An Object-Oriented Approach to Modeling and Simulation of Routing in Large Communication Networks

Armin R. Mikler, Johnny S.K. Wong, Vasant Honavar
Department of Computer Science
Iowa State University
Ames, Iowa 50011, USA
e-mail: mikler|wong|honavar@cs.iastate.edu

Keywords: Object-oriented simulation, discrete event simulation, communication networks, routing

Abstract

The complexity (number of entities, interactions between entities, and resulting emergent dynamic behavior) of large communication environments which contain hundreds of nodes and links make simulation an important tool for the study of such systems. Given the difficulties associated with complete analytical treatment of complex dynamical systems, it is often the only practical tool that is available. This paper presents an example of a flexible, modular, object-oriented toolbox designed to support modeling and experimental analysis of a large family of heuristic knowledge representation and decision functions for adaptive self-managing communication networks with particular emphasis on routing strategies. It discusses in detail, the design, implementation, and validation of the toolbox using the discrete event simulation paradigm. It addresses several major practical design challenges presented by potentially large computational time and memory requirements of simulation of large communication networks and describes novel solutions adapted from the simulation and performance modeling and analysis literature to meet such challenges. It also examines some issues that arise in the validation of the resulting object-oriented simulation and modeling toolbox.

1 Introduction

Simulation is a useful, and often essential tool for the design, implementation, and verification of systems with large number and variety of entities. Their behavior is the result of complex interactions among its entities. This complexity, and the resulting dynamics make an analytical study often impossible. One example of such systems is a high-speed communication network with hundreds or even thousands of nodes and links.

Many network simulation models, such as that of a single network node or a local area network (LAN) only have a small number of entities. Hence, a detailed performance analysis is generally feasible. Larger systems can often be decomposed into smaller sub-units, which then can be analyzed independently to determine the performance of the overall system. The ever increasing need for rapid and reliable data transfer over very long distances has led to unprecedented increase in size and complexity of global communication networks. The system's dynamic behavior emerges from the interaction among individual entities. Their interaction is often designed to optimize global performance criteria. This makes it difficult to understand the behavior of such systems by purely analytical means.

Various well designed simulation packages with a wide range of features are available [1, 20, 3]. Off the shelf simulation packages generally provide a large set of pre-defined functions implemented in various types of simulation modules that enable users to easily construct a variety of simulation models. In addition, they usually provide a graphical user interfaces, and a wide variety of analysis tools. However, users do not have control over the degree of detail of the individual modules. Hence, the resource demands for very large simulation models may become prohibitive which in turn limits the complexity of a model. The need to simulate a system with hundreds or even thousands of independent objects made it necessary to explore a new, object oriented design for the implementation of an object oriented toolbox simulating large communication networks.

The remainder of this paper is organized as follows: Section 2 provides an overview of the structure of the communication environment and its decomposition into independent entities. Section 3 describes our implementation of an object oriented toolbox for network simulation. It also discusses some issues that arise in the validation of the resulting toolbox. Section 4 concludes with a summary of the paper.

2 Framework for Simulating Large Network

The issue of uncertainty in communication networks and the simulation of learning from observation in such an environment to enhance fault management has been discussed in [8]. Our efforts to design an object oriented simulation environment were driven by the need to evaluate heuristic routing strategies for large communication infrastructures [10, 11, 12, 14]. The network to be simulated may consist of hundreds or even thousands of nodes, connected by communication links [18]. Network nodes (routers) and communication links are the basic entities that need to be modeled in simulation. Individual nodes and links are modeled as independent entities together with their associated functions. The behavior of each node is determined by various parameters such as link bandwidth, service rate, the choice of a routing algorithm, as well as parameters which control the acquisition and representation of the network state.

2.1 The Model

Routing [2, 19] is the task of propagating a message from its source node towards its destination node. The routing algorithm used in the network determines how an intermediate node selects one of its neighbors as the *next* node to which the message should be sent. Routing decisions dictate the path travelled by a message and hence determine the performance of the network as measured by metrics such as path length and total delay.

One of the differences among the various routing algorithms is the complexity of network state information that must be acquired by each network node in order to make appropriate routing decisions. The complexity of network state information is reflected by the resource demand of the routing algorithm in terms of

- memory requirements,
- computational complexity, and
- bandwidth overhead.

While for some routing algorithms such as random routing and hot potato routing induce very little resource overhead, others, such as distance vector and link state routing, have to acquire information about the entire network. There clearly exists a tradeoff between the associated resource overhead and the average network performance. That is, the amount and precision of network state information upon which routing decisions are based will ultimately determine their quality. In the context of our research, we have designed a set of heuristic knowledge representation and decision functions capable of finding low delay routes while minimizing the associated resource overhead.

