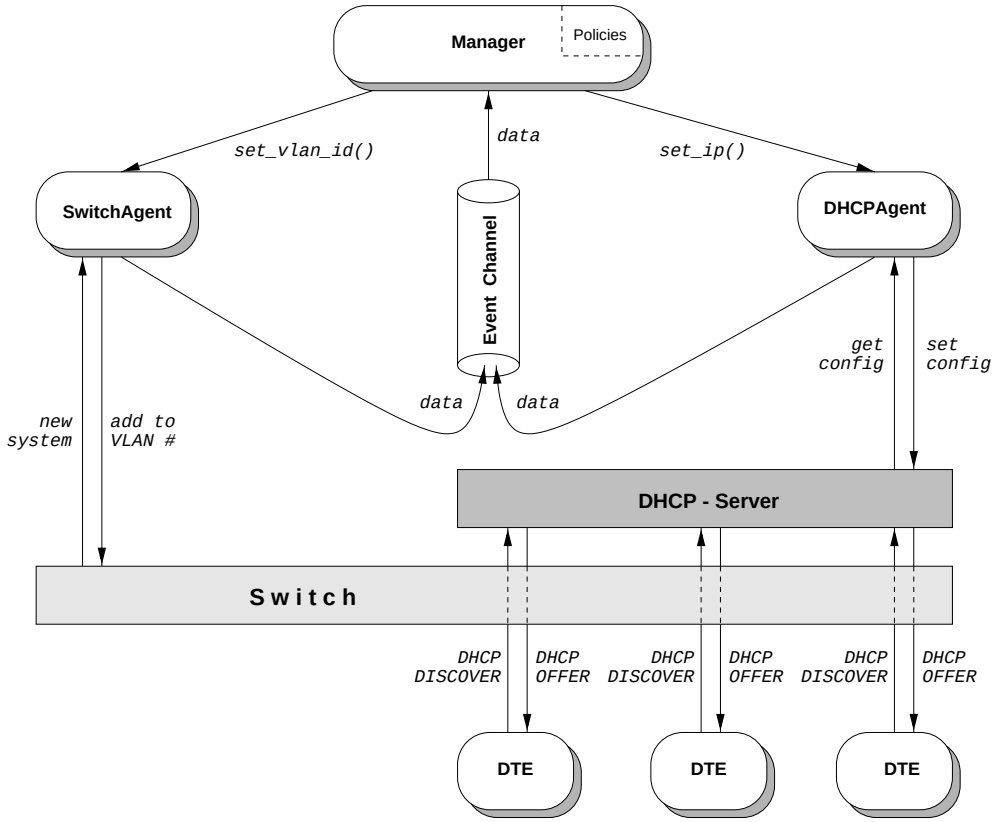


Abbildung 3.4: Managementszenario

Verfügt der DHCP-Server über die Möglichkeit, Systeme anhand starker Authentifizierung mittels DHCP zu identifizieren [Dro96], können diese durch Interaktion mit dem Manager weiteren VLAN's zugeordnet werden. Beim Anschluß von stationären Rechnern oder DTE's, die nicht das DHCP-Protokoll verstehen, fällt der Fall der Authentifizierung mittels DHCP im Managementszenario weg. Stattdessen muß das Managementsystem in der Lage sein, nur mit Hilfe der übermittelten MAC-Adresse das System zu identifizieren und einem VLAN zuzuordnen (Kapitel 3.2.3).

Abbildung 3.5 soll den Anschluß eines neuen Systems an ein geswitchtes LAN in Anlehnung an [Hei97] darstellen. Dabei wird der zeitliche Verlauf der Anmeldung skizziert.

Zu Beginn muß der Switch eine Verbindung von einem Port zum DHCP-Server gewährleisten. Dies kann durch einen Defaulteintrag für bestimmte Ports geschehen. Im Gegensatz zu [Hei97] meldet in diesem Szenario der Switch, der den DHCPDISCOVER des nomadischen Systems an den DHCP-Server weiterleitet, dem Managementsystem das „Erscheinen“ des neuen Systems. Das Managementsystem entscheidet aufgrund seiner *Policies* über die „Zulassung“ des Endsystems unter anderem nach obigen Beispielkriterien (siehe Kapitel 3.2.3). Darüberhinaus teilt es dem DHCP-Server die zu verwendende Konfigurations-

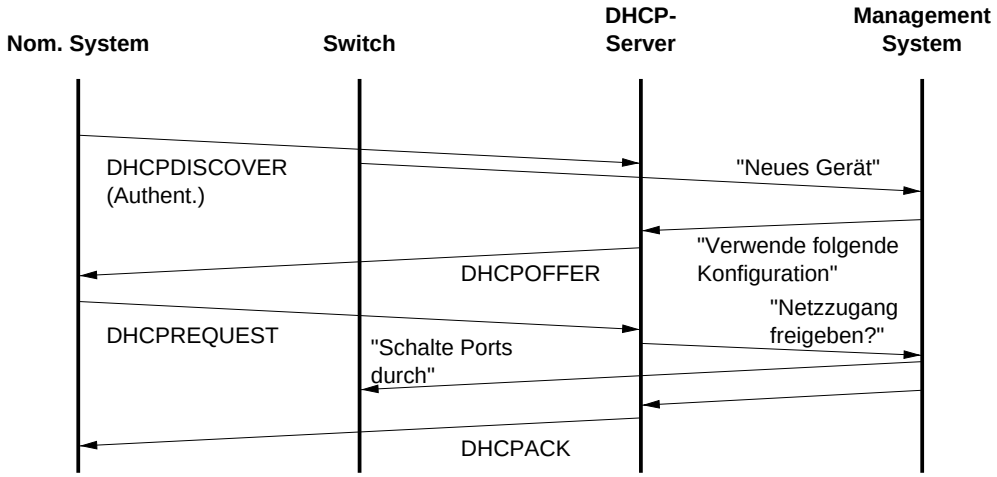


Abbildung 3.5: Zeitlicher Verlauf des Anschlusses eines nomadischen Systems an ein geswitchtes LAN

information für dieses System mit. Erst dann kann der DHCP-Server die Konfiguration mit dem Client aushandeln.

Der DHCP-Server muß aus diesem Grund bis zu diesem Zeitpunkt warten. In der Zwischenzeit könnte der DHCP-Server zum Beispiel nichts tun und nur auf die Beendigung seiner Konfiguration warten, um anschließend einen *DHCPOFFER* an den Client zu schicken. Eine andere Möglichkeit besteht darin, einen *DHCPOFFER* mit einer ungültigen Konfiguration an den Client zu schicken, um Zeit zu gewinnen, und den anschließenden *DHCPREQUEST* des Clients mit einem *DHCPNAK* zu verweigern. Daraufhin muß der Client eine neue Abfrage initialisieren (*DHCPDISCOVER*).

Der DHCP-Client wartet eine gewisse Zeit auf das Eintreffen eines *DHCPOFFER*. Trifft dieses nicht in dieser Zeitspanne ein, so versucht er erneut eine Abfrage (*DHCPDISCOVER*) an den DHCP-Server zu senden. Gleiches gilt für das Warten auf eine Antwort auf einen *DHCPREQUEST*.

Handeln DHCP-Server und Client schließlich eine gültige Konfiguration aus, die das Endsystem akzeptiert, so teilt das Managementsystem das nomadische Endsystem einem VLAN zu. Der Switch schaltet die entsprechenden Ports frei.

Zur Verwirklichung dieses und ähnlicher Szenarien bedarf es eines Managementsystems, das die Steuerung der einzelnen Komponenten übernimmt (siehe dazu [Rad98]). Die technische Realisierung dieser Managementlösung wird auf CORBA aufbauen, das für die Interaktion verteilter Objekte besonders geeignet ist, wie in Kapitel 2.3 herausgestellt wurde.

Kapitel 4

Managementmodelle

Das Ziel dieses Kapitels ist die Erstellung von Managementmodellen auf Basis der in Kapitel 3.2 entworfenen Szenarios. Dabei sollen die in Kapitel 2.2 vorgestellten Modellierungstechniken eingesetzt werden. Einleitend wird erklärt, in welche Funktionsbereiche sich das Management einer Ressource einteilen läßt. Ausgehend davon soll in einer Art Analysephase festgestellt werden, welche Anforderungen an die Funktionalität sich aus den besprochenen Einsatzszenarien unter Beachtung der Managementeinteilung in verschiedene Funktionsbereiche ergeben. Dazu werden die einzelnen Ressourcen, die ein Switch bietet gesammelt. Anschließend wird versucht, die gesammelten Ressourcen den vorher spezifizierten Management-Funktionsbereichen zuzuordnen. Die Modellierung geeigneter Managementmodelle sowie dessen Spezialisierung (siehe Kapitel 2.2.3) im Hinblick auf die Implementierung runden das Kapitel ab. Dazu werden zwei Objektmodelle designt, eines aus Managementsicht und eines aus Sicht der späteren Implementierung. Abschließend wird ein Modell für die Versendung asynchroner Ereignismeldungen betrachtet.

4.1 Management-Funktionsbereiche

In einem ersten Schritt soll nach dem Konzept der Top-Down Modellierung vorgegangen werden. Das Management wird nach OSI in fünf Funktionsbereiche eingeteilt (vgl. [HNW95] und [HA93]), die im folgenden näher betrachtet werden sollen.

Das Konfigurationsmanagement soll durch Beeinflussung von Parametern eine Ressource so einrichten, daß sie in Kooperation mit anderen im Normalbetrieb den gewünschten Dienst erbringt. Neben dem Auslesen der aktuellen Konfiguration müssen auch Änderungen an dieser über die Managementsoftware möglich sein. Die Konfiguration eines Dienstes soll dabei über dienstspezifische Parameter steuerbar sein.

Das Fehlermanagement dient dem Erkennen und Beheben von Fehlern. Dazu gehört auch die Verarbeitung und Diagnose von Fehlermeldungen. Denkbar ist hier z. B. eine Weiterreichung von Fehlern an ein Trouble Ticket System.

Das Leistungsmanagement soll Daten über die Performance eines Systems sammeln und diese auswerten. Ziel ist es, die Gesamtleistung eines Systems beurteilen und durch Optimierung verbessern zu helfen. Bezogen auf einen bestimmten Dienst bedeutet das, daß der Server alle an ihn gestellten Aufgaben mit einer zumindest zufriedenstellenden Antwortzeit bearbeitet. Hieraus ergeben sich Meßgrößen für die Dienstgüte wie z. B. Durchsatz, Auslastung und Antwortzeit, die im laufenden Betrieb ständig überwacht werden müssen.

Das Abrechnungsmanagement protokolliert die Nutzung von Betriebsmitteln. Anhand der Auswertung von Durchsatzraten oder Onlinezeiten kann dann mit den entsprechenden Benutzern abgerechnet werden.

Dem Sicherheitsmanagement gehört unter anderem die Benutzerverwaltung an. Hierzu gehört Authentifizierung der Clients, die bestimmte Dienste nutzen wollen. Darüberhinaus ist die Autorisierung der Clients für bestimmte Dienste ein Aspekt des Sicherheitsmanagements.

4.2 Anforderungen an die Funktionalität

In Kapitel 3.2 wurde das Szenario eines *Office Parks* anhand eines Beispiels verdeutlicht. Dabei wurde zwischen einer physischen und einer logischen Netzsicht unterschieden. Betrachtet man dieses Beispielszenario als zugrundeliegendes Einsatzszenario, so können ausgehend von der physischen Netzsicht drei Managementobjekte lokalisiert werden: der Switch, dessen Ports und die angeschlossenen Systeme, hier DTE genannt.

- Der **Switch** ist das zentrale Gerät. Es dient dem Anschluß weiterer Netzkomponenten.
- Der Switch beinhaltet unter anderem **Ports**. An diese werden die Netzkomponenten physisch angeschlossen.
- An die Ports wiederum werden **DTE**'s angeschlossen. Hierbei kann es sich um Arbeitsplatzrechner, Server, Drucker u. ä. handeln, aber auch nomadische Systeme; weitere Switches und Router sollen in diesem Fall unter die Definition des Begriffs DTE fallen.

Erweitert man das Objektmodell um eine logische Sichtweise auf das Einsatzszenario, so kann man zwei weitere Objekte einführen: das Objekt VLAN und das Objekt Managementsystem.

- Die DTE's können zu **VLAN**'s gruppiert werden. Der Switch implementiert dabei die Zuordnung der DTE's zu den VLAN's.

- Das **Managementsystem** verwaltet den Switch. Es ist für die Einteilung der VLAN's zuständig und überwacht den Status der Ports. Damit verbunden ist die Forderung nach Zugriffskontrollen (*access controls*) für den Switch. Diese sollen aber nicht zentraler Bestandteil dieser Arbeit sein, so daß das Thema erst in Kapitel 6 nochmals aufgegriffen wird.

Die Managementinformation und Anforderung an die Funktionalität der Objekte wird durch Analyse der Anforderungen aus den fünf Funktionsbereichen des Managements hergeleitet.

4.2.1 Switch

Aus der Definition der Management-Einteilung leiten sich die Aufgaben eines Switch im Bereich des Konfigurations-, Leistungs- und Fehlermanagements her. Das Abrechnungsmanagement wird im Rahmen des Leistungsmanagements behandelt. Ein in Frage kommendes Sicherheitsmanagement bezieht sich auf die Einführung von Zugriffskontrollen, die jedoch, wie oben schon erwähnt wurde, nicht zentraler Bestandteil dieser Arbeit sein sollen, sondern gänzlich auf das Managementsystem verlagert werden.

Konfigurationsmanagement

Das Konfigurationsmanagement für einen Switch hat die Aufgabe, den Switch so einzurichten, daß er seinen Dienst im Normalbetrieb möglichst optimal erbringt. Hierfür besitzt der Switch bestimmte Parameter, die im objektorientierten Sinne als Attribute einer Ressource modelliert werden. Das Setzen verschiedener Parameter geschieht im Objektmodell durch definierte Methoden.

Im folgenden sollen die Attribute eines Switch im Bereich des Konfigurationsmanagements vorgestellt werden. In [ID97] werden verschiedenen Attribute spezifiziert. Die Nummern, die in Klammern angegeben werden, verweisen auf die Kapitelnummern in diesem Standard.

- `ID` (8.5.3.7) kennzeichnet den Switch eindeutig. Wird ein oder werden mehrere Switches in einem LAN eingesetzt, kann über dieses Attribut der Switch eindeutig identifiziert werden.
- `max age` (8.5.3.4) bezeichnet das maximale Alter, das eine empfangenen Protokoll-Information (*BPDU*) haben darf, bevor diese verworfen wird.
- `bridge max age` (8.5.3.8) nimmt den Wert von *max age* an, wenn der Switch der *root*-Switch[1] ist oder versucht jener zu werden.

[1] *root* ist ein Begriff aus dem *Spanning-Tree* Algorithmus und bezeichnet den Switch, der bei Verwendung redundanter Switches zunächst zum Einsatz kommt ([ID97]: Standardwerk der IEEE, bietet eine ausführliche, aber nicht unbedingt leicht verständliche Erklärung; dafür eignen sich besser: [Hal92, BRIa, BRIb].

- `available protocols` beinhaltet eine Liste aller Protokolle[2], die geswitcht werden können. Die Liste gibt die Protokolle sowohl textuell als auch mit einer zugehörigen numerischen ID an[3].

Als Methoden eines Switch im Sinne des Konfigurationsmanagements gelten:

- `start()` und `stop()` zum Starten bzw. Beenden der Switchingfunktionalität des PC-basierten Switch. In der *mnm-unix.mib*, die sich auf den Rechnern des MNM-Lehrstuhl im Verzeichnisbaum unter `/proj/sysagent/SnmpMibs/UCD-SNMP` befindet, werden zwei Einträge definiert, die sich an den ISO Standard 10164-2 [ISO93] anlehnen. Zum einen ist dies der *Operational State* und zum anderen der *Administration State*. Beide Objekte sind als *read-only* gekennzeichnet. Der Operational State definiert zwei Einträge: *enabled* und *disabled*. Enabled zeigt an, daß das Objekt (im Falle der mnm-unix.mib der Prozessor) für Benutzer oder Anwendungen bereitsteht. Disabled zeigt das Gegenteil an. Hiermit läßt sich die Betriebsbereitschaft der Hardware anzeigen. Der Administration State definiert drei Zustände: *unlocked*, *locked* und *shuttingdown*. Unlocked bedeutet, daß das Objekt benützt werden kann. Im konkreten Fall des Switch könnte es z. B. dafür verwendet werden, anzuzeigen, daß der Switch-Agent aktiv ist (siehe Kapitel 4.5: agentUp). Shuttingdown zeigt an, daß das Objekt für neue Benutzer nicht zugänglich ist. Nachdem der letzte Auftrag bearbeitet worden ist, wechselt das Objekt in den Zustand *locked*, was bedeutet, daß das Objekt nicht für Managementanweisungen bereitsteht. Im Objektmodell für den Switch ist der Einsatz als Attribute denkbar, die von den Methoden `start()` und `stop()` entsprechend gesetzt werden. *Enabled* und *disabled* zeigen dabei an, daß der Switch aktiv ist oder eben nicht. *Shuttingdown* wird vor dem „Herunterfahren“ des Switch-Agenten gesetzt. Die Zustände *unlocked* und *locked* werden beim Hoch- bzw. Herunterfahren des Switch Agenten gesetzt. *Enabled* und *disabled* werden durch die Methoden `start()` und `stop()` beim Starten bzw. Beenden der Switchingtätigkeit gesetzt.
- `protocols()` gibt die Protokolle an, die geswitcht werden sollen. Mögliche Protokolle, die geswitcht werden können, zeigt obiges Attribut *available protocols* an.

Leistungs- und Fehlermanagement

Mit Hilfe des Leistungs- und Fehlermanagements wird die Überwachung und Fehlerdiagnose des Switch durchgeführt. Besonders in Netzen mit mehreren Switches kann es bedingt durch den *Spanning-Tree* Algorithmus (siehe dazu [ID97, Hal92, BRIa, BRIb]) zu Problemen oder Leistungseinbußen kommen. Dies ist dann der Fall, wenn der *Spanning-Tree* Algorithmus versagt. Ein Grund

[2]Protokolle bezieht sich auf das entsprechende Feld im Ethernetframe, das die schichtspezifisch höhergelegenen Protokolle angeben kann, die das Frame beinhaltet.

[3]Die ID's stellen die offiziellen Ethernet Protocol ID's dar.

hierfür kann zum Beispiel ein Switch sein, der den *Spanning-Tree* Algorithmus nicht versteht, was eher selten der Fall sein dürfte, oder der per Hand an die Bedürfnisse des geswitchten Netzes angepaßt wurde. Die vorgestellten Attribute und Methoden dienen dem Auffinden von Störungen oder Engpässen und sollen helfen, den Switch durch Veränderung seiner Attribute dem gewünschten Einsatzgebiet optimal anzupassen.

Kommen mehrere Switchen in einem Rechnernetz zum Einsatz, so müssen folgende Attribute eines Switch definiert werden, die Eigenschaften des Spanning-Tree Algorithmus beschreiben:

- `designated root` (8.5.3.1) Der eindeutige Verweis auf den Switch, der als *root* akzeptiert wird. Dieser Wert wird in allen BPDU's, die der Switch verschickt, versendet.
- `root path cost` (8.5.3.2) besitzt den Wert, den es kostet den *root*-Switch zu erreichen. Der *root*-Switch selber erhält den Wert null.
- `root port` (8.5.3.3) verweist auf den Port des Switch, der den niedrigsten Wert (*root path cost (8.5.3.2)*) auf dem Weg zum *root*-Switch annimmt.
- `hello time` (8.5.3.5) zeigt das Zeitintervall an, das zwischen dem Verschicken von BPDU's vergeht, wenn der Switch selbst der *root*-Switch ist oder versucht dieser zu werden.
- `forward delay` (8.5.3.6) bezeichnet die Zeit, die ein Port im *listening* oder *learning state* verbringt, bevor er in den *learning* bzw. *forwarding state* übergeht.
- `bridge hello time` (8.5.3.9) zeigt das Zeitintervall an, das zwischen dem Verschicken von BPDU's vergeht, die dem Anzeigen von Topolgieänderungen dienen.
- `bridge forward delay` (8.5.3.10) nimmt den Wert von *forward delay* an, wenn der Switch selbst der *root*-Switch ist oder versucht dieser zu werden.
- `flags` kann den Wert *TOPOLOGY_ CHANGE_ DETECTED* (8.5.3.11), *TOPOLOGY_ CHANGE* (8.5.3.12) oder *NONE* annehmen. Ersterer Wert gibt an, daß eine Topologieänderung auf- oder angezeigt wurde; zweiterer, ob die letzte empfangene *BPDU* eine Topologieänderung angezeigt hat (im Falle eines *root*-Switch), oder ob eine solche während einer definierten Zeit, der sogenannten *Topology Change Time* (8.5.3.13), entdeckt wurde. Ist beides nicht der Fall, so steht das Flag auf *NONE*.

