Integration and Performance Testing of IP over ATM

Alice L. Flower

June 18, 1999

Clemson University
Clemson, SC 29634-1906

Submitted to the graduate faculty of the Department of Computer Science in partial fulfillment of the requirements for the degree of Master of Computer Science.

Abstract

Asynchronous Transfer Mode (ATM) offers the benefit of high speed multimedia transfer of information. However, even if the use of ATM technology spreads, it will not instantly replace all the existing Local Area Networks (LANs) based on other network protocol stacks, such as TCP/IP. The need to interface existing connectionless LANs with connection oriented ATM networks and the need to run TCP/IP applications on an ATM network have led to the development of several software packages that provide the necessary translation services.

This paper deals with two such packages: Classical IP over ATM and LAN Emulation. A brief description of the various protocols involved is discussed, followed by a description of the data transfer process for one of the two packages, and a report on the problems encountered and solutions found in integrating them into an existing ATM network. Performance measurements are provided and an overall evaluation of the two methods is included.

Contents

1 Introduction **1**

2 Configuration **2**

2.1 Hardware 2

2.2 Software 2

3 Asynchronous Transfer Mode **2**

3.1 Overview 2

3.2 ATM Connections 3

3.3 Protocol Reference Model 4

3.4 UNI/NNI 5

3.4.1 Signaling 6

3.4.2 ILMI 6

3.5 ATM Addresses 7

3.6 Maximum Transmission Unit 7

3.7 ATM on Linux 8

4 Classical IP over ATM **8**

5 LAN Emulation **9**

5.1 Components 10

5.2 Connections 11

6 Data Transfer in Linux TCP/IP over ATM **11**

6.1 TCP/IP Protocol Initialization 12

6.2 ATM Device Initialization 15

6.3 Socket Creation 18
6.4 Socket Connection 22
6.5 ATMARP Initialization 23
6.6 Data Transfer 24
6.6.1 Send Side 24
6.6.2 Receive Side 28

7 Software Integration **32**
7.1 CLIP Using PVCs 32
7.2 CLIP Using SVCs 35
7.2.1 Connection Timeouts 36
7.3 LANE Using SVCs 36
7.3.1 ATM Address Errors 36
7.3.2 Linux Code Bugs 38
7.3.3 Other Miscellaneous Oddities 39

8 Performance **39**

9 Evaluation of CLIP and LANE **42**

10 Future Research **44**

11 Conclusions **44**

A Acronyms **47**

1 Introduction

Asynchronous Transfer Mode (ATM) is a network protocol stack providing high speed multimedia transfer with support for guaranteed Quality of Service (QoS), including bandwidth, network delay, and delay variation. ATM may someday become the dominant form of transport throughout the Internet, but there will be a continuing need to use the extensive existing TCP/IP applications such as ftp, telnet, rlogin, and ping in an ATM environment. There will also be a need to connect Ethernet, Token Ring, or other types of Local Area Networks (LANs) with ATM networks.

There are several problems that must be addressed before TCP/IP applications can run on an ATM network. ATM is connection oriented while IP is connectionless. Ethernet and Token Ring exist in a shared medium environment that makes multicast and broadcast traffic easily accomplished. The connection oriented nature of ATM makes such traffic more difficult. Last but not least of the problems is that ATM addresses are different from IP addresses.

A number of solutions to interfacing IP with ATM have been proposed. Two competing solutions discussed in this paper are are Classical IP over ATM (CLIP) and LAN Emulation (LANE). Both deal with the problems of how to encapsulate IP packets on ATM, and how to resolve IP addresses into ATM addresses.

The main focus of this paper is the integration of the Linux implementations of LANE and CLIP with the existing hardware and software configuration. In addition documentation is provided which describes the software supporting data transfer in Classical IP over ATM. An evaluation of CLIP versus LANE considers not only performance measurements but also ease of network administration and relative reliability of the software.

The remainder of this paper is organized as follows: Section 2 describes the hardware and software platform used in this project. Section 3 provides a general overview of ATM and an introduction to services used by IP over ATM such as signaling. Classical IP over ATM is covered in Section 4. An introduction to LAN Emulation is presented in Section 5. Section 6 describes the process of transferring data in CLIP from the perspective of the Linux code.The problems encountered and the solutions found in integrating CLIP and LANE are documented in Section 7. Performance measurements are presented and discussed in Section 8. Section 9 looks at the relative merits and shortcomings of CLIP and LANE. Suggestions for future research are covered in Section 10. Section 11 presents conclusions.

2 Configuration

2.1 Hardware

Our hardware platform consisted of two Gateway 2000 personal computers connected to an IBM 8285 switch. The PCs were a 180 MHz Pentium Pro and a 100 MHz Pentium system. The Network Interface Cards (NICs) were IBM IBM Turboways 25s, which are based on the IBM APE25PCI (ATM Protocol Engine, 25 Mbps, Peripheral Component Interconnect) interface controller. The controller provides a 25.6 Mbps transmission rate.

2.2 Software

We used the Slackware Linux 2.0.25 operating system. Linux is a POSIX compatible implementation licensed under the GNU Public License. ATM support for Linux is available in a separately distributed software package also covered under the GNU Public License. The ATM utilities used are version 0.31. The device driver code was written by Drs. J. Westall and R. Geist of Clemson University.

3 Asynchronous Transfer Mode

3.1 Overview

ATM is a network architecture developed to provide high speed transfer of voice, video, and data. The original standards documents were produced by the Consultative Committee for International Telegraph and Telephone (CCITT) which has since been renamed The International Telecommunications Union-Telecommunications Standardization Sector (ITU-T). Other standards organizations involved in ATM are the Internet Engineering Task Force (IETF), and the ATM Forum. The ATM Forum is not strictly speaking a standards organization but rather a consortium of vendors and other parties with a financial interest in the development of ATM technology. The ATM Forum was formed in 1991 with the goal of accelerating the process of developing standards. Goralski [9] describes ATM Forum specifications as "implementation agreements binding only on their own members." The presence of multiple bodies developing ATM standards inevitably leads to multiple and competing protocols.

The CCITT Blue Books define ATM as "a multiplexing technique in which a transmission capability is organized in undedicated slots filled with cells with respect to each application's instantaneous real need" (CCITT I.113, p.2). [9] In ATM, the variable length packets of today's LANs are replaced with short fixed length cells. This difference is mainly visible at

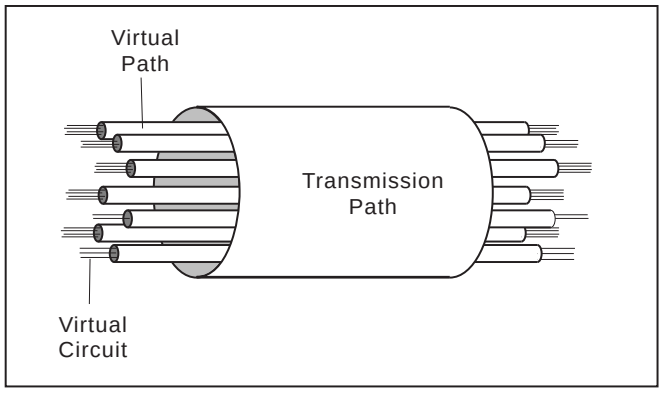

Figure 1: Model of ATM VCC.

the lower layers of the protocol stack because the higher layers usually use larger units of data which are transformed to and from cells by the ATM Adaptation Layer. The ATM Adaptation Layer will be discussed in Section 3.3 below. An ATM cell is 53 bytes long, of which 48 bytes are payload and 5 bytes are header.

The asynchronous part of ATM refers to the ability of hosts on a network to send data "on demand". A synchronous transfer network is channelized. Each host is assigned bandwidth based on position: either a frequency band in frequency division multiplexing (FDM) or a time slot in time division multiplexing (TDM). The bandwidth is reserved for that host whether it has anything to send or not. In fact, if a source has no data to send, it must transmit a special idle bit pattern to keep the source and destination synchronized. With ATM, the network is unchannelized. Data may be sent whenever there is need and only when there is need. Synchronization through the idle pattern is not necessary because every cell contains enough information, including destination, to identify the cell.

3.2 ATM Connections

An ATM network is connection oriented; before data can be sent from one host to another, a virtual circuit must be set up. This is analogous to a telephone system in which a call must be connected before conversation can begin. A virtual circuit, also called virtual channel connection (VCC) does not represent a physical path from source to destination but rather a particular route that is chosen through all the switches along the way and is stored in each routing table. All cells sent after connection set-up will follow the same path to the destination. ATM does not guarantee delivery, but because it is connection oriented, the order of delivery is guaranteed. Cells may be lost, damaged, or discarded but they will never arrive out of order.

VCCs are designated by three numbers: an interface number, a virtual path identifier (VPI) and a virtual circuit identifier (VCI). These identifiers are hierarchical: many VPIs may be supported by one interface, and many VCIs belong to a virtual path. A VPI represents a

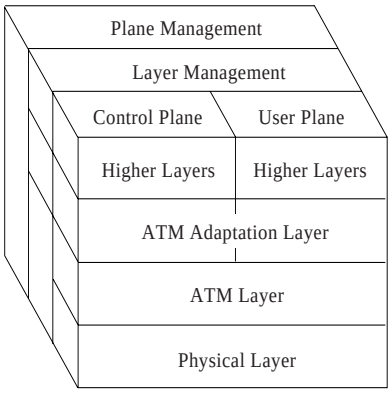

Figure 2: ATM Protocol Reference Model.

group of VCIs with a common source and common destination. Figure 1 shows a conceptual model of the relationship between VPIs and VCIs. A VCC is generally designated Interface.VPI.VCI, eg. 0.0.32. Frequently, the interface is assumed to be zero and the circuit is identified simply by the last two fields, 0.32. Virtual circuits are unidirectional, but pairs of VCs can be created at the same time using the same identifiers, making them effectively full duplex. Permanent virtual connections (PVCs) are set up manually by a network administrator and represent a permanent connection between end points. Connections which are set up and torn down dynamically as needed by the application are called switched virtual circuits (SVCs).

3.3 Protocol Reference Model

The protocol reference model for ATM contains not only layers similar to the TCP/IP protocol stack, but also several planes (see figure 2). The planes are a way of visualizing the categories of services provided by the network. The user plane is concerned with transportation of data, flow control, and error correction. The control plane takes care of connection management including signaling. The management plane, which is subdivided into plane management and layer management, is responsible for managing resources and interlayer coordination.

The lowest layer in the protocol stack is the physical layer. This layer deals with such physical medium issues as bit timing, voltages, and with the conversion of a stream of cells into a bit stream on the sending end and conversion of a bit stream back into a sequence of cells on the receiving end. The physical layer is also responsible for the generation and verification of the header checksum.

The ATM layer manages the cells. The cell header is generated and processed here. Also, the incoming VCC is translated into the corresponding outgoing VCC. Cells from different sources bound for the same outgoing link are multiplexed. Flow control also resides here.

The physical layer and the ATM layer typically reside in the switch, but the ATM Adaptation

layer (AAL) is present in the end system. Responsibilities of this layer include segmentation of packets from higher layers into cells, reassembly of cells into packets at the destination, and dealing with quality of service (QoS) requirements of the connection. Different classes of traffic have differing QoS requirements. Table 1 summarizes the characteristics of the four classes of service defined by ITU.

	Class A	*Class B*	*Class C*	*Class D*
Timing	Real time		None	
Bit Rate	Constant	Variable		
Mode	Connection Oriented			Conn-less

Table 1: AAL service classes

Protocols AAL1 through AAL4 were designed to carry Classes A through D traffic respectively. Eventually, AAL3 and AAL4 were found to be not sufficiently different to warrant separate protocols and were combined into AAL3/4. AAL1 through AAL3/4 protocols were standardized by the ITU but were subsequently found unacceptable for computer data traffic. AAL5 was designed for this purpose and was accepted by the ATM Forum. It should be noted that the ATM Forum now recognizes five service categories rather than Classes A through D. The categories are:

ABR	Available Bit Rate
CBR	Constant Bit Rate
NRT-VBR	Non-Real Time - Variable Bit Rate
RT-VBR	Real Time - Variable Bit Rate
UBR	Unspecified Bit Rate

AAL5 is intended for connection-oriented, variable bit rate, timing insensitive data transfer.

3.4 UNI/NNI

ATM networks may be one of two types: public or private. Public ATM networks are those controlled by a public service provider such as a telephone company. An example of a private ATM network is a campus network or a corporate network. Figure 3 shows the types of ATM interfaces that are possible. An interface between a user and an ATM switch is called a user network interface (UNI). The interfaces between switches are network network interfaces (NNIs) or network node interfaces. The ATM Forum has published standards for public and private UNI [6]. Currently, the only way to interface a private network ATM switch with a public ATM network switch is through public UNI. Efforts to develop private NNI are ongoing.

The UNI protocol defines not only the signaling protocols, but also such things as the physical media, bit rates, line codings, and configuration protocols. The most common version of UNI currently in use is 3.1 but version 4.0 has been implemented by some providers of ATM protocol stacks. UNI 3.0 was used in this project almost exclusively.