The evaluation of adaptive routing algorithms in a dynamic network environment requires simulation experiments with network models of different sizes, topologies, and traffic patterns. The underlying simulation environment must therefore be flexible enough to accommodate the various models. Some of the requirements of such simulation environments have been elaborated in [1, 20]. The simulation environment must provide for efficient instantiation of entities, such as network nodes, links, and messages. This enables the user to make appropriate changes without redesigning and reprogramming the entire model and hence allows for the reuse of available modules.

With increasing network size, the resource requirement for the simulation becomes a critical issue to be considered in the design of the simulation environment. In order to accomodate very large network models simulation, experiments should focus on only those functions/features that are deemed essential for the analysis; other functions are to be abstracted and represented implicitly or left out completely in order to limit the required system resources.

The simulation environment used in this research is flexible enough to host a variety of models. Since most changes to a simulation model only impact a small part of the testbed, the testbed design is kept modular, separating different functional units. The simulation environment provides for the performance evaluation of the model in terms of standard performance metrics such as average delay, throughput, and load measures. However, the user can easily implement mechanisms to capture other measurements of interest. The modular design of the testbed makes this task significantly easier and more efficient.

2.1.1 Network Nodes

Nodes in our model act as source (i.e., inject new messages into the network) as well as destination (i.e., remove messages from the network). The influx of messages into a network node n_i is assumed to be Poisson distributed with a mean of λ, the arrival rate. This Poisson stream, however, can be broken down into two sub-streams, S_1 and S_2 with means of λ_1 and λ_2, respectively [7, 16]. S_1 consists of messages generated in n_i and injected into the network. Stream S_2 consists of messages that were sent to n_i by nodes in the neighborhood H_i of n_i. A node n_j is said to be in H_i if n_i and n_j are connected via a communication link $l_{i,j}$.

Upon receiving a message m_k, it is added to a central queue in n_i to await service. The service performed by a node n_i consists of the removal of m_k from the queue and its propagation to a node $n_j \in H_i$ or delivery of m_k to a user process if n_i is the destination of m_k. Otherwise, a neighbor node $n_j \in H_i$ must be selected for the propagation of m_k towards its destination. In order to make a routing decision (i.e., selecting n_j) a node must acquire and maintain an adequately precise and up-to-date representation of the state of the network. The type and amount of information upon which routing decisions are based clearly depends on the routing algorithm that is used. For example, the network state may be represented in the form of distance tables or routing tables as in distance vector routing, or in the form of a distance tree as used in link state routing. In our system, the network state is represented by a knowledge base consisting of local load measures that are supplemented by global summary information [10, 11, 12, 13, 14]. The functions associated with a network node are: message generation, message routing, network state acquisition, and network state representation.

2.1.2 Communication Links

Two nodes, n_i and n_j communicate via a communication link $l_{i,j}$. Links are assumed to be uni-directional. Various parameters such as bandwidth, cost, error rate, etc., are generally associated with a network link $l_{i,j}$. A link is generally used to communicate both data messages as well as network state information (i.e., control messages).

Within a simulation model, various functions can be associated with a communication link. In our model it is primarily used to simulate the transmission delay between nodes due to bandwidth constraints. In general, functions which model the dropping of messages, corruption of data, or simulated link failures may be associated with a communication link.

2.1.3 Network State

The state of the network is determined by the rate at which messages arrive and depart from various queues, as well as the set of messages that are awaiting service. Hence, the state of the network is the collection of all individual node and link states. For large network models, it is not feasible to maintain a centralized complete and up-to-date knowledge of the network state. Our model assumes that network nodes acquire global network state information individually through appropriate update procedures.

Network traffic consists of messages in the network and can generally not be abstracted by a generic representation. Messages must be represented explicitly as they determine the behavior of nodes and links at a particular point in time. Each message is therefore instantiated and represented by a tuple of values, such as, source address, destination address, message id, as well as information used for performance analysis.

The network model in our simulation environment is an abstraction of the real communication environment at the level of nodes, links, and messages. As the state of an individual node and link is determined by the rate of message arrival and departure, different network load patterns can be simulated by changing the packet generation or service rate in some or all nodes. It should be noted that our simulation testbed can be extended to facilitate the investigation of various functions performed by network nodes and links (such as admission policies, queueing disciplines, etc.).

2.2 Simulation Methodology

We have chosen to base the underlying simulation driver on discrete event simulation, although process emulation has been considered [1, 3]. The fact that each *active* entity in the model would have to be implemented as an independent process renders process emulation non-suitable as the overhead associated with creating large number of proceses and communicating among them becomes prohibitive as the model size increases.