Mit der Methode `debug()` läßt sich der PC-basierte Switch dazu veranlassen, mehr über seine Tätigkeit auszugeben. So kann er bspw. Statusmeldungen über den Empfang und Versand von Ethernet-Frames an einem bestimmten Port mit Quell- und Zieladresse ausgeben oder das Senden einer BPDU anzeigen. Dies dient der Fehlersuche und dessen Behebung. Aus Sicht der Top-Down Modellierung wäre eine Ausgabe dieser Meldungen an einen *event channel* (siehe Kapitel 2.3.1) eine geeignete Methode, da somit die erzeugten Meldungen durch das

Konzept der Entkopplung von Sender und Empfänger durch den *event channel* optimal in eine verteilte Managementlösung eingebunden werden. Durch den Einsatz der Filterungsmöglichkeiten bei der Eventbehandlung können die Debugmeldungen von anderen Statusmeldungen des Switch getrennt werden. Betrachtet man den PC-basierten Switch hingegen aus Sicht des Bottom-Up Ansatzes, so ist eine Ausgabe der Debugmeldungen an den klogd sinnvoll, der die Meldungen aufgrund ihrer Kernelherkunft entgegennimmt. Grund hierfür ist die vorhandene Implementierung des Switchingcodes (siehe hierzu Kapitel 5.1.1), der die Ausgabe von Debugmeldungen an den klogd vorsieht. Bei der späteren Implementierung wird deswegen zweiterer Ansatz gewählt. Eine — auch parallele — Erweiterung um das *event channel*-Konzept ist jedoch durchaus denkbar.

Mittels der Methode `stats()` lassen sich im Hinblick auf das Leistungsmanagement die Anzahl der Pakete pro Protokoll mitprotokollieren (*stats(enable/disable)*), anzeigen (*stats(show)*) und zurücksetzen (*stats(zero)*). Es ist denkbar, diese Daten in einem Abbrechnungsmanagement einzusetzen.

Schließlich weist die Methode `set_bridge_priority()` dem Switch eine neue Identität zu. Dabei laufen mehrere Schritte ab, die in [ID97] (8.8.4) genauer erläutert sind. Zusammengefaßt läßt sich sagen, daß die Methode `set_bridge_priority()` den Switch komplett neu initialisiert.

4.2.2 Port

Ein Bestandteil eines Switch sind seine Ports. Daher wird der Port hier als Managementobjekt eingeführt, das zum Objekt Switch eine Aggregationsbeziehung unterhält.

Konfigurationsmanagement

Im Bereich des Konfigurationsmanagement gilt für den Port, der einen Teil eines Switch darstellt, bzgl. der Betriebsziele das für den Switch gesagte. Ein störungsfreier und optimaler Betrieb eines Ports soll durch das Konfigurationsmanagement sichergestellt werden.

Folgende Attribute eines Ports sind hierfür von Belang:

- `ID` (8.5.5.1) definiert den Port eindeutig in Bezug auf den jeweiligen Switch, dem er angehört.
- `port state` (8.5.5.2) gibt den Betriebsmodus (Disabled, Listening, Learning, Forwarding, Blocking etc.) des Ports an. Eine genaue Beschreibung der einzelnen Betriebsmodi findet sich in [ID97].
- `policy` besitzt zwei Werte: *accept* oder *reject*. *accept* bedeutet, daß alle Protokolle, die nicht explizit ausgeschlossen wurden, akzeptiert werden,

reject hingegen akzeptiert nur die Protokolle, die explizit angegeben wurden.

- `exempt protocols` zeigt die Protokolle an, die von der jeweiligen Policy (*accept* bzw. *reject*) ausgeschlossen sind.

Das Portmanagement bietet im Bereich des Konfigurationsmanagements die Methoden `disable_port()` und `enable_port()` an, die den Port sperren bzw. freigeben. Auch im Hinblick auf das Sicherheits- und Fehlermanagement sind diese beiden Methoden interessant.

Leistungs- und Fehlermanagement

Das für den Switch gesagte gilt auch hier. Insbesondere in Rechnernetzen, in denen sich mehr als ein Switch befindet, kann es nötig sein, bei Störungen, die im Zusammenhang mit dem Spanning-Tree Algorithmus auftreten, die Attribute eines Ports in dieser Hinsicht näher zu betrachten:

- `bandwidth` gibt die Bandbreite des Ports an.
- `path cost` (8.5.5.3) bezeichnet den Beitrag zu den Kosten eines Pfades über diesen Port zum *root*-Switch.
- `designated root` (8.5.5.4) beinhaltet den eindeutigen Bezeichner des *root*-Switch.
- `designated cost` (8.5.5.5) Handelt es sich um einen *designated port*, beinhaltet dieses Attribut den *path cost*, der dem angeschlossenen Netzsegment angeboten wird. Im anderen Fall zeigt dieses Attribut den *path cost* an, der vom *designated port* des LAN-Segments übermittelt wird, an dem der Port angeschlossen ist.
- `designated bridge` (8.5.5.6) beinhaltet die *ID* des Switch, zu dem dieser Port gehört, falls er als *designated port* gekennzeichnet ist, anderenfalls bezeichnet dieses Attribut die *ID*, von der der Switch annimmt, daß er die *designated bridge* des angeschlossenen LAN-Segments ist.
- `designated port` (8.5.5.7) beinhaltet die *Port_ID* einer *designated bridge*, über den Konfigurationsmeldungen (BPDU) ausgesandt werden.
- `flags` kann den Wert *TOPOLOGY_CHANGE_ACK* (8.5.5.8), *CONFIG_PENDING* (8.5.5.9) oder *NONE* annehmen. Hiermit wird angegeben, daß das *Topologie Change Acknowlegement*-Flag gesetzt ist bzw. das Senden einer neuen *BPDU* ansteht. Ist beides nicht der Fall, so steht das Flag auf *NONE*.

Mittels der Methode `set_port_priority()` (8.8.5) kann die Priorität eines Ports verändert werden. Unter „Setzen der Priorität“ versteht man hier die Ausführung mehrere Schritte, die dazu führen, obige Attribute neu zu initialisieren.

Die Methode `set_path_cost()` verändert den Wert des Attributs *path cost* auf den übergebenen Wert.

Sollte es aus technischer Hinsicht, wegen einer Betriebsstörung bspw., nötig sein, so kann ein Port auch gesperrt werden. Diese Methode (`disable_port()`) wird im Konfigurationsmanagement beschrieben.

Sicherheitsmanagement

Unter dem Konfigurationsmanagement wurde die Möglichkeit angesprochen, einen Port gezielt sperren zu können. Unter dem Gesichtspunkt des Sicherheitsmanagements kommt dieser Möglichkeit eine ganz entscheidene Rolle zu. Das angesprochene Einsatzszenario schließt nicht aus, daß sich ein unbekanntes System an einen Port anschließt, der für das unbekannte System nicht zugelassen ist. In diesem Fall läßt sich der Port bspw. abschalten und damit dem System den Zugang zum Netz verweigern.

Das gezielte Auswählen bzw. Zurückweisen bestimmter Protokolle durch den Switch ist mit den Methoden `set_policy()` und `exempt_policy()` möglich. Dabei sind die beiden Methoden so definiert, daß sie ihre zugehörigen Attribute entsprechend setzen können.

Auch die Zuordnung von VLAN's zu einem Port läßt sich im Zusammenhang mit dem Sicherheitsmanagement bringen. Dann nämlich, wenn wie im Einsatzszenario beschrieben die Ports in öffentliche und nicht öffentliche Gruppen eingeteilt werden. Man sieht also, daß man die Methoden eines Ports nicht unbedingt eindeutig einem Managementbereich zuordnen kann. Je nach Sichtweise der zu tätigenden Aktion fällt es in unterschiedliche Bereiche.

Abrechnungsmanagement

Denkbar wäre ein Attribut `frames`, das die Anzahl der empfangenen und gesendeten Frames, aufgeschlüsselt nach ihren Protokollen, auf diesem Port zählt. Damit lassen sich zwar bzgl. des Abrechnungsmanagements nur ungenaue Angaben über die Nutzung bestimmter Dienste machen, jedoch wird eine genaue Statistik, aufgeschlüsselt nach Art und Menge der Protokolle, sämtlicher weitergeleiteten Frames angelegt. Mit Hilfe dieser Daten können dann gezielt Protokolle ausgewählt bzw. zurückgewiesen werden. Als Erweiterung wäre eine Größe *kbyte* interessant, die die Gesamtgröße aller empfangenen und weitergeleiteten Frames beinhaltet.

4.2.3 VLAN

Ein Switch realisiert die Gruppierung von VLAN's. Die Managebarkeit eines VLAN's beschränkt sich auf das Konfigurationsmanagement. Attribute eines VLAN's sind:

- `ID` kennzeichnet das VLAN eindeutig.
- `name` ordnet dem VLAN einen Bezeichner zu.

Da ein VLAN auch leer sein kann, ihm also keine Objekte zugeordnet sind, bietet die Methode `create()` eine Möglichkeit, ein neuen VLAN's zu erstellen und die Methode `remove()` das Entfernen eines solchen. Durch das Angeben geeigneter Konstruktoraufrufe ist es möglich, daß die Methode `create()` eine neue Instanz der Klasse **VLAN** erzeugt. Weitere Möglichkeiten, die Funktionsweise dieser Methoden zu implementieren werden in Kapitel 5 diskutiert. Darüberhinaus bietet das Objekt VLAN zwei weitere Methoden an:

- `addmember(type,ID)` ordnet dem VLAN ein neues Objekt zu. Die Art des Objekts, also eine DTE oder ein Port, wird mit dem Parameter `type` übergeben.
- `remmember(type,ID)` entfernt obiges Objekt wieder.

4.3 Managementobjektmodell

In diesem Kapitel soll das Konzept eines Objektmodells erläutert werden. Es stellt die Modellierungsphase dar. Die Objekte, die in der Analysephase zusammengetragen wurden, werden nun in Beziehung zueinander gesetzt. Die Sicht auf dieses Modell wird von den Anforderungen an das Management eines PC-basierten Switch geprägt sein. Im Gegensatz dazu wird im Kapitel 4.4 ein Objektmodell aus Sicht der Implementation einer Managementschnittstelle für einen PC-basierten Switch entwickelt.

4.3.1 Physische Sicht

Abbildung 4.1[4] zeigt einen Ansatz für ein Objektmodell aus Sicht des Managements. Die drei Objekte aus der Analysephase, *Switch*, *Port*, und *DTE*, bilden dabei die physische Sicht auf das Objektmodell. Schließt sich ein neues System an einen Port eines Switch an, wird eine neue Instanz der Klasse *DTE* erzeugt, die eine Unterklasse von *Objekt* ist. Einziges Attribut der Klasse *DTE* ist dessen ID, also die MAC-Adresse des angeschlossenen Systems.

Angeschlossen wird die DTE an einen Port. Die Beziehung *connects* verdeutlicht, daß an einem Port mehrere DTE's hängen können. Sie stellt also die Tabelle für die Zuordnung von Port zu MAC-Adresse dar. Die Klasse *Port* erbt von der Oberklasse *Objekt* ihre ID. Weiterhin erhält sie die in der Analysephase spezifizierten Attribute für das Konfigurations-, Leistungs- und Fehlermanagement. Die Methoden `disable_port()` und `enable_port()` sind für das An- und Abschalten des Ports vorgesehen. `set_port_priority()` setzt die Priorität des Ports neu, wie in der Analysephase beschrieben wurde.

[4] Leicht von der OMT abweichende Notifikation, wegen Verwendung von StP.

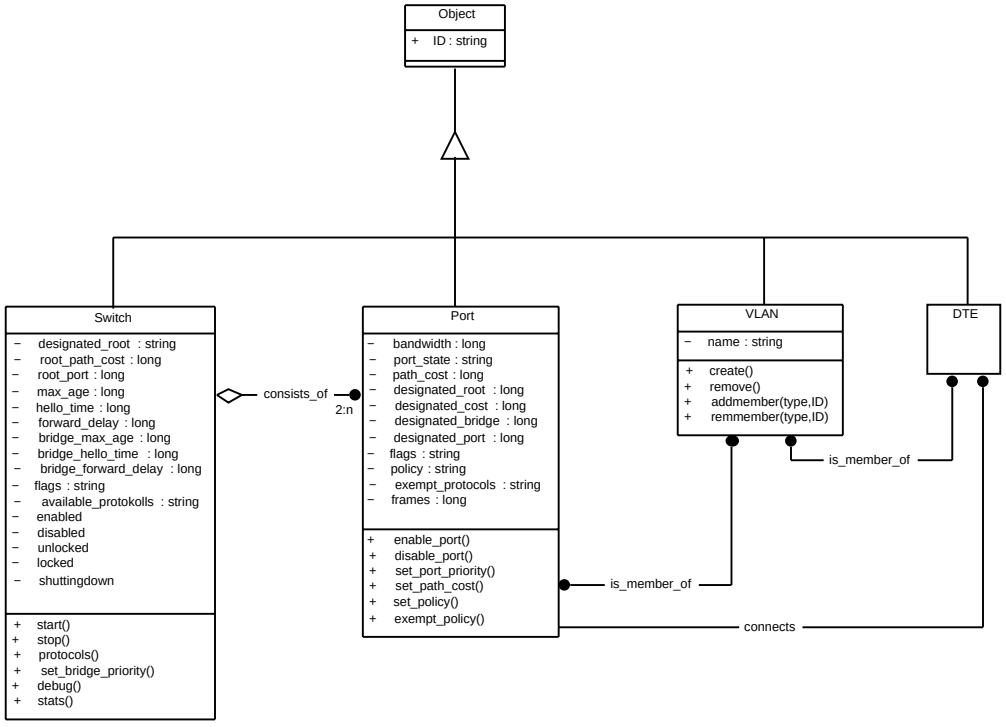


Abbildung 4.1: Objektmodell aus Management-Sicht

Die Klasse Port ist eine Aggregation der Klasse *Switch*. Die Beziehung *consists_of* ordnet dem Objekt *Switch* zwei oder mehr Instanzen der Klasse *Port* zu. Das Objekt *Switch* besitzt die Attribute aus der Analysephase und erbt das Attribut *ID* der Oberklasse *Objekt*. Hinzu kommen die Methoden `start()` und `stop()` zum Starten bzw. Stoppen des Switch, was dem Ein- bzw. Ausschalten eines Switch gleichkommt. Die Methoden für das Leistungs- und Fehlermanagement aus der Analysephase sind ein weiterer Bestandteil des Objekts *Switch*.

4.3.2 Logische Sicht

Zur logischen Sicht des Objektmodells gehören die Objekte *DTE*, *Port* und *VLAN*. Beim Anschluß einer neuen DTE an einen Switch wird durch die Methode `addmember(type,ID)` des Objekts *VLAN* ein neuer Tabelleneintrag für die Zuordnung von DTE zu VLAN erzeugt. `type` steht hier für die Art des übergebenen Objekts. Handelt es sich wie hier um eine DTE, so lautet der zu übergebene Parameter `(DTE,<MAC-Adresse>)`, wobei `<MAC-Adresse>` für eine 6 Byte große Zahl steht, deren Bytes durch Doppelpunkte voneinander getrennt werden. Die Beziehung *is_member_of* soll die Zuordnung verdeutlichen. Durch die Methode `remmember(type,ID)` kann das Objekt *DTE* wieder aus der Beziehung gelöst werden.

Analog dazu wird eine Beziehung zwischen den Objekten *Port* und *VLAN* erzeugt. Jedem Objekt *Port* kann eine oder mehr Instanzen der Klasse *VLAN* zu-

geordnet werden. Die Beziehung *is_ member_ of* verdeutlicht diese Zuordnung. Die Methoden `create()` und `remove()` erzeugen eine neue Instanz der Klasse *VLAN*, dessen eindeutiger Bezeichner das von der Oberklasse *Objekt* geerbte Attribut *ID* ist.

4.4 Implementierungsmodell

Das im vorigen Kapitel entworfenen Objektmodell ist aus der Bottom-Up Modellierung entstanden. Von Bedeutung für dessen Erstellung waren die Eigenschaften der Ressourcen eines PC-basierten Switch, die dem Konfigurationsmanagement zuzuordnen sind. Hinzu kamen Attribute und Methoden aus dem Leistungs- und Fehlermanagement. In diesem Kapitel soll das erstellte Objektmodell im Hinblick auf die Implementierung spezialisiert werden. Für die Umsetzung des Objektmodells müssen architektur- und implementierungsspezifische Aspekte beachtet werden.

Abbildung 4.2 zeigt das der Implementierung zugrundeliegende Objektmodell. Im folgenden sollen kurz die Unterschiede zum im vorigen Kapitel erstellten Managementobjektmodell erläutert und gerechtfertigt werden.

Der Switch-Agent muß eine Schnittstelle für die Dienstanbindung an den ORB zur Verfügung stellen. Dazu wird das Objekt Switch aus dem Managementmodell (4.1) in ein IDL-Interface (*Switch*) und dessen Implementierung (*SwitchStationaryAgent*) umgesetzt. Das IDL-Interface bietet alle Methoden und Attribute an, die das ursprüngliche Objekt auch angeboten hat. Ein zusätzliches Attribut `port_array`, das aus einem Array von Objekten der Klasse *Port* besteht, implementiert die Beziehung *consists_ of* zur Klasse *Port*. Das Attribut `dte_list` nimmt in einem Datentyp *Vector* der Programmiersprache Java eine beliebige Anzahl von Objekten der Klasse DTE auf. Der Grund für diese beiden Attribute liegt darin, daß die Methoden `addmember()` und `remmember()` aus der Klasse *VLAN* in die Klassen *Port* und *DTE* verschoben wurden. Dies hat u. a. implementierungstechnische Gründe. Einem Port oder einer DTE wird immer genau eine VLAN ID zugeordnet. Dahingegen können einer VLAN ID mehrere Ports bzw. DTE's zugeordnet sein. Geht man objektorientiert vor, so läßt sich die Beziehung *is_ member_ of* dadurch realisieren, daß das Objekt Port (oder DTE) ein Attribut VLAN ID besitzt, das durch eine Methode (`set_vlan_id`) der Klasse Port (oder DTE) verändert werden kann.

Das Interface *Switch* stellt die einzige Schnittstelle des Agenten dar. Der Grund hierfür liegt darin, daß es so einfacher ist, den übrigen Klassen des Implementierungsmodells neue Attribute und Methoden hinzuzufügen, ohne unbedingt auch die Schnittstellenspezifikation zu ändern. Darüberhinaus umgeht man so das Problem der Mehrfachvererbung, die Java nicht anbietet. Um die Methoden der übrigen Klassen auch dem Manager zugänglich zu machen, muß das Interface Switch diese anbieten. So realisiert das Interface die Methoden `set_vlan_id()` der Klassen Port und DTE dadurch, daß es diese aus den Methoden `set_port_vlan_id()` und `set_mac_vlan_id()` aufruft. Auf die gleiche

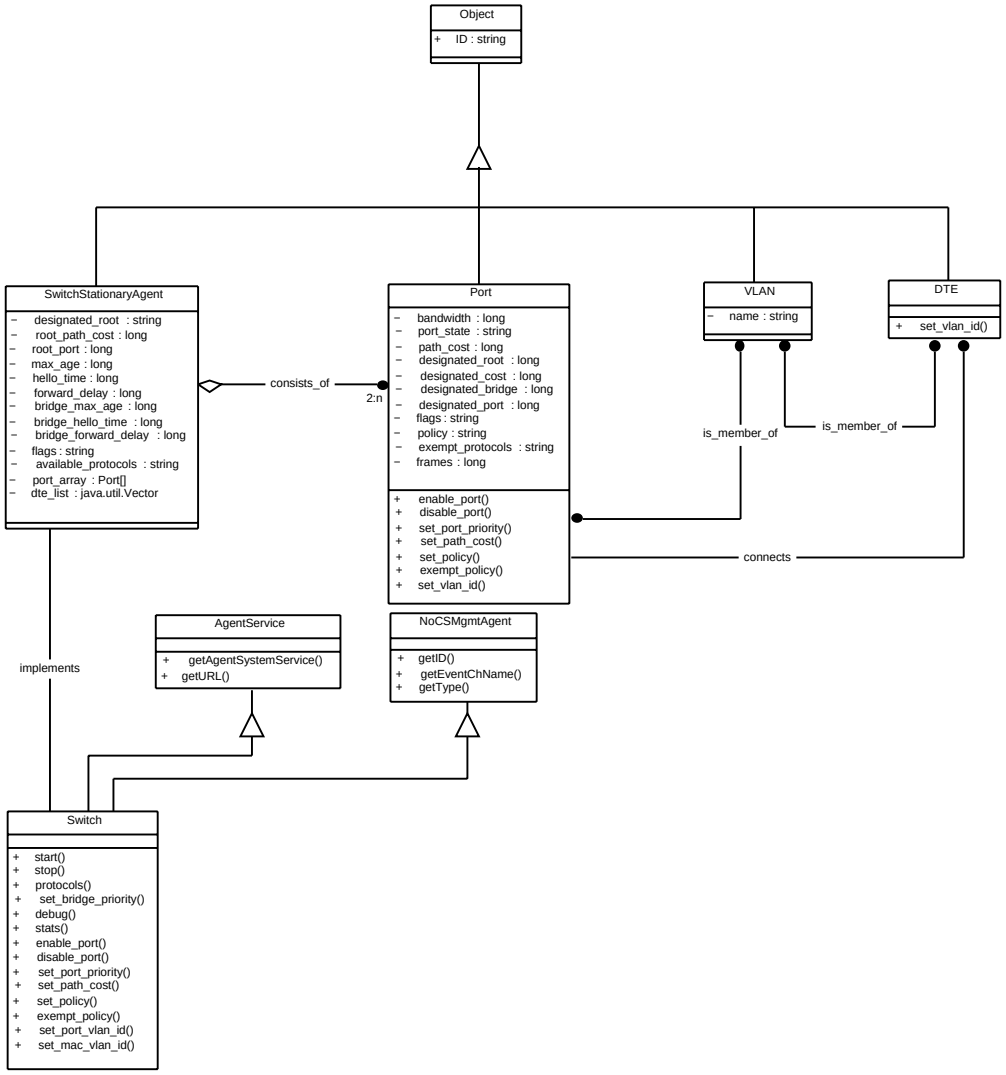


Abbildung 4.2: Objektmodell aus Implementierungssicht

Art und Weise bietet das Interface alle weiteren Methoden an, die in Kapitel 4.3 spezifiziert worden sind.