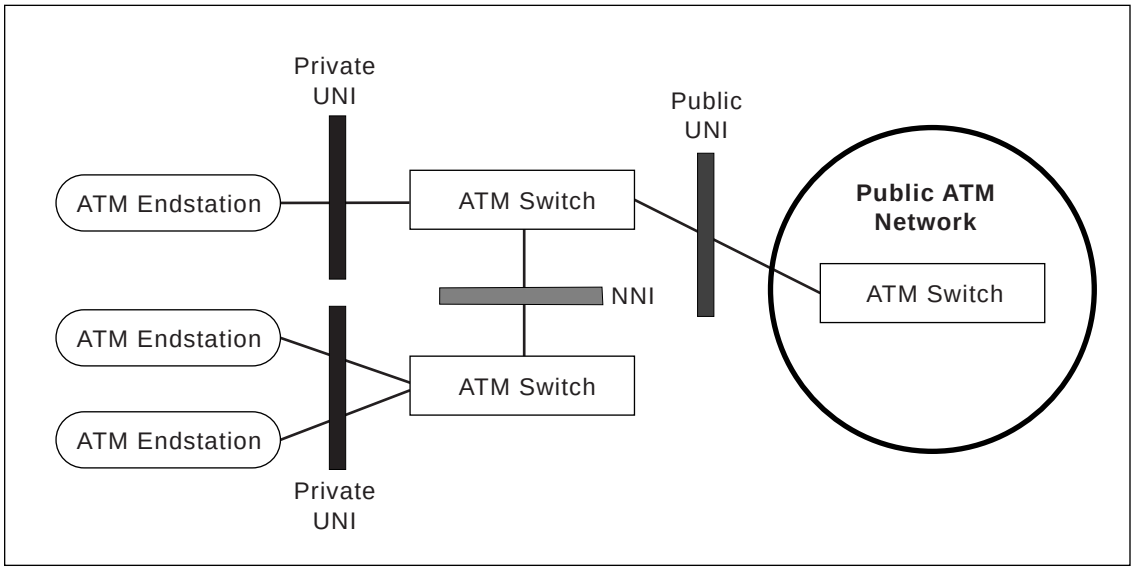

Figure 3: ATM Network Interfaces.

3.4.1 Signaling

Connection management of SVCs requires communication between entities in the network. The procedures for accomplishing communication are called UNI signaling. The ATM Forum User-Network Interface Specification [6] requires that connection establishment take place using a common out-of-band channel, specifically VPI = 0, VCI = 5. In Linux, UNI signaling is handled by the `atmsigd` demon. RFC 1755 [14] gives the specifications for signaling in IP over ATM.

3.4.2 ILMI

ATM Management plane procedures have not yet been formalized. Interim Local Management Interface (ILMI) was designed to provide the necessary services until such time that formal standards are available. However, in recognition of the fact that ILMI is becoming a de facto standard, it is sometimes now called Integrated Local Management Interface. The ILMI specifications are described in [6]. Using the Simple Network Management Protocol (SNMP) and an ATM UNI Management Information Base (MIB), ILMI provides ATM user devices with status, configuration, and control information. The types of ATM related information that can be communicated using ILMI range from the type of coaxial cable used to the number of VCIs allowable on the interface, but the Linux ILMI demon `ilmid` seems to serve the sole purpose of communicating the network prefix and registering the end system's ATM address with the switch. Communication with ILMI takes place using the reserved connection identifier pair, VPI = 0, VCI = 16;

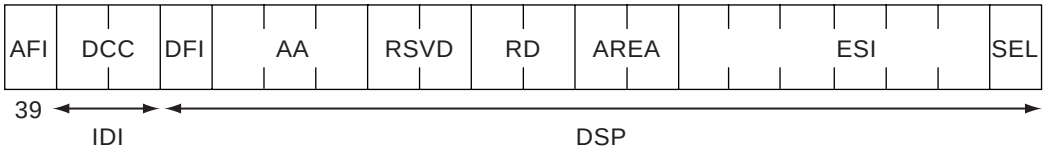

Figure 4: ATM Data Country Code Address Format.

3.5 ATM Addresses

Two ATM addressing schemes are specified by the ATM Forum [6]. ATM addresses may be encoded as an OSI Network Service Access Point (NSAP) type address or an E.164 address, which contain telephone numbers. All private networks are required to accept addresses of either sort. The length of the address is always 20 bytes, of which 13 bytes comprise the network prefix and 7 bytes are the end system part. Although we do not have officially assigned ATM addresses for our test bed, we created addresses using the Data Country Code (DCC) format (see figure 4), which is an NSAP type address [5].

Field		Meaning
AFI	Authority and Format Identifier	Identifies authority allocating the DSP part of the address
IDI	Initial Domain Identifier	Specifies network addressing domain
DSP	Domain Specific Part	Part of address under control of IDI
DCC	Data Country Code	Country in which the address is registered
DFI	Domain Specific Identifier	Specific format of remaining fields
AA	Administrative Authority	Entity that allocates addresses for remainder of domain specific part
RSRVD	Reserved for future use	Coded to 00 00
RD	Routing Domain	Specifies unique routing domain
AREA	Area	Identifies unique area with RD
ESI	End System Identifier	Uniquely identifies end system
SEL	Selector Byte	Used by the end system for selecting an end point

Of primary importance to this project are the End System Identifier and the Selector Byte. The end system identifier is a Medium Access Control (MAC) address. It uniquely identifies the host. The selector byte is not used by the ATM network for routing. The host, therefore, can assign different values to the SEL to distinguish up to 256 addresses for the same ESI.

3.6 Maximum Transmission Unit

The Maximum Transmission Unit (MTU) is the largest frame a data link layer can handle without fragmentation. Typical MTUs range from 65,535 bytes for Hyperchannel to 576 bytes for X.25. For Ethernet, the limit is 1500 bytes. If a datagram larger than the link layer

MTU is sent, the protocol has to fragment it. The default MTU for ATM was chosen with a view to compatibility with existing Internet protocols. RFC 1626 [3] specifies the default MTU for ATM to be 9180 bytes to conform to the IP MTU over Switched Multimegabit Data Service (SMDS). It was felt that this choice would facilitate interoperability and reduce the incidence of IP fragmentation. Smaller MTUs are permitted if both parties agree to use some other value.

3.7 ATM on Linux

The Linux software package supporting ATM includes components for handling signaling, ILMI functions, IP over ATM, and device drivers for several ATM cards. The primary author of this package was Werner Almesberger [1] [2] of the Laboratoire de Réseaux de Communication at the Swiss Federal Institute of Technology in Lausanne. The ATM protocol stack and device drivers reside in kernel space, but the components which handle signaling, address registration, and address resolution execute in user mode. The decision to place some of the ATM Linux code in kernel space and some in user space was design decision based on several considerations [13]. The driver and much of the ATM Linux code resides in kernel space to make efficient use of interrupts and DMA, to make use of resource sharing and access synchronization, and to avoid system call overhead when using kernel resources. The demons, on the other hand, run in user mode to allow for easier development and debugging, e.g. changes do not require the entire kernel to be recompiled. Program failure does not disable the entire operating system. Beyond program development, the design decision to put as much of the code as possible in user mode processes prevents bloat of the kernel [1].

4 Classical IP over ATM

Any attempt to create an interface between TCP/IP and ATM must deal with the issues of address resolution, routing, and packet encapsulation. The IETF favors Classical IP over ATM which uses an extended version of Address Resolution Protocol (ARP) called, not surprisingly, ATMARP. ATMARP translates network layer (IP) addresses into ATM addresses and vice versa. The IETF specifications for CLIP are defined in RFC 1577 [12].

RFC 1483 [10] describes the encapsulation of IP datagrams in AAL5. Two methods of encapsulation are possible but the method used by CLIP is LLC (Logical Link Control) encapsulation. This method consists of adding an 8 byte LLC/SNAP (Subnetwork Attachment Point) header containing the information needed to identify that this is an IP packet.

Hosts that are connected to the same physical ATM network may be configured into multiple virtual LANs known as Logical IP Subnetworks (LISs). Each host within a LIS may connect directly to every other host within the same LIS but must connect through an IP router to a host on a different LIS even if both are on the same ATM network. The value of this feature

may not be immediately obvious. RFC 1577 requires that every member of a LIS be able to communicate through ATMARP to every other member; i.e., the topology is fully meshed. The total number of connections needed, therefore, varies as the square of the number of members. By forming LISs, the number of connections required can be limited. The selector byte of the ATM NSAP address can be used to differentiate multiple LISs for the same ESI. Each LIS must have its own ATMARP server and each host on that LIS is responsible for registering itself with the server.

CLIP supports both a strictly PVC or SVC environment. In a PVC environment, all connections are manually configured by a network administrator. Therefore, the signaling demon and the ILMI demon are not required. Only the ATMARP demon is necessary. The function of the ATMARP demon in this case is to register the translation of IP address into ATM address for each host connected by PVC and to enter this information into its own ATMARP table. Each entry is associated with its own VCC.

For SVCs IP addresses must be resolved into ATM addresses. Every host on the LIS is required to register its IP address and ATM address with the ATMARP server. Whenever a client wants to send data to another client, it sends an ATMARP request containing the destination IP address to the ATMARP server. If the server knows the ATM address of the destination, it responds with an ATMARP reply. Otherwise, it sends an ATMARP NAK, which is an extension of the ARP protocol. Given the ATM address of the destination, the client can set up an SVC to the destination using signaling.

Regardless of whether PVCs or SVCs are used, data as well as ATMARP requests and replies can be carried over the same VCC in CLIP. LANE, on the other hand, requires separate connections for data and control messages.

5 LAN Emulation

The ATM Forum prefers a different approach to address translation. LAN Emulation "emulates services of existing LANs across an ATM network" [7]. Instead of translating IP addresses to ATM addresses, LANE takes a two step approach. It first translates IP addresses to MAC layer addresses and then translates the MAC layer address to an ATM address. An advantage of such an approach is that by emulating a MAC service, the LAN Emulation protocol, in principle, can be used with other protocol stacks such as NetBIOS, IPX (Internet Packet Exchange), or AppleTalk, in addition to IP.

The LANE specifications support emulation of either an IEEE 802.3/Ethernet or an IEEE 802.5/Token Ring LAN. However, the Linux implementation of LANE provides only Ethernet encapsulation. The data frame format for an emulated Ethernet must also conform to the Ethernet standard. The ATM Forum specifies [7] that IEEE 802.3 data frames including header and data shall be less than 1536 bytes. Therefore, the MTU for LANE is capped at 1500 bytes.

Just as CLIP can have multiple Logical IP Subnetworks, LANE may have multiple emulated LANs (ELANs). An emulated LAN is simply a group of ATM-attached devices. There can be several ELANS configured for the same ATM network. Membership in an ELAN does not depend on where the end system is physically connected, and an end system could belong to multiple ELANs. Communication between ELANs is possible only through routers or bridges. The Linux implementation of LANE was written by Marko Kiiskilä [11] as a master's thesis at Tampere University of Technology in Finland.

5.1 Components

An ELAN is made up of a number of LAN Emulation Clients (LECs) and a single LAN Emulation Service. The LAN Emulation Service is composed of a LAN Emulation Configuration Server (LECS), a LAN Emulation Server (LES), and a Broadcast and Unknown Server (BUS). The LANE client resides in an end system. The LES, BUS, and LECS may reside in either an end system or a switch. They also may be centralized in a single location or may be distributed.

The LEC performs data forwarding, address resolution, and other control functions. It provides a MAC level emulated Ethernet or Token Ring interface to higher level software. The LEC communicates with the other LANE components such as the LES using the LAN Emulation User to Network Interface (LUNI). Among the functions of the LUNI are obtaining the ATM addresses of the LANE service, joining or leaving an ELAN, address resolution, and data transfer. In Linux, the LANE client functions are performed by the `zeppelin` demon.

The LAN Emulation Configuration Server takes care of the distribution of LANE clients to different ELANS. The LECS is only needed if multiple ELANs are configured on the same ATM network. It can be bypassed by directly telling the ATM address of the LES to the LANE client. This was the approach we took; with only two machines, we needed only one ELAN.

The LAN Emulation Server coordinates the control of the emulated LAN. All clients must register with the LES on joining the ELAN and all address resolution queries are sent to it. In addition to these ATMARP like functions, the LES also distributes control messages which must be sent to every client.

The Broadcast and Unknown Server is primarily responsible for handling data sent by clients to broadcast and multicast addresses. It also can be used by the clients to send initial data to a unicast address to which a data connection has not yet been completely set up. The BUS and the LES are both required elements in the ELAN and all clients must register with both.

5.2 Connections

LAN Emulation requires separate connections for control and data traffic. If the LECS is present, each client must set up a Configuration Direct VCC to the LECS.

Two connections each are required for the client to connect with both the LES and the BUS. The client first sets up a bidirectional point to point Control Direct VCC to the LES. This connection carries control traffic from the client to the LES and may be used by the LES to send to the client. The LES then sets up a unidirectional point to point VCC or adds the client to a unidirectional point to multipoint VCC known as the Control Distribute VCC. The Control Distribute VCC is optional but is implemented in the Linux code.

When the client has successfully joined the ELAN, it sets up a bidirectional point to point Multicast Send VCC to the BUS, after obtaining the address of the BUS from the LES. The BUS then sets up a point to point unidirectional VCC or adds the client to its point to multipoint unidirectional VCC known as the Multicast Forward VCC. Both of these connections are data connections.

In addition to the data connections to the BUS, if the client wishes to send data to another client, it must set up a Data Direct VCC. These are bidirectional point to point VCCs. The client may send initial data to the destination via the Multicast Send VCC, but may not continue to use this connection if the Data Direct connection fails.

