A Concurrent Object-Oriented Framework for Simulation of Network Protocols

Hossein Saiedian and Stan Wileman
Department of Computer Science, University of Nebraska, Omaha, NE

The benefits of simulation to the development of complex systems are well known and frequently exploited. This article discusses the advantages of simulating an inherently concurrent system, namely, a collection of computer systems using the carrier sense multiple access protocols to access a bus-based local area network. The rationale for developing an object-oriented simulation is presented and suitably illuminated source code outlines are provided. Implementation of the network based on the simulation is described and additional simulations are discussed. Owing chiefly to the concurrent object-oriented approach taken, the work described here can be used as the foundation of a network simulation toolbox, thus vastly simplifying the study of proposed networks and protocols.

1. INTRODUCTION

Object orientation, as it originally appeared in SIMULA, was motivated by two overriding concerns: to provide natural structural components for simulation purposes and construct reusable program components. Object orientation has now proven to be a successful methodology outside of these areas. Its concepts have been used for conceptual modelling, analysis, design, and programming. Furthermore, the methodology has now become widely accepted as an effective and promising approach to software development. Therefore, there is no longer a need to build a case for its use.

The central concept in object orientation is an object. Objects provide a useful basis for an organizational paradigm. That is, they allow a large system to be decomposed into manageable units. Furthermore, they promote reusability, i.e., for composing systems from "plug-compatible" [1] objects. Thus, new simulation models or real applications potentially should be constructed by combining existing objects. The basic features of objects (which are extensively emphasized in this article) include the following:

- Encapsulated local state: that is, the local state of an object as well as the "services" it provides are protected from outside access.
- State changes via message passing: messages are sent to an object to obtain its services. After accepting a message, an object may change its state. Thus, state changes are triggered by acceptance of a message. (Message passing is sometimes referred to as *method invocation.*) Two other essential characteristics of objects—concurrency and nondeterminism—have recently been emphasized:
- Objects should be inherently concurrent, i.e., they should perform their activities concurrently.
- Objects should have a nondeterministic input to computational steps [2]. This kind of nondeterminism mirrors the real world, in which the sequence of events in which objects participate cannot be predicted.

Of course, both of these characteristics imply that the underlying language of implementation should support concurrency. Examples of such languages include Ada and ABCL [3]. Important references to concurrency and nondeterminism issues in object-orientation include [2–5].

This article describes an object-oriented approach to the simulation of medium-access protocols in a local area network (LAN). LANs are becoming the most common way of connecting a series of computer systems to allow them to communicate with each other and share common resources. It is therefore very important for network designers to evaluate the ability or expected behavior of LANs that use different communication protocols. We report

Address correspondence to Prof. Hossein Saiedian, Dept. of Computer Science, University of Nebraska, Omaha, NE 68182.

an object-oriented simulation of the medium-access control (MAC) sublayer (as defined by IEEE-802). There are several classes of MAC protocols. We focus on a class of protocols known as carrier sense multiple access (CSMA).[1]

A LAN in our simulation-model abstractions is a set of user nodes connected to a communication channel via network interface units in a particular LAN topology. We begin by describing LAN concepts in object-orientation terms:

- Nodes of the network are represented as objects.
- The communication channel connecting the nodes is modelled as an object.
- Network interface units between network nodes and the communication channel are objects; such objects are instantiated from a uniform and basic class structure.
- Communication between nodes of the network, interface units, and the communication channel is modelled as patterns of message passing.
- All objects are self-contained and are by nature nondeterministic and concurrent processes.
- The simulation model is independent of a particular implementation (although we suggest a language that supports concurrency and nondeterminism).

2. OBJECT-ORIENTED SIMULATION

In 1969, R. M. Graham [6, p. 17] made the following comments about the way computer systems were being built: "We build systems like the Wright brothers built airplanes—build the whole thing, push it off a cliff, let it crash, and start over again." Unfortunately, this approach is still common in some areas of systems development. Computer simulation avoids this approach by allowing a software system to be analyzed and test driven before an actual system is designed and constructed. The goal of analysis is to further our understanding of the design and operations of an actual product and to examine its behavior under certain conditions.

In an object-based framework, we speak of objects, messages, and object responsibility. To quote Dan Ingalls [7, p. 290], "instead of a bit-grinding processor... plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires. This approach to programming, which deals with a "universe" of objects and describes their interaction, is similar to computer discrete-event simulation [8]. In discrete-event simulation, a computer "model" of various elements of simulation is developed and their interaction is described. This is the principal of object-oriented programming, that is, the programmer describes the various objects in the universe and how they interact with one another. Thus, in object-oriented programming, computation by definition is simulation [8]. Object orientation can therefore prove very useful in simulation (for which it was originally intended). The objects of the real world and their interaction can be modelled in an object-oriented simulation. Potential changes in the behavior of the outside universe can easily be mirrored in the simulation model by modifying (or even replacing) the corresponding object of the model. Furthermore, systems that have not yet been developed and exist only on drawing boards can be modelled and simulated, thus avoiding the Wright brothers' approach. In addition, the object-oriented approach facilitates simulation of important concepts; that is, it emphasizes the objects of the real world and what they do, rather than a particular object implementation. This feature of object orientation allows one to consider experimenting with a simple implementation consisting of important objects as well as objects that serve as "stub" or "driver" (or, in general, "placeholder"), which can aid in the definition, formulation, and testing of object interfaces and interactions; subsequent modifications can be made to the model to develop a more detailed implementation of some or all of the objects on a controlled, gradual basis.[2]

Simulation consists of modelling objects that have a state and a set of operations. The objective is to model a physical entity (or an abstract concept or process) so that one can observe its behavior or experiment with it. For example, before an airplane is built, engineers develop a simulated model to verify that the approach to airplane construction is workable. Thus, simulation makes it possible (and affordable) to try out different approaches to designing an airplane to determine which approach is more plausible and will lead to design of actual planes that do not crash.