Die IDL-Schnittstelle für den Switch hat folgendes Aussehen:

```
#ifndef _Switch_idl_
#define _Switch_idl_

#include "/proj/fagent/masa_0.2/src/idl/AgentService.idl"
#include "/proj/fagent/NoCScontrol/src/idl/NoCSMgmtAgent.idl"

interface Switch;

interface Switch : agent::AgentService : mo::NoCSMgmtAgent
```

```
{
    void start();
    void stop();
    void protocols();
    void set_bridge_priority();
    void debug(in string para);
    void stats(in string para);
    void enable_port(in long number);
    void disable_port(in long number);
    void set_port_priority(in long number);
    void set_path_cost();
    void set_policy(in string para);
    void exempt_policy(in string para);
    void set_port_vlan_id(in long port,in long vlan_id);
    void set_mac_vlan_id(in string mac,in long vlan_id);
};
#endif
```

Dabei fällt auf, daß das Switch-Interface von den Interfaces *AgentService* und *NoCSMgmtAgent* die Module *agent* und *mo* erbt. Die beiden Module werden in [Kem98] und [Rad98] beschrieben. Ebenso die Notwendigkeit des hier geschilderten Agenten, von diesen zu erben. Die Vererbung ist im Implementierungsmodell (Abbildung 4.4) durch eine Generalisierung dargestellt.

Der Aufbau des Moduls *AgentService* sieht dabei folgendermaßen aus:

```
module agent {
  interface AgentService {
    agentSystem::AgentSystemService getAgentSystemService();
    string getURL();
  };
};
```

Die Methode `getAgentSystemService()` hat als Rückgabewert die CORBA-Objektreferenz auf das Agentensystem, auf dem der Agent abläuft. Die Methode `getURL()` liefert die URL zur Homepage des Agenten [Kem98].

Das Modul *mo* ist so aufgebaut:

```
module mo {
  interface NOCSMgmtAgent {
    string getID();
    string getEventChName();
    string getType();
  };
};
```

Dabei liefert die Methode `getID()` den *compound name* des Agenten, die Methode `getEventChName()` den Namen des vom Agenten kreierten Event Channels

und die Methode `getType()` den Typ des Agenten (siehe Kapitel 4.5). Näheres dazu findet sich in [Rad98].

4.5 Das Event Channel-Modell

Der allgemeine Aufbau eines *Structured Events* wurde bereits in Kapitel 2.3.2 beschrieben. Im weiteren soll geschildert werden, wie sich der Event Service in das vorhandenen Managementmodell einfügt, und wie ein Structured Event für das bestehende Einsatzszenario aussehen muß.

Der Switch-Agent muß einen eigenen Event Channel kreieren, an den sich die Managementanwendung und der Agent binden. Der Switch-Agent benutzt den Event Channel, um Zustandsänderungen des Switch in Form von Notifikationen an den Manager zu senden. Die gesendeten Notifikationen sind dabei jeweils im Rahmen eines *structured events* verpackt, wie er in Abbildung 2.14 dargestellt ist.

Für die Managementsoftware von Bedeutung ist die Information über neu angeschlossene Systeme an den Switch. Die Zustandsänderung äußert sich somit durch das Hinzufügen oder Entfernen einer DTE an einen Port des Switch. Problematisch ist dabei nur das Entfernen einer DTE, da der Switch davon nicht explizit benachrichtigt wird. Stattdessen sorgt ein *Timeout*-Parameter dafür, daß DTE's, die eine gewisse Zeit keine Frames verschickt haben, aus der Switchingtabelle entfernt werden. Für das Modell des Switch-Agent bedeutet dies, daß nur das Hinzufügen eines neuen Systems angezeigt wird. Diese Vorgehensweise ist nicht unbedingt eine Einschränkung, da aus Sicht des zugrundeliegenden Einsatzszenarios die Managementsoftware für die VLAN-Einteilung *neu* angeschlossener Systeme zuständig ist. Unterliegt ein angeschlossenes System einem Timeout, so stuft der Switch dieses System als neu ein, sobald er wieder ein Frame von diesem erhält.

Der Switch-Agent sendet beim Hoch- und Herunterfahren ein festgelegtes Datenpaket in Form eines *structured events* an den sogenannten *wellknown event channel* des Managementsystems, der im folgenden als *AgentInitChannel* bezeichnet wird. Dadurch erfährt dieses, daß ein neuer Agent gestartet oder dieser beendet wurde.

Der Aufbau der vom Switch Agent gesendeten Structured Events in den *wellknown event channel* wird in Abbildung 4.3 dargestellt. Bezüglich der Notation wurden unter Berücksichtigung von [Kem98], [Rad98] und [Dem98] folgende Festlegungen vereinbart:

1. Der Event Header besteht lediglich aus dem Fixed Header. Es gibt keinen Variable Header. Die Feldbelegungen sehen wie folgt aus:

   domain_type wird mit dem Wert `Systemmanagement` belegt.

event_type bezeichnet die Aktion: `agentUp`, falls der Agent hochgefahren wird und `agentDown`, falls er heruntergefahren wird.

event_name ist nicht von Bedeutung und wird deswegen freigelassen.

2. Der Event Body enthält folgende Felder:

   agent_type zeigt den Typ des Agenten an. Im Falle des PC-basierten Switch ist der Wert des Feldes `PCSwitchAgent`.

   agent_name gibt den *compound name* des Agenten in Form des erweiterten Domänenbezeichners ([Rad98]) als String an. Da sich an den AgentInitChannel mehrere Agenten anbinden, soll dieses Feld helfen, die Quelle des erhaltenen Events festzustellen. Der Switch-Agent übergibt den Wert `mnm.domain/PCSwitch.motype/pcswitch_1.agent`.

   event_channel_name bezeichnet den *compound name* des Event Channels, an den sich der Agent gebunden hat. Dieser kann zuvor vom Agenten kreiert worden sein. Die Wertbelegung für den Switch-Agent ist `PCSwitch`.

   Zusätzlich zu diesen drei Feldern können im Filterable Body noch zusätzliche Felder definiert werden. Zu beachten ist lediglich, daß die Belegung des Wertefeldes auf die Datentypen `string`, `int` und `boolean` eingeschränkt ist.

   Der Remaining Body wird nicht benötigt und bleibt daher leer.

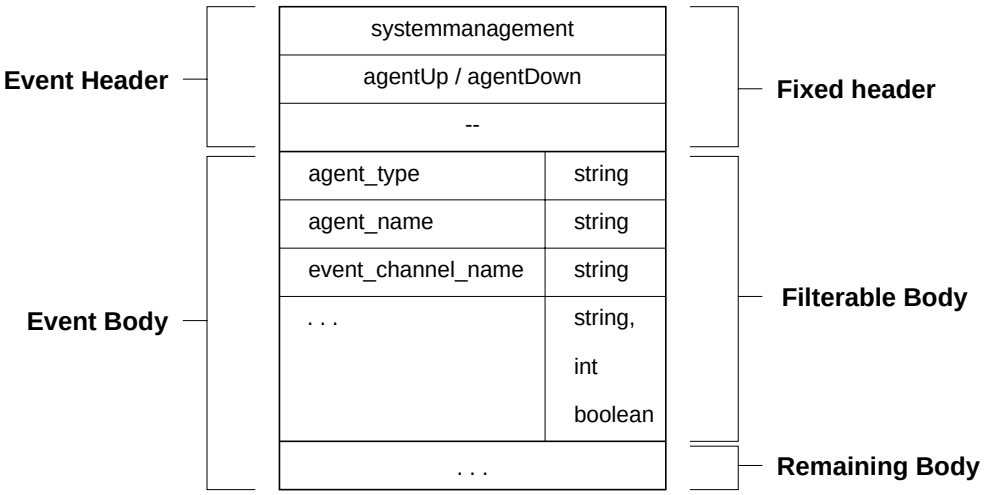


Abbildung 4.3: Event-Struktur für das An- und Abmelden des Agenten

Kapitel 5

Realisierung der Managementschnittstelle

Die Schnittstellenimplementierung für das Management eines PC-basierten Switch besteht aus zwei Teilen. Die Realisierung der Switch-Funktionalität übernimmt der Kernel des Betriebssystems Linux. Im Gegensatz zu kommerziellen Betriebssystemen stellt Linux den Quellcode des Kernels frei[1] zur Verfügung. Dies war ein wichtiger Grund für die Entscheidung zugunsten von Linux. Konfiguriert wird der Bridging-Code durch ein kleines C-Programm, *brcfg.c*, daß auf Kommandozeilenebene vom Superuser ausgeführt werden kann. Dieses ist der erste Teil. Die Schnittstelle zwischen dem in Java geschriebenen *Switch Management Agent*, der mit seinem Manager mittels CORBA Objekte austauscht, und dem C-Programm bildet den zweiten Teil. Abbildung 5.1 soll die Zusammenhänge graphisch erläutern helfen.

5.1 Schnittstelle zur Basissoftware

Der Begriff Basissoftware steht hier für den zentralen Switch-Code sowie einem Programm, das diesen Code steuert. Dieses Programm ist wegen seiner Systemnähe in der Programmiersprache C geschrieben. Mittels einfacher Eingabeparameter läßt sich so der Switch konfigurieren. Der Switch selbst besteht aus einem Stück Code im Linux-Kernel, der im Rahmen dieser Arbeit um VLAN Funktionalität erweitert wird.

Als Basisdistribution dient die Debian 2.0 [Deb], eine Linuxdistribution, die auf der *glibc* aufbaut. Grund für die zum damaligen Zeitpunkt noch als *unstable*[2] geltende Distribution ist der Einsatz eines Entwicklerkernels, der von neueren

[1] Frei bedeutet hier, daß er für jederman zugänglich ist (im Internet zum Beispiel); das heißt nicht, daß er auch umsonst zu haben ist.

[2] Projektname *hamm*

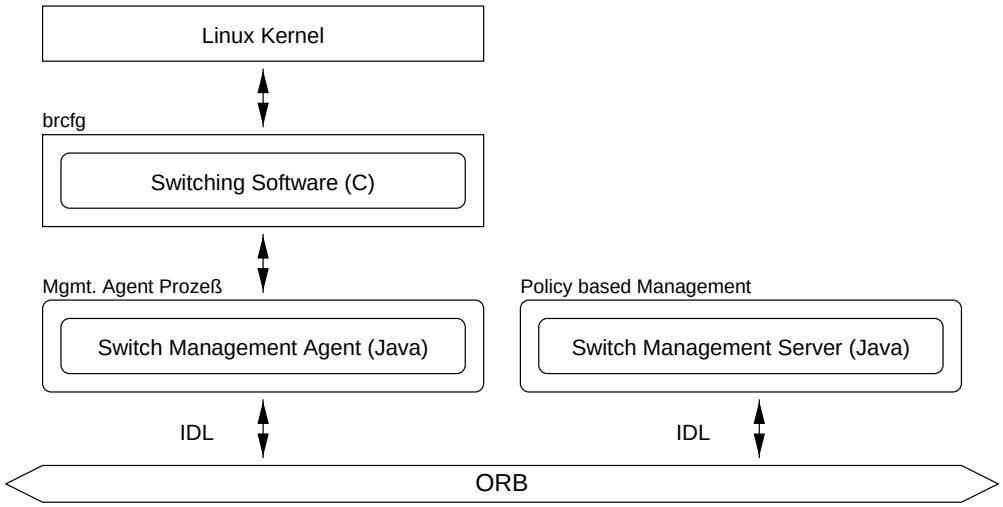


Abbildung 5.1: Managementschnittstellen

Softwarepaketen abhängig ist, die in der als stabil bezeichneten Distributionsversion[3] nicht vorhanden waren.

5.1.1 Erweiterung des Kernels

Grundlage der Erweiterung ist die Kernelversion 2.1.59. Die ungerade Zahl zwischen den beiden Punkten deutet dabei auf einen als nicht stabil geltenden Kernel hin. Da aber der Bridging-Code in den stabilen Kernelversionen (2.0.x) Bugs aufweist (bspw. ließ sich der Switch selbst nicht mehr von außen erreichen), kommt nur ein Entwicklerkernel in Frage. Der Kernel mit der höchsten Versionsnummer war zum Zeitpunkt des Arbeitsbeginns 2.1.80. Leider stürzt bei diesem Kernel das ganze System ab, sobald der Bridging-Code gestartet wird. So war es nötig, schrittweise downzugraden, bis sich die Version 2.1.59 als geeigneter Ausgangskernel erwies. Tests mit dem zum Ende dieser Arbeit aktuellen Entwicklerkernel 2.1.113 haben ergeben, daß auch hier der als *experimental* gekennzeichnete Bridging-Code nur sehr instabil läuft und nach kurzer Zeit zu heftigen Betriebssystemabstürzen führt. Daher muß auf eine Portierung der Kernelerweiterung um VLAN-Funktionalität auf einen Kernel mit höherer Versionsnummer vorerst verzichtet werden.

Den Bridging-Code findet man im Verzeichnisbaum (siehe Abbildung 5.2) unter `/usr/src/linux/net/bridge`. Neben dem Makefile befinden sich dort die Quelltextdateien `br.c` für den Bridging-Code und `br_tree.c`, der die Forwardingtabellen implementiert. Die Datei `sysctl_net_bridge.c` soll später mal die Schnittstelle zum `/proc`-Dateisystem von Linux bilden. Gegenwärtig existiert diese Schnittstelle jedoch noch nicht.

[3] Projektname *bo*

Die Erweiterung des Codes um VLAN-Funktionalität teilt sich in zwei Schritte auf. Zum einen müssen VLAN's auf Port-Ebene (Schicht 1) implementiert werden zum anderen VLAN's auf MAC-Adressen-Ebene (Schicht 2). Ausgangspunkt für die Erweiterung ist die Datei `br.h`. Dabei gilt es zunächst festzustellen, welche Funktionen des Codes die Bridge ausführt, wenn sie ein Datenpaket erhält.

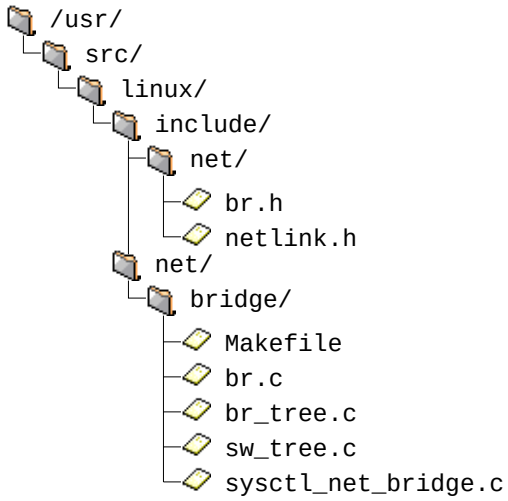


Abbildung 5.2: Verzeichnisbaum für den Bridgingcode

Vorhandener Code

Der Switch hat zwei Fälle zu unterscheiden: Ein Ethernet-Frame kann vom eigenen TCP/IP-Stack oder von einem der Ports kommen. Im ersteren Fall ruft der Switch die Funktion br_tx_frame — das *tx* im Funktionsnamen deutet schon an, daß hier ein IP-Paket gesendet werden soll — im zweiteren die Funktion br_receive_frame auf. In beiden Fällen wird dem Switch die Struktur sk_buff übergeben, die unter `/usr/src/linux/include/linux/skbuff.h` deklariert wird. Unter anderem enthält diese Struktur die Quell- und Ziel-Adresse des empfangenen Schicht-2-Frames.

Kommt der Frame vom *host stack*, so wird, nachdem das *Loopback Device* ausgeschlossen wurde, die Funktion br_forward(struct sk_buff *skb, int port) aufgerufen, der die Struktur sk_buff und der Ursprungsport als Integerwert übergeben werden. Diese Funktion leitet den Frame entweder an eine bekannte Adresse an einem bestimmten Port weiter oder schickt einen Broadcast an alle Ports, wenn die Zieladresse des Frames nicht bekannt ist.

Liegt der Frame hingegen an einem Port an, so wird zunächst die Funktion static int br_learn(struct sk_buff *skb, int port) aufgerufen. br_learn ruft die Funktion br_avl_find_addr aus `br_tree.c` auf, die u. a. den Port und die MAC-Adresse

des empfangenen Frames in eine Tabelle einträgt[4] oder den vorhandenen Eintrag mit einem neueren Datum versieht. Das Datum dient als *Timestamp*, damit nach einer gewissen Zeit der Inaktivität der Eintrag wieder gelöscht werden kann. Ist die gewünschte Information in der Tabelle abgepeichert worden, wird das Ziel des Frames festgestellt. Ist der Frame für den Switch selber, so wird er an den *host stack* weitergeleitet, ansonsten die Funktion br_forward aufgerufen, die für die Auslieferung des Frames verantwortlich ist.

Je nachdem, ob die Zieladresse bekannt ist, also in der Switch Tabelle vorhanden ist, oder nicht, ruft br_forward die Funktion extern int dev_queue_xmit(struct sk_buff *skb) auf, die den Frame an den gewünschten Port ausliefert oder static int br_flood(struct sk_buff *skb, int port), die den Frame an alle Ports außer dem Quellport weiterleitet.

Erweiterter Code

Ansatzpunkt für die Implementation der VLAN Eigenschaften auf Port-Ebene ist zunächst einmal die Struktur Port_data, die alle für einen Port relevanten Daten enthält und um einen Eintrag für VLAN's erweitert werden muß. Port_data ist in der Headerdatei br.h deklariert, die unter /usr/src/linux/include/net zu finden ist. Ein Integerwert (unsigned int vlan_id) soll das VLAN eindeutig identifizieren. Zusätzlich wird noch eine Konstante DEFAULT_VLAN_ID definiert, die den Hexadezimalwert 0x01 erhält. DEFAULT_VLAN_ID steht wie der Name schon sagt für den Defaultwert des VLAN's, den der Port annimmt, wenn ihm kein anderer Wert zugewiesen wird.

Gesetzt wird der Defaultwert in der Funktion br_init_port in br.c, die den Port initialisiert. Dazu wurde eine Funktion set_port_vlan_id geschrieben, die auch später verwendet werden kann, um einen Port einem VLAN zuzuweisen.

```
771 static void set_port_vlan_id(int port_no, unsigned int vlan_id)
772 {
773         port_info[port_no].vlan_id = vlan_id;
774 }
```

Für den Fall, daß der Frame vom *host stack* empfangen wird und dieser noch keinem VLAN zugeordnet ist, wird diesem in der Funktion br_tx_frame explizit ein VLAN zugeordnet. Sinnvollerweise ist der Switch selbst Bestandteil aller VLAN's. Das führt zu der Frage, wie der Switch durch die Zuordnung nur einer VLAN ID mehreren VLAN's angehören kann. Offensichtlich besteht zwischen dem Objekt Port bzw. MAC-Adresse und dem Objekt VLAN eine n:n Beziehung. Ein Port (oder eine MAC-Adresse) kann einem oder mehreren VLAN's zugeordnet sein. Umgekehrt enthält ein VLAN einen oder mehrere Ports (oder MAC-Adressen). Um den rechnerischen Aufwand bei der Prüfung einer verketteten Liste von Objekten zu umgehen, wird an dieser Stelle eine binäre UND-Verknüpfung von Integerwerten angewandt. Somit ergeben sich 32

[4] Um einen effizienten Zugriff auf diese Tabelle zu bekommen, ist sie als AVL-Baum realisiert worden.

unterschiedliche VLAN's. Liegt ein Port bspw. in VLAN 1 und VLAN 2, so bekommt er die VLAN ID 3. Die Funktion check_vlan_id verdeutlicht dies:

```
1712 static int check_vlan_id(struct fdb *s, struct fdb *f)
1713 {

[...]

1736         if ((s->vlan_id & f->vlan_id) == 0)
1737                 return(FALSE);
1738         return (TRUE);
1729 }
```

Die Struktur fdb stellt den Datentyp dar, der als Knoten im AVL-Baum gespeichert ist. Verglichen werden hier die VLAN ID's von Quell- und Zieladresse. Bei Übereinstimmung liefert der Code *true* zurück, ansonsten *false*.

