NETMOS :-

A Local Area Network Simulation Context

By : Paul Marriott

Supervisor : Dr W. Kreutzer

CONTENTS

Chapter 1 : Background

Section 1.1 :- DEMOS

DEMOS is a package built on top of SIMULA67, designed to provide building blocks that a modeller can tailor to his own personal needs, or which can be used as predefined units. Also, DEMOS provides builtin probability sampling functions, report generation, and tracing facilities. What impresses most about DEMOS is the ease with which a modeller can combine the simple blocks into quite complex systems, with relatively few simplifying assumptions.

The tracing facilities provided give very adequate diagnostic information for the user, as to the actions of the simulation. These can be turned on if desired; the default is for no trace information to be given.

DEMOS offers a relatively full range of distributions, split into three classes - *rdist* 's which return real values, *idist* 's which return integers, and *bdist* 's which return true or false. The sampling of these distributions is a simple matter of a function call SAMPLE to the relevant distribution, once the distribution has been declared and initialized.

Reporting and data collection are unobtrusive. The user of the package need only worry about his own specialized statistics - and tools are provided for the collection of such values. All other objects for which a report is generated, collect their own statistics, without any thought necessary from the user.

Also maintained by the system, but hidden from the user, are the manipulations of the queues which arise in all simulations. All queues are operated on a "first-in-first-out" basis, with the option of a prioritized queueing discipline available within this structure.

The basic building blocks provided by DEMOS fall into three categories. The first is that of statistical devices. These are designed for the user to collect his own customized statistics. They include a COUNT, for counting occurrences and the like; a TALLY for observing averages over occurrences; an ACCUMULATE for observing averages over time; and a HISTOGRAM for presenting statistics as a histogram.

The second class is that of QUEUEs. The following four blocks provide the facilities often used in simulation work. The first such structure is a RES, for simulating a limited resource for which queueing occurs. Secondly, a BIN is provided to model the dependence and synchronisation often observed between entities. Next, a WAITQ is provided for those structures which require a number of entities to be present before an event occurs. Finally, a CONDQ is provided for entities which must wait until some condition is satisfied before they are free to continue.

I will explain the functions of WAITQs and CONDQs further, as they are often used in NETMOS. A WAITQ is actually a pair of queues - one known as the slave, and the other as the master. One of the entities which uses the WAITQ is treated as being subordinate to the other. This entity enters the slave queue, and does not leave it until it has been COOPTed by an entity from the master queue. The subordinate entity is then bound to the master until the master releases it. This is usually after a number of actions have taken place, for which the two entities must cooperate. Upon release, the slave entity resumes whatever it should after this entity interaction.

A CONDQ, on the other hand, consists of just one queue. An entity enters the queue with a call to the procedure WAITUNTIL, and passes it a condition. This condition is evaluated immediately. If the result is true, the entity is allowed to continue. Otherwise, it is entered into the queue. Whenever the CONDQ is SIGNALed, either the condition of just the entity at the head of the queue is tested, or all conditons are tested. This depends on the CONDQ variable, ALL. Those entities whose condition evaluates to "true" are then released from the queue, and rescheduled to continue immediately.

The third class provided is for those items which can be scheduled, and take up time. This class is called an ENTITY. An ENTITY can be scheduled at another time, be interrupted, or placed in a queue for a resource as outlined previously.

It is on top of this package, (or more correctly, SIMULA context), that NETMOS is designed to fit. This move was taken because of the features DEMOS offers, and the ease with which extensions can be made to such a system, and was taken after looking at other systems available. Notably, RESQ and TETRASIM were examined to note the features and structures offered.

RESQ is a script-driven interpreter that was written for the simulation of networks in general. It provides a number of primitive tools, such as queues and statistics collection devices. However, it proved to be unsuitable for transcription into a SIMULA context because it is not just a simulator, but a system of analytical submodels.

TETRASIM proved to be more promising, being a SIMULA context like DEMOS. However, it was designed solely as a simulation tool for telephone networks, and did not appear to have much application to local area network simulation. A few ideas as to statistics of networks were obtained from TETRASIM, though.

<u>Section 1.2</u> :- <u>Changes made within DEMOS</u>

Because of the size of DEMOS source code(approximately 2500 lines), it is infeasible to attach a copy of my amended copy to this report. However, it is necessary to note the changes I have made within the standard DEMOS code.

Firstly, several new global variables were added for use with network simulation. These changes were included in the body of the DEMOS code to preserve the tidiness of the system, and to preserve the overall structure of the code.

A global CONDQ, WAITSTN, was introduced as a resting queue for idle STATIONS. Because, by default, only the condition at the head of the CONDQ is tested , the necessary flag ALL was set so that all STATIONs would be tested. The standard procedure for waking STATIONs then, is by setting the condition for the STATION required to true, and SIGNALing the CONDQ.

Also added was a WAITQ for use in my token ring model. The use of this queue will be discussed in chapter 3. A global variable, NET, which holds the reference to the current NETWORK was added as well. Because the details associated with a NETWORK are required frequently, it was desirable to have this as a global variable to the NETMOS context.

Lastly, a global MATRIX, MASTERDIST, was added to hold the distance matrix on which all timing calculations are made. The basis for this method is discussed in section 1.3.

In addition to these global variables, a further probability distribution was added which was missing for my purposes. The CONSTINT distribution is an IDIST, and therefore returns an integer value. CONSTINT is the complement of the DEMOS CONSTANT, which supplies a constant real value. CONSTINT was required because I wanted a USERs message length parameter to be an integer probability distribution, and a distribution returning a constant integer was not provided by DEMOS.

Furthermore, to aid in the reporting of the statistics collected, two more REPORTQs and report headings were added to those used in the procedure REPORT. One was added for all buffers, and the second was for reporting on the NETWORK class.

<u>Section 1.3</u> :- <u>Design Problems Encountered</u>

Several problems were encountered in the design of NETMOS. These are set out here, and their solutions discussed.

One of the first problems tackled was the handling of the links between stations. Two possibilities were thought of, those being the creation of a new class, or a distance matrix with no direct representation of the physical link. A benefit of having a LINK class was that a station need only pass a message down the link on which it did not arrive. The possible properties of these links were their length, their error rates, as well as other measures of link performance. Also, it was thought possible to use a link as a station-to-station hook to which messages were attached. The disadvantage was that in a real network, a message occupies several links at one time, which would have been very hard to simulate. Furthermore, in an Ethernet system, a collision would have been very hard to detect, because if you hang a second message over the top of a first, the reference to the first is lost.

The alternative method required only the distance and next station to be stored in an array. This method is more space efficient than that described above. All a station need do to pass a message to another station, is to hook the message onto the appropriate message slot in the receiving station, and alert the receiver as to the messages presence, after delaying for the appropriate amount of time.

From the start, it was my intention to build a model which was as close as possible to the real world situation. This involved following the ISO-OSI layered architecture for LANs wherever possible. It is for this reason that PROTOCOLs have the potential to reference other PROTOCOLs through their two links, UP and DOWN. This feature has not been implemented, but it would be very easy to do so.

Another problem encountered is one which is inherent in LANs. These networks usually operate at transmission speeds ranging between 1 Mbps and 10 Mbps. However, devices on the networks rarely produce messages at a rate much greater than one or two per second. Hence, with a network that can deal with bits in the order of microseconds, and devices which produces bits in the order of seconds, it is hard to come up with a convenient timescale for a model. In the end, I chose milliseconds as the basic time unit to work from.

On deciding to use DEMOS as my base system from which to build NETMOS, I envisaged being able to use BINs as the coordinating tools between STATIONs and USERs. However, that a BIN only has information about the number of objects at presently stored within it, and no specific details of the individual items. For my purposes, I required these details to be known, and so I had to build my own tool that acted the same as a BIN in most respects. In fact the only two differences between BUFFERs and BINs are the storing of specific instances of MESSAGEs in BUFFERS, and that entities do not queue up waiting for things to be dropped in BUFFERs. Instead, an interrupt is generated whenever something is put in a BUFFER to alert the receiver to the presence of a waiting MESSAGE. Until then, USERs and STATIONs can circulate doing their own thing, generating or receiving other MESSAGEs.

A further reason for originally using DEMOS as a base was its (apparent) ability to handle external compilation of program modules. Because of the slow compilation rate of the SIMULA compiler to be used, this seemed extremely desirable. It appeared from reading DEMOS code that the SIMULA statement EXTERNAL could be used to cascade EXTERNAL classes down from DEMOS. However, when this was tried, all that came back were error messages. On reading the documentation associated with the compiler, it was discovered that, although intending to allow such cascading of modules, only one level of EXTERNAL modules are allowed.

Section 1.4 :- NETMOS Model Definition

NETMOS programs are set out in a very simple fashion, as follows :-

```
begin
external class netmos;  (external module definition)
netmos begin        (your program is a NETMOS subclass)

(* definition of any subclasses the user desires *)

(* definition of the users required into the array USERS *)

nettrace;
net :- new tring ("Token ring",10,0.00000000001,
                  5,2,10,users);
hold(warmup time);
reset;
hold(simulation time);
end;
end;
```

The two commands to notice are NETTRACE and RESET. NETTRACE is part of the second level of tracing facilities built on top of DEMOS. NETTRACE will give tracing information pertinent to the simulation of networks, while TRACE will give all tracing - both NETTRACE information and the DEMOS tracing.