6 Data Transfer in Linux TCP/IP over ATM

Data communication using network protocols is by its very nature a complex task. Transferring data with TCP/IP protocols in an ATM network can only add to that complexity. This section presents a Linux code level view of the transfer of data from the application level send at the source to the application level receive at the destination. Only CLIP using SVCs is considered. The relevant data structures are given, the initialization and binding of those structures are detailed, and the general flow of control is described. Note that all file paths given in this section are relative to /usr/src. Paths are not given for functions from the device driver files because the driver code has not yet been integrated into the Linux distribution. The TCP/IP levels of the transfer are covered in Beck et al. [4]. However, their description is for kernel version 1.2. There are significant differences between that earlier version and version 2.0.25, which is used in this project. Westall [18] describes data transfer in a strictly ATM environment but much of that discussion is relevant and is used here.

6.1 TCP/IP Protocol Initialization

At system boot, all the network protocols known to Linux are initialized. The protocols are declared in a static table:

```
[ linux/net/protocols.c ]
struct net_proto protocols[] = {
   { "UNIX",     unix_proto_init },    /* Unix domain socket family */
   { "802.2",    p8022_proto_init },   /* 802.2 demultiplexor       */
   :
   { "INET",     inet_proto_init },    /* TCP/IP                    */
   :
   { "ATMPVC",   atmpvc_proto_init },
   { "ATMSVC",   atmsvc_proto_init },
   { NULL,       NULL            }     /* End marker                */
};
```

There is an initialization function for each protocol driver. Each one is executed at boot time. Of interest here is `inet_proto_init()` (linux/net/ipv4/af_inet.c). `inet_proto_init()` adds the protocol handler for each protocol to a single hash table for the system, called `inet_protos[]`. Each protocol is assigned an `inet_protocol` structure:

```
[ linux/include/net/protocol.h ]
struct inet_protocol {
   int           (*handler)(struct sk_buff *skb, struct device *dev,
                            struct options *opt, __u32 daddr,
                            unsigned short len, __u32 saddr,
                            int redo, struct inet_protocol *protocol);
   void          (*err_handler)(int type, int code, unsigned char *buff,
                            __u32 daddr, __u32 saddr,
                            struct inet_protocol *protocol);
   struct inet_protocol *next;
   unsigned char         protocol;
   unsigned char         copy:1;
   void                  *data;
   const char            *name;
};
```

For TCP, the structure is:

```
[ linux/net/ipv4/protocols.c ]
static struct inet_protocol tcp_protocol =
```

```
{
   tcp_rcv,          /* TCP handler          */
   tcp_err,          /* TCP error control    */
   NULL,             /* next                 */
   IPPROTO_TCP,      /* protocol ID          */
   0,                /* copy                 */
   NULL,             /* data                 */
   "TCP"             /* name                 */
};
```

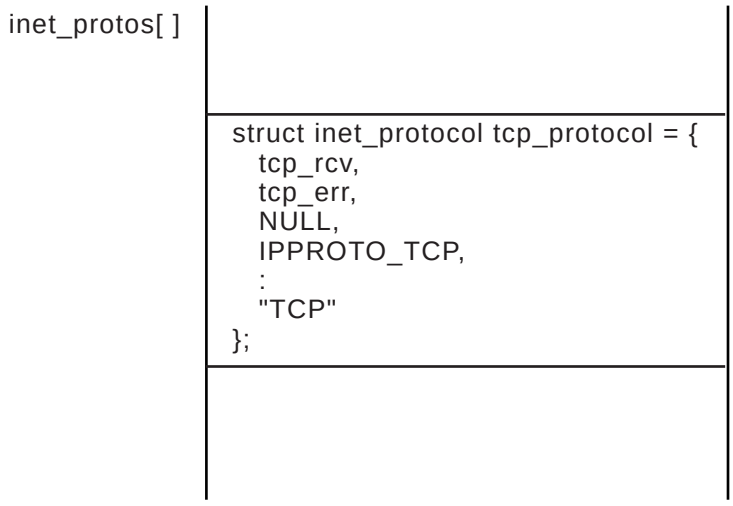

Figure 5: Initialization of TCP protocol "handler" function.

`inet_proto_init` also calls `ip_init()` (linux/net/ipv4/ip_output.c), which adds the IP protocol identification to a list, `ptype_base[]`, of all recognized protocols by means of a call to `dev_add_pack()` (linux/net/core/dev.c) passing the address of `ip_packet_type`. The data structure for packet type is defined below:

```
[ linux/include/linux/netdevice.h ]
struct packet_type {
   unsigned short     type;   /* This is really htons(ether_type). */
   struct device *    dev;
   int                (*func) (struct sk_buff *, struct device *,
                               struct packet_type *);
   void               *data;
   struct packet_type *next;
};
```

In the case of IP, the variable `ip_packet_type` is:

```
[ linux/net/ipv4/ip_output.c ]
```

```
static struct packet_type ip_packet_type =
{
   0,      /* MUTTER ntohs(ETH_P_IP),*/
   NULL,   /* All devices */
   ip_rcv,
   NULL,
   NULL,
};
```

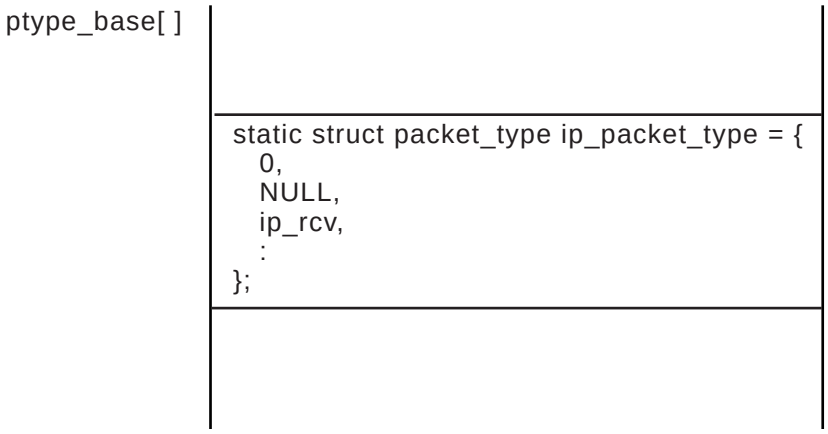

Figure 6: Initialization of IP packet type.

`inet_proto_init()` also registers the protocol using

```
(void) sock_register(inet_proto_ops.family, &inet_proto_ops);
```

(linux/net/socket.c). The protocol family is `PF_INET`. The structure `inet_proto_ops` is a table of entry point addresses for functions that provide an interface to the BSD socket layer. The `proto_ops` struct has the following form:

```
[ linux/include/linux/net.h ]
struct proto_ops {
   int family;

   int (*create) (struct socket *sock, int protocol);
   :
   int (*connect) (struct socket *sock, struct sockaddr *uservaddr,
                   int sockaddr_len, int flags);
   :
   int (*sendmsg) (struct socket *sock, struct msghdr *m, int total_len,
                   int nonblock, int flags);
   int (*recvmsg) (struct socket *sock, struct msghdr *m, int total_len,
                   int nonblock, int flags, int *addr_len);
};
```

For TCP/IP, this structure is filled in as follows:

```
[ linux/net/ipv4/af_inet.c ]
static struct proto_ops inet_proto_ops = {
   AF_INET,

   inet_create,
   inet_dup,
   inet_release,
   inet_bind,
   inet_connect,
   :
   inet_sendmsg,
   inet_recvmsg
};
```

`AF_INET` stands for the INET address family. It is equivalent to `PF_INET` and they are interchangeable. `sock_register()` saves the address of each protocol's `proto_ops` table in

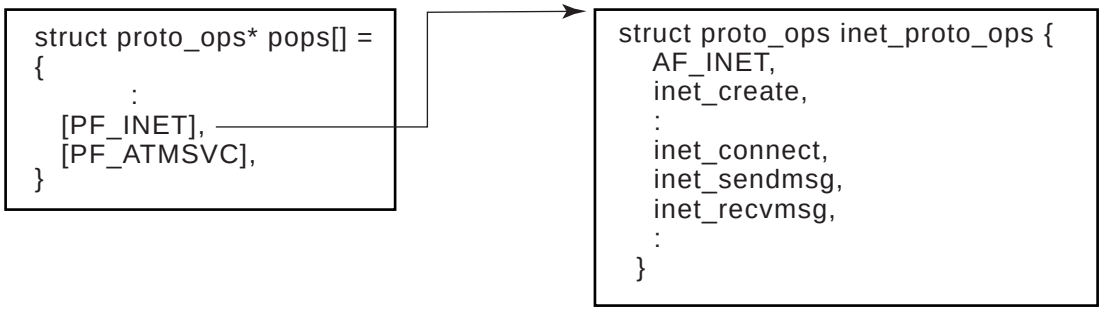

Figure 7: Binding after protocol initialization.

the kernel's `pops[]` table indexed by the protocol family number. Figure 7 shows the binding after protocol initialization.

6.2 ATM Device Initialization

When the ATM device driver module is installed, `init_module()` (atmbase.c) is executed. This function performs various initialization functions such as initializing buffer pools and registers, and then calls `ape25_register()` (atmape25.c) to register the device with the ATM protocol stack. `ape25_register()` registers the device with a call to `atm_dev_register()` (linux/net/atm/dev.c) and then reads the machine's ESI from the adaptor card's Non-Volatile RAM NVRAM. `atm_dev_register()` takes three parameters: the name of the device, a pointer to the device driver operations vector table, and a flags variable which is set to zero here. It allocates space in the ATM device table, which is indexed by the interface number. Each device table entry consists of an `atm_dev` struct:

```
[ linux/include/linux/atmdev.h ]
struct atm_dev {
   const struct atmdev_ops *ops; /* device operations; */
   const struct atmphy_ops *phy; /* PHY operations, may be undefined */
   const char      *type;        /* device type name */
   int             number;       /* device index */
   struct atm_vcc  *vccs;        /* VCC table (or NULL) */
   struct atm_vcc  *last;        /* last VCC (or undefined) */
   void            *dev_data;    /* per-device data */
   void            *phy_data;    /* private PHY date */
   unsigned long   flags;        /* device flags, TBD */
   struct atm_dev_addr *local;   /* local ATM addresses */
   unsigned char   esi[ESI_LEN]; /* ESI ("MAC" addr) */
   struct atm_cirange ci_range;  /* VPI/VCI range */
   struct atm_dev_stats stats;   /* statistics */
   int sending;
};
```

The ops field points to the device driver operations vector table, which has the following form for the ATM device:

```
[ atmape25.c ]
static struct atmdev_ops atm_ops =
{
   ape25_open,           /* open         */
   ape25_close,          /* close        */
   ape25_ioctl,          /* ioctl        */
   ape25_getsockopt,     /* getsockopt   */
   ape25_setsockopt,     /* setsockopt   */
   ape25_send,           /* send         */
   :
   ape25_free_rx_skb,    /* free_rx_skb  */
};
```

atm_dev_register() returns a pointer to the atm_dev structure to the device driver and links the driver ape structure to the device table entry. The bindings created at ATM device initialization are shown in figure 8. ape is pointer to a structure of type pddtype.

```
[ atmdd.h ]
struct pddtype
{
   unsigned char      pci_bus;
   unsigned char      pci_devfn;
```

```
   :
   unsigned char      esi[7];          /* End system identifier data */
   :
   struct   xbptype  *xbpool;          /* Transmit buffer pool.       */
   struct   rbptype  *rbpool;          /* Receive buffer pool.        */

   struct   tfdtype  *free_tfds;
   struct   tfdtype  *stcl;            /* -> xmit complete list.      */

/* Head and tail pointers to the free list of transmit */
/* frame descriptors.                                  */

   struct   tfdtype  *sftfdl;          /* -> xmit complete list.      */
   struct   tfdtype  *eftfdl;          /* -> xmit complete list.      */

   struct   rbhtype  *erfl;            /* -> END of receive free list */
   struct   rbhtype  *srrl[8];         /* Start of recv ready lists   */
   :
/* The atm device structure */

   struct   atm_dev  *atmdev;         /* -> registered device struct */
   :
};
```

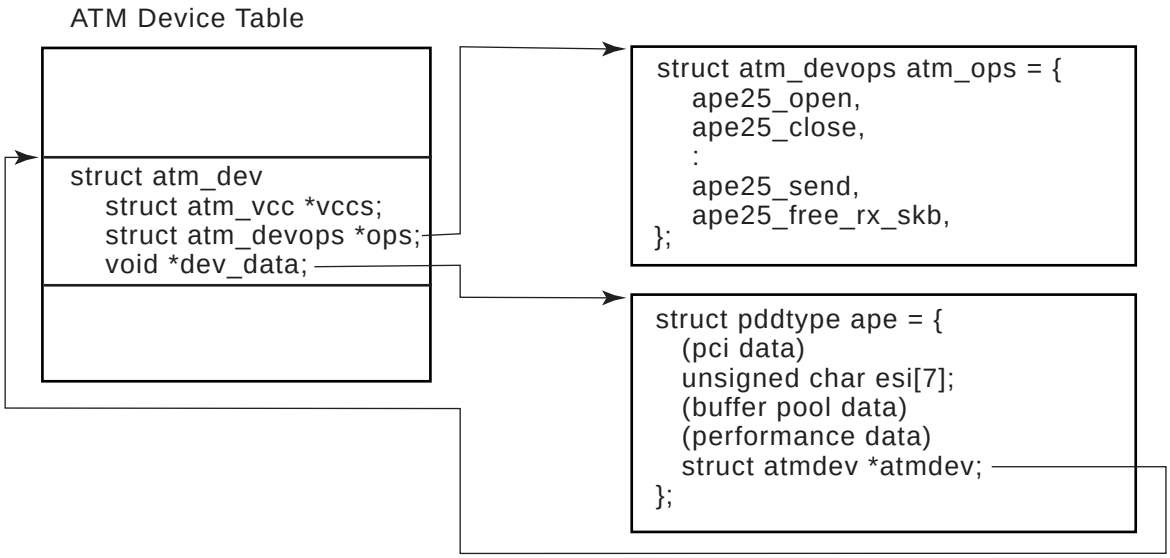

Figure 8: Bindings created during device driver initialization

6.3 Socket Creation

Two types of socket structures must be considered. BSD sockets represent the general data structure for sockets. INET sockets, on the other hand, are communication end points for IP based protocols and are represented by the `sock` data structure. A BSD socket is created with the system call:

```
[ linux/net/socket.c ]
socket(int family, int type, int protocol)
```

