SHIPNET: A REAL-TIME LOCAL AREA NETWORK FOR SHIPS

Robert Simoncic
Alfred C. Weaver
Brendan G. Cain
M. Alexander Colvin

Computer Science Report No. TR-88-15
June 6, 1988

SHIPNET: A REAL-TIME LOCAL AREA NETWORK FOR SHIPS

Robert Simoncic
Alfred C. Weaver
Brendan G. Cain
M. Alexander Colvin

Computer Networks Laboratory
Thornton Hall
University of Virginia
Charlottesville, Virginia 22903

(804) 979-7529

ABSTRACT

We have developed a real-time messaging system for token ring networks. The system is currently operational in a shipboard environment. The system conforms to the IEEE 802.2 LLC and 802.5 token ring standards and is consistent with the Navy's emerging SAFENET specification. We provide a library of 'C' routines which establish and manage sockets, transmit and receive messages, and report network status. We show a sample user interface for a basic datagram service and another for a reliable virtual circuit service. We present performance data from our implementation on PCs and PC/ATs. Using our software, a single transmitting station can generate up to 1.42 Mbits/sec, and we can process 100-byte messages every 4 ms. We can deliver 100-byte messages with an end-to-end latency of under 10 ms.

SHIPNET: A REAL-TIME LOCAL AREA NETWORK FOR SHIPS

1. BACKGROUND

Historically, electronic communication aboard ships has been accomplished using point-to-point wiring. Due to the recent rapid increase in the number and type of shipboard electronic devices, wiring a ship has now become a logistics nightmare. Both shipbuilders and vendors of marine electronics are now actively trying to replace this old-style wiring with modern local area networks.

Sperry Marine Inc. specializes in the design, fabrication, and sale of marine electronics. In 1986 Sperry Marine asked the Computer Networks Laboratory (CNL) at the University of Virginia to help them design a local area network which would replace all shipboard point-to-point wiring with a local area network, yet still retain the real-time communications characteristics of direct wiring. After a six month study, we selected the token ring LAN architecture as being best suited for the purpose and began building a prototype network (the Navy's SAFENET committee subsequently adopted a similar token ring architecture for Navy ships).

Our initial goal was to develop hardware and software which could effectively interconnect the various systems on a ship's bridge: autopilot, heading indicator, situation display, gyrocompass, collision avoidance system, radar, and navigation systems (e.g., Loran). In the future we will extend the network to include the cargo area, engine room, and officers' quarters.

Our analysis of the communications patterns aboard a ship suggested that our system needed to support three distinctly different types of traffic: (1) slow periodic messages (e.g., a latitude/longitude update once per minute), (2) background file transfer (e.g., moving a program or data file intermittantly), and (3) real-time control messages (e.g., the gyrocompass can generate up to 72 100-byte messages per second). The first two traffic types had negligible impact on performance, so our system was designed around the needs of the real-time control systems. Our resulting performance requirements were that (1) each network station should be able to process at least one hundred 100-byte messages per second; (2)

once generated, a message should be delivered within 20 ms; (3) the network itself would support a sustained data rate of at least one megabit/second; and (4) each network station would be based on "ordinary" microprocessor technology, in the class of an Intel 8086 or Intel 80286.

We acquired a commercial token ring network, the Proteon ProNET-4 [1], and then began the major task of developing a user-friendly, yet robust, real-time messaging system. This task was equivalent to rewriting all the communications services within the kernel of a modern operating system. We call the result SHIPNET.

2. THE USER INTERFACE

One possibility for the user interface would be the full suite of OSI protocols; such a choice would have provided a messaging service in the application layer such as X.400 or the Manufacturing Messaging Standard, MMS. But we knew from our previous network performance evaluations [2,3,4] that the commercially available OSI protocol packages (1) would not meet our performance requirements and (2) were really overkill for a relatively short, single-segment LAN with a modest number of stations. Therefore we chose to write our own user interface, provided as a set of library functions which the user links into his application program. To encourage interoperability, our system adheres to the IEEE 802.2 logical link control (LLC) standard.

SHIPNET provides a basic datagram service, with optional acknowledgements and checksums, to multiple application processes running on microcomputers. The user interface is a set of 'C' procedure calls which create and manage sockets, set options, send and receive messages, and report network status.

The LLC architecture is shown in Figure 1. Communication occurs through *sockets*, which are equivalent to IEEE 802.2 LSAPs (link service access points). Programmers use the LLC interface by linking into their 'C' programs as many of the following communications primitives as are needed for the intended application:

LLCon (socks) initializes the network interface. Table space for *socks* number of sockets is allocated.

LLCoff( ) disables and closes the socket interface.

LLCopen (sock) opens the socket numbered *sock*. A socket must be opened before use. Sockets must be even-numbered integers in the range 2 to 254 inclusive.

LLCclose (sock) closes the socket numbered *sock*. The socket must be idle (no pending transmit or receive calls).

LLCoption (sock, xsignal, rsignal, xtime, rtime, priority, ack, tries) sets options on socket number *sock*. *xsignal* and *rsignal* are pointers to functions to be called when a packet has been transmitted or received, respectively. The variables *xtime* and *rtime* are the amount of time allowed for the operation to complete (a value of zero never times out). The value of *priority* sets the transmission priority of the message in the range 0 (lowest) to 7 (highest). The LLC software manages an eight-level queue, serving the highest priority messages first. The flag *ack*, if set, requires that the packet be specifically acknowledged by a reply message. When the token ring's frame acknowledgement bit shows that the destination did not receive a packet correctly, *tries* tells how many times the transmitter should transparently resend the packet before declaring an error.

LLCxmit (sock, destination, packet, size) delivers the packet pointed to by *packet*, of length *size*, to socket number *sock* for delivery to the network entity whose address is pointed to by *destination*.

LLCrecv (sock, source, packet, size) enables reception of a packet at socket *sock*. The programmer provides *source* (a pointer to a buffer to hold the incoming packet's source address), *packet* (a pointer to a buffer that will hold the incoming packet), and *size* (the length of the packet buffer).

LLCxmit and **LLCrecv** move messages between the LLC entity and the MAC engine in the same computer (not end-to-end). This frees the CPU to operate in parallel with the network hardware. Using IEEE 802.2 terminology, the procedure call represents the data *request*, the return from the call represents the *confirm*, and an appropriate change in the status byte provided by the **LLCstatus** call represents the *indication*.

LLCreset (sock, direction) resets any pending operation on the transmit side (if *direction* = *'x'*) or receive side (if *direction* = *'r'*) of the socket and releases the associated buffer. A socket is bidirectional and so can send and receive simultaneously.

LLCstatus (sock, direction, status) returns the current status of a socket. For socket *sock*, on the transmit or receive side as determined by *direction*, **LLCstatus** returns *status*, a pointer to a status variable, which indicates operation pending, no operation pending, I/O in progress, operation failed, or operation timed out.

