Stanford Ethernet Applebus Gateway (SEAGATE)

Bill Croft

Stanford University
Medical Center
SUMEX Project *, rm TB105
Stanford, CA 94305
croft@sumex.arpa

beta release, 1/85

ABSTRACT

This note explains how to make your own gateway between ethernet and applebus. Such a gateway allows UNIX (or other) systems on the ethernet to act as *servers* for the Macintosh.

1. Introduction

This note describes SEAGATE, a *gateway* (Apple term: *bridge)* that connects an ethernet using the DARPA *internet protocols* (IP), to an applebus using Apple or IP protocols. The IP protocol family was chosen because many campuses and engineering groups are using it on their ethernets; most such groups have access to Berkeley UNIX. With such a gateway in place, it becomes possible to create UNIX server daemons to provide file, printing, mail, etc. services for the Macintoshes.

In addition, it would be possible for the UNIX systems to become integrated into a *Macintosh Office* such that UNIX users could access Apple provided services such as printing on a *LaserWriter* or sending mail to Macintosh users via an Apple file server.

This distribution of SEAGATE provides all the information and software you should need to setup your own gateway. Please bear in mind that this distribution is not 'supported' and that we can't give extensive help about the mechanics of putting your gateway together. I would like to hear about bug reports or enhancements however.

2. Protocol packages / servers

UNIX provides a large number of IP based servers. With a stripped down C based IP package, many of the UNIX user level programs (such as TELNET and FTP) could be ported over to the Mac straightforwardly. Alas, such a package does not yet exist. [I could envision creating such a package by sniping sections out of the 4.2 BSD UNIX kernel].

What does exist currently is a port of the MIT IP package (for the IBM PC) to the Macintosh. This was done by Mark Sherman of Dartmouth in the summer of 84. Since there were no commercial C compilers available at the time, Mark transliterated the MIT code from C into Workshop Pascal. At this writing, the TFTP (trivial file transfer protocol) and TIME (fetch time-of-day from server) programs from MIT have been ported. These programs work correctly between Macintoshes, or through the gateway between a Macintosh and UNIX. Written by MIT, but as yet unported are the TCP and TELNET packages.

While this porting was a large and admirable project, I am not sure that it is the right base to build Mac IP services upon. For one thing, the MIT TCP implementation (in the original C) is incomplete and

* SUMEX-AIM (Stanford University Medical Experiment in Artificial Intelligence) is supported by the Biotechnology Resources Program, National Institutes of Health.

cannot handle data streams in both directions (it's only good enough for TELNET, where the sending stream is low volume). My hope is that someone will take a relatively full and debugged IP package and adapt it to the Mac, all in the C language.

Meanwhile, the gateway provides another alternative. All Apple services on applebus are based on the applebus datagram protocol, called DDP (datagram delivery protocol). In addition to passing IP packets back and forth, the gateway will do a small amount of protocol conversion: if it receives a DDP from the applebus destined for the ethernet, it will 'wrap' it with an IP/UDP header, doing appropriate address and port number conversions. This allows Apple DDP services to be written as UDP daemons on UNIX, without requiring any UNIX kernel changes.

Conversely, a UDP packet received by the gateway from the ethernet will be converted to a DDP (by stripping the IP/UDP headers) if the UDP destination port number matches a certain 'magic number'. While these protocol conversion functions are currently compiled into the gateway, and easily altered, one could also imagine them being selected dynamically based on any packet fields (such as host address). This would allow for hosts that understand DDP packets directly at the kernel level.

3. Addressing and routing

3.1. IP 'nets' versus 'subnets'

The gateway can be configured to treat each (ether/applebus) cable as a separate IP 'net' number, or as separate IP 'subnets'. Unless you are at a site which implements subnets, such as Stanford, MIT, or CMU, you will probably use plain 'net' numbers.

As mentioned above, the gateway can translate DDP addresses (2 bytes of net number, 1 byte of node number) to IP addresses (4 bytes total for both net number and node number). When subnets are NOT used, a mapping table inside the gateway is used to convert between network/node numbers. The information to setup this table is in the **Configuration** section below. If your site does not use subnets, you can probably skip or skim over the next couple sections below.