Two forms of Discrete Event Simulation (DES), namely time driven and event driven, are deemed to be appropriate for simulating a testbed for routing algorithms. Both approaches are briefly discussed and contrasted below.

2.2.1 Discrete Event Simulation

Typically, performance modeling involves the simulation of different system states which are represented by the presence or absence of countable units, such as jobs, requests, processes, messages, users, or errors. A new state can only be entered through the execution of an event that modifies one or more of these units. As each event, and hence state change, involves a specific number of discrete units, this type of simulation is generally referred to as Discrete Event Simulation (DES) [3, 5].

DES is further divided into two broad classes, namely, time driven simulation and event driven simulation. For time driven simulation, an event e is selected from an event set E at every tick of a global clock. E is said to contain all plausible events that can execute in the current state. The fact that no state change takes place at time t can be simulated by a null event as part of E. A simulated random walk in a regular grid in which a particle is moved with each clock tick by unit distance in one of four possible directions is an example of a time driven approach.

In event driven simulation, events are scheduled for various (future) instants of time at which they will execute. The system must maintain an *event list* into which all scheduled events are inserted. Associated with each event e is a time instant t_e at which the event is to occur. At all times, a partial order of events on the event list is maintained. That is, event e_1 will appear before event e_2 if $t_{e_1} < t_{e_2}$. If $t_{e_1} = t_{e_2}$ the order in which e_1 and e_2 appear on the event list can be left unspecified under the assumption that the corresponding changes of the network state are not visible instantaneously. In addition, this constrains new events to be scheduled not earlier than the current time T. Upon execution of an event e at time t_e, the global clock must be advanced to that time ($T := t_e$). The execution of event e may trigger the generation of new events which are then inserted into the event list as discussed above. A typical event driven simulation cycle is shown in Figure 1.

The simulation of a multi-agent environment such as a computer system, or a communication network lends itself to the event driven approach as

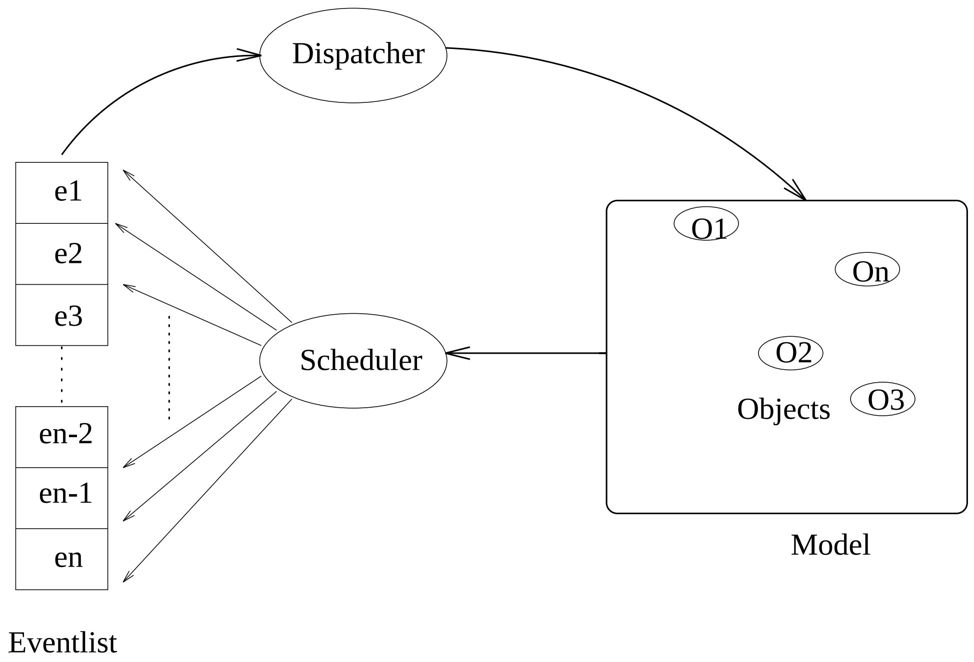

Figure 1: Event-driven simulation cycle

components tend to schedule their actions at various time instants.

The differences between the two approaches becomes apparent when considering the simulation of a large communication environment with many different agents (components). The set E of plausible events at time T is defined by the applicable actions at all agents to the system state at time T. Effectively, a time driven simulation would have to query each entity with every clock tick so as to determine whether or not an event needs to be executed. For most queries, the result is likely to be the null event since the occurrence of an event in the network is determined by parameters such as, traffic patterns and service rates. Time driven simulation of such an environment is thus inefficient and often infeasible. Hence, we have chosen an event driven approach for the simulation of a large communication infrastructure. The details of the design and implementation of our simulation testbed are discussed below.