[1]There are several versions of CSMA protocols, which will be discussed later.

[2]As observed in Nelson and Byrnes [9], this approach is similar to Boehm's spiral model of software development, in which a series of prototypes are designed and tested before arriving at an actual version.

In an experimental simulation, the simplest objects are designed and experimented with, then larger objects are developed until the entire physical entity (or abstract concept) has been simulated. Object-orientation concepts best facilitate such an approach. They can provide a better solution because different solutions can be examined to identify fundamental objects that are common to many parts of a solution. Not only do simulation components have a correspondence with real-world entities, but new simulation models can be developed that are mostly specializations of earlier models through reuse of earlier components. This leads to low development cost. Once the simulation model has proven successful, the designer's job is merely to implement the software or hardware version of the solution. This latter discussion reveals another major advantage of an object-based approach to simulation, that is, libraries of reusable objects can be produced that allow new simulation or application models to be developed from preexisting objects. This unique advantage of object-based simulation decreases the development time and cost and improves the quality of future simulations.

3. CSMA PROTOCOLS

A number of standards have been defined for the MAC sublayer for LANs. These standards address different approaches used to control access to the physical transmission medium. CSMA is the most commonly used access method for LANs that employ a bus (or tree) topology. In fact, one of the most popular LAN architectures, Ethernet, uses the CSMA protocol at the MAC layer.

Each network station in the CSMA protocols listens to the transmission medium (i.e., the carrier) and acts accordingly. That is, if the carrier is "quiet" and a station has data to send, then it can transmit its data. If the carrier is busy, then the station waits until the carrier becomes idle. There are a number of variations in CSMA protocols. They are called 1-persistent, nonpersistent, and *p*-persistent CSMA [10]:

- In a 1-persistent CSMA protocol, a station that has data to send listens to the carrier. If the carrier is busy, then the station waits until it becomes idle. When the carrier becomes idle, the station sends its data. If a collision occurs, the station waits a random time and starts again. This protocol is called 1-persistent because the sending station transmits with a probability of 1.
- In a nonpersistent CSMA, an attempt is made to be less greedy: if the carrier is idle, the station sends its data. If the carrier is being used, the station does not sense it continually; instead it waits a random time and then repeats the process. This protocol leads to better utilization of the transmission channel.
- *P*-persistent CSMA applies to slotted channels: the sending station transmits after sensing an idle carrier with a probability p. With a probability $q = 1 - p$, it waits until the next slot.

For the purpose of this simulation, we considered simulating the nonpersistent CSMA, which facilitates communication between more conscious user stations.[3] All elements of the network are represented as objects. Communication among user stations is modeled as patterns of message passing.

4. NOTATION

We will use the simple pictorial notation shown in Figure 1 to represent objects. When explicitly shown, an object is conceptually depicted by a circle. An object's interface (i.e., the messages it accepts) are shown in rectangular box(es). Although it is not distinctly important, we will show via dashed lines which objects send messages to a given object. Similarly, a solid arrow is used to show the objects to which a given object sends messages.

5. THE OBJECT-ORIENTED SIMULATION MODEL

The basic model of simulation is fairly simple. There are n user stations (nodes) that are connected to a

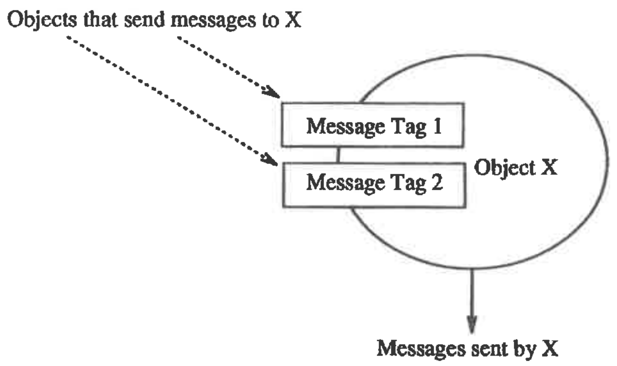

Figure 1. Graphical object representation.

[3]An improvement for persistent and nonpersistent CSMA protocols allows the stations to abort transmission once they detect a collision. This improved protocol is called CSMA with collision detection (CSMA-CD). We are enhancing our simulation model to include collision detection.