Analog dazu werden in br_forward die Portzuordnungen verglichen:

```
1650 /*
1651 *      Sending
1652 *      only if: source-port != destination-port
1653 *               port.state == Forwarding
1654 *               source.address.vlan_id == dest.address.vlan_id
1655 *               source.port.vlan_id == dest.port.vlan_id
1656 */
1657 if (f->port!=port && port_info[f->port].state == Forwarding
1658   &&
1659     check_vlan_id(s, f)
1660   &&
1661     ((port_info[f->port].vlan_id & port_info[port].vlan_id) != 0) )
1662 {

[...]

1701 }
```

Bei Erfolg wird der Frame an den entsprechenden Port geforwardet, bei Mißerfolg durch die Funktion br_dev_drop vernichtet.

Das Setzen der VLAN ID's ist ungleich aufwendiger. Die Funktion br_ioctl interagiert mittels I/O Aufrufen zwischen dem Konfigurationsprogramm und dem Kernel. Sie erwartet als Eingabe einen Parameter vom Typ Integer. Die Aufschlüsselung zwischen Integerwert und dem zugehörigen *Define* ist in br.h beschrieben. Für die VLAN-Zuordnung sind zwei neue *Defines* hinzugekommen: BRCMD_VLAN_CONFIG mit Wert 16 und BRCMD_MAC_VLAN mit Wert 17. Erhält der Kernel nun einen ioctl, dann ruft er die entsprechende Routine auf: set_port_vlan_id, um die Port-VLAN-Zuordnung zu realisieren und set_mac_vlan_id für die MAC-Adressen-VLAN Zuordnung. Als Argumente werden jeweils Strukturen vom Typ br_cf übergeben, die in br.h folgendermaßen deklariert wird:

```
212 struct br_cf {
213         unsigned int cmd;
214         unsigned int arg1;
```

```
215         unsigned int arg2;
216         unsigned char ula[6];
217 };
```

cmd ist das schon angesprochene Kommando, arg1 und arg2 können Zahlenwerte aufnehmen, bspw. die Portnummer. Interessant ist lediglich das 6 Byte große Array. Dieses wird nur für die Aufnahme der MAC-Adresse in hexadezimaler Form benötigt, der gesamte Kernelcode arbeitet mit diesem Array, so daß hier nicht davon abgewichen werden soll. Die Funktion set_port_vlan_id wurde weiter oben schon beschrieben.

Die Implementierung der Zuordnung von MAC-Adresse zu einem oder mehreren VLAN's wird im folgenden beschrieben. Zunächst wird die switchinterne Tabelle, die die Information über die Zuordnung von Port- zu MAC-Adressen beinhaltet, von der Tabelle für die MAC-Adressen zu VLAN Zuordnung getrennt. Der Einfachheit halber wurde dazu der vorhandene Code br_tree.c kopiert und in sw_tree.c umbenannt. Im Code selber wurden sämtliche Funktionen umbenannt. Aus addr_cmp wurde so sw_addr_cmp u.s.w. Weiterhin wurde die Struktur fdb aus br.h nach swdb kopiert und dessen Zeiger mit dem Präfix sw_ versehen. Damit entstand eine neue Tabelle für die Zuordnung von MAC-Adressen zu VLAN's in Form eines AVL-Baums.

Das Setzen der VLAN ID geschieht wieder durch einen Funktionsaufruf in br_ioctl.

```
1904 case BRCMD_MAC_VLAN:
1905          set_mac_vlan_id(bcf.arg1, &bcf.ula[0]);
1906          break;
```

Der Funktion set_mac_vlan_id werden als Parameter die VLAN ID und die MAC-Adresse übergeben. Die Funktion selber sieht folgendermaßen aus:

```
776 static int set_mac_vlan_id(unsigned int vlan_id, unsigned char *ula)
777 {
778     struct swdb *s, *m = NULL;
779
780     s = (struct swdb *)kmalloc(sizeof(struct swdb), GFP_ATOMIC);
781     if (!s) {
782             printk(KERN_DEBUG "br_learn: unable to malloc swdb\n");
783                return(-1);
784     }
785
786     m = sw_avl_find_addr(ula);
787     if (m) {
788             s->port = m->port;
789     }
790     else { /* unknown MAC-address */
791             printk(KERN_DEBUG "set_mac_vlan_id: unknown mac address\n");
792             return(-2);
793     }
794
795     memcpy(s->ula, ula, 6); /* specified mac address */
796     s->timer = CURRENT_TIME;
797     s->flags = FDB_ENT_VALID;
```

```
798
799     s->vlan_id = vlan_id;   /* specified vlan_id */
800
801     if (sw_avl_insert(s) == 0) {    /* update */
802             kfree(s);
803             return(0);
804     }
805     else {                          /* unknown MAC-address */
806             printk(KERN_DEBUG "set_mac_vlan_id: unknown mac address\n");
807             kfree(s);
808             return(-2);
809     }
810 }
```

Die Variablen s und m sind Zeiger auf eine Variable der Struktur swbd. Zunächst wird Speicherplatz für die Struktur allokiert und mit s eine Referenz darauf angelegt. Sollte der Speicher korrekt allokiert worden sein, prüft der Aufruf von sw_avl_find_addr(ula), ob die übergebene MAC-Adresse in der Switchtabelle schon bekannt ist und übergibt den zugehörigen Port an die durch den Zeiger s referenzierte Struktur. Danach werden die Felder ula, timer, flags und vlan_id gefüllt. Die Variable ula beinhaltet die übergebene MAC-Adresse, vlan_id die Zugehörigkeit zu den gewünschten VLAN's. Durch das Setzen von timer auf die aktuelle Zeit verlängert sich der *timestap* des aktuellen Eintrages. flags zeigt an, daß es sich um einen gültigen Eintrag handelt. Als nächstes wird durch Aufruf der Funktion sw_avl_insert(s) der betreffende Eintrag im AVL-Baum durch die neuen Werte überschrieben und danach der Speicherplatz wieder freigegeben.

Die Initialisierung einer neuen MAC-Adresse mit einem Default-VLAN Wert geschieht in der Funktion br_learn nach einem ähnlichem Schema:

```
1472   m = sw_avl_find_addr(skb->mac.ethernet->h_source);
1473   if (m) {
1474           s->vlan_id = m->vlan_id;
1475   }
1476   else { /* set default vlan_id */
1477           s->vlan_id = DEFAULT_VLAN_ID;
1478   }
```

Hierbei wird zunächst überprüft, ob zu der neu hinzugekommenen MAC-Adresse schon eine VLAN-Zuordnung besteht. Wenn dem so ist, wird diese Zuordnung beibehalten, ansonsten die neue DTE mit dem Defaultwert versehen. Grund für diese Abfrage ist, daß die Funktion br_learn sowohl dann aufgerufen wird, wenn der Status des betreffenden Ports auf Learning steht, als auch dann, wenn Frames geforwarded werden sollen. Bei jedem Frame, den der Switch weiterforwarded, wird der *timestamp* der betreffenden DTE an dieser Stelle erneuert. Übergeben wird der Funktion br_learn jedoch nur ein Konstrukt des Typs sk_buff, das keinen Eintrag für VLAN's vorsieht. Der Erweiterung von sk_buff um VLAN's steht entgegen, daß diese Struktur in weiten Bereichen des Netzcodes innerhalb des Kernels als temporärer Puffer eingesetzt wird. Daher sind die Folgen einer Änderung von sk_buff unvorhersehbar.

Die Event-Schnittstelle

Sie dient dazu, den Anschluß neuer Systeme bekannt zu geben. Um Meldungen über Ereignisse weiterzuleiten, hat der Kernel mehrere Möglichkeiten. Zum einen kann er Ereignisse über die Funktion printk an den *syslogd* zur Verarbeitung weitergeben. Eine andere Möglichkeit ist die Schnittstelle des /proc-Filesystems. Diese wird im Allgemeinen dazu verwendet, Statusinformationen mittles Tabellen auszugeben. Beispiele hierfür sind die *arp*- und die *routing*-Tabelle. Die dritte Variante ist das sogenannte *netlink*-Device. Dabei handelt es sich um einen Kernel-Treiber, der bidirektionale Kommunikation erlaubt. Ein Beispiel für die Implementierung dieser Schnittstelle liefert der *arpd*. Bei vielen Systemen im Netz verlagert der *arpd* seine Tabellen aus dem Kernel hinaus in den User-Bereich. Die Einführung der *netlink*-Schnittstelle steht auf der Todo-Liste der Bridge Entwickler.

Voraussetzung für die Verarbeitung von Nachrichten ist das Einrichten eines neuen Device. Die Majornummer der netlink-Schnittstelle ist dabei 36. Für die Minornummer war die 9 noch frei. Somit mußte die Datei /usr/src/linux/include/net/netlink.h um den Eintrag #define NETLINK_BRD 9 erweitert werden. Das Device wird in der Funktion __initfunc des Switching Codes br.c durch den Aufruf von netlink_attach(NETLINK_BRD, brd_callback) initialisiert. Durch den Aufruf von netlink_post(NETLINK_BRD, skb) in der Funktion brd_send wird dem Device /dev/brd das gewünschte Event übergeben. Momentan besteht das Event aus dem Port, der MAC-Adresse, der VLAN ID, einem Feld Flags und einem Timer-Feld.

Die Funktion brd_send wird in br_learn nach dem Einfügen eines neuen Systems in den AVL-Baum aufgerufen und damit dem netlink-Device das Hinzukommen eines neuen Systems angezeigt. Die für das Event nötigen Informationen werden der Struktur fdb entnommen.

Einbinden des neuen Codes

Grundlage für den Bridging-Code ist, wie schon erwähnt wurde, die Kernelversion 2.1.59. Um den Code auf neue Kernelversionen übertragen zu können, braucht man den Patch, der im Anhang A beigefügt ist. Neue Kernelversionen finden sich im Internet unter ftp.cs.helsinki.fi/pub/Linux_Kernel oder auf einem bekannten Mirror. Die Installation des Kernels ist in der Datei README beschrieben und soll hier daher nur kurz geschildert werden.

Der Sourcecode des Kernels kommt als gepacktes TAR-Archiv. Zum Entpacken in /usr/src verwendet man GNUzip:

```
cd /usr/src
gzip -cd linux-2.1.XX.tar.gz | tar xfv -
```

Dabei bezeichnet "XX" die Versionsnummer des Kernels. Anschließend wird der benötigte Patch für die VLAN-Funktionalität eingespielt:

```
cd /usr/src
gzip -cd patchXX.gz | patch -p0
```

Danach können die Originaldateien (xxx~ oder xxx.orig) gelöscht werden. Schlägt der Versuch, den Kernel zu Patchen fehl, was man an Dateien mit der Endung `#` oder `.ref` erkennt, müssen die entsprechenden Dateien „per Hand“ gepatcht werden.

Das Standardverzeichnis für den Kernel ist `/usr/src/linux`. Unter `/usr/include` müssen symbolische Links angelegt werden (`/usr/include/asm`, `/usr/include/linux` und`/usr/include/scsi`):

```
cd /usr/include
rm -rf asm linux scsi
ln -s /usr/src/linux/include/asm-i386 asm
ln -s /usr/src/linux/include/linux linux
ln -s /usr/src/linux/include/scsi scsi
```

Nach einem `cd /usr/src/linux` kann der Kernel durch die Eingabe von `make config` konfiguriert werden. Für das korrekte Funktionieren des Bridging-Codes müssen mindestens folgende Optionen gesetzt werden:

```
CONFIG_EXPERIMENTAL=y
CONFIG_NET=y
CONFIG_NETLINK=y
CONFIG_INET=y
CONFIG_BRIDGE=y
CONFIG_NET_ETHERNET=y
```

Wichtig ist, mit `CONFIG_EXPERIMENTAL=y` dafür zu sorgen, daß auch nach experimentellem Code abgefragt wird. Der Bridging-Code gehört dazu, obwohl er im vorliegenden Kernel 2.1.59 stabil läuft[5]. Für die Eventschnittstelle ist das Einbinden des netlink-Device (`CONFIG_NETLINK=y`) nötig. Außerdem sollten noch die Netzkarten, die sich im Rechner befinden, in den Kernel eingebunden werden. Nach der Konfiguration des Kernels finden sich alle gemachten Einträge in der Datei `.config` wieder.

Nach der Eingabe von `make dep` kann der Kernel übersetzt werden. Dazu sind die in der README-Datei angegebenen Programme mit der entsprechenden Versionsnummer notwendig. Die für die Testversion benutzte Linux-Distribution Debian 2.0 beinhaltet diese. Das Übersetzen des Kernels geschieht mittels:

```
make clean
make zImage
make modules
make modules_install
```

[5]Der Testrechner lief mehr als eine Woche mit aktiviertem Bridging-Code stabil durch, ebenso nach dem Einfügen der VLAN-Funktionalität.

Anschließend ist der erstellte Kernel noch ins Rootverzeichnis zu kopieren: `cp /usr/src/linux/arch/i386/boot/zImage /vmlinuz`. Um von diesem Kernel booten zu können, ist noch der Aufruf vom Linux-Bootloader `/sbin/lilo` nötig. Leider erkennt der Linuxkernel nicht von selbst, daß mehr als eine Netzkarte im System vorhanden ist. Daher ist dem lilo durch die Option `eth=0,0,eth1` mitzuteilen, daß eine zweite Karte im System steckt. Für weitergehende Dokumentation soll auf das Ethernet HOWTO [Gor] verwiesen werden, daß sich üblicherweise unter `/usr/doc/HOWTO` findet.

Um den Bridging-Code zu aktivieren muß die Datei `/etc/init.d/bridgex` dahingehend angepaßt werden, daß alle für das Bridging vorgesehenen Netzschnittstellen (eth0, eth1, ...) dort eingetragen werden. Gestartet wird der Code entweder über `/etc/init.d/bridge start` oder durch die Managementsoftware.

5.1.2 Anpassung der Bridgesoftware

Für die Konfiguration des vorhandenen Bridging-Codes ist ein Programm namens *brcfg* verantwortlich. Die Schnittstelle zum Eventservice von CORBA nimmt ein Programm ein, das das netlink-Device abhört und die Ausgaben nach `<stderr>` ausgibt.

Das Konfigurationstool *brcfg*

Die Anpassung des Codes umfaßt drei Teile. Zum einen muß der Hilfsausdruck (`brcfg -h`, siehe unten) dahingehend angepaßt werden, daß Einträge für das Setzen der Port- und der MAC-Adressen VLAN's hinzukommen:

```
brcfg - Bridge Configuration tool v0.2
--------------------------------------
brcfg sta[rt]                           Start Bridge
brcfg sto[p]                            Stop Bridge
brcfg p[ort] x e[nable]                 Enable a port
brcfg p[ort] x d[isable]                Disable a port
brcfg p[ort] x p[riority] y             Set the priority of a port
brcfg p[ort] x v[lan] y                 Set the vlan_id of a port
brcfg m[ac] x v[lan] y                  Set the vlan_id of a mac address
brcfg pr[iority] y                      Set bridge priority
brcfg pa[thcost] y                      Set the pathcosts
brcfg d[ebug] on                        Switch debugging on
brcfg d[ebug] off                       Switch debugging off
brcfg pol[icy] r[eject]/a[ccept]        Switch the policy/flush protocol list
brcfg e[xempt] <protocol> ..            Set list of exempt protocols
brcfg l[ist]                            List available protocols
brcfg stat[s] z[ero]                    Reset Statistics counters
brcfg stat[s] d[isable]                 Switch protocol statistics off
brcfg stat[s] e[nable]                  Switch keeping protocol statistics on
brcfg stat[s] s[how]                    Show protocol statistics
brcfg statu[s]                          Show bridge status

Examples:
---------
```

```
brcfg start exempt atalk aarp          Bridge start dont do LocalTalk bridging
brcfg stop                             Bridge stop
```

Dann muß die gewünschte Funktion implementiert werden und schließlich noch eine Statusausgabe, die ausgegeben wird, wenn *brcfg* ohne Parameter gestartet wird, eingefügt werden. Der Hilfsausdruck beschränkt sich auf das Einfügen zweier Zeilen. Für die Statusausgabe ist der Funktion disp_ports die Zeile `printf("\tvlan id 0x\%08x\n",ports[i].vlan\_id)` hinzuzufügen, die die VLAN ID des betreffenden Ports ausgibt. Aufwendiger ist die Implementierung der Funktionen create_vlan und create_mac_vlan.

Zunächst war die Case-Anweisung des Kommandozeilenparsers durch das Hinzufügen der Parameter für den Aufruf obiger Funktionen zu erweitern. Die Funktion create_vlan erwartet als Parameter den Port als String und ebenso als String die VLAN ID. Diese Werte werden im nächsten Schritt in Integerwerte umgewandelt. Alsdann werden die Werte durch den Aufruf von cmd(BRCMD_VLAN_CONFIG,po,id,) der Funktion cmd übergeben, die den eigentlichen I/O Aufruf absetzt.

```
323 void create_vlan(char *port,char *vlan)
324 { int po=atoi(port),id=atoi(vlan);
325   cmd(BRCMD_VLAN_CONFIG,po,id,"");
326   printf("Vlan_id for port %d set to 0x%08x\n",po,id);
327 }
```

Als Parameter erwartet cmd den Typ des I/O Aufrufs, zwei Integerwerte und schließlich ein Stringfeld zur Aufnahme der MAC-Adresse. Dieses dritte Feld muß der Struktur `br_cf`, die in `br.h` zu finden ist, noch hinzugefügt werden. Durch die Übergabe der MAC-Adresse als String kann dem Kernel der Wert der MAC-Adresse schon im richtigen Format übergeben werden.

```
329 void create_mac_vlan(char *mac,char *vlan)
330 { unsigned int i,j;
331   unsigned char ula[6];
332   int id=atoi(vlan);
333
334   for ( i=0; i<6; i++) {
335         j = i;
336         sscanf( &mac[j*2], "%02x", &ula[j]);
337   }

[...]

347 cmd(BRCMD_MAC_VLAN,id,0,ula);
348 }
```

brcfg selbst erwartet die Eingabe der MAC-Adresse als String ohne die sonst üblichen Doppelpunkte. Mittels der Funktion sscanf wird dieser String als 6 Byte großer, hexadezimaler String in den Speicher geschrieben. Anschließend wird der I/O Aufruf durch den Aufruf der Funktion cmd initiiert.

Für die Einbindung des neuen Codes in die Distribution geht man analog zu der Beschreibung aus Kapitel 5.1.1 vor. Der Patch für die Konfigurations-

software ist im Anhang B abgedruckt. Besitzt man eine Debian-Distribution, so kann man sich den Source zum Binärpaket (`bridgex_0.25.deb`) unter `ftp.de.debian.org` aus dem Internet laden. Für andere Distributionen steht die Datei `bridgex-0.25.tar.gz` zum *Download* bereit. Für das Entpacken und Patchen des Sourcecodes gilt das, was schon für den Kernelsource gesagt wurde. Nach dem Auspacken des Sourcecodes in einem beliebigen temporären Verzeichnis, sollte ein einfaches `make` den Source übersetzen. Anschließend ist das neu erstellte Programm `brcfg` nach `/sbin` zu kopieren.

Der Bridge-Daemon

Dieser horcht auf das Device `/dev/brd` und gibt die dort anliegenden Informationen an `<stderr>` aus. Beim Starten des Bridge Daemons gibt dieser eine Statusmeldung an den *syslogd* ab. Ebenso, wenn sich die Datei `/dev/brd` nicht zum Lesen öffnen läßt. In einer `while` Schleife liest der Prozeß aus dem Device die Struktur `fdb` und gibt die darin enthaltenen Informationen an die Funktion `brd_print` weiter. Diese hat lediglich noch die Aufgabe, die Felder der Struktur getrennt voneinander nach `<stderr>` auszugeben. Zusätzlich werden die Werte noch mit einem eindeutigen *Identifier* versehen.

Der Sourcecode zu diesem Programm ist im Anhang C abgedruckt. Durch das ebenfalls angehängte `Makefile` (Anhang D) läßt sich das Programm mit einem einfachen Aufruf von `make` übersetzen. Es ist anschließend noch an einen geeigneten Ort zu kopieren (bspw. `/usr/local/bin`). Für einen ordnungsgemäßen Ablauf des Programms muß noch ein Device unter `/dev` angelegt werden:

```
mknod /dev/brd c 36 9
chown root.root /dev/brd
chmod 660 /dev/brd
```

Für diesen Vorgang sind Rootrechte notwendig.

5.2 Schnittstelle zum Manager

Wie aus Abbildung 5.1 zu ersehen ist, besitzt der *Management-Agent-Prozeß* Schnittstellen zwischen dem in der Sprache C geschriebenen Konfigurationswerkzeug *brcfg* und dem ORB. Die Schnittstelle zum ORB wird in [Kem98] näher erläutert. In dieser Arbeit soll nur ein prinzipieller Überblick über die Anbindung an den ORB gegeben werden. Der *Management-Agent-Prozeß* ist ein Java geschriebenes Programm, dessen Methoden durch Aufruf von *brcfg* mit entsprechenden Parametern Funktionen der Switchingsoftware ausführen.

5.2.1 Entwicklungswerkzeuge

Die Entwicklung des *Switch-Management-Agents* erfolgt aus dem in Kapitel 4.4 vorgestellten Objektmodell. Das Objektmodell wurde mit dem CASE-Tool StP erstellt. StP ist in der Lage, aus dem graphischen Objektmodell IDL-Code zu erzeugen. Mit Hilfe von `idl2java` wurde schließlich Java-Code aus der IDL-Beschreibung generiert. Dieser Java-Code bildet das Gerüst. Er stellt Interfaces für die Methoden und Attribute des Objektmodells zur Verfügung. Die Implementierung dieser Interfaces stellt die eigentliche Aufgabe der Schnittstellenimplementierung dar.

