TAMPEREEN TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto

Implementation of LAN Emulation Over ATM in Linux

Kiiskilä Marko

Diplomityö

Aihe hyväksytty osastoneuvoston
kokouksessa 21.8.1996
Tarkastajat: Prof. Jarmo Harju ja
Dos. Juha Heinänen

Foreword

This Master of Science thesis was done at the Tampere University of Technology as a part of the Finnish Broadband Telecommunication Research Project, FASTER, which also funded the research. The thesis was written under the supervision of Prof. Jarmo Harju from the Telecommunications Laboratory at TUT and Dr. Juha Heinänen from Telecom Finland. I would like to thank them for advice and support.

Many thanks also go to Prof. Markku Renfors, MSc Mika Grundstöm, MSc Werner Almesberger at the Swiss Federal Institute of Technology in Lausanne, and my fellow workers Mika and Otto.

Tampere, October 14, 1996

Marko Kiiskilä

Mekaniikanpolku 6A15
33720 Tampere
Finland

Tel. +358 40 512 8776

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology
Telecommunications Laboratory
KIISKILÄ MARKO: Implementation of LAN Emulation Over ATM in Linux
Master's Thesis, 49 pages
Auditors: Prof. Jarmo Harju and Dr. Juha Heinänen

October 1996
Keywords: LAN Emulation, LANE, asynchronous transfer mode, ATM, Linux

This thesis describes the implementation of LAN Emulation over ATM in Linux operating system. This includes both the LE client and the LE service as described in ATM Forum's specification LAN Emulation Over ATM - Version 1.0. The work was done as a part of Finnish Broadband Telecommunication Research Project (FASTER) at Tampere University of Technology.

LAN Emulation is intended to emulate the services of existing Local Area Network (LAN) technologies in ATM networks providing upper protocol layers an interface similar to the MAC layer. It was developed to give the possibility for connecting endstations to ATM networks, while network software can interact as if they were connected to traditional LANs, e.g., to IEEE 802.3 Ethernet or IEEE 802.5 Token Ring network.

Linux is an independent implementation of the POSIX operating system specification with System V Unix and BSD Unix extensions with freely available source code. This combined with the fact that LE support for Linux was still missing made it an ideal target system for LE software development.

The software was developed using ATM application programming interface and ATM protocol stack on Linux package. This package implements ATM support for Linux system and currently contains drivers for a number of ATM cards, components responsible for ATM signalling and ILMI address registration, ATM API and protocol stacks, and also LAN Emulation support after this project was completed.

Abstract in Finnish

TAMPEREEN TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto
Tietoliikennetekniikka
KIISKILÄ MARKO: Implementation of LAN Emulation Over ATM in Linux
Diplomityö, 49 sivua
Tarkastajat: Prof. Jarmo Harju ja Dos. Juha Heinänen

Lokakuu 1996
Avainsanat: LAN Emulation, LANE, asynchronous transfer mode, ATM, Linux

Työssä toteutettiin LAN-emulaation komponentit LE-asiakas ja LE-palvelu Linux käyttöjärjestelmään. Toteutuksessa noudatettiin ATM Forumin spesifikaatiota LAN Emulation Over ATM - Version 1.0. Työ toteutettiin osana suomalaista laajakaistaisen tietoliikenteen tutkimushanketta FASTERia.

LANE emuloi nykyisten lähiverkkotekniikoiden tarjoamia palveluita ATM-verkoissa tarjoten ylemmille protokollakerroksille MAC-kerroksen kaltaisen rajapinnan. Tämä mahdollistaa nykyisiin lähiverkkoihin (IEEE 802.3 Ethernet ja IEEE 802.5 Token Ring) kehitettyjen ohjelmistojen käyttämisen ATM:n ylitse.

Linux toteuttaa POSIX-käyttöjärjestelmäspesifikaation lisäten siihen ominaisuuksia System V Unix ja BSD Unix:sista. Linux oli oiva kohde ohjelmistolle, koska käyttöjärjestelmästä on lähdekoodi vapaasti saatavilla eikä tähän ympäristöön ollut vielä LAN-emulaatiota toteutettu.

Ohjelmisto kehitettiin käyttäen hyväksi ATM on Linux -paketin tarjoamaa ohjelmistorajapintaa ja ATM protokollapinoa. ATM on Linux sisältää ajureita ATM-korteille, signaloinnin ja ILMI-osoitteenrekisteröinnin toteuttavat komponentit, ohjelmointirajapinnan ATM yhteyksien muodostamiseen ja protokollapinot käyttöjärjestelmän ytimessä. Tämän projektin jälkeen paketissa on myös tuki LAN-emulaatiolle.

Table Of Contents

Glossary

AAL	ATM Adaptation Layer
AFI	Authority and Format Identifier
API	Application Programming Interface
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
ATMARP	Asynchronous Transfer Mode Address Resolution Protocol
BUS	Broadcast and Unknown Server
DSP	Domain Specific Part
ELAN	Emulated LAN
ESI	End System Identifier
FSM	Finite State Machine
ICD	International Code Designator
ICMP	Internet Control Message Protocol
IDI	Initial Domain Identifier
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IGMP	Internet Group Multicast Protocol
IISP	Inter-Switch Signalling Protocol
ILMI	Interim Local Management Interface, Integrated Local Management Interface
INET	Internet
IP	Internet Protocol
IPX	Internet Protocol Exchange
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
LAN	Local Area Network
LANE	LAN Emulation
LE	LAN Emulation

LEC	LAN Emulation Client
LECS	LAN Emulation Configuration Server
LES	LAN Emulation Service, LAN Emulation Server
LLC	Logical Link Control
LUNI	LAN Emulation User to Network Interface
MAC	Medium Access Control
MIB	Management Information Base
MMF	Multimode Fibre
MTU	Maximum Transfer Unit
NFS	Network File System
NIC	Network Interface Card
NNI	Network-Network Interface
NSAP	Network Service Access Point
P-NNI	Private Network-Network Interface
PDU	Protocol Data Unit
PLIP	Parallel Line Internet Protocol
POSIX	Portable Operating System Interface
PPP	Point-to-Point Protocol
PVC	Permanent Virtual Connection
QoS	Quality of Service
RCS	Revision Control System
RFC	Request For Comments
SAP	Service Access Point
SLIP	Serial Line Internet Protocol
SNAP	SubNetwork Attachment Point
SNMP	Simple Network Management Protocol
STP	Spanning Tree Protocol
SVC	Switched Virtual Connection
TCP	Transport Control Protocol
TDM	Time Division Multiplexing
TLI	Transport Layer Interface
UDP	User Datagram Protocol
UNI	User-Network Interface
VC	Virtual Circuit
VCC	Virtual Channel Connection
VCI	Virtual Channel Identifier
VPI	Virtual Path Identifier
WAN	Wide Area Network

Introduction 1

This master's thesis describes the design and implementation of components for LAN Emulation over ATM in the Linux environment. This work was done as a part of Finnish Broadband Telecommunication Research Project (FASTER) at Tampere University of Technology.

Asynchronous Transfer Mode is being developed to solve the problem of carrying all types of traffic end-to-end. The goal is to deliver voice, video and data across the network at very high speed. Today's local area networks (LANs) use variable length packets which are slow and difficult to handle. In ATM these basic units of data transfer are replaced by short fixed length cells that can be processed in the network components in a manner similar to time division multiplexing (TDM) used in telephone networks. ATM concepts and architecture are described in Chapter 2.

The services provided by ATM differ considerably from those of today's LANs. In order to use the vast base of existing applications built on top of LANs, it was necessary to define a new ATM service, LAN Emulation (LANE, LE). The LE emulates the services provided by existing LANs across ATM network. Operation of LANE is described in Chapter 3.

Linux is an independent implementation of the POSIX (Portable Operating System Interface)[POSIX] operating system specification with System V Unix and BSD Unix extensions. The Linux kernel has been written by Linus Torvalds from the Department of Computer Science, University of Helsinki with assistance from a loosely-knit team of volunteers across the Internet. The Linux system was chosen to target because a) it has freely distributable source code, b) it has quite powerful networking capabilities and c) some work has already been done for ATM connectivity in that environment. Chapter 4 describes some aspects of interest in the Linux environment.

ATM on Linux is a software package containing device drivers for a number of ATM cards, components responsible for ATM signalling, ILMI address registration, IP over ATM[RFC 1577] and several ATM related protocols. Most of the work done for this pack-

age has been done by Werner Almesberger from the Laboratoire de Réseaux de Communication at the Swiss Federal Institute of Technology in Lausanne. There are also many people who have contributed to the project in various ways. This package is described in Chapter 5.

The components of a LANE network are LE Clients and LE Service. The clients are responsible for providing a LAN-like access to upper protocol layers. The client implementation is described in Chapter 6. The LE Service provides to the clients the necessary information about the configuration of the network, a method to resolve MAC addresses to ATM addresses, and a possibility to send broadcast and multicast packets to all clients present in the virtual LAN. The implementation of LE Service is described in Chapter 7.

The project conclusions are discussed in Chapter 8.

Asynchronous Transfer Mode - ATM 2

2.1. Overview

An ATM network is composed of ATM switches and endstations (routers, workstations etc.) interconnected by point-to-point links. There are two kinds of interfaces: user-network interface (UNI) and network-network interface (NNI) (Figure 1).

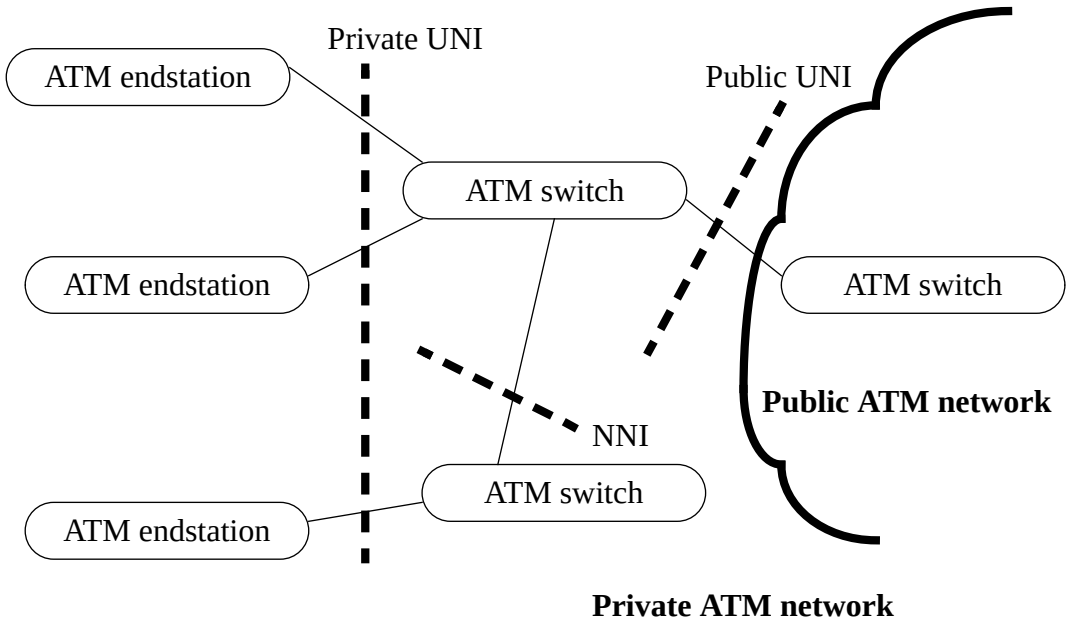

Figure 1: Interfaces in ATM Network

Public UNI connects a user with a private ATM network to a public service provider's network. Private UNI is used to interconnect ATM endstations to switches within private ATM network. Both of these are defined in ATM User-Network Interface Specification [ATM UNI]. NNI is defined to operate between ATM switches within private ATM net-

work. The NNI differs from the UNIs in such way that it exchanges routing information between switches. This information is then used to route calls across the ATM network. There is an ongoing effort to define Private NNI (P-NNI) routing protocol, which will be used in private ATM networks. First version of this specification is ready, but most of the ATM switches don't support this yet. Instead they use an earlier version of P-NNI, the Inter-Switch Signalling Protocol (IISP), which doesn't exchange routing information. In IISP the routing decisions are based on manually entered static entries in routing tables.

Standardization organizations working on ATM are the ATM Forum, IETF, and the Telecommunication Standardization Sector of International Telecommunications Union (ITU-T). The ATM Forum is a consortium representing a number of telephone operators, research organizations, users, and network industry, while ITU-T is an agency of the United Nations.

ATM networks are connection oriented. This means that before ATM applications can transfer data, a virtual circuit needs to be set up across the ATM network. These circuits (VCCs, virtual channel connections) are identified by a set of virtual path identifiers (VPIs) and/or a set of virtual channel identifiers (VCIs). These identifiers differ link by link, e.g. data in the connection goes from the user to the first switch in a channel identified by VPI_1/VCI_1, from the first switch to the second switch in a channel identified by VPI_2/VCI_2 and so on. A connection can be permanent (PVC, permanent virtual connection) or dynamically set up (SVC, switched virtual connection). PVCs are usually set up manually by the local network administrator, while SVCs can be formed dynamically whenever ATM endstations need to communicate across the network. After the connection is established, the applications can transfer data via the VCC.

In addition to point-to-point connections, which are from one ATM endstation to another, ATM networks support point-to-multipoint connections. In these kind of connections one ATM link, called Root Link, serves as the root in simple tree topology. When the root node sends data, all remaining parties in connection, leaf nodes, receive a copy of the data. The duplication of the data is performed in ATM switches. These connections are asymmetric, i.e., only the root node can send data.

A point-to-multipoint connection is set up by first creating a point-to-point link between the root node and one of the leaf nodes. After that connection is established, other leaves can be added to connection. A leaf node can be dropped from the connection at any time after it has been added to it.

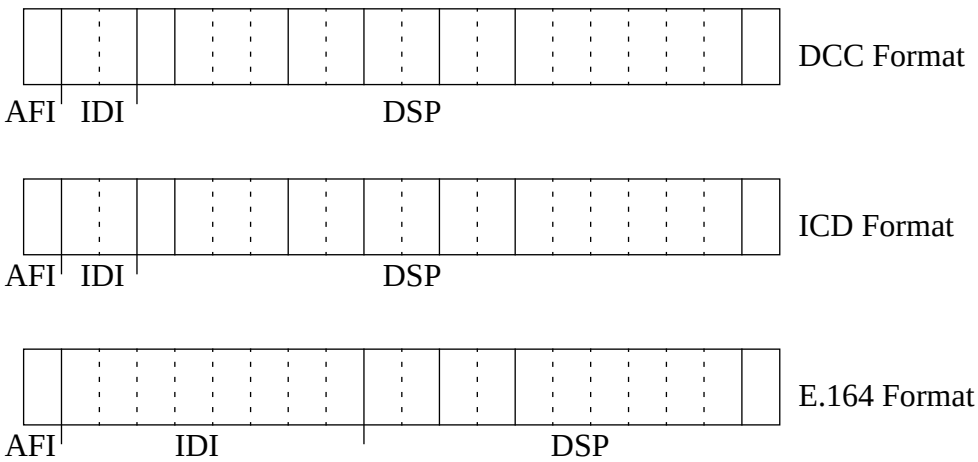

Figure 2: ATM Address Formats

Addresses are used in networks to identify endstations. A data path in ATM is identified by a VPI/VCI pair, but before such a connection is established, a route from caller to called party must be found. Network's switching elements try to find this path using information from their routing tables and the ATM address given in the call establishment request.

ATM address format is modelled after OSI Network Service Access Point (NSAP) (Figure 2). These addresses consist of three parts: AFI (Authority and Format Identifier), IDI (Initial Domain Identifier) and DSP (Domain Specific Part). The AFI identifies the authority allocating the IDI part of the address. Syntax of the address also depends on AFI, i.e., size and syntax of the IDI and the DSP (Figure 3).