3.2. Subnets

An IP address can be divided into two parts: the network number, and the 'local address', which is the part not used for the network number. For example at Stanford, our net number (for the whole campus) is 36, so our addresses look like: 36.XX.XX.XX. Within Stanford, we can use the last three bytes anyway we wish. 36 is an example of a 'class A' network number, which is one byte in length. There are also class B and class C network numbers, which are two and three bytes in length, respectively.

A large organization is likely to have many cable segments comprising its one IP network number. To help gateways and hosts within such a network, it is useful to adopt a 'subnet' notion. In this scheme, one of the 'local address' bytes refered to above, is taken over by the 'subnet number'. This subnet number can then be used by gateways (and/or hosts) to locate the particular cable segment on which the destination host is found. At Stanford we use the second byte of the IP address; for example, 36.12.00.05 would refer to local host address 5 on subnet (cable) 12 on network 36.

When the gateway is configured for subnet operation, each applebus cable segment (DDP network number) is assigned an IP subnet number. For example, a given IP subnet number (12 in the example above) is mapped onto the same applebus DDP network number 12, and vice versa. The position of the subnet byte within the IP address (second or third byte) can be altered with configuration parameters.

For IP hosts sitting on the ethernet or applebus, this 'subneting' requires no modifications to their IP network code. This is because the gateway participates in the ARP broadcast protocol used by a host to find a 'hardware address' (6 byte ether, or one byte applebus), given the IP address. For example, if an ethernet host with address 36.44.00.01 wishes to find applebus host 36.12.00.05, the ethernet host broadcasts an ARP saying in effect 'who has IP address 36.12.00.05'.

The host being sought would answer directly if it were on the same cable, but it isnt. Instead, the applebus gateway sees that someone is asking for an address on subnet 12 (his own applebus segment). So the gateway answers the ARP, making the ethernet host THINK the desired host has responded.

The gateway does this ARP stuff in both directions: ether hosts looking for applebus hosts are guided in the right direction. Applebus hosts looking for a subnet number OTHER THAN the applebus subnet number, are guided over to the ether side.

3.2.1. Subnet addressing limitations

Given the small size of applebus segments, isn't devoting an entire IP subnet to each segment somewhat 'wasteful'? For example, here at Stanford we have already used 30 or so of our 256 possible subnet numbers, for ethernet cables. If we have already got this many ethernets, imagine how many applebuses could pop up.

I dont know what the answer is. The IP address space is awfully tiny. There are only 128 class A networks, and 16000 class B numbers. There are two million class C numbers, but these are useless for most subnet schemes.

If you are not officially part of the DARPA internet, you can pick any class A or B number that you like. But this seems rather anarchistic. Perhaps the new ISO protocols have the answer: I have heard they allocate a dozen bytes or so for the internet address. This seems more in line with the explosive growth in network interconnectivity we are seeing.

Certainly the most natural mapping between physical cables from one protocol family to another, is to use part (or all of) the network number fields in each family (as we have done). However the scarcity of IP subnet address space may force other approaches:

1 The gateway could make its applebus hosts look like part of the same subnet that is used by the ethernet side of the gateway. The gateway could do this by responding to ARPs for a subrange of the hosts on a given ether subnet. This would complicate the protocol and address conversion issues.

2 In a class B subnet scheme, there are 16 bits of address left up to the local administration. Perhaps we should be partitioning these 16 bits in a way other than 8 bits of subnet and 8 bits of host. Perhaps 10 bits of subnet and 6 bits of host would be more realistic.

3.3. Routing protocols

The gateway broadcasts an applebus RTMP (routing table maintenance protocol) packet every 30 seconds. This informs the Macintoshes of the DDP network number of their applebus cable.

When routing a packet, if the IP (major) network number of the destination does not match that of any interface, the packet is forwarded to a 'default' gateway specified at configuration time. In the subnet case, the gateway assumes that there are other 'smarter' gateways or hosts that will answer ARPs for subnets not matching its own.

Actually IP gateways are supposed to use a 'standard' protocol between themselves (IGP or EGP) to decide how to find other nets and to advertise the accessability of their locally attached cables. At least at Stanford, this has not happened yet. For example, the Stanford gateways still use the old Xerox PUP routing table protocols to inform themselves of subnet connectivity. What we will probably do for the time being is add another little 30 second timeout task to the applebus gateway. It will broadcast (on the ether side) a trivial PUP routing table entry showing the connection to its applebus segment(s). Such code would be #ifdef'ed by our site name, so that you could substitute your own local convention.