Visibroker 3.0 for Java

Der *Visibroker for Java* [Vis97b, Vis97a, Vis97c, OH97] der Firma *Visigenic* ist ein CORBA 2.0 konformer Client und Server ORB, der gänzlich in Java geschrieben wurde. Er unterstützt sowohl dynamische als auch statische CORBA-Methodenaufrufe. Servermethoden können entweder über Clientanwendungen aufgerufen werden, oder von Java Applets aus einem Browser heraus. Neben dem ORB liefert *Visibroker for Java* eine komplette Entwicklungsumgebung für die Generierung von CORBA-Objekten sowie Administrationstools für die Überwachung der Kommunikation mit. Für die Implementierung des Prototypen kamen folgende Komponenten zum Einsatz:

- `idl2java:` Dabei handelt es sich um einen Compiler, der aus vorhandenen IDL-Schnittstellen die für die Implementierung nötigen Client-Stubs, Server-Skeletons und Interfaces in Java generiert.
- `osagent:` Der *osagent* bildet die Kommunikationsgrundlage für CORBA-Objekte. Er wird als der sogenannte *Smart Agent* bezeichnet, der den ORB enthält. Nach dem Hochfahren auf einem Rechner im Netz kann der *Smart Agent* von den Client- und Server-Objekten durch einen UDP-Broadcast ausfindig gemacht werden. Die für den Broadcast nötige Portadresse kann über eine Environmentvariable (`$OSAGENT_PORT`) gesetzt werden.

Der *idljava*-Compiler erzeugt aus den vom Entwickler geschriebenen IDL-Schnittstellen die notwendigen Client-Stubs und Server-Skeletons. Folgende Programme werden beim Aufruf von *idl2java* erzeugt (siehe Abbildung 5.3):

- `<InterfaceName>.java` enthält das Java-Interface, daß dem IDL-Interface entspricht.
- `_<InterfaceName>ImplBase.java` ist in der Regel die Oberklasse aller Server-Objektklassen. Diese Klasse erbt von den relevanten CORBA/Java-Klassen und implementiert das Java-Interface `<InterfaceName>.java`.
- `_portable_stub_<InterfaceName>.java` liefert den Client-Stub.
- `_sk_<InterfaceName>.java` entspricht dem Server-Skeleton.

- `_example_<InterfaceName>.java` stellt eine Klasse zur Verfügung, die als Beispiel für die Objektimplementierung der Klasse `<InterfaceName>` hergenommen werden kann.

Folgende Klassen werden nur der Vollständigkeit halber angesprochen. Nähere Beschreibungen dazu finden sich in [Vis97b, Vis97a, Vis97c, OH97] und [Kem98, Rad98].

- `_tie_<InterfaceName>.java` wird benötigt, falls die Server-Objektklasse nicht von der `_<InterfaceName>ImplBase.java` Klasse erben kann.
- `<InterfaceName>Operations.java` wird im Zusammenhang mit der vorherigen Klasse benötigt.
- `<InnterfaceName>Helper.java` stellt die Helper-Klassen zur Verfügung. Diese bieten nützliche Methoden wie bspw. das *Casting*, also Typumwandlungen, von CORBA-Objekten.
- `<InterfaceName>Holder.java` ist die zugehörige Holder-Klasse.

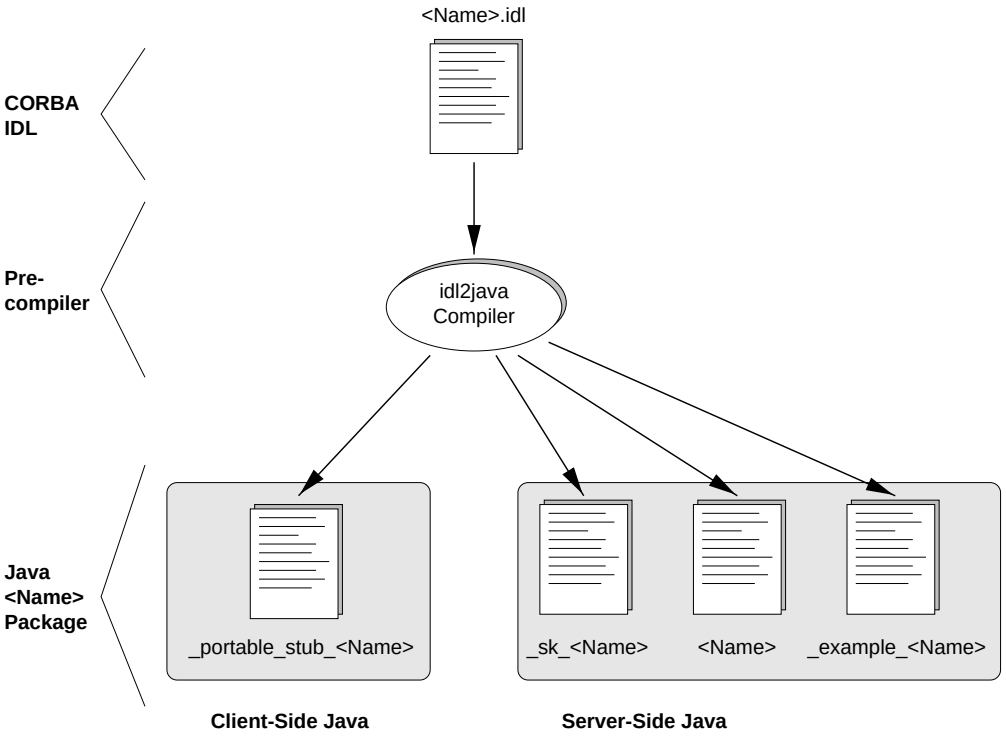


Abbildung 5.3: Von `idl2java` generierte Java Klassen und Interfaces

Software through Picture (StP)

Zur visuellen Erstellung des in Kapitel 4.3 erarbeiteten Objektmodells kam das CASE-Tool *Software through Pictures* (StP) der Firma *Aonix* [Aon97] zum Einsatz. Neben der graphischen Unterstützung der Entwickler bei der Erstellung

OMT-konformer Objektmodelle, generiert diese Tool, daß in der Version 2.4.2 vorliegt, Programm- und Klassenskelette für verschiedene Spezifikations- und Programmiersprachen, darunter auch IDL. Durch Verwendung eines sogenannten *Classtables*-Editors können Klassenattribute und Methoden näher spezifiziert werden. Neben Schlüsselwörtern wie `public` und `private` können auch andere programmiersprachenspezifische Elemente festgelegt werden. Die erstellten Klassen, Objekte und Assoziationen werden in einem gemeinsamen *Repository* verwaltet.

5.2.2 Der Switch-Agent

Der Switch Agent wurde anhand des Objektmodells aus Abbildung 4.4 entwickelt. Die Klasse `SwitchStationaryAgent` implementiert dabei die Methoden des Interface `Switch`. Um die benötigten Codegerüste zu bekommen, mußte der von StP generierte IDL-Code mittels `idl2java` nach Java übersetzt werden. Die daraus generierte Klasse `_example_Switch.java` wurde nach `SwitchStationaryAgent.java` umbenannt und anschließend erweitert.

Zum Einbinden des Agenten in das Managementsystem muß der Code der Klasse folgenden Kopf besitzen:

```
1 import de.unimuenchen.informatik.mnm.masa.agent.*;
2 import Port;
3 import DTE;

4 public class SwitchStationaryAgent
5   extends StationaryAgent
6   implements SwitchOperations
```

Diese Vorgaben werden in [Kem98] gemacht. Die Klasse stützt sich auf Methoden aus dem Paket `de.unimuenchen.informatik.mnm.masa.agent` auf. Sie erbt von der Klasse `StationaryAgent`, die ebenfalls in [Kem98] beschrieben ist und implementiert die Klasse `SwitchOperations`, die die Interface-Klasse für das Objekt Switch darstellt. Darüberhinaus müssen die Klassen `Port` und `DTE` eingebunden werden.

Wie man sieht, erbt die Klasse `SwitchStationaryAgent` nicht wie üblich von der Klasse `_SwitchImplBase`, die die Anbindung an den ORB realisierten würde, sondern von der Klasse `StationaryAgent`, die von Bernhard Kempter im Rahmen von [Kem98] entwickelt wurde. Das dort entworfene Agentensystem kümmert sich auch um die Anbindung der Agenten an den ORB.

Das folgende Programmstück stellt die Konstruktermethode vor. Für insgesamt acht Ports[6] werden neue Objekte instantiiert und einem Array von Ports zugeordnet.

```
1  int n = 8;
```

[6] Es werden acht Ports vom Linux-Kernel als Default verwendet.

```
2  Port port_array[] = new Port[n];
3
4  /** Construct a persistently named object. */
5  /** Construct a transient object. */
6  public SwitchStationaryAgent() {
7    super();
8
9    for (int i=0; i<n; i++) {
10      Port p = new Port(i);
11      port_array[i] = p;
12   }
13
14 }
```

Die Methode `bridge_config()` ruft das in Kapitel 5.1.2 erweiterte Konfigurationstool `brcfg` mit dem ihr übergebenen String-Parameter auf:

```
1  public void bridge_config(java.lang.String parameter) {
2    // executes brcfg with specified parameter
3    try {
4      Runtime runshell = Runtime.getRuntime();
5      Process process =
6        (Process)runshell.exec(new String("/sbin/brcfg "
7                                          +parameter));
8    }
9
10    catch (java.io.IOException e) {
11     // error, read() or exec() failed
12     //throw new ResourceException();
13     e.printStackTrace();
14   }
15 }
```

Anschließend folgen die in 4.4 spezifizierten Methoden. Diese rufen dann nur noch die Methode `bridge_config()` mit dem entsprechenden Parameter auf. Die Implementierung des Prototypen beschränkt sich aus zeitlichen Gründen auf exemplarische Methoden wie `start()`, `stop()`, `debug()`, `stats()`, `enable_port()`, `disable_port()`, `set_port_vlan_id()` und `set_mac_vlan_id()`, die im folgenden beschrieben werden.

Die Methode zum Starten des Switch sieht so aus:

```
1 public void start() {
2   // implement operation...
3   this.bridge_config("start");
4 }
```

Sie macht nichts anderes, als `bridge_config()` mit dem Parameter `start` aufzurufen. Analoges gilt für die Methode zum Stoppen des Switch. Interessanter sind da schon die Methoden, die auf dem Objekt Port arbeiten:

```
1 public void enable_port(int number) {
2   // implement operation...
3   // enable the specified port
4   port_array[number].enable_port();
5 }
```

Veranlaßt den Port mit der übergebenen Nummer seine Bridging-Funktionalität (wieder) aufzunehmen. Dazu wird die Methode `enable_port()` des Objekts `Port` aufgerufen. Das Abschalten eines Ports gleicht diesem Methodenaufruf, mit der Ausnahme, daß statt `enable_port()` die Methode `disable_port()` aufgerufen wird.

Der Methode `debug()` wird ein String als Parameter übergeben, der wiederum der Methode `bridge_config()` übergeben wird. Gleiches gilt für die Methode `stats()`.

```
1 public void debug(java.lang.String para) {
2   // implement operation...
3   this.bridge_config("debug "+para);
4 }
```

Für das Setzen der VLAN ID eines Ports wird der Methode `set_vlan_id()` der Klasse Port als Parameter die VLAN ID übergeben:

```
1 public void set_port_vlan_id(
2                               int number,
3                               int vlan_id
4 ) {
5   // implement operation...
6   port_array[number].set_vlan_id(vlan_id);
7 }
```

Das Setzen der VLAN ID für eine MAC-Adresse veranschaulicht folgende Methode. Dieser wird die MAC-Adresse als String und die VLAN ID als Integer übergeben. In Zeile 7 wird ein neues Objekt der Klasse DTE instatiiert, mit der Variable `dte` eine Referenz auf die Instanz gelegt und dem Konstruktor die MAC-Adresse übergeben. Anschliessend wird in Zeile 10 der neu erzeugten Instanz über die Methode `set_vlan_id()` eine VLAN ID zugewiesen:

```
1  public void set_mac_vlan_id(
2                               java.lang.String mac,
3                               int vlan_id
4  ) {
5    // implement operation...
6    // instantiiere neues Objekt
```

```
7    DTE dte = new DTE(mac);
8
9    // Setze vlan_id
10   dte.set_vlan_id(vlan_id);
11 }
```

Am Ende stehen die Methoden `cleanUp()` und `finalize()`, die dazu dienen, nicht mehr benötigte Referenzen beim Beenden des Agenten explizit aufzuräumen. Java sollte das aber ohnehin von selber bewerkstelligen, so daß diese Methoden hier leer bleiben.

Der Code der Klasse Port besteht aus einem Konstruktor, der der erzeugten Instanz eine Nummer, die Portnummer, zuordnet. Ausserdem wurden die Methoden `enable_port()`, `disable_port()` und `set_vlan_id()` für diesen Prototypen exemplarisch implementiert. Der Code ansich bietet nichts Neues. Auch hier dient eine Methode `brcfg` mit einem String als Parameter zum Aufruf des C-Programms `brcfg`, das die eigentliche Konfiguration des Switch übernimmt.

Die Klasse `DTE` ist ähnlich wie die Klasse `Port` aufgebaut. Ein Konstruktor weist der neu erzeugten Instanz der Klasse die übergebene MAC-Adresse zu. Sie stellt die ID des Objekts dar. Neben der Methode `brcfg`, die der gleichnamigen Methode des Objekts `Port` gleicht, stellt das Objekt `DTE` nur die Methode `set_vlan_id()` zur Verfügung, der als Parameter die VLAN ID als String übergeben wird:

```
1 public void set_vlan_id(int vlan_id)
2   {
3     //java.lang.String mac = new java.lang.String(mac);
4     Integer vlan = new Integer(vlan_id);
5     this.brcfg(("mac ")+mac+(" vlan ")+vlan.toString());
6   }
```

5.2.3 Binden an den Event Channel

In diesem Kapitel soll die Implementierung der Anbindung des Agenten an den Event Channel betrachtet werden. Dazu ist es nötig, auf das Prinzip der Anbindung an den Event Channel näher einzugehen.

Wird ein Event Channel zum ersten Mal erzeugt, besitzt er weder Supplier noch Consumer. Diese beiden führen folgende Schritte aus, um sich an den Event Channel zu binden bzw. mit diesem zu kommunizieren:

- Binden an den Event Channel,
- Referenz auf das Proxy-Objekt gewinnen,
- sich an das Proxy-Objekt anschließen,
- Daten verschicken oder Empfangen.

Im Falle des PC-basierten Switch wird das üblichere Verfahren (nach [Vis97c]) des Push-Modells angewendet. Das Pull-Modell soll hier nicht betrachtet werden, da das Push-Modell sich für das Verschicken von Ereignismeldungen besser eignet. Hierzu muß das Agentenobjekt das *PushSupplier*-Interface implementieren. Der Manager implementiert entsprechend das *PushConsumer*-Interface. Die für den Agenten nötigen Schritte sind in Tabelle 5.1 dargestellt.

Schritt	**Push Supplier**
Binden an den Event Channel	EventChannelHelper::bind()
Referenz auf das Proxy Objekt	EventChannel::for_suppliers()
Consumer Proxy anfordern	SupplierAdmin::obtain_push_consumer()
Supplier an den Proxy binden	ProxyPushConsumer::connect_push_supplier()
Daten verschicken	ProxyPushConsumer::push

Tabelle 5.1: Binden an den Eventchannel

Für den Prototypen wurde eine Klasse eventPushSupplier.java geschrieben. Diese greift die Information, die der Switch über die *netlink*-Schnittstelle des Linux-Kernels an einen externen Dämonen (`brd`) weiterleitet und von diesem auf `<stderr>` ausgegeben wird, auf und schickt sie dann an einen vorher definierten Event Channel. Der Name dieses Event Channels wird im Feld `event_channel_name` des in Kapitel 4.5 geschilderten *structured events* übergeben.

Der Event Channel wird vom Eventprogramm des Agenten (eventPushSupplier.java) erzeugt und beim BOA angemeldet.

```
124       // Initialize the ORB and BOA
125       org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
126       org.omg.CORBA.BOA boa = orb.BOA_init();
```

Das Eventprogramm erbt von der Klasse _PushSupplierImplBase. Nach der Initialisierung des ORB und BOA erzeugt es einen Event Channel (create_channel()):

```
128   // Create the channel
129   EventChannel channel = EventLibrary.create_channel(boa,
130                                        "AgentInitChannel",
131                                                    false,
132                                                     100);
```

Anschließend fordert es einen *ProxyPushConsumer* an (obtain_push_consumer()). Um den *PushSupplier* des Agenten mit dem Proxy-Objekt im Event Channel zu verbinden, führt der Agent einen *connect* aus (connect_push_supplier()).

```
133   // Create the proxy-consumer
134   ProxyPushConsumer proxyPushConsumer =
135     channel.for_suppliers().obtain_push_consumer();
136
```

```
137   // Connect a supplier to proxy-consumer
138   eventPushSupplier supplier = new eventPushSupplier(channel,
139                                                 proxyPushConsumer);

[...]

146   try {
147     proxyPushConsumer.connect_push_supplier(supplier);
148   }
```

Durch Aufruf der Methode push() auf dem Proxy-Objekt ist der Agent nun in der Lage Statusmeldungen in den Event Channel zu pushen.

```
202   // Push message in the channel
203   System.out.println("Supplier pushing: "+any.toString());
204   try {
205     proxyPushConsumer.push(any);
206   }
```

Der VisiBroker Event Service unterstützt momentan lediglich generische Ereignismeldungen. Daher müssen die Daten, die in den Event Channel gepusht werden, zwingend als CORBA *Any*-Datentyp verpackt werden (Vorgabe des EventChannel-Interfaces). Überdies gibt es keine garantierte Datenübertragung. Die Datensicherheit stützt sich allein auf die Sicherheitsmechanismen von TCP/IP, welches zur Kommunikation zwischen Sender und Empfänger zur Anwendung kommt.

Der Vorgang für das Versenden der Ereignismeldungen des Switch gleicht dem bisher beschriebenen. Zunächst werden der BOA und der ORB initialisiert und ein neuer Event Channel mit dem Namen `PCSwitch` kreiert. Anschließend wird ein *ProxyPushConsumer* angefordert (obtain_push_consumer()). Um den *PushSupplier* des Agenten mit dem Proxy-Objekt im Event Channel zu verbinden, führt der Agent einen *connect* aus (connect_push_supplier()).

```
228   // Create the proxy-consumer
229   ProxyPushConsumer myproxyPushConsumer =
230     myEventChannel.for_suppliers().obtain_push_consumer();
231
232   // Instantiiere neues Objekt
233   myEventPushSupplier = new eventPushSupplier(myEventChannel,
234                                           myproxyPushConsumer);

[...]

243   try {
244     myEventPushSupplier.myproxyPushConsumer.
245       connect_push_supplier(myEventPushSupplier);
246   }
```

Dabei ist `eventPushSupplier()` die Konstruktormethode. Dieser werden die Referenz auf den Event Channel und dem proxyPushConsumer übergeben, damit diese beiden Attribute später in den Event Channel gepusht werden können:

```
295   myEventPushSupplier.push_event( port, mac);
```

Die Methode `push_event()` konstruiert den nötigen *structured event*. Der Aufbau dieses Events für die Übermittlung der Port- und MAC-Adresse an den Manager zeigt Abbildung 5.4.

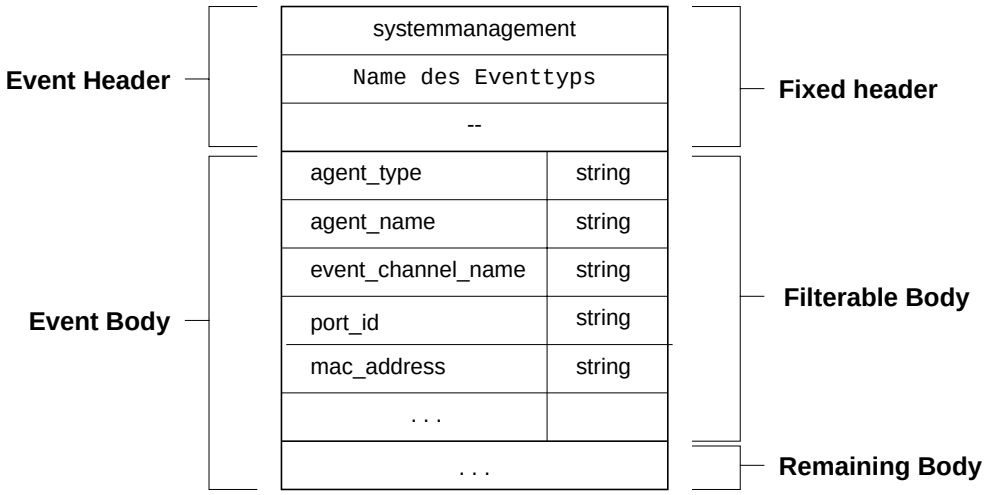


Abbildung 5.4: Aufbau eines Structured Events für den PCSwitch Event-Channel

Der Fixed Header besteht aus dem Eintrag systemmanagement und dem Namen des Eventtyps (newsystem). Der Filterable Body sieht die Felder für den agent_type, agent_name und den event_channel_name vor, die mit den gleichen Werten wie in Kapitel 4.5 belegt werden. Anschließend folgen die Felder port_id und mac_address, die die Portadresse und die MAC-Adresse aufnehmen.