Finally, every operation on a socket returns an operation status: operation accepted, invalid socket, duplicate socket, too many sockets, socket busy, or packet too large.

We also provide a real-time network monitor which displays color-coded histograms of recent network traffic. The monitor can trace all network traffic, or selected traffic as defined by a user-specified filter. The monitor displays network load in packets/sec and in bits/sec, sampled at a user-defined rate, and calculates current, average, and maximum data rates. In trace mode the monitor can trap and later display the last 1,000 network events.

3. EXAMPLE PROGRAMS

To show the simplicity of the user interface, we present two example programs. The first is extremely simple: a transmitter broadcasts an unacknowledged datagram on the network. The second is more complex: the sender creates a reliable virtual circuit to guarantee in-order delivery with no duplicates. This requires the use of acknowledgements and sequence numbers.

Services are provided by a software library which consists of several C subroutines. Subroutine calls implement socket operations through which programs are able to communicate. External names of subroutines which are important for the user interface to the system begin with 'LLC'.

When the LLC service is used, operations (LLC calls) must be issued in a certain sequence to obtain the desired result. The network interface must first be initialized, then one or more sockets may be

opened for communication and their options set. Only then can receive and transmit operations be applied. Each socket can be used for both reception and transmission in parallel; receive and transmit operations are full duplex. To close a connection the socket should first be reset, then closed. To stop all communication, each socket should be reset and closed, then the communications disabled.

If communication is to be established between two machines, one a receiver and the other a transmitter, a possible sequence of operations could be as follows.

At the transmitter:

1. **LLCon** will initialize the network interface and allocate table space for the socket.
2. **LLCopen** will open the socket.
3. **LLCoption** will set options on the socket for transmission.
4. Several **LLCxmit** operations can be issued to send a number of packets through the socket to another socket on the remote machine.
5. After all information is transmitted, **LLCreset** will reset the state of the socket.
6. **LLCclose** will close the socket.
7. **LLCoff** will disable and deallocate the socket interface.

At the receiver:

1. **LLCon** will initialize the network interface and allocate table space for the socket.
2. **LLCopen** will open the socket.
3. **LLCoption** will set options on the socket for reception.
4. The appropriate number of **LLCrecv** operations should be issued to receive all packets from the remote machine.
5. After all information has been received, **LLCreset** will reset the state of the socket.
6. **LLCclose** will close the socket.
7. **LLCoff** will disable and deallocate the socket interface.

3.1 Example 1 - A Broadcast Datagram Service

To illustrate LLC operations we offer two examples. Both are written in Turbo C 1.5 (as is all the LLC software). The first program, **blast**, can be run on any station. It broadcasts MSND messages of fixed length SIZE to all stations on the network.

```
/*******************************************************/
/*   Program "blast" will broadcast MSND messages      */
/*   of size SIZE to all nodes in the network.         */
/*******************************************************/

#include      <stdio.h>              /* standard I/O  */
#include      <llcio.h>              /* LLC interface */
#define       MSND    100            /* number of messages to send */
#define       SIZE    1000           /* size of the message     */
#define       BROADC  255            /* broadcast address */

int     cls     =       0,           /* service class */
        tout    =       0,           /* timeout       */
        retry   =       1,           /* retry limit   */
        ns      =       2,           /* source socket */
        mnum;
unsigned
  int  status;                       /* returned status */
char    message[SIZE]  =       { "Message..."};   /* broadcast message */
unsigned
  char  netadr[2]  =  { 2, 255 };      /* destination: socket number, node address */

nop() { }                            /* signal function */

down(opn)                            /* close down */
  char opn;
  { switch (opn) {
    case 'r' :
        while  (status = LLCreset(ns,'x'))
            printf("*** LLCreset rejected, status : %s\n ", LLCopbits(status));
    case 'c' :
        if (status = LLCclose(ns))
            printf("*** LLCclose rejected, status : %s\n", LLCopbits(status));
    case 'o' :
        if  (status = LLCoff())
            printf("*** LLCoff rejected, status : %s\n", LLCopbits(status));
    default:
        exit(1);
   }
}

main()                               /* transmit MSND messages */
  {
   /* initialize the network, open the socket and set the options */

   if  (status = LLCon(1)) {
     printf("*** LLCon rejected, status : %s\n", LLCopbits(status));
     exit(1);
   }
   if  (status = LLCopen(ns)) {
     printf("*** LLCopen rejected, status : %s\n", LLCopbits(status));
     down('o');
   }
   if (status = LLCoption(ns, nop, nop, tout, tout, cls, retry)) {
     printf("*** LLCoption rejected, status : %s\n", LLCopbits(status));
     down('c');
   }

   /* transmit MSND messages, wait for nonbusy status after each transmission */

   for (mnum = 1; mnum < MSND; mnum++) {
     if (status = LLCxmit(ns, netadr, message, sizeof(message))){
       printf("*** LLCxmit rejected, status : %s\n", LLCopbits(status));
       down('r');
     }
     while (LLCstatus(ns, 'x', &status), (status&STBUSY))
               ; /* wait until not busy */
   }
   down ('r'); /* close down */
  }
```

Program **blast** starts by initializing the network interface and allocating table space for one socket by calling **LLCon**. If **LLCon** is successful, **LLCopen** is called to open the socket.

LLCoption is used to set the socket options. If the returned status indicates unsuccessful operation, an error message is printed and the program exits with **down('c')** which will call **LLCclose** and **LLCoff** before exiting. A successful call to **LLCoption** in this program will set the following options for the socket ns=2:

1. The signal routine for transmit and receive is the same — **nop()** which does nothing.
2. Timeout for transmit and receive is set to 0, which means that operation will never be timed out.
3. Class is set to 0 — connectionless data link service without acknowledgements. Priority is 0 (lowest).
4. Number of transmission attempts is 1; LLC will not try to retransmit the packet if the first attempt fails.

Now the program calls **LLCxmit**. After each transmission, the return status of the operation is tested; if it indicates an error, the program exits. Otherwise it waits for the non-busy status of the socket, indicating that the socket is ready, and then issues the next transmit operation.

3.2 Example 2 - Reliable Virtual Circuit Service

In the second example we present one way to build a reliable virtual circuit service using the basic datagram service. It consists of two programs running on different stations. Program **sendrel** will send messages from a certain socket to the remote machine where program **recvrel** receives them. Then it waits for an acknowledgement from the receiver at the same socket. If the acknowledgement does not arrive within the specified timeout, the sender will retransmit the message and again wait for an acknowledgement. The sender will tag the messages with one bit sequence numbers, 0 or 1. For every new message transmitted, the receiver will alternate the sequence number. While retransmitting, the sequence number stays the same.

Program **recvrel** will receive messages from the sender. On each reception it will test the sequence number. If it is the same as the previous one, it means the message is being retransmitted by the

sender so the receiver just emits another acknowledgement. If the sequence number is different from the previous one, the receiver will process the message, send the acknowledgement, and await another message.