The other command, RESET, allows the user to reset the statistics after a warmup time as dictated by simulation theory. This helps the system to give statistically accurate information.

Also there are two procedures added to NETMOS that allow the user to generate his own trace and error messages. NETNOTE takes four parameters. The first is an index as to the type of the trace message. The second is a string which describes the action happening. Thirdly, a reference to a message is passed, if needed. Lastly, an integer field is provided. This tracing mechanism works in the same manner as that provided by DEMOS.

NETERR also works in the same way as that provided by DEMOS. It was hoped that a number of network-specific errors could be caught and flagged before generating DEMOS errors. In a number of cases this has been done. However, the resulting set of error messages is by no means comprehensive, so DEMOS errors do sometimes occur.

NETERR has only three parameters :- an index as for NETNOTE; a string to tell how the error arose; and a reference to the offending entity.

Chapter 2 : NETMOS Class Definitions

Section 2.1 :- MESSAGES

MESSAGEs are the items of interest within a network simulation because the statistics of interest concern the service which MESSAGEs receive in their travelling from source to destination. MESSAGEs have a very simple structure because the actual contents are (generally) unimportant. Instead, they are generalized so that only the vital parameters are stored.

The structure of a message is as follows :-

```
UNIT CLASS message (sender);
     ref(user) sender;

begin
    integer size, dest, source, status, attempts;
    real timein;
    ref(message) link;

end   *** message ***;
```

The parameter SENDER is used to initialize the size, destination, and source components of the message by sampling the distributions held by USERs (see next section), and the USER parameter, MY-ID. LINK is used to link messages together in BUFFERS, while STATUS and ATTEMPTS are used for protocol specifications, as detailed further on. TIMEIN is set on a PUTIN(m) into a BUFFER for sampling the average waiting time in the BUFFER.

Note that the MESSAGE definition is prefixed by the class UNIT. At present, UNIT is an empty class definition. It has been included so that items used in communications, such as packets and messages, which have similar structures, can be grouped together under a common prefix. However, MESSAGES are the only such structure implemented so far.

This leaves room for further extensions and refinements, so that a more realistic simulation of splitting a message into a number of component packets can be accomplished.

Section 2.2 :- MESSAGE GENERATORS - USERS

USERs are used to describe the message sending characteristics of various nodes on a network. This includes message length distributions and "think" times - the distribution of times between consecutive messages. The basic USER class provided can be extended by the simulator to simulate specific characteristics. The in-built CLASS USER has the following outline :-

```
entity class user(my-id, msge-leng,
                  destination, thinktime);

     integer my_id;
     ref(idist) msge_leng, destination;
     ref (rdist) thinktime;

begin

ref(buffer) buffin, buffout;
ref(station) s;

procedure  generate;
procedure  receive;

end  *** user ***;
```

The parameter MY-ID is the unique identifier of a user - the

USER's "address" on the network. It is used to initialize the SOURCE element of all the MESSAGEs sent by a particular USER. A specific USER can be modelled in the specification of the distribution parameters used by a USER as detailed on the next page.

MSGE_LENG - an integer distribution giving message lengths for the USER.

DESTINATION - a further integer distribution telling to whom messages are sent, (usually RANDINT or CONSTINT).

THINKTIME - a real distribution describing the thinktime of the USER.

Two predefined USERs are offered. The first models a USER who both sends and receives messages. This is accomplished as shown below :-

```
USER CLASS stduser;

begin
loop:
     hold(thinktime.sample);
     if interrupted = 0 then
       generate
     else
       receive;
repeat;
end    *** stduser ***;
```

Notice that this USER does not wait for acknowledgements of messages received correctly. This is because at present this reply facility is not implemented in the PROTOCOLs supplied. Message arrival is signified by an interrupt generated by the station S. Such an interrupt received by the USER signifies the arrival of a new message in the buffer BUFFIN.

Message generation is accomplished in a different fashion. For most of the time, a USER is idle. This is signified by the HOLD (THINKTIME.SAMPLE) call. Unless it is interrupted out of this "thinking" , the USER generates a message on completion of the HOLD. GENERATE is then entered, and a new message is produced and placed in BUFFOUT (see the previous section on MESSAGEs).

The second predefined USER provided is an idle USER (one who never sends but only receives) and is defined as shown below :-

```
USER class idle_user ;

begin
loop:
   PASSIVATE;  (same as "wait until message to receive")
   RECEIVE;
   repeat;
end;
```

This USER is useful for modelling logged off USERs, or USERs who never send MESSAGEs. I used this class in my validation experiments described in Chapter 4.

EXAMPLE

User 27 of a network sends messages of lengths ranging between 100 and 120 bytes in length to user 4 only with a negative exponential delay with a mean of 10 milliseconds.

```
users(27) :-
   new stduser (27,
                 new randint("msge. lgths",100,120),
                 new constint("destination",4),
                 new negexp("think time",0.1));
```

Section 2.3 :- **MESSAGE HOLDERS - BUFFERS**

Buffers are used to hold messages passing between USERS and STATIONS. They provide the synchronisation required for sending and receiving messages. They contain a linked list of messages waiting in them, with a pointer to the message at the top of the queue. Each buffer is associated with both a user and a station.

They offer the following functions :-

```
CLASS BUFFER ( u, s);
     ref(user) u;
     ref(station) s;

begin

procedure report;
procedure reset;
boolean procedure full;
procedure putin(m); ref(message) m;
message procedure takeout;

end *** buffer ***;
```

A buffer contains a (hidden) queue of messages awaiting either transmission or reception, depending on its direction. Each USER and STATION has two buffers associated with it, arranged in the manner shown in figure 2.1, overleaf. As can be seen, a USER inserts MESSAGEs into its BUFFOUT, and takes them out of its BUFFIN. These directions also apply for a station. By this protocol, message swapping is accomplished between stations and users.

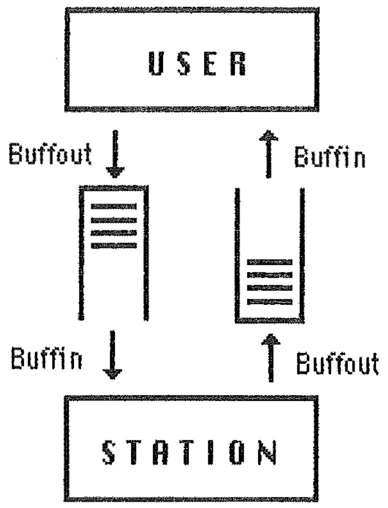


Figure 2.1 : Buffer interaction between stations and users.

The declaration and connection of buffers are handled by each station. This was done because they are really only hooks for a USER into the network. They provide the only means by which the USER is able to interface with the network, but he does not create the interface - it is inherent within the station. All the user must do to be able to start communicating, is connect to the BUFFER.

The statistics, as shown below, are reported for each buffer at the end of each run, and whenever REPORT is called :-

```
TITLE   / (RE)SET/ OBS/QMAX/QNOW/ Q AVERAGE/ZEROS/ AV. WAIT
*****************************************************************
```

These fields are :-

TITLE = user given title

RESET = time at which last RESET issued

OBS = number of MESSAGEs through buffer

QMAX = maximum number of MESSAGEs waiting

QNOW = number of MESSAGEs waiting now

Q AVERAGE = average queue length of MESSAGEs waiting

ZEROES = number of times there are no MESSAGEs waiting

AV. WAIT = average waiting time in the buffer

EXAMPLE

At present, BUFFERs are implemented without any queue length limits. This was done so that the user could observe the maximum and average queue lengths without the interference of limits. However, if a user desired limited queue lengths, then a subclass of BUFFERs can be declared as below:-

```
BUFFER class limitedbuff (limit);
  integer limit;
begin
procedure limitedPutin(m);
  ref(message) m;

begin
if avail >= limit then
   (* failed putin *)
else
  putin(m);
end;
end *** limitedbuff ***;
```

What happens on a failed PUTIN is up to the desires of the implementor, but usually the entity trying to PUTIN would be queued in a waiting state, and notified of possible available space after a TAKEOUT.

Section 2.4 :- MESSAGE PROCESSORS - STATIONS

STATIONs simulate the physical USER - NETWORK interface, the actual physical connection between the USER and the communications line. They implement different facilities depending on the set of PROTOCOLs being used on the link. There is no (foreseeable) way of separating out this dependency as each PROTOCOL set demands different capabilities from the STATION.

Class STATION has the following outline :-

```
ENTITY CLASS station (u);
    ref(user) u;

begin

    ref(buffer) buffin, buffout;
    ref(protocol) p;
    ref(message) min, mout;
    ref(matrix) d;
    real rate;
    boolean  transmitting, receiving,
                  rready, sent, token;
    integer nomsges;


procedure alert;
procedure initialize;

end   *** station ***;
```

The reference to a USER is used to initialize the links between the STATION and its associated USER via the two buffers BUFFIN and BUFFOUT, as described in the previous section, and shown in figure 2.1. The protocol P is used for sending and receiving messages. Examples of two protocols are given further on, in the discussion of the two models implemented. The two messages MIN and MOUT are used to handle incoming and outgoing messages, relative to the network. Therefore, MIN contains messages destined for the USER associated with the STATION, while MOUT handles those messages destined for other USERs. The boolean values are used by the various models to indicate the state of the station, while the matrix D holds the relative distances from this instance of a STATION to every other STATION on the network. This implementation will be discussed further on, in section 2.6.

