1991

An object-oriented framework for computer network simulations

Bruce R. Varnerin
Lehigh University

AN OBJECT-ORIENTED FRAMEWORK FOR COMPUTER NETWORK SIMULATIONS

by
Bruce R. Varnerin

A thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Computer Science

Lehigh University
(April, 1991)

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

4/25/91
Date

Edwin J. Kay
Advisor in Charge

Lawrence J Varnoni
CSEE Department Chairperson

ACKNOWLEDGEMENTS

There are many people deserving thanks for their support while completing this thesis. Above all others, my wife, Debbie, deserves thanks for her understanding and encouragement. Without her, this thesis would not have been possible.

The guidance offered by Professor Edwin Kay was instrumental in bringing my abstract ideas into a cohesive project. I thank him also for making time for me in his hectic schedule.

I should also mention the help Pam Woodbury provided by sharing the files that implemented the Smalltalk-80 simulation framework. This enabled me to implement and experiment with the network simulation framework much more quickly than I could have otherwise.

CONTENTS

LIST OF FIGURES

I. Abstract

Today's local area networks are expanding and diversifying at an alarming rate. Many vendors offer network traffic monitoring applications to assist the network manager in detecting network faults. Some network simulation packages have been made available to provide pro-active network management. These approaches have proved to be too restrictive and cumbersome for network managers in the field.

This thesis proposes an innovative approach to the problem of network simulation - an approach that is both flexible and easy to implement. At the core of this framework is an object-oriented programming environment (Smalltalk-80) and a set of predefined network objects. All that is left to the network manager is to specify host-to-host traffic and what statistics are to be collected. At the same time, the network manager is encouraged to adapt the existing network objects to his own environment (implementing new protocols, adding random errors, etc).

The objectives of this thesis are (1) to present the Smalltalk-80 environment, with emphasis on its simulation framework, (2) to present the newly developed network simulation framework, and (3) to demonstrate how this framework would be used. The framework has been implemented in Objectworks\Smalltalk on a Sun Microsystems SparcStation 1+ workstation. Results of a simple study on local area network expansion are also presented in this thesis.

II. Introduction

Queuing network analysis has provided the primary framework for determining network delay behavior. Since accurate representation of a network protocol requires a queue at every control point (flow control, media access, error checking, etc.), this method quickly falls short for even the most trivial real world examples [BeG87]. Simplifying assumptions can be made to reduce the number of queues at the expense of sacrificing accuracy and usefulness.

Today's network analysts must apply new techniques. A combination of simulation, queuing network analysis and heuristics, is the only approach that can make sense of the complexity in today's networks. And, indeed, the literature is flooded with various implementations of this theme, broadly classified as "network simulations" regardless of the analytic content. For the most part, these simulations have done a thorough job of modeling the networking protocols and computing environments around which they were designed.

The network simulation tools reported to date fall into two categories: those that present a procedural programming interface [ASA85,DeP85,KiL87,JaJ87] and those that present a "user-friendly" interface [Bac87,ZTD88]. The programmatic approaches offer infinite flexibility to the user adept at programming, simulation, queuing theory, and networking protocols (and infinite complexity to those who are not). The "user-friendly" approaches allow the user to select predefined protocols, topology, traffic patterns, and statistics (but afford little flexibility for the adventuresome user). One approach worth noting [HeM88] offers a "user-friendly" interface, but relies on the user to develop the queuing network model for the system under analysis.

These techniques have failed to gain strong popularity in the network management community owing to a familiar problem: network managers are not usually simulation experts, but they need a tool with enough flexibility to be adaptable to their environment. (That is, they want some flexibility, but not at the expense of usability.)

Object-oriented programming and user interfaces are gaining increasing popularity in many applications. Simulation model development has not yet taken advantage of the many benefits offered [Coo86]. Several of these advantages look particularly promising for network simulation. Objects are completely independent of each other (just as network nodes are). Objects communicate with each other via messages (the analogy here is obvious). And finally, inheritance allows for expandability and adaptability to almost any environment. At least one leading network architecture is actually specified in object-oriented terminology [ZDL90].

The potential for object-oriented programming in the network management arena is best espoused by Hiebert:

> *Mapping and representing networks via object-oriented definition principles and employing inheritance and method/data-encapsulation concepts could simplify and reduce overhead in real-world networks. The benefits of associating the methods and properties with the proper class of objects may also reduce the software maintenance costs in network-management and -planning software. [Hie88]*

This paper will present the high-level design of a network simulation framework based on the Smalltalk-80 object-oriented programming environment. This method takes advantage of the general benefits of object-oriented programming as well as several specific advantages offered by the Smalltalk-80 environment (easy to execute simulation constructs, a windowed interface, and the ability to customize the entire environment to a particular application) [GoR83].

The framework introduced treats all network entities as independent objects. The network itself, its nodes, and its messages are separate instances of Smalltalk-80 classes, with their own memory and protocol (behavior). Simulation control and all global statistics gathering are performed by an instance of yet another class.

The network manager interacts with this framework through the standard Smalltalk-80 programming environment. New object classes are defined for each network and network node. Setting network parameters and gathering statistics are programmed by the user using the utilities provided by the framework. The inherent flexibility of Smalltalk-80 allows the more adventuresome user to make changes to any part of the framework. Defining new object behaviors, changing statistics gathering, and changing network behavior are all possible within the framework.

This paper will first present an overview of the Smalltalk-80 programming environment, concentrating on its simulation tools (Section II). I present the newly developed extension of these simulation tools that make up the network simulation framework in Section III. In Section IV, this framework is applied to a simple example of network reconfiguration. The paper is concluded with Section V, with an emphasis on identifying areas for future work in developing this framework.

III. The Smalltalk-80 Environment

Object-oriented programming languages have been developed around three underlying ideas: data/program encapsulation, message passing between objects, and object inheritance [Coo86]. Encapsulating the data with the program modules that act on it effectively creates "smart data" (an *object*) [Par90]. The programmer does not need to know the underlying details of the data, just how it responds to other objects. These responses are elicited by sending the object a *message*. Inheritance is used by the programmer to extend the capabilities of objects, without having to reprogram the object from the ground up.

This object-oriented paradigm is implemented to varying degrees by the languages available on the commercial market. C++, Simula, Clu, and Ada have all managed to gain some acceptance despite some inadequacies (e.g., inconsistent treatment of primitive objects, limited inheritance capabilities, inadequate development tools, etc) [Coo86].

Smalltalk-80 is more than just an object-oriented programming *language*; it is a completely consistent, object-oriented programming *environment* [GoR83]. This section of the paper will first provide an overview of the Smalltalk-80 language, with the intent of providing enough knowledge to understand the network simulation framework presented in the next section. After the *language* is explained, I will briefly present the *programmer's environment*. The final part of this section will detail the Smalltalk-80 simulation tools described in [GoR83]. It is from these basic simulation tools that the network simulation framework is built.

A. The Smalltalk-80 Language

Because of the consistent implementation of Smalltalk-80, it is only necessary to learn a few basic concepts to understand the language. These concepts include: objects, messages, methods, classes, and instances [GoR83]. I present these ideas below.

1. Objects, Messages, and Methods

An *object* is made up of private variable(s), the data, and a set of operations that act on that data. These two parts of an object are also called its *state* and *behavior*. The value of an object's private variables define its state; its *methods* define its behavior [Par90]. A *message* is sent to an object as a request for it to invoke one of its methods and send a reply back to the sender (another object). [GoR83]

In Smalltalk-80, every conceivable part of the environment, is itself, a Smalltalk-80 object (e.g., numbers, characters, queues, geometric shapes, dictionaries, text editors, user menus, windows, debuggers, etc. are all objects accessible to the programmer). To understand an object is to understand Smalltalk-80.

An object that must maintain multiple pieces of information is called a composite object. Indeed, even an object's private variables are themselves objects [Par90]. However, these private variables are not directly accessible to other objects in the system. They can only be accessed through the composite object's methods.

Messages sent to objects identify a particular method for an object to execute and also supply the necessary arguments. A message is a two-way pipe, however, since the message receiver must return some object to the sender. If no object is explicitly returned by the method, the receiver itself is returned. Whether the returned object is used is completely up to the sender. The set of messages to which an object responds is called its interface, or protocol, with the rest of the system [GoR83].

Messages may be *unary*, *keyword* or *binary*. A unary message takes no arguments (e.g., "clock time" represents the message "time" being sent to the object "clock"). A keyword message takes one or more arguments (e.g., "clock setHour: 10" is the message "setHour:" sent to "clock" with the argument "10"; "clock setHour: 10 andMinutes: 30" is the message "setHour:andMinutes:" sent to "clock" with the arguments "10" and "30"). A binary message, specified by either one or two symbols (e.g., +,--, etc.) takes a single argument (e.g., "3 + 4" is the message "+" sent to "3" with the argument "4"). [GoR83]

The methods that define how an object responds to messages are grouped into *message categories*. The standard category names used by the predefined Smalltalk-80 objects allow easy access for copying code, or just to learn how an object functions.

2. Classes and Instances

Since many programming objects share the same properties (e.g., the integer "3" and the integer "4" should have the same methods in order for arithmetic to be consistent), it would be useful to be able to do this without having to reprogram each method for each object. This is done by separating an object's state from its behavior. The data-only object is known as an *instance*; the method-holding object is known as a *class*. A class can be viewed as a *template* for the object's data affixed to its behavior. [Par90]

A class is an object in and of itself. It responds to messages through its *class methods* (as opposed to the *instance methods*), and may contain its own memory as *class variables*. Class methods are used to create new instances and to initialize the class. Class variables serve as global variables, accessible to all instances of a class, as well as the class.

Examples of Smalltalk-80 classes include Integer, Float, Array, Dictionary, Rectangle, Stream, File (and many more). Examples of instances are "4" (instance of class Integer), "3.1415" (instance of Float), "#(1 2 3 4 5)" (instance of Array), etc.

3. Inheritance

Classes allow the Smalltalk-80 programmer to share methods and instance variable templates among *identical* types of objects. Inheritance, or subclassing, allows the Smalltalk-80 programmer to share instance variable templates and methods among *mostly similar* types of objects. [GoR83]

Consider a small integer object versus a standard integer object (a small integer occupies less memory than a standard integer). Aside from the range of values that can be represented, these two objects are identical. Or, consider two collections of objects: the first, an ordered collection, maintains the order in which objects are added; the second, a sorted collection, sorts objects according to some algorithm. Aside from the method to add an object to the collection, these two objects are almost identical. To define a new class just to provide these differences would be wasteful.

Smalltalk-80 allows the programmer to define a new class as a *subclass* of an existing class. In so doing, all the instance variable declarations and methods become accessible to the subclass transparently. If a *superclass's* method needs to be modified (e.g., the method to add an element in the class SortedCollection), the programmer simply overwrites the superclass method by redefining it for the subclass.

This ability to add functionality incrementally is the key to Smalltalk-80's flexibility. A novice user merely makes use of existing object classes, while the expert programmer is free to create new ones. Smalltalk-80 is so open to change and expandability, that the programmer is free to change all but the most primitive classes in the system.

Inheritance is the key to turning the network simulation framework into a functioning simulation. The simulation user (programmer) is asked to define subclasses to describe the network and its nodes.

4. Syntactical/Notational Conventions

Smalltalk-80 has a simple syntax, although unusual by procedural programming standards. Since I will need to make use not only of Smalltalk-80 code, but of several diagrams to explain the network simulation framework, I will present the Smalltalk-80 syntax and introduce some notational conventions designed to make this thesis more readable.

To distinguish Smalltalk-80 code from the rest of the text, I will present it in the Helvetica font. Comments within the Smalltalk-80 code are delimited with double quotation marks. Most Smalltalk-80 code will also be separated from the text by spacing. When Smalltalk-80 objects (classes or instances) are mentioned in the passing, they will not be physically separated, but will appear in the Helvetica font. When talking about a Smalltalk-80 class, I will explicitly use the word *class*. All other references to Smalltalk-80 objects can be assumed to refer to an instance. Method names and messages mentioned in the text will appear in **Helvetica bold** font.

a. Smalltalk-80 Expression Syntax

As mentioned above, Smalltalk-80 syntax is pretty simple to understand, but it is different from what most programmers are used to. Smalltalk-80 code consists of four expression types: literals (constants), variable names (current value of a variable), message expressions (invoking a receiver's methods), and block expressions (for deferred actions and control structures). This section that describes Smalltalk-80's expression syntax is taken almost exclusively from [GoR83].

Smalltalk-80 literals are expressed in a fashion familiar to most programmers:

- Numbers are a sequence of an optional radix, an optional minus sign, digits, an optional decimal point, digits and an optional exponent (e.g., 3.1415, 8r777, 6.03e23).

- Character constants are prefixed by the dollar sign (e.g., $a, $b, $d).

- Strings are enclosed in single quotes (e.g., 'Hello').

- Symbols a prefixed with a pound sign (e.g., #bill).

- Arrays are delimited with parentheses and preceded by a pound sign (e.g., #(1 2 3)).

Variable names may be any string of characters starting with a letter or digit. By convention, class, pool and global variables begin with upper case letters; temporary and instance variables begin with lower case letters. Another convention adopted by Smalltalk-80 programmers is that when variable names consist of two or more concatenated words, the first letter of each embedded word should be capitalized (e.g., MaximumNumberAllowed or numberOfMessagesPending).

A message expression consists of a receiver (an object), a message selector (the name of the method the receiver should invoke) and possibly some arguments. The value of a message expression is the return value of the method invoked (by default the receiver itself if no value explicitly returned). A value is explicitly returned from a method using the "^" operator (e.g., ^true returns the boolean true value). The following are all message expressions:

message expression	receiver	message	argument(s)
3 + 4	3	+	4
theta sin	theta	sin	
anArray at: index	anArray	at:	index
inventory at: #apples put: 1000	inventory	at:put:	#apples,1000

A message expression's return value can be assigned to a variable (using the ":=" assignment operator) or used in more complex expressions (use parentheses if unsure of operator precedence).

Block expressions (delimited by square brackets) are primarily used within control structures. A block represents a series of actions to be performed later (i.e., during execution of the control structure). Within control structures blocks are used either as arguments of a message or as the receiver of a message. The following code fragments execute familiar control structures:

```
"if-then-else" (counterVariable = 0)
                    ifTrue: [Transcript show: 'counter is zero']
                    ifFalse: [Transcript show: 'counter is nonzero']

"for loop"     (1 to: 10) do: [ :index |Transcript show: 'index = '.
                    index printOn: Transcript]

"while loop"   [self anyMessagesToSend]
                    whileTrue: [self sendMessage]
```

b. Smalltalk-80 Class/Method Syntax

[GoR83] mentions two means of describing a class in print. The first, called a *protocol description*, gives a functional description of the messages in the instances' message interface. The second, an *implementation description*, shows how this functionality would be implemented in the system.

A protocol description lists each message that an instance responds to, giving a comment about its functionality. No mention is made of how it is implemented. This approach provides a *black-box* approach that is appropriate to object-oriented programming. This is the approach that will be taken in the text of this thesis.

Implementation descriptions show the private instance variables and the Smalltalk-80 code for each message that an instance will respond to. Implementation descriptions are provided in the appendices.

c. Pictorial Representations

Diagrams will be used extensively to describe the network simulation framework. These diagrams will maintain the conventions mentioned above for class names, instance variable names and message selectors (Helvetica font, messages in **bold**).

In addition, when depicting class hierarchy, classes will be shown in boxes. The class name, super class, class variables and instance variables will be listed. Abstract classes will appear within dotted lines; concrete classes are shown in solid. Figure 1 depicts the pictorial conventions for classes.

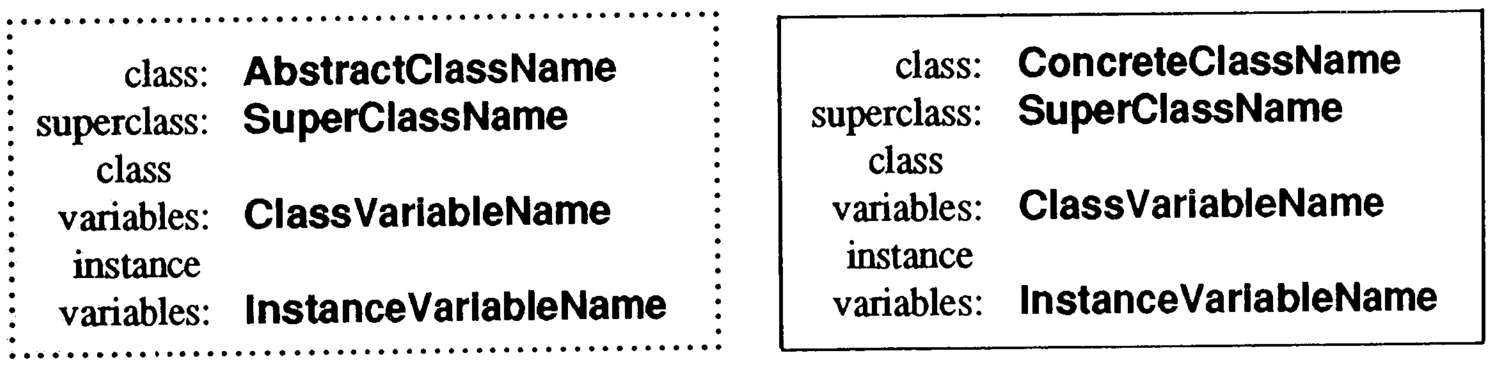

class: AbstractClassName
superclass: SuperClassName
class variables: ClassVariableName
instance variables: InstanceVariableName
class: ConcreteClassName
superclass: SuperClassName
class variables: ClassVariableName
instance variables: InstanceVariableName

Figure 1. Class Notation Examples

It will also be necessary to show the linkage between instances of the classes of the network simulation framework. An instance will be shown with a name indicative of its class and with its instance variables as depicted in Figure 2.

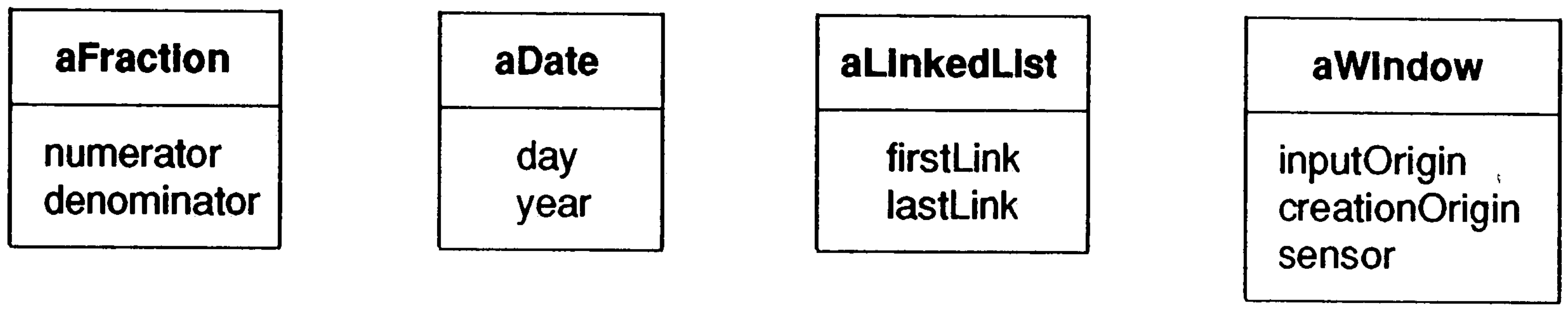


Figure 2. Instance Notation Examples

Finally, some pictorial representation of an instances message interface must be shown. It is useful to see how the various simulation objects interact with each other during the simulation. The object interaction notation will depict objects in solid boxes with a name indicative of its class; messages are shown as solid arrows; return values are shown as dotted arrows. When a message merely returns self (the receiver), the return path is omitted. Objects that are classes will be explicitly labeled (see Figure 3).