In this example the communication service is reliable. If either the message or the acknowledgement is lost it will cause retransmission from the side of the sender. The receiver correctly handles duplicate messages. The source code of **sendrel**:

```
        /*********************************************************************/
        /* "sendrel" - transmitter for the reliable communication service  */
        /*********************************************************************/

#include        <stdio.h>
#include        <llcio.h>               /* LLC interface */
#define         MSND    100             /* number of messages to send */

int     cls     =       0,              /* service class */
        toutr   =       60,             /* timeout units */
        touts   =       0,
        retry   =       0,              /* retry limit   */
        ns      =       2,              /* source socket number */
        dones   =       0,               /* set when message sent */
        doner   =       0,              /* set when acknowledgement received */
        rpt     =       10,             /* retries */
        mnum;
struct  {
        unsigned char seqn;             /* sequence number */
        char    message[100];           /* message sent    */
        } dout;
unsigned
int     status1,                        /* status returned */
        status2;
unsigned
char    netadr[2]  =  { 2, 100 };       /* destination: socket number, node address */
char    seqack;                         /* received acknowledgement */

/* receive signal handler  */
sigr() {if (LLCstatus(ns, 'r', &status1), !(status1&STBUSY)) doner++;}

/* transmit signal handler */
sigs() {if (LLCstatus(ns, 'x', &status2), !(status2&STBUSY)) dones++;}

opstat(s,tag)                   /* check the operation status and return it */
  int s;
  char  *tag;
  { if (s)
      printf("*** LLC%s rejected, status: \n \t %s \n", tag, LLCopbits(s));
      return (s);
  }

down(opn)                       /* close down */
  char opn;
  { switch (opn) {
      case 'r' :  while  (opstat(LLCreset(ns,'r'), "reset") ) ;
      case 'c' :  opstat(LLCclose(ns), "open");
      case 'o' :  opstat(LLCoff(), "off");
      default  :  exit(1);
    }
  }
```

```
main()                          /* reliable transmitter */
  {
    /* initialize, open and set options for the socket */
    if ( opstat(LLCon(1), "on") )  exit(1);
    if ( opstat(LLCopen(ns), "open") )  down ('o');
    if ( opstat(LLCoption(ns,sigs,sigr,touts,toutr,cls,retry), "option") ) down ('c');

    sprintf(dout.message, "MESSAGE....");     /* initialize out message */
    dout.seqn = 0;                            /* initialize seq number  */

    /* send MSND messages with 10 retries upon failure */

    for (mnum = 1; mnum < MSND; mnum++) {
       if ( rpt == 0 )  down('r');           /* if 10 retries -> down! */
       /***.........PREPARE THE MESSAGE HERE.............***/
       dout.seqn = (dout.seqn == 0) ? 1 : 0 ;        /* flip sequence number */
       printf( " Sending %d %s \n", dout.seqn, dout.message );
       rpt  =  10;     /* transmission retries */
       do {
           doner = dones = 0;          /* signal handler indicators */
           /* receive acknowledgement  */
           if(opstat(LLCrecv(ns, netadr, &seqack, sizeof(seqack)), "recv"))  down('r');
           /*  send message */
           if(opstat(LLCxmit(ns, netadr, &dout, sizeof(dout)), "xmit")) down('r');
           while ( !dones  || !doner  );     /* wait until done  */
           if (status1&STFAIL) {             /* transmission failed */
              printf("*** hexstat = %x\n", status1);
              printf("*** transmit failed, status: %s\n", LLCstbits(status1,'x'));
           }
           else
              if ( dout.seqn  == seqack )
                    break;                /* O.K. -> send next message */
                 else
                    down ('r' );          /* error -> down */
       } while ( (rpt--) , ( rpt > 0 ));
    }
     down('r');
 }
```

Program **sensrel** makes use of signal handlers for both transmit and receive operations. First **LLCstatus** is called to get the transmit or receive status of the socket; if it is not busy **doner** or **dones** is incremented to indicate the completion of the operation.

The main module first initializes the network interface, opens the socket, sets options for the socket and initializes the output message. Then it starts transmission and waits for the acknowledgement. If the acknowledgement does not arrive within the timeout the message is retransmitted up to 10 times. If the sequence number is out of order the program exits, else the next message is transmitted.

recvrel is the reliable receiver for **sendrel**. It runs on another station.

```
/*********************************************************************/
/* "recvrel" - receiver for the reliable communication service     */
/*********************************************************************/

#include       <stdio.h>
#include       <llcio.h>              /* LLC interface */
#define        MRCV    1000           /* number of messages to receive */

int     cls     =       0,            /* service class */
        toutr   =       100,          /* timeout units */
        touts   =       0,
        retry   =       0,            /* retry limit   */
        ns      =       2,            /* source socket number */
        dones   =       0,             /* set when acknowledgement sent */
        doner   =       0,            /* set when message received */
        mnum;
struct  {
        unsigned char seqn;           /* sequence number  */
        char    message[100];         /* message received */
        } din;
unsigned
int     status1,                      /* status returned */
        status2;
unsigned
char    netadr[2]  =  { 2, 5 };        /* remote: socket number, node address */
char    seqack =      0;              /* acknowledgement - sequence number */

/* receive signal handler  */
sigr() {if (LLCstatus(ns, 'r', &status1), !(status1&STBUSY)) doner++;}

/* transmit signal handler */
sigs() {if (LLCstatus(ns, 'x', &status2), !(status2&STBUSY)) dones++;}

opstat(s,tag)                  /* check the operation status and return it */
  int s;
  char  *tag;
  { if (s)
      printf("*** LLC%s rejected, status: \n \t %s \n", tag, LLCopbits(s));
      return (s);
  }

down(opn)                      /* close down */
  char opn;
  { switch (opn) {
      case 'r' :  while  (opstat(LLCreset(ns,'r'), "reset") ) ;
      case 'c' :  opstat(LLCclose(ns), "open");
      case 'o' :  opstat(LLCoff(), "off");
      default  :  exit(1);
    }
  }

main()                         /* receiver for reliable communication */
  {
    /* initialize, open and set options for the socket */
    if ( opstat(LLCon(1), "on") )  exit(1);
    if ( opstat(LLCopen(ns), "open") )  down ('o');
    if ( opstat(LLCoption(ns,sigs,sigr,touts,toutr,cls,retry), "option") ) down ('c');
    /* post the initial receive buffer for the message to arrive */
    if(opstat(LLCrecv(ns, netadr, &din, sizeof(din)),"recv"))  down('r');

    /* receiving messages and sending acknowledgements */

    for (mnum = 1; mnum <= MRCV; mnum++) {
        while ( !doner  ) ;             /* wait until receive done   */
        dones = doner = 0;              /* signal handler indicators */
        /* post another receive buffer for the message  */
        if(opstat(LLCrecv(ns, netadr, &din, sizeof(din)),"recv"))  down('r');
```

```
      /* receive failed or timeout? */
      if (status1&(STFAIL|STLATE)) {
         printf("*** hexstat = %x\n", status1);
         printf("*** transmit failed, status: %s\n", LLCstbits(status1,'x'));
         down('r');
      }
      /* is this the next message ? */
      if ( din.seqn != seqack ) {
         seqack = din.seqn;                    /* set the acknowledgement seq.number */
         /* send the acknowledgement */
         if(opstat(LLCxmit(ns, netadr, &seqack, sizeof(seqack)), "xmit")) down('r');
         while (!dones) ;     /* wait until done  */
         /***........PROCESS THE MESSAGE .........***/
         printf(" received: %d %s \n", din.seqn, din.message);
      }
      else {                                   /* retransmitted message */
          printf(" the same message \n");
         /* send the old acknowledgement */
         if(opstat(LLCxmit(ns, netadr, &seqack, sizeof(seqack)), "xmit")) down('r');
         while (!dones) ;     /* wait until done  */
       }
   }
}
```