There are nearly a dozen protocol families recognized by Linux. The two families of interest here are `PF_INET` and `PF_ATMSVC`. The type field specifies the type of connection requested. `SOCK_STREAM` indicates a reliable connection-oriented byte stream while `SOCK_DGRAM` is used for connectionless unreliable messages.

```
s = socket(PF_INET, SOCK_STREAM, 0);
```

```
s = socket(PF_ATMSVC,SOCK_DGRAM,0)
```

Consider the process for the `PF_INET` family first. The system call `socket()` locates the protocol family in the `pops[]` table and allocates memory for the socket structure:

```
[ linux/include/linux/net.h ]
struct socket {
   short                type;         /* SOCK_STREAM, ...             */
   socket_state         state;
   long                 flags;
   struct proto_ops     *ops;         /* protocols do most everything */
   void                 *data;        /* protocol data                */
   :
   struct inode         *inode;
   struct fasync_struct *fasync_list; /* Asynchronous wake up list    */
   struct file          *file;        /* File back pointer for gc     */
};
```

The `ops` field stores a pointer to the INET protocol operations table which was retrieved from `pops[PF_INET]`. The protocol specific create function, `inet_create()` (linux/net/ipv4/af_inet.c) is then called. `inet_create` allocates an INET socket structure `sock`.

```
[ linux/include/net/sock.h ]
struct sock
{
   :
   struct sock             *next;
   struct sock             *prev;     /* Doubly linked chain.. */
   :
   struct proto            *prot;
   struct wait_queue       **sleep;
   __u32                   daddr;
   __u32                   saddr;     /* Sending source */
   __u32                   rcv_saddr; /* Bound address */
   :
   unsigned short          mtu;       /* mss negotiated in the syn's */
   :
   struct socket           *socket;
   :
   void                    (*data_ready)(struct sock *sk,int bytes);
};
```

`inet_create()` initializes some of the fields in the `sock` structure most of which are specific to TCP. Of particular interest is the `data_ready` field which is set to `def_callback2()` and the `prot` field which is set to point to `tcp_prot`.

```
[ linux/net/ipv4/tcp.c ]
struct proto tcp_prot = {
   tcp_close,
   ip_build_header,
   tcp_connect,
   tcp_accept,
   ip_queue_xmit,
   :
   tcp_rcv,
   tcp_select,
   tcp_ioctl,
   :
   tcp_sendmsg,
   tcp_recvmsg,
   :
};
```

The `sock` and `socket` structures are linked by a pointer from the `data` field of the BSD socket to the INET socket and from the `socket` field of the INET socket to the BSD socket.

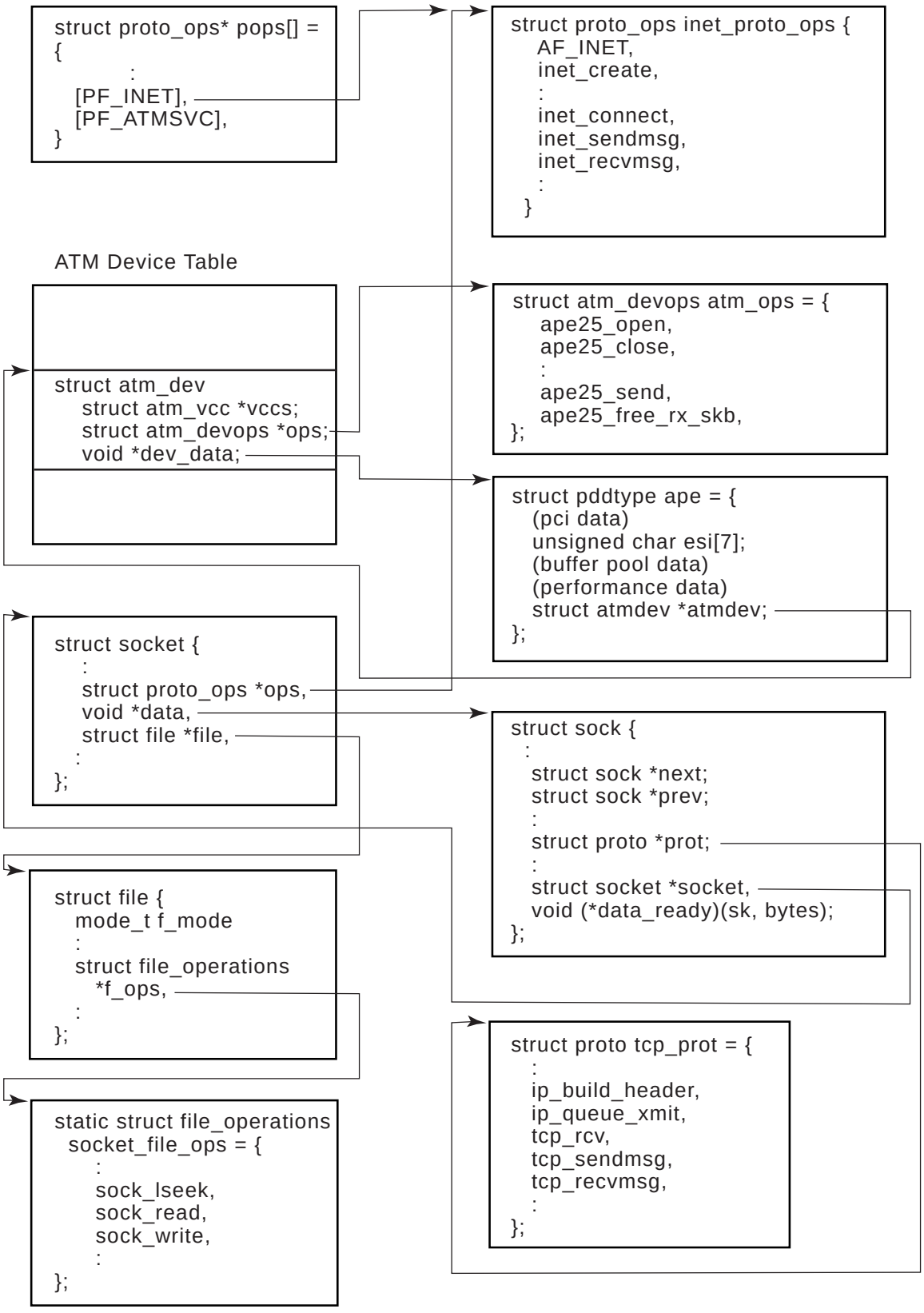

Figure 9: Bindings after socket creation

After the call to `inet_create`, `socket()` has one more task to accomplish. It calls `get_fd()` (linux/net/socket.c) to retrieve a file descriptor. `get_fd()` associates the file descriptor with a `file` data structure and sets the `f_op` pointer to `socket_file_ops`.

```
[ linux/include/linux/fs.h ]
struct file {
   mode_t f_mode;
   loff_t f_pos;
   :
   struct file *f_next, *f_prev;
   int f_owner;             /* pid or -pgrp where SIGIO should be sent */
   struct inode * f_inode;
   struct file_operations * f_op;
   :
};
```

```
[ linux/net/socket.c ]
static struct file_operations socket_file_ops = {
   sock_lseek,
   sock_read,
   sock_write,
   NULL,                   /* readdir */
   sock_select,
   sock_ioctl,
   :
   sock_fasync
};
```

Figure 9 shows the bindings that have taken place to this point.

The process is similar with the `PF_ATMSVC` family, except that instead of an INET `sock` structure, the create function, `atm_create` (linux/net/atm/common.c) allocates and initializes an `atm_vcc` structure, which contains ATM specific fields.

```
[ linux/include/linux/atmdev.h ]
struct atm_vcc {
   unsigned short  flags;          /* VCC flags (ATM_VF_*) */
   unsigned char   family;         /* address family; 0 if unused */
   unsigned char   aal;            /* ATM Adaption Layer */
   short           vpi;            /* VPI and VCI (types must be equal */
                                   /* with sockaddr) */
   int             vci;
   unsigned long   aal_options;    /* AAL layer options */
```

```
    unsigned long   atm_options;    /* ATM layer options */
    struct atm_dev  *dev;           /* device back pointer */
    struct atm_qos  qos;            /* QOS */
    unsigned long   tx_quota,rx_quota; /* buffer quotas */
    atomic_t        tx_inuse,rx_inuse; /* buffer space in use */
    void (*push)(struct atm_vcc *vcc,struct sk_buff *skb);
    void (*pop)(struct atm_vcc *vcc,struct sk_buff *skb); /* optional */
    :
    void            *dev_data;      /* per-device data */
    void            *proto_data;    /* per-protocol data */
    /* SVC part --- may move later */
    short           itf;            /* interface number */
    struct sockaddr_atmsvc local;
    struct sockaddr_atmsvc remote;
    :
};
```

The `pop()` function is optional, and it is set to `NULL` in `atm_create`.

6.4 Socket Connection

After creation of the socket at the sender, the IP address of the destination, the port for the connection, and the address family must be assigned to a `sock_addr` socket address structure. The socket address structure is then passed as a parameter to `connect()` (linux/net/socket.c).

```
struct sockaddr_in name;

bcopy((char *)&dummyad, (char *)&name.sin_addr, sizeof(dummy));
name.sin_family = AF_INET;
name.sin_port = htons(33456);
status = connect(s, (struct sockaddr *)&name, sizeof(name));
```

The system call `connect()` calls the protocol specific function `inet_connect()` (linux/net/ipv4/af_inet.c) which it has retrieved from the `proto_ops` field of the socket structure. `inet_connect()` makes sure that the state is a valid state and then sets the state to `SS_CONNECTED`.

The receiver of a data transfer must be prepared to handle incoming packets. The `socket()` system call creates an unnamed socket. The system call `bind()` (linux/net/socket.c) must be used to assign an address to the socket. After the address is bound to the socket, the receiver calls `listen()` (linux/net/socket.c) to indicate willingness to accept connections. Finally,

the receiver invokes `accept()` (linux/net/socket.c), which takes the first connection request from the queue of pending connections or blocks if a connection request is not available.

6.5 ATMARP Initialization

The following commands can be used to start up the CLIP demons on the host where the ATMARP server will reside:

```
/usr/local/sbin/atmsigd -b
/usr/local/sbin/ilmid -b
/usr/local/sbin/atmarpd -b
/usr/local/sbin/atmarp -c atm0
/sbin/ifconfig atm0 $IP_ADDR_THIS_HOST up mtu $MTU
/sbin/route add -net $NET_IP_ADDR netmask 255.255.255.0 dev atm0
```

If the machine is not to be the ATMARP server host, then the following command is also needed:

```
/usr/local/sbin/atmarp -s $IP_ADDR_ATMARPSRV $ATM_ADDR_ATMARPSRV pub arpsrv
```

The first three lines start up the signaling demon, the ILMI demon, and the ATMARP demon. The `-b` option runs them in the background and ensures synchronization. An atm interface is then created via a call to `atmarp -c atm0`. The module `atmarp` (atm/ip/atmarp.c) first parses the command line options and, for the `-c` option, sets the operation `op` to `SIOCMKCLIP`. For this operation, the interface name `atm0`, is first processed. The first three characters are stripped off leaving the interface number, which is stored in the variable `req_num`. Next, a socket is created:

```
s = socket(PF_ATMSVC,SOCK_DGRAM,0)
```

Communication with the ATMARP server will take place using the socket just created. The socket handle `s` and the `req_num` variable are then passed to the protocol family's `ioctl`.

```
num = ioctl(s,op,req_num)
```

For the protocol family, `PF_ATMSVC`, the call goes to `svc_ioctl()` (linux/net/atm/svc.c), which calls `atm_ioctl()` (linux/net/atm/common.c). In turn, `atm_ioctl()` simply calls `clip_create()` (linux/net/atm/atmarp.c) passing the interface number as the only parameter. `clip_create()` allocates memory for a device structure, sets the device name to `atm0`,

the device number to 0, the device initialization function to `clip_init()` and registers the device with a call to `register_netdev()` (linux/drivers/net/net_init.c) passing the device `atm0` as the parameter. `register_netdev()` then fires off `clip_init()` (linux/net/atm/atmarp.c), which sets a number of device functions, but in particular, it sets:

```
dev->hard_start_xmit = clip_xmit;
```

The assignment of `clip_xmit()` to `dev->hard_start_xmit()` creates the cross-over point between the TCP/IP protocol stack and the ATM stack.

The next command to be executed after `atmarp -c` is `ifconfig`. `ifconfig` configures the `atm0` interface. It associates the IP address of this host (`atm`, in this case) with the ATM interface that was just created using `atmarp`. The interface and the network address are added to `atm`'s routing table with the `route` command.