3 Design and Implementation of a Simulation Toolbox

As mentioned above, the communication environment to be simulated may consist of several hundred network nodes connected by communication links. Messages are packaged into units consisting of actual message content and necessary protocol information and are transmitted on links between network nodes until the destination is reached. The functions of the communication network can therefore be expressed at the abstract level of nodes, links, and messages. In our implementation, nodes and links correspond to the active entities in the simulation model as they generate the necessary events that change the state of the network. The network model is embedded in the simulation testbed which maintains the event list and dispatches events to their corresponding entities for execution. The design and implementation of our simulation environment is described below.

3.1 The Object Oriented Approach

The decomposition of the simulation model into the functional components above suggests the application of the object oriented programming paradigm. Furthermore, employing the object oriented paradigm to the simulation of large communication networks yields advantages in the design as well as the implementation of both, the simulation testbed and the simulation model. The resulting structured design, significantly simplifies the implementation, particularly when functional modules are self-contained and communicate with other modules through well defined interfaces. Other advantages of employing the object oriented paradigm in the design and implementation of a simulation testbed are

- modularity,
- scalability,
- flexibility, and
- reusability

of the simulation environment.

Object classes together with the concept of inheritance provide for a very flexible framework which can meet the demands of many different simulation enterprises. The users can evolve the degree of detail of the various entities by gradually adding new features to the base classes. This can lead to an ever expanding simulation toolbox from which users can select the functional units necessary to implement their particular simulation environment. For instance, while network nodes are currently designed to support the analysis of a particular class of routing algorithms, the implementation can easily be tailored to support the simulation of other network management functions.

We refer to our design as an object oriented discrete event simulation (OODES), and we have chosen C++ as the implementation language [4, 9]. For the implementation of our OODES environment we distinguish the model under investigation from the simulation testbed which can be considered the host for the model. Clearly, model and testbed are interrelated as the model will determine the type of statistics which is to be collected. However, simple changes should suffice to modify the simulation testbed to host other models as needed. The interface between the simulation testbed and the model is realized through a status class through which information about the simulation model as well as the state of the simulation itself is made available to both, testbed and model. The structure of the status class is described below. The decomposition of the simulation environment into various functional components is shown in Figure 2.

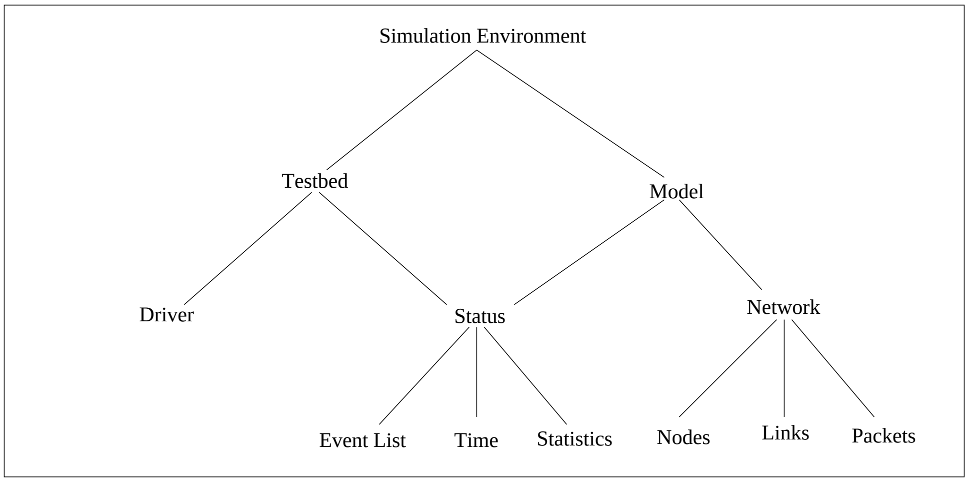

Figure 2: Decomposition Tree

3.2 The Simulation Testbed

The simulation testbed basically consists of two modules, namely the simulation status and the simulation driver.

3.2.1 Simulation Status

Upon starting the simulation, all definitions necessary to control the simulation are obtained from a parameter file provided by the user. Parameters in this file include all necessary information to dynamically build the network model. Other information provided through the parameter file such as, simulation time, and random number seeds is used to control the simulation. Various filenames are specified to allow the tracing of individual simulation runs. The status contains all necessary information to enable the simulation driver to execute the same model several times with different random sequences or different parameter settings, thereby providing for an automated execution of multiple experiments.