The IDI specifies the network addressing domain from which the values of the DSP are allocated and by whom. The ATM address allocation is thus done in a hierarchical fashion.

AFI	Format
39	DCC ATM Format
47	ICD ATM Format
45	E.164 ATM Format

Figure 3: Specified AFI Codes in ATM Address Formats

For the ISO ICD IDI format, the International Code Designator (ICD) is allocated and assigned by the ISO 6523 registration authority (currently the British Standards Institute). In DCC ATM Format the DCC is currently allocated and assigned to countries. The E.164 format is useful for organizations that may wish to use the existing largely geographically based ISDN/telephony numbering format. The full ISDN number identifies an authority responsible for allocating and assigning values of the DSP. For more information about OSI NSAP addresses refer to [RFC1237].

The last byte in ATM address format, the selector byte, is not used in NNI routing. This means that call establishment requests with different values in this field are sent to a same ATM endstation. The endstation has thus 256 different ATM addresses.

2.2. Protocol Layers in ATM

Protocol reference model of ATM is shown in Figure 4.

The model is divided into several planes. The User Plane is responsible for transportation of data between user applications. The Control Plane takes care of call establishment, release and other control functions, e.g. UNI signalling is part of this plane. The Management Plane takes care of management functions in all layers. It also has the capability to exchange information between the User Plane and the Control Plane.

The purpose of the Physical Layer is to transport ATM cells given to it by the ATM layer from sender to destination. The ATM layer provides a transparent transfer of ATM cells between communicating upper layer entities in a pre-established connection. The transfer has to be done according to the negotiated traffic contract.

The ATM Adaptation Layer (AAL) is responsible for packet segmentation and reassembly, packet timing and bit rate control. There are several Adaptation Layer specifications

from which AAL5 [CCITT AAL5] is designed for computer data traffic.

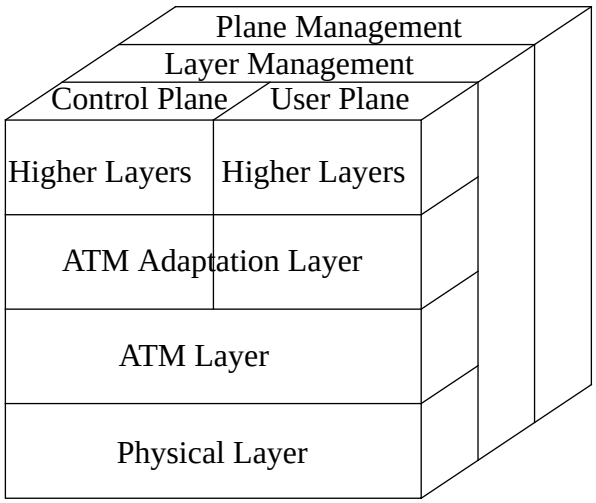

Figure 4: Protocol Reference Model

2.3. UNI Signalling

Signalling is a term describing a set of procedures for dynamically establishing, maintaining and clearing connections, SVCs, across an ATM network. The UNI Signalling is based on a subset of the broadband signalling protocol standard identified as Q.2931 [Q93B]. It is performed in a common out-of-band channel (usually VPI=0, VCI=5). It is agreed that implementation of ATM is done in phases. UNI 3.1 describes implementation of Phase 1 signalling, which has the following listed basic capabilities:

1. Demand (switched) channel connections.
2. Point-to-point and point-to-multipoint switched channel connections.
3. Connections with symmetric or asymmetric bandwidth requirements.
4. Single-connection (point-to-point or point-to-multipoint) calls.
5. Basic signalling functions via protocol messages, information elements, and procedures.
6. Class X, Class A, and Class C ATM Transport services.
7. Request and Indication of signalling parameters.
8. VPCI/VPI/VCI assignment.
9. A single, statically defined out-of-band channel for all signalling messages.
10. Error recovery.
11. Public UNI and Private UNI addressing formats for unique identification of ATM endpoints.
12. A client registration mechanism for exchange of addressing information across a UNI.
13. End-to-end Compatibility Parameter Identification.

For more detailed discussion of these features see [ATM Forum UNI31] page 154.

Signalling Messages

Call establishment is initiated by sending a SETUP message specifying the destination ATM address. Also Quality of Service (QoS) and the traffic parameters of the requested connection, fields identifying Service Access Point (SAP) of the called party and other AAL depended information are included. The SETUP message travels through the ATM

network components (i.e., switches) to destination, indicated by the target ATM address, causing a reservation of resources according to the traffic contract parameters along the message path. The network has the possibility to reject the call if the components don't have enough capacity to support the traffic profile requested in the SETUP message. The VPI/VCI values for the connection are added to the SETUP by the ATM network.

The called party can send a CALL_PROCEEDING message before parsing the SETUP it received. The CALL_PROCEEDING clears the timer in the network side of the called party's interface, which means that the ATM endstation gets more time to react to the SETUP message. If the call is accepted, the called party sends a CONNECT to the calling party. The CONNECT frame also contains the VPI/VCI values of the connection when it reaches the caller. Either the local switch of the called party or the calling entity then sends a CONNECT_ACKNOWLEDGE message in response to the CONNECT. The called party can start sending data to VCC identified by the negotiated VPI/VCI value after it has received the CONNECT_ACKNOWLEDGE, the calling party after it has received the CONNECT.

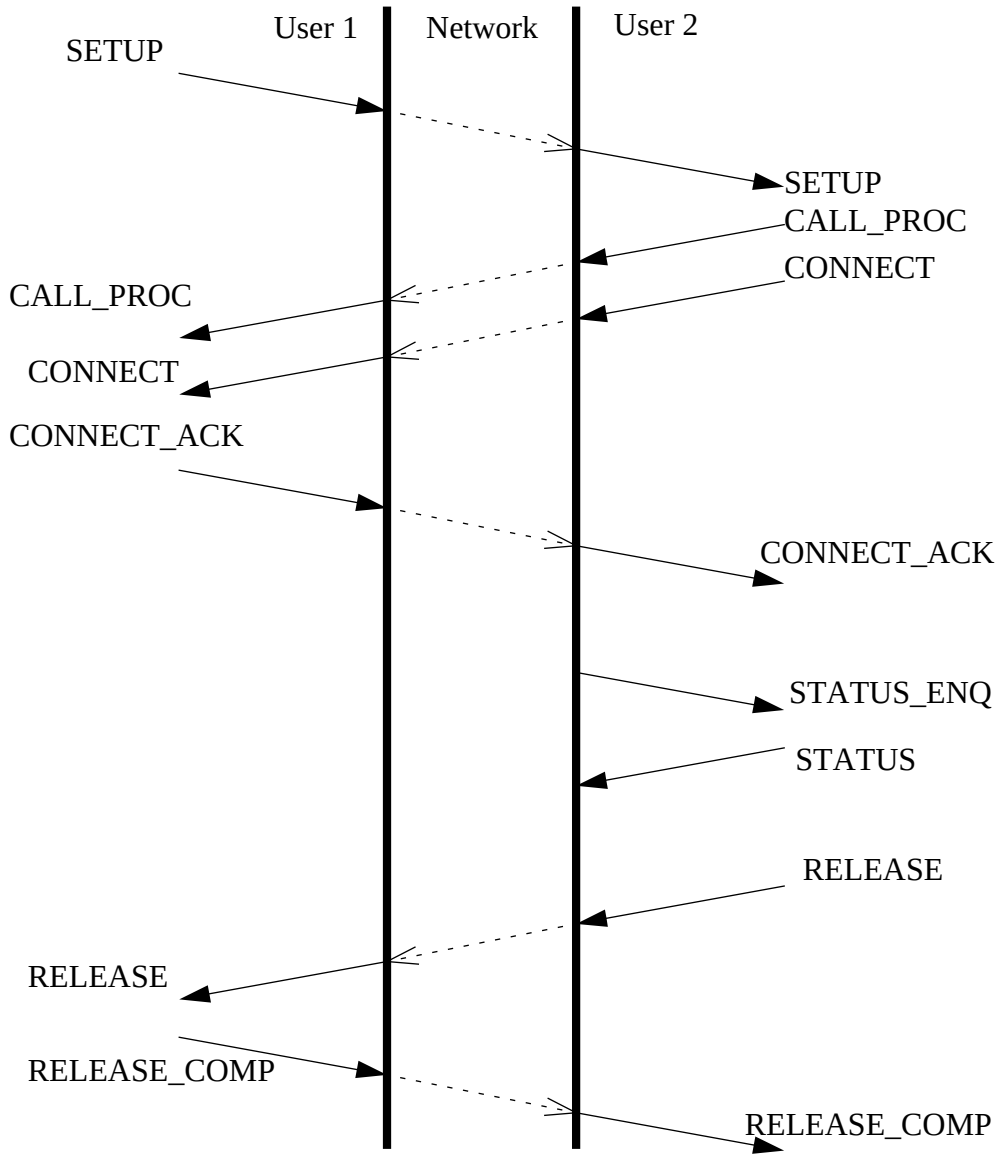

Figure 5: Signalling Message Flow Example.

Status of the connection can be queried using STATUS_ENQUIRY message. These can be generated by either the local switch or the endstation. The destination of the message responds with a STATUS message describing what it is thinking about the state of the connection and in the case of an error, the cause value of the error.

Connection teardown can be initiated by either party. It is achieved by sending a RELEASE message containing the cause of the teardown. The other end of the connection responds by sending a RELEASE_COMPLETE message thus acknowledging the teardown. An example of a signalling sequence is presented in Figure 5.

In point-to-multipoint connections the first connection is formed as described above. Leaves can be added to the connection by the root node by sending an ADD_PARTY message. The network forms a SETUP message from the ADD_PARTY message and sends it to appropriate endstation. After that the leaf node has the possibility to either reject or accept the incoming call. The root node gets notified of the successful addition of the leaf when it receives an ADD_PARTY_ACKNOWLEDGE message. If the ADD_PARTY was not successful, an ADD_PARTY_REJECT is sent. A DROP_PARTY is used in dropping a party from an existing point-to-multipoint connection. A DROP_PARTY_ACKNOWLEDGE is sent in response.

2.4. Interim Local Management Interface

Interim Local Management Interface (ILMI, Integrated Local Management Interface)[ATM Forum UNI31] is a specification for controlling and managing the links between the ATM components. It is agreed that this is a temporary solution for the management and procedures for final models are under study.

ILMI uses Simple Network Management Protocol (SNMP) and ATM UNI Management Information Base (MIB) to query and set any ATM user device status and configuration information. The messages are transferred on top of AAL5 using a well known channel (VPI=0, VCI=16).

ATM ILMI MIB contains data about following subjects:

1. Physical Layer. Information about the interface in the physical layer, e.g. type of media, status of the link etc.
2. ATM Layer. E.g. configuration information about range of valid VPI and VCI values, the amount of configured virtual paths etc.
3. Statistics of ATM Layer.
4. Virtual Path Connections.
5. Virtual Circuit Connections.
6. Address Registration Information. ILMI provides a way for ATM endstations to inform ATM switch across the UNI of its unique MAC address, ESI (End System Identifier). It then receives the remainder of the node’s full ATM address, the network prefix, in return.

2.5. Internetworking of Existing Protocols

A key to ATM’s success will be the ability to use existing network protocols in ATM. There exists a vast base of network applications using the existing network protocols that users will want to use in the future. They will also want connectivity to existing networks. The connectivity and the ability to use the applications is achievable either by using existing network protocols in ATM, both IP (Internet Protocol) and IPX (Internet Packet

Exchange) or MAC layer protocols (Figure 6).

There are currently two different ways of running network layer protocols across ATM networks. In native mode operation, address resolution is used in mapping network layer addresses to ATM addresses and network layer packets are then carried across the ATM network. Even though all network layer protocols could be enhanced to run directly over ATM, the only protocol for which this has been done is IP. This work has mainly been done in Internet Engineering Task Force's (IETF) working groups.

Two aspects need to be addressed when transporting network layer protocols over ATM, address resolution and packet encapsulation. Two different methods for the packet encapsulation are described in [RFC1483]. The first method uses LLC/SNAP (Logical Link Control/SubNetwork Attachment Point) encapsulation to carry different protocols over a single VCC. In this method information about the protocol is encoded in the LLC header which enables the receiver to determine the protocol of the received packet. The second method uses VC (ATM Virtual Circuit) Based Multiplexing. The carried interconnect protocol is identified implicitly by the VC connecting the two ATM endstations, i.e., each protocol must be carried over a separate VC. More common encapsulation method today is the LLC/SNAP encapsulation used in the IP over ATM protocols. The address resolution from network addresses to ATM addresses is described in [RFC1577].

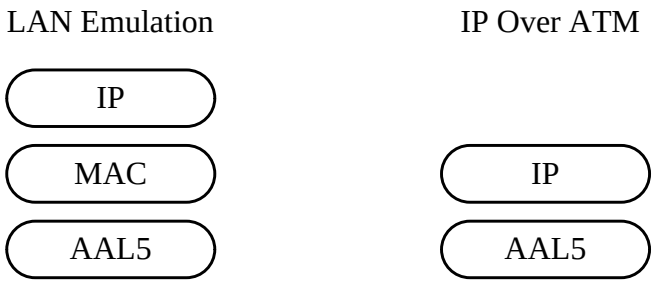

Figure 6: Protocol Stacks in Internetworking Protocols

Another way of carrying network protocols over ATM is LAN Emulation. It emulates the behaviour of local area networks on top of ATM. Endstations are identified by a MAC (Media Access Layer) address. The packet encapsulation is done to appropriate MAC packet format. Separation of different protocols is encoded automatically to the MAC header and the packets carrying different protocols can thus be sent over a single VCC. The address resolution is done from MAC addresses to ATM addresses. LANE is discussed more thoroughly in the following chapter.

When comparing performance of these two technologies, IP over ATM is faster. This is because it lacks the need to perform IP address to MAC address resolution, and because used data packets are larger. Standard for Maximum Transfer Unit (MTU) for IP over ATM is 9180 octets, while in LANE the MTU depends on LAN technology being emulated. The most popular LAN technology is IEEE 802.3 Ethernet, which has 1500 octets MTU.

In favour of LANE is the fact that expensive routers are needed when IP over ATM subnetworks are connected to traditional networks. LANE does the emulation in the MAC layer, so interconnecting with existing LANs can be done via bridging technology. Bridging is much easier and cheaper to implement than routing. IP over ATM also lacks the ability to send multicast and broadcast packets effectively.

LAN Emulation Over ATM 3

3.1. Overview

LANE emulates the services of existing Local Area Network (LAN) technologies and provides upper protocol layers an interface similar to the MAC layer. It was developed to give the possibility for connecting endstations to ATM networks, while network software can interact as if they were connected to traditional LANs, e.g., to IEEE 802.3 Ethernet [IEEE802.3][Ethernet] or IEEE 802.5 Token Ring [IEEE802.5] network.

The services provided by the ATM network differ from those in today's LANs. The LAN endstations can send data without first establishing a connection, while ATM is connection-oriented technology. In traditional LANs the endstations share the same physical media, so there is a possibility to effectively send packets to all or some of the endstations (broadcast and multicast operation).

The goal in LE has been to keep upper protocol layers unchanged, i.e., the interface to the LANE entity is exactly the same as to traditional MAC layers. This enables protocol stacks like NetBIOS, AppleTalk, IPX etc. to be run over the LANE. In some occasions it is desirable to use several separate domains within a single network. This is achieved in ATM networks using LANE by defining an "emulated LAN" which consists of several ATM endstations. This kind of emulated LAN (ELAN) can be thought to be analogous to a Ethernet/Token Ring segment. One ATM endstation can be a member of several ELANs and there can be several ELANs in operation in a single ATM network.