Kapitel 6

Zusammenfassung und Ausblick

Es wurde eine Managementschnittstelle für einen PC-basierten Switch modelliert und implementiert, die aus folgenden Teilbereichen besteht:

- Der Erweiterung des Switching-Codes um VLAN-Funktionalität.
- Der Anpassung eines Werkzeuges zur Konfiguration des Switch.
- Dem Agenten, der in eine Managementumgebung mit verteilten Agenten eingebunden wurde.
- Der Anbindung des Switching-Codes an eine Schnittstelle zum Austausch von Ereignismeldungen (netlink-Device).
- Dem Ereignisdienst, der die asynchrone Kommunikation ermöglicht, durch den CORBA Event Service.
- Einer Schnittstelle zwischen dem CORBA Event Service und dem Switching-Code.
- Der Anbindung des Switching-Codes an eine Schnittstelle zum Austausch von Ereignismeldungen (netlink-Device).

Als Kommunikationsinfrastruktur wurde CORBA verwendet, das eine Verteilung und Kommunikation verschiedener Agenten unterstützt. Die Schnittstellen der Agenten sind in IDL spezifiziert und damit in ihrer Funktionalität festgelegt.

Die Verwendung von Java als Implementierungssprache für den Agenten bietet den Vorteil, daß Java plattformunabhängig ist und der benötigte Java-Interpreter auf vielen Plattformen erhältlich ist. Wegen seiner objektorientierten Struktur eignet sich Java hervorragend für die Zusammenarbeit mit CORBA.

Ein Problem, das sich bei der Erstellung dieser Arbeit herausstellte, ist die Tatsache, daß der Switching- (Bridging-) Code im Linux-Kernel noch relativ unstabil ist. Ohnehin mußte für einen funktionstüchtigen Betrieb des Switch ein noch in der Entwicklung befindlicher Kernel hergenommen werden. Dadurch sind Betriebssystemabstürze nicht auszuschließen und schon gar nicht immer deutlich

einzuordnen, von welchem Teil des Kernel-Codes der Absturz ausging. Von einem produktionstechnischen Einsatz des entwickelten PC-basierten Switch ist daher eher abzuraten. Für den Betrieb in einem Forschungsumfeld ist der Code allerdings hinreichend stabil.

Eine weitere Einschränkung ergab sich aus dem Umstand, daß sich die Implementierung des *Notification Service* noch in der Entwicklungsphase befindet. So konnten die erweiterten Filtermöglichkeiten ebensowenig genutzt werden wie die erweiterten Möglichkeiten der Nutzung von anderen Datentypen als dem CORBA *Any*-Datentyp.

Zu tun bleibt noch die vollständige Implementierung aller im Objektmodell angebotenen Methoden des Agenten. Dies betrifft v. a. die Methoden die beim Umgang mit dem Spanning Tree-Algorithmus in Frage kommen. Außerdem fehlt noch ein Erfahrungsbericht über das Zusammenspiel der im Rahmen des Gesamtprojekts entwickelten Agenten und dem darüberliegenden Agentensystem.

Auch wäre noch eine Betrachtung von Sicherheitaspekten interessant. Problematisch ist in dieser Hinsicht v. a. der Umstand, daß der Agent auf dem Switch mit Rootrechten läuft. Diese Rechte benötigt er, um die von ihm zu verwaltenen Ressourcen setzen zu können. Nun kann im Prinzip jeder Agent, der sich an den ORB bindet, über den Naming Service eine CORBA-Objektreferenz erlangen und auf dieser eine Methode, die die Schnittstelle anbietet, anwenden. Dazu muß der aufrufende Agent weder mit Rootrechten laufen, noch auf dem PC-basierten Switch selber. Es wird also ein geeignetes Autorisierungsverfahren benötigt, daß es dem auf dem Switch ablaufendem Agenten ermöglicht, Methodenaufrufe nur dann zuzulassen, wenn sich der „Anfragesteller“ vorher eindeutig identifiziert hat. Zu diesem Thema gibt es eine recht ausführliche Spezifikation[GMD97]. Die Implementierung dieser Spezifikation steht noch aus.

Abkürzungen

API	Application Programming Interface
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
BOA	Basic Objekt Adapter
BOOTP	Bootstrap Protocol
BPDU	Bridge Protocol Data Unit
CORBA	Common Object Request Broker Architecture
DII	Dynamic Invocation Interface
DNS	Domain Name Service
DSI	Dynamic Skeleton Interface
DTE	Data Terminal Equipment
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
ISO	International Organization for Standardization
IEEE	Institute of Electrical Electronics and Engineers
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol
LAN	Local Area Network
MAC	Medium Access Control
MASA	Mobile Agent System Architecture
MIB	Management Information Base
NIS	Network Information Service
OMA	Object Management Architecture
OMG	Object Management Group
OMT	Object Modeling Technique
ORB	Object Request Broker
OSI	Open Systems Interconnect
PC	Personal Computer
PDU	Protocol Data Unit
QoS	Quality of Service
RARP	Reverse Address Resolution Protocol
RFC	Request for Comment
SNMP	Simple Network Management Protocol
StP	Software through Pictures
TCP	Transmission Control Programm
UDP	User Datagram Protocol
URL	Uniform Resource Identifiers
VLAN	Virtual Local Area Network
WAN	Wide Area Network

Literaturverzeichnis

[Aon97] AONIX: *Software through Pictures/Object Modeling Technique: Creating OMT Models.* Aonix, Inc., 1997. Release 3.4.

[BAS] *Basic Bridging: What, Why, and How.* World Wide Web. http://www.otimized.com/tech_cmp/bridge.html.

[BRIa] *Bridges.* World Wide Web. http://www.scit.wlv.ac.uk/~jphb/comms/bridge.html.

[BRIb] *Bridging.* World Wide Web. http://www.FreeSoft.org/CIE/Topics/30.htm.

[Col] COLE, CHRISTOPHER: *Bridge mini-HOWTO.* World Wide Web. http://sunsite.unc.edu/LDP/HOWTO/mini/Bridge.html.

[COR97] *CORBAservices: Common Object Service Specification.* Technischer Bericht Object Management Group, 1997.

[COR98] *The Common Object Request Broker: Architecture and Specification.* Technischer Bericht Object Management Group, 1998.

[Deb] *Debian GNU/Linux.* World Wide Web. http://www.debian.org.

[Dem98] DEMMEL, S.: *Implementierung eines portierbaren Java-DHCP-Servers mit einer CORBA-Managementschnittstelle.* Fortgeschrittenenpraktikum, Universität München, 1998.

[DR] DAWSON, TERRY und ALESSANDRO RUBINI: *NET 3-Networking HOWTO.* World Wide Web. http://sunsite.unc.edu/LDP/HOWTO/NET-3-HOWTO.html.

[Dro93] DROMS, RALPH: *RFC 1541: Dynamic Host Configuration Protocol.* World Wide Web, Dezember 1993. http://info.internet.isi.edu:80/innotes/rfc/files/rfc1541.txt.

[Dro96] DROMS, RALPH: *Authentication for DHCP Messages.* World Wide Web, Mai 1996. http://www.ietf.org/internet-drafts/draft-ietf-dhc-authentication-08.txt.

[Dro97] DROMS, RALPH: *RFC 2131: Dynamic Host Configuration Protocol.* World Wide Web, März 1997. http://info.internet.isi.edu:80/innotes/rfc/files/rfc2131.txt.

[DvT97] DITTRICH, JENS und UWE VON THIENEN: *VLANs - Migration zu modernen Netzwerken.* Thomson, 1997.

[Ell] ELLIS, SCOTT K.: *Debian libc5 to libc6 Mini-HOWTO.* World Wide Web. http://www.gate.net/ storm/FAQ/libc5-libc6-Mini-HOWTO.html.

[Fla97] FLANAGAN, D.: *Java in a Nutshell.* O'Reilly, 2. Auflage, 1997.

[GHW98] GRUSCHKE, B., S. HEILBRONNER und N. WIENOLD: *Managing Groups in Dynamic Networks.* In: *To be published*, Juli 1998.

[GMD97] GMD FOKUS: *Mobile Agents System Facility.* Technischer Bericht Object Management Group, 1997.

[Gor] GORTMAKER, PAUL: *Ethernet HOWTO.* World Wide Web. http://sunsite.unc.edu/mdw/HOWTO/Ethernet-HOWTO.html.

[Gut95] GUTSCHMIDT, M.: *Ein Objektmodell für ein integriertes Management von Systemdiensten mit Client/Server-Struktur.* Doktorarbeit, Ludwig-Maximilians-Universität München, September 1995.

[HA93] HEGERING, H.-G. und S. ABECK: *Integriertes Netz- und Systemmanagement.* Addison-Wesley, 1993.

[Hal92] HALSALL, FRED: *Data Communications, Computer Networks and Open Systems.* Addison-Wesley, Bonn, 3. Auflage, 1992.

[Hei97] HEILBRONNER, S.: *DHCP und das Management nomadischer Systeme in Intranets.* In: *Proceedings of the Opennet' 97*, Berlin, Germany, November 1997. Internet Society German Chapter.

[Hei98a] HEILBRONNER, S.: *Policy-based Management in the Support of Nomadic Computing.* In: *Proceedings of the 4th EUNICE Open European Summer School*, Munich, Germany, September 1998.

[Hei98b] HEILBRONNER, S.: *Requirements for Policy-based Management of Nomadic Computing Systems.* In: *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, Newark, DE, USA, Oktober 1998.

[Hil] HILLIARD, ROBERT D.: *README-autoup.sh.* World Wide Web. ftp://debian.vicnet.net.au/autoup/autoup.README.

[HNW95] HEGERING, H.-G., B. NEUMAIR und R. WIES: *Integriertes Management verteilter Systeme – Ein Überblick über den State-of-the-Art .* GI/ITG, Februar 1995.

[ID97] IEEE-802.1D/D15: *Information Technologie — Telecommunications and information exchange between systems — Local and metropolitan area networks — Common specification — Part 3: Media Access Control (MAC) Bridges: Revision.* Technischer Bericht IEEE, November 1997.

[ISO93] *Information Technology – Open Systems Interconnection – Systems Management – Part 2: State Management Function.* ISO 10164-2, ISO/IEC, Juni 1993.

[IW98] IEEE-802.1-WG: *Draft Standard P802.1Q/D11 - IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks.* Technischer Bericht IEEE, Juli 1998.

[Kem98] KEMPTER, B.: *Entwurf eines Java/CORBA-basierten Mobilen Agenten.* Diplomarbeit, Technische Universität München, August 1998.

[KRW] KLEINMANN, SUSAN G., SVEN RUDOLPH und JOOST WITTEVEEN: *The Debian GNU/Linux FAQ.* World Wide Web. http://www.debian.org/doc/FAQ/.

[Mau98] MAUL, O.: *Prototypische Implementierung des CORBA Notification Service.* Fortgeschrittenenpraktikum, Technische Universität München, 1998.

[Mic] MICROSYSTEMS, SUN: *JDK 1.2 Beta 4 Documentation.* World Wide Web. http://java.sun.com/products/jdk/1.2/docs/index.html.

[Mül98] MÜLLER, T.: *CORBA-basiertes Management von UNIX-Workstations mit Hilfe von ODP-Konzepten.* Diplomarbeit, Technische Universität München, Februar 1998.

[OH97] ORFALI, ROBERT und DAN HARKEY: *Client/Server Programming with JAVA and CORBA.* John Wiley & Sons, Inc., New York, 1. Auflage, 1997.

[OMG] *Object Management Group.* World Wide Web. http://www.omg.org.

[OMG98] *Notification Service (Joint Revised Submission).* TC Document telecom 98-06-15, Object Management Group, Juni 1998.

[Per93] PERLMAN, RADIA: *Interconnections – Bridges and Routers.* Addison-Wesley, Bonn, 3. Auflage, 1993.

[PRG+] PERENS, BRUCE, SVEN RUDOLPH, IGOR GROBMAN, JAMES TREACY und ADAM P. HARRIS: *Installing Debian Linux 2.0 For x86.* World Wide Web. ftp://ftp.debian.org/debian/dists/stable/main/disks-i386/current/install.html.

[R+93] RUMBAUGH, J. und OTHERS: *Objektorientiertes Modellieren und Entwerfen.* Hanser, 1993.

[Rad98] RADISIC, I.: *Konzeption eines policy-basierten Konfigurationsmanagements für nomadische Systeme in Intranets.* Diplomarbeit, Ludwig-Maximilians-Universität München, August 1998.

[Rie96] RIEGERT, G.: *Entwicklung einer Managementschnittstelle für DHCP-Server.* Diplomarbeit, Technische Universität München, August 1996.

[Ros94] ROSE, M.: *The Simple Book.* Prentice Hall, 2. Auflage, 1994.

[Vis97a] VISIGENIC: *Visibroker for Java Programmer's Guide Version 3.0*, März 1997.

[Vis97b] VISIGENIC: *Visibroker for Java Reference Manual Version 3.0*, März 1997.

[Vis97c] VISIGENIC: *Visibroker Naming and Event Services Programmer's Guide Version 3.0*, März 1997.

[Vuk96] VUKELIC, S.: *Entwurf eines generischen Objektmodells für das VLAN–Management und Integration in das INMS-Informationsmodell.* Diplomarbeit, Technische Universität München, November 1996.

[War] WARD, BRIAN: *Kernel HOWTO.* World Wide Web. http://sunsite.unc.edu/LDP/HOWTO/Kernel-HOWTO.html.