The host that is not acting as the ATMARP server must be told the ATM address and IP address of the server. It will add the address information to its ATMARP table and will immediately try to set up a connection to the server. If the connection request is successful, the server will create a new entry for the host in its ATMARP table associating the network assigned VCC to that entry. It will also set the `push()` function of the VCC to `atm_push_ip()`.

6.6 Data Transfer

Suppose data are to be transmitted from host `atm` to host `chattooga` using IP over ATM. Suppose further that sockets have been created at each end point and `connect()` has been executed on `atm` and `bind()`, `listen()`, and `accept()` have been called on `chattooga`.

6.6.1 Send Side

For an overall view of the function call sequence see figure 10.

On `atm`, the transfer will start with

```
write(s, buf, blk);
```

where `s` is the socket handle, `buf` is the buffer containing the data to write, and `blk` is the size in bytes of the data.

`write()` invokes the system call `sys_write()` (linux/fs/read_write.c) which does a number of sanity checks including a check on whether the area of memory referenced by `buf` has read

Figure 10: Overview of simplest case send sequence.

access. It treats the socket handle as a file descriptor, associates a file structure with that file descriptor and looks up the inode that corresponds to this file. It then calls:

```
file->f_op->write(inode,file,buf,count)
```

The write operation that will be called is `sock_write` (linux/net/socket.c). `sock_write` looks up the socket associated with this inode, verifies that the memory area referred to has read access, and calls:

```
sock->ops->sendmsg(sock, &msg, size,(file->f_flags & O_NONBLOCK),0)
```

This socket is a member of the `PF_INET` family, so the operation called is `inet_sendmsg()` (linux/net/ipv4/af_inet.c).

```
inet_sendmsg(struct socket *sock, struct msghdr *msg, int size, int noblock,
             int flags)
```

`inet_sendmsg()`extracts the pointer to the INET socket structure `sock` and stores it in a variable `sk`. The `prot` field of this structure points to `tcp_prot`: Consequently, when the following line in `inet_sendmsg()` is executed, `tcp_sendmsg()` will be called.

```
sk->prot->sendmsg(sk, msg, size, noblock, flags)
```

`tcp_sendmsg()` (linux/net/ipv4/tcp.c) merely does some sanity checking, and then invokes `do_tcp_sendmsg()` (linux/net/ipv4/tcp.c) where the real business of transferring the data begins. First there is some error checking such as verifying that the connection is established. Next there is a check of the window size. If the send window has shrunk, a partial frame is sent immediately. Otherwise, if the frame is partially filled, it will continue to be filled. If all data are copied and the frame is still not full, the partial frame will be enqueued and a timer is set. The system will wait up to 0.1 second for more data to fill out the frame. If it doesn't arrive by then, a partial frame will be sent. Memory for an `sk_buff` is allocated and the IP header is built with a call to:

```
sk->prot->build_header(skb, sk->saddr, sk->daddr, &dev, IPPROTO_TCP,
                       sk->opt, skb->truesize, sk->ip_tos,
                       sk->ip_ttl, &sk->ip_route_cache);
```

which is a call to `ip_build_header()` (linux/net/ipv4/ip_output.c). MAC and IP headers are added to the packet by `ip_build_header()`. If the interface on which to send has not yet been set, it looks up the destination address in the routing table and stores the device (i.e.

interface) on which to send it. A route was added to the routing table for this destination when the ATM demons were started and the device name associated with this route was `atm0`.

The TCP header is added next with a call to `tcp_build_header()` (linux/net/ipv4/tcp.c). Finally, `do_tcp_sendmsg()` copies the data from the user buffer to the socket and calls `tcp_send_skb(sk, skb)` (linux/net/ipv4/tcp_output.c) if the frame is full, and enqueues a partial frame if it is not.

`tcp_send_skb()` does more sanity checking, fills in some information in the header, and decides if the packet is to be sent immediately or queued. It must be queued if the right edge of the frame exceeds the send window, or if this is a retransmission, or if there are too many unacked packets. Otherwise, it goes straight out with a call to:

```
sk->prot->queue_xmit(sk, skb->dev, skb, 0);
```

which in this case will be `ip_queue_xmit()` (linux/net/ipv4/ip_output.c). `ip_queue_xmit()` adds some information to the `sk_buff` struct, sets the IP header length, and adds the packet to the send queue. It then checks the length of the packet to see if it is larger than the MTU allowed on this interface, in which case the packet will be fragmented. The packet, the device, and the priority for sending are then passed to `dev_queue_xmit()` (linux/net/core/dev.c) which increments the interrupt count variable `intr_count` and calls `do_dev_queue_xmit()` (linux/net/core/dev.c). The interrupt count is incremented and will be decremented when the send is complete. The packet is placed on the device queue and the device specific send function is called via `dev->hard_start_xmit`. In the case of `atm0`, the function called is `clip_xmit()` (linux/net/atm/atmarp.c).

`clip_xmit()` first checks to see if there is an entry in the ATMARP table for the next hop address. The VCC created for connection to the ATMARP server will be found. The data structure for an ATMARP table entry follows:

```
[ linux/net/atm/atmarp.h ]
struct atmarp_entry {
   unsigned long   ip;             /* IP address, 0 if none */
   struct atm_vcc  *vcc;           /* active VCC */
   unsigned char encap;            /* 0: NULL, 1: LLC/SNAP */
   struct device   *dev;           /* device back pointer */
   void (*old_push)(struct atm_vcc *vcc,struct sk_buff *skb);
                                   /* keep old push fn for detaching */
   unsigned long last_use;         /* last send or receive operation */
   unsigned long idle_timeout;     /* keep open idle for so many jiffies*/
   struct sk_buff *queued;         /* queue one skb when resolving */
   struct atmarp_entry *next;      /* ugly linked list ... */
};
```

The VCC for this entry is retrieved, an LLC/SNAP header is added to the packet, and the counter for the number of transmits in progress over this VCC is incremented. Finally the following line of code is invoked:

```
entry->vcc->dev->ops->send(entry->vcc,skb);
```

The function invoked will be the driver send routine `ape25_send()` (atmape25.c). A pointer to the `ape` structure obtained from the `dev_data` field of the `atm_vcc` structure is recovered and `atm_send5skb()` (atmxmit.c) is called passing pointers to `ape`, the `atm_vcc` struct, and the socket buffer. `atm_send5skb()` recovers a transmit frame descriptor (TFD) from the free list maintained by the driver. It then binds the socket buffer to the TFD and stores the address of the `atm_vcc` in an extension to the TFD. At this point the packet is handed on to the hardware layer for transmission.

When the transmit is complete, the driver receives a transmit complete interrupt, which causes `atm_xmitint()` (atmxmit.c) to be executed. `atm_xmitint()` removes the TFD from the transmit complete list, retrieves the `atm_vcc` pointer from the TFD extension, and frees the `sk_buff` either by invoking the `pop()` routine, if it exists, pointed to by the `atm_vcc` or by calling `dev_kfree_skb()` (linux/net/core/skbuff.c).

6.6.2 Receive Side

The overall call sequence for receiving a packet is shown in figure 11. On the receive side, when the hardware receives a packet, it generates a receive complete interrupt, which causes `atm_recvint()` (atmrecv.c) to be called. `atm_recvint()` filters out unsolicited Operations and Maintenance (OAM) packets and invokes `atm_recvdq()` (atmrecv.c) to process the input. `atm_recvdq()` goes through the list of ready descriptors and consumes the data. It then checks to see if there is a valid connection identifier for this buffer. If there is, the packet is pushed to the protocol layer using the `push()` field of the recovered `atm_vcc`.

```
vcc->push(vcc, rbh->skb);
```

At ATMARP initialization, `vcc->push` was set to `atm_push_ip()` (linux/net/atm/atmarp.c) which will now be invoked passing a pointer to the `atm_vcc` struct and a pointer to the `sk_buff`. `atm_push_ip()` sets the device field of the `sk_buff` to the device pointed to by the `vcc` and calls the function `netif_if()` (linux/net/core/dev.c) passing the packet. There is on each system a single list called `backlog` of all the packets received and not yet processed. For what are probably purely ad hoc reasons, the length of this list is limited to 300 in Linux. If there are more than 300 packets queued, this packet will be dropped. Otherwise, the packet is added to the queue and the `NET_BH` bottom half is marked active in the bottom half mask.

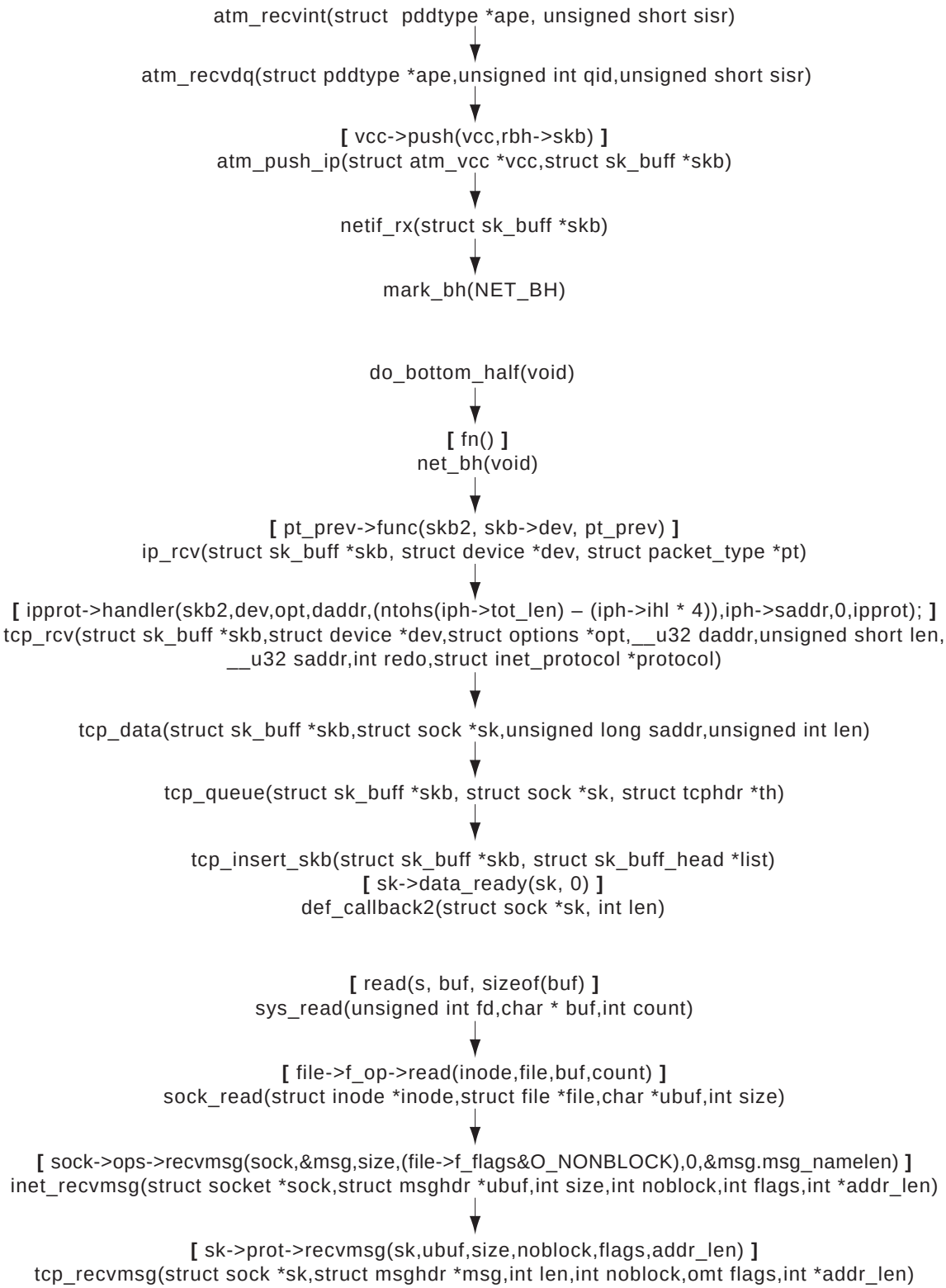

Figure 11: Receive side call sequence.

Some interrupts have functions that do not need to be performed immediately or take too long to execute creating unnecessary delays. To process these less timing-critical functions, bottom halves have been created. After every return from a system call, if there are no other pending interrupts, i.e. `intr_count` is zero, `do_bottom_half()` (linux/kernel/softirq.c) is executed. `do_bottom_half()` scans the system wide list of bottom halves and those which are marked as active are invoked. In this case, the function `net_bh()` (linux/net/core/dev.c) will be executed.

`net_bh()` first tries to send any data that may be ready. It then dequeues a packet from the backlog queue and retrieves the protocol ID from the packet header. It calls the `func()` procedure for that packet type after removing the MAC header from the packet.

```
pt_prev->func(skb2, skb->dev, pt_prev);
```

The packet just received will be of type `ip_packet_type` for which the `func()` field was set to `ip_rcv()` (linux/net/ipv4/ip_input.c) at protocol initialization.