The procedure ALERT issues an interrupt to the STATION, to notify it of the presence of an incoming MESSAGE. INITIALIZE creates the two buffers used, links them to the USER U, and initializes the other local variables.

The actual working of a station is quite simple, and can be generalized in the following manner :-

```
loop:
    waitstn.waituntil (able to send or able to receive)
    if able to receive then
        receive
    else
        transmit;
repeat;
```

WAITSTN is a global CONDQ (see discussion on DEMOS in section 1.) in which the stations wait until the condition is fulfilled. The actual condition depends on the protocol being used.

Receiving is accomplished by a call to P.RECEIVE. On successful return from this procedure, the sending station will have set MIN to reference the message being sent. If no errors have been detected, the message is PUTIN BUFFOUT.

Transmission is the opposite procedure, whereby MOUT is taken out of BUFFIN and passed to P.SEND(mout).

<u>EXAMPLE</u>

According to a simple protocol, a STATION can transmit whenever it has a message to send. All STATIONs have direct, point-to-point links between them, so there can be no collisions. Also, the links offer perfect communication, therefore there are no errors in transmission. Such a STATION can be modelled as follows:-

```
STATION CLASS simplestat;

begin
initialize;
loop :
   waitstn.waituntil (buffin.full or rready)
   if buffin.full then
      begin
      mout :- buffout.takeout;
      p.send(mout);
      (* collect statistics *);
      end
   else
      begin
      p.receive;
      buffin.putin(min);
      end;
repeat;
end    *** simplestat ***;
```

Futher examples of STATION implementation are discussed in Chapter 4.

Section 2.5 :- MESSAGE VERIFIERS - PROTOCOLS

Strictly speaking, in communications terms, protocols are the rules for transmission by which an error-free , point to point line is operated over less than perfect communication lines. These rules specify the format and contents of messages, when stations can transmit, and so on. If all stations obey the protocols, assuming no hardware faults, error-free communication is enjoyed by all.

Protocols can essentially be split into two facets: sending and receiving. It is this split which I have tried to show in the structure of my CLASS PROTOCOL.

```
CLASS protocol;

    virtual: procedure send, receive, resume;

begin
    ref(protocol) up, down;

    procedure send;
    procedure receive;
    procedure resume;

end   *** protocol ***;
```

By making the three procedures virtual, I ensure that all PROTOCOLs have these procedures as defaults, but they can be replaced if desired. SEND, as the name suggests, contains the rules for sending messages. The default is a procedure which produces an error message and program abortion. RECEIVE contains a PASSIVATE. RESUME is activated on the successful completion of a SEND to interrupt the PASSIVATEd receiver, and sets the variable MIN of the receiving STATION to the reference of the sent MESSAGE.

UP and DOWN are included for a layered model. As a STATION calls a PROTOCOL to send a MESSAGE, it is intended that this PROTOCOL would split the MESSAGE into smaller packets, and itself call a further PROTOCOL, referenced via DOWN. This scheme has not yet been implemented.

EXAMPLE

A simple protocol similar to Ethernet, can be described in the following way:- a station will send if it can not hear another message on the line. Once the message being sent has had time to travel to the far end of the network, and return, the station checks to see if a collision has occurred. If so, a backoff algorithm is invoked, otherwise transmission is completed. This simple protocol can be implemented as follows :-

```
PROTOCOL CLASS simplep(s);
    ref(station) s;

begin
procedure send (m);
    ref(message) m;

    begin
    hold(net.length * net.prop);
    stations(m.dest).alert;
    hold(net.length * net.prop);
    if collision then
        backoff
    else
        begin
        hold((m.size / net.speed)
              - (2 * net.length * net.prop);
        stations(m.source).p.resume(m);
        end;
    end   *** send ***;

procedure resume(m);
```

```
            begin
            s.min :- m;
            s.interrupt(0);
            end   *** resume ***;

        end   *** simplep ***;
```

Note that the RESUME procedure that is activated is the procedure associated with the destination STATION, and is accessed via the reference "STATIONS(M.DEST).P". The procedure RESUME is very simple for most examples that could be given. The only task given to it is to set MIN correctly, and alert the receiving station that a message has arrived.

RECEIVE is not specified because the default PASSIVATE is all that is required of the procedure in this example. As can be seen, the main work of a PROTOCOL is carried out in the SEND procedure. It is in this phase that most of the rules for behaviour are enacted.

Further examples of the implementation of PROTOCOLs are discussed in the following chapter.

<u>Section 2.6</u> :- <u>Miscellaneous</u>

The following classes have been included in NETMOS for house-keeping purposes, rather than for direct use by a user. However, it is necessary to describe them, for completeness of documentation.

Section 2.6.1 :- NETWORK

For the sake of simulation specification, and tidy statistics reporting, it was decided to group all STATION instances, and PROTOCOLs together under one super-class - that of the network. This also gives a greater degree of realism to the simulation, because to a connected USER- the NETWORK is merely a black box through which he can communicate with others. Hence the idea was that a user of NETMOS should need to specify only the necessary parameters of the NETWORK, without having to specify the internal workings, unless they wanted to.

The class NETWORK has been designed to reflect the true parameters of a network :-

```
TAB CLASS network (speed, errate, prop, length,
                             nostats, users);

    real speed, errate, prop, length;
    integer nostats;
    ref(user) array users;

    begin

    integer erred, messages;
    real tsend, through, sqthru;
    ref(station) array stations (1:nostats);
    ref(rdist) err;

    procedure report;
    procedure reset;

end   *** network ***;
```

The parameters given are as follows :-

speed = data rate of the communications line, expressed in Mbps. This is converted internally to bits per millisec (the basic time unit of the model).

errate = error rate on the line, usually in the range 10^{-8} to 10^{-11}

prop = propagation delay per kilometre, given in microsecs. This, also, is converted internally to millisecs.

length = length of the network.

nostats = the number of stations on the network.

users = the array of USERs to be connected to the network.

The network will create a STATION for each USER, which then makes the necessary BUFFER connections between itself and the USER. The array STATIONS stores the references to these individual STATION instances. Also, an ERRORS sampling distribution is created. This is associated with the NETWORK because it is inherent in the network. Also, if created at this level, reporting space is saved if only one distribution is created globally, rather than one for each USER. The trade off is against statistical independence for the individual USERs error rates.

The REPORT procedure for a NETWORK calculates and reports the following statistics :-

```
TITLE/(RE)SET/OBS/ERRS/SPEED/#STNS/THROUGHPUT/EST.ST.DEV/WALK TIME/NET LOAD
**************************************************************************
```

These fields represent :-

TITLE, (RE)SET = see page 16

OBS = number of successfully sent messages.

ERRS = number of unsuccessful "SENDs" tried.

SPEED = network speed in Mbps.

*STNS = number of stations on the network.

THROUGHPUT = actual data rate experienced by MESSAGEs
i.e. (data sent)/(total time in a sending mode).

EST.ST.DEV = estimated standard deviation of the THROUGHPUT.

WALK TIME = average time between when messages come to the head of a buffer, and when they reach their destination.

NET LOAD = average network loading.

Unfortunately, the collection of such statistics is hard for the NETWORK to do itself, because it is really only a shell for the holding of the values, so MESSAGEs do not actually pass through it. So, collection must be done on behalf of the NETWORK by each STATION. This gets away from the DEMOS ideal of statistics collection being completely hidden from the user, but is unavoidable if the user is going to be designing and implementing his own PROTOCOLs and STATIONs.

EXAMPLE

The specification for an Ethernet NETWORK is laid out below :-

```
NETWORK CLASS enet;

begin

   integer i;
   ref(idist) array backtime(1:10);

   for i := 1 step 1 until nostats do
```

```
          begin
          stations(i) :- new estat("station",users(i));
          stations(i).schedule(0.0);
          end;
        for i := 1 step 1 until 10 do
          backtime(i) :- new randint("backoff",0,(2**i));
        join(netq);

      end *** enet ***;
```

The array BACKTIME is used for the Ethernet backoff algorithm as explained in section 3.2. The call to the function JOIN, adds this instance of an ENET to the reporting queue NETQ.

<u>Section 2.6.2</u> :- MATRIX

As discussed in section 1.3, this was one of the first design problems encountered. The purpose of a MATRIX is to store the distance between consecutive STATIONs, and the identity of the connected station. A MATRIX, therefore, is merely an array of DISTRECORDs, with the following declarations :-

```
CLASS distrecord;

   begin
   integer station;
   real distance;
   end    *** distrecord ***;

CLASS matrix(nostats);
   integer nostats;

   begin
      ref(distrecord) array dist(1: nostats);
   end    *** matrix ***;
```

The idea is that the connection between STATION i and STATION i + 1 would be stored as is shown in figure 2.2, overleaf.