3.4. Protocol conversion

Incoming applebus DDPs which are destined for the ethernet are encapsulated in IP/UDP datagrams. The IP fields are derived as follows. First, the incoming DDP must obviously be in 'long' format, since a destination network number is required. This net number either (1) becomes the destination IP subnet number or (2) is converted to an IP address through the mapping table. The one byte DPP 'node numbers' become the low byte of the IP addresses. The DDP 'socket' numbers are mapped into UDP 'ports' as follows: if the DDP socket is between 0 and 127 (a 'well known socket'), it is translated to the UDP port range 0x300 to 0x37F, the top of the reserved port range on UNIX. If the DDP number is between 128 and 255, it is mapped to the range 0x4000 to 0x407F, an unreserved port range on UNIX.

Once again, all of this mapping and conversion is isolated to a small set of mapping functions, so it is easily changed. One could also imagine altering the default mappings by sending a message to a small mapper daemon (socket) within the gateway.

3.5. DDP routing

At present the gateway only really knows about routing IPs. In the future it would be desirable to participate more in applebus RTMP protocol, and to allow the ethernet (or even the whole DARPA internet) to be used as a long-haul backbone between applebus segments.

4. Prerequisites

To assemble your own gateway, you will need at least the items below:

The hardware is a 3 card multibus system: A 'SUN' 68000 CPU board, an Interlan NI3210 ethernet card, and a homebrew applebus card (about 8 chips) which takes an afternoon to wirewrap. More details in the hardware section below.

A UNIX (usually VAX) running 4.2 BSD, 4.1 BSD or Eunice. This is because the source distributed is written in the PCC/MIT 68000 C compiler. [This is the same compiler included with the SUMACC Mac C cross development kit.] You can probably make due with any 68K C compiler and assembler, but it will be harder.

Inside Mac, update service, and the *Mac software supplement.*

Applebus Developer's Kit, includes: protocol manual, applebus taps and interconnecting cable, Mac applebus drivers on SONY disks.

Dartmouth's IP package from Mark Sherman (mss%dartmouth@csnet-relay). The gateway distribution includes the binary for TFTP, but if you want the whole package (and source), you should get it from Mark.

A Lisa Workshop system is handy to have around; you would need it to compile Mark's sources. Even if you are doing development in C, Apple releases Applebus updates as a combination of Mac and Lisa disks. The Mac disks contain the 'driver' binary resources. The Lisa disks contain source for header files.

5. Hardware used

5.1. CPU board

The SUN CPU board is a common one used here at Stanford for our gateways and controller projects. We buy ours now from Forward Technology, model FT68-X. It has a 10 MHz 68000 CPU, 256K of RAM, a simple PROM monitor (used to download the gateway), 2 RS232 serial ports, clocks, etc. Although the FT68-X supports DMA via the multibus, the gateway does not require use of DMA. We made this decision since many older 'SUN' CPU boards do not have DMA support.

You will need a 50 cond. ribbon cable that terminates in two ribbon DB25 connectors; these are the console and serial downline load lines. The gateway takes less than a minute to load at 9600 baud from your UNIX.

5.2. Ethernet board

The Interlan NI3210 is about the least expensive multibus ethernet card we have found. It has a low part count because it uses the Intel 82586 ethernet chip to do most of the work. In addition it contains 8K bytes of on board memory that (our driver) chains together as a group of circular buffers. This lets the applebus card be somewhat lazy about ethernet interrupt latency.

In addition to the NI3210, you need two cables from Interlan: the NM10-10 is a flat ribbon cable that connects to the card; the NA1040-10 is the standard round transceiver cable that goes from one end of the flat cable to the tranceiver. The '-10' means 10 feet long.

You can use any 'standard' ethernet tranceiver. The NT10 from Interlan seems to work well and is both V1.0 and 2.0 compatible.

5.3. Applebus board

The wiring of the applebus card is more fully described in the attached file ’seagate.hard’. The main chip it uses is the Zylog Z8530 serial communications controller (SCC). This is the same chip used inside the Macintosh. The Z8530 does complete SDLC framing, checksum, address matching, FM0 encoding and clock extraction. The current card is not DMA (neither is the Macintosh). The software driver for the link layer (LAP) is a modified version of the driver that is inside the Macintosh; after the address match interrupt comes in, the driver polls the SCC every 35 usecs for a new byte.