`ip_rcv()` deals primarily with information from the IP header. It first checks that the packet length is at least as long as the IP header. The packet must also be an IP version 4 packet. Next the header checksum is verified. If any of the above tests fail, the packet will be discarded. If padding has been added to the IP packet, it is removed at this time. The packet is also checked for fragmentation and is reassembled if it has been fragmented. If IP forwarding is required or if the packet destination is a multicast address, it is also taken care of now. A pointer is set to the next header in the packet, which, in this case, will be the TCP header. Finally, the packet is handed on to the TCP protocol with the following code:

```
ipprot->handler(skb2, dev, opt, daddr,
                (ntohs(iph->tot_len) - (iph->ihl * 4)),
                iph->saddr, 0, ipprot);
```

At boot time, during protocol initialization, the `handler` function for TCP was set to `tcp_rcv()`.

`tcp_rcv()` (linux/net/ipv4/tcp_input.c) finds the TCP header in the packet and verifies the checksum. The INET socket for the source/destination pair is retrieved. If the socket is already in use, the packet is placed back on the backlog queue. The current state is checked, and, if the state is a closed connection, the packet is dropped. Other errors in the state are also checked and dealt with. `tcp_rcv()` next checks that the receive window is non-zero and that the sequence number falls within that window. Assuming that everything has checked out to this point, the incoming acknowledgement is processed and any urgent data are dealt with. Finally, the actual processing of the data in the packet is accomplished via a call to `tcp_data()` (linux/net/ipv4/tcp_input.c) passing the source address and the total length of the packet as well as pointers to the `sk_buff`, and the `sock` struct.

tcp_data() does some preliminary checking, making sure that the length of the data is non-zero and that there is still an active connection. It then estimates the delayed acknowledgement timeout with a call to tcp_delack_estimator() and hands the packet on to tcp_queue() (linux/net/ipv4/tcp_input.c) passing pointers to the sk_buff, the INET socket, and the TCP header structure.

tcp_queue() invokes tcp_insert_skb (linux/net/ipv4/tcp_input.c) which adds the sk_buff to the TCP receive queue. It also checks to see if any new ACKs need to be sent but delays sending them if possible. The data are now ready for the application to read, so a call is made to:

```
sk->data_ready(sk,0);
```

At socket creation time, inet_create() set the data_ready() field of the INET socket structure to def_callback2() (linux/net/ipv4/af_inet.c). def_callback2() wakes the process waiting in the socket's wait queue.

On host chattooga, the application waiting for the data will have the following line of code:

```
status = read(s, buf, sizeof(buf));
```

where s is the socket handle and buf is the memory area assigned to hold the data. The system call sys_read() (linux/fs/read_write.c) uses the socket handle as a file descriptor and retrieves the file structure associated with that descriptor and the inode structure associated with the file struct. It checks for error conditions including write permission to the memory area and the existence of a valid read operation associated with the file structure. sys_read() then executes:

```
file->f_op->read(inode,file,buf,count);
```

Back at socket creation time, the read field of the file operations field for the socket file was set to sock_read() (linux/net/socket.c), which is now invoked.

sock_read() makes sure that the size of the area to which to write is greater than zero bytes and that it is writable by the user. It then initializes a variable msg of type struct msghdr and calls:

```
sock->ops->recvmsg(sock, &msg, size,(file->f_flags & O_NONBLOCK),
                   0,&msg.msg_namelen)
```

In this case, inet_recvmsg() will be called (see figure 9).

inet_recvmsg() (linux/net/ipv4/af_inet.c) retrieves a pointer sk to the INET socket from the BSD socket, and calls:

```
sk->prot->recvmsg(sk, ubuf, size, noblock, flags,addr_len)
```

The function called will be `tcp_recvmsg()` (linux/net/ipv4/tcp.c). `tcp_recvmsg()` checks to see that the connection is up. If the packet has urgent data, it is processed right away. Otherwise, the process is added to the wait queue for the socket. When the process is awakened by `def_callback2()`, `tcp_recvmsg()` copies the data into user space, thus completing the data transfer.

7 Software Integration

This section details the problems encountered in the integration of the Linux CLIP and LANE code with the hardware and software configuration. Where possible, specific error messages are reported and explained and solutions to those problems/errors are provided.

7.1 CLIP Using PVCs

The simplest possible setup with Classical IP over ATM is ATMARP in a strictly PVC environment. With PVCs, no signaling is required and address registration via ILMI can be avoided. All that is needed is to start up the ATMARP demon, configure the interface, add the route to the routing table, and set up the PVCs between the two hosts. We assigned a bogus IP address to each host to give the ATM interface a unique IP identifier. The VCC for each PVC is associated with the IP address. All of the ATM demons must be owned by root. If anyone except the superuser tries to start them up, errors such as the following will be seen:

```
atmarpd:IO: ioctl ATMARPD_CTRL: Operation not permitted
Fatal error - Terminating
ioctl SIOCMKCLIP: Operation not permitted
ioctl SIOCxARP: Operation not permitted
```

A quick and easy check of whether CLIP or LANE is working at all is to ping one host from the other across the ATM interface. With the correct ownership of `atmarpd`, ping operated successfully but there were problems if large amounts of data were transferred. In general, the failures at this point were catastrophic, such as:

```
Kernel panic: task[0] trying to sleep
```

followed by many lines of

```
In swapper task - not synching
```

More frequently, the system hung requiring a reboot. The solution to this problem had several parts. First, the maximum transmission unit (MTU) for the network had to be set using the `mtu` option to `ifconfig`. In the absence of such a limit, CLIP uses a default MTU value of 9180, which is too large for the current setup to handle. Second, several modifications to the device driver were necessary.

The kernel structure containing the data being transferred is called an `sk_buff`. Each `sk_buff` used in receiving AAL5 frames must be associated by the driver with a receive buffer header (RBH), whose format is dictated by the Network Interface Card (NIC). The RBH is used by the NIC to manage two lists. When the NIC initiates reassembly of an AAL5 frame, it consumes an RBH and the associated `sk_buff` from the Receive Free List (RFL). When the reassembly of the frame is complete, the NIC enqueues the RBH on the Receive Ready List (RRL) bound to the VCC on which the frame arrived and generates a receive complete interrupt.

The device driver maintains a pool of 256 statically allocated RBH structures. At device driver initialization, 32 of these headers are bound to `sk_buff`s and placed on the RFL. The remaining 224 RBH structures are enqueued on a driver managed list called the Free RBH list.

When a receive operation completes, the device driver detaches the RBH from the associated `sk_buff`. The driver forwards the `sk_buff` to the next protocol layer above by calling the `push` function pointed to by the `atm_vcc` structure associated with the connection. The newly freed RBH is then returned to the Free RBH list.

If there were no mechanism for replenishing the RFL, no more data could be received after the initial allocation of 32 buffers were consumed by the NIC. To replenish the RFL, the device driver exports a `free_rx_skb` function in its `atmdev_ops` table. The presence of such a function indicates to the protocol above the device driver that the buffer should be returned to the device driver at the `free_rx_skb` entry point as soon as the data have been consumed by the ultimate receiver.

The device driver's `free_rx_skb` function is named `ape25_free_rx_skb()`. Its normal function is to dequeue an RBH from the Free RBH list, bind the RBH to the `sk_buff` being returned, and enqueue the RBH on the RFL. This operation would appear to "close the loop" as far as receive buffer management goes. However, in practice, there are at least two complicating factors. For one thing, when Classical IP over ATM was used, the `sk_buff`s were *never* returned to the driver.

The second complicating factor is that the rate at which buffers are returned to the driver is limited to the rate at which data are consumed by the application. Under heavy system loads and high incoming frame rates, it is easily possible to totally deplete the RFL. When this situation occurs, the NIC simply drops frames until the RFL is replenished.

Therefore, it was necessary to modify the original closed loop design of the device driver and incorporate a demand-based mechanism for dynamically allocating and freeing `sk_buff`s. This mechanism works by maintaining a count of the number of elements on the Free RBH list. Since the RBH structures reside only on the Free RBH list or on the RFL, the number of elements on the Free RBH list also determines the number of elements on the RFL. Each time a receive operation completes, the device driver computes the number of elements on the RFL. If the length of the RFL reaches a low-water-mark, the device driver allocates new `sk_buff`s until the RFL returns to its original length.

When the internal congestion abates, the `sk_buff`s are gradually returned to the device driver and the number of elements on the RFL list increases. However, the RFL is not permitted to grow longer than its initial length. Whenever an `sk_buff` is returned to the device driver, and the length of the RFL is equal to or greater than its original length of 32, the device driver simply frees the `sk_buff`.

Finally, the size and number of the device driver transmit and receive buffers also had to be adjusted to support larger MTUs.

After making these adjustments, all sizes of data transfers completed but on large data transfers, the following message was repeated many times:

`AIEE: scheduling in interrupt 001105da`.

This error comes from the CPU scheduler, `schedule()`. The cause of this error was traced back to the function `atm_send5skb()` in the device driver file `atmxmit.c`. Among the tasks of `atm_send5skb()` is the assignment of a transmit frame descriptor. If one were not available the process waits by calling `interruptible_sleep_on()`, which places the process on a wait queue and calls `schedule()`. As one of its first actions, `schedule()` checks the variable `intr_count`, and if it is not zero, `schedule()` complains about being called during an interrupt. It appeared that `atm_send5skb()` was being called from an interrupt handler. However, further investigation showed that along the normal send path in TCP is the function `dev_queue_xmit()` (see figure 10). In `dev_queue_xmit()`, just before the call to `do_dev_queue_xmit()`, is a call to `start_bh_atomic()` which increments `intr_count`. After `do_dev_queue_xmit` returns (when the transmit is complete) `intr_count` is decremented. This meant that `intr_count` was *always* non-zero in `atm_send5skb()`, and hence, the complaints from `schedule()`. Linux developer, Alan Cox says the increment of the interrupt count "help[s] serialise calls to the driver transmission functions". [1] The driver code has been modified so that when sufficient resources are not available, `atm_send5skb()` checks to see if the interrupt count is zero. If it is, it sleeps. Otherwise, the packet is dropped. TCP is, of course, reliable, and the higher layers will resend the packet. Since this correction, we have transferred files larger than 1 Gigabytes without error.

[1] Posted to the ATM-Linux mailing list, March 20, 1998

7.2 CLIP Using SVCs

We considered testing LAN Emulation using PVCs only, but it does not appear that the Linux implementation of LANE supports such a configuration [11], although it is allowed by the specifications[2]. Since we had already dealt with the problems involved in running `atmarpd`, the next logical step was to run CLIP using SVCs. This, however, meant that we had to get signaling up and running. Unfortunately, we had several errors to find and correct before anything functioned at all.

One of the first error messages we saw when trying to start up `atmarpd` was:

```
atmsigd:  SSCOP: layer management - error 0 "VT(CC)>=MaxCC"
```

The message indicates a total failure of the signaling components. Werner Almesberger interprets it as "It just says that it tried so many times to talk to the other SSCOP entity without getting a useful response."[3] In other words, one side of the connection is trying to signal, but it doesn't get a response back. We still see this message when we start up the signaling demons but generally only once. It is necessary to start up the demons on the end system serving as the ATMARP server before the demons in the other systems, because the clients will immediately try to contact the server and it must be there. There will always be a few moments when one system's demons are ready but the others' are not.

The next error message we had to understand was:

```
atmarpd:IO:[2]connect:  Cannot assign requested address
```

It turned out that the device driver had hard coded the ESI portion of each end system's ATM address. Not only did this give both machines the same address, but that address did not match the addresses configured for the respective ports in the switch. Dr. Westall wrote a utility that allows us to write an end system's ESI into the Non-Volatile RAM (NVRAM) of the adapter card. The driver now reads its own system's ESI from the NVRAM.

Another discrepancy between switch configuration and end system configuration was the IP address of the switch itself. The switch thought that its IP address was `221.1.1.1` while the end systems, `atm` and `chattooga`, believed it to be something else. This conflict resulted in the following error:

```
atmarpd:ARP: got unroutable IP address 221.1.1.1
```

This problem was easily corrected by changing the IP address in the switch configuration.

One of our most persistent and frustrating problems produced the following error message from the recipient of a request for a connection:

[2]Even though it is allowed, the details of a PVC configuration are not given.

[3]Posted in a message to the ATM-Linux mailing list, October 6, 1997

```
atmarpd:IO:accept:  Invalid argument
```

and

```
atmarpd:IO:[2]connect:  Connection refused
```

from the sender of the connection request.

Although it is the end systems which request connections, only the network assigns the connection identifiers. The range of valid VCIs currently supported by the device driver is 64. However, the switch was set to expect 1024. As connections time out and new connections are set up, the VCI assigned is incremented. Without a way of informing the switch of our limited range, it rather soon started assigning identifiers outside of the supported range. The range of VCIs and VPIs are among the values supported by the ATM MIB. However, the Linux version of ILMI provided us with no method of setting those values. Fortunately, there was a firmware upgrade available for the switch from IBM that allowed us to set the valid VCI range manually.

7.2.1 Connection Timeouts

We found that while running CLIP with SVCs, our connections timed out if they were left idle. A new connection was immediately started up without intervention on our part. This occurred approximately 20 minutes after the connection was initialized and then every 25 minutes thereafter. An investigation showed that it was the server ATMARP table entries which were expiring. RFC 1577 states "Server ATMARP table entries are valid for a minimum of 20 minutes." This value is set in `arp.c` to 25 minutes. Further experimentation showed that the connection timed out only when the switch was configured to expect machine `atm` to be the ATMARP server. If the switch was given its own address or "0.0.0.0" as the IP address of the ATMARP server, the connections did not time out at all. This may indicate some failure in communication between the switch and the server, or it may simply be that IBM has a different interpretation of the protocols from the Linux group.