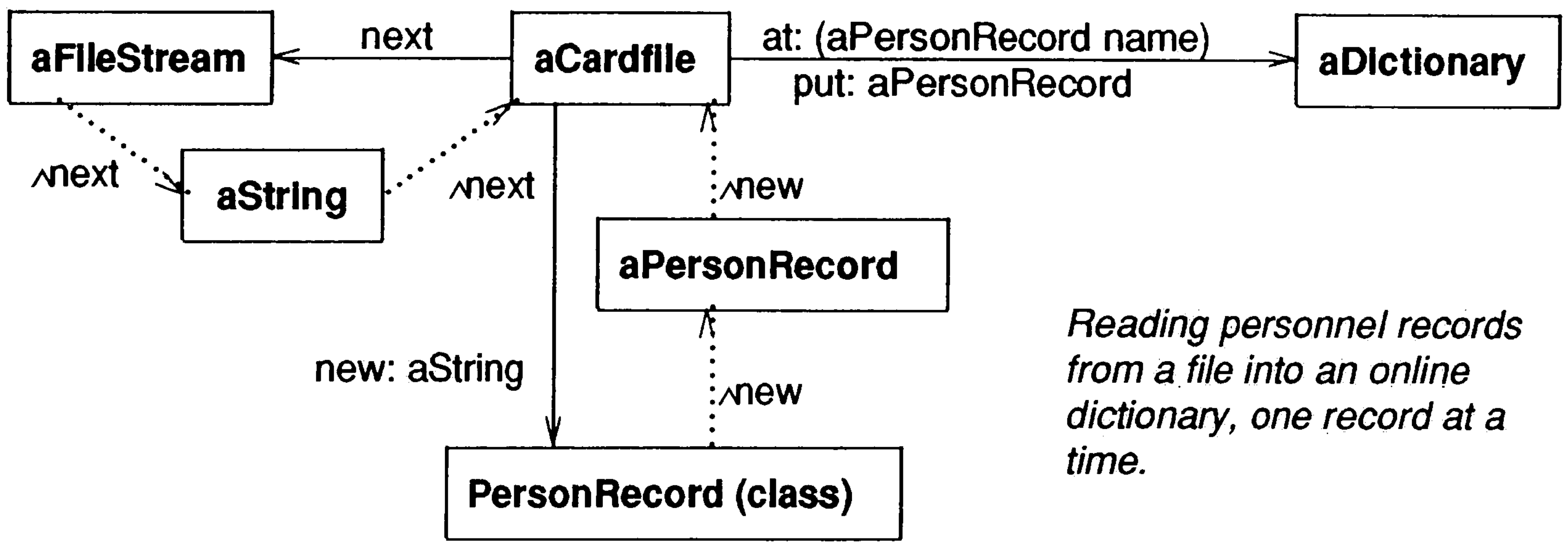


Figure 3. Object Interaction Notation Example

B. Programmer's Interface

As was mentioned earlier in this section, Smalltalk-80 provides more than just an object-oriented programming language; it provides a complete object-oriented programming environment [GoR83]. This interactive environment consists of a workstation, a bit-mapped display, a mouse, and a window-based front-end that functions as a programmer's interface. It is this programmer's interface that sets Smalltalk-80 apart from other object-oriented languages. [Coo86]

The starting point to this interface is the Launcher, a main menu for creating windows, saving changes, interacting with the file system, etc. The programmer's interface is itself programmed in Smalltalk-80. The compiler, the source code editor, the debugger, etc. are all Smalltalk-80 objects whose methods are directly accessible to the Smalltalk-80 programmer. This gives the Smalltalk-80 programmer the uncommon ability to change the programming language itself.

Smalltalk-80 provides three utilities that will have to be used by the implementor of the network simulation framework. A *browser* allows one to browse or edit the source code of all classes in the Smalltalk-80 system. An *inspector* allows one to view the state of any instance of a class. A *debugger* is a combination of a browser and an inspector that appears whenever errors are encountered in the Smalltalk-80 code.

Browsers, inspectors, and debuggers are presented in detail in [GoR83] and [Par90]. An overview of these three utilities along with an important programming approach called the model-view-controller paradigm are presented in the following subsections.

1. Browsers

A browser window is the primary means to view, add, and edit Smalltalk-80 code. It consists of a window with 5 subwindows, the largest of which is a source code editor. The smaller subwindows allow the user to select categories of classes (e.g., "Collections-Text"), individual classes (e.g., String), categories of methods (e.g., "comparing"), and individual methods (e.g., **match:**), to view or edit.

The *system browser* makes all classes in the system available to the user. Other browsers view a given category of classes, classes that send or implement the same message, or just one particular class.

A browser may be created from the Launcher or by sending an object the message **browse**.

2. Inspectors

An inspector shows the current state of any active object. It is a two column window showing each instance variable of the object and its corresponding value. An inspector can be used as a debugging aid, as a "what-if" tool (you can change a value with an inspector), or simply as an output mechanism. Inspectors are created from several other utilities (e.g., debuggers and workspaces), or can be invoked simply by sending any object the message **inspect**.

3. Debuggers

A debugger can be invoked after an error is encountered as a method is run. A debugger presents four subwindows. At the top is a trace of the operations that were executed just before the error was encountered. In the middle is a source code editor that shows the offending piece of code. At the bottom are two inspectors, one for instance variables, and one for temporary variables.

The debugger allows the programmer to step through the events leading up to the error. At any point, code can be modified in the source code editor and saved. The debugger allows you to single-step, resume execution or quit.

4. Model-View-Controller Paradigm

The Model-View-Controller (MVC) paradigm is a programming methodology designed to modularize the task of developing user interfaces to Smalltalk-80 programs [Par90]. In this approach, the user interface (the view) is developed as a separate Smalltalk-80 object from the main program object (the model). The two are linked by a third object, called the controller.

The network simulation framework presented in this thesis provides a means for systematically developing the *model* object in the MVC triad. An approach to developing a user-friendly interface (i.e., the *view* and the *controller*) will be mentioned, but implementation details are not provided.

C. Simulation Tools

[GoR83] proposes a Smalltalk-80 framework for developing event-driven simulations. Since the network simulation framework is a direct extension of these utilities, a thorough explanation will be presented.

An event-driven simulation is one in which the simulated clock is advanced to the point at which the next event occurs (rather than a fixed time increment) [GoR83]. This implies that an event must be scheduled beforehand to occur (i.e., everything must be expected). A queue of events, ordered by the (simulated) time at which they are to occur, must be maintained to implement this approach.

In many real-world systems, objects enter and leave independently (e.g., customers in a store). The may produce or consume resources (e.g., paying cash and buying merchandise, respectively). They may also need to coordinate their activities with each other (e.g., two customers may want the same piece of merchandise, or may require help from a salesperson). [GoR83]

The simulation framework in [GoR83] provides three classes (or class hierarchies) for implementing simulations. The Simulation class describes instances that control the entire simulation: maintaining the event queue, handling the overhead of objects entering and leaving; storing and coordinating resources. Class SimulationObject (and its subclass, EventMonitor) provide the framework for describing the individual objects to be simulated. Class Resource (and its subclass ResourceCoordinator), provide a mechanism for producing, storing, consuming, and releasing resources.

1. ProbabilityDistribution Class Hierarchy

Before getting into the simulation framework of [GoR83], I present the ProbabilityDistribution class hierarchy that is frequently needed in the simulation framework. Probability distribution functions are widely used to model random events such as the frequency with which customers enter a store, the probability that two customers might arrive at the same time, or the time it takes to service individual customers [GoR83]. In performing network analysis, probability distributions are used to model message interarrival times, message lengths, queue times, transmission times, probable destinations, etc [BeG87].

[GoR83] presents both discrete and continuous probability distribution functions, represented as abstract subclasses (DiscreteProbability and ContinuousProbability) of an abstract superclass (ProbabilityDistribution). It is intended that subclasses of DiscreteProbability and ContinuousProbability will implement the **density: x** method that is used to compute the next sampled value (returned by the **next** method).

Also presented in [GoR83] are concrete subclasses to model the various distribution functions needed by simulation programmers. The discrete probability distribution classes SampleSpace, Bernoulli, Binomial, Geometric, Poisson, as well as the continuous probability classes, Uniform, Exponential, Gamma, and Normal are all predefined and available for use. [GoR83]

In the network simulation framework, we will need to use the SampleSpace, Uniform, and Exponential classes to simulate picking a random destination for a message, message length distributions, and message interarrival times, respectively.

The discussion here will be limited to creating instances of these three classes for use in the network simulation framework.

A SampleSpace consists of a set of all possible values. The **next** method answers one of these values, selected at random each with an equal probability of success. A SampleSpace is created by simply sending an array of all possible values as the argument of the **data:** method of the SampleSpace class. For example, the following code fragment creates a SampleSpace for the possible outcomes of rolling a die:

```
aSampleSpace := SampleSpace data: #(1 2 3 4 5 6).
```

A Uniform probability distribution function is the continuous equivalent to a SampleSpace. Instead of choosing one value from all possible values each with equal probability, you choose a value within a range of values where each possible value has an equal probability. The value returned from the **next** method is a Float. The following code fragment shows how to create a Uniform distribution function:

```
aUniformDistribution := Uniform from: 1 to: 100.
```

Rather than answering the question "What is the next value ?" Exponential distributions determine how long it will be until the next event occurs. This is obviously an important distribution for event-driven simulations, as we will see. Again, Exponential is a continuous distribution and returns the time until the next event when sent the message **next**. The Exponential distribution is used most often when the likelihood of an event not occurring decreases with time (or, when the likelihood that an event *does* occur *increases* with time)[GoR83]. This is precisely what happens as messages arrive in a network [HeM88]. An Exponential distribution is created by specifying the average number of events per unit time:

```
anExponentialDistribution := Exponential mean: 0.1.
"0.1 events per unit time = 1 event every 10 units of time"
```

2. Simulation Class

Class Simulation manages the objects in the simulated system and schedules events to occur according to the simulated clock [GoR83]. Instances of class Simulation contain the event queue, the simulated clock, a set of resources and a count of the active processes.

Methods are provided for starting, finishing, and initializing the simulation, producing and acquiring resources, inquiring about resources, and scheduling future actions. Several methods are left to the developer to implement in subclasses (e.g., **exit:**, **enter:**, **defineArrivalSchedule**, **defineResources**). Simulation is therefore an abstract class.

An instance of class Simulation mediates among objects and resources. When an object is scheduled to enter the system, it is placed into the event queue. When it produces resources, they are stored in the resources instance variable. When they need to acquire resources, they must ask the class Simulation instance to do so. It is only by having all scheduling and resource interactions go through the class Simulation instance that the simulation can be controlled.

Simulation instances use the class DelayedEvent to store events in the eventQueue. A DelayedEvent represents a process paused until its resumptionCondition (time to resume) has been reached. Class DelayedEvent does not have to be accessed directly by the simulation programmer. Therefore, its protocol description will not be shown below.

The [GoR83] protocol description of class Simulation is given below. A full implementation description (including the DelayedEvent class) may be found in Appendix A.

Simulation instance protocol

initialization	
initialize	Initialize the receiver's instance variables.
modeler's initialization language	
defineArrivalSchedule	Schedule simulation objects to enter the simulation at specified intervals of time, typically based on probability distribution functions. This method is implemented in subclasses. It involves a sequence of messages to the receiver (i.e., self) that are of the form **schedule:at:, scheduleArrivalOf:at:, scheduleArrivalOf:accordingTo:, or scheduleArrivalOf:accordingTo:startingAt:**
defineResources	Specify the resources that are initially entered into the simulation. These typically act as resources to be acquired. This method is implemented by subclasses and involves a sequence of messages to the receiver (i.e., to self) of the form **produce: amount of: resourceName**.

modeler's task language

produce: amount of: resourceName	An additional quantity of amount of a resource referred to by the String, resourceName, is to be part of the receiver. If the resource does not as yet exist in the receiver, add it; if it already exists, increase its available quantity.
coordinate: resourceName	Use of a resource referred to by the String, resourceName, is to be coordinated by the receiver.
schedule: actionBlock after: timeDelayInteger	Set up a program, actionBlock, that will be evaluated after a simulated amount of time, timeDelayInteger, passes.
schedule: actionBlock at: timeInteger	Schedule the sequence of actions (actionBlock) to occur at a particular simulated time, timeInteger.
scheduleArrivalOf: aSimulationObject at: timeInteger	Schedule the simulation object, aSimulationObject, to enter the simulation at a specified time, timeInteger.
scheduleArrivalOf: aSimulationObjectClass accordingTo: aProbabilityDistribution	Schedule simulation objects that are instances of aSimulationObjectClass to enter the simulation at specified time intervals, based on the probability distribution, aProbabilityDistribution. The first such instance should be scheduled to enter now.
scheduleArrivalOf: aSimulationObjectClass accordingTo: aProbabilityDistribution startingAt: timeInteger	Schedule simulation objects that are instances of aSimulationObjectClass to enter the simulation at specified time intervals, based on the probability distribution, aProbabilityDistribution. The first such instance should be scheduled to enter at time timeInteger.

accessing

includesResourceFor: resourceName	Answer if the receiver has a resource that is referred to by a String, resourceName. If such a resource does not exist, then report an error.
provideResourceFor: resourceName	Answer a resource that is referred to by the String, resourceName.
time	Answer the receiver's current time.

simulation control

startUp	Specify the initial simulation objects and the arrival of new objects.
proceed	This is a single event execution. The first event in the queue, if any, is removed, time is updated to the time of the event, and the event is started.
finishUp	Release references to any remaining simulation objects.
enter: anObject	The argument, anObject, is informing the receiver that it is entering. This is a "do nothing" method that should be implemented by subclasses.
exit: anObject	The argument, anObject, is informing the receiver that it is exiting. This is a "do nothing" method that should be implemented by subclasses.

3. SimulationObject and EventMonitor Classes

[GoR83] proposes an abstract class, SimulationObject, to describe objects within a simulation that have tasks to perform. The methods provided by this class allow the object to interact with the Simulation instance. Subclasses of SimulationObject are required to implement the **initialize** and **tasks** methods.

An abstract subclass of the SimulationObject class, EventMonitor, is also presented in [GoR83]. This class is functionally equivalent to the SimulationObject class and still leaves the **initialize** and **tasks** methods unimplemented. EventMonitor automatically logs to the file, DataFile (a class variable) every event that the object performs. Since SimulationObject and EventMonitor present the same message interface, SimulationObject will be used hence forward to refer to both classes.

A SimulationObject enters the simulation, initializes itself, performs its tasks and then exits. As a SimulationObject is carrying out its tasks, it can produce or consume resources, hold for periods of time, reschedule itself, etc. These actions, although sent as messages to the SimulationObject itself (to "self"), are actually carried out by the Simulation instance.

The protocol description for both SimulationObject and EventMonitor follow. Full implementation descriptions appear in Appendix A. The use of these classes will become clearer as the network simulation framework is unveiled. EventMonitor becomes the superclass of all subclasses used to simulate the network, network nodes, and messages on the network.

SimulationObject and EventMonitor instance protocol

---

initialization	
initialize	Initialize instance variables, if any.

simulation control

startUp	Initialize instance variables. Inform the simulation that the receiver is entering it, and then start the receiver's tasks.
tasks	Define the sequence of activities that the receiver must carry out.
finishUp	The receiver's tasks are completed. Inform the simulation.

task language

holdFor: aTimeDelay	Delay carrying out the receiver's next task until aTimeDelay amount of simulated time has passed.
acquire: amount ofResource: resourceName	Ask the simulation to provide a simple resource that is referred to by the String, resourceName. If one exists, ask it to give the receiver amount of resources. If one does not exist, notify the simulation user (programmer) that an error has occurred.
acquire: amount of: resourceName withPriority: priorityNumber	Ask the simulation to provide a simple resource that is referred to by the String, resourceName. If one exists, ask it to give the receiver amount of resources, taking into account that the priority for acquiring the resource is to be set to priorityNumber. If one does not exist, notify the simulation user (programmer) that an error has occurred.
produce: amount ofResource: resourceName	Ask the simulation to provide a simple resource that is referred to by the String, resourceName. If one exists, add to it amount more of its resources. If one does not exist, create it.
release: aStaticResource	The receiver has been using the resource referred to by the argument, aStaticResource. It is no

	longer needed and can be recycled.
inquireFor: amount ofResource: resourceName	Answer whether the simulation has at least a quantity, amount, of a resource referred to by the String, resourceName.
resourceAvailable: resourceName	Answer whether the simulation has a resource referred to by the String, resourceName.
acquireResource: resourceName	Ask the simulation to provide a resource simulation object referred to by the String, resourceName. If one exists, ask it to give the receiver its services. If one does not exist, notify the simulation user (programmer) that an error has occurred.
produceResource: resourceName	Have the receiver act as a resource that is referred to by the String, resourceName. Wait for another SimulationObject that provides service to (acquires) this resource.
resume: anEvent	The receiver has been giving service to the resource referred to by the argment, anEvent. The service is completed so that the resource, a SimulationObject, can continue its tasks.
numberOfProvidersOfResource: resourceName	Answer the number of SimulationObjects waiting to coordinate its tasks by acting as the resource referred to by the String, resourceName.
numberOfRequestersOfResource: resourceName	Answer the number of SimulationObjects waiting to coordinate its tasks by acquiring the resource referred to by the String, resourceName.
stopSimulation	Tell the simulation in which the receiver is running to stop. All scheduled events are removed

and nothing more can happen in the simulation.

4. Resource and ResourceCoordinator Classes

A framework consisting of the classes Resource, ResourceProvider, StaticResource and ResourceCoordinator are proposed by [GoR83] to manage resources in event-driven simulations. Instances of the Resource class maintain a queue of requests for the named resource. It accesses the active simulation for timing and process management. Its subclasses, ResourceCoordinator and ResourceProvider, provide complete access and control of static and fluctuating resources.

In a system, fixed resources can be consumable (e.g., food) or nonconsumable (e.g., a plate). Fluctuating resources are either renewable or producer/consumer coordinated. Using a typical restaurant as a model will help visualize coordinated resources. Customers enter and line up waiting to be seated. They must wait until a waitress arrives to seat and serve them before they can continue with the rest of their day. On busy days, the customer line grows long, as there are more customers than waitresses. On slow days, the only line is that of the waitresses waiting for customers. Since the customer needs the waitress to be able to eat, and the waitress needs the customer to earn a living, these resources are said to be coordinated.

In the network simulation framework, the network and the messages on the network are considered resources. The network nodes are the *servers* of the system. All these objects need to be managed as coordinated resources, since they are all independent SimulationObjects with other tasks to perform (i.e., statistics gathering). So, the discussion of the Resource class hierarchy will be limited to just the superclass Resource and the subclass ResourceCoordinator.

The Resource class provides the basic mechanism for handling resource requests: an instance variable that contains the resource name (resourceName), an instance variable that maintains a request queue (pending), and methods that manipulate these instance variables. Subclass ResourceCoordinator implements the functionality for satisfying coordinated resource requests. It must also be able to distinguish between three conditions: either there are customers waiting for servers, servers waiting for customers, or no one waiting at all. A ResourceCoordinator instance will answer an object that points to the next server or customer in the pending request queue when asked to provide resources. The object returned, a WaitingSimulationObject instance, will answer the server or customer (a SimulationObject) when sent the message **resource**.