Because LANE emulates traditional LAN technologies in the MAC layer, the connectivity to LAN networks can be done using existing bridging methods. Note that this is cheaper than in case of IP over ATM, where a router is needed to interconnect ATM to LAN network.

3.2. Architecture

An emulated LAN consists of a number of LE Clients (LECs) and a LE Service. LE Clients are located in ATM endstations and they request services from LE Service using LAN

Emulation User-Network Interface (LUNI). LE Service consists of three components: LAN Emulation Configuration Server (LECS), LAN Emulation Server (LES) and Broadcast and Unknown Server (BUS) (Figure 7).

Data transportation is carried over AAL5. Connections can be either PVCs or SVCs. Normal ATM signalling is used to establish connection in SVC environment. During normal operation the LEC has connections open to the LE Service and to LECs it wants to communicate with. To LES and BUS it has two connections for each, one is used for sending LE control frames or data packets and the other is for receiving responses and data packets.

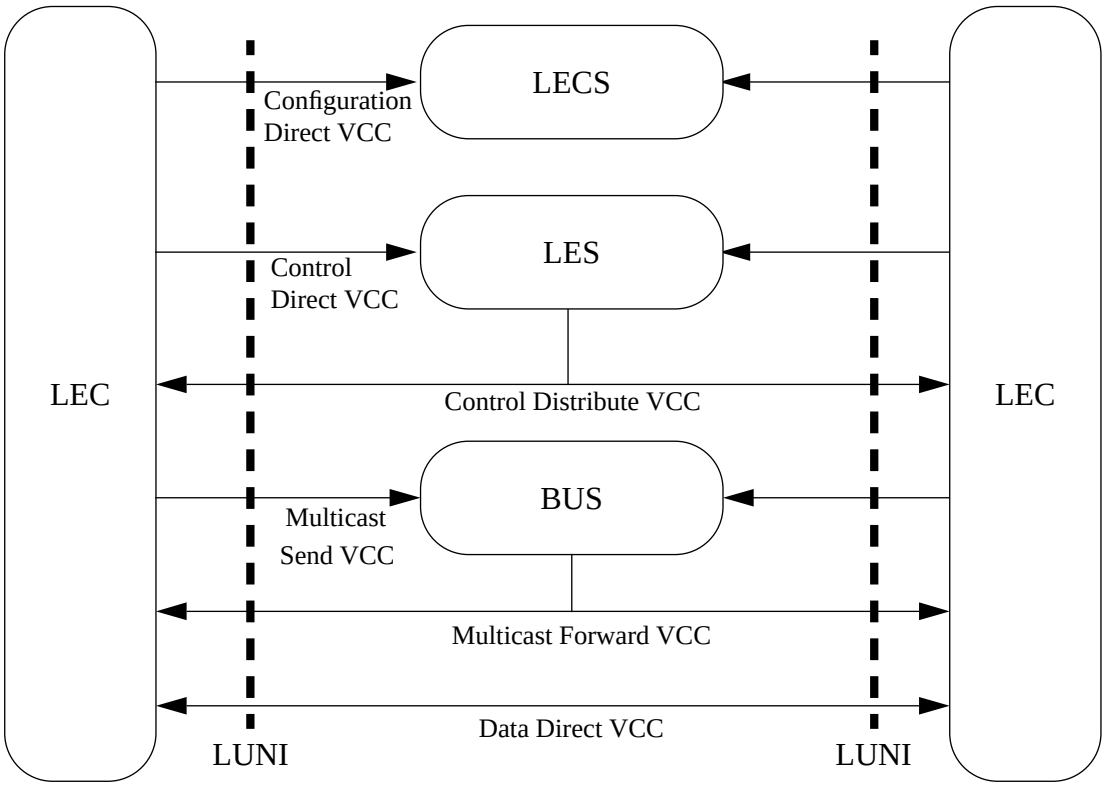

Figure 7: Emulated LAN components, connections and LUNI

The connections from LEC to LEC and from LEC to LE Service components are point-to-point connections, while the VCCs used to distribute frames from the LE Service to LECs are point-to-multipoint. The components of LANE are shown in Figure 7. As one can see from the figure, there exists no specification for communication between the servers, and there is only one instance of each server in ELAN, i.e., no redundancy. If one of the servers goes down, the whole ELAN is unoperational.

When the LEC wants to detach from an emulated LAN, it must tear down all connections. In SVC environment this is done using UNI signalling messages.

It is not required that LE operates in SVC environment, but the LANE specification doesn't specify operation in PVC environment very accurately. It is also possible to use point-to-point connections from LE service to LECs.

3.3. Components

A LAN Emulation Client is an entity in an ATM endstation that performs data forwarding, address resolution and other control functions. It uses the LUNI interface when communicating with other components in emulated LANs. These ATM components can be in workstations, routers, bridges, ATM switches, etc. It provides upper protocol layers a MAC like interface similar to IEEE 802.3/Ethernet or IEEE 802.5/Token Ring LAN.

A LAN Emulation Configuration Server implements the distribution of LECs to different emulated LANs. This is done by giving different LAN Emulation Server ATM addresses to the LECs. The distribution is based on LECS's configuration database and information provided by the LE client. It is not required that a LECS exists for all emulated LANs. It is possible to bypass this configuration phase by directly telling the ATM address of the LES to the LEC.

A LAN Emulation Server performs the control coordination function for the emulated LAN. The LE clients register MAC addresses and/or route descriptors they represent to the LES, and later query it when they want to resolve MAC addresses/route descriptors into ATM addresses. Other LE control messages which are to be distributed to every client in the ELAN are also sent to the LES. The LES forwards these messages using a Control Distribute VCC which it has set up as a point-to-multipoint connection to every client in the ELAN.

A Broadcast and Unknown Server handles data sent by the clients to broadcast and multicast MAC addresses and some of the data directed to unicast addresses. The LE Client has a possibility to send data directed to some unicast address to the BUS before the target's ATM address has been resolved and a Data Direct VCC to that destination has been established.

3.4. Operation

Initialization and Configuration Phase

A LE Client starts its initialization in an emulated LAN by getting the ATM address of the LECS and setting up a Configuration Direct VCC to it. The address is obtained either via manual configuration or by obtaining it from the network using ILMI. The extension to ILMI for this is described in [ATM Forum LANE]. There is also the possibility to use the Well-Known LECS address when operating in SVC environment or the LECS PVC VPI=0 VCI=17 in PVC environment.

There are also other parameters for operation, e.g., MAC addresses the LEC represents and several timer values which must be set to some initial values before the joining can begin. These timer values control the operation of a LEC by telling e.g. how long it holds entries in its ARP cache.

The parameters identifying the type of the connection and thus the Service Access Point (SAP) in the endstation containing the LECS are defined in the LANE specification. This information is required by ATM signalling when establishing connections.

After the connection has been set up, the LEC sends a LE_CONFIGURE_REQUEST to the LECS. All control protocol data units (PDUs) follow the same form (Figure 8). The

type of the control message is encoded in the OP-CODE field (Figure 9).

Size	Name	Function
2	MARKER	Control Frame = 0xFFFF.
1	PROTOCOL	LANE protocol = 0x01.
1	VERSION	LANE protocol version = 0x01.
2	OP-CODE	Control Frame Type. List of valid OP-CODES is shown in figure 9.
2	STATUS	Shows whether request was accepted in responses. Always 0x0000 in requests.
4	TRANSACTION-ID	Arbitrary value supplied by the sender and returned by the responder.
2	REQUESTER-LECID	LECID of LE Client sending the request.
2	FLAGS	Bit flags used in some control frames.
8	SOURCE-LAN-DESTINATION	MAC address or Route Descriptor of the client. Not used in all control frames.
8	TARGET-LAN-DESTINATION	MAC address or Route Descriptor of the target. Not used in all control frames.
20	SOURCE-ATM-ADDRESS	ATM address of the LEC. Not used in all control frames.
1	LAN-TYPE	Specifies type of the ELAN. Not used in all control frames
1	MAXIMUM-FRAME-SIZE	Maximum data frame size used in ELAN. Not used in all control frames.
1	NUMBER-TLVS	Number of Type-Length-Value elements encoded in control frame.
1	ELAN-NAME-SIZE	Number of octets in ELAN-NAME.
20	TARGET-ATM-ADDRESS	ATM address of the target. Not used in all control frames.
32	ELAN-NAME	Name of the ELAN.
4	ITEM_1-TYPE	TLV item type. Three octets of OUI (0x00A03E) and one octet of identifier.
1	ITEM_1-LENGTH	TLV item length.
??	ITEM_1-VALUE	
	Etc.	

Figure 8: LE Control PDU Format

The LECS responds by sending a LE_CONFIGURE_RESPONSE containing information about:

• ATM address of a LES,
• type of emulated LAN (i.e., IEEE 802.3 or IEEE 802.5),
• maximum data frame size used in emulated LAN,
• name of the ELAN (can be empty),
• optional configuration info about the ELAN.

These optional fields are encoded in T-L-V fields (Type - Length - Value) in the LE_CONFIGURE_RESPONSE. These fields can contain, e.g., timer values mentioned above. The LE Client must update its configuration based on these values.

OP-CODE value	OP-CODE function
0x0001	LE_CONFIGURE_REQUEST
0x0101	LE_CONFIGURE_RESPONSE
0x0002	LE_JOIN_REQUEST
0x0102	LE_JOIN_RESPONSE
0x0003	READY_QUERY
0x0103	READY_IND
0x0004	LE_REGISTER_REQUEST
0x0104	LE_REGISTER_RESPONSE
0x0005	LE_UNREGISTER_REQUEST
0x0105	LE_UNREGISTER_RESPONSE
0x0006	LE_ARP_REQUEST
0x0106	LE_ARP_RESPONSE
0x0007	LE_FLUSH_REQUEST
0x0107	LE_FLUSH_RESPONSE
0x0008	LE_NARP_REQUEST
0x0009	LE_TOPOLOGY_REQUEST

Figure 9: LE Control Frame Operation Codes

Join Phase

During the join phase the LEC establishes connection to the LES and sends a LE_JOIN_REQUEST telling that it wants to join the ELAN indicated by the ELAN-NAME field in request. The LES sends a LE_JOIN_RESPONSE and thus confirms or denies the joining of the LEC. The ATM address used in forming connection to the LES was returned in the LE_CONFIGURE_RESPONSE during the configuration phase.

The LEC first sets up a Control Direct VCC to the LES. After that it sends a LE_JOIN_REQUEST and waits for the LES to establish a Control Distribute VCC to it. The LES then sends a LE_JOIN_RESPONSE either via the Control Direct VCC or the Control Distribute VCC. If any of these procedures fail, the LES tears down its connections to the LE client and the LEC is required to terminate the join phase.

The LE Client has the possibility to register one of the MAC addresses/route descriptors it represents in the join request. This is not required, and the LEC can do this at a later time with a registration protocol which is explained later. The LES assigns each LEC in the ELAN an unique LECID and tells it to LEC in the join response. This value is used by the LEC in control PDUs and later in data frame headers.

Connecting to the BUS

After the join phase is over, the LEC sets up a Multicast Send VCC to the BUS. It obtains the BUS's ATM address by sending a LE_ARP_REQUEST for broadcast MAC address (all 1's) to the LES. The BUS then adds the LEC to its Multicast Forward VCC (point-to-multipoint connection). The LEC must use the Multicast Send VCC when sending data frames to broadcast and all multicast MAC addresses. Initial unicast frames can also be sent via this VCC while the destination's ATM address is being resolved. This unicast traffic to the BUS is limited with variable values assigned during initialization of the LEC. These variables tell how many unicast data packets can be sent to the BUS in a certain time.

After the LEC has accepted the Multicast Forward VCC, it is ready to start the data forwarding in the ELAN, i.e., it is operational.

Registration of MAC Addresses

When a LE client wants to register MAC address/ATM address pairs, it sends a LE_REGISTER_REQUEST to the LES. The LES accepts or rejects this registration by sending a LE_REGISTER_RESPONSE. Unregistering an address pair is done with a LE_UNREGISTER_REQUEST, to which the LES responds by sending a LE_UNREGISTER_RESPONSE. A LEC can represent several MAC addresses for example when it is acting as a proxy between an emulated LAN and a traditional LAN (e.g. a bridge).

Address Resolution

The LEC must resolve MAC addresses to ATM addresses when it wants to send data to other clients in the ELAN. This is done by sending a LE_ARP_REQUEST to the LES. The LES can either respond directly basing its information on its internal database of registered MAC addresses or it can forward the request to the Control Distribute VCC and the LEC responsible for that MAC address will respond. The response is returned in a LE_ARP_RESPONSE. When a LEC receives a LE_ARP_RESPONSE containing an ATM address for which it hasn't established a Data Direct VCC, it must try to form a connection to that ATM address.

Each LEC maintains a cache of MAC address to ATM address mappings. When a LEC gets a request to send data to a MAC addresses it first consults its cache and if that fails it tries to resolve the ATM address from the LES. This ensures that excess LE_ARP_REQUESTs are not sent and that the response time of the LEC is better. The cache entries are aged out based on the policy of the LEC and timer values specified in the

LANE specification.

The cache entries are aged out quite slowly. LE was designed keeping in mind that typically a spanning tree protocol [STP] would be run within the ELANs and external networks bridged to it. If the LAN bridges running STP detect loops in the networks, they will shut off appropriate ports to break the loop. This might cause the network topology to change dynamically so that the reachability of a MAC address would change, i.e., MAC addresses will be reachable via some other ATM address. Figure 10 shows an example where network topology changes because of Spanning Tree Protocol. Workstation 2 will have old reachability information in its cache after the topology has changed.

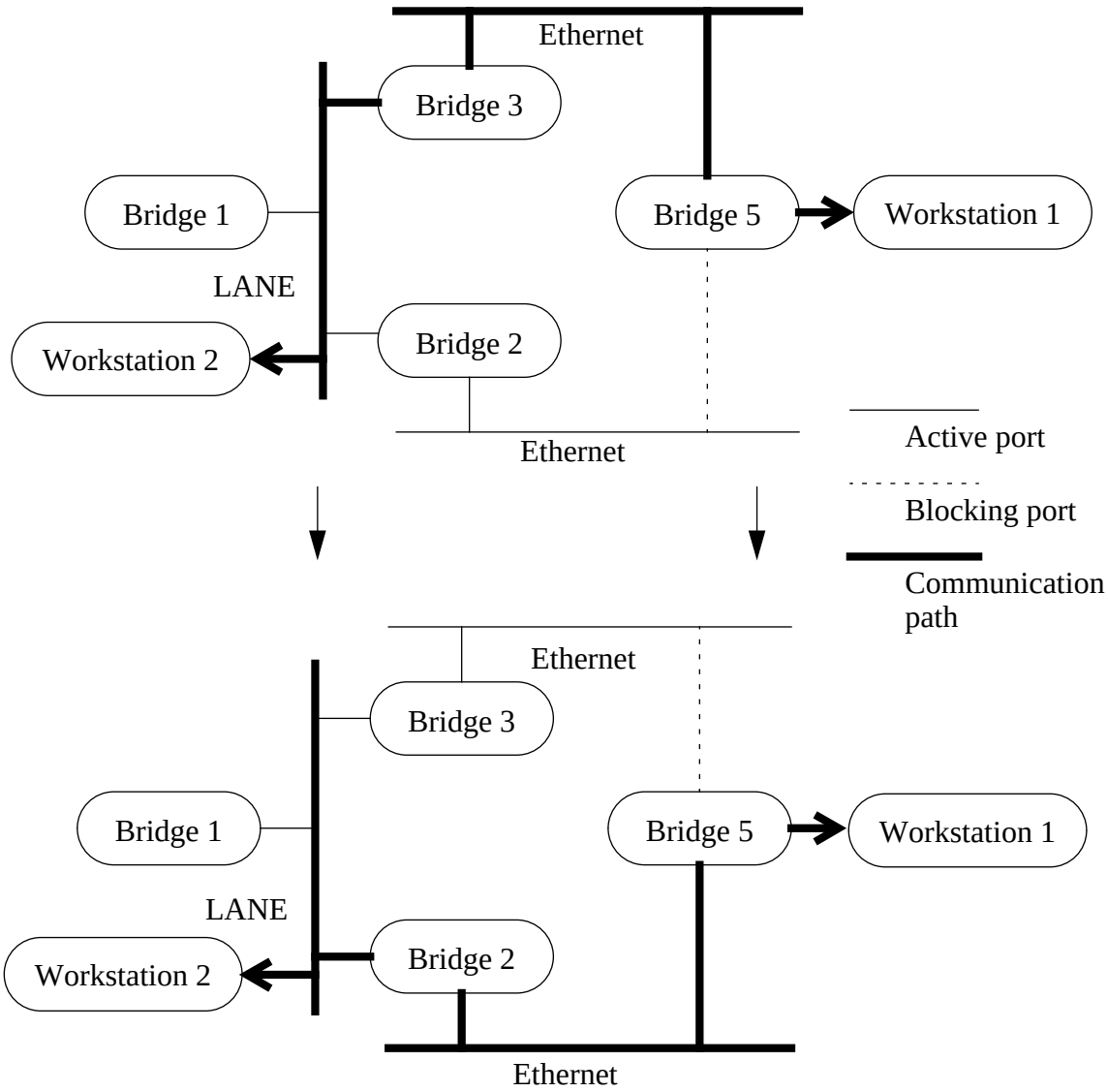

Figure 10: Spanning Tree Algorithm Changing Network Topology

This change might have been triggered by, e.g., change in path costs. Each path (i.e., in bridge a port) is assigned a cost, which is used in calculating a favourable route across the network. If there exists several routes between endstations, the one with the lowest cost is used.