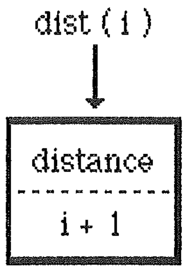


Figure 2.2 :- DISTRECORD format

To aid in the creation and manipulation of this structure, two functions are provided. LOADDISTANCE prompts the user for the distances between STATIONS, and loads these distances into a MATRIX, which is returned. Also provided is a function SORTED, which takes as parameters a USER id and a MATRIX. What SORTED returns is a matrix that is sorted by ascending relative distances from the USER. The distance component of each DISTRECORD contains the relative distance from the previous station to the present. For example, for a network with the MATRIX on the left, SORTED returns the MATRIX on the right for USER 2:-

dist(i)	distance	station
1	0.25	2
2	0.35	3
3	0.00	0

dist(i)	distance	station
1	0.25	1
2	0.10	3
3	0.00	0

Chapter 3 : Model Discussions

Section 3.1 :- Token Ring

A token ring is implemented on a connected loop to which the communicating devices are connected. Each device is connected to a station which forms a part of the ring. The basic protocol behind a token ring network is the simple rule that a station may not start transmitting until it has been given permission by having control of the single token which circulates around the network. Once the token has been received, a station can start transmitting as many bits as it has waiting, up to a set limit. The packet, thus sent, circulates around the network behind the token.

When the receiving station ascertains that the message is destined for it, it takes a copy of the message off the network as the message passes by, and checks for transmission errors. If an error is detected, the final bytes of the packet are set to show the type of error. As the message arrives back at the sending staion, it removes the message from the network, and releases a free token. If the message was correctly received, it is discarded; otherwise the station holds the message until it gets a chance to try sending it again.

This structure was successfully translated for simulation in the following model. A new class was created to play the role of the token. This class is prefixed by the class ENTITY, which allows the TOKEN to be scheduled in simulation time. It is set off on its trek around the STATIONs, starting at STATION 1, and using the distance MATRIX to simulate the time delays between STATIONs. To alert a station as to the presence of the token, the STATION variable TOKEN is set to true and an interrupt is issued.

The STATION leaves the CONDQ WAIT-STN and, if it has something to send, moves into the send phase of the TSTAT body. Rather than spending time stepping from STATION to STATION on the way to the receiver, I tried to save on running time by stepping directly to the destination STATION. This bypasses part of the token ring standard as published, because the standard allows for STATIONs to set a priority count within the token on its way past. This facility can still be implemented, but it would increase the time required to simulate the network by quite alot.

On being ALERTed, the receiving STATION passes into the receive code of the PROTOCOL. This PASSIVATEs the STATION. On being reactivated by the sending STATION, the receiver sets the error flag in the MESSAGE. If it has been correctly received, the MESSAGE is placed in the TSTAT's BUFFIN. If not, it is rejected.

After waiting for the MESSAGE end to return, the sending STATION releases the TOKEN, and forgets the MESSAGE sent if no errors were noticed. If they were detected, the MESSAGE is remembered, and resent at the next opportunity.

To save more time, I PASSIVATE the token whenever there are no messages waiting for transmission in the NETWORK. It is reactivated whenever a message is put into any buffer. Rather than start the token back at the waiting STATION, I start its stepping from where it last stopped. This does not accurately reflect the time delay involved in waiting for the token, but because the delay is minimal, it makes little difference.

My model also differs slightly from the standard in that the standard specifies that the token is delayed by one bit time at each station it passes. My model does not do this, but it could be easily modified. However, to do this would require the token to pass through all the stations when delivering a MESSAGE, which, as mentioned before, increases the run time of the simulation.

The relevant code sections for this model are on the pages of Appendix B as shown below :-

Class	**Page**
TRING	2
TOKEN	4
TOKENRING (PROTOCOL)	8
TSTAT (STATION)	16

Section 3.2 :- Ethernet

The Ethernet standard was developed by Xerox back in 1975, in conjunction with Digital and Intel. Ethernet is a CSMA/CD protocol. This means that it is a Carrier Sense, (the STATION is continuously listening for traffic on the transmission line), Multiple Access, (all STATIONs can access an empty line at any time), with Collision Detection, (on hearing a collision, the STATION backs down according to a backoff algorithm).

According to this definition, there are less events in a time period with an Ethernet than with a Token Ring, because there is no token passing from station to station. Instead, if the STATION has a message to send, and can't hear anything else on the network, it starts sending.

An important parameter in an Ethernet network is the "walk time" of the network, which is the time required for a message to go from one end of the network to the other, and then to return. This is important, because the longest time that the sending STATION must wait before it knows whether not its message has collided with another, is equal to this "walk time". Also important is the time required by the network to transmit 512 bits. This time, known as the "slot time", is used for the basic backoff time. It is also defined as being greater then or equal to the "walk time". Therefore, all messages must be at least 512 bits long.

The backoff algorithm used is that a colliding STATION backs off for a length of time dependent on the number of retries the message has had to make already. A uniformly distributed random integer is generated in the range 0 -> 2^k, where k is the number of collisions already incurred, or 10, whichever is smaller. The delay then experienced by the STATION is this random number multiplied by the slot time of the network, as described previously. However, only 16 attempts are made. If all these attempts fail, the STATION is alerted to a network error.

This PROTOCOL is implemented in the creation of a new class a NOTIFIER. A NOTIFIER is designed to simulate three events of the same type. The first is the arrival at a STATION of a new message. In this case, the STATION is issued an interrupt of type 1, and is ALERTed, as described in section 2.4. The second event is the successful completion of a message transmission. This is simulated by RESUMEing the STATION with an interrupt of type 0. If a STATION receives one of these interrupts, it puts its copy of the MESSAGE in its BUFFOUT, if it is the destination.

The third class is to signify a collision. This need not be flagged because each STATION will already have detected the collision, and will be waiting for an empty line. However, whenever a NOTIFIER is spawned, it will either increment, or decrement, the STATION variable NOMSGES. This variable shows the number of messages the STATION is listening to, and is my equivalent to the Ethernet Carrier Sense mechanism. Hence the line is quiet if NOMSGES equals 0, busy if it equals 1, and in a collision state if NOMSGES is greater than 1.

If a second message arrives at a STATION while it is receiving a first, the interrupt issued by ALERT does not put the STATION into a further receiving phase, but, because the interrupt is of the wrong type, puts it into the waiting queue for STATIONs. When the transmitting STATION detects the collision, it sends off a NOTIFIER that decreases the number of messages a station can hear. If a NOTIFIER of this sort detects a STATION that can hear nothing, (i.e. NOMSGES = 0), it signals the CONDQ WAITSTN to wake the STATION. It is then free to send a message if it has one to send.

The code segments relevant to the Ethernet model are given on the following pages of Appendix B :-

Class	**Page**
ENET	3
NOTIFIER	5
BACKER	5
ETHERNET (PROTOCOL)	9
ESTAT (STATION)	17

Chapter 4 : Model Validation

This brief chapter is a discussion of the results graphed in Appendix A. The results of my experiments are compared with those discussed in the paper by S. T. Chanson et. al, ([1]), where they developed analytical models for three common local area networks. These models are graphed at the back of their paper. I have carried out experiments to compare my results to those they analytically predicted.

Their graphs reflect the relationship between packet walk time and network loading. The walk time of a packet is taken as being the time between when it is ready for transmission and the time at which it is put into the destination buffer. This time consists of two main components - the transmission time for the message, and the time spent waiting in the buffer. For my model it seemed that a message was not ready for transmission until it reached the head of a buffer queue. Hence, the statistics reported are the average time between the messages reaching the head of the buffer queue, and being put in its destination buffer.

The network load is the proportion of time the network is sending data. This is also reported by my model.

The Token ring model, as shown in the top graph, seems to follow fairly closely the slope of the graph predicted by Chanson. It is quite smooth, with a gentle slope until network loading reaches approximately 80%. At this point, it begins to climb.

However, the difference that arises in the actual values recorded, can be attributed to one factor. The main difference between my simulation model, and their analytical model, is that they assume that on receiving a token, a station can send all the packets it has backed up at that point of time. However, my model allows a station to only transmit only one packet each time. This means that the average walk time for packets tends to be higher, because they must wait longer in the buffer before being sent.

This also explains the difference in the curves at the top of the range, because if their network becomes 90% loaded, each station will send more and more packets each time it is able to. However, in my model, there is a limit to the amount of time a station must wait before it can send a packet. This limit is given by the expression :-

$$\text{max. walk} = (\text{N}^{\underline{\text{o}}}\ \text{sending stations} + 1) * (\text{propagation delay}) + (\text{N}^{\underline{\text{o}}}\ \text{sending stations}) * (\text{pkt trans time})$$

For the experiments conducted, the number of sending stations was set to four, the network length to 1km, the propagation delay to 5μsecs/km, and the packet transmission time to 0.2667 millisecs. Hence, the maximum wait for any single packet would be 0.9317 msecs.

The Ethernet curves, shown in the lower graph, reflect a similar trend to the Token ring model. That is, the shapes of the two graphs are similar, but the values are different. I suspect the main difference is in the backoff slot time employed. My model uses a slot time of 512 bit times as its basic backoff time unit. This is based on the Ethernet standard as published by the ISO, which assumes a minimum packet size of 512 bits. However, Chanson et. al. use a minimum packet size of only 256 bits. This would result in them having more collisions, and having to attempt sending an individual packet more than just once or twice.

A further difference which would account for some of the difference is that the paper graphs a curve for 128 active (i.e. sending) stations. However, because of the limitations on stack size, I was unable to simulate more than about 70 stations without overflowing the VAX heap. This factor would also result in my messages encountering less collisions than those in the Chanson model, thus further reducing the walk time for my packets, especially at the top end of the loading scale.