When using coordinated resources, the pending request queue represents customers or servers waiting for service or to serve. These requests are represented as instances of the DelayedEvent subclass, WaitingSimulationObject. WaitingSimulationObject provides for priority resource requests and puts additional utilities around the DelayedEvent class to handle resources more easily. For instance, the customer or server is held in an instance variable called resource instead of having to be extracted from the DelayedEvent's resumptionCondition variable. As a result, a server that is passed a WaitingSimulationObject need only send the message **resource** to access the object itself. Since this is the only WaitingSimulationObject message relevant to the network simulation framework, this class will be left out of the protocol description that follows.

When a server finishes its service of a coordinated resource, it sends the resource a **resume** message. This accomplishes two tasks. It breaks the linkage to the server, enabling the resource to exit the system without having to worry about garbage collection. Also, it resumes the SimulationObject at the point in its **tasks** method that it requested service.

The protocol for classes Resource and ResourceCoordinator (interpolated from [GoR83]) is shown below. As with the rest of the Smalltalk-80 simulation framework classes, the implementation descriptions (including class WaitingSimulationObject) are found in Appendix A.

Resource class protocol

---

class initialization

activeSimulation: existingSimulation	Set the currently active simulation instance to an existingSimulation.

instance creation

named: resourceName	Answer a new instance of the receiver with its instance variable, resourceName initialized to resourceName.

Resource instance protocol

accessing

addRequest: aDelayedEvent	Add a request for resources, aDelayedEvent, to the pending request queue.
name	Answer the resourceName.

private

provideResources	Answer the receiver itself.
setName: aString	Set the receiver's resourceName to the String, aString, and initialize the pending request queue.

ResourceCoordinator instance protocol

accessing

customersWaiting	Answer whether there are customers waiting in the receiver's pending request queue.
serversWaiting	Answer whether there are servers waiting in the receiver's pending request queue.
queueLength	Answer the number of requests in the receiver's pending request queue.

task language

acquire	Answer a WaitingSimulationObject (a customer) to serve from the receiver's pending request queue if any exist. If not, pause the receiver and add the WaitingSimulationObject to the pending request queue. When a customer does become available, resume the receiver and return the customer's WaitingSimulationObject.

producedBy: aCustomer	Get service for aCustomer, if a server is available. Otherwise, add the request to the pending request queue.

private

getServiceFor: aCustomerRequest	Send a server the customer request, aCustomerRequest, if a server is available, otherwise answer #none.
giveService	Answer the first customer request in the pending request queue if one exists. Otherwise, answer #none.
setName: aString	Call **super setName** (from Resource class), then initialize whoIsWaiting (the instance variable that keeps track of whether customers or servers are in the pending request queue) to #none.

IV. The Network Simulation Framework

A. Overview

Computer networks, like other real-world systems, consist of independent entities (objects) sending random traffic into a shared system [ZDL90]. This thesis models each network node, its messages, and the network as separate Smalltalk-80 instances and classes. This approach allows maximum flexibility and ease of understanding. This section will explain the network simulation framework in Smalltalk-80 terminology. The tools employed here are direct extensions of the simulation tools proposed by [GoR83] and explained in Section II.C.

I first present a conceptual view of a computer network, taken roughly from [KiL87]. This conceptual model is purposefully vague, to allow extension to a variety of network architectures, but is based inherently on a CSMA/CD (Carrier Sense Multiple Access with Collision Detection) type of network. In this model, messages are generated and enter an outgoing message queue at the source node. A node must sense when the network becomes idle, at which time it can start to serve the message at the head of the queue. It is assumed that the message contains addressing information. The source node does not know how to deliver a message, it can only send it out onto the network. The network, in turn, can pass the message on to the destination node, where it is received. Figure 4 depicts this conceptual network model.

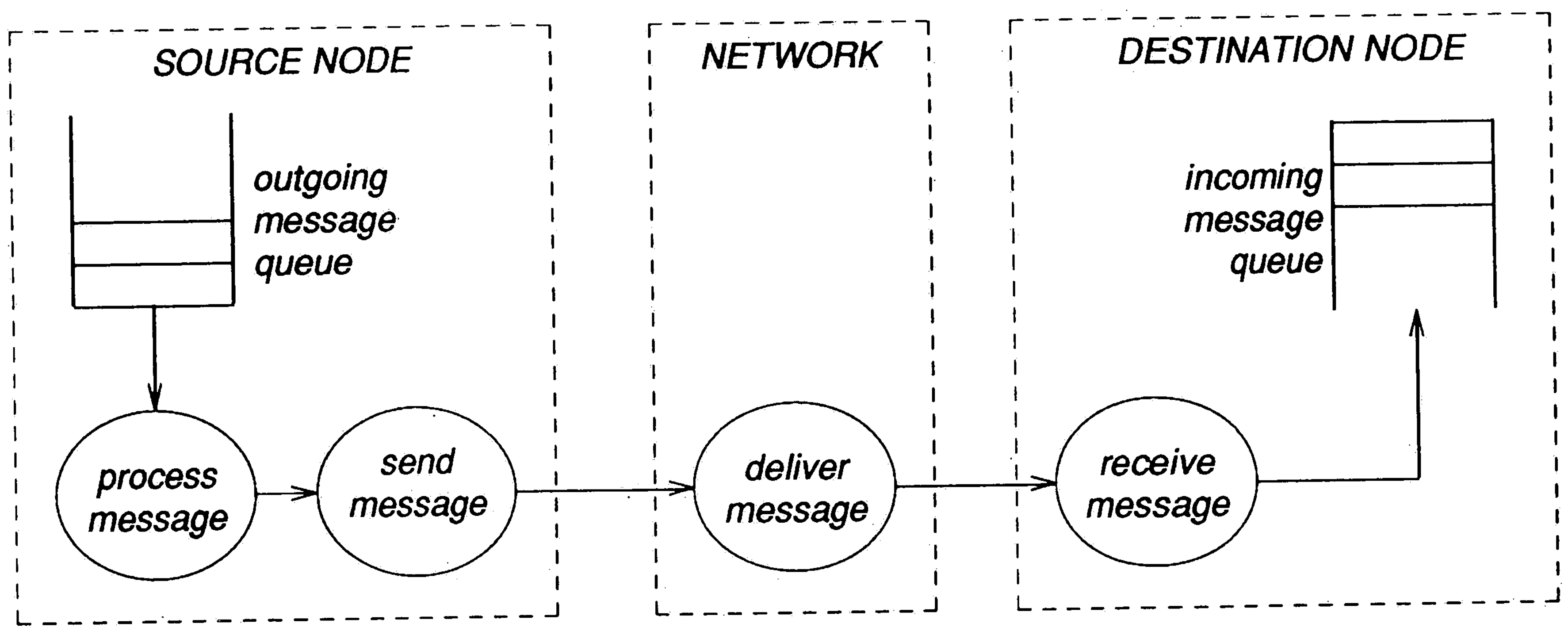


Figure 4. Conceptual Model of a Computer Network

Three types of SimulationObjects implement this conceptual model: NetworkMessages, a Network, and NetworkNodes. The framework defines NetworkMessages and Networks as renewable (coordinated) resources. Defining resources in this way allows messages to enter and leave the simulation in a pseudo-real-time fashion. It also allows for the possibility of the network leaving the simulation for periods of time to simulate network down time. The NetworkNode then becomes the "server" of the system. Since the network nodes are the only active entities in a real-world network, this representation is not only functionally correct, but easy to grasp.

The three objects described above are controlled by the NetworkSimulation object. NetworkSimulation does not represent an object in the conceptual model; it functions as the coordinator of the three "real" SimulationObjects. These four classes make up the network simulation framework. The roles and relationships of these classes are shown in Figure 5.

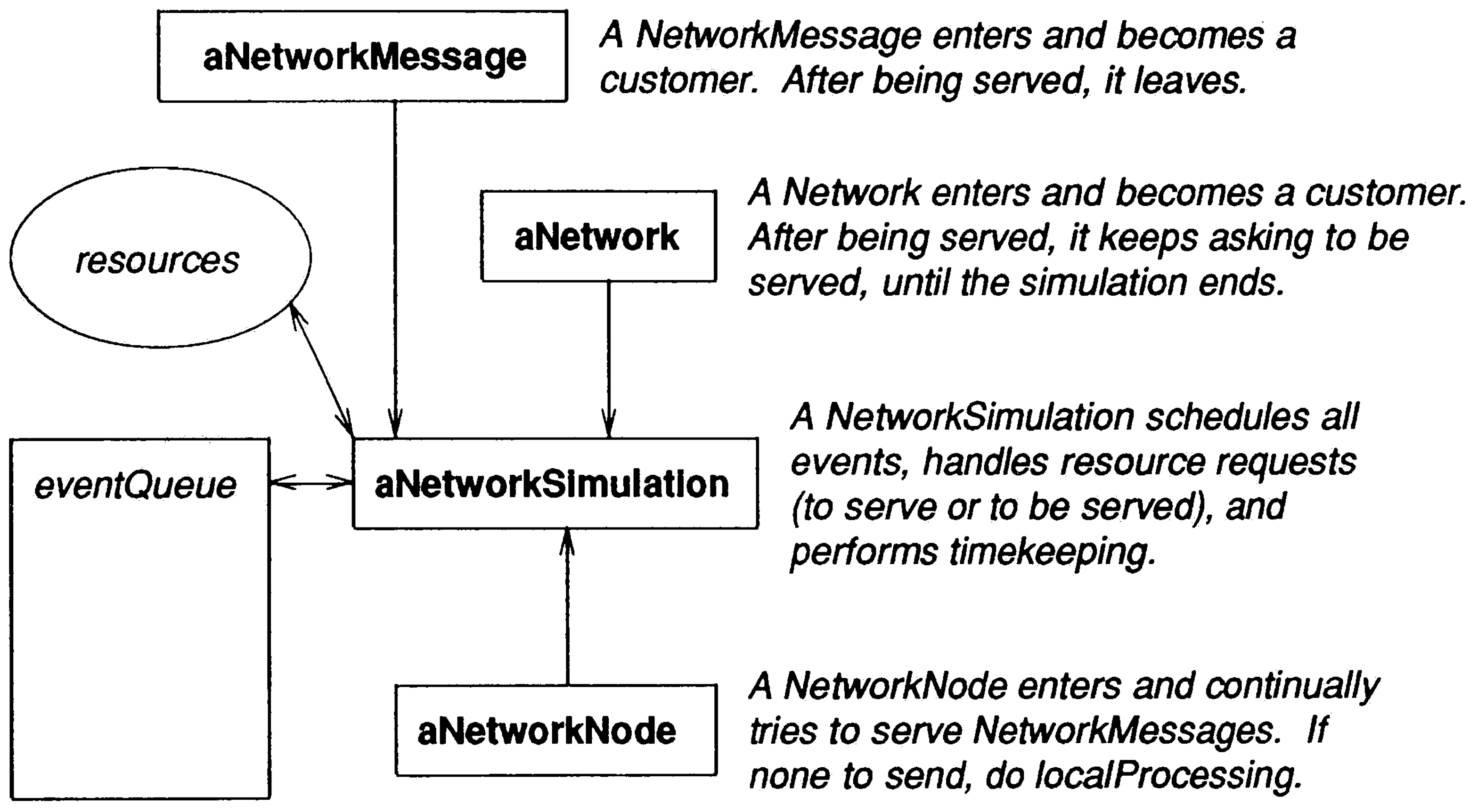

aNetworkMessage
A NetworkMessage enters and becomes a customer. After being served, it leaves.
resources
aNetwork
A Network enters and becomes a customer. After being served, it keeps asking to be served, until the simulation ends.
eventQueue
aNetworkSimulation
A NetworkSimulation schedules all events, handles resource requests (to serve or to be served), and performs timekeeping.
aNetworkNode
A NetworkNode enters and continually tries to serve NetworkMessages. If none to send, do localProcessing.

Figure 5. Roles Within the Network Simulation Framework

1. Architecture of the Model

A subclass of Simulation called NetworkSimulation controls the operation of the network model. It maintains global parameters (stopTime, logfile, etc.), accumulates the overall statistics of the system (the number of simulation objects that have entered, exited, etc.), and maintains a link to the Network instance through its instance variables. An instance creation method of this class also creates the network and network nodes. When adding a user interface (See Section III.D), the NetworkSimulation instance is referenced as the model in the MVC architecture

[GoR83].

As with its superclass, NetworkSimulation performs the bulk of the work in these simulations. It coordinates resources, schedules the arrival/exit of all objects, handles event execution, and performs the timekeeping. NetworkSimulation has already implemented the **enter:**, **exit:**, **defineArrivalSchedule** and **defineResources** methods discussed in Section II.C.2. As with any Smalltalk-80 class, NetworkSimulation can be extended via subclasses, however, it is already complete as defined in the framework.

No two computer networks behave the same. Not only are there a wide variety of network architectures and protocols from which to choose, but each network is affected by the individual characteristics of its nodes. So, the network simulation framework contains only vague descriptions for networks and nodes; two abstract classes, Network and NetworkNode. Subclasses of Network set their own parameters, define their own statistics, calculate their own service times and do their own statistics gathering. Subclasses of NetworkNode define their own statistics, statistics collection, traffic and local processing tasks. The network simulation framework class hierarchy is shown in Figure 6.

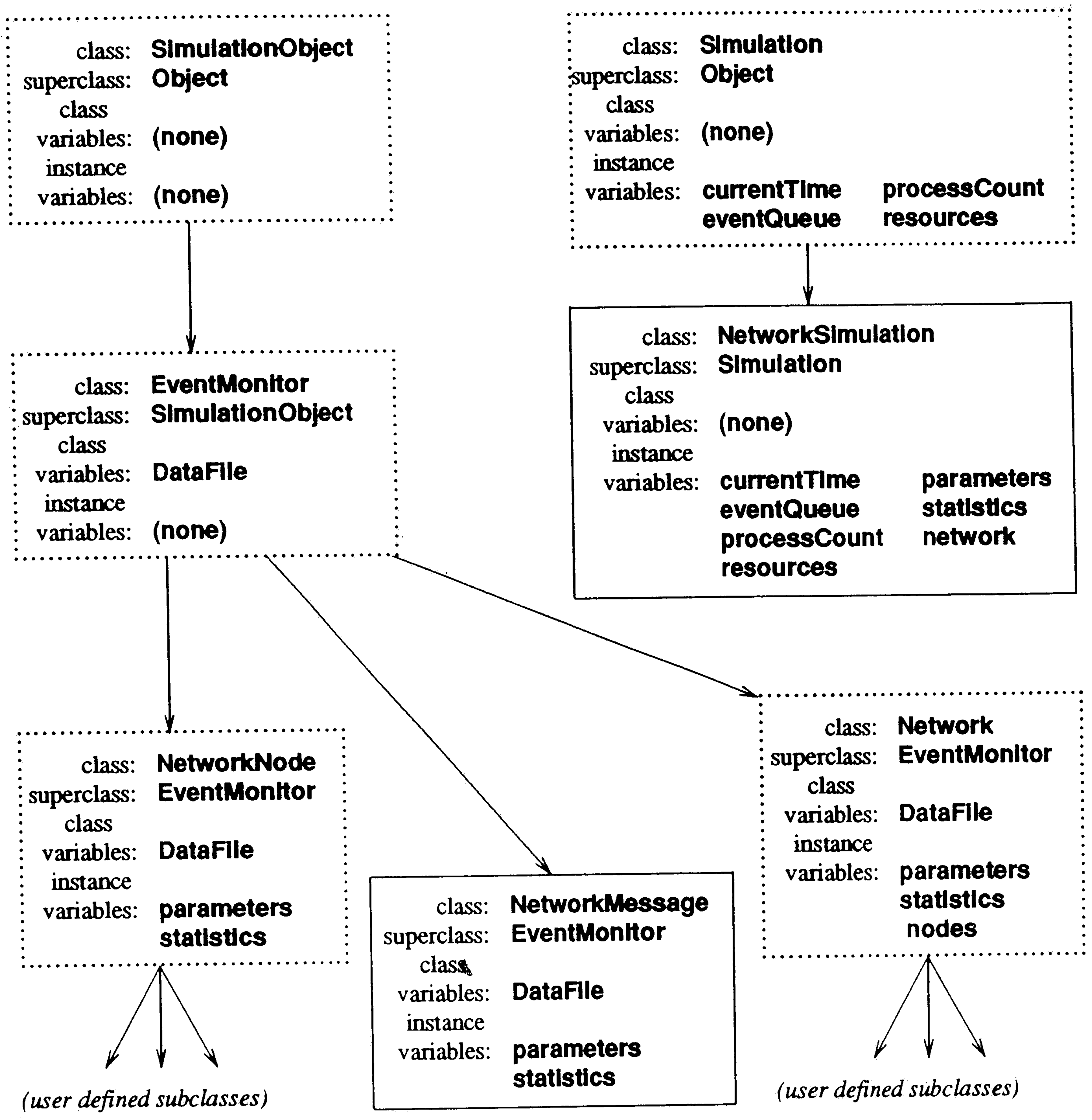

class: SimulationObject
superclass: Object
class variables: (none)
instance variables: (none)
class: Simulation
superclass: Object
class variables: (none)
instance variables: currentTime eventQueue processCount resources
class: NetworkSimulation
superclass: Simulation
class variables: (none)
instance variables: currentTime eventQueue processCount resources parameters statistics network
class: EventMonitor
superclass: SimulationObject
class variables: DataFile
instance variables: (none)
class: NetworkNode
superclass: EventMonitor
class variables: DataFile
instance variables: parameters statistics
class: NetworkMessage
superclass: EventMonitor
class variables: DataFile
instance variables: parameters statistics
class: Network
superclass: EventMonitor
class variables: DataFile
instance variables: parameters statistics nodes
(user defined subclasses)
(user defined subclasses)

Figure 6. Class Hierarchy

The instances created during the simulation are all linked in some way through their instance variables. The NetworkSimulation links to the Network with its network instance variable. Network is, in turn, linked to the NetworkNodes via its nodes instance variable. NetworkMessages enter the simulation as customers, and are therefore linked to the resources instance variable of NetworkSimulation, through instances of ResourceCoordinator. Figure 7 depicts these linkages.