7.3 LANE Using SVCs

7.3.1 ATM Address Errors

The Linux implementation of LAN Emulation was the most difficult to set up. The documentation available with the Linux ATM distribution, consisting of on-line man pages for the demons and a usage file (/usr/src/atm/lane/USAGE) is poorly written, confusing, and incomplete. Fortunately, the implementation does permit the capture of large amounts of debugging information, which, eventually, helped to identify errors. The master's thesis by Linux LANE developer Marko Kiiskilä [11] also provides much better documentation than

what was provided with the distribution. LANE continues to be our most fragile configuration.

There are three demons which must be started up to make LANE work: the signaling demon, `atmsigd`, the ILMI demon, `ilmid`, and the LANE client demon, `zeppelin`. In addition, there are two required servers: the LES and the BUS. There must also be a LAN Emulation Configuration Server, unless `zeppelin` is given the address of the LES explicitly. We did not need to configure multiple ELANs, so we bypassed the LECS. Even so, we needed unique ATM addresses for the LES and BUS separate from the host ATM addresses. This is where the SEL byte of the ATM address becomes useful. The network does not consider the contents of this byte for making routing decisions, but the Linux software does distinguish between addresses differing only in the value of the SEL byte. By choosing different values, we could configure each host to support more than one ATM address.

Each ELAN must contain one LES and one BUS. The physical location of the servers, however, is open. They may reside in one host, run on different hosts, or they may reside in the switch itself. We tried running them on machine `atm` and on the switch but eventually settled upon the switch as the best place for them. When the servers were on one of the hosts, far too many connections were used for our limited supply. As mentioned earlier, each host sets up two connections to the LES and two to the BUS. Data may be only temporarily sent over one of the connections to the BUS and must thereafter have its own connection. When we ran LES and BUS on `atm`, this meant that we also had two connections from LES to `atm`, two from LES to `chattooga`, two from BUS to `atm`, and two from BUS to `chattooga`, for a grand total of 12 connections on `atm` before any data were sent across the switch. If we had more machines connected to the ELAN, an additional four connections from `atm` for each additional host would be necessary for connections to the servers. Data connections between `atm` and the other hosts would consume even more connections.

In addition to the connections to the LES and BUS, data must be sent over a separate connection. For reasons that are not at all clear, three connections are set up at the onset of data transfer. It may be that these represent new connections to the LES and BUS as well as one connection for data transfer. If this is the case, there does not seem to be a good reason why the existing connections could not have been used for the LE ARP request and reply. After, connection timeout, two new connections are opened each time.

Another advantage of placing the responsibility for LES and BUS in the switch is that a configuration file `.lanevars` is required by the Linux implementation if the servers were to run on one of the Linux hosts. By allowing the switch to take over these functions, the configuration file was not needed.

Errors in specifying the various ATM addresses, either in the option to the `zeppelin` demon or in the switch configuration, generally resulted in an error message like the following:

```
CM NERR 'conn.c':  Bind failed:  Cannot assign requested address
```

A specific problem with our original invented ATM addresses was that we had inadvertently

made them all multicast addresses. The software was quick to notice our lapse and gave us the following reminders:

```
LECCTL NERR 'lec_ctrl.c' Join Request Rejected by LES
Error = Invalid_LAN_Destination
DEBUG: conn: MAC address not found from database
DEBUG: conn: Destination lan address is multicast
```

The high order byte of all our ESI fields in the ATM addresses was 99. However, when the low order bit of the high order byte in an Ethernet address is turned on [16], the address is a multicast address. We changed the byte to 18, which cleared both the high order and low order bits.

7.3.2 Linux Code Bugs

At this point, we were reasonably confident that our ATM addresses were correct and that we were starting up the demons in the proper order with valid options, but LANE was still failing to start. By luck, a posting to the ATM-Linux mailing list[4] pointed us to the solution: there were several bugs in the Linux code for which patches were available. The first patch was by Heikki Vatiainen[5] of Tampere University of Technology in Finland, who had this to say about the problem.

> Since you are using an IBM switch it is possible that LES and BUS share the same ATM address. If this is true, then zeppelin (wrongly) refuses to connect to the BUS since it already has a connection open to LES.

In addition to the patch for `zeppelin`, there were also patches by Werner Almesberger for the signaling demon. His comments about the bugs are below.

> What happens before is less healthy: SSCOP rejects a perfectly valid STAT PDU. The problem seems to be that I'm comparing the poll sequence number in the data sequence number space, which is of course a stupid idea...

> Here's another one: if recovering from an SAAL restart and releasing calls at the same time, atmsigd was ignoring the STATUS(0) and waiting for the RELEASE COMPLETE instead.

We applied the suggested patches to `conn.c`, `sscop.c`, and `q2931.c` and finally got LANE working.

[4] Posted Monday, December 15, 1997

[5] also known as "the zeppelin guy"

7.3.3 Other Miscellaneous Oddities

Connection timeouts were not just limited to CLIP connections. Idle connections in LANE typically timed out every 5 to 6 minutes. This was traced to a LANE client parameter which the ATM Forum calls C17, or Aging Time. C17 is "[t]he maximum time that an LE Client will maintain an entry in its LE_ARP cache in the absence of a verification of that relationship." [7] The default and maximum value of this parameter is specified to be 300 seconds and the minimum is 10 seconds. This value is set in `lec_ctrl.c` to 300 seconds. Each time a connection times out, a new connection is quickly set up with a new VPI.VCI. Before we upgraded the switch, this meant that our connections went out of range very quickly. This is currently not a problem, as the connection identifiers correctly wrap on reaching the limit.

We also noticed that the VCIs that were assigned were not sequential. It turned out that the switch still expected the machine `atm` to be the ATMARP server, even though we were running LANE and not CLIP. The switch was frequently trying to contact the ATMARP server, which was, of course, not running. Each attempt used up another connection identifier. The switch only needed to know the address of the ATMARP server if we were using the switch itself as an end point for data transfer. This is normally not the case so we set the IP address of the ATMARP server in the switch to the switch itself.

A problem that we were unable to correct involved the UNI. At present, we are using UNI 3.0. UNI 3.1 is purportly supported by both the switch and the Linux code. We tried upgrading the Linux demons to use 3.1 and could successfully fire up CLIP, but we were totally unable to get LANE to run. Error messages included:

```
atmsigd:  Q2931:  Cause 100 (invalid information element contents)
```

and

```
atmsigd:IO: TO NET: DROP_PARTY_ACK (0x84) CR 0x800001 (28 bytes)
atmsigd:Q2931: Cause 102 (recovery on timer expiry)
```

It appears to be a failure of the signaling protocol using UNI 3.1.

Wayne Salamon [15] reported problems with the ATM software and UNI 3.0. It did not appear, however, that they were using LANE. We had no problems with CLIP with either UNI 3.0 or 3.1.

8 Performance

Two measures of data transfer performance are throughput and CPU utilization. In this section, throughput measurements for CLIP using PVCs, CLIP using SVCs, and LANE

using SVCs are compared. The effect of maximum transmission unit size on throughput is also examined, and an attempt is made to measure CPU utilization for CLIP.

Throughput is a measure of the amount of data transferred per unit time. However, the number of bytes actually transferred includes not only the data to be sent but overhead such as headers and trailers which are included in the packet. The length of an ATM cell is fixed at 53 bytes: 48 bytes of data and 5 bytes of header. Additionally, each AAL5 frame has an 8 byte trailer at the end. If the length of the frame is not evenly divisible by 48, the ATM protocol requires that padding bytes be appended to fill out the last cell.

Suppose there are F bytes of application level data to send. Assume for the moment that the data will be contained in a single TCP/IP packet. Forty bytes of TCP and IP headers will be attached. In addition, if Classical IP over ATM is used, each packet will have an 8-byte LLC/SNAP header in accordance with RFC 1483. With LAN Emulation, there will be a 16-byte header attached if the type of network being emulated is IEEE 802.3/Ethernet or a 16 to 36-byte header attached if the emulated network is IEEE 802.5/Token Ring. When the packet reaches the ATM protocol levels, the 8-byte trailer is appended. The data are then divided into 48-byte cells with padding, if necessary, to fill out the last cell. Each cell has a 5-byte header added. The total number of cells created with CLIP will be at least

$$\frac{(F + 40 + 8 + 8)bytes}{48bytes/cell}$$

The total number of bytes transmitted is

$$\frac{53(F + 56)}{48}bytes$$

However, this is a simplified view. TCP buffers data for sending. The actual amount of data sent in a single network level send operation cannot be predetermined. The data could end up divided into more than one frame, in which case, more than one set of TCP/IP headers will be included. However, there will be at least one set of headers. The amount of padding added is also indeterminable, but cannot be more than 47 bytes per packet. An upper limit on the possible throughput can be calculated as follows. The physical layer of the APE25PCI adaptor offers a nominal bit rate of 25.6 Mbps. Hence, the maximum throughput at the application layer using CLIP can be no more than

$$25.6\left(\frac{48F}{53(F + 56)}\right)Mbps.$$

By a similar computation, the maximum possible throughput for LANE can be shown to be

$$25.6\left(\frac{48F}{53(F + 64)}\right)Mbps.$$

In the following graphs, all data were collected at the transmitter `atm`. which is the faster of the two machines.

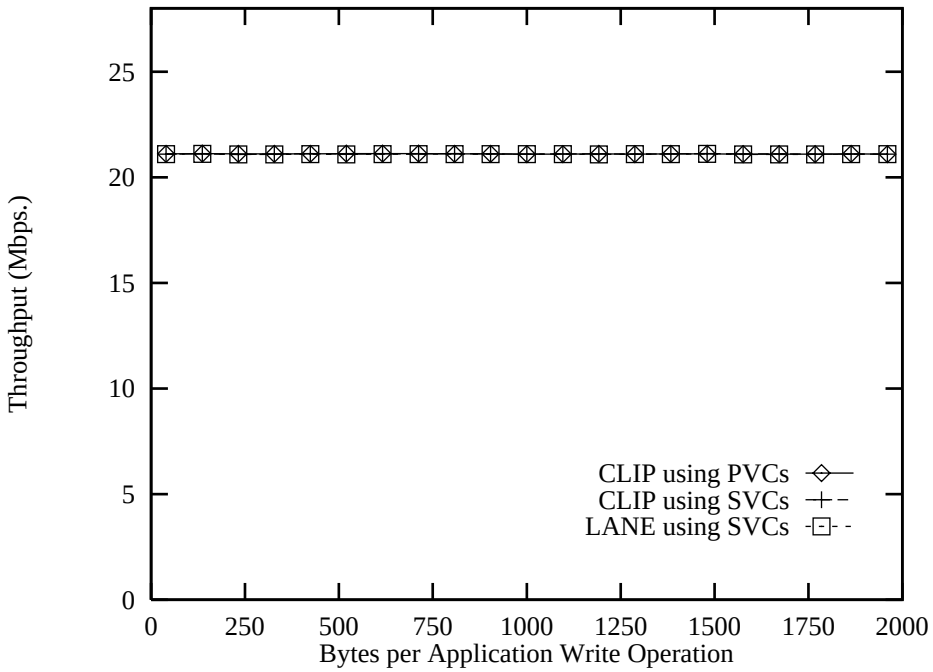

Figure 12: Throughput for CLIP using PVCs and SVCs and for LANE using SVCs. Maximum transmission unit set to 1500 bytes.

Figure 12 shows the throughput measurements for CLIP using PVCs, CLIP using SVCs, and LANE using SVCs. The application level write sizes were varied between 40 and 1960 bytes with an increment of 96 bytes. A single data buffer was retransmitted as fast as possible until a total of 50 Mbytes were sent. For each transport method a total of three benchmark runs were performed with the mean of the three reported. The graph makes it clear that there was no discernible difference between CLIP and LANE. In spite of the slightly higher header overhead of LANE, the difference was not detectable. Most of the differences in overhead between CLIP and LANE will be at connection setup time, not data transfer time. The flatness of the graph gives evidence of TCP's buffering of send requests. Typically, the amount of data sent in each physical transmission is equal to the MTU which was held constant at 1500 bytes for this test. The MTU of 1500 bytes was chosen because that is the largest MTU supported by LANE.

The throughput measurements for various MTUs are shown in figure 13. Since LANE does not permit MTUs greater than 1500, only CLIP using SVCs was used. Except for very small packet sizes, increasing the MTU increases the throughput. Larger MTUs will mean fewer sends at the physical layer. Notice that the increase in throughput is not linear. The larger the MTU, the smaller is the increase in performance. The formula above for maximum possible throughput for CLIP shows that, at an MTU of 1000 bytes, the transmission operates at about 91% of the maximum throughput possible. At 8000 bytes, the throughput is more than 97% of the maximum possible (except at packet size of 40 bytes).