LE gives a possibility to speed up the cache aging with LE_TOPOLOGY_CHANGE frames. They are usually sent by these LAN bridges when they suspect that the network

topology has changed. When a LEC receives such a message, it reduces the aging period thus causing the caches to refresh faster.

LE_NARP_REQUESTs are sent when a LE client thinks that a remote MAC/ATM address binding has changed. This causes the LECs receiving that message to update their address caches accordingly.

The following shows an example of operation of the address resolution, when an ATM endstation wants to establish a connection to another in IP environment implementing IEEE 802.3 /Ethernet type of LAN Emulation.

- Caller knows destination's IP address, but not MAC and ATM addresses.
- The caller sends an IP-ARP to local emulated LAN requesting the MAC address bound to the destination IP address. The destination MAC address in this packet is the broadcast address, so it is sent to the BUS and thus to all LECs in the ELAN.
- The IP-ARP request contains also sender's IP address and MAC address, so the reply can be sent directly to the requester. The called party must now send an IP-ARP response to the caller. It knows the caller's MAC address (and IP address), so it issues a LE_ARP_REQUEST requesting the caller's ATM address.
- The called party receives the LE_ARP_RESPONSE for the request and it forms a Data Direct VCC between these endstations. The IP-ARP response and following data transfer can be done via this Data Direct VCC.

Data Transfer Phase

After the Data Direct VCC is established, The LE clients are ready to transfer data in it. The data encapsulation method depends on which MAC protocol the clients emulate. The first is based on IEEE 802.3 / Ethernet data frame (Figure 11).

Size	Name	Function
2	LE header	Usually contains LECs LECID. This can be used to filter received frames.
6	Destination Address	MAC address of the destination component.
6	Source Address	MAC address of the originating component.
2	Type/Length	Contains either EtherType of the frame (DIX Ethernet) or the length of the data frame (LLC).
??	Info	Actual data

Figure 11: LE Data Frame Format for IEEE 802.3 / Ethernet Frames

The second is for IEEE 802.5 / Token Ring data frames (Figure 12).

Size	Name	Function
2	LE header	Usually contains LECs LECID. This can be used to filter received frames.
1	Access Control Pad	Not used in LAN Emulation.

Size	Name	Function
1	Frame Control	LE allows only LLC frames to be sent, so octet is of form ‘0100YYY’ binary, where YYY is priority set by the source LLC.
6	Destination Address	MAC address of the destination component.
6	Source Address	MAC address of the originating component.
0-20	Routing information field	Used in determining where to send the frame.
??	Info	Actual data

Figure 12: LE Data Frame Format for IEEE 802.5 / Token Ring Frames

Data transfer must be done either via the Data Direct VCC or the Multicast Forward VCC according to rules explained before.

Flush Protocol

Before a Data Direct VCC is set up, initial unicast frames can be sent via the BUS, so there is a possibility for frames to be delivered out of order, i.e., data frames sent via the BUS arrive after the packets sent to just established Data Direct VCC. Because the traditional LAN protocols deliver frames in order and the upper protocol layers expect to receive them accordingly, there must be a way to switch data paths without losing the data frame ordering.

When switching from the Multicast Forward VCC to a Data Direct VCC, the sender first transmits a LE_FLUSH_REQUEST down the old VCC. The flush message is returned to the sender by the receiver via a control VCC. The sender doesn’t send any data packets down the old data path after it has transmitted the LE_FLUSH_REQUEST, so it either drops the frames or queues them for later transmission. It switches to use the new data path either when it receives an appropriate LE_FLUSH_RESPONSE or after a configurable Path Switching Delay has elapsed.

3.5. Future Directions of LANE

The Technical Committee of the ATM Forum is currently specifying enhancements that will appear in LANEv2. This will bring two important extensions: interserver protocols and Quality of Service (QoS) support.

Interserver protocols will allow greater scalability for LANE and support server function redundancy. This redundancy is needed, because in the current implementation failure of one LE Service component leads to a situation where the whole ELAN is unoperational. The QoS support will be accomplished by setting several VCCs between ATM endstations with different QoS parameters. This is particularly important for real-time applications such as videoconferencing or video-on-demand.

Networking in the Linux Operating System 4

4.1. Overview

It is expected that the reader is familiar with concepts and operation of UNIX systems. To learn more about the UNIX operating systems in general, readers are to refer [BSD]. Most of the claims and information present in this chapter can be found from [LINUX], Linux Networking HOWTOs, kernel source code and [KHG].

Originally Linux was written to run in Intel 80386 and 80486 processors, but nowadays it has been ported to several other architectures. The development of the Linux system was started by Linus Torvarlds, and currently software for the operating system kernel is being developed around the world by several individuals. Linux is mostly compatible with a number of UNIX standards on the source code level, including IEEE POSIX.1, System V, and BSD features. The operating system kernel is nowadays written mostly in C, some parts are written using assembly language.

Some of the features of the Linux system are:

- Multitasking, multiuser operating system.
- Various filesystem types for storing data. For example ext2fs filesystem, which has been developed specifically for Linux. Other filesystem types, such as the MS-DOS and Xenix filesystem, are also supported.
- A complete implementation of TCP/IP networking. This includes device drivers for many popular Ethernet cards, SLIP (Serial Line Internet Protocol), PLIP (Parallel Line Internet Protocol), PPP (Point-to-Point Protocol), NFS (Network File System), and so on.
- The Linux kernel supports demand-paged loaded executables. That is, only those segments of a program which are actually used are read into memory from disk. Also, copy-on-write pages are shared among executables, meaning that if several instances of a program are running at once, they will share pages in physical memory, reducing overall memory usage.
- The kernel also implements a unified memory pool for user programs and disk cache. In

this way, all free memory is used for caching, and the cache is reduced when running large programs.
- Executables use dynamically linked shared libraries, meaning that executables share common library code in a single library file found on disk. Linux shared libraries are dynamically linked at run-time.

4.2. Networking in Linux

Linux supports a full implementation of the TCP/IP (Transport Control Protocol/Internet Protocol) networking protocols, which has become the most successful mechanism for networking computers worldwide. The current implementation of TCP/IP and related protocols for Linux is called ``NET-3", which in this context means the third implementation of TCP/IP for Linux.

The original kernel based networking code for Linux was written by Ross Biro. He used ethernet drivers written by Donald Becker, a SLIP driver written by Laurene Culhane and a D-Link driver by Björn Ekwall.

The further development of the Linux networking code was later taken up by Fred van Kempen, who took Ross's code and produced the NET-2 release of network code. NET-2 went through a number of revisions until release NET-2d, when Alan Cox set about debugging Fred's code with the aim of producing a stable and working release of code for incorporation into the standard kernel releases. This code was originally called NET-2D(ebugged) and was incorporated into the standard kernel releases some time before Linux version 1.0 was released.

The latest version of the code, NET-3, appears in kernel releases 1.1.5 and later. It is essentially same code as in NET-2D, but with many fixes, corrections and enhancements.

Many other people have made contributions by way of bug fixes, ports of applications and by writing device drivers.

The rest of the chapter assumes that the reader is familiar with TCP/IP protocol suite.

4.3. BSD Socket API

There exists several application programming interfaces (APIs) for communication protocols in UNIX. Two of the most important ones are Berkeley sockets and the System V Transport Layer Interface (TLI). The Linux system uses socket based approach. An API is defined as a number of system calls and relating data structures.

System calls are requests to operating system (kernel) to do a hardware/system specific or privileged operations. In the Linux system they are implemented as stubs in libc library, which processes the call according to its parameters and then passes the request to kernel.

There are several types of sockets (e.g. UNIX domain sockets, Internet (INET) sockets). The system calls are the same for each of them, but parameters and operation differ. Figure 13 shows a layered representation of the communication data path from an application using INET sockets to the actual hardware. Each of the socket types “registers” its function in kernel’s socket layer. When an application wants to e.g. create a socket, the protocol family is given as a parameter and the socket layer uses this information to determine which function to call. Protocol families have different types of addresses (e.g. INET sockets might use an IP address and a UDP/TCP port), so system calls needing protocol

addresses are passed different address structures depending on the protocol used.

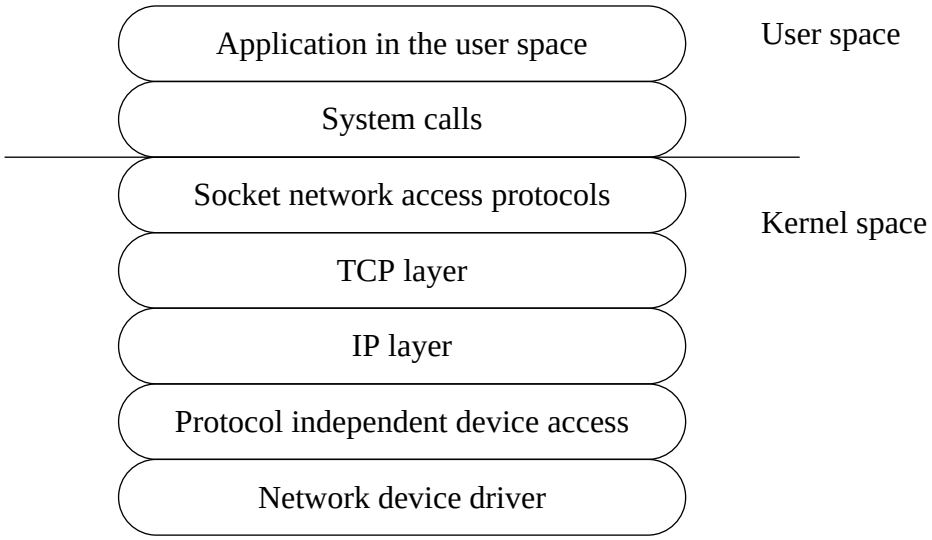

Figure 13: Layered representation of Networking in Linux

Figure 14 shows an example of system call sequence when two processes communicate using sockets. System call `int socket(int domain, int type, int protocol)` creates an endpoint for communication. It specifies the communication protocol for the socket and creates a local instance of the socket in the kernel. `Socket()` returns a socket descriptor, which is used to identify the socket in later system calls. Parameter `domain` defines the address domain of the socket, i.e., UNIX, INET etc.

`Int bind(int sockfd, struct sockaddr* my_addr, int addrlen)` is used to a give name (local address) to an unnamed socket. E.g. when a socket is used for listening incoming connections, the binding tells the system that it should route calls for the bound address to this socket.

System call `int listen(int s, int backlog)` is used in connection-oriented protocols to indicate to the system that the application wants this socket to receive connection forming requests. The parameter `backlog` tells how many pending connections this socket will hold.

These calls are received using `int accept(int s, struct sockaddr *addr, int *addrlen)`. `Accept()` takes the first connection request from a queue of incoming connections and creates another socket with similar properties as the listening socket. This new socket is used in transferring data with the entity that made the call. The parameter `addr` is filled with the caller's address.

To establish a connection to a socket that is waiting connections after `listen()` system call `int connect(int sockfd, struct sockaddr* serv_addr, int addrlen)` is used. It has the destination address in its arguments. For most connection-oriented protocols `connect()` causes the actual connection establishment.

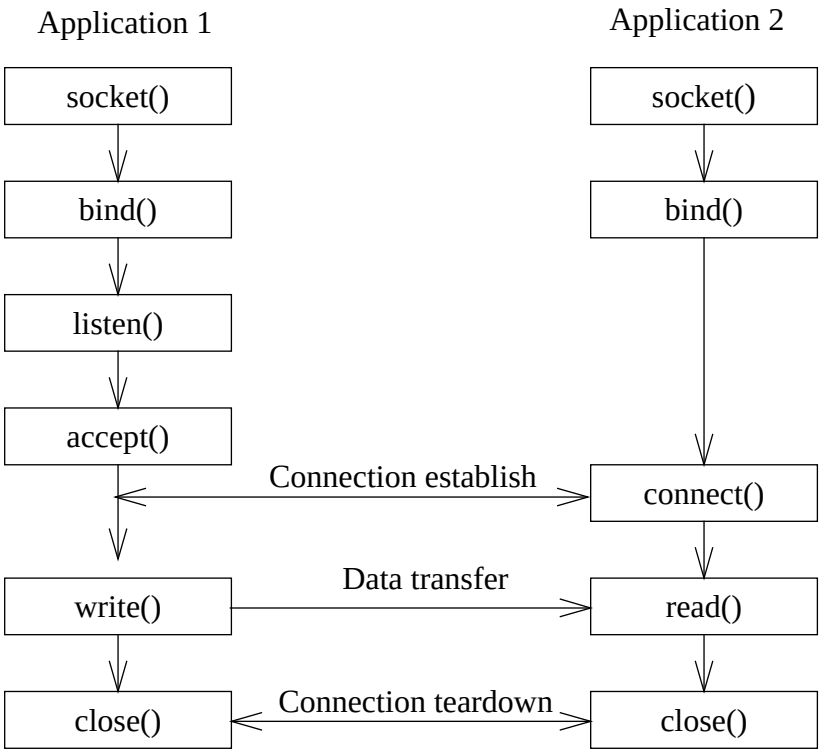

Figure 14: Example of BSD socket API usage with connection-oriented protocol

Data can be transferred with different system calls. The transmission can be done using e.g.:

- `int send(int s, const void *msg, int len, unsigned int flags)`
- `size_t write(int fd, const char *buf, size_t count)`
- `int sendmsg(int s, const struct msghdr *msg, unsigned int flags)`

and receiving with

- `int read(int fd, char *buf, size_t count)`
- `int recv(int s, void *buf, int len, unsigned int flags)`
- `int recvmsg(int s, struct msghdr *msg, unsigned int flags)`.

Behaviour and arguments of these are protocol specific. The system calls mentioned above are used in connection-oriented protocols.

When the socket is not needed anymore, the connection is torn down with `int close(int sockfd)`. If the socket being closed is associated with a protocol that provides reliable data transfer (e.g. TCP), the operating system might still try to send undelivered data before tearing down the connection.

Due to fact that the Linux's ATM API uses BSD sockets and ATM is connection oriented, behaviour of the BSD Socket API was presented in connection-oriented protocol case. The ATM sockets are described in chapter 5.