After initializing the network, opening a socket, and setting its options, **recvrel** starts receiving messages from **sendrel** and acknowledges them one at a time by sending back the sequence number that was received in the message from the transmitter. If the receive operation fails or times out, the program exits.

sendrel and **recvrel** are written to run in parallel on two different stations. Together they guarantee reliable exchange of information.

4. NETWORK COMPONENTS

Our prototype network for this system is the ProNET-4, an IEEE 802.5-compatible local area network manufactured by Proteon Inc. The experimental environment consists of the following components:

PCs — IBM Personal Computer or compatible. We used the Leading Edge Model D with Intel 8088 microprocessor operating at 4.77 MHz (hereafter called the PC) and at 7.16 MHz (hereafter called the LED), and the PC/AT with Intel 80286 microprocessor operating at 8 MHz (hereafter called the PC/AT).

p1340 ProNET-4 Network Interface Board (p1340) for each station (PC, LED, and PC/AT) connecting them to the IEEE 802.5 LAN operating at 4 Mbits/sec. The p1340 is built around an integrated LAN adapter architecture using the Texas Instruments TMS380 chipset (TMS38030 System Interface, TMS38010 Communication Processor, TMS38020 Protocol Handler, and TMS38051-TMS380051 Ring Interface). All the programmable options for the p1340 were set to the factory defaults.

Pro4EUI The ProNET-4 Extended User Interface is a software and firmware product for use with Proteon's p1340 PC interface. The firmware portion is installed as an Upgrading Kit; the software portion is loaded at each station and stays resident in memory. Pro4EUI emulates IBM's Adapter Support Interface (TOKREUI) for their token ring adapter and provides **direct** and **data link control** (DLC) interfaces. Our LLC implementation is built on top of the direct interface.

p2700 The ProNET-4 Multistation Wire Center is the physical interconnect device for the Proteon ProNET-4 network.

Cabling Network cabling connects stations to the Multistation Wire Center.

5. PREDICTION vs. MEASUREMENT

Traditional analytic and simulation models of token rings often focus on *network* performance metrics such as throughput and utilization, or on *station* performance at the level of the station's interaction with the communications protocol (typically at the Media Access Control sublayer). While this approach can give accurate predictions at the MAC interface, it sheds little light on the performance which an actual user would see because it does not take into account the fact that (a) stations do not inject messages at the MAC sublayer (there is always some degree of higher layer processing, even if it is just framing at the LLC), and (b) stations often incur significant overhead in delivering messages to the communications subsystem.

Thus we have focused on performance as observed by the user, not the performance of the network itself. For a given message size, measurements of interest include the maximum message generation rate at a station, maximum station throughput, and message end-to-end latency. In general, these types of measurements expose a number of dependencies, including:

(1) the programming language in which the communications system was written,

(2) the proportion of protocol implemented in firmware vs. software,

(3) the efficiency of the user interface,

(4) the overhead of moving messages from user memory to OS memory, and from OS memory across a backplane to a system packet buffer, and

(5) the speed of the host processor and its DMA channel.

Our measurements give us a very accurate picture of the performance which one of our application programs can achieve.

6. PERFORMANCE RESULTS

There are three types of stations connected to the experimental network: PC, LED, and PC/AT. The application program runs on one of those hosts and exchanges messages with a peer application program on another host. Transmitter metrics refer to how fast the transmitting station can send to its peer; receiver metrics refer to how fast a receiver can process received messages; end-to-end latency is the elapsed time from the moment a message is enqueued by the sending application process until the moment the message is delivered to the receiving application process.

6.1 Transmitter Metrics

We show three metrics for the transmitting station:

(1) transmitter load (Kbits/sec) is the offered load which a single station can generate when sending messages of a certain length

(2) transmitter load (packets/sec) is the number of messages which a single station can generate per second

(3) transmitting station delay (ms) is the elapsed time between successive `LLCxmit` calls when

the transmitter is running in an infinite loop.

Figures 2, 3, and 4 show the relative performance of the PC, LED, and PC/AT machines. From Figure 2 we see that the transmitter on the PC can generate about 90 short (100-byte) or 35 long (2000-byte) packets/sec. The LED was about 33% faster than the PC for short packets and 27% faster for long packets, while the PC/AT was 230% faster than the PC for short packets and 150% faster for long packets. When compared with the PC, neither the LED nor the PC/AT maintained its speed advantage over the entire range of packet sizes. This is due to the different amounts of fixed overhead encountered with each different machine type. This maximum message generation rate will be important for applications where the concern is not absolute throughput, but rather how frequently a message may be sent to update some value (e.g., reading a sensor or commanding an actuator).

For 2000-byte packets, Figure 3 shows that the transmitter throughput was about 567 Kbits/sec for the PC, 772 Kbits/s for the LED, and 1.42 Mbits/sec for the PC/AT. This metric is important when the concern is the total amount of information to be transmitted, as in file transfer. Note that a single PC/AT is able to generate a load of 1.42 Mbits/sec, which is over 35% of the channel's 4 Mbits/sec capacity.

Transmitter latency, Figure 4, was another important metric. With the transmitter operating in an infinite loop generating messages, we measured the elapsed time between successive messages. The PC could generate short messages about every 11 ms, or long messages every 28 ms. The LED could generate short messages every 7.5 ms and long messages every 20 ms, whereas the PC/AT could emit short messages every 3.4 ms and long ones every 11 ms. Three important values on this graph are the station delays for 0 length packets, which are 10.1 ms, 7.0 ms and 3.1 ms for the PC, LED and PC/AT respectively. This value represents the transmitting station overhead for each packet which includes LLC, MAC and physical layer services. This is the cost every packet has to pay to be transmitted. From these numbers we can see how dependent the transmission rate is on the speed of the host processor. If we subtract this value from the appropriate curve on Figure 4, it gives us the actual transmitting station delay for user data alone. The transmitting station delay has three main components: (1) copying the data

from the user buffer to the LLC buffer (2.75 μsec/byte in the PC, 1.90 μsec sec/byte in the LED, and 0.41 μsec/byte in the PC/AT), (2) copying the data from the LLC buffer to the shared RAM (packet buffer) on the front-end processor (done by firmware), and (3) transmitting the data on the channel (4 Mbits/sec).