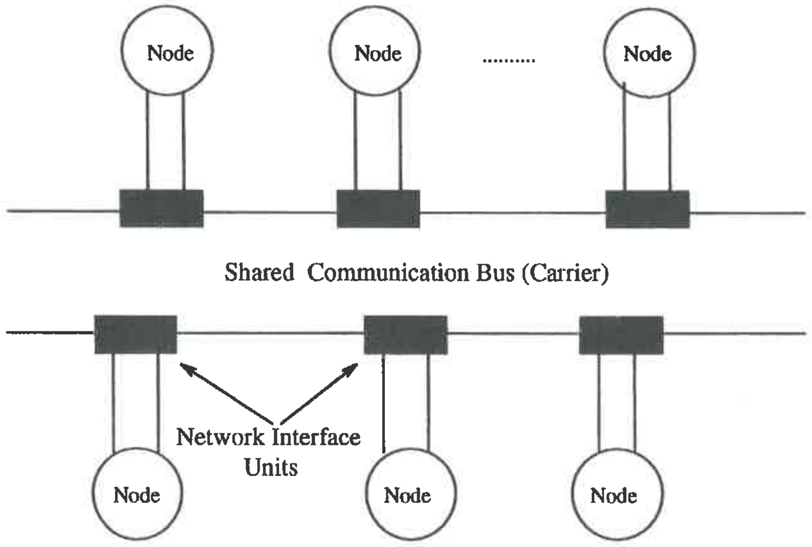

Figure 2. Basic components of network.

shared communication bus through network interface units, as depicted in Figure 2. At this level of abstraction we see three major classes of objects:

- An object representing the shared communication bus connecting the user stations. We will refer to this object as the Carrier object.
- A class of objects representing the network interface units (i.e., the interface between the Carrier object and the user stations)
- A class of objects representing the user stations (i.e., nodes of the network)

These object classes are discussed next.

5.1 The Carrier Object

The Carrier object behaves like a bus. Its state changes from idle to busy and vice versa as described below:

- Initially, the state of the Carrier object is idle. When a message carrying a character is sent to the Carrier object, it will change its state from idle to busy. The Carrier object will deliver the character, by sending messages, to all user stations connected to it.[4]
- When a message carrying a postamble character is received, the Carrier changes its state from busy to idle once the postamble character has been sent to all user station objects.

The Carrier object thus accepts two kinds of messages, as depicted in Figure 3. Both of the messages are sent by the (Transmitter object of the) network interface unit. Message tags are as follows:

ReceiveChar,[5] to receive a character from the network interface units

Sense. The Carrier object replies by returning a "busy" or "idle" response.

The Carrier object communicates with the (Receiver object of the) network interface units by sending *ReceivePacket* messages to them.

5.2 The Network Interface Object

The network interface unit is modeled by two objects.

The Transmitter object. Each Transmitter object receives messages from its local user object. Such messages contain data to be delivered to another user object via the Carrier object. The Transmitter object constructs a packet[6] from the user object's data and sends the packet, one byte at a time, to the Carrier object using the nonpersistent CSMA algorithm (to be described later). Before sending messages (which contain a single byte) to the Carrier object, the Transmitter has to "sense" the Carrier to

[4]More specifically, the exchange of messages between the Carrier object and the user stations takes place through the network interface objects. Thus, the Carrier object will communicate with the user interface objects and the user interface objects will in turn communicate with the user stations.

[5]The parameters of the messages sent to this and other objects will be shown later.

[6]The elements of each packet will be explained later.

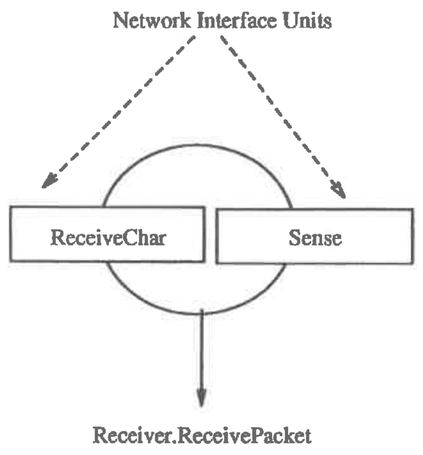

Figure 3. Carrier object's interface.

ensure that it is idle. The interface of the Transmitter object includes only one method for receiving data from the user objects. We use the name *ReceiveData* for this interface method (Figure 4).

The Receiver object. This object has individual characters delivered to it by the Carrier object. The Receiver object

- receives single characters from the Carrier object and constructs a packet
- recognizes whether or not the packet is free of error
- buffers the packet if the destination address in a correct packet indicates the local user object's address
- strips out the useful data of a packet and delivers it to the local user object

The interface of the Receiver object includes only two methods. One method is called *ReceivePacket*, to be called by the Carrier, and the other one is called *GiveData*, to be used by the local User object (Figure 5).

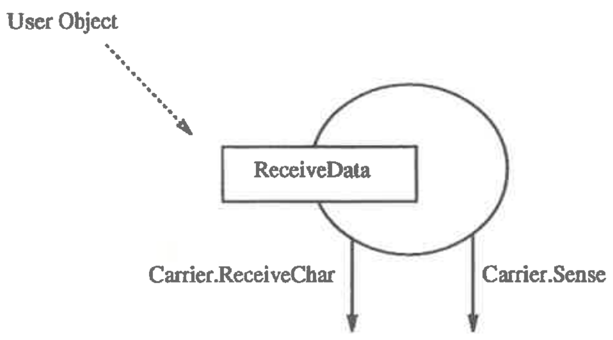

Figure 4. Transmitter object.

5.3 User Objects

The user stations or nodes of the network are also modeled as objects.[7] Each User object interacts with the network interface objects to send and/or receive data packets. Each node has its own Transmitter and Receiver objects.

When a User object has data to transmit, it sends it to the Transmitter object by calling the *ReceiveData* method of the Transmitter object. For each packet sent to the Transmitter, the User object awaits an acknowledgment. If an acknowledgment is not received within a reasonable time (to be discussed later), the User objects use a binary exponential backoff algorithm to calculate a random time to wait before retransmitting the packet. The transmission of a packet is abandoned after four unsuccessful attempts. The interface of the User object is shown in Figure 6.