4.4. Networking Layers in the Kernel

Networking in the Linux system is usually done with IP, so the following describes how network protocols are layered in the kernel in the TCP/IP protocol suite.

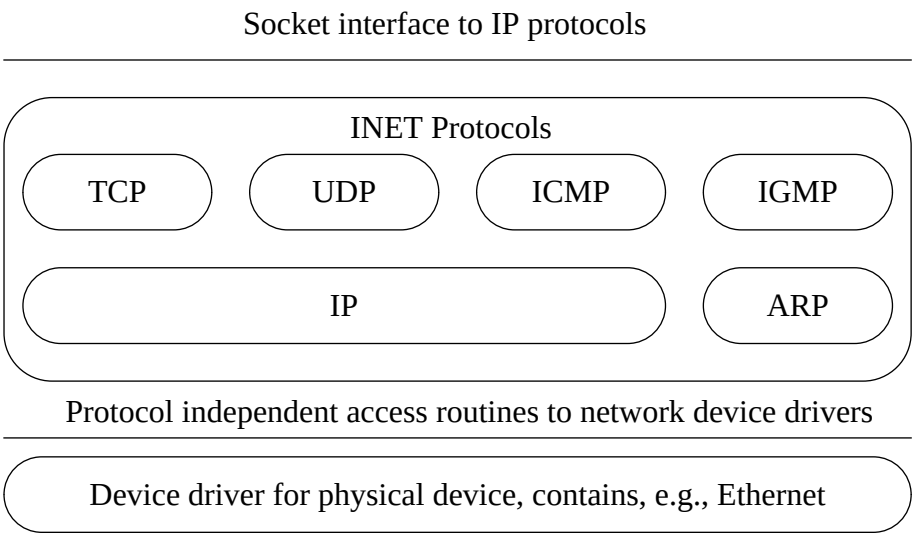

Figure 15: IP Protocols and Ethernet

Figure 15 shows the placement of IP protocols and Ethernet in the kernel. The protocols run over IP (TCP, UDP, ICMP, IGMP) have their handling function registered to this IP entity. These IP protocols need only functions for error control and receiving data packets from the network. When protocol entities need to send data to the network, they either receive it from the upper layer (INET socket interface), or from some other entity (e.g., IP ARP is triggered from the network device driver whenever IP address to link layer address resolving is needed). It is also possible that control messages need to be sent due to internal need of the protocol, e.g., TCP KEEP_ALIVE option causes this kind of operation.

The frames are encapsulated according to the protocol and then passed down to IP's transmit function. When the IP layer receives frames from a lower layer, i.e., in this case the network device driver, it determines to which IP protocol to forward the data, and then calls its handling function.

Address resolution protocol (ARP) is drawn inside the INET protocols. Because IP layer uses IP addresses and Ethernet uses MAC addresses, mapping from IP addresses to appropriate MAC addresses is required. ARP protocol is used in this mapping.

Description of these IP protocols can be found from [RFC768], [RFC 791], [RFC 792], [RFC 793] and [RFC 826].

Ethernet drivers (and other network devices, e.g., SLIP interface) are accessed through protocol independent interface. Network device drivers register to this layer with a structure that contains information about underlying header length, hardware address, functions for handling that device, etc.

Network Device Driver

A network device is an entity that receives and transmits packets. It is usually responsible for an interface to a physical device, e.g., Ethernet card. Another possibility is that it is

implemented in software, like the loopback device, which is used in sending data back to oneself.

Each network device is responsible for transmitting data it receives from upper protocol layers to the physical device and receiving the frames it gets from that device. It is also responsible for control of the hardware, e.g., initialization, stopping and starting of frame receiving. It is also responsible for physical encapsulation of the data, i.e., in the case of Ethernet it fills in the MAC header. Data flow through the network device interface is shown in Figure 16.

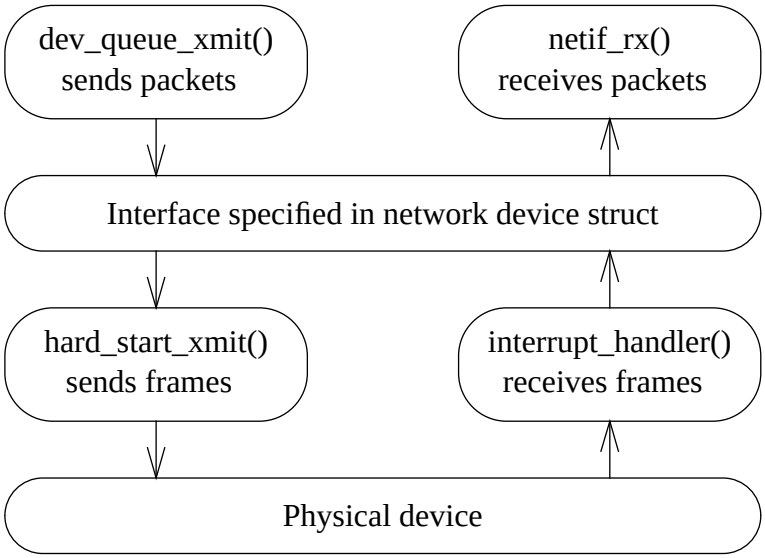

Figure 16: Data flow through Network Device Interface

A device is created by filling in a `struct device` and passing it to the `int register_netdev(struct device *dev)` call. This links the device to kernel's network device tables.

The fields in the `struct device` include the name of the device as a string, e.g., "eth*n*" for Ethernet devices. The device has a set of variables used in network protocol layer. These include `mtu`; largest payload that can be sent over this interface, e.g. 1500 bytes for Ethernet, `pa_addr`; IP address for this interface, `pa_mask`; the IP netmask, etc.

The properties related to the link layer are `hard_header_len`, which tells how much space is reserved to the beginning of the data buffer in memory allocation. The memory allocation for data buffer to be sent is done by upper protocol layers. The network device tells with this variable how much space it needs for its link layer header. Other properties are physical media address, `dev_addr`, and broadcast address, `broadcast`. The length of these physical addresses is stored in `addr_len`.

Each network device has a set of functions it must implement. These include `int (*open)(struct device *dev)` and `int (*stop)(struct device *dev)`, which are used in starting and stopping the network interface, respectively. Initialization, `int (*init)(struct sk_buff *skb)`, should include physical device detection and initialization. It should return an error code if the physical device was not found or if the initialization fails for some other reason.

All the data buffers used by the networking layers are `struct sk_buff`s. The control for these is provided by core low-level library routines available to the whole of the networking. An `sk_buff` is a control structure with a block of memory attached. They are usually buffered through the networking layers in doubly linked lists. The control structure contains information about data it is carrying and where. The attached block of memory is the actual data packet sent/received from the network.

Each network device must have a frame transmit function `int (*hard_start_xmit)(struct sk_buff*, struct device*)`, which is used in sending a network data buffer `struct sk_buff` to the actual medium. If the device cannot achieve this, but thinks that it might be possible later it must set busy flag and return value 1. The upper network layers may choose to retry transmission of the buffer at a later time when the network device is ready to accept data again. If the buffer can be processed, or if the driver thinks that it can't send data for some time, the function must return 0 and free the allocated space.

The data buffer handed to the transmit function already contains the link layer header appended to it. The upper protocol layers take care of calling networking device's header building function `int (*hard_header)(struct sk_buff* skb,struct device* dev, unsigned short type, void *daddr, void *saddr, unsigned len)`. If a header cannot be completed (we know IP address of the destination, but not the MAC address) the protocol layers will attempt to resolve the address necessary. When this occurs, `int (*rebuild_header)(void *eth, struct device *dev, unsigned long raddr, struct sk_buff* skb)` is called. If this can be done (ARP) then it fills in the physical address and returns 1. If the header cannot be resolved, it returns 0 and the buffer will be retried at later time.

There isn't a function in the interface for receiving data from the physical device. This is because the device itself must notice that data is coming in, e.g., from an interrupt. The device allocates memory for the incoming data and then passes it to `netif_rx()`. The network device must deduce from the packet it receives some link layer properties, i.e., in the case of the Ethernet, the suggested protocol and packet type. The packet type can be a broadcast, multicast or unicast packet directed to this endstation or to some other host.

Other properties of the network device are the statistics gathering function, which returns `struct enet_statistics` for the device. This structure contains information about the number of sent and received frames, number of errors occurred, etc.

ATM on Linux 5

5.1. Overview

ATM on Linux is a package which currently contains device drivers for a number of ATM cards (e.g., from manufacturers Zeitnet and Efficient) and the possibility to use both PVC and SVC connections through an ATM socket API. ATM signalling implements a subset of UNI 3.0/3.1. The ATM on Linux also supports IP over ATM with ATMARP[RFC1577] and dynamic address registration using ILMI. The LANE support implemented in this project is also part of this package.

Werner Almesberger from Laboratoire de Réseaux de Communication has been responsible for most of the development work, but there have been several contributions from various sources. The contributions have been either code (Scott Shumate, ILMI address registration), documents (Pedro Paiva, IP over ATM and ATM signalling) or bug fixes. It is not possible to list all people who have affected the ATM on Linux project in this document. A more complete list can be found from the ATM on Linux documentation in the source distribution package.

The ATM on Linux package is described in [ATM Linux].

5.2. Architecture

ATM on Linux is composed of ATM protocol stacks and device drivers in the kernel and components in the user space. The user space applications are, e.g., daemons responsible for ATM signalling and ILMI address registration. Because the protocols used in signalling are rather complex but do most of their work only when connections are established or torn down, it was decided to implement them as a demon in user mode.

The API has been implemented using BSD Socket interface, i.e., the system calls are the same as in the Ethernet case. (Even the entry points to kernel are the same as in INET sockets.) BSD sockets have been extended to support ATM by defining two new socket classes, SVC and PVC classes, to the common socket layer interface.

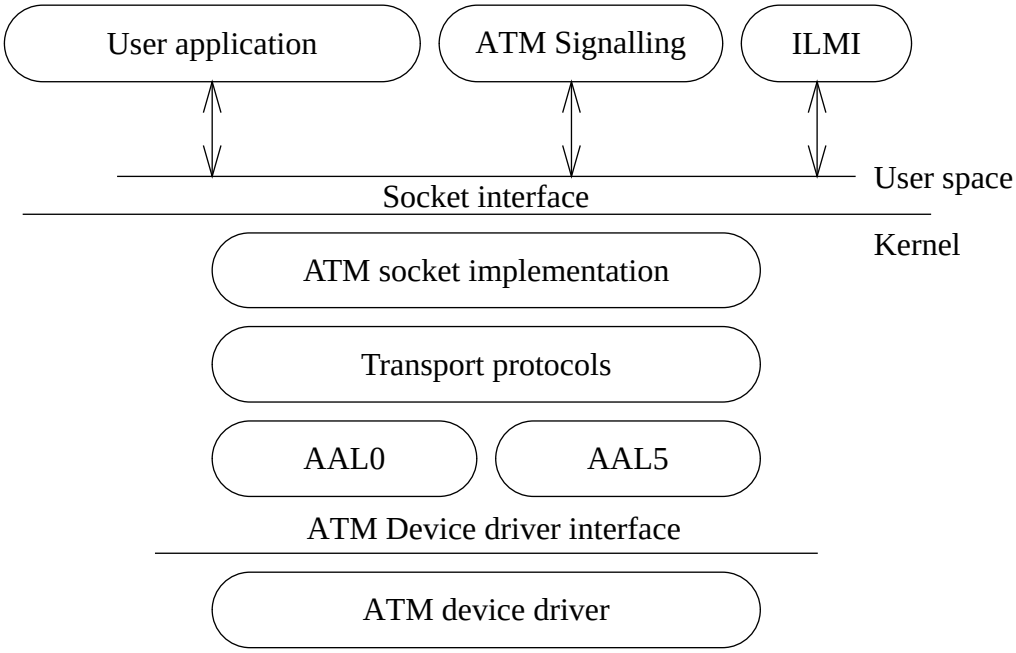

Figure 17: Main components of ATM on Linux

Figure 17 shows the general environment of ATM on Linux. The user application is using ATM socket implementation to connect to remote ATM endstations. The protocol layers in the kernel are the ATM socket layer, transport protocols layer (can be raw data transfer or, e.g., IP over ATM) and ATM Adaptation Layers.

5.3. ATM Device Driver

ATM device driver interface defines a number of device operations, physical operations, and several variables describing the state and information about the device.

An entity acting as an ATM device can be something else than one controlling operation of an ATM Network Interface Card (NIC), e.g., the signalling kernel component acts as an ATM device. Because most of the ATM signalling code runs in user space, this attachment point is needed for communication between these two entities. The signalling daemon creates an ATM socket and then turns it into "signalling control socket" using `ioctl(s, ATMSIGD_CTRL, 0)`system call.

One example of an ATM NIC is Efficient Network Inc.'s 155Mbps MMF (Multimode Fibre) adapter. Its hardware takes care of segmentation and reassembly (SAR), which means that the adapter segments outgoing data into cells and then transmits them to an ATM switch. The adapter also receives cells and reassembles them to PDUs that are processed by the protocol stack.

The device operations include `int (*open)(struct atm_vcc *vcc, int vpi, int vci)` and `void (*close)(struct atm_vcc *vcc)`, which are used to open a VCC with specified VPI and VCI values and to close the VCC, respectively.

There exists also a possibility to read and modify properties of the VCCs with `getsock-`

`opt()` and `setsockopt()`. The function `ioctl()` is also implemented, which has similar properties as the system call with that name. It works as a general tool with several purposes and arguments depending on the device being called. The possibility to write and read directly from card is given with operations `phy_put()` and `phy_get()`.

The driver's only operation which is required from each device is the transmit function `int (*send)(struct atm_vcc *vcc, struct sk_buff *skb)`. It is used for sending data to the indicated VCC, or in the case of a special device, i.e., not a NIC driver, to pass messages to that entity. The data receiving is done with the same idea as in normal network devices, namely that the device driver itself notices when data is coming in. The driver does its magic and passes the `struct sk_buff` to upper protocol layers.

The physical operations operate directly on the ATM card's chip. These are called from the device driver itself.

The variables in ATM device struct are the Endstation Identifier (ESI), which is the hardware address of the card and the valid VPI/VCI range (depends on the hardware). The allocated VCCs for the device are stored in `struct atm_vcc *vccs` and `struct atm_vcc *last` points to the last VCC structure in the table. Statistics and flags of the driver are stored in `struct atm_dev_stats stats` and `unsigned long flags`, respectively.

ATM device driver interface is described in [ATM DD].

5.4. ATM Signalling

ATM Signalling is implemented in the Linux system as a demon process. It constructs the signalling messages according to requests that are forwarded to it from the kernel and decodes the incoming signalling messages from the signalling channel. Signalling messages are transported over AAL5 on well known signalling channel VPI=0, VCI=5.

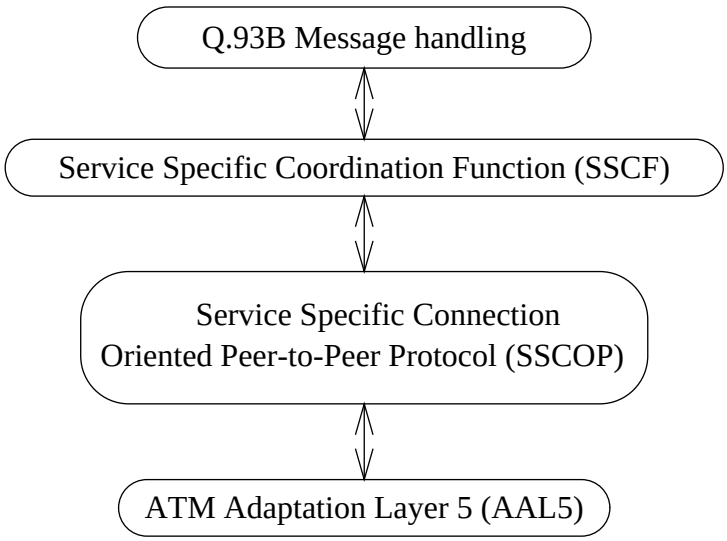

Figure 18: Protocol layers used in ATM Signalling

The signalling daemon keeps track of open connections and local bindings of SAPs. When a user application wants to open a connection to a remote ATM endstation, the signalling demon gets notified of this request and it negotiates the connection open as described in

"UNI Signalling" on page 6. The call requests from the network are forwarded to appropriate SAPs (created with `socket()` and `listen()`).