6.2 Receiver Metrics

The performance of the receiver was very similar to that of the transmitter. The receiver could receive about 6% fewer packets per second than the transmitter could transmit. This occured because the receiver process involved one more system interrupt than did the transmitter process. This means that if a transmitter sent a receiver two packets back-to-back, the receiver would occasionally miss the second packet. Ideally, the receiver should be faster than the transmitter.

6.3 End-to-end Metrics

For end-to-end measurements we consider throughput and end-to-end delay.

The transmitter process, the network, and the receiver process formed a three-stage pipeline for messages from one station to another. Of the three, the receiver process was the slowest and thus limited the speed of the pipeline. Throughput between any pair of user processes was thus about 6% less than the transmitter load shown in Figures 2 and 3.

End-to-end delay is the elapsed time from the moment that one application program sends a message (via `LLCxmit`) until it is received in another application program (via `LLCrecv`). This includes moving the message from user memory to the front-end processor, waiting for the token, transmitting the message, propagation delay, receiving the message, moving the message from the receiver's front-end processor to user memory, and all associated interrupt handling. This is a true measure of how long it takes to move messages from one peer process to another.

Figure 5 shows the end-to-end delay as a function of message length. Short (100-byte) messages could be moved in 19 ms, 13 ms, and 9.7 ms for the PC, LED, and PC/AT respectively; long (2000-byte) messages required 55 ms, 39.5 ms, and 29.5 ms respectively.

The delay for 0 length packets was 17 ms, 12 ms and 8.5 ms for the PC, LED, and PC/AT respectively. As with the transmitting station delay, this is the overhead every packet has to suffer to be delivered.

6.4 Broadcast Messages

At first glance it would seem that broadcasting a message would have a performance profile similar to that of sending messages to a specific receiver. Figures 6 and 7 show that this is not the case. Figure 6 compares the load which the PC and PC/AT transmitters can generate when using broadcast messages vs. normal messages. Figure 7 compares the transmitting station delay under the same circumstances. These figures show that broadcasting reduces the transmitter's generation capacity by as much as 50%, while as much as doubling the transmitting station delay. The cause, of course, is that a broadcast message is received by every station, including the transmitter. This means that the transmitter is slowed by the operation of the receiver which is sharing the same CPU.

6.5 Processor Idle Time

All our experiments have put the transmitter (receiver) in an infinite loop transmitting (receiving) packets. This drives the station's communications system at full speed. Under these conditions, we measured the processor's idle time — time which could otherwise be devoted to other tasks besides message handling. Figure 8 shows that this measurement ranged from effectively 0% to over 40%, depending upon configuration.

7. CONCLUSIONS

(1) It is possible to build an effective real-time communications service on modest equipment. The PCs and ProNET-4 equipment we used are unmodified, factory-standard products. Custom hardware would have increased our performance, but we could achieve our requirements without it.

(2) Ten LLC primitives are sufficient to define and manage sockets, send and receive messages, and report network status. With these primitives we can easily implement a basic datagram service or a

reliable virtual circuit service.

(3) Real-time communication, like a real-time program, works best if it is "lean and mean." Fewer primitives are better than more primitives if they are the right primitives! Our entire communications system occupies only 16K of code.

(4) Our requirements were that each station would process at least one hundred 100-byte messages/sec, with a delivery time of at most 20 ms. Using a PC/AT, a station could send or receive almost 300 messages/sec, one message every 3.4 ms. We could deliver 100-byte messages with a true end-to-end latency (i.e., user memory to user memory transfer time) of under 10 ms.

(5) Our receiver was about 6% slower than our transmitter. Ideally this should be reversed to reduce the probability of receiver overruns when two messages are transmitted back-to-back to the same destination.

(6) Broadcast messages extract a performance penalty from their transmitter because its receiver must process (and presumably discard) its own message.

(7) Even when the LLC framing is handled by firmware on a front-end processor, the speed and capacity of the host CPU markedly affects performance. The 8.0 MHz 80286-based PC/AT consistently outperformed the 7.16 MHz 8088-based LED which in turn consistently outperformed the 4.77 MHz 8088-based PC.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support of Sperry Marine Inc., Proteon Inc., and the Virginia Center for Innovative Technology.

REFERENCES

[1] Proteon p1340 User's Manual, Proteon Inc., Westborough, MA.

[2] W. Timothy Strayer and Alfred C. Weaver, "Performance Measurements of Data Transfer Services in MAP," *IEEE Networks*, May 1988.

[3] W. Timothy Strayer and Alfred C. Weaver, "Performance of Motorola's Implementation of MAP," *13th Local Computer Networks Conference*, Minneapolis, MN, October 1988.

[4] Jeffery H. Peden and Alfred C. Weaver, "Are Priorities Useful in an 802.5 Token Ring?", *IEEE Transaction on Industrial Electronics*, Vol. IE-35, No. 3, August 1988.