The four basic objects described so far, as well as their interactions (i.e., the patterns of message passing), are depicted in Figure 7 (other objects will be described later.) Directed arcs show the patterns of messages sent. For example, an arrow going from a User object to a Transmitter object indicates that the User object sends a message to the Transmitter object. The pattern *Transmitter.ReceiveData* states that the *ReceiveData* interface of Transmitter has been invoked.

6. LANGUAGE OF IMPLEMENTATION

Object orientation is a philosophy and should not be limited to any programming language. However, some languages are better suited to modelling objects than others. A language that provides facility for defining objects is better than another language that does not. Ideally, a language used for object-oriented programming should support all the features of objects. The facility for data abstraction and the ability to define many objects of a particular class are perhaps the most important.

[7]At each node, there may be a PC, minicomputer, or mainframe computer with many users. We are not modelling the behavior of each such individual user. Rather, we consider the communication patterns of each node (regardless of the kind of machine used at that node) with other nodes of the network. Thus, we abstract out low-level details and simply view each node as being an object that communicates with other nodes through the Carrier object.

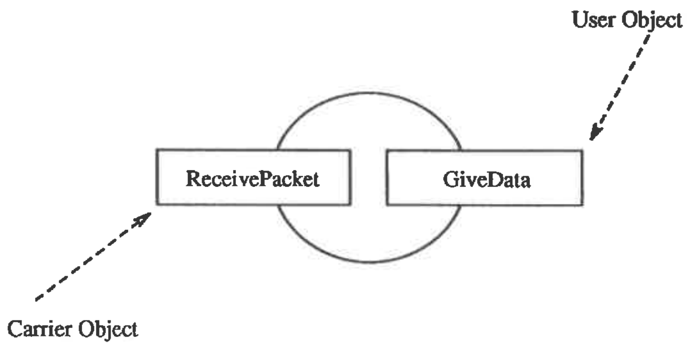

Figure 5. Receiver object.

For implementation of our simulation, we considered two different languages that support objects, C^{++} and Ada. Fisher [11] stated that "programming languages are neither the cause of nor the solution to software problems, but because of the central role they play in all software activity, they can either aggravate existing problems or simplify their solution." In the case of both Ada and C^{++}, the effect is one of simplification. They both provide powerful mechanisms for the development of software systems and allow us to deal with problems whose solutions were previously too complex to manage.

We chose Ada. The primary reason for choosing Ada was its support for concurrency. We wanted to simulate a real situation. In a real network environment, the objects (e.g., the carrier, receiver, transmitter, users, and so forth) proceed in parallel. Because Ada supports concurrency, our simulation was a closer representation of a real environment.

Ada is not an object-oriented programming language pr se, but it provides a rich set of constructs for building and enforcing abstraction and encapsulation and for defining many objects of a particular class. There is considerable interest in object-oriented development using Ada. In fact, Ada has frequently been used in conjunction with and to demonstrate object-oriented design methodologies [12–16].[8] With the requirement that an Ada compiler cannot be distributed as an "Ada compiler" until it has successfully passed a series of strict validation tests, and with its inherent features for modern software engineering principles [12], program portability and reliability can be virtually guaranteed. In addition, Ada has also been frequently used for modelling systems that exhibit concurrent and nondeterministic behavior.

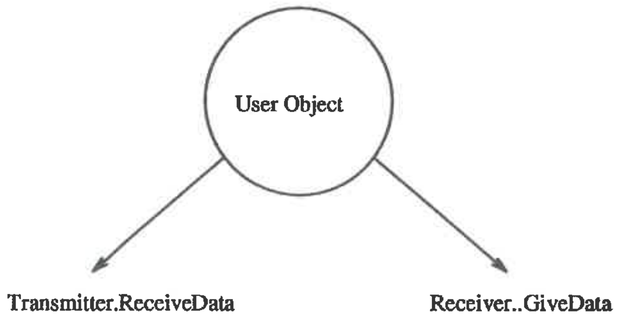

Figure 6. Interface of a user object.

6.1 Concurrent Objects

In sequential object-oriented programming languages (e.g., C^{++}), objects are implemented as sequential procedures. In Ada, there are both sequential and concurrent objects. Sequential objects are represented via packages, whereas concurrent objects are implemented as tasks. As stated earlier, we would like to exploit Ada's concurrent facility to represent objects of the network that operate concurrently. Thus, we use Ada's task facility. Ada allows task types to be declared; this acts as class specification (Figure 8). Concurrent objects as represented via tasks are independent and execute their operations (including sending and receiving message) concurrently.

In Ada, concurrent objects can be developed to be of either of the following kinds:

- *Passive*. Such an object is still realized as independent and capable of executing in parallel, but while no calls are made to it (i.e., no messages are sent), the object suspends itself and awaits calls.

[8]One feature of object orientation not supported by Ada is inheritance. However, it is common in practice to design as if inheritance were available and then use implementation techniques to fake if a language such as Ada does not directly support inheritance [13]. The common approach in Ada, as Booch has stated, is to use a combination of generic packages (representing parameterized classes) and discriminated private types (representing abstract classes) [13].