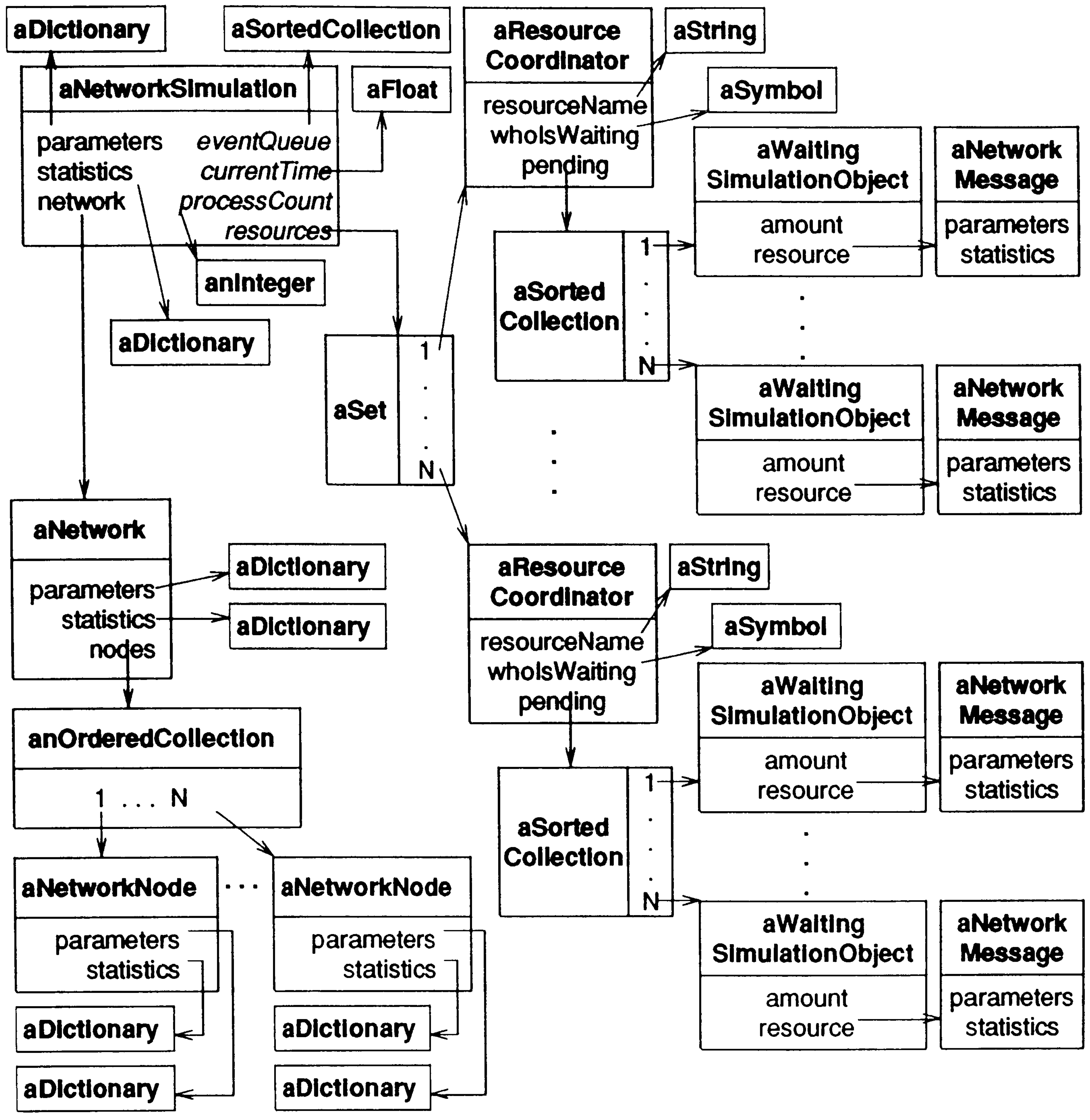


Figure 7. Linking of Instance Variables

An instance of the user-defined NetworkNode subclass enters the simulation at time zero, schedules its traffic, then enters an infinite loop, continuously checking for messages to send. An instance of the user-defined Network subclass also enters at time zero. However, its only function is to continuously renew its own resource (signaling to all nodes that it is idle). The nodes grab the network, hold it for some time, then release it.

NetworkMessage scheduling is requested by each NetworkNode instance. They enter the simulation as renewable resources, linked via the NetworkSimulation resources instance variable. NetworkMessage instances merely enter the simulation asking for service, get serviced, then exit after recording statistics.

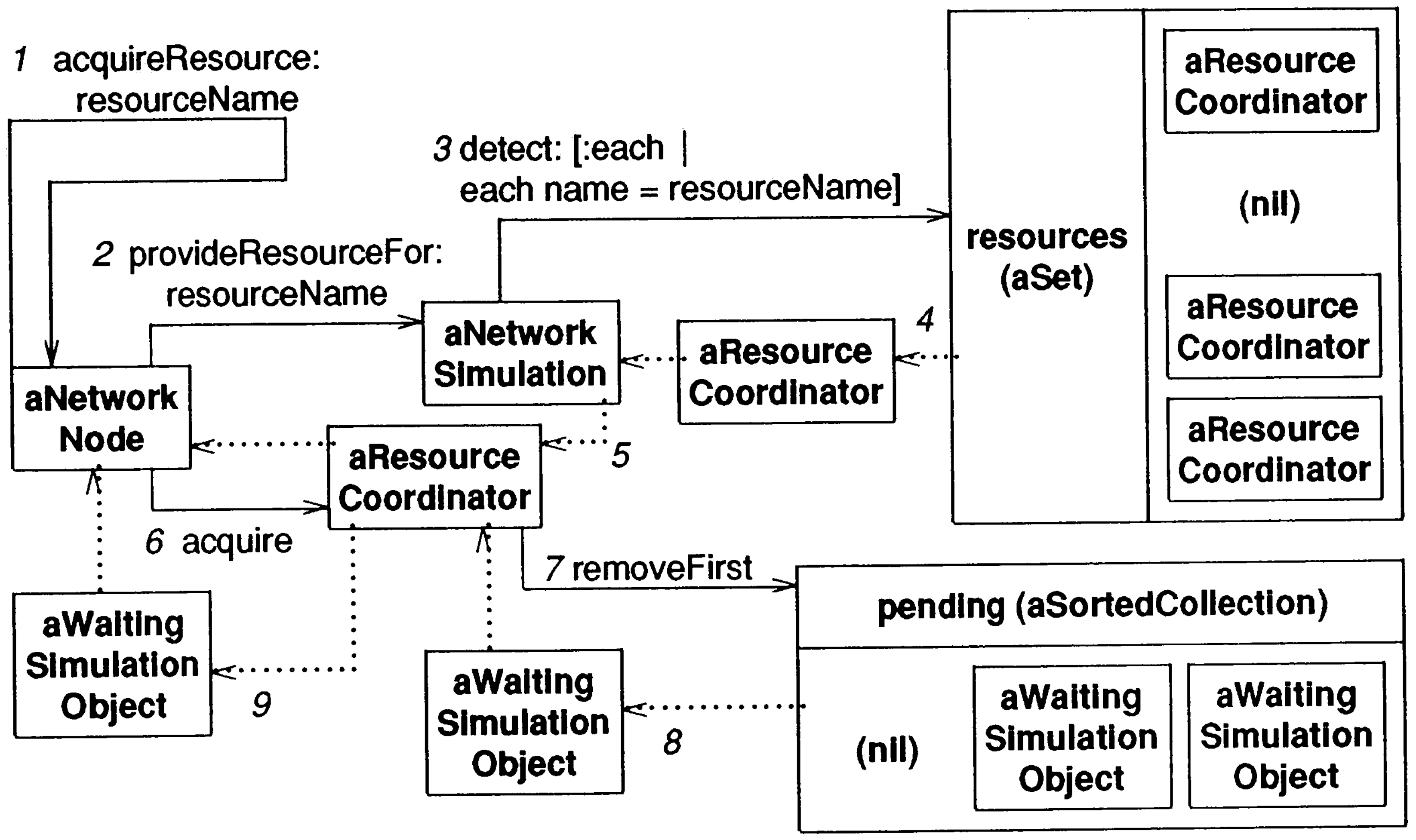
2. Customers, Servers and Resources

The network simulation framework makes use of coordinated resources to model the server/client relationship between messages and nodes (sending) and between nodes and networks (delivering). In this model, the node is the server of both messages and the network. Since it is the hardware, software, and firmware of the network node that implements a networking protocol, it is logical to make the node the active element.

The analogy of a network message as a customer is almost intuitive. In store-and-forward (packet-switched) networks, one can actually visualize a message passing from node to node, much like a customer at the unemployment office being passed from line to line [BeG87].

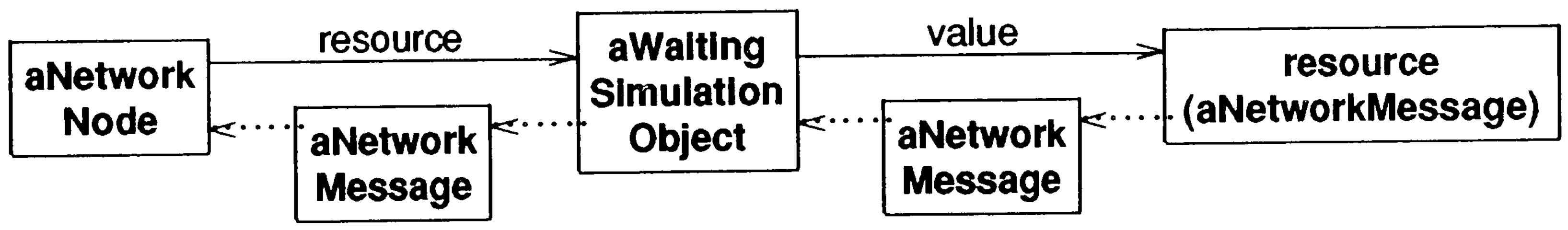
It is not as easy to visualize the network as a customer of the node. In CSMA/CD protocols, however, this relationship is somewhat straightforward. The node must sense that the media is idle, then transmit its message, freeing up the network afterwards [KiL87]. In this sense, the network acts as a resource that is acquired and released by nodes on demand.

Having defined messages and the network as resources (customers), we need to further identify whether they are static or dynamic [GoR83]. Since messages entering and leaving the system is precisely the phenomenon to model, messages are defined as dynamic. And, although the current framework does not model dynamic routing and network down time, the possibility of further development along these lines prompted me to define the network as a dynamic resource as well. Figure 8 depicts the interaction between a node and its message resources.



1 acquireResource: resourceName
3 detect: [:each | each name = resourceName]
2 provideResourceFor: resourceName
aNetwork Node
aNetwork Simulation
aResource Coordinator
4
5
resources (aSet)
aResource Coordinator
(nil)
aResource Coordinator
aResource Coordinator
aResource Coordinator
6 acquire
7 removeFirst
pending (aSortedCollection)
aWaiting Simulation Object
9
aWaiting Simulation Object
8
(nil)
aWaiting Simulation Object
aWaiting Simulation Object

(a) aNetworkNode obtains a ResourceCoordinator from the NetworkSimulation and sends it the message "acquire" to get the WaitingSimulationObject. Message sequencing can be seen by the numeric message labels (1-9).



(b) aNetworkNode retrieves the actual NetworkMessage instance from the WaitingSimulationObject.

Figure 8. The Model Resources

3. The Life Cycle of a NetworkMessage

The most thorough understanding of the model is achieved by following the life cycle of a message as it flows through the simulation. I will trace a message sent from node A to node B.

At $t = 0$, node A schedules the arrival of its outgoing messages, including the one in question (call it message X). An instance of NetworkMessage gets placed into the eventQueue of the NetworkSimulation instance with an associated startup time ($t = t_{start}$).

At $t = t_{start}$, message X enters the simulation as an active object. After recording its entrance time into the simulation, it requests service as a 'MessageFromA' resource. In so doing, aWaitingSimulationObject (whose resource is the NetworkMessage object itself) is placed into the pending queue of the appropriate ResourceCoordinator in NetworkSimulation instance's resources pool. The message is suspended until served (for an undetermined amount of time, t_{queue}).

When node A becomes free of serving other messages to be sent out onto the network, or of performing local processing, it will see that message X needs service (if no messages are queued in front of message X). If the network is free as well, node A acquires the message and the network. If the message is too long or too short, it must be processed (broken into smaller messages or padded up to the minimum length). The time it takes to process the message will be denoted t_{proc}.

Now the node holds (itself and the resources it has acquired, the network and the message) to account for the service time (t_{serve}) or transmission time. The node then asks the network to deliver message X.

At $t = t_{exit} = t_{start} + t_{queue} + t_{proc} + t_{serve}$, the message is delivered to the receiving node (B). Now, node A resumes the message and the network. Message X merely exits the simulation, after recording its exit time and other relevant statistics.

A timeline of a message's life cycle is depicted in Figure 9.

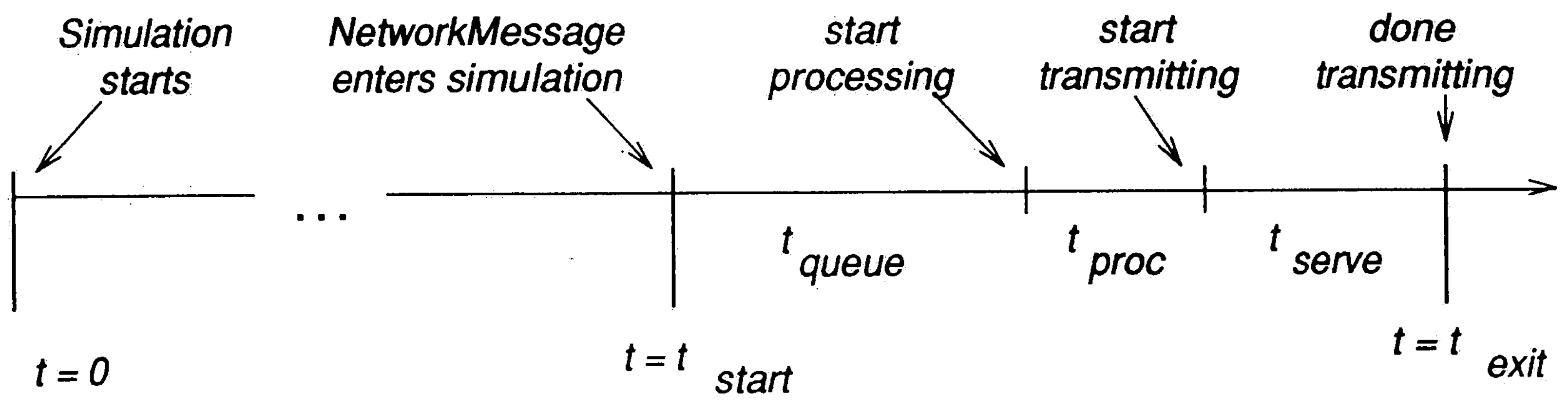


Figure 9. Timeline of a NetworkMessage

4. Statistics Gathering

The primary purpose of performing simulations is to learn something useful about the system under analysis. Most often this information takes the form of statistics. The statistics gathered by the network simulation framework are completely user-defined. Each simulation object maintains its own statistics. Since messages carry their own statistics, any object that interacts with a message may serve as an accumulation point. Statistics that involve many messages (number sent, average delays, etc.) may be accumulated at the nodes, in the network, or by the simulation controller.

For the Network and NetworkNode objects, statistics gathering is completely in the hands of the framework implementor through subclasses. The statistics collected by NetworkMessages are just the times that it entered the simulation, left the outgoing message queue, started transmission and finished transmission. As the NetworkMessage is passed from the source node to the network to the destination node, these message statistics can be accumulated. Each node can maintain its own statistics, while the network can maintain the overall network statistics.

The NetworkSimulation currently maintains just the number of simulation objects entering and exiting the simulation. Since a link is maintained to the network, the NetworkSimulation object does not need to maintain network statistics itself; it can access the information from the network.

B. Implementing the Framework - Class Descriptions

After implementing the generic Smalltalk-80 simulation classes discussed in Section II, there are minimally six additional classes required to implement the network simulation framework. All classes are subclasses of the generic simulation classes in [GoR83], and require only minimal additional programming. In this section, I will fully describe the protocol of the four classes of the framework. Full implementation descriptions are located in Appendix B.

The framework for network simulation defines four classes (two of them abstract) to extend the general simulation capabilities of Simulation and EventMonitor to better fit our needs. NetworkSimulation and NetworkMessage instances can be used without any additional programming. On the other hand, Network and NetworkNode are abstract classes; subclasses must implement certain features that have been left out. Since every network and node behaves uniquely, it is best to force a unique definition from the user.

To describe the classes of the framework, I will again use the protocol description [GoR83]. The implementation descriptions are provided in Appendix B.

1. NetworkSimulation

NetworkSimulation class protocol

---

instance creation

network: aNetworkClass nodes: nodeArray logfile: aFilename
stopTime: anInteger

Answer an instance of NetworkSimulation. Call **initialize:nodes:logfile:stopTime:** to set up instance variables.

NetworkSimulation instance protocol

---

accessing

logfile: aFileStream	Set the logfile parameter to aFileStream.
network	Answer the network to simulate.
network: aNetwork	Set the network instance variable.
parameters	Answer the parameters dictionary.
statistics	Answer the statistics dictionary.
stopTime	Answer the simulation stop time.
stopTime: anInteger	Set the simulation stop time.

initialization

initialize: aNetworkClass nodes: nodeArray logfile: aStream stopTime: anInteger	Create the parameters and statistics instance variables (dictionaries). Set network to a new instance of aNetworkClass, its nodes to new instances of the classes in nodeArray, logfile to aStream and stopTime to anInteger.
defineArrivalSchedule	Schedule the arrival of the network and its nodes into the simulation at time zero.
defineResources	Define the network and the messages as the (coordinated) resources of the simulation. They will act as customers.

statistics

printStatisticsOn: aStream	Print the simulation statistics on aStream.

simulation control

enter: anObject	Record an object's entrance into the simulation.
exit: anObject	Record an object's exit from the simulation.
proceedUntilStopTime	Send **self proceed** until the current time = stopTime.

2. Network (Abstract Class)

Network instance protocol

---

accessing

addNode: aNetworkNode	Add a new node to the network.
nodes	Answer the network nodes as anOrderedCollection.

nodes: anArray	Add each network node in anArray.
parameters	Answer the dictionary of parameters.
statistics	Answer the dictionary of statistics.
initialization	
initialize	Set up instance variables, call **setParameters** and **setStatistics**.
setParameters	Set the bit rate, minimum/maximum packet sizes, and overhead bits.
setStatistics	Set the initial values in the statistics dictionary - subclass responsibility.
simulation control	
tasks	What the network does after startUp. Here, an endless loop to replenish the network resource.
message handling	
deliver: aNetworkMessage	Deliver the message, can be used to gather statistics, log events, etc. - subclass responsibility.
serviceTime: aNetworkMessage	Answer the time to service a message in msec - subclass responsibility.
printing	
printOn: aStream	How the network will print itself.
statistics	
doStatistics: aNetworkMessage	Collect statistics after delivery of aNetworkMessage - subclass responsibility.
incrementStatistic: aSymbol by: aNumber	Increment the statistic, aSymbol, by aNumber.

printStatisticsOn: aStream	Step through the statistics dictionary, printing each key and value on aStream.

3. NetworkNode (Abstract Class)

NetworkNode class protocol

---

instance creation

address: aString network: aNetwork	Answer an instance of NetworkNode whose address is aString and resides on aNetwork.

NetworkNode instance protocol

---

accessing

address	Answer the String representing the receiver's network address.
address: aString	Set the receiver's node address to the String, aString.
network	Answer the network to which the node is connected.
network: aNetwork	Set the network to which the node is connected.
parameters	Answer the dictionary of node parameters.
statistics	Answer the dictionary of node statistics.

initialization

initialize	Initialize the receiver's instance variables and call **setStatistics**.
setStatistics	Set the initial value of any statistics to be collected - subclass responsibility.

simulation control

localProcessing	Define local processing tasks of the node - subclass responsibility.
tasks	What the node does after startUp - schedule traffic originating from the node, check for outgoing messages, send if the network available, otherwise process local jobs.

message scheduling

broadcastMessageStream: aDist fixedLength: anInteger	Schedule fixed-length messages to all nodes using aDist for frequency.
broadcastMessageStream: aDist variableLength: bDist	Schedule variable-length messages to all nodes using aDist for frequency and bDist for length.
messageStream: aDist to: anAddress fixedLength: anInteger	Schedule fixed-length messages to the node at anAddress, using aDist for frequency.
messageStream: aDist to: anAddress variableLength: bDist	Schedule variable-length messages to the node at anAddress, using aDist for frequency and bDist for length.
randomDestination	Answer another network node, selected at random.
randomMessageStream: aDist fixedLength: anInteger	Schedule fixed-length messages to random nodes using aDist for frequency.
randomMessageStream: aDist variableLength: bDist	Schedule variable-length messages to random nodes using aDist for frequency and bDist for length.
traffic	Schedule messages generated from this node - subclass responsibility.

message handling

isNetworkAvailable	Answer true if network is idle.
makePackets: aNetworkMessage	Break aNetworkMessage into smaller messages (packets) according to the network parameter, #maxPacketSize. Answer the size of the first packet (i.e., #maxPacketSize) to transmit and schedule the rest as independent messages.
messagesToSend	Answer true if there are messages to send from this node.
receiveMessage: aNetworkMessage	Receive aNetworkMessage - acts as a place to gather statistics.
sendMessage	Send a message onto the network. If network is idle, acquire the message at the head of the outgoing queue, acquire the network, hold for the transmission time (provided by **self network serviceTime: aNetworkMessage**), ask the network to deliver the message, and finally resume the network and the message. If network busy or no messages to send, do local processing.

statistics

doReceiveStatistics: aNetworkMessage	Gather relevant statistics after receiving a message - subclass responsibility.
doSendStatistics: aNetworkMessage	Gather relevant statistics after sending a message - subclass responsibility.
incrementStatistic: aSymbol by: aNumber	Increment the statistic, aSymbol, by aNumber.
printStatisticsOn: aStream	Step through the statistics dictionary, printing each key and value on aStream.

printing

printOn: aStream	How the node will print itself.