The application can then `accept()` the connection after which the signalling demon negotiates the connection open.

The protocol layers present in Figure 18 up from AAL5 are implemented in the ATM signalling daemon. All of the properties required in UNI 3.0/3.1 are not implemented.

One example of such limitation is the lack of point-to-multipoint (p2mp) connection support. Specifically, Linux ATM signalling doesn't support acting as the root node for a p2mp connection. Acting as a leaf succeeds though.

5.5. Application Programming Interface

The ATM Forum is specifying the Native ATM Services API specification[ATM Forum API], but it doesn't specify an interface for BSD'ish sockets. Because the ATM API in Linux is based on BSD sockets, it doesn't directly conform to the one specified in that document.

The semantics and design philosophy of the BSD socket interface was tried to follow as closely as possible. The system calls and their parameters have generally the same meaning in the ATM socket implementation.

Address Structures

In order to use ATM through a common socket interface it was necessary to define new address structures. Therefore two new address families, `AF_ATMPVC` and `AF_ATMSVC` were created.

The information given in these structures via the ATM socket calls specifies the remote SAP, where a connection is tried to establish to or the local SAP, when listening incoming calls.

```
struct sockaddr_atmpvc {
  unsigned short sap_family;/*address family,AF_ATMPVC*/
  struct {                  /*PVC address */
    short itf;              /*ATM interface*/
    short vpi; /*VPI (only 8 bits at UNI)*/
    int   vci; /*VCI (only 16 bits at UNI)*/
  } sap_addr;               /*PVC address*/
};

struct sockaddr_atmsvc {
  unsigned    short    sas_family;    /*address    fam-
ily,AF_ATMSVC*/
  struct {                  /*SVC address*/
    unsigned char prv[ATM_ESA_LEN];
                            /*private ATM address*/
    unsigned char pub[ATM_E164_LEN+1];
                            /*public address (E.164)*/
    struct atm_blli *blli;
         /*local SAP, low-layer information*/
```

```
    struct atm_bhli bhli;
         /*local SAP, high-layer information*/
  } sas_addr;               /*SVC address*/
};

struct atm_blli {
  unsigned char l2_proto;/* layer 2 protocol */
  union {
...
  } l2;
  unsigned char l3_proto;/* layer 3 protocol */
  union {
...
  } l3;
  struct atm_blli *next;
      /*next BLLI or NULL (undefined when used*/
      /*in atmsvc_msg)*/
};

struct atm_bhli {
  unsigned char hl_type;/*high layer information type*/
...
};
```

The kernel part of ATM on Linux and the signalling demon use this information when opening connections. The address structure for PVC connections defines, which network card to use together with the VPI/VCI values. The SVC connection forming uses information given in these structures to fill the necessary fields for the SETUP signalling message, when connecting to remote endstations. The listen sockets created with these structures are notified when a SETUP message with appropriate fields is received from the network.

5.6. Call establishment

The use of PVCs is very straightforward. One creates a PVC socket with `socket()` system call specifying `AF_ATMPVC` as the address family. QoS requirements are indicated with `setsockopt()`, and the connection is opened either with `bind()` or `connect()` (behaviour in PVC connection establishment with these system calls is the same). Once the connection exists, data can be exchanged in that VCC. A connection is torn down with `close()`.

The SVC connection establishment is a bit more complex. In the following one possible scenario in SVC connection forming is presented (Figure 19). The signalling demon has created a PVC to communicate with the signalling entity of the network and a special SVC socket, which is used when communicating with the kernel. Description of the message exchange between the kernel part and signalling daemon is presented more thorougly in [ISP].

An application creates a socket with `socket()` system call specifying AF_ATMSVC as the address family. This reserves a socket descriptor and other resources needed for the socket in the kernel. After that the connection QoS parameters are set with `setsockopt(s, SOL_ATM, SOL_ATMQOS, &qos, sizeof(qos))`, where `qos` contains

the traffic descriptors for transmit and receive directions.

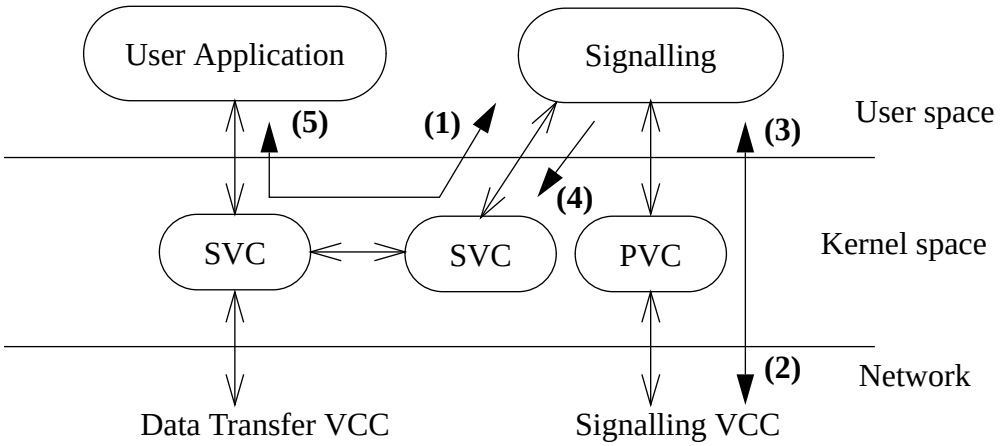

Figure 19: Signalling procedure in the kernel

In the case of active open it is tried to establish a connection to a remote ATM endstation. The local address can be defined with `bind()`. In this case, only local ATM address has any effect in the structure which is passed to kernel. Notification about the binding is passed to the signalling daemon, which checks whether the local binding point is acceptable (Figure 19, 1). The connection establishment is then tried with `connect()` system call (Figure 19, 2).

If the user application uses blocking connect, the operation blocks while the signalling negotiates the connection open with the network's signalling entity. After the connection is ready, i.e., a CONNECT message is received (Figure 19, 3), and a CONNECT_ACK is sent to the network, the local part of the connection is set up, e.g., the ATM adapter is notified about the negotiated VPI/VCI pair (Figure 19, 4) and the application gets notified (Figure 19,5). Now the application is ready to transfer data on this new VCC. In non-blocking connect operation the difference is that the application doesn't "hang" when the signalling tries to establish the connection. It continues operation and gets notified after the VCC is ready.

The connection teardown is done with `close()` as with PVC connections. The difference is that the signalling daemon negotiates the connection teardown with the network.

LE Client 6

6.1. System Design

Functional specification of LAN Emulation is given in [ATM Forum LANE]. This was followed in the implementation as closely as possible.

The LAN Emulation client code is divided into two parts: user space application LAN Emulation Demon called (LED) Zeppelin, and a kernel component. The design philosophy of BSD Unixes encourages the approach where almost everything is included in operating system kernel, i.e., monolithic kernel design. The model used in the LE client, Classical IP with ARP, and the ATM signalling are more “System V” -like.

It would not have been feasible to include all functionality to the kernel, because the LANE protocol is quite complex, i.e., prone to programming errors. The application development is much faster when one doesn’t need to reboot the target machine after testing code which has errors. The LEC is also quite big, so it would bloat the Linux kernel. A user level application also has easier access to the ATM socket interface as described in previous chapter. This interface is needed because the ATM signalling is used and it can be easily done via ATM sockets.

On the other hand, it is not feasible to make LEC solely a user level application and leave only a stub in kernel which would take care of data forwarding. The data packets received by the LEC from network protocols would first get sent to Zeppelin, which would encapsulate those packets and then forward them to an appropriate VCC. This would mean that data from user application using would be copied from memory to memory three times: from the user application to kernel buffers, from the LE client’s kernel component to Zeppelin and then to kernel again. The processing overhead would be enormous, remembering that the weak point in PC performance has been the memory bandwidth.

Another possibility would be to use the `send()` function of VCC’s ATM device to forward data, and keep other parts in Zeppelin. This scenario would also cause too much processing overhead, because it would mean that resolving MAC addresses to ATM addresses would need two messages between the LEC’s kernel part and Zeppelin, a query

and the answer. A small cache between VCC mappings and MAC addresses could be kept in kernel, thus making these questions rarer. This scenario was discarded, because information about every resolution should still be forwarded to Zeppelin. This operation would be needed for correct operation of LE ARP cache ageing.

It was thought that the best approach would be to keep communication between the kernel components and Zeppelin at minimum, and bring as much functionality to Zeppelin as possible. This should minimize processing overhead while making program development easier. It was thus decided that LE control operations (e.g. connection establishment) will be implemented in user space and LE ARP cache and data forwarding would be kernel operations.

To speed up program development, Digital's ATM Starter Kit [DEC LANE] was chosen as the base for the implementation. It is a freely available collection of software components that provide ATM endstation functions. It was designed to be reusable in a wide range of applications and on a wide range of platforms. The intended use of the Digital ATM Starter Kit is in ATM-attached endstations (using ATM adapters), ATM edge-devices, and ATM switches. This package includes an example implementation of LE client.

The most widely used network protocol in Linux is IP and it is usually encapsulated in Ethernet frames. It was thus chosen that the LE client for Linux should only support IEEE 802.3/Ethernet data encapsulation. Other reasons favouring this decision were that DEC's ATM Starter Kit supports only that encapsulation, and test environment did include components capable of only IEEE 802.3/Ethernet encapsulation. If a need should arise for other data encapsulation methods, this decision could be reconsidered.

Only one LE interface is currently supported. This decision was done to simplify the implementation phase. Operation in several virtual LANs simultaneously (several LE interfaces) is under further study.

Copyright of Digital's ATM Starter Kit denies the distribution of the code for a fee. This prohibits source code portions from ATM Starter Kit to be included e.g. in Linux distribution CD-ROMs. It is possible that this leads into a decision to discard all portions of ATM Starter Kit in the future and thus the next versions of LE Linux might differ considerably from the current version.

6.2. System Architecture

The LAN Emulation client components are placed in the Linux operating system as shown in Figure 20.

Communication between Zeppelin and the LEC's kernel component is done via a special ATM socket. Zeppelin creates a SVC socket and then turns this into a communication pathway with a special `ioctl()` call. This system call is also used with different parameters when notifying the kernel components that this VCC is used in LE data transfer.

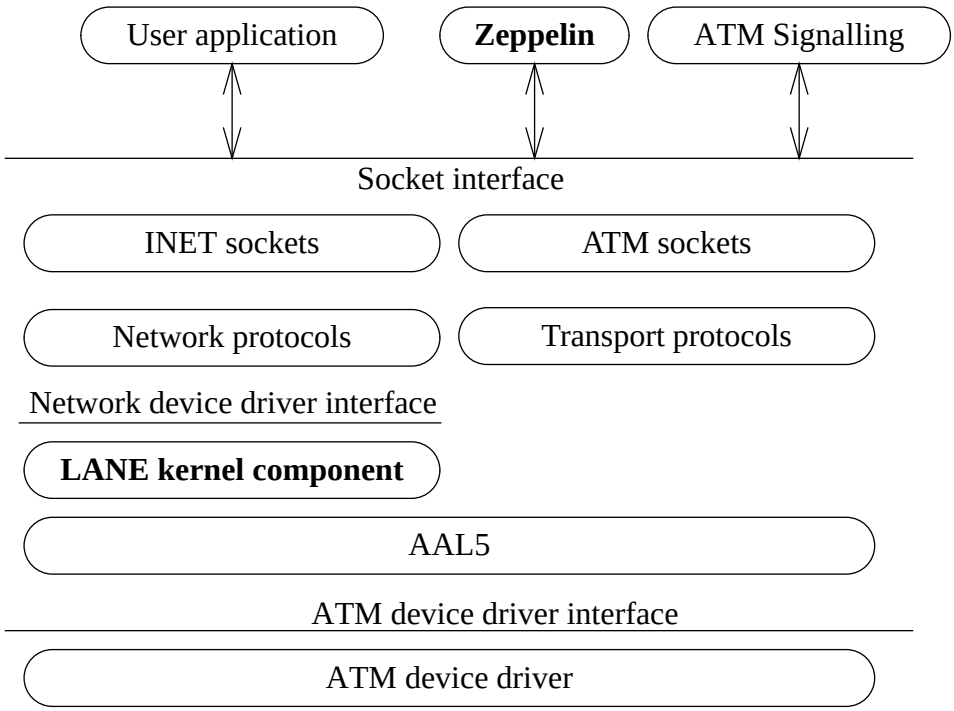

Figure 20: Components of the LE Client in Linux

6.3. LED Zeppelin

Overview

Zeppelin is responsible for control operations needed in LAN Emulation clienthood. It forms the necessary VCCs and receives all the LE control frames and acts accordingly. It also controls the operation of the LEC kernel component. Zeppelin can be woken as:

```
Usage: zeppelin [-c LECS_address]|[-s LES_address] [-e
esi][-n VLAN_name][-m mesg_mask][-l our_address]
```

The ATM address of the LECS or LES can be given as command line parameters. If neither one of these is given, the Well Known LECS address is used. There is no support in Linux ILMI for querying LECS address from ATM network. Here it is also possible to specify the MAC address used in ELAN. If this option is not used, the LEC uses ATM adapter's ESI as its MAC address.

Virtual LAN name is the name of the ELAN. This is used in the LE_CONFIGURE_REQUEST or, if configuration phase is bypassed by giving directly the LES address, in the LE_JOIN_REQUEST. It is possible to limit event messages with the mesg_mask parameter. Explanation of this parameter usage can be found from the source files. ATM address given as command line parameter is used in binding the local Service Access Point. This is required when similar codepoints are used in the machine, e.g., when LE service is operating in this machine.

The demon was based on source code components from Digital's ATM Starter Kit. Most of that code was unneeded or replaced with more fitting parts, e.g. components responsible for data encapsulation (done in kernel), most of the connection management (done auto-

matically by the ATM signalling demon and the kernel socket implementation), and MAC to ATM address mapping cache (done in kernel).

LE client control functions are built around a Finite State Machine (FSM). This model has the advantage that it is quite easy to produce and debug. A state machine has a current state, action functions and input. Based on input (can be, e.g., a LE control frame or timer expiration) the new state is determined from the state transition table. The action functions specified in the state transition table are executed when this state change occurs. The structure of the FSM table entries is the following:

```
typedef struct _sm_table {
  struct _sm_table *p_next;
  UINT16 curr_state;
  UINT16 sm_event;
  UINT16 next_state;
  ACTION_ROUTINE action1;
  ACTION_ROUTINE action2;
  ACTION_ROUTINE action3;
} STATE_TBL_ENTRY;
```

It has fields for the current state (`curr_state`), the input that triggers this state change (`sm_event`), the next state which is reached (`next_state`), and space for three action routines which are executed during a state change. The state transitions for the LEC were taken almost directly from the one presented in [ATM Forum LANE].

The design philosophy of FSM was not followed very strictly, because it would have meant touching Digital's code in more fundamental ways. This was quite difficult, because the approach taken in Linux ATM support implementation is quite different from the one assumed by the ATM Starter Kit, e.g., interface to ILMI support in Linux is almost nonexistent. Drastic changes to the original FSM were tried to avoid to ease the embedding of possible future versions of ATM Starter Kit, which might include, e.g., support for LANEv2.