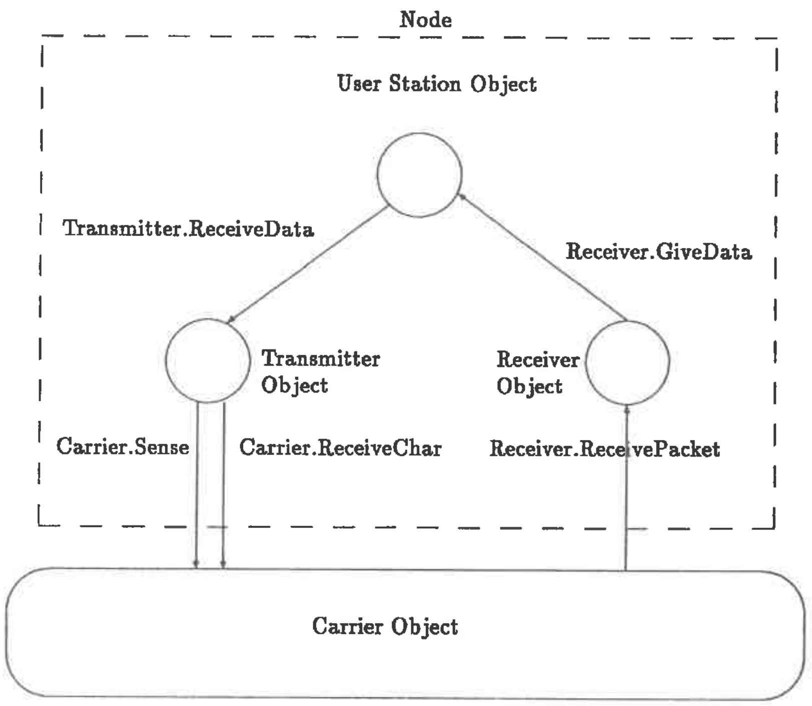

Figure 7. Network objects and their interfaces (arrows show data flow).

An example is the Carrier object described before. While the User objects are not exchanging messages, the Carrier object sits idle.

- *Active*. Such an object is conceptually continually executing its own internal operations, resulting in changes in its own state. An example is the User object. The User objects continually execute internal operations, which may include sending or processing messages.

For an overview of active and concurrent objects, see Agha [4].

```
task type ConcurrentObject is
    entry Operation$_1$
    entry Operation$_2$
      :
    entry Operation$_n$
task body ConcurrentObject is
begin
  loop
    select
      accept Operation$_1$
      ...
      end Operation$_1$
      accept Operation$_2$
      ...
      end Operation$_2$
        :
      accept Operation$_n$
      ...
      end Operation$_n$
    end select
  end loop
end ConcurrentObject;
```

Figure 8. A concurrent object class specification in Ada.

7. OBJECT REPRESENTATION

In this section, we give Ada code for major aspects of both interface and behavior of important objects in the simulation. We include sufficient comments in the code to make them self-descriptive.

7.1 Simulation of the Carrier Object

The Carrier object behaves like a bus. It accepts messages with single-character parameter and delivers the character to every (Receiver object of a) User object connected to the network. It has a state variable called *current_status*, which represents its state (busy or idle). The state of the Carrier object changes from idle to busy when it accepts the first character of a packet[9] from the (Transmitter object of the) User object. The state is changed back to idle when the last character of a packet (called postamble) is delivered to all User objects.

The interface of the Carrier object is shown in Figure 9. It accepts two kinds of messages, *Sense* and *ReceiveChar*. *Sense* messages are sent by the Transmitter objects in an effort to sense the state of the Carrier. *ReceiveChar* is used to send a character to the Carrier. The Carrier's behavior (body) is shown in Figure 10.

[9]The structure of network packets is given in the next subsection.

```
task Carrier is
  entry Sense (s: out CARRIER_STATUS_TYPE);
  entry ReceiveChar (c: in CHARACTER);
end Carrier
```

Figure 9. Interface of the carrier object.

7.2 Simulation of the Transmitter Object

The Transmitter object waits for data from its own User object. Once data has arrived, the Transmitter object constructs a packet. Important fields of the packets used in the simulation are shown below:

Destination Addr	Source Addr	Data	Check Sum	Postamble Char
3 Bytes	3 Bytes	1-32 Bytes	2 Bytes	1 Byte

A simple modulo operation was used to compute the checksum. The interface of the Transmitter object is shown in Figure 11. Note that it is an object class. In other words, one Transmitter object class was defined and an actual object was instantiated for each User object.

Once a packet is constructed by the Transmitter object, it is sent to the Carrier object to be delivered to its destination. The Transmitter object accesses the Carrier object using a CSMA algorithm. If the Carrier object is busy delivering another packet, the Transmitter waits up to one second before trying again. Once the Transmitter senses an idle Carrier object, the packet is sent to it one character at a time. In the object's interface, there is a message entry called *Get_User_Addr*. The User object is able to send its name (that is, its network address) in such a message to its Transmitter object. To make the figures simpler, this aspect of the Transmitter's interface was not shown. The behavior (that is, its body) of the Transmitter is shown in Figure 12.

```
task Transmitter_Class type is
  entry ReceiveData (arc, dest: in ADDR_TYPE;
          data: in MSG_TYPE)
  entry Get_User_Addr (addr: in ADDR_TYPE);
  —see object's body for definition of the above entry
end Transmitter_Class
```

Figure 11. Interface of the transmitter object.