I have left a question mark at the end of my curve for the Ethernet model because I was unable to record a network loading of more than about 0.848. Hence, I was unable to make observations at this end of the scale.

Chapter 5 : Conclusion

This project has produced a set of building blocks useful in modelling a local area network. This tool seems most suited to the testing of LANs before they are installed physically. Classes provided include the MESSAGE, STATION, USER, BUFFER, and NETWORK. This set is by no means complete, as the two models created show. In fact, it is inconceivable that a context could be developed which could satisfy all possible users. Therefore, it is convenient that SIMULA allows for the redefinition and extension of modules. Some of the extensions which could be easily made are discussed finally, with the view of someone, someday, being inclined to make them.

Firstly, the class UNIT could very easily be extended to allow different classes of message, such as the packet. This development would probably go hand in hand with the development of a layered PROTOCOL scheme for a model. As already explained, the implementation features necessary for meshing in such a tool are already in place in this SIMULA context as it stands. At present, only the layer two and layer one (physical layer) are provided. But there is potential for including the HDLC-like protocol as defined for LANs by the ISO.

At present, only two predefined NETWORKs are included - the Ethernet, and the Token ring. This was done because it was felt essential to include models of the two most common protocols in use today. However, it is hoped that it would be quite easy to include further models of other protocols as these are implemented.

A further useful extension could be the inclusion of specific predefined USER classes. Such subclasses could include objects such as terminals, disk units, shared CPUs, etc. This would make modelling in NETMOS easier for the user, so that he need not worry about modelling such items, but concentrate entirely on the simulation of the network in which he is really interested.

A further potential application of NETMOS is in the field of wide area networks. Although these are fundamentally different in design to local area networks, many concepts are shared. Hence, it is conceivable that wide area network applications could be modelled using NETMOS.

BIBLIOGRAPHY

[1] S. T. Chanson, A. Kumar, A. V. Nadkarni, "Performance of some Local Area Network Technologies", Dept. of Computer Science, University of British Columbia, Vancouver, B. C., Canada.

[2] W. R. Franta, "The Process View of Simulation", North-Holland Publishing Co., New York, 1977.

[3] C. H. Sauer, E. A. MacNair "Simulation of Computer Communication Systems", Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

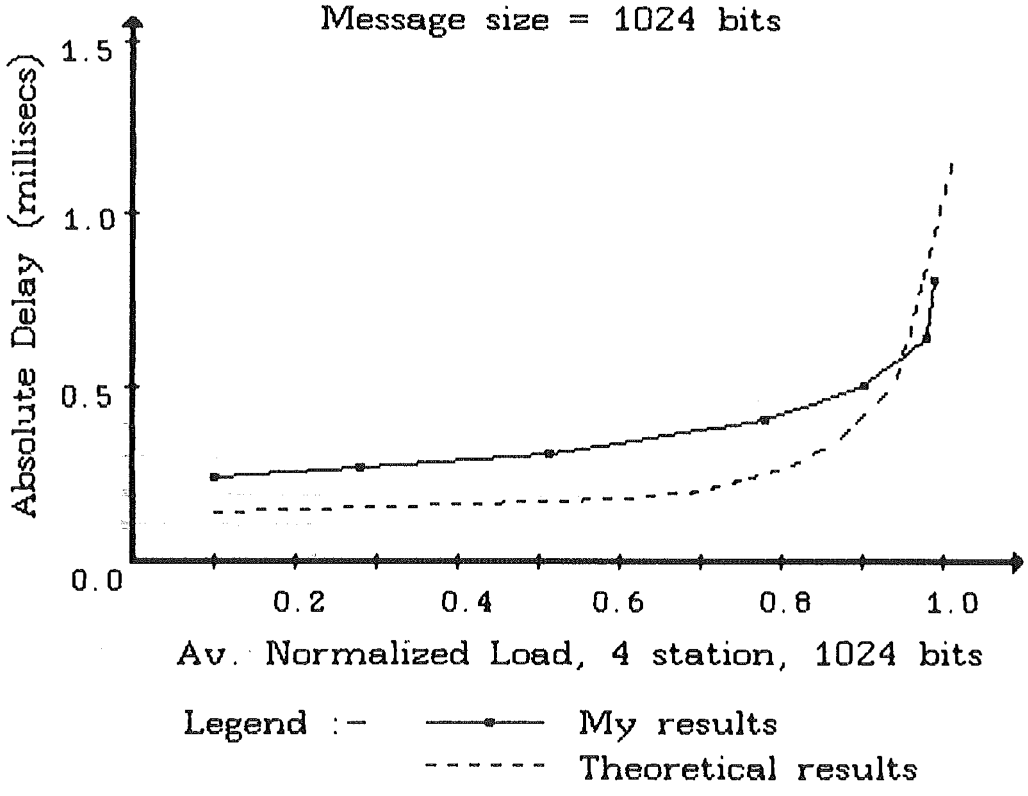
[4] T. Røgeberg "Tetrasim : A Program System for the Simulation of Telephone Networks", in S. Schoemaker (ed.) "Computer Networks and Simulation II", North-Holland Publishing Co., Amsterdam, 1982.

[5] W. Stallings, "Local Networks", ACM Computer Surveys, Vol 16, № 1, March 1984.