4. NetworkMessage

NetworkMessage class protocol

---

instance creation

from: aString to: bString length: aNumber	Answer a new instance of NetworkMessage with from, to and length initialized.

NetworkMessage instance protocol

---

initialization

initialize	Initialize instance variables and call **setStatistics**.

accessing

from	Answer the source address.
from: aString	Set the source address to aString.
length	Answer the receiver's length (bits).
length: anInteger	Set the receiver's length to anInteger (bits).
to	Answer the destination address.
to: aString	Set the destination address to aString.

statistics

entranceTime	Answer the time the receiver entered the simulation.
receivedAt: currentTime	Tell the receiver that it has been received at the destination node - set #timeToTransmit.

setStatistics	Set up the initial values for the receiver's statistics.
startProcessingAt: currentTime	Tell the receiver that it has been removed from the outgoing queue and is being processed before transmission - set #timeInQueue.
timeInQueue	Answer the amount of time the receiver spent in the outgoing message queue.
timeToProcess	Answer the amount of time it took to process the receiver before transmission.
timeToTransmit	Answer the amount of time it took to transmit the receiver.

printing

printOn: aStream	How a message prints itself.

simulation control

tasks	What a message does after entering the simulation. It merely produces itself as a resource.

C. Expanding the Framework - Implementing Subclasses

As mentioned previously, the framework classes Network and NetworkNode are abstract; subclasses must be defined to implement methods purposely left undefined in the abstract classes [GoR83]. This section will describe how to implement these subclasses to handle a variety of real-world systems.

1. Network Subclasses

A subclass of Network needs to specify **setParameters**, **setStatistics**, **serviceTime:**, and **doStatistics:** methods. The **setParameters** and **setStatistics** methods are invoked by the **initialize** method just after the network enters the simulation (at time zero). As their names imply, they are used to set the network parameters such as bit rate, minimum packet size, etc., and to set initial values for statistics. The **serviceTime:** method is invoked in response to a message from a network node. Its purpose is to tell the node how long it should hold to simulate the message transmission time. Finally, the **doStatistics:** method is called just after message delivery and is used to update user-defined statistics (defined in **setStatistics**).

a. Implementing a **setParameters** *method*

The parameters defined in the **setParameters** method are intended to be accessed only by other user-defined methods (especially **serviceTime**:). Therefore, the choice of parameters to set (and use) is entirely up to the user. The parameters already defined include bitRate, minPacketSize, maxPacketSize, overheadBits, propagationTime, bitErrorRate, and collisionProbability.

For an errorless virtual circuit network, it would be feasible to use just the bitRate and propagationTime parameters as the notions of packets, collisions, and errors would not exist [BeG87]. The service time would merely be:

(bits to send) / (bit rate) + (propagation time)

On a real-world CSMA/CD network such as commercial Ethernet, all of the above parameters, except propagationTime, would be needed to calculate the service time or to split large messages into smaller ones (packets) [GbR87]. A **setParameters** method for a commercial Ethernet network would look like:

```
setParameters
  self bitRate: 10000000.      "10 Mbps"
  self minPacketSize: 368.     "368 bits"
  self maxPacketSize: 12000.  "12,000 bits"
  self overheadBits: 208.      "208 bits"
  self propagationTime: 0.     "negligible for Ethernet"
```

b. Implementing a **setStatistics** *method*

The only purpose of the **setStatistics** method is to initially set all user-defined statistics values to zero (or some other value, if necessary). A series of messages sent to the statistics instance variable (a Dictionary) is all that is required. The message to send is **at: aKey put: aValue**, where aKey is the statistic to collect, expressed as a symbol, and aValue is the initial value desired (usually 0). A typical implementation of **setStatistics** might resemble the following.

```
setStatistics
  statistics at: #numberOfMessagesSent put: 0.
  statistics at: #numberOfBitsSent put: 0.
  statistics at: #totalTimeTransmitting put: 0
```

This method can be simplified somewhat by making use of the "for-loop" control structure shown previously in Section II.A.4:

```
setStatistics
  #(#numberOfMessagesSent #numberOfBitsSent #totalTimeTransmitting)
    do: [ :key | statistics at: key put:0 ]
```

c. Implementing a **serviceTime:** *method*

The delay a message encounters from source to destination may be broken down into four components: processing delay, queuing delay, transmission delay, and propagation delay [BeG87]. In most cases, the processing delay may be neglected, if the computing resources of each node are not constrained (unless one is attempting to compare different processing schemes - for example, the transport control protocol, TCP, versus the user datagram protocol, UDP). For our purposes, it is enough just to be concerned with queuing, transmission, and propagation delays.

The simulation framework is structured such that the queuing delay is completely simulated (i.e., no computations involved - contention for the network is the only factor). Once a message has made it to the network (finished queuing), it is only necessary to simulate the transmission and propagation delay.

To tell the sending node how long to hold the network before releasing it, a **serviceTime:** method is defined. The following implementation will suffice for most errorless networks [BeG87].

```
serviceTime: aNetworkMessage
 ^(aNetworkMessage length + self overheadBits) * (self bitRate)
    + (self propagationTime)
```

One can simulate the effects of errors and collisions by randomly adding delay to the above service time. This level of detail may be beyond the needs of most network managers.

d. Implementing a **doStatistics:** *method*

The **doStatistics:** method is invoked from the **deliver:** method and provides the implementor the opportunity to accumulate statistics previously initialized in the **setStatistics** method. Since this method is called just *after* message delivery, all the NetworkMessage statistics are available. The message just delivered is sent as an argument to **doStatistics**. A message as been provided by the Network class to assist in gathering statistics. The expression "**self incrementStatistic: aSymbol by: aNumber**" will increment any statistic previously defined in the **setStatistics** method.

An implementation that makes use of the statistics defined in the **setStatistics** example above is shown below. This example simply takes the old values and adds the current message's data to them.

```
doStatistics: aNetworkMessage
  self incrementStatistic: #numberOfMessagesSent by: 1.
  self incrementStatistic: #numberOfBitsSent
    by: (aNetworkMessage length + self overheadBits).
  self incrementStatistic: #totalTimeTransmitting
    by: (aNetworkMessage timeToTransmit)
```

2. NetworkNode Subclasses

Subclasses of **NetworkNode** like subclasses of **Network** must handle their own statistics initializations and manipulations. Since the sending statistics and receiving statistics are usually of interest to network managers, separate methods are defined. In addition to the statistics methods (**setStatistics**, **doSendStatistics**, **doReceiveStatistics**), a **traffic** method and a **localProcessing** method must be defined by these subclasses. The **traffic** and **localProcessing** are used to specify the individual node characteristics.

a. Implementing a **setStatistics** *method*

The NetworkNode implementation of **setStatistics** is functionally equivalent to that of the **Network**. An example is shown below where both send and receive statistics are desired.

```
setStatistics
  #(#numberOfMessagesSent #numberOfBitsSent #totalTimeInQueue
    #totalTimeToProcess #totalTimeToTransmit #numberOfMessagesReceived
    #numberOfBitsReceived) do: [ :key |statistics at: key put: 0 ]
```

b. Implementing a **traffic** *method*

The **traffic** method must describe the message traffic generated by the node. This is done by sending messages to the node itself. Messages have been implemented to define broadcast messages (sent to *all* nodes), random messages (sent to a *random* node) and directed messages (sent to a *selected* node). These messages are detailed in Section II.B.C, under the message category "message scheduling." The example below sets up two message streams. The first sends variable length messages (uniformly distributed from 500 to 50,000 bits) to randomly selected nodes with a frequency exponentially distributed with a mean of 50 milliseconds. The second sends a fixed length message (128 bits) to node 'D' with a frequency exponentially distributed with a mean of 25 milliseconds.

```
traffic
  self randomMessageStream: (Exponential mean: 50)
    variableLength: (Uniform from: 500 to: 50000).
  self messageStream: (Exponential mean: 25)
    to: 'D' fixedLength: 128
```

c. Implementing a **localProcessing** *method*

The **localProcessing** method *must* cause some time to elapse from the simulated clock, simulating the node being busy with local processing demands. The longer the delay, the more "busy" the node appears to be. A simple method to simulate a 1 millisecond local processing delay is shown below.

```
localProcessing
  self holdFor: 1
```

d. Implementing a **doSendStatistics:** *method*

This method is invoked just after a message has been sent from the node, and is functionally equivalent to the Network **doStatistics** method. Shown below is an example that uses the statistics initialized previously. Note that the NetworkNode class provides the same **incrementStatistic:by:** as the Network class.

```
doSendStatistics aNetworkMessage
  self incrementStatistic: #numberOfMessagesSent by: 1.
  self incrementStatistic: #numberOfBitsSent
    by: (aNetworkMessage length).
  self incrementStatistic: #totalTimeInQueue
    by: (aNetworkMessage timeInQueue)
  self incrementStatistic: #totalTimeToProcess
    by: (aNetworkMessage timeToProcess).
  self incrementStatistic: #totalTimeToTransmit
    by: (aNetworkMessage timeToTransmit)
```

e. Implementing a **doReceiveStatistics:** *method*

The **doReceiveStatistics** is invoked by the Network just after a message has been delivered to the node. It accumulates the receive statistics at this node. An example is shown below. The **incrementStatistic** method is employed once again.

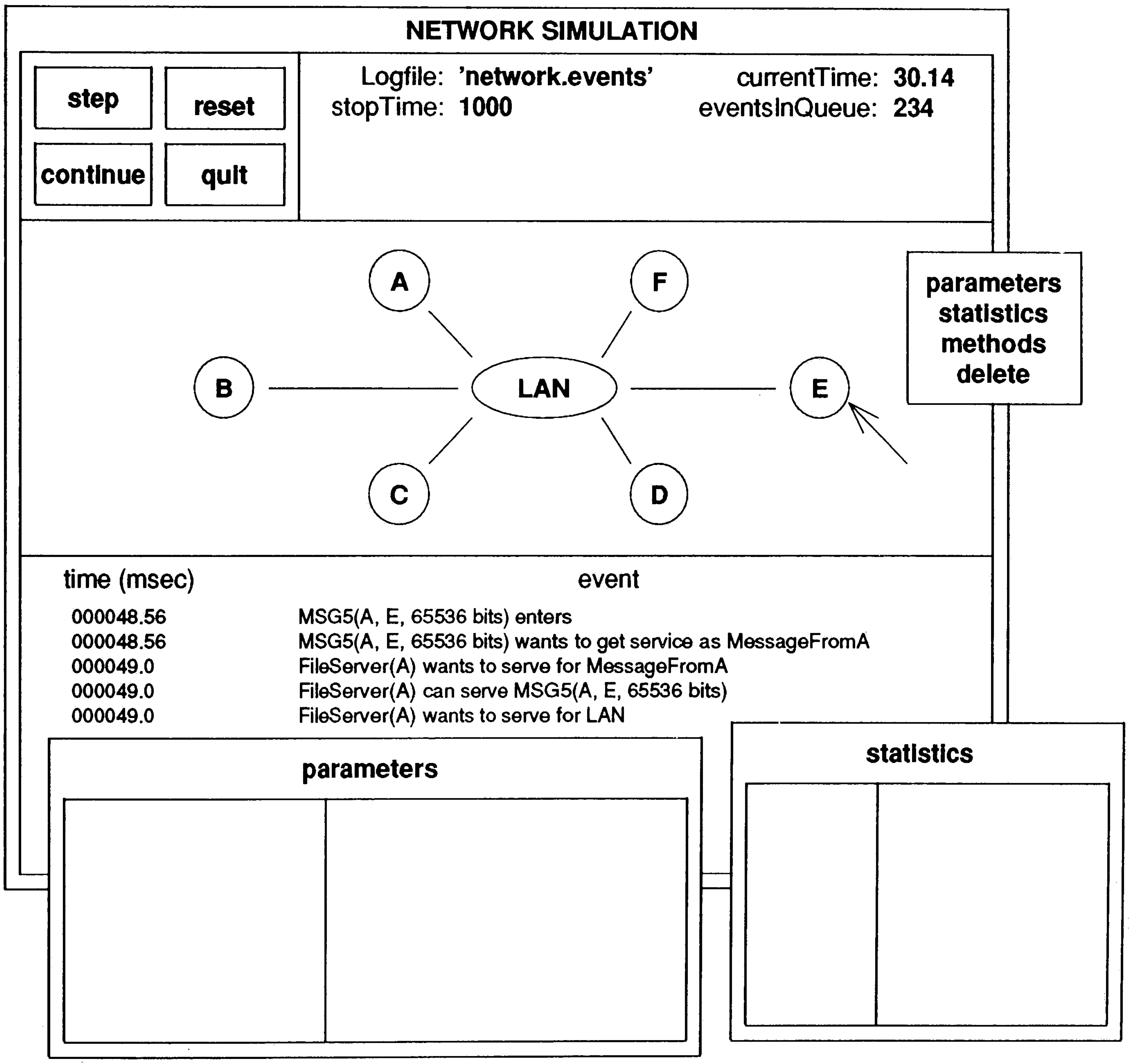
```
doReceiveStatistics aNetworkMessage
  self incrementStatistic: #numberOfMessagesReceived by: 1.
  self incrementStatistic: #numberOfBitsReceived
    by: aNetworkMessage length
```

D. Designing a User Interface

The framework for network simulation presented in this thesis has shown a "hands-on" approach to the problem of developing flexible, yet simple network models. It relies on the implementor being able to actually program in Smalltalk-80. There are instances where one would like a slightly less detailed interface to the simulation model. Even though this would sacrifice flexibility, in environments where the network is stable, flexibility is not a major concern.

The network simulation framework does not directly address the problem of designing a high-level front-end. Smalltalk-80 does make this easy to develop. The following subsection suggests ways to approach the problem of designing the user interface. The description is conceptual in nature; the interested reader is directed to [GoR83] and [Par90] for more concrete descriptions of user-interface implementations. This discussion is intended for the knowledgeable software developer who would like to extend the network simulation framework.

There are three ways in which a user can interact with the framework: setting object parameters, defining object behaviors, and viewing simulation results (statistics). Since each object in the simulation maintains its own parameters and statistics, it would make sense to use an object-oriented front-end. A possible user interface might be implemented as depicted in Figure 10.



```
000048.56     MSG5(A, E, 65536 bits) enters
000048.56     MSG5(A, E, 65536 bits) wants to get service as MessageFromA
000049.0      FileServer(A) wants to serve for MessageFromA
000049.0      FileServer(A) can serve MSG5(A, E, 65536 bits)
000049.0      FileServer(A) wants to serve for LAN
```

Figure 10. Proposed User Interface

The two window panes at the top interact with the NetworkSimulation object. Action buttons for simulation control appear on the left; overall simulation statistics and parameters appear on the right. The lowest window pane would show network events (e.g., messages entering, resources being coordinated, etc.) in the same format as the simulation logfile. The middle pane of the window would be used to select simulation objects (the network or a node) and to bring up a new window to interact with the objects parameters, statistics, or methods.

Objectworks\Smalltalk release 4 provides the ability to attach several different views (from the MVC paradigm) to a display window [Par90]. The ScehduledWindow class obtains a window from the host system's window manager. The CompositePart class enables the programmer to tack multiple views onto the window. Several predefined controllers (called PluggableAdaptors) have been provided to handle such things as selector buttons, scrollable menus, etc. Not only should these classes be employed, but also several classes that make up a part of the programmer's interface.

1. Interacting with the Simulation

The upper window panes interact with the NetworkSimulation object. The action buttons should be implemented as instances of class LabeledBooleanView, attached to PluggableAdaptors that send the appropriate message to the NetworkSimulation object [Par90]. For example, the "step" button might send the message **proceed** whereas the "continue" button might send the message **proceedUntilStopTime**. The upper right pane could be constructed of many StringHolderViews that relate back to individual instance variables of the NetworkSimulation. Or, a single TextHolderView can be constructed to interact with all the instance variables. [Par90]

2. Interacting with SimulationObjects

Interaction with the network and nodes should be more object-oriented. It would seem best that the user point to an object and receive a menu that allows modifying/viewing parameters, modifying/viewing statistics, modifying/viewing methods. If the user wants the ability to change the network topology, one might consider adding a "delete" option to the object menu, then offering a one item ("add node") menu when no object is selected. The ability to create context sensitive action menus is described in [Par90].

Once this controller is developed, standard views taken from the system itself should be used to actually interact with the objects. For instance, when the user selects "parameters" from an object menu, the program would merely create an Inspector on that object's parameters dictionary (i.e., aNetworkNode parameters inspect). The user would thus be able to view or modify any parameter. Similarly, an Inspector would be created for the "statistics" option.

The "methods" option might bring up a Browser on the selected object's class (i.e., aNetworkNode browse). The "add" and "delete" options would have to be custom programmed (i.e., link "add" to the Network's **addNode:** method, program a new **deleteNode:** method).

3. Event Logging

The Smalltalk-80 programming environment provides the TextCollector and TextCollectorView classes for displaying simple messages. An instance of TextCollectorView behaves just like a Stream, the underlying mechanism behind the network simulation framework's event logfile. Implementing the event-monitoring pane is as simple as creating a TextCollectorView, and setting the simulation logfile to that instance.

V. Example: Evaluating LAN Expansion

The following section will show an example of applying the network simulation framework to a real-world example, evaluating the effect on message delay time and effective data rate after adding nodes to an existing network. By doing this analysis it can be determined whether the current network can handle the additional load. In approaching this problem, the initial network is described first. Then, class descriptions for the LAN are shown (classes Workstation, FileServer, and LAN). Network statistics will be shown for the existing network. Then, after adding the additional nodes, network statistics will be shown and compared to the previous results.

A. The Existing LAN

The LAN to be simulated consists of five dataless workstations (i.e., they contain only enough disk space for the operating system and memory swapping) and one file server connected to an IEEE 802.3 CSMA/CD ethernet. The file server will be addressed as 'A'; the workstations will use addresses 'B', 'C', etc. It is assumed that the workstations are all used similarly and can be modeled as one Smalltalk-80 class (identical traffic patterns and local processing demands). It is also assumed that the file server is dedicated (i.e., it is not used for general-purpose computing and therefore has few local processing demands).

Since the workstations are dataless, all data must be retrieved from the file server. The traffic from the workstations to the file server will consist of requests for data. It is assumed that the workstations are used primarily for engineering CAD work. Based on my experience in industry, this type of activity is characterized by heavy local processing demands (graphics, statistical analysis, etc.) and heavy demands for data (3-D graphics files, large data collection files, etc.) Distributed processing is not assumed, therefore traffic from workstation to workstation should be virtually nonexistent.

One popular network architecture allows transmission of up to 8192 bytes per file server request [Sun90]. Given that the applications are dealing with large quantities of data, it is reasonable to assume that maximum size blocks will be requested by the workstations. The file server's traffic can be characterized by randomly address fixed-size messages (8192 bytes * 8 bits/byte = 65,536 bytes), with a frequency *N* times higher than an individual workstation's requests for data, where *N* is the number of client workstations on the network.