Anhang A

Patchfile für den Kernel


```
diff -u --recursive --new-file linux/include/net/br.h linux-vlan-2.1.59/include/net/br.h
--- linux/include/net/br.h Thu Mar 27 23:40:11 1997
+++ linux-vlan-2.1.59/include/net/br.h Sat Jul 25 10:44:41 1998
@@ -19,6 +19,8 @@
 #define Forwarding 3   /* (4 4 4)  */
 #define Blocking 4   /* (4.4.1)  */

+#define DEFAULT_VLAN_ID 0x01
+
 #define No_of_ports 8
 /* arbitrary choice, to allow the code below to compile */

@@ -131,6 +133,7 @@
  unsigned short   designated_port;   /* (4.5.5.7)  */
  unsigned int     top_change_ack;   /* (4.5.5.8)  */
  unsigned int     config_pending;   /* (4.5.5.9)  */
+ unsigned int     vlan_id;
  struct device *dev;
  struct fdb *fdb; /* head of per port fdb chain */
 } Port_data;
@@ -151,6 +154,9 @@
  unsigned int timer;
  unsigned int flags;
 #define FDB_ENT_VALID 0x01
+#define FDB_ENT_STATIC 0x11
+/* List of vlan's for each MAC address */
+ unsigned int vlan_id;
 /* AVL tree of all addresses, sorted by address */
  short fdb_avl_height;
  struct fdb *fdb_avl_left;
@@ -159,6 +165,24 @@
  struct fdb *fdb_next;
 };

+/* Types for MAC-address - VLAN_ID table */
+struct swdb {
+ unsigned char ula[6];
+ unsigned char pad[2];
+ unsigned short port;
+ unsigned int timer;
+ unsigned int flags;
+#define FDB_ENT_VALID 0x01
+/* List of vlan's for each MAC address */
+ unsigned int vlan_id;
+/* AVL tree of all addresses, sorted by address */
+ short swdb_avl_height;
+ struct swdb *swdb_avl_left;
+ struct swdb *swdb_avl_right;
+/* linked list of addresses for each port */
+ struct swdb *swdb_next;
+};
+
 #define IS_BRIDGED 0x2e


@@ -189,6 +213,7 @@
  unsigned int cmd;
  unsigned int arg1;
  unsigned int arg2;
+ unsigned char ula[6];
 };

 /* defined cmds */
@@ -207,6 +232,8 @@
 #define BRCMD_ENABLE_PROT_STATS 13
 #define BRCMD_DISABLE_PROT_STATS 14
 #define BRCMD_ZERO_PROT_STATS 15
+#define BRCMD_VLAN_CONFIG 16
+#define BRCMD_MAC_VLAN 17

 /* prototypes of exported bridging functions... */

@@ -219,8 +246,9 @@
 struct fdb *br_avl_find_addr(unsigned char addr[6]);
 int br_avl_insert (struct fdb * new_node);

+struct swdb *sw_avl_find_addr(unsigned char addr[6]);
+int sw_avl_insert (struct swdb * new_node);
+
 /* externs */
```

```
 extern struct br_stat br_stats;
-
-
diff -u --recursive --new-file linux/include/net/netlink.h linux-vlan-2.1.59/include/net/netlink.h
--- linux/include/net/netlink.h Thu Sep  4 22:25:28 1997
+++ linux-vlan-2.1.59/include/net/netlink.h Wed Jun 17 14:05:58 1998
@@ -57,6 +57,7 @@
 #define NETLINK_FIREWALL 3 /* Firewalling hook */
 #define NETLINK_FREE 4 /* PSI devices - 4 to 7 (obsolete) */
 #define NETLINK_ARPD 8 /* ARP daemon for big switched networks */
+#define NETLINK_BRD 9 /* Bridge daemon */
 #define NETLINK_IPSEC 10 /* IPSEC  (JI) */
 #define NETLINK_ROUTE6 11 /* Af_inet6 route communication channel */
 #define NETLINK_IP6_FW 13 /* IPv6 firewall trap outs */
diff -u --recursive --new-file linux/net/bridge/Makefile linux-vlan-2.1.59/net/bridge/Makefile
--- linux/net/bridge/Makefile Mon Apr  7 20:35:32 1997
+++ linux-vlan-2.1.59/net/bridge/Makefile Wed Jun 17 14:06:59 1998
@@ -8,7 +8,7 @@
 # Note 2! The CFLAGS definition is now in the main makefile...

 O_TARGET := bridge.o
-O_OBJS  := br.o br_tree.o
+O_OBJS  := br.o br_tree.o sw_tree.o
 M_OBJS   := $(O_TARGET)

 ifeq ($(CONFIG_SYSCTL),y)
diff -u --recursive --new-file linux/net/bridge/br.c linux-vlan-2.1.59/net/bridge/br.c
--- linux/net/bridge/br.c Sun Sep  7 23:00:24 1997
+++ linux-vlan-2.1.59/net/bridge/br.c Thu Aug 13 13:12:37 1998
@@ -26,7 +26,7 @@
  * Bridge SNMP stats.
  *
  */
-
+
 #include <linux/errno.h>
 #include <linux/types.h>
 #include <linux/socket.h>
@@ -46,6 +46,7 @@
 #include <asm/uaccess.h>
 #include <asm/system.h>
 #include <net/br.h>
+#include <net/netlink.h>

 static void transmit_config(int port_no);
 static int root_bridge(void);
@@ -77,6 +78,8 @@
 static void topology_change_timer_expiry(void);
 static void hold_timer_expiry(int port_no);
 static void br_init_port(int port_no);
+static void set_port_vlan_id(int port_no, unsigned int vlan_id);
+static int  set_mac_vlan_id(unsigned int vlan_id, unsigned char *ula);
 static void enable_port(int port_no);
 static void disable_port(int port_no);
 static void set_bridge_priority(bridge_id_t *new_bridge_id);
@@ -109,9 +112,13 @@
 static int send_tcn_bpdu(int port_no, Tcn_bpdu *bpdu);
 static int send_config_bpdu(int port_no, Config_bpdu *config_bpdu);
 static int find_port(struct device *dev);
+static int check_vlan_id(struct fdb *s, struct fdb *f);
 static int br_flood(struct sk_buff *skb, int port);
 static int br_drop(struct sk_buff *skb);
 static int br_learn(struct sk_buff *skb, int port); /* 3.8 */
+static int brd_send(unsigned short port, unsigned char *ula,
+ unsigned int timer, unsigned int flags, unsigned int vlan_id);
+static int brd_callback(int minor, struct sk_buff *skb);

 static unsigned char bridge_ula[ETH_ALEN] = { 0x01, 0x80, 0xc2, 0x00, 0x00, 0x00 };
 static Bridge_data     bridge_info;   /* (4.5.3)  */
@@ -128,6 +135,9 @@
 /* entries timeout after this many seconds */
 unsigned int fdb_aging_time = FDB_TIMEOUT;

+/* set if bridged wasn't started */
+static int brd_not_running;
+
 struct br_stat br_stats;

 static struct timer_list tl; /* for 1 second timer... */
@@ -729,6 +739,9 @@
  port_state_selection();   /* (4.8.1.5)  */
  config_bpdu_generation();   /* (4.8.1.6)  */

+ /* attach the netlink device */
+ netlink_attach(NETLINK_BRD, brd_callback);
+
  /* initialize system timer */
  tl.expires = jiffies+HZ; /* 1 second */
  tl.function = br_tick;
@@ -747,11 +760,66 @@
  set_port_state(port_no, Blocking);   /* (4.8.1.4.2)    */
  port_info[port_no].top_change_ack = 0;
  port_info[port_no].config_pending = FALSE;/* (4.8.1.4.4)  */
+ /* port belongs to default vlan */
+ set_port_vlan_id(port_no, DEFAULT_VLAN_ID);
+/* port_info[port_no].vlan_id = DEFAULT_VLAN_ID; */
  stop_message_age_timer(port_no);   /* (4.8.1.4.5)  */
  stop_forward_delay_timer(port_no);   /* (4.8.1.4.6)  */
  stop_hold_timer(port_no);   /* (4.8.1.4.7)  */
 }

+static void set_port_vlan_id(int port_no, unsigned int vlan_id)
+{
+ port_info[port_no].vlan_id = vlan_id;
+}
+
+static int set_mac_vlan_id(unsigned int vlan_id, unsigned char *ula)
+{
+ struct swdb *s, *m = NULL;
+
+ s = (struct swdb *)kmalloc(sizeof(struct swdb), GFP_ATOMIC);
+ if (!s) {
+ printk(KERN_DEBUG "br_learn: unable to malloc swdb\n");
+ return(-1);
+ }
+
+ m = sw_avl_find_addr(ula);
+ if (m) {
+ s->port = m->port;
+ }
+ else { /* unknown MAC-address */
+ printk(KERN_DEBUG "set_mac_vlan_id: unknown mac address\n");
+ return(-2);
+ }
+
+ printk("port %i vlan 0x%08x\
+ mac %02x:%02x:%02x:%02x:%02x:%02x\n",
+ m->port,
+ vlan_id,
+ ula[0],
+ ula[1],
+ ula[2],
+ ula[3],
+ ula[4],
+ ula[5]);
+
+ memcpy(s->ula, ula, 6); /* specified mac address */
```

```
+ s->timer = CURRENT_TIME;
+ s->flags = FDB_ENT_VALID;
+
+ s->vlan_id = vlan_id; /* specified vlan_id */
+
+ if (sw_avl_insert(s) == 0) { /* update */
+ kfree(s);
+ return(0);
+ }
+ else { /* unknown MAC-address */
+ printk(KERN_DEBUG "set_mac_vlan_id: unknown mac address\n");
+ kfree(s);
+ return(-2);
+ }
+}
+
 static void enable_port(int port_no)   /* (4.8.2)  */
 {
  br_init_port(port_no);
@@ -1317,6 +1385,11 @@
  skb->mac.raw = skb->h.raw = skb->data;
  eth = skb->mac.ethernet;
  port = 0; /* an impossible port */
+
+ /* set vlan_id for the host stack, if it isn't set yet */
+ if ( !port_info[port].vlan_id )
+ set_port_vlan_id(port, DEFAULT_VLAN_ID);
+
  if (br_stats.flags & BR_DEBUG)
  printk("br_tx_fr : port %i src %02x:%02x:%02x:%02x:%02x:%02x\
     dest %02x:%02x:%02x:%02x:%02x:%02x\n",
@@ -1344,7 +1417,9 @@

 static int br_learn(struct sk_buff *skb, int port) /* 3.8 */
 {
-struct fdb *f;
+ struct fdb *f,*n = NULL;
+ struct swdb *s,*m = NULL;
+ int status;

  switch (port_info[port].state) {
  case Listening:
@@ -1359,6 +1434,7 @@
  if (skb->mac.ethernet->h_source[0] & 0x01)
  return(-1);

+
  f = (struct fdb *)kmalloc(sizeof(struct fdb),
  GFP_ATOMIC);

@@ -1366,10 +1442,52 @@
  printk(KERN_DEBUG "br_learn: unable to malloc fdb\n");
  return(-1);
  }
+
  f->port = port; /* source port */
  memcpy(f->ula, skb->mac.ethernet->h_source, 6);
  f->timer = CURRENT_TIME;
  f->flags = FDB_ENT_VALID;
+
+ n = br_avl_find_addr(skb->mac.ethernet->h_source);
+ if (n) {
+ f->vlan_id = n->vlan_id;
+ }
+ else { /* set default vlan_id */
+ f->vlan_id = DEFAULT_VLAN_ID;
+ }
+
+                        s = (struct swdb *)kmalloc(sizeof(struct swdb),
+ GFP_ATOMIC);
+
+                        if (!s) {
+                                printk(KERN_DEBUG "br_learn: unable to malloc swdb\n");
+ return(-1);
+                        }
+
+                        s->port = port; /* source port */
+                        memcpy(s->ula, skb->mac.ethernet->h_source, 6);
+                        s->timer = CURRENT_TIME;
+                        s->flags = FDB_ENT_VALID;
+
+ m = sw_avl_find_addr(skb->mac.ethernet->h_source);
+ if (m) {
+ s->vlan_id = m->vlan_id;
+ }
+ else { /* set default vlan_id */
+ s->vlan_id = DEFAULT_VLAN_ID;
+ }
+
+ /*
+  * add entity to SW_AVL tree.  If entity already
+  * exists in the tree, update the fields with
+  * what we have here.
+     */
+ if (sw_avl_insert(s) == 0) { /* update */
+ kfree(s);
+ } /* new MAC-address */
+ else
+ printk(KERN_DEBUG "sw_avl_insert: new MAC\n");
+
   /*
    * add entity to AVL tree.  If entity already
    * exists in the tree, update the fields with
@@ -1379,6 +1497,21 @@
  kfree(f);
  return(0);
  }
+
+ status = (brd_send(f->port,
+ f->ula,
+ f->timer,
+ f->flags,
+ f->vlan_id));
+
+ if (status != 0) {
+ printk(KERN_DEBUG "br_learn: unable to contact bridge daemon, status: %d\n",status);
+ }
+
+ /* add to head of port chain */
+ s->swdb_next = port_info[port].fdb;
+ port_info[port].fdb = s;
+
   /* add to head of port chain */
   f->fdb_next = port_info[port].fdb;
   port_info[port].fdb = f;
@@ -1388,6 +1521,68 @@
 }

 /*
+ * netlink support
+ * post port and ula to netlink
+ */
+
+static int brd_callback(int minor, struct sk_buff *skb)
+{
+ struct fdb *retreq;
+
+ brd_not_running = 0;
```

```
+
+ if (skb->len != sizeof(struct fdb))
+ {
+ kfree_skb(skb, FREE_READ);
+ return -EINVAL;
+ }
+
+ retreq = (struct fdb *)skb->data;
+
+/*  start_bh_atomic();
+ * brd_send(retreq->port, retreq->ula, rtereq->timer, retreq->flags, retreq->vlan_id);
+ */ end_bh_atomic();
+
+ kfree_skb(skb, FREE_READ);
+ return sizeof(struct fdb);
+}
+
+static int brd_send(unsigned short port, unsigned char *ula,
unsigned int timer, unsigned int flags, unsigned int vlan_id)
+{
+ int retval;
+ struct sk_buff *skb;
+ struct fdb *brreq;
+
+ skb = alloc_skb(sizeof(struct fdb), GFP_ATOMIC);
+ if (skb == NULL)
+ return(-2);
+
+        brreq=(struct fdb *)skb_put(skb, sizeof(struct fdb));
+ brreq->port = port;
+        brreq->timer = timer;
+        brreq->flags = flags;
+        brreq->vlan_id = vlan_id;
+
+        if (ula)
+                memcpy(brreq->ula, ula, sizeof(brreq->ula));
+
+ retval = netlink_post(NETLINK_BRD, skb);
+
+ if (retval)
+ {
+ kfree_skb(skb, FREE_WRITE);
+ if (retval == -EUNATCH)
+ brd_not_running = 1;
+ }
+ else brd_not_running = 0;
+
+ if (brd_not_running)
+ return(-1);
+
+ return (0);
+}
+
+/*
  * this routine always consumes the frame
  */

@@ -1418,7 +1613,7 @@

 static int br_forward(struct sk_buff *skb, int port) /* 3.7 */
 {
-struct fdb *f;
+ struct fdb *f, *s;

  /*
      * flood all ports with frames destined for a group
@@ -1441,7 +1636,58 @@
  return(0);
  } else {

  /* locate port to forward to */
-f = br_avl_find_addr(skb->mac.ethernet->h_dest);
+ f = sw_avl_find_addr(skb->mac.ethernet->h_dest);
+
+ if ((f) && (br_stats.flags & BR_DEBUG)) {
+ printk("sw_avl_find_addr(f): port %i vlan 0x%08x\
+ dest-mac %02x:%02x:%02x:%02x:%02x:%02x\n",
+ f->port,
+ f->vlan_id,
+ f->ula[0],
+ f->ula[1],
+ f->ula[2],
+ f->ula[3],
+ f->ula[4],
+ f->ula[5]);
+ }
+
+ /* locate source port */
+ s = sw_avl_find_addr(skb->mac.ethernet->h_source);
+
+ if (br_stats.flags & BR_DEBUG) {
+ if (s) {
+ printk("sw_avl_find_addr(s): port %i vlan 0x%08x\
+ source-mac %02x:%02x:%02x:%02x:%02x:%02x\n",
+ s->port,
+ s->vlan_id,
+         s->ula[0],
+ s->ula[1],
+ s->ula[2],
+ s->ula[3],
+ s->ula[4],
+ s->ula[5]);
+ }
+ else printk("sw_avl_find_addr(s): not found\n");
+ }
+
+ /*
+  * frame received from the host stack;
+  * no source MAC in avl tree ->
+  * bridge itself is in all VLAN's
+  */
+ if (!s && port==0) {
+ if (br_stats.flags & BR_DEBUG)
+ printk("frame received from the host stack\n");
+ s=f;
+ }
+
+ /* source MAC not found */
+ else if (!s) {
+ if (br_stats.flags & BR_DEBUG)
+ printk("source MAC not known\n");
+ return(br_dev_drop(skb));
+ }
+
  /*
   * Send flood and drop.
   */
@@ -1452,10 +1698,20 @@
  }
  /*
   * Sending
+  *      only if: source-port != destination-port
+  *               port.state == Forwarding
+  *               source.address.vlan_id == dest.address.vlan_id
+  *               source.port.vlan_id == dest.port.vlan_id
   */
-if (f->port!=port && port_info[f->port].state == Forwarding) {
+ if (f->port!=port && port_info[f->port].state == Forwarding
+ &&
```

```
+  check_vlan_id(s, f)
+ &&
+  ((port_info[f->port].vlan_id & port_info[port].vlan_id) != 0) )
+ {
  /* has entry expired? */
-if (f->timer + fdb_aging_time < CURRENT_TIME) {
+ if ( (f->timer + fdb_aging_time < CURRENT_TIME )
+   && (f->flags != FDB_ENT_STATIC) ) {
  /* timer expired, invalidate entry */
  f->flags &= ~FDB_ENT_VALID;
  if (br_stats.flags & BR_DEBUG)
@@ -1482,6 +1738,8 @@
   * We send this still locked
   */
  skb->priority = 1;
+ if (br_stats.flags & BR_DEBUG)
+ printk("forward on known port\n");
  dev_queue_xmit(skb);
  return(1); /* skb has been consumed */
  } else {
@@ -1493,6 +1751,44 @@
  }
 }

+
+/*
+ * This routine compares the vlan_id of the source and the destination
+ * MAC address. It returns true, if they belong to the same VLAN,
+ * false, if not.
+ */
+
+static int check_vlan_id(struct fdb *s, struct fdb *f)
+{
+ if (br_stats.flags & BR_DEBUG) {
+ printk("port %i vlan 0x%08x\
+ source-mac %02x:%02x:%02x:%02x:%02x:%02x\n",
+ s->port,
+ s->vlan_id,
+ s->ula[0],
+ s->ula[1],
+ s->ula[2],
+ s->ula[3],
+ s->ula[4],
+ s->ula[5]);
+
+ printk("port %i vlan 0x%08x\
+ dest-mac %02x:%02x:%02x:%02x:%02x:%02x\n",
+ f->port,
+ f->vlan_id,
+ f->ula[0],
+ f->ula[1],
+ f->ula[2],
+ f->ula[3],
+ f->ula[4],
+ f->ula[5]);
+ }
+ if ((s->vlan_id & f->vlan_id) == 0)
+ return(FALSE);
+ return (TRUE);
+}
+
+
 /*
  * this routine sends a copy of the frame to all forwarding ports
  * with the exception of the port given.  This routine never
@@ -1508,8 +1804,15 @@
  {
  if (i == port)
  continue;
-if (port_info[i].state == Forwarding)
-{
+
+ /*
+  * forward only, if port state is forwarding
+  * and source.port.vlan_id == dest.port.vlan_id
+  */
+ if ((port_info[i].state == Forwarding)
+     &&
+   ((port_info[i].vlan_id & port_info[port].vlan_id) != 0))
+   {
  nskb = skb_clone(skb, GFP_ATOMIC);
  if(nskb==NULL)
  continue;
@@ -1639,6 +1942,19 @@
  printk(KERN_DEBUG "br: disabling port %i\n",bcf.arg1);
  disable_port(bcf.arg1);
  break;
+
+ /*
+  * configure vlan_id of specified port
+  */
+ case BRCMD_VLAN_CONFIG:
+ printk(KERN_DEBUG "br: configuring vlan_id %i for port %i\n",bcf.arg2,bcf.arg1);
+ set_port_vlan_id(bcf.arg1, bcf.arg2);
+ break;
+
+ case BRCMD_MAC_VLAN:
+ set_mac_vlan_id(bcf.arg1, &bcf.ula[0]);
+ break;
+
  case BRCMD_SET_BRIDGE_PRIORITY:
  set_bridge_priority((bridge_id_t *)&bcf.arg1);
  break;
diff -u --recursive --new-file linux/net/bridge/br_tree.c linux-vlan-2.1.59/net/bridge/br_tree.c
--- linux/net/bridge/br_tree.c Thu Mar 27 23:40:14 1997
+++ linux-vlan-2.1.59/net/bridge/br_tree.c Wed Jun 17 14:08:02 1998
@@ -217,6 +217,7 @@
  if (addr_cmp(new_node->ula, node->ula) == 0) { /* update */
  node->flags = new_node->flags;
  node->timer = new_node->timer;
+ node->vlan_id = new_node->vlan_id;
  return(0);
  }
  if (addr_cmp(new_node->ula, node->ula) < 0) {
diff -u --recursive --new-file linux/net/bridge/sw_tree.c linux-vlan-2.1.59/net/bridge/sw_tree.c
--- linux/net/bridge/sw_tree.c
+++ linux-vlan-2.1.59/net/bridge/sw_tree.c Wed Jul 22 13:06:56 1998
@@ -0,0 +1,404 @@
+/*
+ * this code is derived from the avl functions in mmap.c
+ */
+#include <linux/kernel.h>
+#include <linux/errno.h>
+#include <linux/string.h>
+#include <linux/malloc.h>
+#include <linux/skbuff.h>
+
+#include <net/br.h>
+#define _SW_DEBUG_AVL
+
+/*
+ * Use an AVL (Adelson-Velskii and Landis) tree to speed up this search
+ * from O(n) to O(log n), where n is the number of ULAs.
+ * Written by Bruno Haible <haible@ma2s2.mathematik.uni-karlsruhe.de>.
+ * Taken from mmap.c, extensively modified by John Hayes
+ * <hayes@netplumbing.com>
+ */
+
```

```
+static struct swdb sw_fdb_head;
+static struct swdb *sw_fhp = &sw_fdb_head;
+static struct swdb **sw_fhpp = &sw_fhp;
+static int sw_fdb_inited = 0;
+
+static int sw_addr_cmp(unsigned char *a1, unsigned char *a2);
+
+/*
+ * sw_fdb_head is the AVL tree corresponding to swdb
+ * or, more exactly, its root.
+ * A swdb has the following fields:
+ *   swdb_avl_left     left son of a tree node
+ *   swdb_avl_right    right son of a tree node
+ *   swdb_avl_height   1+max(heightof(left),heightof(right))
+ * The empty tree is represented as NULL.
+ */
+
+#ifndef avl_sw_empty
+#define avl_sw_empty (struct swdb *) NULL
+#endif
+
+/* Since the trees are balanced, their height will never be large. */
+#define sw_avl_maxheight 127
+#define heightof(tree) ((tree) == avl_sw_empty ? 0 : (tree)->swdb_avl_height)
+/*
+ * Consistency and balancing rules:
+ * 1. tree->swdb_avl_height == 1+max(heightof(tree->swdb_avl_left),heightof(tree->swdb_avl_right))
+ * 2. abs( heightof(tree->swdb_avl_left) - heightof(tree->swdb_avl_right) ) <= 1
+ * 3. foreach node in tree->swdb_avl_left: node->swdb_avl_key <= tree->swdb_avl_key,
+ *    foreach node in tree->swdb_avl_right: node->swdb_avl_key >= tree->swdb_avl_key.
+ */
+
+static int
+sw_fdb_init(void)
+{
+ sw_fdb_head.swdb_avl_height = 0;
+ sw_fdb_head.swdb_avl_left = (struct swdb *)0;
+ sw_fdb_head.swdb_avl_right = (struct swdb *)0;
+ sw_fdb_inited = 1;
+ return(0);
+}
+
+struct swdb *sw_avl_find_addr(unsigned char addr[6])
+{
+ struct swdb * result = NULL;
+ struct swdb * tree;
+
+ if (!sw_fdb_inited)
+ sw_fdb_init();
+#if (SW_DEBUG_AVL)
+ printk("searching for ula %02x:%02x:%02x:%02x:%02x:%02x\n",
+ addr[0],
+ addr[1],
+ addr[2],
+ addr[3],
+ addr[4],
+ addr[5]);
+#endif /* SW_DEBUG_AVL */
+ for (tree = &sw_fdb_head ; ; ) {
+ if (tree == avl_sw_empty) {
+#if (SW_DEBUG_AVL)
+ printk("search failed, returning node 0x%x\n", (unsigned int)result);
+#endif /* SW_DEBUG_AVL */
+ return result;
+ }
+
+#if (SW_DEBUG_AVL)
+ printk("node 0x%x: checking ula %02x:%02x:%02x:%02x:%02x:%02x\n",
+ (unsigned int)tree,
+ tree->ula[0],
+ tree->ula[1],
+ tree->ula[2],
+ tree->ula[3],
+ tree->ula[4],
+ tree->ula[5]);
+#endif /* SW_DEBUG_AVL */
+ if (sw_addr_cmp(addr, tree->ula) == 0) {
+#if (SW_DEBUG_AVL)
+ printk("found node 0x%x\n", (unsigned int)tree);
+#endif /* SW_DEBUG_AVL */
+ return tree;
+ }
+ if (sw_addr_cmp(addr, tree->ula) < 0) {
+ tree = tree->swdb_avl_left;
+ } else {
+ tree = tree->swdb_avl_right;
+ }
+ }
+}
+
+
+#if (0)
+/*
+ * Rebalance a tree.
+ * After inserting or deleting a node of a tree we have a sequence of subtrees
+ * nodes[0]..nodes[k-1] such that
+ * nodes[0] is the root and nodes[i+1] = nodes[i]->{swdb_avl_left|swdb_avl_right}.
+ */
+static void sw_avl_rebalance (struct swdb *** nodeplaces_ptr, int count)
+{
+ if (!sw_fdb_inited)
+ sw_fdb_init();
+ for ( ; count > 0 ; count--) {
+ struct swdb ** nodeplace = *--nodeplaces_ptr;
+ struct swdb * node = *nodeplace;
+ struct swdb * nodeleft = node->swdb_avl_left;
+ struct swdb * noderight = node->swdb_avl_right;
+ int heightleft = heightof(nodeleft);
+ int heightright = heightof(noderight);
+ if (heightright + 1 < heightleft) {
+ /*                                                      */
+ /*                           *                          */
+ /*                          /   \                       */
+ /*                       n+2     n                      */
+ /*                                                      */
+ struct swdb * nodeleftleft = nodeleft->swdb_avl_left;
+ struct swdb * nodeleftright = nodeleft->swdb_avl_right;
+ int heightleftright = heightof(nodeleftright);
+ if (heightof(nodeleftleft) >= heightleftright) {
+ /*                                                      */
+ /*               *                  n+2|n+3            */
+ /*             /   \                /    \              */
+ /*          n+2     n      -->      /   n+1|n+2         */
+ /*          / \                     |    /    \         */
+ /*        n+1 n|n+1                 n+1  n|n+1  n       */
+ /*                                                      */
+ node->swdb_avl_left = nodeleftright;
+ nodeleft->swdb_avl_right = node;
+ nodeleft->swdb_avl_height = 1 + (node->swdb_avl_height = 1 + heightleftright);
+ *nodeplace = nodeleft;
+ } else {
+ /*                                                      */
+ /*               *                    n+2              */
+ /*             /   \                /     \            */
+ /*          n+2     n      -->    n+1     n+1          */
+ /*          / \                   / \     / \          */
+ /*         n  n+1                n   L   R   n         */
+ /*           / \                                        */
```

```
+ /*           L   R                                        */
+ /*                                                        */
+ nodeleft->swdb_avl_right = nodeleftright->swdb_avl_left;
+ node->swdb_avl_left = nodeleftright->swdb_avl_right;
+ nodeleftright->swdb_avl_left = nodeleft;
+ nodeleftright->swdb_avl_right = node;
+ nodeleft->swdb_avl_height = node->swdb_avl_height = heightleftright;
+ nodeleftright->swdb_avl_height = heightleft;
+ *nodeplace = nodeleftright;
+ }
+ } else if (heightleft + 1 < heightright) {
+ /* similar to the above, just interchange 'left' <--> 'right' */
+ struct swdb * noderightright = noderight->swdb_avl_right;
+ struct swdb * noderightleft = noderight->swdb_avl_left;
+ int heightrightleft = heightof(noderightleft);
+ if (heightof(noderightright) >= heightrightleft) {
+ node->swdb_avl_right = noderightleft;
+ noderight->swdb_avl_left = node;
+ noderight->swdb_avl_height = 1 + (node->swdb_avl_height = 1 + heightrightleft);
+ *nodeplace = noderight;
+ } else {
+ noderight->swdb_avl_left = noderightleft->swdb_avl_right;
+ node->swdb_avl_right = noderightleft->swdb_avl_left;
+ noderightleft->swdb_avl_right = noderight;
+ noderightleft->swdb_avl_left = node;
+ noderight->swdb_avl_height = node->swdb_avl_height = heightrightleft;
+ noderightleft->swdb_avl_height = heightright;
+ *nodeplace = noderightleft;
+ }
+ } else {
+ int height = (heightleft<heightright ? heightright : heightleft) + 1;
+ if (height == node->swdb_avl_height)
+ break;
+ node->swdb_avl_height = height;
+ }
+ }
+#ifdef SW_DEBUG_AVL
+ printk_sw_avl(&sw_fdb_head);
+#endif /* SW_DEBUG_AVL */
+}
+#endif /* (0) */
+
+/* Insert a node into a tree. */
+int sw_avl_insert (struct swdb * new_node)
+{
+ struct swdb ** nodeplace = sw_fhpp;
+ struct swdb ** stack[sw_avl_maxheight];
+ int stack_count = 0;
+ struct swdb *** stack_ptr = &stack[0]; /* = &stack[stackcount] */
+ if (!sw_fdb_inited)
+ sw_fdb_init();
+ for (;;) {
+ struct swdb *node;
+
+ node = *nodeplace;
+ if (node == avl_sw_empty)
+ break;
+ *stack_ptr++ = nodeplace; stack_count++;
+ if (sw_addr_cmp(new_node->ula, node->ula) == 0) { /* update */
+ node->flags = new_node->flags;
+ node->timer = new_node->timer;
+ node->vlan_id = new_node->vlan_id;
+ return(0);
+ }
+ if (sw_addr_cmp(new_node->ula, node->ula) < 0) {
+ nodeplace = &node->swdb_avl_left;
+ } else {
+ nodeplace = &node->swdb_avl_right;
+ }
+ }
+#if (SW_DEBUG_AVL)
+ printk("node 0x%x: adding ula %02x:%02x:%02x:%02x:%02x:%02x\n",
+ (unsigned int)new_node,
+ new_node->ula[0],
+ new_node->ula[1],
+ new_node->ula[2],
+ new_node->ula[3],
+ new_node->ula[4],
+ new_node->ula[5]);
+#endif /* (SW_DEBUG_AVL) */
+ new_node->swdb_avl_left = avl_sw_empty;
+ new_node->swdb_avl_right = avl_sw_empty;
+ new_node->swdb_avl_height = 1;
+ *nodeplace = new_node;
+#if (0)
+ sw_avl_rebalance(stack_ptr,stack_count);
+#endif /* (0) */
+#ifdef SW_DEBUG_AVL
+ printk_sw_avl(&sw_fdb_head);
+#endif /* SW_DEBUG_AVL */
+ return(1);
+}
+
+
+#if (0)
+/* Removes a node out of a tree. */
+static int sw_avl_remove (struct swdb * node_to_delete)
+{
+ struct swdb ** nodeplace = sw_fhpp;
+ struct swdb ** stack[sw_avl_maxheight];
+ int stack_count = 0;
+ struct swdb *** stack_ptr = &stack[0]; /* = &stack[stackcount] */
+ struct swdb ** nodeplace_to_delete;
+ if (!sw_fdb_inited)
+ sw_fdb_init();
+ for (;;) {
+ struct swdb * node = *nodeplace;
+ if (node == avl_sw_empty) {
+ /* what? node_to_delete not found in tree? */
+ printk(KERN_ERR "sw: avl_remove: node to delete not found in tree\n");
+ return(-1);
+ }
+ *stack_ptr++ = nodeplace; stack_count++;
+ if (sw_addr_cmp(node_to_delete->ula, node->ula) == 0)
+ break;
+ if (sw_addr_cmp(node_to_delete->ula, node->ula) < 0)
+ nodeplace = &node->swdb_avl_left;
+ else
+ nodeplace = &node->swdb_avl_right;
+ }
+ nodeplace_to_delete = nodeplace;
+ /* Have to remove node_to_delete = *nodeplace_to_delete. */
+ if (node_to_delete->swdb_avl_left == avl_sw_empty) {
+ *nodeplace_to_delete = node_to_delete->swdb_avl_right;
+ stack_ptr--; stack_count--;
+ } else {
+ struct swdb *** stack_ptr_to_delete = stack_ptr;
+ struct swdb ** nodeplace = &node_to_delete->swdb_avl_left;
+ struct swdb * node;
+ for (;;) {
+ node = *nodeplace;
+ if (node->swdb_avl_right == avl_sw_empty)
+ break;
+ *stack_ptr++ = nodeplace; stack_count++;
+ nodeplace = &node->swdb_avl_right;
+ }
+ *nodeplace = node->swdb_avl_left;
+ /* node replaces node_to_delete */
```

```
+ node->swdb_avl_left = node_to_delete->swdb_avl_left;
+ node->swdb_avl_right = node_to_delete->swdb_avl_right;
+ node->swdb_avl_height = node_to_delete->swdb_avl_height;
+ *nodeplace_to_delete = node; /* replace node_to_delete */
+ *stack_ptr_to_delete = &node->swdb_avl_left; /* replace &node_to_delete->swdb_avl_left */
+ }
+ sw_avl_rebalance(stack_ptr,stack_count);
+ return(0);
+}
+#endif /* (0) */
+
+#ifdef SW_DEBUG_AVL
+
+/* print a tree */
+static void printk_sw_avl (struct swdb * tree)
+{
+ if (tree != avl_sw_empty) {
+ printk("(");
+ printk("%02x:%02x:%02x:%02x:%02x:%02x",
+ tree->ula[0],
+ tree->ula[1],
+ tree->ula[2],
+ tree->ula[3],
+ tree->ula[4],
+ tree->ula[5]);
+ if (tree->swdb_avl_left != avl_sw_empty) {
+ printk_sw_avl(tree->swdb_avl_left);
+ printk("<");
+ }
+ if (tree->swdb_avl_right != avl_sw_empty) {
+ printk(">");
+ printk_sw_avl(tree->swdb_avl_right);
+ }
+ printk(")\n");
+ }
+}
+
+#if (0)
+static char *sw_avl_check_point = "somewhere";
+
+/* check a tree's consistency and balancing */
+static void sw_avl_checkheights (struct swdb * tree)
+{
+ int h, hl, hr;
+
+ if (tree == avl_sw_empty)
+ return;
+ sw_avl_checkheights(tree->swdb_avl_left);
+ sw_avl_checkheights(tree->swdb_avl_right);
+ h = tree->swdb_avl_height;
+ hl = heightof(tree->swdb_avl_left);
+ hr = heightof(tree->swdb_avl_right);
+ if ((h == hl+1) && (hr <= hl) && (hl <= hr+1))
+ return;
+ if ((h == hr+1) && (hl <= hr) && (hr <= hl+1))
+ return;
+ printk("%s: sw_avl_checkheights: heights inconsistent\n",sw_avl_check_point);
+}
+
+/* check that all values stored in a tree are < key */
+static void sw_avl_checkleft (struct swdb * tree, swdb_avl_key_t key)
+{
+ if (tree == avl_sw_empty)
+ return;
+ sw_avl_checkleft(tree->swdb_avl_left,key);
+ sw_avl_checkleft(tree->swdb_avl_right,key);
+ if (tree->swdb_avl_key < key)
+ return;
+ printk("%s: sw_avl_checkleft: left key %lu >= top key %lu\n",sw_avl_check_point,tree->swdb_avl_key,key);
+}
+
+/* check that all values stored in a tree are > key */
+static void sw_avl_checkright (struct swdb * tree, swdb_avl_key_t key)
+{
+ if (tree == avl_sw_empty)
+ return;
+ sw_avl_checkright(tree->swdb_avl_left,key);
+ sw_avl_checkright(tree->swdb_avl_right,key);
+ if (tree->swdb_avl_key > key)
+ return;
+ printk("%s: sw_avl_checkright: right key %lu <= top key %lu\n",sw_avl_check_point,tree->swdb_avl_key,key);
+}
+
+/* check that all values are properly increasing */
+static void sw_avl_checkorder (struct swdb * tree)
+{
+ if (tree == avl_sw_empty)
+ return;
+ sw_avl_checkorder(tree->swdb_avl_left);
+ sw_avl_checkorder(tree->swdb_avl_right);
+ sw_avl_checkleft(tree->swdb_avl_left,tree->swdb_avl_key);
+ sw_avl_checkright(tree->swdb_avl_right,tree->swdb_avl_key);
+}
+
+#endif /* (0) */
+#endif /* SW_DEBUG_AVL */
+
+static int sw_addr_cmp(unsigned char a1[], unsigned char a2[])
+{
+ int i;
+
+ for (i=0; i<6; i++) {
+ if (a1[i] > a2[i]) return(1);
+ if (a1[i] < a2[i]) return(-1);
+ }
+ return(0);
+}
+
```