7.3 Simulation of the Receiver Object

The Receiver object is also defined as a class. Its primary responsibilities are

```
task body Carrier is
  xfer_char: CHARACTER;
  current_status: CARRIER_STATUS_TYPE;
begin
  loop
    current_status := IDLE;
    while current_status = IDLE loop
      select
        accept Sense (...) do;
            —Return the current status.
            —First Transmitter to sense
            —an idle carrier will gain
            —access. Status is changed:
            current_status := BUSY;
        end Sense
      or
        terminate;
      end select
    end loop;
    xfer_char := NO_VALUE;
    while xfer_char/= POSTAMBLE loop
      select
        accept Sense (...) do
            —Just return the current status
        end Sense;
      or
        accept ReceiveChar (...) do
            xfer_char := c;
            —Save c in xfer_char;
        end
        —Send xfer_char to all Receiver objects
      end select
    end loop—Stop when postamble was sent
  end loop
end carrier;
```

Figure 10. Behavior implementation of the carrier object.

```
task Transmitter_Class type is
  —Many local variables, including
  packet_out: PACKET_TYPE;
  packet_length: INTEGER;
  line_status: CARRIER_STATUS_TYPE;
  char_to_send: CHARACTER;
begin
  accept Get_User_Addr (...)
    —The purpose of this message is to pass
    —the User object name to the Transmitter
    —To make diagrams simple, this part of
    —the interface was not shown in diagrams
  end Get_User_Addr;
  loop
    select
      accept ReceiveData (...) do
          —Save data in local storage
      end ReceiveChar;
    or
      terminate;
    end select;
    —Checksum is computed; a packet is built;
    —Packet is sent over the network:
    Carrier.Sense(line_status);
    while line_status = BUSY loop
      —delay for random time 0–1 second;
      —Sense the carrier again:
      Carrier.Sense(line_status);
    end loop
    for i in 1..frame_length loop
      char_to_send := packet_out (i);
      Carrier_ReceiveChar(char_to_send);
    end loop
  end loop;
end Transmitter_Class
```

Figure 12. Behavior implementation of the transmitter object.

- To receive packets from the Carrier (one character at a time)
- To reconstruct the packet if necessary (i.e., if packet's destination address matches the address of the local User object)
- To ensure that the packet has not been corrupted during transmission (by recalculating the checksum and comparing it with the checksum of the packet)
- To strip out the data from the packet and buffer it locally
- To make buffered packets available to the User object when they are requested.

The interface and behavior of the Receiver object are shown in Figures 13 and 14, respectively. In the object's interface, there is a message entry called *Get_User_Addr*. The User object is able to send its name (that is, its network address) in such a message to its Receiver object.

7.4 Simulation of the User Objects

The User objects send and receive packets over the carrier. In a sense, the User objects form the application layer of our simulation model.

Once a User object has produced some data to be transmitter over the network, it sends it to the Transmitter object. This object also exchanges messages with the Receiver object to obtain the data as well as acknowledgments sent to it by other User objects in the network. Every time a User object sends some data to another User object, it expects an acknowledgment. If an acknowledgment is not received within a specified period of time, the object uses a binary exponential backoff procedure to calculate a random time to wait before retransmitting the packet. The transmission of a packet is abandoned after four unsuccessful attempts.[10] Although User objects conceptually send messages to each other, a given User object does not receive any messages. It sends messages to its corresponding Transmitter and Receiver object. The messages include:

- Sending its address to both the Transmitter or Receiver objects
- Sending data (to be transmitted to another User object) to the Transmitter
- Sending messages to the Receiver object asking for any buffered data.

The important aspects of the behavioral implementation of the User object are shown in Figure 15.

```
task Receiver_Class type is
  entry ReceivePacket (c: in CHARACTER);
  entry GiveData (src, dest: out ADDR_TYPE;
          data: in MSG_TYPE);
  entry Get_User_Addr (addr: in ADDR_TYPE);
  —see object's body for definition of the above entry
end Receiver_Class
```

Figure 13. Interface of the receiver object.

[10] For simplicity, we decided to cancel any time which elapsed while waiting for an acknowledgment if a data packet was received (when an acknowledgment was expected) and started the time for acknowledgment anew.

8. OBSERVATIONS, DISCUSSION, AND CONCLUSIONS

8.1 Recalcitrant Objects

By simulating flawed network objects, we can observe their behavior before they fail in a corresponding real system. This facilitates the development of network software and hardware components that can continue to operate, perhaps with reduced capacity, in the presence of faulty components. Several of these "rebellious" objects that have been modelled are described below.

A recalcitrant Receiver object that receives packets but does not acknowledge them. Faulty network inter-

```
task body Receiver_Class type is
  —Many local declarations, including
    char_input: CHARACTER;
  packet_buffered: BOOLEAN;
begin
  accept Get_User_Addr (...)
    —The purpose of this message is to pass
    —the User object name to the Receiver
  end Get_User_Addr;
  loop
    select
      when packet_buffered ⇒
          accept GiveData (...)
          :
          end GiveData;
          packet_Buffered := FALSE;
      or
        accept ReceivePacket (...)
        :
        end ReceivePacket
        :
      —A series of algorithmic statements
      —that do the following is put here: determine if the
        a packet
      —belongs to local User object, recalculate the
      —checksum to ensure packet is error free, strip
      —out data from packet, and buffer data locally.
      end select
    end loop
end Receiver_Class
```