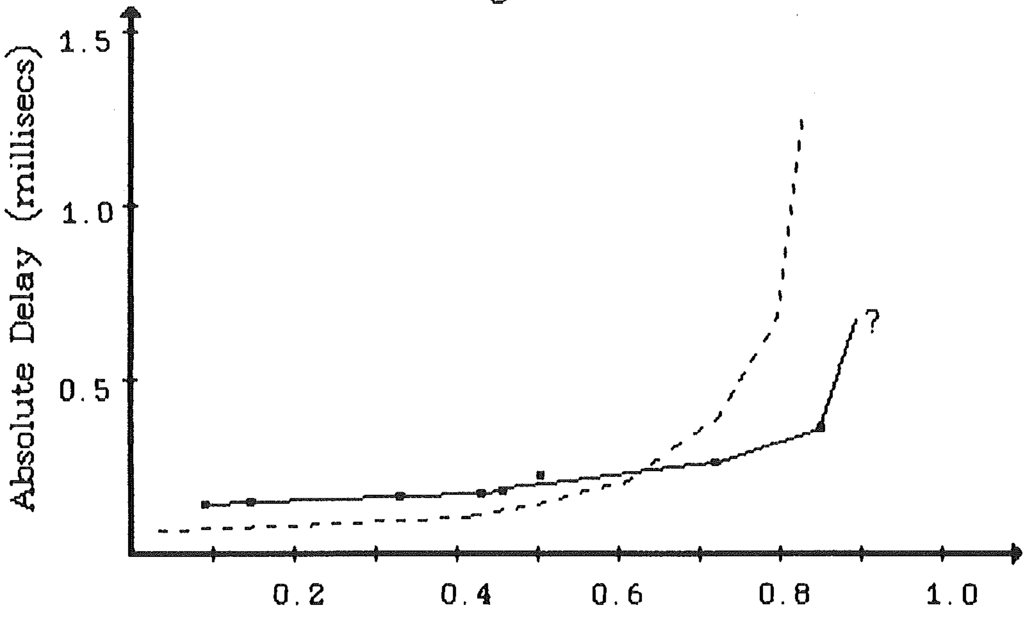
Appendix A

Validation Graphs

Token Ring Model
N = 4 active stations
Message size = 1024 bits



Ethernet Model
N = 64 active stations
Message size = 256 bits



Appendix B

NETMOS Source Code

```
tab class network(speed,errate,prop,length,
                  nostats,users);

    real speed, errate, prop, length;
    integer nostats;
    ref(user) array users;

    begin

COMMENT ---------------- NETWORK ----------------;
        COMMENT
*
* Networks are the central statistics to any
* LAN model.  This class serves as a central
* concentration of stations to which users
* can attach.
*
*  Variables :
*    (from TAB)
*    .title       user supplied descriptive text
*    .obs         no. of entries since resetat
*    .resetat     time when initiated, or last reset
*    .next        ref to next tab in reportq
*
*    (network specific)
*    speed        transmission rate
*                 in bits per millisec
*    prop         propagation delay
*                 in millisecs per km
*    length       network length
*    nostats      no. of stations on the network
*    users        array of network users
*    erred        no. of errors since resetat
*    messages     no. of messages currently in system
*    tsend        time in a sending mode since resetat
*    through      sum of throughputs experienced
*                 since last resetat
*    sqthru       sum of squared throughputs
*    walk         the accumulated walk times of messages
*                 i.e. the time betwwen the message coming
*                 to the head of the buffer, and reaching
*                 its destination
*    stations     array of ref's to stations
*    err          rdist for error detection
*
* Procedures
*    report       prints on one line
*                 title/resetat/obs/erred/speed/
*                 nostats/through/sqthru
*    reset        sets erred, obs, through, sqthru
*                 and tsend to 0, and resetat to time
*;

    integer erred, messages;
    real tsend, through, sqthru, walk;
    ref(station) array stations(1:nostats);
    ref(rdist) err;

    procedure report;

        begin

        real span, mean, t;

        t := time;
        span := t - resetat;
        writetrn;
        outf.image.sub(24,7).putint(obs);
        outf.outint(erred, 6);
        outf.setpos(38);
        outf.outfix(speed/1000, 2, 5);
        outf.outint(nostats, 6);
        outf.outtext(" ");
        if (obs > 0) then
           printreal(through/obs)
        else
           outf.outtext(minuses.sub(1,10));
        outf.outtext(" ");
        if (obs > 1) then
           printreal
              (sqrt(abs(obs*sqthru - through**2)/(obs*(obs-1))))
        else
           outf.outtext(minuses.sub(1,10));
        outf.outtext(" ");
        if (obs > 0) then
           printreal(walk/obs)
        else
           outf.outtext(minuses.sub(1,10));
        outf.outtext(" ");
        printreal(tsend/time);
        outf.outimage;

        end *** report ***;

    procedure reset;

        begin

        erred := obs := 0;
        resetat := time;
        walk := through := tsend := 0.0;

        end *** reset ***;

    err :- new uniform("Errors",0.0,1.0);
    speed := speed * 1000;
    prop := prop/1000;

    end   *** network ***;

network class tring;
```

```
    begin

    integer i;
    ref(condq) waittoken;
    ref(token) t;

    tokenq :- new waitq("idle token");
    waittoken :- new condq("loop token");
    for i := 1 step 1 until nostats do
        begin
        stations(i) :- new tstat("station",users(i));
        stations(i).schedule(0.0);
        end;
    t :- new token("token");
    t.schedule(0.0);
    join(netq);

    end   *** tring ***;

network class enet;

    begin

    integer i;
    ref(idist) array backtime(1:10);

    for i := 1 step 1 until nostats do
        begin
        stations(i) :- new estat("station",users(i));
        stations(i).schedule(0.0);
        end;
    for i := 1 step 1 until 10 do
        backtime(i) :- new randint("backoff",0,(2**i));
    join(netq);

    end   *** enet ***;

entity class token;

    begin

COMMENT --------------------- TOKEN -------------------------;
          COMMENT
*
* Token is used for a token ring simulation, and
* plays the role of waking stations when they may
* send.  My token passivates if no messages are
* waiting for service at present.
*
* Variables :
*    i            current station token is at
*    t            time to get to next station
*    idle         flag to show state of the token
*;

    integer i;
    real t;
    boolean idle;

    i := 1;
loop :
    if net.messages = 0 then
       begin
       if zyqtrace > 0 then
          netnote(9,"LOOPS IDLY",none,0);
       idle := true;
       net qua tring.waittoken.waituntil(net.messages > 0);
       end;
    idle := false;
    net.stations(i).token := true;
    waitstn.signal;
    tokenq.wait;
    t := masterdist.dist(i).distance * net.prop;
    i := (if i = net.nostats then 1 else i + 1);
    hold(t);
    repeat;

    end   *** token ***;

entity class notifier(d,jamming,m);

    ref(matrix) d; boolean jamming;
    ref(message) m;

    begin

COMMENT ------------------------ NOTIFIER ---------------------------;
        COMMENT
*
* Notifiers are used in the Ethernet model to simulate
* the arrival at a station of either the beginning
* or the end of a message.  3 types of notifiers are
* issued, depending on the parameters passed.
*            1 - start of a message
*                   jamming is false, and m == none
*            2 - end of successful transmission
*                   jamming is false, and m =/= none
*            3 - collision detected
*                   jamming is true, and m == none
*
*  Variables :
*   d           distance matrix for the sending station
*   s           ref. to the next station, as given by d
*   t           time required to get to the next station
*;

    ref(station) s;
    integer i;
    real t;

    for i := 1 step 1 until (net.nostats - 1) do
       begin
```

```
        s :- net.stations(d.dist(i).station);
        t := d.dist(i).distance * net.prop;
        if t > 0.0 then
           hold(t);
        if jamming then
           begin
           s.nomsges := s.nomsges - 1;
           if s.nomsges = 0 then
              waitstn.signal;
           end
        else
           begin
           if m == none then
              begin
              s.alert;
              s.nomsges := s.nomsges + 1;
              end
           else
              begin
              s.nomsges := s.nomsges - 1;
              s.p.resume(m);
              end;
           end;
        end;

    end   *** notifier ***;

entity class backer(s,t);

    ref(station) s;
    real t;

    begin

COMMENT ---------------- BACKER -----------------;
        COMMENT
*
* This class is also used by the Ethernet model.  It
* simulates the hold that is put on a station to stop
* it from sending while it supposed to be backing off.
* Once t millisecs have passed, backing is set to
* false, and the station can start sending again.
*;

    hold(t);
    s.backing := false;

    end   *** backer ***;

class distrecord;

    begin

    integer station;
    real distance;
    end   *** distrecord ***;

class matrix(nostats);

    integer nostats;

    begin

    ref(distrecord) array dist(1:nostats);

    end   *** matrix ***;

class unit;

    begin

    end *** unit ***;

unit class message (sender);

    ref (user) sender;

    begin

COMMENT ---------------- MESSAGE ------------------;
        COMMENT
*
* This is the class around which a network model
* necessarily revolves.
*
*  Variables :
*   size         size of the message in bits
*   dest         destination of the message
*   source       source of the message
*   status       status of the message
*   attempts     the number of times this message
*                has previously been sent
*   timein       time the message has entered a buffer
*   link         link to the next message in a buffer
*;

    integer size, dest, source, status, attempts;
    real timein;
    ref (message) link;

    inspect sender do
       begin
       size := msge_leng.sample;
       dest := destination.sample;
       if dest >= my_id then dest := dest + 1;
       source := my_id;
       status := attempts := 0;
       end;
    link :- none;

    end *** message ***;
```

```
class protocol;

   virtual : procedure send, receive, resume;

   begin

COMMENT --------------- PROTOCOL --------------------;
        COMMENT
*
* A protocol is a set of rules according to which
* a message is passed from station to station.
*
*  Variables :
*   up, down     for the layering of protocols
*                unused at present
*
*  Procedures :
*   send         pass message from source to destination
*   receive      passivate until a message arrives
*   resume       used by Ethernet to interrupt a station
*                on successful transmission of a message
*;

   ref(protocol) up, down;

   procedure send(m); ref(unit) m;

      begin
      neterr(5,current,"P.SEND(M); REF(MESSAGE) M;");
      end;

   procedure receive;

      begin

      if zyqtrace > 0 then
         netnote(10, "MESSAGE INCOMING", none, 0);
      passivate;
      if zyqtrace > 0 then
         netnote(11, "MESSAGE END DETECTED", none, 0);

      end;

   procedure resume;

      begin
      neterr(6,current,"P.RESUME;");
      end;

   up :- down :- none;

   end *** protocol ***;

protocol class tokenring(s);
```

```
    ref(station) s;

    begin

    ref(rdist) err;

    procedure send(m);

        ref(message) m;

        begin

        real t;
        integer i,j;

        t := 0.0;
        if zyqtrace > 0 then
           netnote(2,"SENDING TO ",none,m.dest);
        i := m.source;
        for j := 1 while i <> m.dest do
           begin
           t := t + masterdist.dist(i).distance * net.prop;
           i := (if i = net.nostats then 1 else i + 1);
           end;
        hold(t);
        net.stations(m.dest).alert;
        net.stations(m.dest).min :- m;
        hold((m.size + 160)/net.speed);
        net.stations(m.dest).schedule(0.0);
        hold(net.length*net.prop - t);

        end   *** send ***;

    procedure receive;

        begin

        integer error;

        if zyqtrace > 0 then
           netnote(10, "MESSAGE INCOMING",none,0);
        error := (if s.buffout.full then 2 else 3);
        passivate;
        s.min.status :=
            (if net.err.sample > ((1-net.errate)**(s.min.size*8))
                        then 1
                        else error);

        end   *** receive ***;

    end   *** tokenring ***;

protocol class ethernet(s);

    ref(station) s;
    begin

    boolean sent;
    integer i;

    boolean procedure collision;

        collision := s.interrupted <> 0;

    procedure resume(mess);