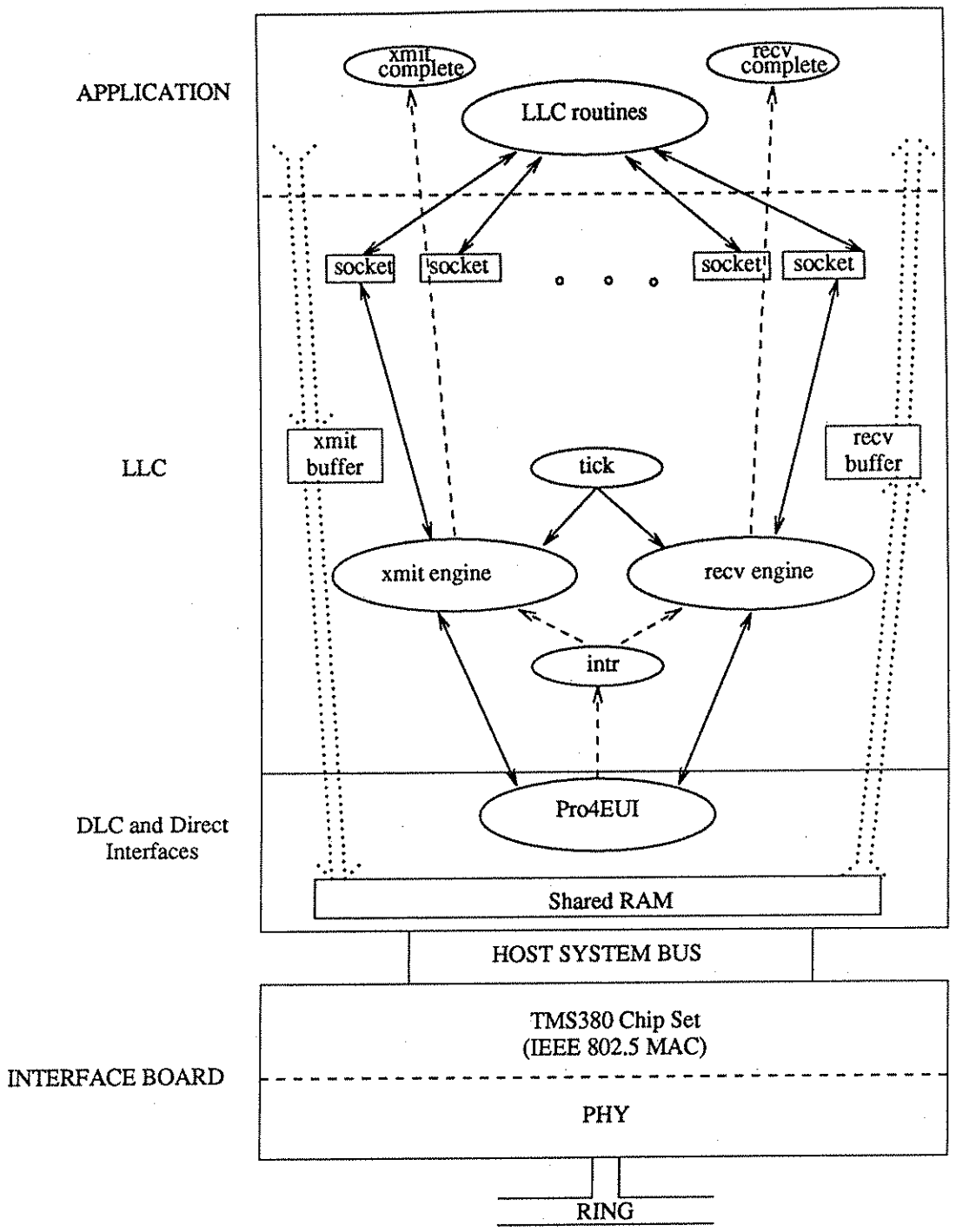
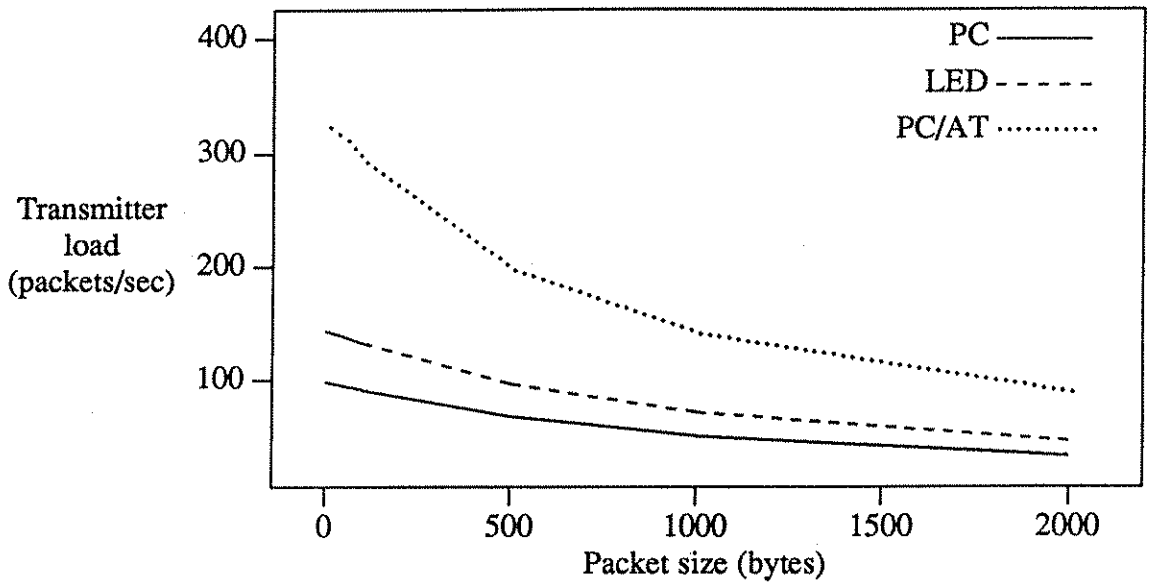
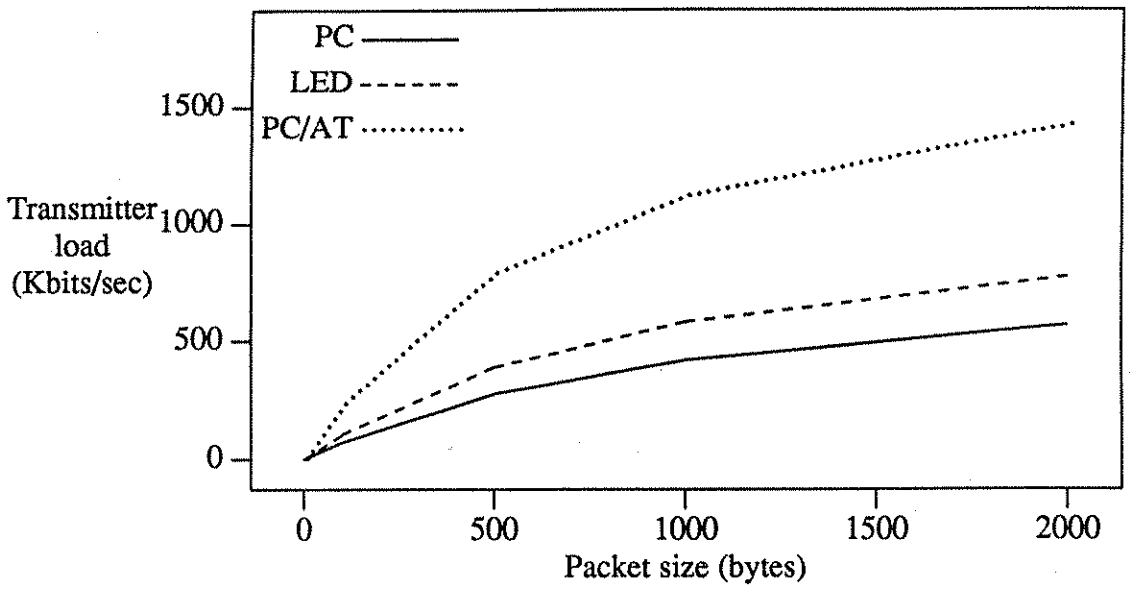