Figure 14. Behavior implementation of the receiver object.

```
task body User_Object is
procedure Construct_Message(dest: out ADDR_Type,...) is
  —A destination object is chosen randomly
  —A message is constructed. Each message
  —has identifying data about sender and receiver
end Construct_Message;
function Exponential_Backoff (wait_cycle: in INTEGER)
return FLOAT is
begin
  return (Random Float (2* *wait_Cycle))
end Exponential_Backoff;
procedure Wait_For_Ack (dest: in ADDR_Type,...)
  —Many local declarations; Procedure loops until an
  —ack is received or wait cycles have finished
begin
  while no_ack and wait_cycle < MAX_WAIT_CYCLE loop
    —Attempt to get something from Receiver object
    —If not an ack but a msg, send an ack to sender
    —If an ack, is it the right one?
    —If not, time out waiting for a packet
  end loop;
end Wait_For_Ack;
begin—task body
  ...
  loop
    select
      —If a packet has been buffered, get it
      —Receiver (My_ID).GiveData(...)
      —Process it appropriately
    or
      —No msgs available; construct one and send:
      Construct_Message(...);
      Transmitter (My_ID).ReceiveData(...);
      Wait_For_Ack (dest, ...);
    end select
  end loop;
end User_Object;
```

Figure 15. Behavior implementation of the user object.

face units (NIUs) can result in a variety of anomalous behaviors. Packets might be received correctly (and then delivered to a User object), but a failing Transmitter object would eliminate any possibility of acknowledgments being successfully received by the originator of the message. Because the model (correctly) assumes that the only communication channel between objects is that implemented by the Carrier object, there is no method for the original sending object to distinguish this situation from one in which the packet was never received (perhaps because of a faulty receiver or channel). A failing Receiver object can be distinguished from the failing Transmitter by other nodes, because they will see transmissions from the failing node, but it never replies to (or acknowledges) transmissions directed to it. Of course, this assumes proper operation of the User objects.

A recalcitrant Transmitter object that includes an erroneous checksum in a packet. Flawed packets (bad checksums, incorrect sizes, and so forth) might be generated as a result of a failing NIU, but might also result from errors generated during propagation through the Carrier. Although such failures are much less likely given the current technology for LANs, they can still result; thus, modelling these failures is useful. This can also be used to verify the ability of Receiver objects to distinguish flawed packets from good ones.

A recalcitrant User object that sends packets to nonexisting nodes. Other unusual network activity can be characterized statistically. The transmission of packets with invalid destination or source addresses could be monitored by a User object operating in "promiscuous" mode. While most User objects respond only to packets addressed explicitly to them, others might wish to function as network monitors, receiving and recording information about every packet that appears on the channel. This is different from responding to broadcasts, which are identified as packets containing specially formed destination addresses. A common User object failure is the generation of excessive broadcast activity that saturates the Carrier. Again, a network monitoring station can identify such failures statistically.

8.2 Implications for a Real Network Environment

A valuable aspect of the model presented here is its similarity to actual networks. Although the current model mirrors only three layers of a typical network (the physical layer, the data-link layer, and the application layer), the relative ease with which the objects modelling these layers can be modified makes it possible to examine other network characteristics (see the description of CSMA-CD below). Additional objects can be added to model other layers in a network without disturbing or requiring detailed knowledge of the existing objects. Eventually, a library of objects can be assembled from which a model of an arbitrary network can be constructed.

Modelling can be done at various levels of detail. Timing is not modelled at all in the current effort, but will be crucial in more detailed modelling of a real network. Likewise, physical distance is not modelled; propagation delays through various network components will become important when a more detailed simulation is attempted. These features can be added without departing from the object-oriented philosophy of the model. Messages could be timestamped when generated or processed by an object. For example, a User object with data to transmit could timestamp each message it generates and sends to the Transmitter object, which would then timestamp each packet it assembles as it is passed to the

Carrier object. The Carrier object would further timestamp messages as they are received and delivered to each network interface unit. Of course, this will require that the Carrier coordinate delivery of messages so that they "arrive" at a time consistent with the (simulated) physical placement of nodes and NIUs on the cable.

Objects can obtain the current time from a single Time-Server object. This object will not model any component of a real network. Rather, it continues the object-oriented philosophy of the simulation. Requests from other objects for the current simulation time and the responses generated by the Time-Server will require no time on the "simulation clock."

An important, and possibly difficult, aspect of a real system to model is the perceived passivity of the physical layer. The exact point of division between the physical layer and the data-link layer is often vague. In some senses, this lack of separation is unimportant. However, as more network functions are embedded in (active) hardware components, it is important to realize (and correctly model) those components that are, at least conceptually, passive. We would like to model the physical layer as an entirely passive object, but to correctly realize such activities as noise and packet collisions, some active nature must be imparted to the physical layer.

8.3 An Actual PC Environment

Using a laboratory of dedicated PC systems equipped with Ethernet interfaces, we can implement the current network described by the model. The Carrier object, without provision for collision detection, is modelled by distributed active code in each system, which then uses the actual Ethernet hardware for communication with the Carrier objects in the other systems. The actual physical and data-link layers are therefore not components of the modelled system.

A useful attribute of this implementation of the model is that it exhibits parallelism not present in a multiprogrammed implementation, such as that described previously. In a sense, this implementation can aid in the model's validation, because it should yield results that are consistent with the multiprogrammed implementation. In addition, by revising the Carrier objects to use the Ethernet hardware directly (instead of assuming it is only a vehicle to permit communication among the distributed Carrier objects), we can implement an actual system based on the object-oriented model, not just a model of such a system. The Carrier objects might be more appropriately labelled Ethernet drivers, or packet drivers in this case. A tightly coupled system (multiple processors with shared primary memory) could also be used for a parallel implementation of the system.