This is the one item that could potentially limit the capacity of the gateway. Given the current gateway and driver architecture, it would be straightforward to replace the current card and lap driver, with a DMA version. The DMA could be done either to some dual ported memory on board the applebus card, or directly to the processor memory (if the FT68-X was used).

Our current applebus card was designed around an ARTEC multibus ’slave’ prototype card, which contains all the bus interface circuitry for an interrupt/polled controller. Other companies (such as Electronic Solutions) offer multibus ’master’ prototype boards, that come with the logic necessary for a multibus DMA interface. It may be simple to fit the Z8530 into such a board.

Another possibility would be to take an existing multibus DMA SDLC board (surely one must exist?) and use a ’daughter board’ with the proper clock, RS422, and FM0 decoding logic. Richard Brown at Dartmouth (richb%dartmouth@csnet-relay) has used a similar scheme to adapt a standard SDLC interface in their existing terminal concentrators, to applebus.

Yet another lead: I have heard a rumor the Heurkion 68000 CPU board uses the Zylog SCC chip. SUN Microsystems also uses it on their ’SUN 2’ CPU boards. Possible hangups: do they allow DMA to the SCC? What about the clocking and RS422 connections? Can these boards be purchased separately, or only as part of an entire workstation?

5.4. Other hardware.

We use a 6 slot multibus card cage and backplane, from Electronic Solutions. The power supply is a small switcher from Power-One, model SHQ-150W. You may want to just buy an integrated card cage, box, and power supply.

6. Software organization

The gateway software has a simple structure: the applebus and ethernet receive-interrupt routines grab packet buffers off a free list, stuff them with data, and enqueue them on the gateway main packet queue ’pq’. At interrupt level 0 (backround), the gateway pulls packets off this queue and routes them as appropriate. Eventually the ethernet or applebus output routines are called. When there are no packets on the input queue, the backround just ’idles’ performing console IO and scheduling timer routines.

Each interface has an ’interface structure’ (ifnet) similar to that used in 4.2 BSD. It contains, per interface, the IP (software) address, the hardware address (ethernet or applebus), and some parameters used by the ARP handler.

Here is a brief rundown of the source files used in the gateway:

ab.h	Applebus protocol header definitions. If ’MAC’ is #defined, some Mac OS specific structures are included.
arp.c	The code for the IP address resolution protocol. The same code runs on both the applebus and ethernet sides, by getting the hardware specific information from the ifnet structure.
ether.h	Standard 10 megabit ethernet header definition.
gw.c	The main gateway code.
gw.h	General gateway definitions.
gwasm.[hs]	Some assembler support routines for the gateway.
il.[hc]	The driver for the interlan/intel ethernet controller.

inet.h	Internet Protocol header definitions.
lap.[hs]	Applebus LAP driver.
lapref.s	Some special hooks (hacks?) needed to allow LAP polling to coexist with the software memory refresh on the SUN cpu board.
pbuf.[hc]	Packet buffer get/free.

7. Configuration

7.1. Software

In file gw.c, choose an IP address for each gateway interface and substitute it the 'ifnet' structures. You will only need to change that one element in each structure initialization.

Also in gw.c, 'iproutedef' is the default route used when no matching interface can be found. It should be an address of a gateway on the ethernet cable.

Depending on your local ethernet configuration, you may need to add some routing table entries to your gateways or hosts, so that packets for the applebus can first hop over to the applebus gateway.

If you are using subnets, set the variables 'ipsubmask' and 'ipsubshift' to correspond to the values used in your situation. If you are NOT using subnets, set 'ipsubmask' to zero, and fill some entries in the 'ipmap' structure; see the comments on that structure for proper format and examples.

gw.h contains the socket mapping functions (macros); you will probably not need to alter these. Also in gw.h is a 'dartarp' #define, with a comment which reads:

> Dartmouth implemented the 1st IP package for the Mac (by porting the MIT IBM PC version). Dartmouth originally placed the ARP protocol on top of DDP, instead of LAP (all other IP ARPs are directly on top of the hardware link level). Until Dartmouth changes their package, this define should be used.