Anhang B

Patchfile für `brcfg.c`


```
--- brcfg.c.DIST Thu Apr  9 09:27:20 1998
+++ brcfg.c Thu Jul 30 14:02:08 1998
@@ -13,8 +13,8 @@
 #include <sys/socket.h>
 #include <linux/skbuff.h>
 #include <linux/sockios.h>
-/*#include <net/br.h>*/
-#include "/usr/src/linux/include/net/br.h"
+#include <net/br.h>
+/*#include "/usr/src/linux/include/net/br.h"*/
 /*#include "br.h"*/


@@ -109,6 +109,7 @@
  printf("BLOCKING (0x%x)\n",ports[i].state);
  break;
  }
+ printf("\tvlan id 0x%08x\n",ports[i].vlan_id);
  printf("designated root ");
  disp_id(&ports[i].designated_root);
  printf("\n");
@@ -198,12 +199,16 @@
  printf("\n");
 }

-void cmd(int cmd,int arg1,int arg2)
+void cmd(int cmd,int arg1,int arg2,char *ula)
 { struct br_cf bcf;

+ char *pt;
+ pt = &bcf.ula[0];
+
  bcf.cmd = cmd;
  bcf.arg1 = arg1;
  bcf.arg2 = arg2;
+ memcpy(pt, ula, 12);
  if (ioctl(fd, SIOCSIFBR, &bcf) < 0) {
  perror("ioctl(SIOCSIFBR) failed");
  exit(1);
@@ -218,6 +223,8 @@
 "brcfg p[ort] x e[nable] Enable a port\n"
 "brcfg p[ort] x d[isable] Disable a port\n"
 "brcfg p[ort] x p[riority] y Set the priority of a port\n"
+"brcfg p[ort] x v[lan] y Set the vlan_id of a port\n"
+"brcfg m[ac] x v[lan] y Set the vlan_id of a mac address\n"
 "brcfg pr[iority] y Set bridge priority\n"
 "brcfg pa[thcost] y Set the pathcosts\n"
 "brcfg d[ebug] on Switch debugging on\n"
@@ -225,10 +232,10 @@
 "brcfg pol[icy] r[eject]/a[ccept] Switch the policy/flush protocol list\n"
 "brcfg e[xempt] <protocol> .. Set list of exempt protocols\n"
 "brcfg l[ist] List available protocols\n"
-"brcfg stat[s] z[ero] Reset Statistics counters"
-"brcfg stat[s] d[isable] Switch protocol statistics off"
-"brcfg stat[s] e[nable] Switch keeping protocol statistics on"
-"brcfg stat[s] s[how] Show protocol statistics"
+"brcfg stat[s] z[ero] Reset Statistics counters\n"
+"brcfg stat[s] d[isable] Switch protocol statistics off\n"
+"brcfg stat[s] e[nable] Switch keeping protocol statistics on\n"
+"brcfg stat[s] s[how] Show protocol statistics\n"
 "brcfg statu[s] Show bridge status\n"
 "\n"
 "Examples:\n"
@@ -240,11 +247,11 @@

 void debug(char *option)
 { if (strcasecmp(option,"on")==0)
-  { cmd(BRCMD_ENABLE_DEBUG,0,0);
+  { cmd(BRCMD_ENABLE_DEBUG,0,0,"");
     printf("Debug on.\n");
   } else
   if (strcasecmp(option,"off")==0)
-  { cmd(BRCMD_DISABLE_DEBUG,0,0);
+  { cmd(BRCMD_DISABLE_DEBUG,0,0,"");
     printf("Debug off.\n");
   } else
   { printf("Debug mode can only be on or off!");
@@ -259,7 +266,7 @@
```

```
   for(p=protocols;p->nr && strcasecmp(protocol,p->name)!=0;p++) ;
   if (p->nr)
-  { cmd(BRCMD_EXEMPT_PROTOCOL,p->nr,0);
+  { cmd(BRCMD_EXEMPT_PROTOCOL,p->nr,0,"");
     printf("Exempt protocol ");disp_protocol(p->nr);printf(".\n");
     return;
   }
@@ -275,11 +282,11 @@

 void policy(char *kind)
 { if (kind[0]=='a')
-  { cmd(BRCMD_SET_POLICY,1,0);
+  { cmd(BRCMD_SET_POLICY,1,0,"");
     printf("Policy accept all protocols.\n");
   } else
   if (kind[0]=='r')
-  { cmd(BRCMD_SET_POLICY,0,0);
+  { cmd(BRCMD_SET_POLICY,0,0,"");
     printf("Policy reject all protocols.\n");
   } else
   { printf("Policy must be either accept or deny.\n");
@@ -289,18 +296,18 @@

 void pathcost(char *port,char *cost)
 { int p=atoi(port),c=atoi(cost);
-  cmd(BRCMD_SET_PATH_COST,p,c);
+  cmd(BRCMD_SET_PATH_COST,p,c,"");
   printf("Pathcost for port %d set to %d",p,c);
 }

 void port(char *no,char *mode)
 { int port=atoi(no);
   if (mode[0]=='e')
-  { cmd(BRCMD_PORT_ENABLE,port,0);
+  { cmd(BRCMD_PORT_ENABLE,port,0,"");
     printf("Enable Port %d\n",port);
   } else
   if (mode[0]=='d')
-  { cmd(BRCMD_PORT_DISABLE,port,0);
+  { cmd(BRCMD_PORT_DISABLE,port,0,"");
     printf("Disable Port %d\n",port);
   } else
   { printf("Port option can only be enable or disable");
@@ -309,13 +316,40 @@

 void portprior(char *port,char *prio)
 { int po=atoi(port),pr=atoi(prio);
-  cmd(BRCMD_SET_PORT_PRIORITY,po,pr);
+  cmd(BRCMD_SET_PORT_PRIORITY,po,pr,"");
   printf("Port Priority for port %d set to %d\n",po,pr);
 }

+void create_vlan(char *port,char *vlan)
+{ int po=atoi(port),id=atoi(vlan);
+  cmd(BRCMD_VLAN_CONFIG,po,id,"");
+  printf("Vlan_id for port %d set to 0x%08x\n",po,id);
+}
+
+void create_mac_vlan(char *mac,char *vlan)
+{ unsigned int i,j;
+  unsigned char ula[6];
+  int id=atoi(vlan);
+
+  for ( i=0; i<6; i++) {
+ j = i;
+ sscanf( &mac[j*2], "%02x", &ula[j]);
+  }
+  printf("Vlan_id for mac %02x:%02x:%02x:%02x:%02x:%02x set to 0x%08x\n",
+ ula[0],
+ ula[1],
+ ula[2],
+ ula[3],
+ ula[4],
+ ula[5],
+ id);
+
+  cmd(BRCMD_MAC_VLAN,id,0,ula);
+}
+
 void priority(char *prio)
 { int p=atoi(prio);
-  cmd(BRCMD_SET_BRIDGE_PRIORITY,p,0);
+  cmd(BRCMD_SET_BRIDGE_PRIORITY,p,0,"");
   printf("Bridge priority set to %d\n",p);
 }

@@ -323,15 +357,15 @@
 { int x;
   switch (a[0]) {
   case 'e' :
-cmd(BRCMD_ENABLE_PROT_STATS,0,0);
+ cmd(BRCMD_ENABLE_PROT_STATS,0,0,"");
   printf("Protocol Statistics enabled\n");
   break;
   case 'd' :
-cmd(BRCMD_DISABLE_PROT_STATS,0,0);
+ cmd(BRCMD_DISABLE_PROT_STATS,0,0,"");
   printf("Protocol Statistics disabled\n");
   break;
   case 'z' :
-cmd(BRCMD_ZERO_PROT_STATS,0,0);
+ cmd(BRCMD_ZERO_PROT_STATS,0,0,"");
   printf("Protocol Statistics counters reset.\n");
   break;
   case 's' :
@@ -379,7 +413,7 @@
           argv--;
   break;
   case 'p' : if (p[2]=='l') { policy(*++argv);break; }
-else if (p[2]=='a') { pathcost(*++argv,*++argv);break; }
+ else if (p[1]=='a') { pathcost(*++argv,*++argv);break; }
   else if (p[1]=='r') { priority(*++argv);break; }
   else
   if (argv[2][0]=='p')
@@ -387,15 +421,26 @@
     break;
   }
   else
+ if (argv[2][0]=='v')
+ { create_vlan(argv[1],argv[3]);argv+=3;
+   break;
+ }
+ else
   {  port(*++argv,*++argv);break;
   }
+ case 'm' : if (argv[2][0]=='v')
+ {
+ create_mac_vlan(argv[1],argv[3]);argv+=3;
+
+ break;
+ }
   case 's' : if (p[3]=='r') {
-cmd(BRCMD_BRIDGE_ENABLE,0,0);
+ cmd(BRCMD_BRIDGE_ENABLE,0,0,"");
   printf("Bridge started.\n");
   break;
   }
```

```
  else if (p[2]=='o') {
-cmd(BRCMD_BRIDGE_DISABLE,0,0);
+ cmd(BRCMD_BRIDGE_DISABLE,0,0,"");
printf("Bridge stopped.\n");
break;
}
```

Anhang C

Patchfile für `brd.c`


```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
#include <fcntl.h>
#include <string.h>
#include <limits.h>
#include <sys/types.h>
#include <errno.h>
#include <net/br.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

static void brd_print(char*, struct fdb*);

int main(int argc, char **argv)
{
int status;
int fd;
struct fdb req;

openlog ("brd", LOG_PID | LOG_CONS, LOG_DAEMON);

syslog(LOG_NOTICE, "Initializing, version %s\n", BRD_VERSION);

fd = open("/dev/brd", O_RDWR);
if (fd < 0) {
syslog(LOG_CRIT, "cannot open /dev/brd: %m");
exit(-1);
}

while (1) {
status = read(fd, &req, sizeof(req));
if (status < 0) {
if (errno == EINTR)
continue;
syslog(LOG_CRIT, "cannot read from /dev/brd: %m");
exit(-1);
}
if (status != sizeof(req))
{
syslog(LOG_CRIT, "read from /dev/brd returns %d",
 status);
exit(-1);
}
brd_print("UPDATE", &req);
}

return 0;
}

static void brd_print(char * type, struct fdb* request)
{
unsigned short port=request->port;
unsigned char *ula=request->ula;
unsigned int timer=request->timer;
unsigned int flags=request->flags;
unsigned int vlan_id=request->vlan_id;


fprintf(stderr, "Type: UPDATE,\n");
fprintf(stderr, "Port: %i,\n", port);
fprintf(stderr, "Mac:  %02x:%02x:%02x:%02x:%02x:%02x,\n",
ula[0], ula[1], ula[2], ula[3], ula[4], ula[5]);
fprintf(stderr, "v_id: %d,\n", vlan_id);
fprintf(stderr, "Flag: %d,\n", flags);
fprintf(stderr, "Updt: %d;\n\n", timer);
}
```

Anhang D

Makefile für `brd.c`


```
# brd makefile, Peter Allgeyer, 15.08.1998

VERSION = 0.0.1
KERNELVERSION = linux-2.0.0

# DEBUG = -DDEBUG

CC = gcc
CFLAGS = -m486 -O6 -pipe -fomit-frame-pointer -Wall \
-DBRD_VERSION=\"$(VERSION)\" $(DEBUG)

all: brd

clean:
rm -f brd

package: clean
-(cd /usr/src; \
 for file in \
include/linux/if_arp.h \
include/linux/kerneld.h \
include/linux/interrupt.h \
net/ipv4/arp.c \
net/ipv4/Config.in \
Documentation/Configure.help \
MAINTAINERS \
; do \
diff -u -d $(KERNELVERSION).orig/$$file linux/$$file;\
 done) > brd-$(VERSION).kernel.patch
cd ..; tar -czf brd/brd-$(VERSION).tar.gz brd
```