User objects in this implementation can implement various "real" applications, including file transfer between systems and file-serving functions for other, conceptually diskless, systems. The ability to implement the model beyond a relatively limited stage depends, however, on the availability of an operating system that supports concurrency or a system that implements concurrency on top of an existing uniprogrammed system (e.g., MS-DOS).

8.4 Variation on Simulation: CSMA-CD

To successfully model CSMA-CD (i.e., CSMA with collision detection) first requires that the possibility of collisions exists. In the current CSMA model, each Transmitter object patiently awaits an idle channel and then transmits its packet without the possibility of collision (because the Carrier object instantly becomes busy when a sense operation returns an idle status, and status sensing is serialized by the Carrier object). Without the possibility of characters being sent by several transmitters in such a way that they coexist in the transmission medium, collisions are impossible (excluding possibly the generation of signals as a result of "echoes" from the modelling of faulty cable termination, another potential enhancement of the model).

Collision generating can be accommodated by the model in several ways. A simple but imprecise approach would allow one or more characters to be accepted by the Carrier object before it reports BUSY to a sense operation. This would model the transmitted signal reaching each remote (from the transmitting) NIU only after transmission of the specified number of characters. Each of these remote NIUs would then receive BUSY status reports. Before receiving such a BUSY status report, these remote NIUs would incorrectly assume that the Carrier object was IDLE (although locally, the IDLE status report might be correct), thus enabling them to transmit and generate a collision. Removing the mandatory transition to BUSY after a sense operation reports IDLE also facilities the addition of nodes that only monitor the status of the Carrier object, because they might sense carrier status without the intention of transmitting.

A more precise implementation of collision generation and detection will require the enhancements described in section 8.2, specifically, that objects are aware of the passage of time, the physical distance between objects, and the speed with which signals propagate in the transmission medium. A collision results from two signals, starting at NIUs physically separated by some minimum distance, encountering each other at a common point on the medium (and in time). The resulting signal, altered in such a way that it can be distinguished from any valid transmission, then propagates bidirectionally to each NIU that receives a collision report. This approach will also require that the Carrier object implement a different technique for detecting the IDLE state because it is unrealistic to expect the Carrier to interpret packets at all, in particular, to interpret certain codes as delimiters marking the end of a packet. The IDLE status can only be reported locally to an NIU when no signals, regardless of content, are present on the channel at the point of physical attachment of the NIU. This implication, requires actions on the essentially passive Carrier object to proceed concurrently. As noted in Bézivin [17], this possibility is precluded by the conception of an object modelled by a sequential process.

8.5 Conclusions

We have described the benefits of an object-oriented approach to the simulation of complex systems that are inherently concurrent. The description included the simulation of the CSMA network protocols. The concurrent object-oriented approach (using Ada) can be used as the foundation of a network simulation toolbox to simplify the study of proposed networks and protocols.

REFERENCES

1. O. Nierstrasz, What is an object? (Panel discussion), in *Proceedings of ECOOP'91 Workshop*, Lecture Notes in Computer Science vol. 612, Springer-Verlag, Berlin, Germany, 1992, pp. 257–264.
2. P. Wegner, Design issues for object-based concurrency, in *Object-Based Concurrent Computing* (M. Tokoro, O. Nierstrasz, and P. Wegner, eds.), Springer-Verlag, Berlin, Germany, 1992, pp. 245–256.
3. A. Yonezawa, ed., *ABCL: An Object-Oriented Concurrent System*, MIT Press, Cambridge, Massachusetts, 1990.
4. G. Agha, Concurrent Object-Oriented Programming, *Commun. ACM* 33, 125–140 (1990).
5. A. Yonezawa and M. Tokoro, eds., *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, Massachusetts, 1987.
6. R. M. Graham, Software Engineering and Society, in *Proceedings of 1969 NATO Conference on Software Engineering*, Scientific Affairs Division, NATO, Brussels, Belgium, 1969, pp. 15–18.
7. D. Ingalls, Design Principles behind SMALLTALK, *Byte* 6, 260–298 (1981).
8. T. Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley, Reading, MA, 1991.
9. M. Nelson and R. Byrnes, Rapid prototyping in an object-oriented pictorial dataflow language, in *Proceedings of the 25th Annual Hawaii International Conference on Systems Sciences*, IEEE-CS, Los Alamitos, California, 1992, pp. 562–563.
10. A. Tanenbaum, *Computer Networks*, 2nd ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
11. D. Fisher, The Common Programming Language Effort in the Development of Defense, presented at Computers in Aerospace Conference, 1977.
12. G. Booch, *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, California, 1987.
13. G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, Menlo Park, California, 1991.
14. P. Jalote, *An Integrated Approach to Software Engineering*, Springer-Verlag, Berlin, Germany, 1991.
15. C. Atkinson, *Object-Oriented Reuse, Concurrency, and Distribution: An Ada-Based Approach*, ACM Press, New York, 1991.
16. I. Sommerville, *Software Engineering*, 4th ed., Addison-Wesley, Reading, Massachusetts, 1992.
17. J. Bézivin, Some experiments in object-oriented simulation, in *Proceedings OOPSLA Conference*, ACM, New York, 1987, pp. 394–405.