        ref(message) mess;

        begin

        s.interrupt(0);
        s.min :- mess;

        end   *** resume ***;

    procedure backoff(attempts);

        integer attempts;

        begin

        integer k, slots;

        k := (if attempts > 10 then 10 else attempts);
        slots := net qua enet.backtime(k).sample;
        if zyqtrace >  0 then
           begin
           netnote(1,"BACKS OFF",none,slots);
           end;
        s.interrupted := 0;
        s.receiving := false;
        s.rready := false;
        s.backing := true;
        new backer("Back off",s, slots * 512/net.speed);

        end   *** backoff ***;

    procedure send(m);

        ref(message) m;

        begin

        real transtime;

        if zyqtrace > 0 then
           netnote(2,"SENDING TO ",none,m.dest);
        transtime := (m.size + 144) / net.speed;
        s.sent := false;
        new notifier("notifier",s.d,false,none).schedule(0.0);
        hold(transtime);
        if collision then
           begin
           hold(48/net.speed);
           new notifier("jammer",s.d,true,none).schedule(0.0);
           m.attempts := m.attempts + 1;
           backoff(m.attempts);
           end
        else
           begin
           new notifier("ender",s.d,false,m).schedule(0.0);
           s.sent := true;
           end;

        end   *** send ***;

    end *** ethernet ***;

entity class user(my_id, msge_leng, destination, thinktime);

    integer my_id;
    ref(idist) msge_leng, destination;
    ref(rdist) thinktime;

    virtual : procedure receive, generate;

    begin

COMMENT -------------------- USER --------------------;
        COMMENT
* Users are the generators and receivers of messages
* in a network.
*
*  Variables :
*   my_id        the unique identifier of the user
*   msge_leng    message length probability idist
*   destination destination probability idist
*   thinktime    inter-message sending time probability
*                distribution
*   m            ref to the current message being sent
*   buffin,
*   buffout      ref's to buffers for messages
*   s            ref to station this user is linked to
*
*  Procedures :
*   generate     produce a new message and put it in
*                buffout
*   receive      take a message out of buffin
*                inbuilt is a pause for 0.05 millisecs
*;

    ref(message) m;
    ref(buffer) buffin, buffout;
    ref(station) s;

    procedure receive;
```

```
   begin

   while interrupted <> 0 do
      begin
      interrupted := 0;
      hold(0.05);
      m :- buffin.takeout;
      if zyqtrace > 0 then
         netnote(8,"MESSAGE RECEIVED FROM USER ",none,m.source);
      end;

   end   *** receive ***;

   procedure generate;

      begin
      m :- new message(this user);
      if zyqtrace > 0 then
         netnote (3,"GENERATES MESSAGE FOR ",m,0);
      buffout.putin(m);
      end   *** generate ***;

   end   *** user ***;

user class stduser;

   begin

loop :
      hold(thinktime.sample);
      if interrupted = 0 then
         generate
      else
         receive;
   repeat;

   end   *** stduser ***;

queue class buffer (u,s);

   ref(user) u;
   ref(station) s;

   begin

COMMENT -------------------- BUFFER --------------------;
        COMMENT
*
* Class buffer provides the structure necessary for
* queuing of messages between users and stations
*
*  Variables :
*   head, tail,
*   buff        ref's to messages in the queue
*               waiting for service
*   avail       number waiting in the buffer
*    zeroes      the number of times the buffer is empty
*
*   Procedures :
*    report      prints on one line
*                /title/resetat/obs/maxlength/avail/
*                av. q.length/av. waiting time
*    reset       resets obs,zeroes,qint and cum to 0
*                maxlength to avail and lastqtime
*                and resetat to time
*    putin(m)    allows an entity to place a message in
*                buffer.  Error is generated if entity
*                is not a user or a station.
*    takeout(m)  allows an entity to remove a message
*                from a buffer.  Error is produced if
*                entity is not a user or station.
*    full        returns true if avail > 0
*;

   ref(message) head, tail, buff;
   integer avail, zeroes;
   real t;

   procedure report;

      begin
      real span;
      t := time;
      span := t - resetat;
      writetrn;
      outf.image.sub(24,7).putint(obs);
      outf.outint(maxlength, 6);
      outf.outint(avail, 6);
      outf.setpos(44);
      if span < epsilon then
         outf.outtext(minuses.sub(1,10))
      else
         printreal((qint + (t - lastqtime)*avail)/span);
      outf.outint(zeroes, 6);
      outf.outtext(" ");
      if obs > 0 then
         printreal(cum/obs)
      else
         outf.outtext(minuses.sub(1,10));
      outf.outimage;

      end   *** report ***;

   procedure reset;

      begin

      obs := zeroes := 0;
      maxlength := avail;
      lastqtime := resetat := time;
      qint := cum := 0.0;
```

```
        end    *** reset ***;

    boolean procedure full;

        full := (avail > 0);

    procedure putin(m);

        ref(message) m;

        begin

        if m == none then neterr(1,current,
                   "B.PUTIN(M); REF(BUFFER) B; REF(MESSAGE) M;");
        m.timein := time;
        if avail > 0 then
            begin
            if tail =/= none then
                begin
                tail.link :- m;
                tail :- m;
                end
            else
                head :- tail :- m;
            end
        else
            buff :- m;
        if current in user then
            begin
            net.messages := net.messages + 1;
            if (net is tring) then
                net qua tring.waittoken.signal;
            s.transmitting := true;
            waitstn.signal;
            end
        else
            if current in station then
                u.interrupt(1)
            else
                neterr(4,current,
                  "B.PUTIN(M); REF(BUFFER) B; REF(MESSAGE) M;");
        if zyqtrace > 0 then
            netnote(4,"PUTS MESSAGE IN BUFFER",m,0);
        t := time;
        qint := qint + (t - lastqtime) * avail;
        lastqtime := t;
        avail := avail + 1;
        if avail > maxlength then maxlength := avail;

        end    *** putin ***;

    ref(message) procedure takeout;

        begin

        ref(message) mtemp;
        if buff == none then neterr(2,current,
                   "B.TAKEOUT(M); REF(BUFFER) B; REF(MESSAGE) M;");
        mtemp :- buff;
        buff :- head;
        if buff =/= none then
           begin
           cum := cum + (time - buff.timein);
           buff.timein := time;
           end;
        if head == tail then
           head :- tail :- none
        else
           head :- head.link;
        if zyqtrace > 0 then
           netnote(5,"TAKES MESSAGE OUT OF BUFFER",mtemp,0);
        t := time;
        qint := qint + (t - lastqtime) * avail;
        obs := obs + 1;
        lastqtime := t;
        t := lastqtime - mtemp.timein;
        cum := cum + t;
        avail := avail - 1;
        if avail = 0 then zeroes := zeroes + 1;
        takeout :- mtemp;

        end   *** takeout ***;

    head :- tail :- buff :- none;
    avail := 0;
    lastqtime := 0.0;
    join(buffq);

    end   *** buffer ***;

entity class station (u);

    ref(user) u;

    begin

COMMENT ------------------------ STATION ---------------------------;
        COMMENT
*
* Provides the link into the communications line for
* a user.
*
*  Variables :
*    u             ref to user associated with the
*                  station
*    buffin,
*    buffout       buffers linking the station to u
*    p             ref to the top level protocol being
*                  used
*    min, mout     ref's to messages - incoming and
*                  outgoing respectively
*   timesend     time spent sending this message
*   rate         the transfer rate experienced by the
*                last message
*   nomsges      number of messages that the station
*                can hear at the moment
*   transmitting,
*   backing,
*   receiving    booleans indicating the state of the
*                station
*   rready       boolean indicating an incoming message
*   sent         boolean indicating whether the last
*                message was successfully sent or not
*   token        boolean indicating the presence of
*                a free token
*   d            matrix for this station
*
*  Procedures :
*   alert        interrupts the station to indicate an
*                incoming message after setting rready
*   initialize
*;

   ref(buffer) buffin, buffout;
   ref(protocol) p;
   ref(message) min, mout;
   real timesend, rate;
   integer nomsges;
   boolean transmitting, receiving, rready, sent, token, backing;
   ref(matrix) d;

   procedure alert;

      begin

      rready := true;
      interrupt(1)

      end   *** alert ***;

   procedure initialize;

      begin

      if u == none then
         neterr(3,this station,"blarg");
      buffin :- new buffer("in-buff",u,this station);
      buffout :- new buffer("out-buff",u,this station);
      u.buffout :- buffin;
      u.buffin :- buffout;
      timesend := 0.0;
      nomsges := 0;
      sent := true;
      min :- mout :- none;
      d :- masterdist;

      end   *** initialize ***;
```

```
    end *** station ***;

station class tstat;

    begin

    ref(entity) t;
    ref(protocol) p;

    p :- new tokenring(this station);
    initialize;
loop:
    waitstn.waituntil(token or rready);
    if token then
        begin
        t :- tokenq.coopt;
        if zyqtrace > 0 then netnote(6,"TOKEN SEIZED",none,0);
        if buffin.full or not sent then
            begin
            if sent then
                begin
                timesend := 0;
                mout :- buffin.takeout;
                end;
            timesend := timesend - time;
            p.send(mout);
            timesend := timesend + time;
            if sent then
                begin
                net.messages := net.messages - 1;
                net.obs := net.obs + 1;
                rate := mout.size/(timesend * 1000);
                net.through := net.through + rate;
                net.sqthru := net.sqthru + (rate ** 2);
                net.tsend := net.tsend + timesend;
                end
            else
                net.erred := net.erred + 1;
            if zyqtrace > 0 then
                netnote(7,"MESSAGE STATUS IS ",mout,0);
            end;
        t.schedule(0.0);
        if zyqtrace > 0 then netnote(6,"TOKEN RELEASED",none,0);
        token := false;
        end
    else
        begin
        p.receive;
        if min.status = 3 then
            begin
            net.walk := net.walk + (time - min.timein);
            buffout.putin(min);