A workstation's stream of requests to the file server is characterized by small, nearly fixed-length messages (they just tell the file server what file, and disk block within that file is requested). Using the same network architecture as above, these request messages would typically be about 300 bytes (2400 bits) long [Sun90]. The frequency of these requests depends on the requirements of the application running on the workstation. For engineering CAD, these requirements can be as high as a screen-full of high-resolution graphics every 10 seconds (1 screen-full = 2000 bits x 2000 bits = 4 Mbits = 61 maximum size blocks; 61 block requests / 10000 milliseconds = 1 request every 164 milliseconds).

B. Classes Workstation, FileServer and LAN

File servers and workstations are much alike. Many manufacturers market high-end workstations as file servers. It would therefore behoove us to make use of this similarity when designing the subclass hierarchy for the LAN model. We make Workstation a subclass of NetworkNode, and FileServer a subclass of Workstation. (We could have just as easily made Workstation a subclass of FileServer.) The LAN class is a subclass of Network.

The following class descriptions will use the *implementation description* notation of [GoR83]. Note that FileServer need only implement the **localProcessing** and **traffic** methods; the rest are inherited from Workstation.

```
class name          Workstation
superclass          NetworkNode
instance methods

subclass responsibility

  traffic
    "Identify traffic generated by the receiver, requests for
    a block of data roughly every 164 milliseconds."
    self messageStream: (Exponential mean: 164)
        to: 'A' fixedLength: 2400


  localProcessing
    "Workstations have heavy local demands, so hold for 5
    milliseconds whenever not trying to serve messages."
    self holdFor: 5


  setStatistics
    "Identify and initialize statistics."
    #(#numberOfMessagesSent #totalTimeInQueue #averageTimeInQueue)
        do: [ :key | statistics at: key put: 0 ]
```

```
doSendStatistics: aNetworkMessage
    "Collect statistics after a message is sent from the receiver."
    self incrementStatistic: #numberOfMessagesSent by: 1
    self incrementStatistic: #totalTimeInQueue
        by: aNetworkMessage timeInQueue.
    statistics at: #averageTimeInQueue
        put: (statistics at: #totalTimeInQueue) /
            (statistics at: #numberOfMessagesSent)

doReceiveStatistics: aNetworkMessage
    "Not interested in any receive statistics, just answer the receiver."
    ^self
```

class name	FileServer
superclass	Workstation
instance methods	

subclass responsibility

```
traffic
    "Identify traffic generated by the receiver, satisfying
    workstation requests for a block of data roughly every
    164/(# of workstation) milliseconds."
    self randomMessageStream:
        (Exponential mean: 164 / (self network nodes size - 1))
        fixedLength: (8192 * 8)

localProcessing
    "A file server should have little to do locally, hold
    for just 1 millisecond whenever not trying to serve messages."
    self holdFor: 1
```

class name	LAN
superclass	Network
instance methods	

```
subclass responsibility

    serviceTime: aNetworkMessage
        "Answer the time to send a message, aNetworkMessage, in msec."
        ^((aNetworkMessage length + self overheadBits)
            / self bitRate * 1000) asFloat


    setParameters
        "Initialize the network parameters."
        self bitRate: 10000000.
        self minPacketSize: 368.
        self maxPacketSize: 12000.
        self overheadBits: 208


    setStatistics
        "Identify and initialize statistics."
        #(#numberOfMessagesSent #totalTimeInQueue #averageTimeInQueue
          #totalDelayTime #averageDelayTime #totalBitsSent
          #averageEffectiveDataRate #totalTimeTransmitting #percentIdleTime)
            do: [ :key | statistics at: key put: 0 ]
```

```
doStatistics: aNetworkMessage
    "Collect statistics after a message is sent by the receiver."
    self incrementStatistic: #numberOfMessagesSent by: 1.
    self incrementStatistic: #totalTimeInQueue
      by: (aNetworkMessage timeInQueue).
    statistics at: #averageTimeInQueue
        put: (statistics at: #totalTimeInQueue) /
            (statistics at: #numberOfMessagesSent).
    self incrementStatistic: #totalDelayTime
        by: (Simulation active time) - (aNetworkMessage entranceTime).
    statistics at: #averageDelayTime
        put: (statistics at: #totalDelayTime) /
            (statistics at: #numberOfMessagesSent).
    self incrementStatistic: #totalBitsSent
        by: (aNetworkMessage length).
    statistics at: #averageEffectiveDataRate
        put: ((statistics at: #totalBitsSent) /
            (statistics at: #totalDelayTime) * 1000).
    self incrementStatistic: #totalTimeTransmitting
        by: (aNetworkMessage timeToTransmit).
    statistics at: #percentIdleTime
        put: 1.0 - ((statistics at: #totalTimeTransmitting) /
            (Simulation active time))
```

C. Results

Four simulations were performed using the classes described above. The first run simulated the existing network (i.e., five workstations). Succeeding simulation runs modeled networks of 10, 15 and 20 workstations. The following Smalltalk-80 code was executed from a *workspace* to perform the first simulation run. Similar code was used for the other three runs.

```
| aSimulation statFile |  "Temporary variables."

aSimulation :=           "Create the coordinator, an instance of NetworkSimulation."
   network: LAN
   nodes: #(#('A' FileServer) #('B' Workstation) #('C' Workstation)
         #('D' Workstation) #('E' Workstation) #('F' Workstation))
   logfile: 'network.events'
   stopTime: 2000.         "2000 msec."

aSimulation proceedUntilStopTime.    "Run the simulation."

statFile = (Filename named: 'network.stats') writeStream.    "Record the statistics."
aSimulation printStatisticsOn: statFile.

aSimulation logfile close.      "Close opened files."
statFile close.
```

The file "network.stats" is then examined to retrieve the average message delay and effective data rate for each simulation run. This file looks like:

```
Overall Simulation Statistics - a NetworkSimulation

411                        #numberOfObjectsEntered
399                        #numberOfObjectsExited


Network Statistics - LAN

391.382                    #totalTimeTransmitting
3830816                    #totalBitsSent
399                        #numberOfMessagesSent
0.804232                   #percentIdleTime
6.12366                    #averageTimeInQueue
1.35139e6                  #averageEffectiveDataRate
2834.72                    #totalDelayTime
2443.34                    #totalTimeInQueue
```

```
7.10457                     #averageDelayTime

Node Statistics - FileServer(A)

337                         #numberOfMessagesSent
6.57629                     #averageTimeInQueue
2216.21                     #totalTimeInQueue
          •
          •
          •
```

Using these statistics files, the following table was compiled to show the various effects of increasing the number of workstations on the network in question.

Statistic	Number of Workstations			
	5	10	15	20
# of messages sent	399	827	1033	1139
amount of data sent (Kbits)	3831	8099	9763	10542
# of messages not sent	6	4	68	216
% network idle time	80.4	58.6	50.1	46.1
average delay (msec)	7	14	53	158
eff data rate (Kbps)	1351	699	179	59

The results of these simulations show that the network performance becomes significantly degraded at a point somewhere between 10 and 15 nodes on the network. Ethernet provides a 10 Mbps medium, yet contention for the media reduces the effective data rate to less than 200 Kbps for networks of more than 10 nodes. An average one-way message delay of 50 - 150 milliseconds (as seen in the 10 and 15 node networks) can add 6 - 18 seconds for each 4 MB screen-full of data. This amount of one-way delay would certainly be perceptible. Note also that many messages are not delivered, indicating that the network is bottlenecked and just cannot keep up with the traffic demands. The effect on the network is shown graphically in Figure 11.

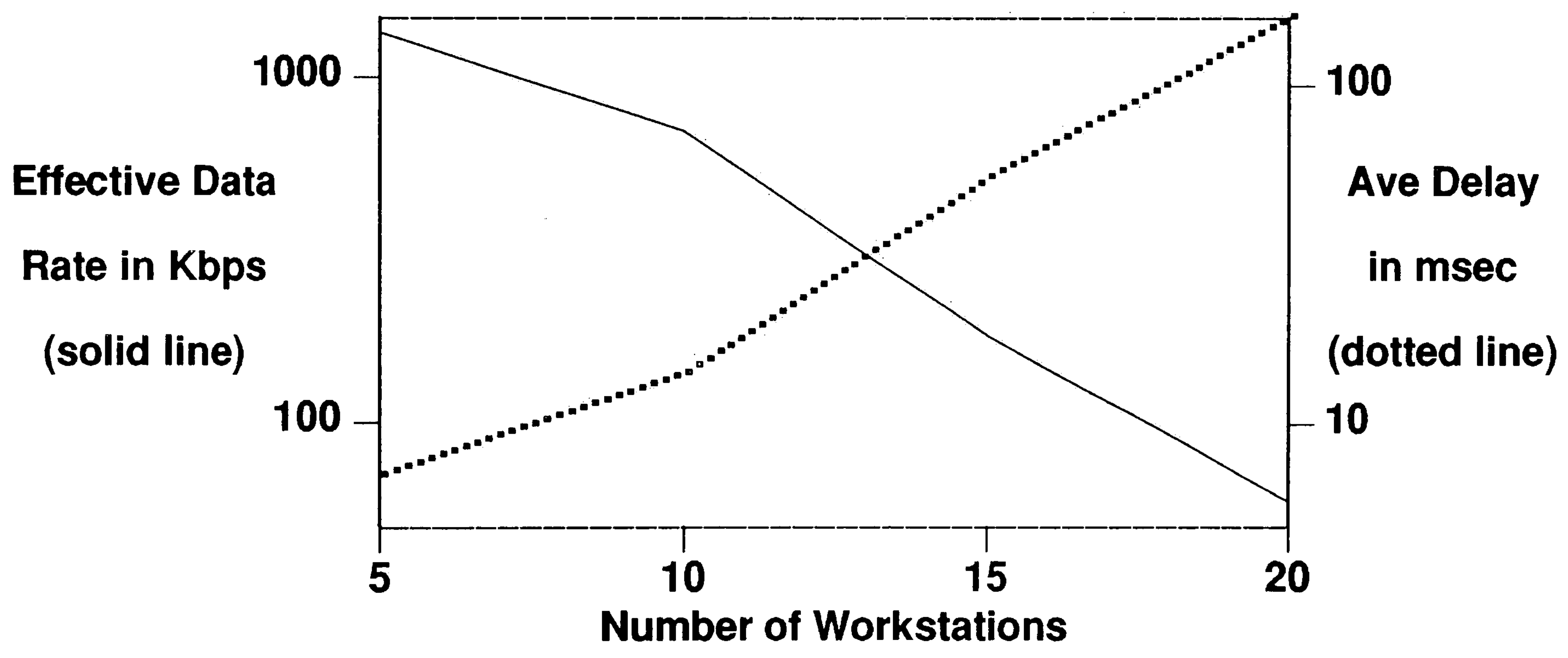


Figure 11. The Effects of Network Expansion

VI. Conclusion

The purpose of this paper was to present an object-oriented framework for developing computer network simulations. This framework was presented as an extension of the Smalltalk-80 general simulation tools proposed by [GoR83]. Statistics gathering, object interaction, and resource coordination were added to make these classes more amenable to the task of simulating computer networks.

The underlying theme throughout this project has been to bring simulation into the hands of the computer network manager. This was accomplished in two ways. First, by selecting a programming language that inherently mimicked the system to be modeled, the concepts of the language were easy to grasp and apply. Second, by implementing much of the detail behind the simulation, little was asked of the network manager other than knowing how his network operated. As is shown in the example in Section IV, little programming is required once the framework has been implemented in Smalltalk-80.

Still, however, the framework does not sacrifice its flexibility for the sake of simplicity. As with all the Smalltalk-80 environment, the computer network simulation framework is modifiable and open for extension at every level of detail. The user interface was already mentioned as likely target for extension (Section III.D). Introducing the effect of collisions and errors into the Network **serviceTime:** method was also mentioned. There are many other areas for future development.

One could define subclasses of Network that implement the **setParameters** and **serviceTime** methods for various standard networks (e.g., SNA, IBM's Standard

Network Architecture, DECNET, Digital Equipment Corporation's NETwork architecture, MAP, the Manuafacturing Automation Protocol, etc.)

The more adventuresome developer might want to rewrite the **sendMessage** and **receiveMessage** methods of NetworkNode to more exactly model a particular networking protocol (i.e., breaking down the message processing delay into its components, introducing delay into the media access method, sending message acknowledgements, etc.)

The network simulation framework should therefore be viewed more as a new approach than as a complete solution. It does not provide an "idiot-proof" front-end, nor does it provide an infinite level of detail. It is meant for the neither the casual user, nor the networking protocol developer. It functions as a core set of tools, around which a computer networking simulation system can be developed by someone needing little more than basic computer programming knowledge.

VII. References

[ASA85] M. Alam, A. Sood, S Akhtar, 1985. "Performance Simulation Model of a Multiple Bus Computer Network Using SLAM", *Proceedings of the 1985 Summer Computer Simulation Conference* p294-298.

[Bac87] Bacon, et al, 1987. "Nest: A Network Simulation and Prototyping Tool", *Technical Report*, IBM T. J. Watson Research Center.

[BeG87] D. Bertsekas, R. Gallager, 1987. *Data Networks* Prentice-Hall, Englewood Cliffs, NJ, p111-283.

[Coo86] S. Cook, 1986. "Languages and Object-Oriented Programming", *Software Engineering Journal* v1 n2 (March 1986) p73-80.

[DeP85] C. Dembeck, C. Porter, 1985. "SIMNET: A Network Simulation Model", *Proceedings of the 1985 IEEE Conference on Computer Simulation* p707-710.

[FLM90] V. Frost, W. LaRue, A. McKee, A. Ernstein, P. Kishore, M. Gormish, 1990. "A tool for local area network modeling and analysis", *Simulation*, November 1990, p283-298.

[GbR87] P. Gburzynski, P. Rudnicki, 1987. "A Better-than-Token Protocol with Bounded Packet Delay Time for Ethernet-type LAN's", *Proceedings of the 1987 IEEE Symposium on Simulation of Computer Networks*, p110-117.

[GoR83] A. Goldberg, D. Robson, 1983. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, p5-73,417-533.

[HeM88] H. Heffes, B. Melamed, 1988. "Visual Simulation of Teletraffic Models", *Proceedings of the 12th International Teletraffic Congress (ITC).*

[Hie88] L. Hiebert, 1988. "AI and Network Planning", *AI Expert*, v3, n9, September 1988, p26-33.

[JaJ87] A. Jayasumana, G. Jayasumana, 1987. "Simulation and Performance Evaluation of 802.4 Priority Scheme", *Proceedings of the 1987 IEEE Symposium on Simulation of Computer Networks*, p232-238.

[KiL87] M. Kim, H. Lin, 1987. "Modeling and Simulation of Channel Access Protocols for Integrated Networks", *Proceedings of the 1987 IEEE Symposium on Simulation of Computer Networks*, p118-122.

[Nic88] S. J. Nichols, et al, 1988. "Design of a High Speed Simulation Tool for WAN Using Parallel Processing", *Microprocessing and Microprogramming* v25 p327-332.

[Par90] ParcPlace Systems, 1990. *Objectworks\Smalltalk User's Guide*, ParcPlace Systems, Mountain View, CA, p1-18,89-106,149-164,189-206.

[Sun90] Sun Microsystems, 1990. *SunOS 4.1 Network Programming Guide*, Sun Microsystems.

[Woo90] P. Woodbury, 1990. "Object-Oriented Menu-Driven Front-End For Simulation of Manufacturing Systems", Master's Thesis, Department of Computer Science and Electrical Engineering, Lehigh University.

[ZDL90] L. Zahn, T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, G. Wyant, 1990. *Network Computing Architecture*, Prentice-Hall, Englewood Cliffs, NJ, p1-39.

[ZTD88] Wei-Dong Zhan, Thanawastien S., Delcambre L, 1988. "SIMNETMAN: An Expert Workstation for Designing Rule-Based Network Management Systems", *IEEE Network* (September 1988) p35-42.