Modular design methodology was tried to follow where possible. Different tasks needed for the LAN Emulation clienthood were identified and separated (Figure 21). Including source code from Digital's ATM Starter Kit imposed some decisions, but modularity could be achieved in most of the cases.

Revision control of source code files was done with RCS. This tool is freely available and it had been used successfully in previous projects.

Modules

The division of Zeppelin to modules is shown in Figure 21.

The Main component is responsible for starting, resetting and stopping the LE client. It first parses the command line arguments, initializes interface to the kernel components, the LEC Control module and the Connection management. The main loop of the program is also in this module. The socket descriptors are watched for incoming messages and timers for expirations. If program reset is required due to a signal or network reset, this module takes care of actions needed. A network reset occurs, e.g. when an ATM switch, which is connected to this workstation is reset or when the cables from the ATM NIC are removed for some time (the SSCOP layer in the signalling fails).

The LE Control module implements the LEC state machine and handles all LE control packets. It is responsible for the control aspects of LAN Emulation clienthood. The Main module first creates an instance of the LEC Control and then registers itself to it. With these calls the Main module delivers enough information needed for operation on ELAN, e.g., the ATM address of the LECS. The ATM address used in LE Control frames for this client is requested from the module responsible for the ATM address handling.

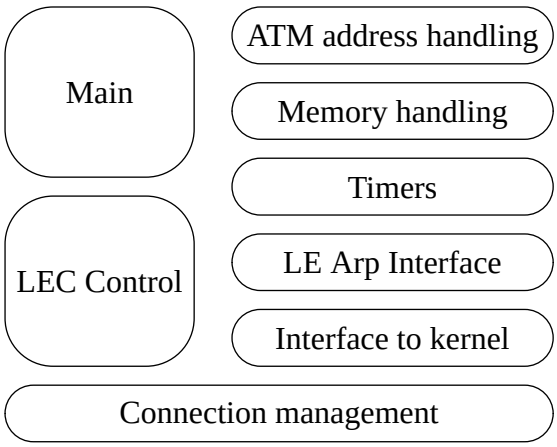

Figure 21: Components of Zeppelin

After these calls the joining to the specified ELAN is started. This joining procedure proceeds as presented in “LAN Emulation Over ATM” on page 10 according to states in FSM’s state transition table. After the joining procedure is over, the LE client is operational and it waits for input.

State changes in the LEC Control module are triggered either by incoming control messages, notifications of established connections, timer expirations, or requests coming from the kernel. These messages might request, e.g., SVC setup to some destination or sending of an LE_ARP_REQUEST.

The Connection management provides an interface for creating and handling of ATM sockets. It keeps track of existing connections, including the connection to the kernel componets of the LEC. Linux ATM socket API is used in this VCC handling. Incoming LE control messages from the network are read from the sockets and passed to the LE Control module. The control messages created by the LE Control module are passed to appropriate VCCs. Information about incoming messages from the kernel component are passed to the module responsible for communicating with the kernel. Incoming calls are accepted and notification about them is passed to the LE Control module and to the kernel component.

The interface to the kernel provides a way to send messages to the kernel and register handlers for incoming messages. The message type is used in determining which handling function to call when messages are received from the kernel.

The interface to LE ARP is specified to be similar with the one in Digital’s ATM Starter Kit. This was done to avoid making changes to original LE Control module which makes most of the calls to this interface. Almost all of the calls to its interface in practice mean that a message to the kernel gets sent.

Required timer support is implemented in the Timers module. Its interface provides calls for allocating, deleting, setting and cancelling timers. When a timer expires, the callback

function associated with that timer is called. This call is done from the Main module, where timer expirations are watched. The next expiration time of the times can be queried and it is returned in a form that can be given directly to `select()` system call. This system call is used in watching for events in allocated sockets and the timer expiration.

Memory handling was implemented to replace system calls `malloc()` and `free()` for other modules. All memory allocation and freeing goes through this module. This was done to aid memory usage tracking of Zeppelin. It is a demon process which will be running for days, if not months, so even a minor memory leak is unacceptable.

The ATM address handling is also implemented as a separate module. It tries to determine when ATM address of this host has changed and then notify appropriate modules about it. Its interface also provides a way for the Connection management module to query it whenever it binds the local SAP, e.g., in creating listen sockets and when trying to connect to other LECs in remote ATM endstations.

6.4. Kernel Component

The kernel part of the LEC is composed of three parts: it acts as a network device, as an ATM device, and it implements the LE ARP cache (Figure 22).

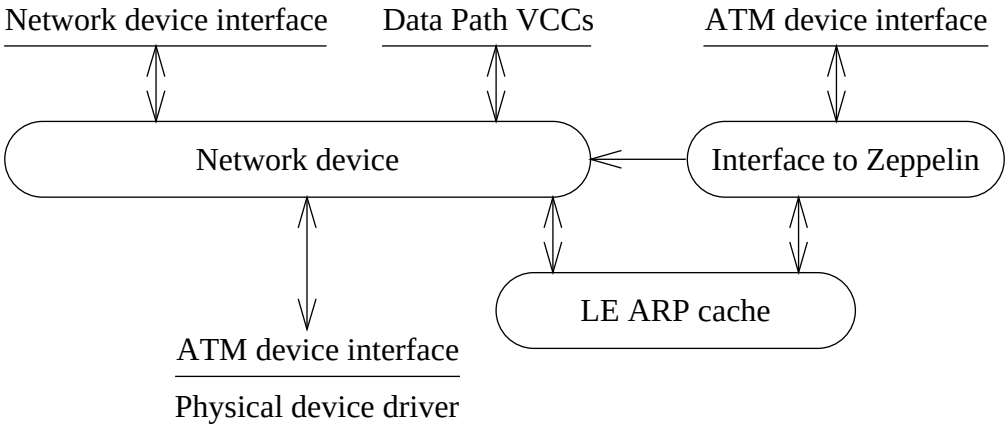

Figure 22: Components of LEC in Kernel

When Zeppelin is started for the first time, it opens a message path to the kernel component. This acts as a start signal to kernel part, and it registers itself as a network device as described in “Networking Layers in the Kernel” on page 23. This network device can be controlled with standard UNIX commands, e.g. `ifconfig(8)`:

```
viulu:~> ifconfig lec0
lec0 Link encap:10Mbps Ethernet HWaddr 00:20:EA:00:0A:E9
     inet addr:193.166.165.5 Bcast:193.166.165.255 Mask:255.255.255.0
     UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
     TX packets:31947 errors:75 dropped:0 overruns:0
     RX packets:61088 errors:0 dropped:0 overruns:0
```

When the LEC network device, “lec0”, starts getting data frames from upper network layers directed to unicast addresses, it tells Zeppelin to send LE_ARP_REQUESTs for those addresses. Zeppelin in turn passes information about responses to these requests to the kernel. VCCs used as data paths in LE are created by Zeppelin, which informs the kernel components about them with a `ioctl()` system call. Data frames received from the net-

working layers are then sent to the appropriate VCCs and frames received from those VCCs are passed either to Zeppelin or to the networking layers, based on frame contents.

The LE ARP cache has three different places where it stores cache entries. When a destination establishes a VCC to this LEC, it is not yet known which MAC addresses that ATM endstation represents. The cache entries containing these VCCs are stored in a linked list, from where the entries are moved when we receive first data frame from that VCC or when the entry expires.

Another linked list is used in storing unused VCC entries for some destination. It is possible that two VCCs exists between LECs, e.g., this might occur when call establishment was started simultaneously in both endstations. Data frames are only sent via the connection for which SETUP message was initiated by the numerically lower ATM address. The other connection is to expire from this list after some time.

Other LE ARP cache entries are stored in a hash table. This lookup method was chosen because it is fast and rather simple to implement. Four least significant bits from the last octet of the MAC address are used as the hash function. The entries in the hash table can be in different states. Behaviour of the entry depends on its state:

- ESI_UNKNOWN, when the next packet sent to entry's MAC address causes a LE_ARP_REQUEST to be sent.
- ESI_ARP_PENDING, when there is no ATM address associated with this entry, but LE ARP protocol is in progress.
- ESI_VC_PENDING, which is entered when there is an ATM address, but there doesn't exists a VCC to that LEC.
- ESI_FLUSH_PENDING, when a VCC to that endstation exists, and the LE flush protocol is in progress.
- ESI_FORWARD_DIRECT, when this entry's VCC can be used in data forwarding.

Timers are used in LE ARP cache aging. For linked lists there are timers for each entry which is to age out, so that unused VCCs expire. The LE ARP entries in the hash table have a common timer triggered function that looks for unused entries from this cache. Operation of Linux LEC is different from the one specified in LANE specification, because in this implementation VCCs that are attached to a MAC address are torn down when the MAC address entry expires. The LANE specification states that the VCCs should time out slower than the ATM address to MAC address mappings. This difference does not affect interoperability.

State of the LE ARP cache can be examined through the `/proc` filesystem (Figure 23). This pseudo-filesystem is generally used as an interface to kernel data structures. Relevant fields in cache entries are shown to ease debugging and to provide information to users about internal state of the LEC. The contents of the hash table and both linked lists are shown.

The network device driver does the LE data encapsulation. The send function queries the LE ARP cache for the VCC which belongs to the MAC address placed in the destination field of the data frame. This address can be either multicast, broadcast or unicast address. Data frames are received by this entity from the ATM NIC. The header of the packet is checked whether it is LE control frame or data frame. LE control messages are passed on to Zeppelin via the VCC it was received, the data frames are queued to network protocols. If this data frame was received from a VCC that hasn't got MAC address associated with it

yet, the MAC address is snatched from the sender field and the LE ARP cache entry is moved to the hash table.

```
viulu:~> cat /proc/atm/lec
MAC   ATM destinationStatus
     Flags VPI/VCI Recv VPI/VCI
0020d4001810 47002300000003030001000201020d400181000 ESI_FORWARD_DIR
ECT   0000 0 91
00603e2fde20 47002300000003030001000201006003e2fde2001 ESI_FORWARD_DIR
ECT   0000 0 90
00c0da30d3e0 47002300000003030001000201006c0da30d3ec00 ESI_FORWARD_DIR
ECT   0001 0 152
0020ea000aa3 00000000000000000000000000000000000000000 ESI_ARP_PENDING
     0000
0020ea0005b3 47002300000003030001000201020ea0005b300 ESI_FORWARD_DIR
ECT   0000 0 193
0020af56e3fd 47002300000003030001000201006c03d30d3ec00 ESI_FORWARD_DIR
ECT   0000 0 152
0800207a4caf 47002300000003030001000201080020 7a4caf00 ESI_FORWARD_DIR
ECT   0000 0 79
ffffffffffff 47002300000003030001000201006003e2fde2201 ESI_FORWARD_DIR
ECT   0002 0 83 0 84
```

Figure 23: LE ARP Data Structures in `/proc` Filesystem

Interface to Zeppelin handles messages coming to the kernel. LEC ATM device interface is used in providing a communication path from the user space to the kernel components. This component also provides a function call for kernel components to send messages to Zeppelin. The message is composed from function call parameters and then queued after VCC's incoming messages.

Interface between Zeppelin and Kernel Components

Both ends in this communication path agree on every detail of the protocol while maintaining some information about other end's state. This well-behaved communication presumes that the messages are delivered in order and that no packets are lost. The different message types are shown in Figure 24.

Message type	Destination	Function
l_set_mac_addr	Kernel	Sets MAC address used in ELAN
l_del_mac_addr	Kernel	Clears the MAC address
l_flush_xmt	Zeppelin	Request initiation of LE flush protocol
l_svc_setup	Zeppelin	Request setup of VCC to destination
l_add_permanent	Kernel	Add permanent entry to LE ARP table
l_addr_delete	Kernel	Delete entry in LE ARP table
l_topology_change	Kernel	Topology flag has changed
l_flush_complete	Kernel	LE flush protocol was finished
l_arp_update	Kernel	MAC address was resolved to ATM

Message type	Destination	Function
l_reset	Kernel	Reset LE ARP cache
l_config	Kernel	Set configurable values for LEC e.g. timers
l_flush_tran_id	Kernel	Sets flush transmission id used in LE flush protocol
l_set_lecid	Kernel	Set LECID for this LEC
l_arp_xmt	Zeppelin	Request sending of LEC_ARP_REQUEST

Figure 24: Messages Exchanged Between Kernel and Zeppelin

These messages can be exchanged in almost any order. The LECID and the MAC address of the interface must be set though before any data exchange can be done.

6.5. Testing

Different testbeds for various software components were set up when needed. The goal was to get the Linux LEC working with LECs from other vendors as soon as possible, so that operation could be tested in a “real” environment.

The part from ATM Starter Kit was tested as whole. Testbeds for other modules were set up as the code for the module was ready. Major changes in the code meant that some testbeds had to be reconstructed. Integration tests for all components, including the kernel part, were done in a real operating environment.

Physical test environment consisted of two ATM switches, Cisco LightStream 100 and ATM Virata VM1000 and an ATM network adapter from Efficient Networks Inc. Operation of signalling and correct data encapsulation was examined by capturing AAL5 frames with ATM analyzer from Adtech. LANE test network was set up as a multivendor environment, where components from 3Com, Efficient Networks Inc, Cisco, Sun, Fore and Zeitnet were tested against Linux LEC for compatibility. The interoperability tests were eventually successful with all of these components.

The software is now part of the ATM on Linux package and is thus widely distributed. This means that the interoperability is currently being tested in several different sites equipped with various ATM hardware configurations around the world. Reports from these sites have proven to be useful in detecting and fixing problems in the software.

LE Service 7

7.1. System Design

The functional specification of LAN Emulation Service is given in [ATM Forum LANE]. This was followed in the implementation as closely as possible.

LE Service consists of three components LECS, LES and BUS. A natural decision was to implement these as three separate demon processes.

Support for both IEEE 802.3/Ethernet and IEEE 802.5/Token Ring type of emulated LANs could be implemented. The difference between these two isn't very big from the LE Service's point of view. This LE Service implementation is mainly aimed to be used in the Linux environment though, so only IEEE 802.3/Ethernet type of LANs will be supported.

ATM address usage in a Linux ATM endstation running both LEC and LE Service with ILMI posed a problem. The Linux ILMI is capable to register only one ATM address, i.e., applications can't request registration of new ATM addresses. Several ATM addresses are needed, because the LEC and the components of LE Service use otherwise similar Service Access Points when listening for incoming calls. Different SAPs are needed so that the ATM signalling can route incoming calls to right entities.

This problem was solved by using local ATM address to all `bind()` system calls (in listen socket creation and in calls to remote ATM endstations) and by modifying the ATM signalling demon's code so that it accepts incoming calls to local ATM addresses that differ from the one registered with ILMI by the selector byte (see "ATM Address Formats" on page 4). IISP routes in the ATM switch might still be required, because the switch doesn't necessarily route calls to an ATM endstation when endstation's registered address differs from the destination ATM address in SETUP message by the selector byte.

This solution thus requires that the user of LE allocates different ATM address for each of these entities.

The LES and the BUS are based on a network oriented FSM implementation by Topi

Miettinen. New action functions and a state transition table were needed as well as an interface to ATM API. The LES was first implemented by emulating ATM connections with TCP sockets. Program was then ported to Sun Solaris 2.4 equipped with Efficient Networks Inc's 155MMF Sbus card using their Software Development Kit (SDK) as the ATM API. The BUS was also implemented during this phase. To port the programs to Linux environment meant that the module responsible for interface to ATM API needed to be changed, and the LECS was written from scratch.

To ease the implementation a decision was made that the LES and the BUS should act only in one ELAN. It is possible to run several instances of these servers in one ATM end-station though. One has to take care of allocation of different ATM addresses for each of these entities.

The LECS needs to return configuration info for several ELANs. It seems natural that, e.g., local private ATM network would have only one LECS, which would divide LECs to emulated LANs according to rules defined by the network administrator.