Figure 1.
LLC Architecture



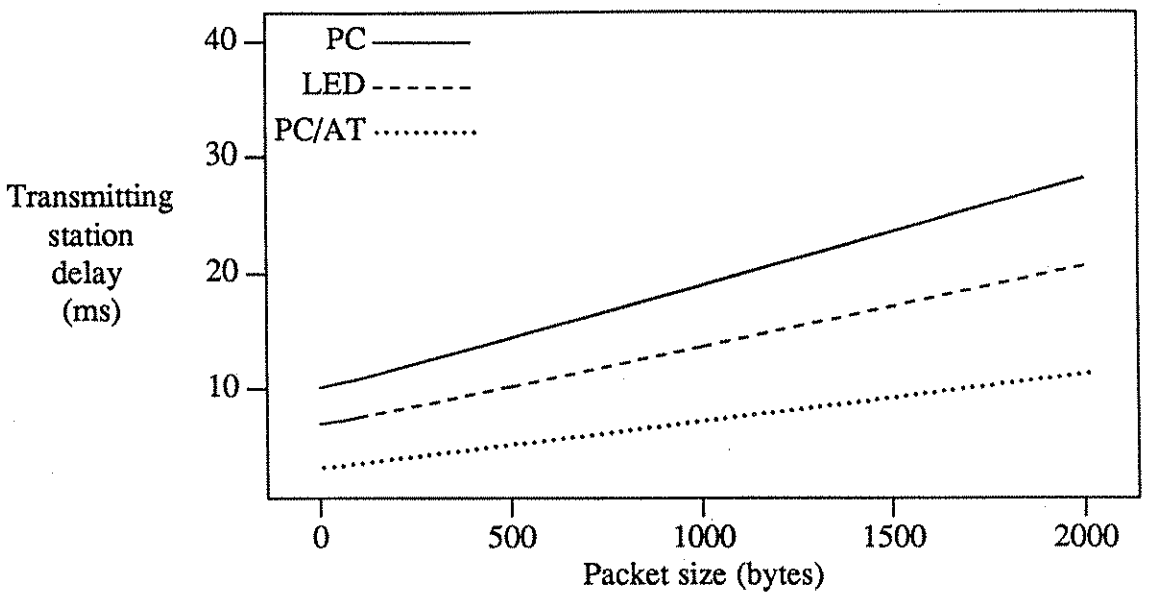
Message length (bytes)	Transmitter load (packets/sec)		
	PC	LED	PC/AT
0	98.9	143.3	322.1
100	92.1	132.7	291.2
500	69.1	97.3	196.3
1000	52.5	72.8	139.7
2000	35.5	48.3	88.8

Figure 2.
Transmitter load (packets/sec) vs. Packet size (bytes) for PC, LED, and PC/AT



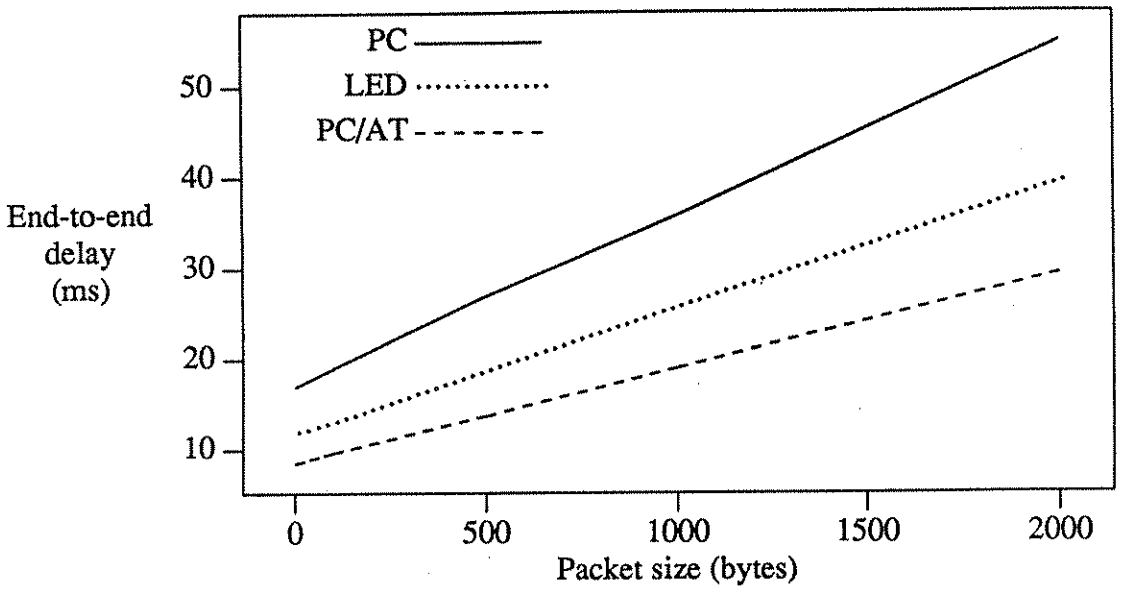
Message length (bytes)	Transmitter load (Kilobits/sec)		
	PC	LED	PC/AT
0	0.0	0.0	0.0
100	73.6	106.2	232.8
500	276.0	388.8	785.2
1000	419.7	582.4	1120.0
2000	567.2	772.8	1420.0

Figure 3.
Transmitter load (Kbits/sec) vs. Packet size (bytes)
for PC, LED, and PC/AT



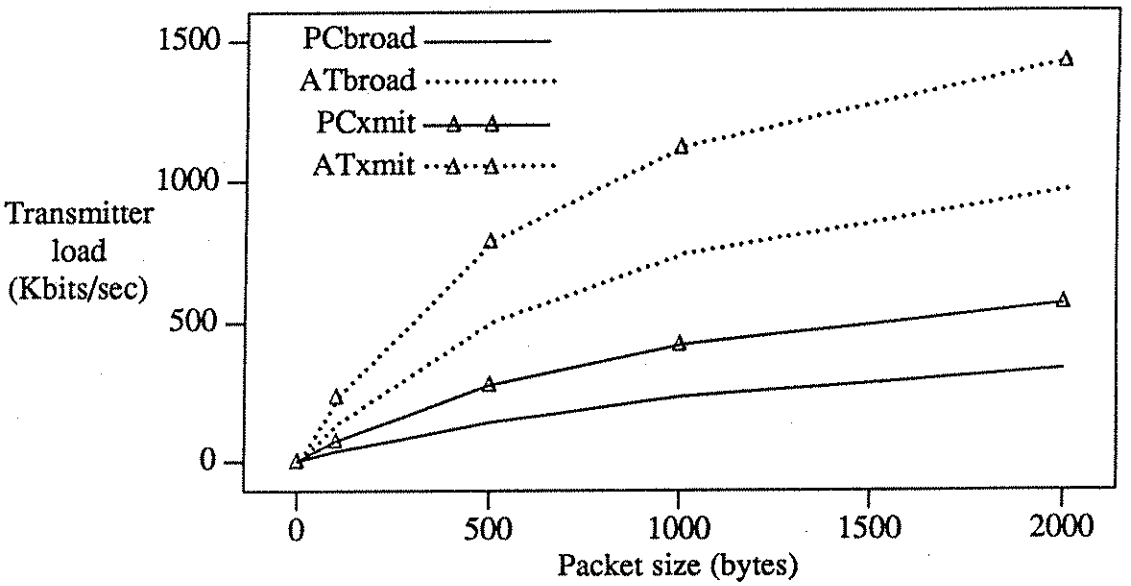
Message length (bytes)	Transmitting station delay (ms)		
	PC	LED	PC/AT
0	10.1	7.0	3.1
100	10.9	7.5	3.4
500	14.5	10.2	5.1
1000	19.0	13.7	7.2
2000	28.2	20.7	11.3