At the smallest packet size the performance gain was reversed. The larger the MTU, the

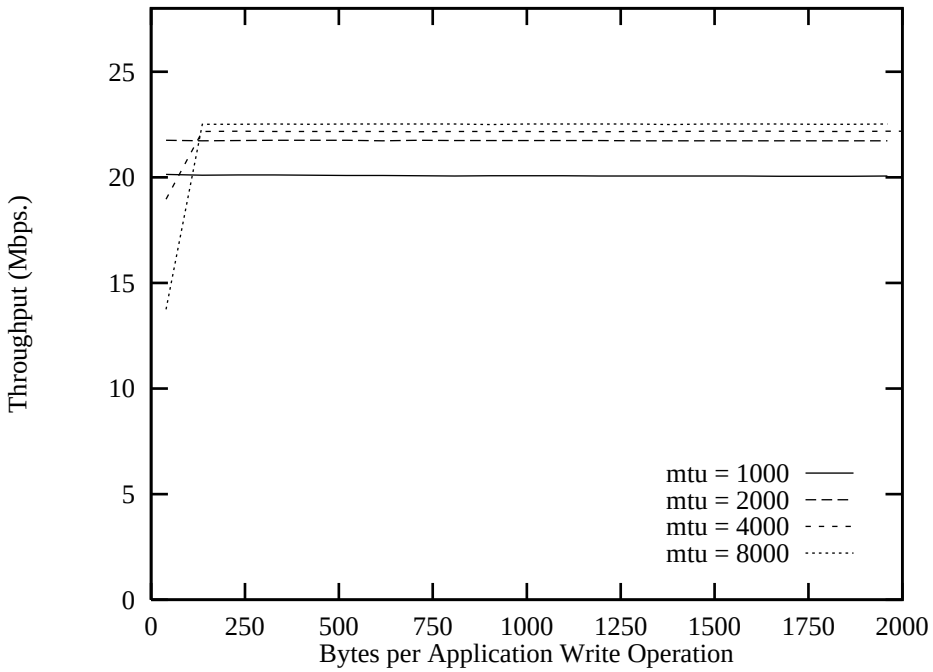

Figure 13: Throughput for CLIP using SVCs for a range of MTUs.

worse the performance became. It appears that the CPU becomes a bottleneck at that packet size. The larger the MTU, the longer it takes to fill up a frame. The network is idle waiting for the next packet to go out, and, thus, is not being fully utilized.

Figure 14 shows the fraction of CPU time used for two different MTUs using CLIP with SVCs. Although only the two graphs are shown, tests were run for other MTUs and for LANE. All were similar to the graph for MTU = 2000. The periodicity of this graph is related to the fact that TCP does much of its work during timer interrupts. Assignment of CPU usage during interrupt service routines is not simple and does not appear to be accurately handled.

9 Evaluation of CLIP and LANE

The evaluation of Linux CLIP and LANE should not be limited to comparisons of their throughput. In deciding which of the two to choose for an ATM LAN, several other factors should be considered.

CLIP supports much larger data packets than LANE. While CLIP can support MTUs up to 9180 bytes, MTUs with LANE depend on the type of network being emulated. For IEEE 802.3/Ethernet, the MTU is 1536. Tests performed in this project indicate that higher MTUs generally mean greater throughput.

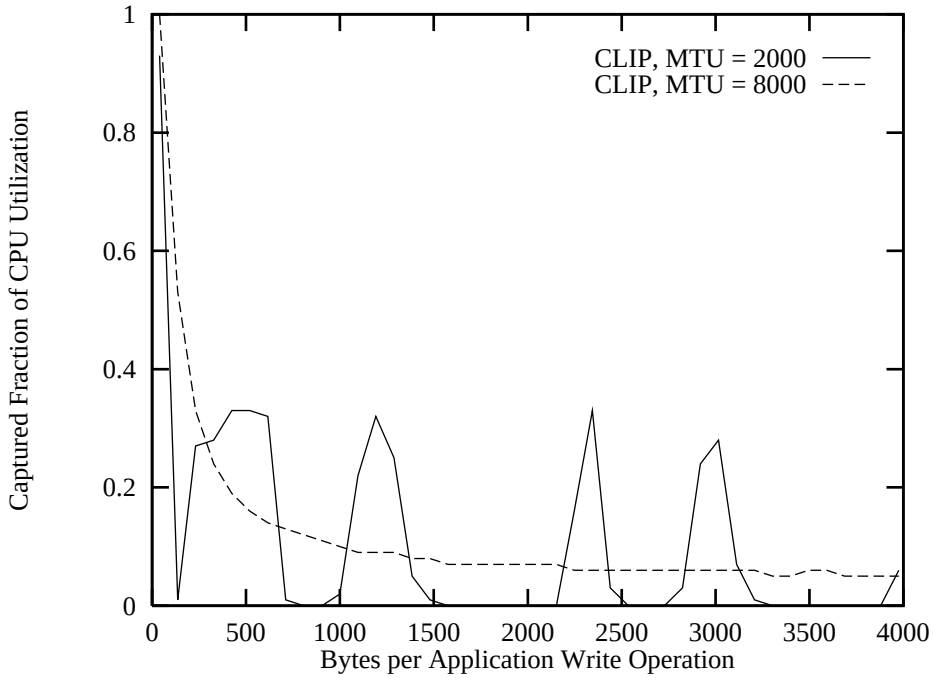

Figure 14: CPU Utilization Using CLIP with SVCs and MTUs of 2000 bytes and 8000 bytes.

CLIP also requires fewer connections than LANE. In an environment where the range of available VCIs is limited, this can be an important consideration. The entity that hosts the LES must be able to support four connections for each end system on the ELAN: a Control Direct VCC and a Control Distribute VCC for connecting to the LES, and a Multicast Send VCC and a Multicast Forward VCC for connecting to the BUS. Even data connections seem to require more than one VCC in LANE. There could easily be a shortage of connection identifiers on even a small ELAN.

The Linux implementation of LANE does not support a PVC only environment. This can be a deterrent in the initial stages of setting up IP over ATM, because it forces the network administrator to deal with the problems of integrating the signaling demon and the ILMI demon at the same time as the LANE servers and the client demon.

Most importantly, the Linux implementation of CLIP was far more stable than LANE. New versions of the Linux ATM code or patches may address the problems seen, but Linux LANE is clearly a "pre-alpha" version at the present.

There are also arguments in favor of LANE. Multiple Logical IP Subnetworks can only be joined by routers. Emulated LANs, however, can be interconnected via bridging technology, which is easier and less expensive than routers [11]. LANE supports multicast and broadcast packets, and CLIP lacks this ability. In addition, LANE is designed to support other protocols besides IP, while CLIP is limited to IP over ATM.

Some defects are common to both implementations. Quality of service parameters are only supported for single hop transmissions. Direct ATM end-to-end connectivity beyond subnet

boundaries and negotiation of QoS requirements are not currently supported.

Both CLIP and LANE suffer from the single point of failure problem. Failure of the ATMARP server in CLIP or any of the LES, BUS, or LECS in LANE will cause failure of the entire subnetwork.

Goralski [9] favors LAN Emulation over Classical IP over ATM. One should note, however, that he was evaluating the protocol, not a specific implementation of the protocol. For our test bed, CLIP was clearly the better choice.

10 Future Research

Asynchronous Transfer Mode is such a new and rapidly developing technology that there is almost no limit to the possibilities for future research. Even if the field is limited to the rather narrow focus of IP over ATM, there are still many areas to investigate.

Application Requested IP over ATM (AREQUIPA) allows applications to request a direct end-to-end ATM connection beyond subnet boundaries with negotiation of QoS. The connection is reserved for the exclusive use of that application. AREQUIPA is included in the ATM Linux distribution and could be integrated into the system configuration.

It would also be interesting to investigate interoperability of the Linux ATM implementation with other operating systems which have CLIP and LANE support such as OS/2.

This project made no attempt to vary the quality of service requirements. It may be possible to set up multiple interfaces with differing QoS requirements.

MultiProtocol Over ATM (MPOA) is a cooperative effort between the IETF and the ATM Forum to allow "shortcut" connections between hosts in different subnetworks without involving the services of a router. As envisioned, MPOA will overcome many of the shortcomings of both CLIP and LANE. As such, it will be an important area of research, although not currently implemented in Linux.

11 Conclusions

The integration of the Linux implementations of Classical IP over ATM and LAN Emulation with the the device driver code by Geist and Westall, the APE25PCI adaptor, and the configuration of IBM's 8285 ATM switch was successful. While both LANE and CLIP were capable of sustaining stable connections and had equivalent performance at an MTU of 1500 bytes, CLIP was the preferred method of running IP over ATM. The reasons for this evaluation were:

1. Better throughput was achievable via higher MTUs with CLIP. LANE does not support MTUs greater than 1500 for Ethernet emulation.

2. CLIP required fewer connections between hosts. This is a significant fact when the total number of available connection identifiers is limited.

3. LANE occasionally falters during start up even though it seems stable once it is up.

In addition, detailed documentation from a code-level view was developed of the data transfer process using CLIP with SVCs. Such documentation is needed but has been largely nonexistent.

References

[1] W. Almesberger. ATM on Linux—The 3rd Year. 4th International Linux Kongress, Würzburg, ftp://lrcftp.epfl.ch/pub/linux/atm/papers/atm_3rd.ps.gz.

[2] W. Almesberger. ATM on Linux. 3th International Linux Kongress, ftp://lrcftp.epfl.ch/pub/linux/atm/papers/atm_on_linux.ps.gz, March 1996.

[3] R. Atkinson. *Default IP MTU for use over ATM AAL5.* RFC 1626, May 1994. http://www.internic.net/rfc/rfc1626.txt.

[4] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *LINUX Kernel Internels.* Addison-Wesley, 1996.

[5] R. Colella, E. Gardner, and R. Callon. *Guidelines for OSI NSAP Allocation in the Internet.* RFC 1237, July 1991. http://www.internic.net/rfc/rfc1237.txt.

[6] ATM Forum Technical Committee. *ATM User-Network Interface Specification Version 3.1.* ATM Forum, September 1994. ftp://ftp.atmforum.com/pub/approved-specs/af-uni-0010.ps.tar.Z.

[7] ATM Forum Technical Committee. *LAN Emulation Over ATM - Version 1.0.* ATM Forum, January 1995. ftp://ftp.atmforum.com/pub/approved-specs/af-lane-0021.000.pdf.

[8] R. Geist and J. Westall. Bringing the High End to the Low End: High Performance Device Drivers for the Linux PC. In *Proceedings 36th Annual ACM Southeast Conference.* ACM, 1998.

[9] W. J. Goralski. *Introduction to ATM Networking.* McGraw-Hill, Inc., 1995.

[10] J. Heinanen. *Multiprotocol Encapsulation over ATM Adaptation Layer 5.* RFC 1483, July 1993. http://www.internic.net/rfc/rfc1483.txt.

[11] M. Kiiskilä. Implementation of LAN Emulation Over ATM in Linux. Master's thesis, Tampere University of Technology, 1996. ftp://viulu.atm.tut.fi/pub/misc/linux-lane.ps.gz.

[12] M. Laubach. *Classical IP and ARP over ATM.* RFC 1577, January 1994. http://www.internic.net/rfc/rfc1577.txt.

[13] P. Paiva. IP over ATM, February 1995. Part of diploma thesis at Swiss Federal Institute of Technology, Lausanne, ftp://lrcftp.epfl.ch/pub/atm/ipatm/ppiatm.ps.gz.

[14] M. Perez, F. Liaw, A. Mankin, E. Hoffman, D. Grossman, and A. Malis. *ATM Signaling Support for IP over ATM.* RFC 1755, February 1995. http://www.internic.net/rfc/rfc1755.txt.

[15] W. Salamon. Experiences Building an ATM/Linux Cluster. http://cmr.ncsl.nist.gov/scalable/publications.html.

[16] W. R. Stevens. *TCP/IP Illustrated, Volume 1, The Protocols.* Addison Wesley, 1994.

[17] A. S. Tanenbaum. *Computer Networks, Third Edition.* Prentice Hall PTR, 1996.

[18] J. Westall. ATM Networking in Linux. Technical report, Clemson University, September 1997. Presented to IBM.

A Acronyms

AAL	ATM Adaptation Layer
AREQUIPA	Application Requested IP over ATM
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
ATMARP	Asynchronous Transfer Mode Address Resolution Protocol
BUS	Broadcast and Unknown Server
CCITT	Consultative Committee for International Telegraph and Telephone
CLIP	Classical IP over ATM
DCC	Data Country Code
DMA	Direct Memory Access
ELAN	Emulated LAN
ESI	End System Identifier
FDM	Frequency Division Multiplexing
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
ILMI	Interim Local Management Interface, Integrated Local Management Interface
IP	Internet Protocol
ITU-T	International Telecommunications Union-Telecommunications Standardization Sector
LAN	Local Area Network
LANE, LE	LAN Emulation
LEC	LAN Emulation Client
LECS	LAN Emulation Configuration Server
LES	LAN Emulation Server
LIS	Logical IP Subnetwork
LLC	Logical Link Control
LUNI	LAN Emulation User to Network Interface
MAC	Medium Access Control
MIB	Management Information Base
MPOA	MultiProtocol Over ATM
MTU	Maximum Transmission Unit
NIC	Network Interface Card
NNI	Network Network Interface (also Network Node Interface)
NSAP	Network Service Access Point
NVRAM	Non-Volatile Read Only Memory
OAM	Operations and Maintenance
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PVC	Permanent Virtual Connection

RBH	Receive Buffer Header
RFC	Request for Comments
RFL	Receive Free List
RRL	Receive Ready List
QoS	Quality of Service
SMDS	Switched Multimegabit Data Service
SNAP	SubNetwork Attachment Point
SNMP	Simple Network Management Protocol
SVC	Switched Virtual Connection
TCP	Transport Control Protocol
TDM	Time Division Multiplexing
TFD	Transmit Frame Descriptor
UNI	User Network Interface
VC	Virtual Circuit
VCC	Virtual Channel Connection
VCI	Virtual Circuit Identifier
VPI	Virtual Path Identifier