The design of LE Service was much easier than LE client's due to fact that it can be implemented as a user level application.

7.2. LECS

The LECS is divided into two major parts: configuration file reading and responding to LE_CONFIGURE_REQUESTs. The LECS is waken as follows:

```
lecs [-f configuration_file][-l listen_address][-d]
```

It is possible to define a configuration file which the LECS reads as its database file. If one is not defined, a default file '.lecs_conf' is used. Listen address is used in `bind():ing` the local ATM address. Option `-d` can be used in checking the configuration file contents. The LECS reads its configuration file and dumps what it understood from that file.

When the LECS is invoked, it parses command line parameters, reads the configuration file and creates a listen socket where it waits for connection requests. Fast lexical analyzer generator `flex(1)` was used in creating a component that parses the configuration file.

In the configuration file (Figure 25) first entry is the ATM address of the LECS. Each of the ELANs have a name of the ELAN enclosed in brackets as a starting point. Other properties for emulated LAN (ATM address of the LES, type of the emulated LAN, maximum data frame size and ATM addresses for clients that are defined to be participants in that ELAN) are specified after this. There is a possibility to define wildcards in ATM addresses of the LECs using character 'x'. Default ELAN can be defined with a reserved word 'default'. This is where a LEC is directed if no other matching ELAN definition is found. Example of a configuration file is shown in Figure 25.

Memory reservation is done via a module that tracks memory usage (as is with Zeppelin), i.e., `malloc()`/`free()` are replaced.

The LECS doesn't age out open connections. It is the LEC's responsibility to tear down connection to the LECS when it has received a LE_CONFIGURE_RESPONSE. The LANE specification doesn't say that the LEC must do this, so the LECS is prepared to tear down old VCCs when it has too many open connections to accept new ones.

When the LECS receives a LE_CONFIGURE_REQUEST, it checks that the request is

valid and then looks up a suitable configuration info from its database and sends LE_CONFIGURE_RESPONSE in return.

```
# Our ATM address (should be included to differentiate us from e.g. LEC)
# Must be before ELAN definitions
47002300000003030001000201 0ea000ae905

# ELAN name is inside brackets.
# Parameters below are for Ethernet type LE (Linux & almost all others)
[tut-lane1]
# Address of the LES
LES:=47002300000003030001000201 0ea000ae901
# 802_3 or 802_5 (802_3 = Ethernet)
Type:=802_3
# 1515, 4544, 9234 or 18190 (1516= Ethernet)
Max_Frame:=1516
# ATM address for hosts that are guided to this ELAN. Wildcard is x or X
47002300000003030001000201 0ea0005aax0
47.00230000000303000100020 1.00603E2FDX23.00
47002300000003030001000201 0ea000Xxx00
# Another ELAN, which has empty name
[]
# This directive sets this elan as the default i.e. it will be
# included if LEC's configure request could not be matched to other ELANs
DEFAULT
LES= 47002300000003030001000201 0ea000ae902
Type=Ethernet
```

Figure 25: Example LECS configuration file

Rules for finding ELAN definitions using information provided in LE_CONFIGURE_REQUEST are as follows:

1. Find an entry where ELAN-NAME matches exactly and the ATM address of the LEC is found in ELAN definition. These are to match exactly with the information given in the LE_CONFIGURE_REQUEST. If a ELAN-NAME is found, but the LEC ATM address doesn't match then reject the request with a reason "Permission denied".
2. Choose the first ELAN which matches the request in type of emulated LAN, maximum frame size and ATM address of the LEC.
3. If a matching ELAN was not found, return default ELAN definitions.
4. No match, so the request is rejected with a reason "No Configuration".

7.3. LES and BUS

Modular design methodology was closely followed. Division to modules was present in the original FSM code and because it was quite good, there was no reason to change it to something else.

Each module has two initialization functions, first that does the actual initialization of the module and second doing initialization operations which need other modules, e.g., memory reservation. Common functions for each are also `dump()`, which prints the internal state of the module, and `release()`, which causes the unit to release all allocated resources.

These servers are event driven: connection requests, messages, signals and timer expirations are converted into events. An event causes a call to appropriate event handler. These

event handlers act as a base for a Finite State Machine. Format of the state transition table in this FSM is a bit different from the one in Zeppelin. Current state and next state are specified, but there exists only one action function. The state transition table and action functions were implemented as specified in [ATM Forum LANE].

The same modules appear in both LES and BUS in such way that object files can be compiled from the same source files (Figure 26). The module containing the state transition table and action functions is the only difference.

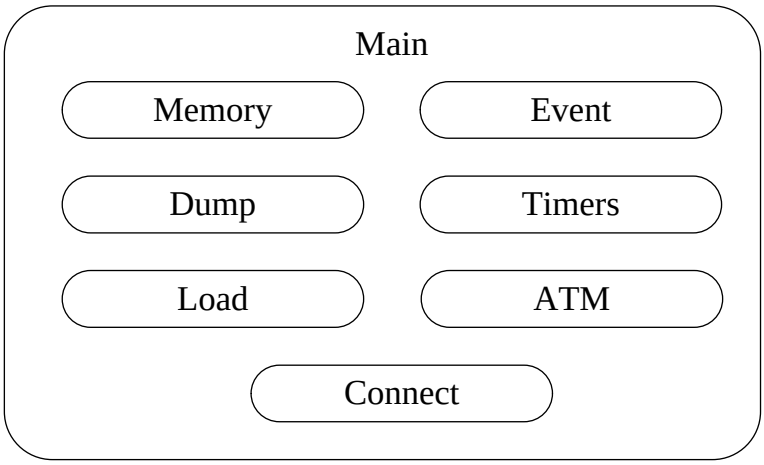

Figure 26: Modules in LES/BUS

The LES can be waken as follows:

```
Usage: les [-d module]... [-m module]
```

It is possible to set debugging for modules in command line parameters. Parameter `-d` followed with a module name enables all debugging messages for that module. Parameter `-m` followed by a module name enables memory debugging for that module. This means that the memory allocation/freeing done by that module is traced. The parameters for invocation of the LES were shown, but same parameters apply to BUS as well.

Memory module controls the reservation and freeing of dynamic memory. This is done so that one is able to follow memory usage of modules and check that they free all allocated memory when restarting. Memory usage debugging for a module can be controlled either by command line parameters or by setting `memdebug` variable in the initialization file for that module to ‘True’ or ‘False’. The reservation and freeing of memory in other modules is done via the interface provided by this module.

Dump module contains all debug message producing functions. These are collected to a single module so that debug messages from modules can be controlled easier with command line parameters or with module’s `debug` variable in initialization file.

Load unit reads the initialization file ‘.lanevars’ from the current directory when LES/BUS is started (Figure 27). The initialization file contains the name of each module in brackets followed by variable definitions for that module. The definitions are of form variable=value, where the value can be either an integer, a truth value (True/False), a string enclosed in double quotes (“string”) or an ATM address in hexadecimal format. Variables that can be set are the aforementioned `debug/memdebug` for each module and variables S1-S6 as defined in the LE specification. These are e.g. type of the ELAN these LE servers

are responsible for (currently only “802.3”) and ATM addresses of the LES and the BUS. These ATM addresses are used in LE control messages sent by the LES and also in binding local SAPs. Additional variable for Connect module, ELANNAME, defines the name of the ELAN these servers control. The LES checks that this name is used in LE_JOIN_RESPONSEs the LECs send to it.

```
[main]
#memdebug=True
#debug=True
[load]
memdebug=True
debug=True

[conn]
debug=True
#S1, LE Server’s ATM address
S1=:47:00:23:00:00:00:03:03:00:01:00:02:01:00:20:ea:00:0a:e9:01
#S2, LAN Type
S2=”802.3”
#S3, Maximum Frame Size
S3=1516
#S4, Join Timeout, s
S4=15
#S5, Maximum Frame Age, s
S5=6
#S6, BUS Atm address
S6=:47:00:23:00:00:00:03:03:00:01:00:02:01:00:20:ea:00:0a:e9:02 #viulu
ELANNAME=”asdf”
```

Figure 27: Example LES/BUS configuration file

Event module is where signals, connection requests, incoming data frames and timer expirations are noticed and converted into events. After this conversion, an appropriate event handler is called. Events in socket descriptors and timer expirations are waited in `select()` system call.

Timers module provides an interface with which timers can be allocated, freed, set and acknowledged. When a timer is set to expire after certain interval, it must be acknowledged or a timer expiration event will occur. Signal handlers are also located in this module. Interface provides a call with which the soonest timer expiration time can be fetched. This call is done by the Events module when it is about to call `select()`.

ATM unit has an interface through which the server can create a VCC to a destination address or to create a listen socket. These operations are done with the ATM socket API. Earlier versions of these servers contained a module which emulated ATM API with TCP sockets (when there weren’t any ATM APIs available) and later Efficient Networks Inc.’s SDK calls in the Solaris version. Local SAP definitions use ATM addresses given in the configuration file mentioned above.

Connect module contains connection management together with the FSM and required databases. This module is different in the LES and BUS. The module is quite big in the LES, so it is divided into three files: the FSM and the connection management in one file, database handling functions in second, and response frame forming in third. The BUS operations are more simple.

The BUS maintains only information about currently open connections, while the LES has databases for open connections, LECIDs in use (these are assigned by the LES), registered MAC/ATM address pairs, and LECs registered as proxies. The information from databases is used in checking incoming LE control frames for errors, forming responses to various queries, e.g. LE_ARP_REQUEST, and deciding to which LECs the control frames are sent. Currently these databases are implemented as linked lists. It is expected that LE control frames are not received very frequently and that there usually aren't a large number of LECs joined to a particular ELAN, so performance should be acceptable. The implementation can be changed, e.g. to hash table, if this is not so in practice.

Main module starts up the servers. It initializes other modules by calling their initialization functions. It also parses the command line arguments. The server reset is done by calling release functions for each of the units and then initializiting the units again.

7.4. Testing

Testbeds were constructed for each of the modules when needed. First integration tests were done with TCP implementation of LES. Functionality was then tested with pseudo clients. The implementation of the LES/BUS for Solaris equipped with Efficient Networks Inc.'s SBUS cards was tested with LECs from that same vendor.

The current implementation of the LE Service as whole was tested in TUT with Linux LECs and LECs for Efficient Networks Inc.'s SBUS MMF adapter. Other ATM network configurations have been tested in other sites and in all cases the LES was interoperable. This was quite a surprise, because the lack of p2mp connection support was thought to mean problems when communicating with LECs from most of the vendors.

Conclusions 8

ATM has ensured its foothold in computer networks. ATM in Wide Area Networks (WANs) has proven itself worthy and the number of ATM networks as backbones is increasing. Transmission of voice and video has also proved to be possible and thus standardization efforts continue in all these sectors. It is still to be seen whether ATM will be used in replacing the dedicated networks reserved for voice/video transmission.

Future of ATM in LANs is also unclear. Improvements in current data networking technologies are casting doubts on choice of ATM as the one and only. Whether ATM will prove to be superior over the competing technologies might depend on future network applications. For example, applications transferring audio and video streams will profit from features of ATM (i.e. Quality of Service).

Success of ATM depends partly on its ability to carry networking protocols, especially IP, over it. IP over ATM [RFC 1577] and LAN Emulation are currently the standardized methods of doing this.

Internet Engineering Task Force and the ATM Forum are currently developing other methods of transporting network protocols and enhancing current specifications. For example, LAN Emulation v2.0 specification correcting some of the shortcomings of LE v1.0 will soon be available.

The goal of the project was to implement LAN Emulation v1.0 support for Linux environment by implementing the LAN Emulation User Interface (LUNI). This was achieved with components that act as a LE client and a LE service. LE servers were implemented as user level processes using ATM socket API provided in ATM on Linux. LE client was divided into kernel part and user level process.

The project was successful. The developed LE support is now part of the ATM distribution for Linux. This distribution channel has made it possible that LE components are now in field trial on several sites.

Work with LE in Linux continues. Bug fixes and new features will be implemented, e.g.,

point-to-multipoint support as soon as ATM support for Linux allows this. Implementation of new specifications for network protocol transportation in the Linux environment might also be quite interesting.

References

[ATM DD] Werner Almesberger, “Linux ATM Device Driver Interface Draft, version 0.1”, ftp://lrcftp.epfl.ch/pub/linux/atm/docs, February 1996.

[ATM Forum API] ATM Forum, “Native ATM Services: Semantic Description Version 1.0”, February 1996.

[ATM Forum LANE] ATM Forum, “LAN Emulation Over ATM - Version 1.0”, ATM Forum, January 1995.

[ATM Forum UNI31] ATM Forum, “ATM User-Network Interface Specification Version 3.1”, ATM Forum, September 1994.

[ATM Linux] Werner Almesberger, “ATM on Linux”, ftp://lrcftp.epfl.ch/pub/linux/atm/papers, March 1996.

[BSD] Maurice J. Bach, “The Design of The Unix Operating System”, Prentice-Hall, 1986.

[CCITT AAL5] CCITT Document TD-XVIII/10, “AAL Type 5, Draft Recommendation text for section 6 of I.363”, January 1993.

[DEC LANE] Theodore L. Ross, Douglas M. Washabaugh, “Digital ATM Starter Kit for LAN Emulation Developers v1.0”, http://www.networks.digital.com:80/npb/html/atm-starter-kit.html, 1995.

[Ethernet] Digital Equipment Corporation, Intel and Xerox, AA-K759B-TK, “The Ethernet - A Local Area Network”, November, 1982.

[IEEE802.3] ISO / IEC 8802-3: “ANSI/IEEE Std. 802.3 Information Processing Systems - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and

Physical Layer Specifications.”.

[IEEE802.5] ISO / IEC 8802-5: “ANSI/IEEE Std. 802.5: Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - LAN/MAN-type Specific Requirements - Part 5: Token Ring Access Method and Physical Layer Specifications.”.

[ISP] Werner Almesberger, “Linux ATM internal signalling protocol Draft, version 0.0”, ftp://lrcftp.epfl.ch/pub/linux/atm/docs/, August 1996.

[KHG] Michael Johnson, “Linux Kernel Hacker’s Guide”, http://www.redhat.com/HyperNews/get/khg.html.

[LINUX] Matt Welsh & Lar Kaufman, “Running Linux”, O’Reilly, February1995.

[POSIX] 1003.1-1988 INT IEEE Standard Interpretations of IEEE Standard Portable Operating System Interface for Computer Environments (IEEE Std 1003.1-1988).

[Q93B] ITU-T draft Recommendation Q.2931 "B-ISDN User-Network Interface Layer 3 Specification for Basic Call/Bearer Control", March 1994.

[RFC1237] R. Colella, E. Gardner, R. Callon, “RFC 1237: Guidelines for OSI NSAP Allocation in the Internet”, July 1991.

[RFC1483] Juha Heinänen, “RFC 1483: Multiprotocol Encapsulation over ATM Adaptation Layer 5”, July 1993.

[RFC1577] M. Laubach, “RFC 1577: Classical IP and ARP over ATM”, January 1994.

[RFC768] J. Postel, “RFC 768: User Datagram Protocol”, August 1980.

[RFC791] J. Postel, ed., “RFC 791: Internet Protocol - DARPA Internet Program Protocol Specification”, September 1981.

[RFC792] J. Postel, “RFC 792: Internet Control Message Protocol - DARPA Internet Program Protocol Specification”, September 1981.

[RFC793] J. Postel, “RFC 793: Transmission Control Protocol - DARPA Internet Protocol Specification”, September 1981.

[RFC826] David Plummer, “RFC 826: An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48-bit Ethernet Addresses for Transmission on Ethernet Hardware”, November 1982.

[STP] IEEE, “IEEE Std 802.1D - 1990 - IEEE Standards for Local and Metropolitan Area Networks:Media Access Control (MAC) Bridges”, 1991.