            end;
        rready := false;
        end;
    repeat;

    end   *** tstat ***;

station class estat;

    begin

    ref(matrix) e;
    integer i;

    boolean procedure abletosend;

        abletosend := (buffin.full or not sent) and
                      (nomsges = 0 and not backing);

    initialize;
    e :- new matrix(net.nostats);
    for i := 1 step 1 until net.nostats do
        begin
        e.dist(i) :- new distrecord;
        e.dist(i).distance := d.dist(i).distance;
        e.dist(i).station := if i<u.my_id then i else i + 1;
        end;
    d :- sorted(u, e);
    p :- new ethernet(this station);
loop:
        begin
        waitstn.waituntil((this station qua estat.abletosend)
                                     or rready);
        transmitting := (nomsges = 0) and (not backing);
        receiving := nomsges = 1;
        rready := false;
        if receiving then
            begin
            interrupted := 0;
            p.receive;
            if (min =/= none) then
                begin
                if (min.dest = u.my_id) then
                    begin
                    net.walk := net.walk + (time - min.timein);
                    buffout.putin(min);
                    end;
                end;
            min :- none;
            receiving := false;
            end
        else
            begin
            if transmitting then
                begin
                if sent then
                    begin
                    mout :- buffin.takeout;
                    timesend := 0.0;
```

```
                  end;
               timesend := timesend - time;
               p.send(mout);
               timesend := timesend + time;
               if sent then
                  begin
                  net.obs := net.obs + 1;
                  rate := mout.size/(timesend * 1000);
                  net.through := net.through + rate;
                  net.sqthru := net.sqthru + (rate ** 2);
                  net.tsend := net.tsend + timesend;
                  end
               else
                  net.erred := net.erred + 1;
               transmitting := false;
               end;
            end;
         interrupted := 0;
         end;
      repeat;

      end   *** estat ***;

      procedure netnote(index,action,m,n);

         value action;text action;
         integer index, n; ref(message)m;

         begin

         real t;
         ref(entity)c;

         procedure intout(n); integer n;

            begin

            integer p;

            if n < 0 then
               begin
               n := -n;
               outf.Outchar('-');
               end;
            p := if n < 10 then 1 else
                  if n < 100 then 2 else
                  if n < 1000 then 3 else
                  if n < 10000 then 4 else
                  if n < 100000 then 5 else 10;
            outf.Outint(n, p);

            end   *** intout ***;

         procedure realout(x); real x;

            begin
         integer p;

         if x < 0 then
            begin
            x := -x;
            outf.Outchar('-');
            end;
         p := if x < 10.0 then 5 else
              if x < 100.0 then 6 else
              if x < 1000.0 then 7 else
              if x < 10000.0 then 8 else
              if x < 100000.0 then 9 else 0;
         if p = 0 then outf.outreal(x, 5, 10)
                  else outf.outfix(x, 3, p);
         end  *** realout ***;

      switch message := n1, n2, n3, n4, n5, n6,
                        n7, n8, n9, n10, n11;
      t := Time;
      c :- Current;
      if (Abs(t)-zyqnotelastt) > 0.0005 then
         begin
         zyqnotelastt := t;
         printreal(t);
         end;
      if zyqnotelastc =/= c then
         begin
         outf.Setpos(12);
         zyqnotelastc :- c;
         outf.Outtext(c.title);
         end;

      outf.setpos(25);
      outf.outtext(action);
      outf.outchar(' ');
      goto message(index);

n6:   comment t.schedule(0)- token t releases (token ring);
n9:   comment t.passivate  - ring idle;
n10:  comment p.receive - message arriving at station;
n11:  comment p.receive - message end arrives at station;
      goto exit;

n1:   comment s.backoff    - station backing off;
      outf.outtext("FOR ");
      intout(n);
      outf.outtext(" SLOT TIMES");
      goto exit;

n2:   comment p.send(m)     - sends m to m.dest;
      intout(n);
      goto exit;

n3:   comment message(u)    - u generates a new message;
      intout(m.dest);
```

```
        outf.outtext(" OF LENGTH ");
        intout(m.size);
        goto exit;

n8:     comment receive        - u receives a message;
        intout(n);
        goto exit;

n4:     comment b.putin(m)     - message put in buffer b;
        if current in user then
            begin
            outf.outtext(" FOR USER ");
            intout(m.dest);
            end
        else
            begin
            outf.outtext(" FROM USER ");
            intout(m.source);
            end;
        goto exit;

n5:     comment b.takeout(m) - message taken out of buffer b;
        if current in user then
            begin
            outf.outtext(" FROM USER ");
            intout(m.source);
            end
        else
            begin
            outf.outtext(" FOR USER ");
            outf.outtext("   ");
            intout(m.dest);
            end;
        goto exit;

n7:     comment p.send(m)      - message sent and its status is ...;
        intout(m.status);
        goto exit;

exit:   outf.outimage;

        end *** netnote ***;

    procedure neterr(index,e,call);

        value call; text call;
        integer index;
        ref(entity) e;

        begin

        procedure intout(n);

            integer n;

            begin
        integer p;

        if n < 0 then
           begin
           n := -n;
           outf.Outchar('-');
           end;
        p := if n < 10 then 1 else
             if n < 100 then 2 else
             if n < 1000 then 3 else
             if n < 10000 then 4 else
             if n < 100000 then 5 else 10;
        outf.Outint(n, p);

        end   *** intout ***;

     procedure realout(x);

        real x;

        begin

        integer p;

        if x < 0 then
           begin
           x := -x;
           outf.Outchar('-');
           end;
        p := if x < 10.0 then 5 else
             if x < 100.0 then 6 else
             if x < 1000.0 then 7 else
             if x < 10000.0 then 8 else
             if x < 100000.0 then 9 else 0;
             if p = 0 then outf.outreal(x, 5, 10)
                 else outf.outfix(x, 3, p);

        end   *** realout ***;

     switch case := e1, e2, e3, e4, e5, e6;

     sysout.setpos(23);
     outtext("CLOCK TIME = ");
     if time > 99999.0 then outreal(time, 5, 12)
     else outfix(time, 3, 10);
     outimage;
     abort(false);

     outtext("**CAUSE : CALL ON '");
     outtext(call);
     outtext("'");

     outimage;
     outtext("CURRENT == ");
     outtext(current.title);
```

```
        outimage;
        goto case(index);

e1:     comment B.PUTIN
        outtext("ATTEMPT TO PUTIN A NULL MESSAGE BY '");
        outtext(e.title);
        outtext("'");
        outimage;
        goto stop;

e2:     comment B.TAKEOUT;
        outtext("ATTEMPT TO TAKEOUT A NULL MESSAGE BY '");
        outtext(e.title);
        outtext("'");
        outimage;
        goto stop;

e3:     comment U.anything;
        outtext("UNDEFINED USER USED BY '");
        outtext(e.title);
        outimage;
        goto stop;

e4:     comment ILLICIT MESSAGE PUTIN;
        outtext("ILLEGAL MESSAGE REMOVAL BY ");
        outtext(e.title);
        outimage;
        goto stop;

e5:     comment UNDEFINED SEND;
        outtext("PROTOCOL UNDEFINED IN CALL BY ");
        outtext(e.title);
        outimage;
        goto stop;

e6:     comment UNDEFINED RESUME;
        outtext("RESUME UNDEFINED WHEN USED BY ");
        outtext(e.title);
        outimage;
        goto stop;
stop:   abort(true);

        end   *** neterr ***;

ref(matrix) procedure loaddistance(nostats);

    integer nostats;

    begin

    integer i;
    ref(matrix) m;

    m :- new matrix(nostats);
    outtext("Insert the distance from one station to the next.");
    outimage;
    outtext("(If modelling an Ethernet then the last distance = 0)");
    outimage;
    for i := 1 step 1 until nostats do
        begin
        outint(i,4);
        outtext(" to ");
        outint((if i = nostats then 1 else i+1),4);
        outtext("?");
        outimage;
        inimage;
        m.dist(i) :- new distrecord;
        m.dist(i).distance := inreal;
        m.dist(i).station := if i = nostats then 1 else i + 1;
        end;
    loaddistance :- m;

    end    *** loaddistance ***;

ref(matrix) procedure sorted(u, d);

    ref(user) u;
    ref(matrix) d;

    begin

    integer i, j, l, r;
    real dleft, dright, totald;
    ref(matrix) temp;

    temp :- new matrix(net.nostats);
    for j := 1 step 1 until net.nostats do
        temp.dist(j) :- new distrecord;
    l := u.my_id - 1;    dleft := 0.0;
    r := u.my_id;        dright := 0.0;
    i := 1;              totald := 0.0;
    for j := 1 while (l > 0 and r < net.nostats) do
        begin
        if (dleft + d.dist(l).distance)
             < (dright + d.dist(r).distance) then
            begin
            temp.dist(i) :- d.dist(l);
            dleft := dleft + d.dist(l).distance;
            temp.dist(i).distance := dleft - totald;
            totald := dleft;
            i := i + 1;
            l := l - 1;
            end
        else
            if (dright + d.dist(r).distance)
                 < (dleft + d.dist(l).distance) then
                begin
                temp.dist(i) :- d.dist(r);
                dright := dright + d.dist(r).distance;
                temp.dist(i).distance := dright - totald;
                totald := dright;
                i := i + 1;
            r := r + 1;
            end
         else
            begin
            temp.dist(i) :- d.dist(r);
            temp.dist(i + 1) :- d.dist(l);
            dleft := dright := dright + d.dist(r).distance;
            temp.dist(i).distance := dright - totald;
            temp.dist(i + 1).distance := 0.0;
            totald := dright;
            i := i + 2;
            r := r + 1;
            l := l - 1;
            end;
      end;
   for j := l step -1 until 1 do
      begin
      temp.dist(i) :- d.dist(j);
      dleft := dleft + d.dist(j).distance;
      temp.dist(i).distance := dleft - totald;
      totald := dleft;
      i := i + 1;
      end;
   for j := r step 1 until (net.nostats - 1) do
      begin
      temp.dist(i) :- d.dist(j);
      dright := dright + d.dist(j).distance;
      temp.dist(i).distance := dright - totald;
      totald := dright;
      i := i + 1;
      end;
   sorted :- temp;

   end   *** sorted ***;
```