VIII. Appendix A - Simulation Framework Implementation

```
Object subclass: #SimulationObject
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-General'!


!SimulationObject methodsFor: 'task language'!

acquire: amount ofResource: resourceName
   "Get the resource and then tell it to acquire amount of it. Answers
    an instance of StaticResource"
   ^(Simulation active provideResourceFor: resourceName)
      acquire: amount
      withPriority: 0!

acquire: amount
    ofResource: resourceName
    withPriority: priority
   ^(Simulation active provideResourceFor: resourceName)
      acquire: amount
      withPriority: priority!

acquireResource: resourceName
   ^(Simulation active provideResourceFor: resourceName)
     acquire!

amountOfResource: resourceName
   ^(Simulation active provideResourceFor: resourceName)
      amountAvailable!

holdFor: aTimeDelay
   Simulation active delayFor: aTimeDelay!

inquireFor: amount ofResource: resourceName
   ^(Simulation active provideResourceFor: resourceName)
       amountAvailable >= amount!
```

```
numberOfProvidersOfResource: resourceName
   | resource |
   resource := Simulation active provideResourceFor: resourceName.
   resource serversWaiting
      ifTrue: [^resource queueLength]
      ifFalse: [^0]!

numberOfRequestersOfResource: resourceName
   | resource |
   resource := Simulation active provideResourceFor: resourceName.
   resource customersWaiting
      ifTrue: [^resource queueLength]
      ifFalse: [^0]!

produce: amount ofResource: resourceName
   Simulation active produce: amount of: resourceName!

produceResource: resourceName
  ^(Simulation active provideResourceFor: resourceName)
     producedBy: self!

release: aWaitingSimulationObject
  ^aWaitingSimulationObject!

resourceAvailable: resourceName
   "Does the active simulation have a resource with this
    attribute available?"
   ^Simulation active includesResourceFor: resourceName!

resume: anEvent
  ^anEvent resume!

stopSimulation
  Simulation active finishUp! !

!SimulationObject methodsFor: 'simulation control'!

finishUp
   "Tell the simulation that the receiver is done with its tasks."
   Simulation active exit: self!
```

```
startUp
   Simulation active enter: self.
    "First tell the simulation that the receiver is beginning to do my tasks."
   self tasks.
   self finishUp.!

tasks
   "Do nothing. Subclasses will schedule activities."
   ^self! !

!SimulationObject methodsFor: 'initialization'!

initialize
   "Do nothing. Subclasses will initialize instance variables."
   ^self! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

SimulationObject class
        instanceVariableNames: ''!


!SimulationObject class methodsFor: 'instance creation'!

new
  ^super new initialize! !
```

```
SimulationObject subclass: #EventMonitor
        instanceVariableNames: 'label '
        classVariableNames: 'Counter DataFile '
        poolDictionaries: ''
        category: 'Simulation-General'!


!EventMonitor methodsFor: 'private'!

timeStamp
        | now |
        DataFile cr.
        now := Simulation active time asFloat.
        "Pad left margin with zeroes."
        (5 to: 1 by: -1) do:   [ :i |    now < (10.0 raisedTo: i)
                                                  ifTrue: [0 printOn: DataFile]].
        now printOn: DataFile digits:4.
        DataFile tab.
        self printOn: DataFile! !

!EventMonitor methodsFor: 'task language'!

acquire: amount ofResource: resourceName
    | waiting |
    "Store fact that resource is being requested."
    self timeStamp.
    DataFile nextPutAll: ' requests '.
    amount printOn: DataFile.
    DataFile nextPutAll: ' of ',resourceName.
    "Now try to get the resource."
    waiting := super acquire: amount
                    ofResource: resourceName.
    "Returns here when resource is obtained; store the fact."
    self timeStamp.
    DataFile nextPutAll: ' obtained '.
    amount printOn: DataFile.
    DataFile nextPutAll: ' of ',resourceName.
    ^waiting!

acquire: amount
```

```
  ofResource: resourceName
  withPriority: priorityNumber
 |waiting |
 "Store fact that resource is being requested."
 self timeStamp.
 DataFile nextPutAll: ' requests '.
 amount printOn: DataFile.
 DataFile nextPutAll: ' at priority '.
 priorityNumber printOn: DataFile.
 DataFile nextPutAll: ' of ',resourceName.
 "Now try to get the resource."
 waiting := super acquire: amount
              ofResource: resourceName
              withPriority: priorityNumber.
 "Returns here when resource is obtained; store the fact."
 self timeStamp.
 DataFile nextPutAll: ' obtained '.
 amount printOn: DataFile.
 DataFile nextPutAll: ' of ',resourceName.
 ^waiting!

acquireResource: resourceName
 |anEvent |
 "Store fact that resource is being requested"
 self timeStamp.
 DataFile nextPutAll: ' wants to serve for '.
 DataFile nextPutAll: resourceName.
 "Now try to get the resource."
 anEvent := super acquireResource: resourceName.
 "Returns here when resource is obtained; store the fact."
 self timeStamp.
 DataFile nextPutAll: ' can serve '.
 anEvent resource printOn: DataFile.
 ^anEvent!

amountOfResource: resourceName
 |amount |
 self timeStamp.
 amount := super amountOfResource: resourceName.
 DataFile nextPutAll: '  '.
```

```
   amount printOn: DataFile.
   DataFile nextPutAll: ' in ', resourceName!

holdFor: aTimeDelay
   self timeStamp.
   DataFile nextPutAll: ' holds for '.
   aTimeDelay printOn: DataFile.
   super holdFor: aTimeDelay!

produce: amount ofResource: resourceName
   self timeStamp.
   DataFile nextPutAll: ' produces '.
   amount printOn: DataFile.
   DataFile nextPutAll: ' of  ', resourceName.
   super produce: amount ofResource: resourceName!

produceResource: resourceName
   self timeStamp.
   DataFile nextPutAll: ' wants to get service as '.
   DataFile nextPutAll: resourceName.
   super produceResource: resourceName!

release: waiting
   self timeStamp.
   DataFile nextPutAll: ' releases '.
   waiting amount printOn: DataFile.
   DataFile nextPutAll: ' of ',waiting.
   super release: waiting!

resume: anEvent
   self timeStamp.
   DataFile nextPutAll: ' resumes '.
   anEvent resource printOn: DataFile.
   super resume: anEvent! !

!EventMonitor methodsFor: 'scheduling'!

finishUp
   super finishUp.
   self timeStamp.
   DataFile nextPutAll: ' exits '!

startUp
   self timeStamp.
   DataFile nextPutAll: ' enters '.
   super startUp! !

!EventMonitor methodsFor: 'accessing'!

label
   ^label!

setLabel
   Counter := Counter + 1.
   label := Counter printString! !

!EventMonitor methodsFor: 'initialization'!

initialize
   super initialize.
   self setLabel! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

EventMonitor class
       instanceVariableNames: ''!


!EventMonitor class methodsFor: 'initialization'!

file: aFile
   DataFile := aFile.
   Counter := 0! !
```

```
Object subclass: #DelayedEvent
        instanceVariableNames: 'resumptionSemaphore resumptionCondition '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-General'!


!DelayedEvent methodsFor: 'private'!

initialize
   resumptionSemaphore := Semaphore new!

setCondition: anObject
   self initialize.
   resumptionCondition := anObject! !

!DelayedEvent methodsFor: 'scheduling'!

pause
   Simulation active stopProcess.
   resumptionSemaphore wait!

resume
   Simulation active startProcess.
   resumptionSemaphore signal.
   ^resumptionCondition! !

!DelayedEvent methodsFor: 'comparing'!

<= aDelayedEvent
   resumptionCondition isNil
      ifTrue: [^true]
      ifFalse: [^resumptionCondition <= aDelayedEvent condition]! !

!DelayedEvent methodsFor: 'accessing'!

condition
   ^resumptionCondition!

condition: anObject
```

```
   resumptionCondition := anObject! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

DelayedEvent class
        instanceVariableNames: ''!


!DelayedEvent class methodsFor: 'instance creation'!

new
  ^super new initialize!

onCondition: anObject
   ^super new setCondition: anObject! !
```

```
DelayedEvent subclass: #WaitingSimulationObject
        instanceVariableNames: 'amount resource '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-General'!


!WaitingSimulationObject methodsFor: 'private'!

setAmount: aNumber resource: aResource
  amount := aNumber.
  resource := aResource! !

!WaitingSimulationObject methodsFor: 'task language'!

consume: aNumber
   amount := (amount - aNumber) max: 0!

release
   resource produce: amount.
   amount := 0!

release: anAmount
   resource produce: anAmount.
   amount := amount - anAmount! !

!WaitingSimulationObject methodsFor: 'accessing'!

amount
  ^amount!

name
 ^resource name!

resource
  ^resource!

resource: aResource
 resource := aResource! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!
```

```
WaitingSimulationObject class
        instanceVariableNames: ''!


!WaitingSimulationObject class methodsFor: 'instance creation'!

for: amount of: aResource withPriority: aNumber
  ^(self onCondition: aNumber) setAmount: amount resource: aResource!

for: amount withPriority: aNumber
  ^(self onCondition: aNumber) setAmount: amount resource: nil! !
```

```
Object subclass: #Simulation
        instanceVariableNames: 'resources currentTime eventQueue processCount '
        classVariableNames: 'RunningSimulation '
        poolDictionaries: ''
        category: 'Simulation-General'!


!Simulation methodsFor: 'simulation control'!

atEnd
   "Push the simulation to the point of readiness to continue and then
    answer whether it is ready to continue."
    [self readyToContinue]
       whileFalse: [Processor yield].
     ^eventQueue isEmpty!

enter: anObject
   ^self!

exit: anObject
   ^self!

finishUp
   "Empty out the eventQueue."
    eventQueue := SortedCollection new.
   ^nil!

proceed
    | eventProcess |
    [self readyToContinue] whileFalse: [Processor yield].
    eventQueue isEmpty
        ifTrue: [^self finishUp]
        ifFalse: [eventProcess := eventQueue removeFirst.
                currentTime := eventProcess condition.
                eventProcess resume]!

startUp
    self activate.
    self defineResources.
    self defineArrivalSchedule! !
```

```
!Simulation methodsFor: 'accessing'!

includesResourceFor: resourceName
   | test |
   test := resources
              detect: [:each | each name = resourceName]
              ifNone: [nil].
    ^test notNil!

provideResourceFor: resourceName
   ^resources detect: [:each | each name = resourceName]!

self defineArrivalSchedule!

time
  ^currentTime! !

!Simulation methodsFor: 'private'!

readyToContinue
  ^processCount = 0!

schedule: aBlock
     startingAt: timeInteger
     andThenEvery: aProbabilityDistribution
   self newProcessFor:
     ["This is the first time to do the action."
      self delayUntil: timeInteger.
      "Do the action."
      self newProcessFor: aBlock copy.
      aProbabilityDistribution
         do: [:nextTimeDelay |
                "For each sample from the distribution,
                 delay the amount sampled,"
                 self delayFor: nextTimeDelay.
                 "then do the action."
                  self newProcessFor: aBlock copy]]! !

!Simulation methodsFor: 'scheduling'!
```

```
delayFor: timeDelay
  self delayUntil: currentTime + timeDelay!

delayUntil: aTime
  | delayEvent |
  delayEvent := DelayedEvent onCondition: aTime.
  eventQueue add: delayEvent.
  delayEvent pause.!

newProcessFor: aBlock
   self startProcess.
   [aBlock value.
     self stopProcess] fork!

startProcess
   processCount := processCount + 1!

stopProcess
   processCount := processCount - 1! !

!Simulation methodsFor: 'task language'!

coordinate: resourceName
  (self includesResourceFor: resourceName)
        ifFalse: [resources add:
                (ResourceCoordinator named: resourceName)]!

produce: amount of: resourceName
  (self includesResourceFor: resourceName)
    ifTrue: [(self provideResourceFor: resourceName) produce: amount]
    ifFalse: [resources add:
                (ResourceProvider named: resourceName with: amount)]!

schedule: actionBlock after: timeDelayInteger
   self schedule: actionBlock at: currentTime + timeDelayInteger!

schedule: aBlock at: timeInteger
   "This is the mechanism for scheduling a single action."
   self newProcessFor:
      [self delayUntil: timeInteger.
```

```
        aBlock value]!

scheduleArrivalOf: aSimulationObjectClass
   accordingTo: aProbabilityDistribution
  "This means start now."
  self scheduleArrivalOf: aSimulationObjectClass
       accordingTo: aProbabilityDistribution
       startingAt: currentTime!

scheduleArrivalOf: aSimulationObjectClass
   accordingTo: aProbabilityDistribution
   startingAt: timeInteger
  "Note that aSimulationObjectClass is the class SimulationObject or
   one of its subclasses. The real work is done in the private message
   schedule: startingAt: andThenEvery:."
  self schedule: [aSimulationObjectClass new startUp]
       startingAt: timeInteger
       andThenEvery: aProbabilityDistribution!

scheduleArrivalOf: aSimulationObject at: timeInteger
   self schedule: [aSimulationObject startUp] at: timeInteger! !

!Simulation methodsFor: 'initialization'!

activate
  "This instance is now the active simulation."
   RunningSimulation := self.!

defineArrivalSchedule
   "A subclass specifies the schedule by which simulation objects
    dynamically enter into the simulation."
   ^self!

defineResources
   "A subclass specifies the simulation objects that are initially
    entered into the simulation."
   ^self!

initialize
   resources := Set new.
  currentTime := 0.0.
  processCount := 0.
  eventQueue := SortedCollection new! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Simulation class
       instanceVariableNames: ''!


!Simulation class methodsFor: 'accessing'!

active
   ^RunningSimulation! !

!Simulation class methodsFor: 'instance creation'!

new
 ^super new initialize! !
```

```
Object subclass: #Resource
        instanceVariableNames: 'pending resourceName '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-Resources'!


!Resource methodsFor: 'private'!

provideResources
   ^self!

setName: aString
   resourceName := aString.
   pending := SortedCollection new! !

!Resource methodsFor: 'accessing'!

acquire
   ^self!

addRequest: aWaitingSimulationObject
    pending add: aWaitingSimulationObject.
    self provideResources.
    aWaitingSimulationObject pause!

name
  ^ resourceName! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Resource class
        instanceVariableNames: ''!


!Resource class methodsFor: 'instance creation'!

named: resourceName
  ^self new setName: resourceName! !
```

```
Resource subclass: #ResourceProvider
        instanceVariableNames: 'amountAvailable '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-Resources'!


!ResourceProvider methodsFor: 'accessing'!

amountAvailable
  ^amountAvailable! !

!ResourceProvider methodsFor: 'private'!

provideResources
    |waiting |
   [pending isEmpty not
        and: [pending first amount <= amountAvailable]]
           whileTrue:
                [waiting := pending removeFirst.
                 amountAvailable := amountAvailable - waiting amount.
                 waiting resume]!

setName: aResourceName with: amount
   super setName: aResourceName.
   amountAvailable := amount! !

!ResourceProvider methodsFor: 'task language'!

acquire: amountNeeded withPriority: priorityNumber
   |waiting |
   waiting := WaitingSimulationObject
                  for: amountNeeded
                  of: self
                  withPriority: priorityNumber.
    self addRequest: waiting.
    ^waiting!

produce: amount
   amountAvailable := amountAvailable + amount.
   self provideResources! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

ResourceProvider class
        instanceVariableNames: ''!


!ResourceProvider class methodsFor: 'instance creation'!

named: aResourceName
   ^self new setName: aResourceName with: 0!

named: aResourceName with: amount
   ^self new setName: aResourceName with: amount! !
```

```
Resource subclass: #ResourceCoordinator
        instanceVariableNames: 'whoIsWaiting '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-Resources'!


!ResourceCoordinator methodsFor: 'accessing'!

customersWaiting
   ^whoIsWaiting == #customer!

queueLength
   ^pending size!

serversWaiting
   ^whoIsWaiting == #server! !

!ResourceCoordinator methodsFor: 'task language'!

acquire
   |waiting |
   self customersWaiting ifTrue: [^self giveService].
   "get here if there is no customer waiting for the service"
    waiting := WaitingSimulationObject for: 1 withPriority: 0.
    whoIsWaiting := #server.
    self addRequest: waiting.
    ^waiting resource!

producedBy: aCustomer
    |waiting |
    waiting := WaitingSimulationObject for: 1
                                of: aCustomer
                                withPriority: 0.
    self serversWaiting ifTrue: [^self getServiceFor: waiting].
    whoIsWaiting := #customer.
    self addRequest: waiting! !

!ResourceCoordinator methodsFor: 'private'!
```

```
getServiceFor: aCustomerRequest
   | aServerRequest |
   aServerRequest := pending removeFirst.
   pending isEmpty ifTrue: [whoIsWaiting := #none].
   aServerRequest resource: aCustomerRequest.
   aServerRequest resume.
   aCustomerRequest pause!

giveService
   | aCustomerRequest |
   aCustomerRequest := pending removeFirst.
   pending isEmpty ifTrue: [whoIsWaiting := #none].
   ^ aCustomerRequest!

setName: aString
    super setName: aString.
    whoIsWaiting := #none! !
```