7.2. CPU board

CPU board jumpers: The Forward FT68-X board has a number of jumper areas. In general, these have correct settings as shipped from the factory. Just as a check list here are our settings: JP1 (no +5 on J1): none; JP2 (RS232): 2-7, 4-9, 5-6; JP3 (2732 PROMs): 1-3; JP4 (test points): none; JP5 (uart:level5int, timer:6) 2-4, 1-3; JP6 (drive BCLK, CCLK, ground BPRN, receive bus INIT): 1-2, 5-6, 7-8, 9-10; JP7 (DMA from BA19): 1-3; JP8 (INT4 to 1 from bus): 7-8, 9-10, 11-12, 13-14; JP9 (cpu board): 1-2, 4-5; JP10 (20 bit addressing): 3-4, 7-8, 11-12

The Forward CPU PROM also looks at connector J2 on reset to get some configuration information. [Our Stanford PROMs for the Forward board are somewhat smarter than this, but they have our net booting code wired in; so you can get by just fine with the default Forward PROMs.] Here are the Forward PROM jumpers: 17-18 (don't clear memory on reset); 19-20 ('verbose' mode); 30-31 (don't run diagnostics on reset); 45 (optional: can be momentarily grounded for 'reset'). Actually, you have to leave out 30-31 to get the board memory management to power up in a reasonable state. What I do is leave out 30-31 and live with the fact that reset will clear memory. This gateway has some code (in the refresh interrupt) to check the console 'break' key status, so you CAN get back to the PROM monitor without a reset.

7.3. NI3210 ethernet board

Again, the factory settings are pretty good; here are ours: W19A (serial multibus arbitration); W2A, W3A (surrender to other master); W20A (16 bit IO address); W21A, W22B, W23A, W24B (IO address XXAX); W4A, W5B, W6A, W7B, W8A, W9B, W10A, W11B (IO address AAXX); W12B, W13A, W14B, W15A (memory address 5XXXX); [note A/B reversal] W16B, W17B, W18B (memory addr X0XXX); [note reversal] W25-32B (24 bit address 050000); W1A (DC xcvr coupling).

7.4. Applebus board

S1 = IO map; S2 = 0110 0000 0000 (sets IO address 6000); S3 = all bits of address significant; ACK DELAY: XACK to CCLK*6.

8. Operation

8.1. Downloading

Plug in the boards and connect the tranceivers/taps. Connect a 9600 baud terminal to serial port A, and a 9600 UNIX tty line to port B.

Power up the gateway, you should get a message on the console: ’Self test successful’, and then the monitor prompt, a ’>’. Type an ’x’ followed by a control-uparrow, followed by a carriage return (CR). This ensures that ’control-^’ is your console escape character. Now type ’it’ followed by CR. This puts the console and tty line in ’transparent’ mode, so you can log onto UNIX and prepare for the download.

Change directory into the place where the gateway code lives. The loadable binary there is called ’b.out’. Escape back to the PROM monitor by typing control-uparrow, then type the letter ’c’ (stands for ’close’ connection). You are now back in the monitor. To download the gateway, type ’l dlq’ CR. ’l’ is the monitor load command; the monitor transmits the remainder of the command line (’dlq’) to serial port B. This executes the ’dlq’ command on UNIX (which stands for ’download, quick’). Dlq uses the file ’b.out’ by default. Dlq will now handle the handshaking with serial port B to complete the download.

You will see a ’.’ print on the screen every 256 bytes that are loaded. After about 30 seconds the load will be complete and the gateway code will begin executing. As a debugging aid, the gateway has linked into the binary, a symbolic ddt. When the gateway starts, this ddt enters a temporary breakpoint and can accept commands at this point. See the ddt manual page in the ddt directory. Normally you will just start the gateway by typing the ’esc’ key, followed by ’g’ (stands for ’go’). The gateway should now be operational.

8.2. Console ’commands’

While the gateway is running, it accepts a few trivial console commands. Typing the letter ’s’ will print some interface statistics. You can escape back to ddt with a ’control-c’. You can then peek or poke around at locations. [use ’pc’+2, ’esc’g to resume]

8.3. Debug printouts

When first installing the gateway, you may want to get packet traces on the ethernet and/or applebus side to check that things are working. The applebus ’Peek’ and ’Poke’ programs can help out here as well; using peek is preferable since it doesnt impose any additional delays.

With ddt, you can open location ’dbsw’ (debug switch) and place a value there. Dbsw is a bit mask, with each bit turning on a different section’s trace printout. For example, setting dbsw to the hex value FF will turn on all traces; setting it to 1A will just print the headers of ethernet and applebus output packets. See ’gw.h’ for the other switch values.