Figure 4.
Transmitting station delay (ms) vs. Packet size (bytes) for PC, LED, and PC/AT



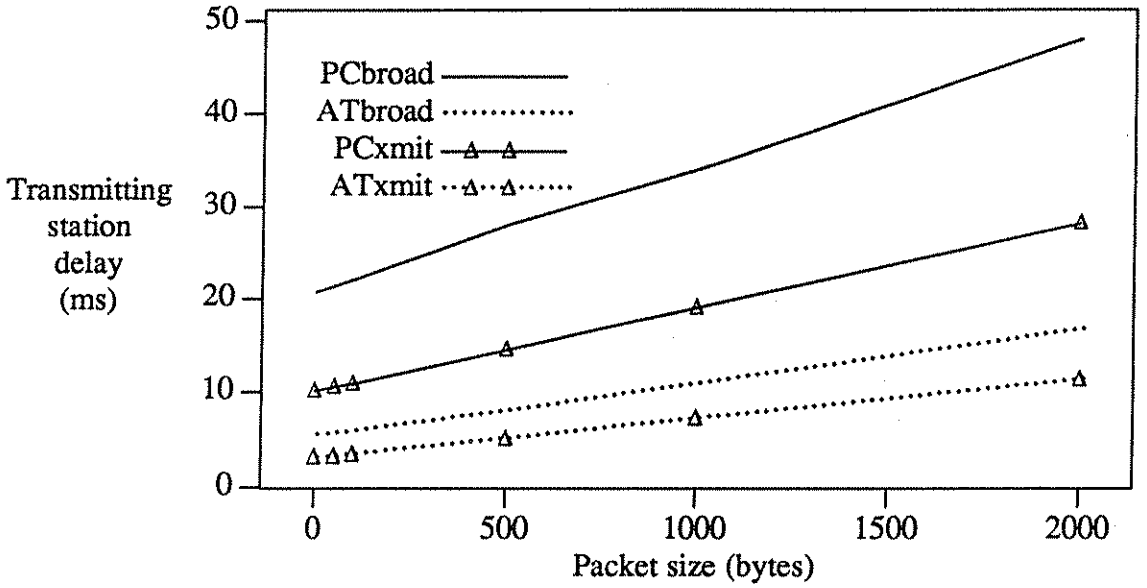
Message length (bytes)	End-to-end delay (ms)		
	PC	LED	PC/AT
0	17.0	11.8	8.5
100	19.0	13.0	9.7
500	27.0	18.7	13.8
1000	35.9	25.7	19.0
2000	55.0	39.5	29.5

Figure 5.
End-to-end delay (ms) vs. Packet size (bytes) for PC, LED, and PC/AT



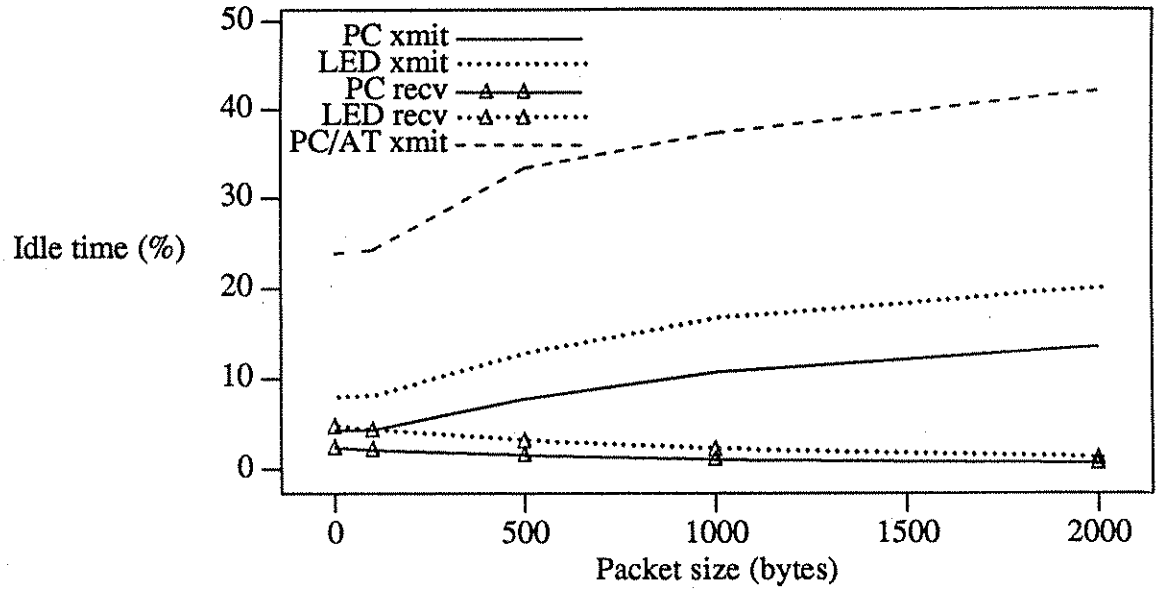
Message length (bytes)	Transmit and Broadcast load (Kbits/sec)			
	PCbroad	ATbroad	PCxmit	ATxmit
0	0.0	0.0	0.0	0.0
100	36.2	136.6	73.6	232.8
500	142.8	499.2	276.0	785.2
1000	233.1	736.1	419.7	1120.0
2000	333.3	963.4	567.2	1420.0

Figure 6.
Transmit and Broadcast load (Kbits/sec) vs. Packet size (bytes) for PC and PC/AT



Message length (bytes)	Transmit and Broadcast delay (ms)			
	PCbroad	ATbroad	PCxmit	ATxmit
0	20.7	5.4	10.1	3.1
100	22.1	5.9	10.9	3.4
500	28.0	8.0	14.5	5.1
1000	34.0	10.9	19.0	7.2
2000	48.1	16.6	28.2	11.3

Figure 7.
Transmit and Broadcast delay (ms) vs. Packet size (bytes) for PC and PC/AT



Message length (bytes)	Idle time (%)				
	PCxmit	PCrecv	LEDxmit	LEDrecv	PC/ATxmit
0	4.7	2.4	7.9	4.7	23.9
100	4.4	2.2	8.1	4.3	24.3
500	7.8	1.6	12.8	3.1	33.3
1000	10.7	1.1	16.6	2.2	37.3
2000	13.6	0.7	19.9	1.3	42.1

Figure 8.
Transmitter and receiver idle time (%) vs. Packet size (bytes)
for PC, LED, and PC/AT