IX. Appendix B - Network Simulation Framework Implementation

```
EventMonitor subclass: #NetworkMessage
        instanceVariableNames: 'parameters statistics '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-Data Networks'!


!NetworkMessage methodsFor: 'simulation control'!

tasks
        "All a NetworkMessage needs to do is to ask for service."

        self produceResource: 'MessageFrom' , self from! !

!NetworkMessage methodsFor: 'statistics'!

entranceTime
        "Answer the time that the receiver entered the simulation."

        ^statistics at: #entranceTime!

receivedAt: currentTime
        "Set the amount of time it took to transmit the message."

        statistics at: #timeToTransmit
                put: currentTime -
                        (self entranceTime + self timeInQueue + self timeToProcess)!

setStatistics
        "Set up initial values for the receiver's statistics."

        statistics at: #timeInQueue put: 0.
        statistics at: #timeToProcess put: 0.
        statistics at: #timeToTransmit put: 0.
        statistics at: #entranceTime put: (Simulation active time)!

startProcessingAt: currentTime
        "Set the amount of time the message waited in queue."

        statistics at: #timeInQueue
                put: currentTime - self entranceTime!

startTransmittingAt: currentTime
        "Set the amount of time to process the message before transmitting."

        statistics at: #timeToProcess
                put: currentTime - (self entranceTime + self timeInQueue)!

timeInQueue
        "Answer the amount of time the receiver spent in the outgoing
        message queue."

        ^statistics at: #timeInQueue!

timeToProcess
        "Answer the amount of time it took to process the receiver prior to
        transmission."

        ^statistics at: #timeToProcess!

timeToTransmit
        "Answer the time it took to transmit the receiver."

        ^statistics at: #timeToTransmit! !

!NetworkMessage methodsFor: 'accessing'!

from
        "Answer the address of the source node of the receiver."

        ^parameters at: #from!

from: aString
        "Set the address of the receiver's source node to the String, aString."
        ^parameters at: #from put: aString!

length
        "Answer the length of the receiver, in bits."

        ^parameters at: #length!

length: anInteger
        "Set the length of the receiver, in bits."

        ^parameters at: #length put: anInteger!

to
        "Answer the destination address of the receiver."

        ^parameters at: #to!

to: aString
        "Set the receiver's destination address to the String, aString."

        ^parameters at: #to put: aString! !

!NetworkMessage methodsFor: 'printing'!

printOn: aStream
        "Define how a NetworkMessage prints itself."

        aStream nextPutAll: 'MSG',self label.
        aStream nextPutAll: '(',self from,', ',self to,', '.
        self length printOn: aStream.
        aStream nextPutAll: ' bits)'! !

!NetworkMessage methodsFor: 'initialization'!

initialize
        "Set up instance variables (dictionaries 'parameters' and 'statistics')."

        super initialize.
        parameters := Dictionary new: 20.
        statistics := Dictionary new: 40.
        self setStatistics! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

NetworkMessage class
        instanceVariableNames: ''!


!NetworkMessage class methodsFor: 'instance creation'!

from: aString to: bString length: aNumber
        "Answer an instance of NetworkMessage of length aNumber.  Set to
        from and to addresses to aString and bString, respectively."

        | aNetworkMessage |
        aNetworkMessage := super new.
        aNetworkMessage from: aString.
        aNetworkMessage to: bString.

        "Make sure length is an integer number of bits."
        aNetworkMessage length: (aNumber asFloat truncated).
        ^aNetworkMessage! !
```

```
EventMonitor subclass: #NetworkNode
        instanceVariableNames: 'parameters statistics '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-Data Networks'!


!NetworkNode methodsFor: 'message scheduling'!

broadcastMessageStream: aDist fixedLength: anInteger
        "Ask the simulation to schedule fixed length messages (anInteger bits long)
        destined for all nodes on the network.  Schedule them to occur according
        to the distribution, aDist."

        self network nodes do: [:eachNode | self = eachNode
                ifFalse: [Simulation active
                                schedule: [(NetworkMessage
                                                from: self address
                                                to: eachNode address
                                                length: anInteger) startUp]
                                startingAt: aDist next
                                andThenEvery: aDist]]!

broadcastMessageStream: aDist variableLength: bDist
        "Ask the simulation to schedule variable length messages destined
        for all nodes on the network.  Schedule them to occur according to
        the distribution, aDist.  Vary the length according
        to the distribution, bDist."

        self network nodes do: [:eachNode | self = eachNode
                ifFalse: [Simulation active
                                schedule: [(NetworkMessage
                                                from: self address
                                                to: eachNode address
                                                length: bDist next) startUp]
                                startingAt: aDist next
                                andThenEvery: aDist]]!

messageStream: aDist to: anAddress fixedLength: anInteger
        "Ask the simulation to schedule fixed length messages (anInteger
        bits long) destined for a particular node on the network (anAddress).
        Schedule them to occur according to the distribution, aDist."

        Simulation active
                schedule: [(NetworkMessage
                                from: self address
                                to: anAddress
                                length: anInteger) startUp]
                startingAt: aDist next
                andThenEvery: aDist!

messageStream: aDist to: anAddress variableLength: bDist
        "Ask the simulation to schedule variable length messages destined
        for a particular node on the network (anAddress). Schedule them
        to occur according to the distribution, aDist. Vary the length
        according to the distribution, bDist."

        Simulation active
                schedule: [(NetworkMessage
                                from: self address
                                to: anAddress
                                length: bDist next) startUp]
                startingAt: aDist next
                andThenEvery: aDist!

randomDestination
        "Answer another node on the network, selected at random. Used for
        sending messages to other nodes at random. This method should not
        be modified !!"

        | aSampleSpace destination |

        "Create a sampling space consisting of all nodes on the network."
        aSampleSpace := SampleSpace data: (self network nodes).

        "Choose one at random."
        destination := aSampleSpace next.

        "Keep choosing if self was chosen."
        [(self = destination)] whileTrue: [destination := aSampleSpace next].
        "Return the node selected."
        ^destination!

randomMessageStream: aDist fixedLength: anInteger
        "Ask the simulation to schedule fixed length messages (anInteger
        bits long) destined for randomly selected nodes on the network.
        Schedule them to occur according to the distribution,
        aDist."

        Simulation active
                schedule: [(NetworkMessage
                                from: self address
                                to: self randomDestination address
                                length: anInteger) startUp]
                startingAt: aDist next
                andThenEvery: aDist!

randomMessageStream: aDist variableLength: bDist
        "Ask the simulation to schedule variable length messages destined
        for randomly selected nodes on the network. Schedule them to
        occur according to the distribution, aDist. Vary the length
        according to the distribution, bDist."

        Simulation active
                schedule: [(NetworkMessage
                                from: self address
                                to: (self randomDestination address)
                                length: bDist next) startUp]
                startingAt: aDist next
                andThenEvery: aDist!

traffic
        "Define traffic generated by this node. This method is invoked
        by the tasks method, and is executed just after the node enters
        the simulation. The messages broadcastMessageStream...,
        messageStream..., and randomMessageStream... can be sent to
        self for scheduling standard traffic patterns. Subclasses must
        implement this method."

        "Example (random traffic to random destinations):
```

```
        self
                randomMessageStream: (Exponential mean: 5)
                variableLength: (Uniform from: 10 to: 10000)."

        ^self! !

!NetworkNode methodsFor: 'message handling'!

isNetworkAvailable
        "Answer whether the network is idle.  Do this by checking if the
        resource exists.  Return a boolean value.  This method should
        not be modified !!"

        ^(self numberOfRequestersOfResource:
                (self network class name asString)) > 0!

makePackets: aNetworkMessage
        "Make packets out of aNetworkMessage, answer the length of the first
        packet to transmit.  Packets are created as new instances of
        NetworkMessage.  Create maximum length packets until
        aNetworkMessage is exhausted.  Create a partial packet for what is left
        over.  Note that the current NetworkMessage will be sent out as a
        maximum size packet, therefore, create one less full packet than is
        necessary.  This method should not be modified !!"

        | fullPackets leftOver now |

        "Record the current time so that Simulation does not get confused
                with two messages later on."
        now := Simulation active time.

        "Determine how many full packets are required (integer division)."
        fullPackets := aNetworkMessage length // self network maxPacketSize.

        "Remember the left over bits."
        leftOver := aNetworkMessage length  self network maxPacketSize.

        "Create one less than the number of full packets required."
        [fullPackets > 1]
                whileTrue:
                    "Ask Simulation to do the scheduling; send each packet ASAP."
                    [Simulation active schedule: [(NetworkMessage
                                    from: self address
                                    to: aNetworkMessage to
                                    length: self network maxPacketSize) startUp]
                              at: now.

                    "Decrement the counter."
                    fullPackets := fullPackets - 1].

          "Schedule the partial packet for the left over bits."
          Simulation active schedule: [(NetworkMessage
                    from: self address
                    to: aNetworkMessage to
                    length: leftOver) startUp]
              at: now.

          "Answer the new size that aNetworkMessage should be."
          ^self network maxPacketSize!

     messagesToSend
          "Answer whether there are any messages for this node to send.
          Do this by checking for the resource, 'MessageFrom(address)',
          where address is this node's address.  Return a boolean value.
          This method should not be modified !!"

          ^(self numberOfRequestersOfResource: 'MessageFrom' , self address) > 0!

     receiveMessage: aNetworkMessage
          "This method merely serves as a place to do statistics gathering."

          aNetworkMessage receivedAt: Simulation active time.
          self doReceiveStatistics: aNetworkMessage!

     sendMessage
          "Send a message.  This first checks to see if the network is idle.
          If not, the node will merely perform localProcessing, then exit.  If
          the network is idle, the node acquires a message resource and
          the network resource.  Note that with coordinated resources, you
          must first acquire the resource, then ask the resource for the
```

```
object (aNetworkMessage).  Once resources are acquired, the
message is packetized (if necessary) or padded (if necessary) to
fit within the packet size bounds of the network.  It is up to this
method to simulate the network service time and to log the sending
of a message.  Remember to resume resources when done.  Great
care should be taken when modifying this method."

| msgResource msg netResource |

"Is network idle ?"
self isNetworkAvailable

        "Network is idle, okay to send."
        ifTrue:

                "Acquire the message resource."
                [msgResource := self
                        acquireResource: 'MessageFrom' , self address.

                "Get the actual object, an instance of NetworkMessage,
                and mark the time for statistics gathering."
                msg := msgResource resource.
                msg startProcessingAt: Simulation active time.

                "Fix up message length, either break into smaller messages..."
                (msg length > self network maxPacketSize)
                        ifTrue: [msg length: (self makePackets: msg)].

                "...or pad to the minimum."
                (msg length < self network minPacketSize)
                        ifTrue: [msg length: self network minPacketSize].

                "Acquire the network."
                netResource := self
                        acquireResource: (self network class name asString).

                "Log a message being sent to the logfile, and mark the
                time for statistics gathering."
                self timeStamp.
                DataFile nextPutAll: ' sending '.
```

```
                        msg printOn: DataFile.
                        msg startTransmittingAt: Simulation active time.

                        "Simulate the network service time."
                        self holdFor: (self network serviceTime: msg).

                        "Ask the network to deliver the message."
                        self network deliver: msg.

                        "Collect statistics."
                        self doSendStatistics: msg.

                        "Free up the network and the message."
                        self resume: netResource.
                        self resume: msgResource]

                "Network is busy, go do something else."
                ifFalse:
                        [self localProcessing]! !

!NetworkNode methodsFor: 'simulation control'!

localProcessing
        "The purpose of this method is to define what the node does when it is
        not sending messages.  This includes times when no messages are
        queued up and times when messages are queued, but the network is
        busy.  Somewhere in this method a 'holdFor:' message must be sent to
        self.  If time is not elapsed during this method, a race condition will
        develop."

        "Hold for one millisecond."
        self holdFor: 1!

tasks
        "Define the tasks that this node will perform from the time it enters the
        simulation until it exits.  This method has been set up in a very generic
        manner, and should not need much tailoring.  It serves to call the
        traffic method (to schedule message streams), and then enters an
        infinite loop.  Within this loop it is checking for messages to send,
        sending them if they exist, or performing local processing if there are
        no messages to send."

        "Schedule traffic streams."
        self traffic.

        "Loop forever."
        [true]

                "Any messages to send ?"
                whileTrue: [self messagesToSend

                                                "Yes - send them."
                                                ifTrue: [self sendMessage]

                                                "No - do something else."
                                                ifFalse: [self localProcessing]]! !

!NetworkNode methodsFor: 'initialization'!

initialize
        "This method sets up the instance variables for the node.  'parameters'
        and 'statistics' are both defined as dictionaries.  Internal methods
        require these variables to respond to the standard dictionary protocol.
        Modifying this method is not recommended !!"

        parameters := Dictionary new: 10.
        statistics := Dictionary new: 40.
        self setStatistics!

setStatistics
        "Set up the initial entries in the statistics dictionary.  Subclasses
        must implement."

        ^self! !

!NetworkNode methodsFor: 'accessing'!

address
        "Answer this node's address as stored in the parameters dictionary."
        ^self parameters at: #address!

address: aString
        "Store the node's address in the parameters dictionary."

        ^self parameters at: #address put: aString!

network
        "Answer the network to which this node is connected."

        ^self parameters at: #network!

network: aNetwork
        "Store the node's network in the parameters dictionary."

        ^self parameters at: #network put: aNetwork!

parameters
        "Answer this node's parameters dictionary."

        ^parameters!

statistics
        "Answer this node's statistics dictionary."

        ^statistics! !

!NetworkNode methodsFor: 'statistics'!

doReceiveStatistics: aNetworkMessage
        "Handle statistics gathering after receipt of a message.  Subclasses
        must implement."

        ^self!

doSendStatistics: aNetworkMessage
        "Handle statistics gathering after transmission of a message.  Subclasses
        must implement."

        ^self!
```

```
incrementStatistic: aSymbol by: aNumber
        "Increment the statistic stored as aSymbol by aNumber."

        statistics at: aSymbol put: (statistics at: aSymbol) + aNumber!

printStatisticsOn: aStream
        "Print the node statistics to the Stream, aStream."

        aStream cr.
        aStream nextPutAll: 'Node Statistics - '.
        self printOn: aStream.
        aStream cr.
        aStream cr.
        statistics keysAndValuesDo:
                [ :stat :val |   val printOn: aStream.
                                        aStream tab.
                                        stat printOn: aStream.
                                        aStream cr].
        aStream cr.! !

!NetworkNode methodsFor: 'printing'!

printOn: aStream
        "Print the node's class and address on aStream."

        aStream nextPutAll: self class name asString,'(',self address,')'! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

NetworkNode class
        instanceVariableNames: ''!


!NetworkNode class methodsFor: 'instance creation'!

address: aString network: aNetwork
        "Answer an instance of NetworkNode with address set to aString and
        on the network aNetwork."

        | aNetworkNode |
        aNetworkNode := super new.
aNetworkNode address: aString.
aNetworkNode network: aNetwork.
^aNetworkNode! !
```

```
Simulation subclass: #NetworkSimulation
        instanceVariableNames: 'parameters statistics network '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-Data Networks'!


!NetworkSimulation methodsFor: 'accessing'!

logfile
        "Answer the simulation's event logging file."

        ^self parameters at: #logfile!

logfile: aFileStream
        "Set the simulation's event logging file to aStream."

        ^self parameters at: #logfile put: aFileStream!

network
        "Answer the network object of the simulation."

        ^network!

network: aNetwork
        "Set the network object to be simulated to aNetwork."

        ^network := aNetwork!

parameters
        "Answer the dictionary of simulation parameters."

        ^parameters!

statistics
        "Answer the dictionary of simulation statistics."

        ^statistics!

stopTime
```

```
        "Answer the simulation scheduled stop time."

        ^self parameters at: #stopTime!

stopTime: anInteger
        "Set the simulation stop time to anInteger (milliseconds)."

        ^self parameters at: #stopTime put: anInteger! !

!NetworkSimulation methodsFor: 'initialization'!

defineArrivalSchedule
        "Schedule the arrival of static objects to the simulation (the network
        and the nodes).  All static objects enter at time zero.  Also schedule
        the finishUp method to clean things up after stopping.  This method
        should not be modified !!"

        "First the network."
        self scheduleArrivalOf: (self network) at: 0.

        "Then each node."
        (self network nodes) do:
                [ :eachNode |self scheduleArrivalOf: eachNode at: 0.0].

        "Schedule cleanup."
        self schedule: [self finishUp]
                at: self stopTime!

defineResources
        "Establish what resources will be present in the simulation.  Resources
        must be defined for the network and for messages.  Since the nodes will
        need to communicate with the network and messages to service them,
        these resources must be 'coordinated'.  This method should not be
        modified !!"

        self coordinate: (self network class name asString).
        (self network nodes) do:
                [ :eachNode |self coordinate: 'MessageFrom', eachNode address]!

initialize: aNetworkClass nodes: nodeArray logfile: aStream stopTime: anInteger
```

```
        "Initialize the simulation.  Set the network to be simulated to an
        instance of aNetworkClass.  Step through the nodeArray generating
        instances and linking them to the network.  Set up instance variables
        for the 'parameters' and 'statistics' dictionaries.  Set the logfile to
        aStream, and stop time to anInteger.  This method should not
        be modified."

        | aNetwork |

        "Do any initialization performed by the Simulation class."
        super initialize.

        "Create the dictionaries."
        parameters := Dictionary new: 10.
        statistics := Dictionary new: 20.
        self setStatistics.

        "Set logfile and stopTime."
        self logfile: aStream.
        self stopTime: anInteger.

        "Create the network."
        aNetwork := aNetworkClass new.

        "Add the nodes."
        nodeArray do:
                [:eachNode | aNetwork addNode:

                        "Create an instance of the class provided in the nodeArray."
                        ((Smalltalk at: (eachNode last) asSymbol)

                                "address:network: is the instance creation method
                                for nodes."
                                address: (eachNode first) network: aNetwork)].

        "Link the network to the simulation."
        self network: aNetwork!

setStatistics
        "Set up the initial entries in the statistics dictionary."
        statistics at: #numberOfObjectsEntered put: 0.
        statistics at: #numberOfObjectsExited put: 0! !

!NetworkSimulation methodsFor: 'statistics'!

printStatisticsOn: aStream
        "Print the overall simulation statistics on aStream.  Then ask network
        to print statistics on aStream."

        aStream cr.
        aStream cr.
        aStream nextPutAll: 'Overall Simulation Statistics - '.
        self printOn: aStream.
        aStream cr.
        statistics keysAndValuesDo:
                [ :stat :val |   val printOn: aStream.
                                        aStream tab.
                                        stat printOn: aStream.
                                        aStream cr].
        aStream cr.

        "Now call the network to print its statistics."
        network printStatisticsOn: aStream! !

!NetworkSimulation methodsFor: 'simulation control'!

enter: anObject
    statistics at: #numberOfObjectsEntered
            put: (statistics at: #numberOfObjectsEntered) + 1!

exit: anObject
    statistics at: #numberOfObjectsExited
            put: (statistics at: #numberOfObjectsExited) + 1!

proceedUntilStopTime
        [self time < self stopTime]
                whileTrue: [self proceed].
        ^self! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!
```

```
NetworkSimulation class
        instanceVariableNames: ''!


!NetworkSimulation class methodsFor: 'instance creation'!

network: aNetworkClass nodes: anArray logfile: aString stopTime: anInteger
        "Answer an instance of NetworkSimulation, with an instance of
        aNetworkClass to be simulated.  Add the nodes from anArray
        to the network.  Set the event logging file to the file named
        aString.  Set the stopTime to anInteger.  This method should not
        be modified !!"

        "Example:

        NetworkSimulation
                network: Network
                nodes: #(#('A' NetworkNode) #('B' NetworkNode))
                logfile: 'netsim.events'
                stopTime: 1000.

        returns a simulation of a Network, with two nodes, of type
        'NetworkNode' that logs events to the file 'netsim.events', and stops
        after 1000 milliseconds."

        | aSimulation aFileStream |

        "Create a writeStream that writes to the file named aString."
        aFileStream := (Filename named: aString) writeStream.

        "Set event logging file for other objects in the simulation."
        aNetworkClass file: aFileStream.
        NetworkMessage file: aFileStream.
        NetworkNode file: aFileStream.

        "Create a new instance, and initialize it."
        aSimulation := super new
                                initialize: aNetworkClass
                                nodes: anArray
                                logfile: aFileStream
```

```
                    stopTime: anInteger.

        "Get things going."
        aSimulation startUp.

        "Answer the instance of NetworkSimulation."
        ^aSimulation! !
```

```
EventMonitor subclass: #Network
        instanceVariableNames: 'parameters statistics nodes '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation-Data Networks'!


!Network methodsFor: 'simulation control'!

tasks
        "Define the tasks that the network will perform from the time it enters the
        simulation until it exits.  This method has been set up in a very generic
        manner, and should not need much tailoring.  It merely has to keep
        reproducing the network resource (since nodes effectively consume it
        when they grab it."

        [true]
                whileTrue:
                        [self produceResource: (self class name)]! !

!Network methodsFor: 'statistics'!

doStatistics: aNetworkMessage
        "Do statistics gathering after a message has been delivered."

        ^self!

incrementStatistic: aSymbol by: aNumber
        "Increment the statistic stored as aSymbol by aNumber."

        statistics at: aSymbol put: (statistics at: aSymbol) + aNumber!

printStatisticsOn: aStream
        "Print the network statistics to the Stream, aStream."

        aStream cr.
        aStream nextPutAll: 'Network Statistics - '.
        self printOn: aStream.
        aStream cr.
        aStream cr.
        statistics keysAndValuesDo:
                [ :stat :val |    val printOn: aStream.
                                  aStream tab.
                                  stat printOn: aStream.
                                  aStream cr].
        aStream cr.

        nodes do: [ :each |each printStatisticsOn: aStream]! !

!Network methodsFor: 'printing'!

printOn: aStream
        "Print the network's class on aStream."

        aStream nextPutAll: self class name asString! !

!Network methodsFor: 'accessing'!

addNode: aNetworkNode
        "Add a node to the network - store it in the OrderedCollection, nodes."

        ^nodes add: aNetworkNode!

bitErrorRate
        "Answer the bit-error-rate of the network."

        ^self parameters at: #bitErrorRate!

bitErrorRate: aNumber
        "Set the bit-error-rate of the network - store in parameters."

        ^self parameters at: #bitErrorRate put: aNumber!

bitRate
        "Answer the network's bit rate (transmission speed) in bits per second."

        ^self parameters at: #bitRate!

bitRate: anInteger
        "Set the network's bit rate (transmission speed) - store in parameters."
        ^self parameters at: #bitRate put: anInteger!

collisionProbability
        "Answer the probability of a collision on the network."

        ^self parameters at: #collisionProbability!

collisionProbability: aNumber
        "Set the probability of a collision on the network - store in parameters."

        ^self parameters at: #collisionProbability put: aNumber!

maxPacketSize
        "Answer the network's maximum packet size in bits."

        ^self parameters at: #maxPacketSize!

maxPacketSize: anInteger
        "Set the network's maximum packet size - store in parameters."

        ^self parameters at: #maxPacketSize put: anInteger!

minPacketSize
        "Answer the network's minimum packet size in bits."

        ^self parameters at: #minPacketSize!

minPacketSize: anInteger
        "Set the network's minimum packet size - store in paramters."

        ^self parameters at: #minPacketSize put: anInteger!

nodes
        "Answer the nodes connected to the network - as an OrderedCollection."

        ^nodes!

nodes: anArray
        "Add nodes listed in anArray to the network."
        anArray do: [:i |self addNode: i]!

overheadBits
        "Answer the number of overhead bits per packet required by the network."

        ^self parameters at: #overheadBits!

overheadBits: anInteger
        "Set the number of overhead bits per packet to anInteger - store
        in parameters."

        ^self parameters at: #overheadBits put: anInteger!

parameters
        "Answer the dictionary of network parameters."

        ^parameters!

propagationTime
        "Answer the network's propogation time in milliseconds."

        ^self parameters at: #propagationTime!

propagationTime: aNumber
        "Set the network's propagation time - store in parameters."

        ^self parameters at: #propagationTime put: aNumber!

statistics
        "Answer the dictionary of network statistics."

        ^statistics! !

!Network methodsFor: 'initialization'!

initialize
        "Set up the instance variables (dictionaries 'parameters' and 'statistics',
        and the ordered collection, 'nodes'). Call setParameters to establish
        the network's parameters."
        parameters := Dictionary new: 20.
        statistics := Dictionary new: 40.
        nodes := OrderedCollection new.
        self setParameters.
        self setStatistics!

setParameters
        "Store the network's parameters.  At a minimum, bitRate, minPacketSize,
        and maxPacketSize should be specified.  Subclasses must implement"

        ^self!

setStatistics
        "Set up the initial entries in the statistics dictionary.  Subclasses
        must implement."

        ^self! !

!Network methodsFor: 'message handling'!

deliver: aNetworkMessage
        "Deliver a message.  This method serves as a statistics gathering and
        event monitoring point in addition to actually passing the message to
        the receiver."

        self timeStamp.
        DataFile nextPutAll: ' delivering '.
        aNetworkMessage printOn: DataFile.

        "Send the NetworkMessage to all nodes with address = to."
        self nodes do:
                [ :each | each address = aNetworkMessage to
                                        ifTrue: [each receiveMessage: aNetworkMessage]].

        self doStatistics: aNetworkMessage!

serviceTime: aNetworkMessage
        "Answer the time to send a message in milliseconds.  Only use
        parameters that are defined in setParameters.  Subclasses
        must implement"
```

```
^self! !
```

X. Biography

Bruce R. Varnerin was born of Lawrence and Marie Varnerin in Summit, N.J. on January 10, 1964. He attended Purdue University and was awarded a B.S. in Computer and Electrical Engineering in 1985. Since then, he has worked for AT&T Microelectronics in Allentown, PA. As a Member of Technical Staff, his responsibilities include international wide-area-network management, distributed database design and development, local-area-network design and general systems engineering. Bruce will receive an M.S. in Computer Science from Lehigh University in June, 1991.