8.4. TFTP usage

Two Dartmouth/MIT programs are included: TFTP and Customize. Customize is first run to setup a small parameter file on the Mac that holds the Mac’s IP address, and the IP address of the gateway; these numbers are four byte values, specified in hex. The other fields shown are unused.

After setting up the parameter file once, you can then run TFTP. Select from the command menu either ’get’ or ’put’. If you are talking to a UNIX, you cannot use the Mac specific ’Macintosh Mode’; instead select ’ascii’ or ’octet’ mode. Fill in the ’local file’ name and the ’remote file’ name (the dialog mistakenly says ’destination’ instead of ’remote’). Finally fill in the remote host; alas, it too is specified as a hex four byte host number.

When you now click ’OK’, the transfer should begin. You can watch the packets flying by if you have a ’peek’ running on the net.

In Mark Sherman’s document on his IP package, he warns that this release of TFTP etc., is not to be considered ’production quality’. TFTP ’gets’ seem to crash after doing about 3 consecutive transfers. ’puts’ seem more stable, but even these start to bomb out after the ’log’ screen starts to scroll.

9. Throughput

Using Mark’s TFTP and the Berkeley 4.2 BSD TFTP daemon, we made some simple timings. On the Mac side, TFTP used a ramdisk to avoid any delays induced by the slow SONY drive. For a UNIX to Mac transfer, we found that the Mac took 43 ms between data received and ack sent, while UNIX spent 25 ms between ack received and next data sent

Since these times were from the applebus peek program, the mac time is artificially high since it includes the 20 ms or so of packet transmission time on applebus (35 usec / byte). So then, each side has about a 20 ms delay before responding.

Most of the transfer occured at 512 data bytes every 70ms = 7314 bytes / sec = 58K baud.

Note however that the IP TFTP protocol is just that, a ’trivial’ FTP. It is purely half-duplex in nature. When we start using Apple’s ATP, which can stream several packets per acknowledgement, it should boost throughput significantly. Gursharan Sidhu tells me that their process-to-process (no disks) ATP throughput is 180K baud (out of the 230K available on the cable). This is very good, considering many TCP’s running on 10 megabit ethernet are lucky to get a few hundred kilobits of thoughput.

10. Future plans

Here are some obvious things that could be done next.

Use a DMA applebus interface. I guess right now we really don’t know how the present interface will hold up under heavy load: it may do just fine. But certainly, if you wished the applebus gateway code to coexist with some other gateway or ethertip (terminal concentrator) code, DMA would be most considerate of those other processes running on the cpu.

With DMA, you could have one gateway handling a whole group of applebus interfaces. Alternatively, perhaps it would be best to connect up the applebus segments as a true ’internet’ interconnected mostly thru Apple supplied bridges, and have just one or two ’seagate’s connecting that whole internet to the ethernet. Certainly this would limit thruput more than the 1st approach.

Before you can add multiple interfaces, the ’routeip’ and ’routeddp’ routines need to be beefed up a bit to scan a group of interfaces rather than just assuming two interfaces per gateway. [This is commented in the code.]

Here is the most interesting thing I would try: try to get the ’per gateway’ cost way down, by building a single board version of it. I picked the Intel 82586 ethernet controller for just this reason: all you should need is a board with the 68000, memory, the 82586 and the Z8530. Hopefully you could get the cost down below $1000 per gateway. Then just sprinkle them around campus, using ethernet as your ’long-haul’ and applebus within a floor, or group of offices.

I would like to quickly finish an ATP subroutine package that runs on the UNIX side. This will allow rapid construction of applebus servers on UNIX. A program equivalent in functionality to FTP or TFTP should be less than 5 pages of Mac C code. [Since the Mac MPP applebus driver package is doing the ’dirty work’ of ATP for you].

11. Acknowledgements

Nick Veizades built and helped debug our applebus hardware interface. Mark Sherman’s Mac IP package allowed easy access to the UNIX TFTP daemon for general debugging. Gursharan Sidhu, the ’applebus architect’, deserves much credit for making this protocol family as simple and elegant as it is. Arnie Lapinig of Apple was always helpful when we needed another tap box or question answered.

In the Stanford network community, Bill Yundt supplied us with free hardware and Ed McGuigan kept the applebus updates flowing in our direction. Ed Pattermann (formerly SUMEX director, now at Intellicorp) made the mistake of turning us onto Macintoshes, when we ’should have been’ hacking on LISP machines.