The most important item hosted by the status class is the event list. It represents the central nervous system of the simulation environment as it contains all events that are scheduled during the time of a simulation run. The list is currently based upon a simple linked-list implementation, however, other implementations (such as hashing) are possible and have been considered. Events on the event list are ordered at all time according to an execution time that is associated with each event. Parallelism in the model is simulated by executing multiple events at the same time. That is, the global simulation time is not being advanced until all events scheduled for a particular instant have been executed. The insertion and removal of events by elements in the model (nodes and links) and the simulation driver are described in detail in the following sections.

Most of the information that is maintained in the status class is used to define the behavior of an individual object within the simulation model. Provided through a parameter file, packet generation rates, service rates, link bandwidths, packet sizes, and thresholds are defined.

Located within the status class is the statistics module (Stats) which provides all necessary functions to simulate various statistical distributions for arrival and service rates as well as destination selection. For the purpose of our research we have implemented the uniform, Poisson, and exponential distribution. Additional distributions can easily be implemented as required by the respective simulation model. The performance monitor for the simulation is also located in this module. The performance monitor consists of functions that collect individual message statistics and functions that compute the various means, variances, and standard deviations. Since the status class is accessible to all modules, it can easily be extended to provide any function necessary to monitor the performance of the model under test.

3.2.2 Simulation Driver

At the heart of the simulation environment is a dispatching function which selects an event e_x with the smallest event time t from an event list L. Event e_x is then dispatched to the object (network node, link, or service routine) which originally scheduled this event to be executed at time t. As mentioned above, the event driven simulation strategy mandates events to be executed in strict order of their event time [3]. The event-driven simulation cycle is shown in Figure 1.

The implementation of this mechanism draws upon two object classes, namely a linked list and generic events.

The implementation of the event list can be based upon any list mechanism as long as it provides for insertion and ordered removal of elements. The class event-list can hence be derived from a basic list class modifying the associated insert function as required. In order to maintain an ordered event list, the append function used for queueing is replaced by an insert function which places events in their proper position in the event list. The concept of class templates, as available in C++, supports the design of a generic list class. As it is conceivable that different lists can store different types of items, the design of a generic list item has proven to be significant.

It should be noted that the underlying list representation together with the associated insertion function significantly impacts the overall performance of the simulation. Since the event list is expected to contain a large number of events at all time, search efforts for the appropriate place in the list for an event e_k can be large. Depending on the required performance of the simulator, the implementation of the event list may be based on mechanisms that accelerate the insertion of elements, such as hashing, to be used instead of a linked-list representation.

A generic event in the OODES is represented by a tuple of the form $< T, A, E, P >$ interpreted as follows:

- T: represents the time instant (with respect to the global simulation clock) at which the event is to be executed.
- A: designates the agent type i.e., network node, link, or service routine for which the event is scheduled and which will ultimately execute the event.
- E: identifies the event and thus the action to be performed by agent A upon executing the event.
- P: is a pointer to the instance of the agent type A for which the event is scheduled.

Associated with the Event-Class is a dispatch function which uses the event-type A to appropriately cast the pointer to the executing entity. It then dispatches the event to that entity by calling an *execute* function which must be provided by every simulation entity. The class definition for generic events is shown in Figure 3. Examples of typical event types in a network simulation are packet generation, dequeue and service function in nodes, and packet transfer in links. In addition, there can be a variety of service event such as check pointing or the invocation of a user interface that can be scheduled at various instants of time.

```
////////////////////////////////////////////////////////////////////////////////
// Event.h          //
////////////////////////////////////////////////////////////////////////////////

#ifndef INC_EVENT
#define INC_EVENT

class Event
{
friend ostream& operator<<(ostream &os, Event &e);

public  :
Event(void) ;
Event(Event&) ;
void Set(TIME time,EVENT_TYPE eventType,EVENT_NUMBER event,void *object);
TIME GetEventTime(void){return time;}
void Dispatch(void);
void Display(void) ;
int operator<(Event &e);

private :
TIME  time  ;
EVENT_TYPE eventType ; // Node event | Link event ...
EVENT_NUMBER  event ; // Service event | Generate pkt....
void  *object ; // points to a node|link ..based on eventType.
} ;

#endif // INC_EVENT

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

Figure 3: Event-Class header file

3.3 Implementation of the Model

The instantiation of the simulation model entails the representation of each individual entity whose behavior is to be simulated. A network environment is constructed with nodes and links as the active entities. Active entities in our model act on messages, which are considered passive. The representation of the network model based upon the implementation of nodes and links is discussed below.

3.3.1 The Network Class

The network class constitutes the module which defines the topology of the communication environment under test. Nodes and links are represented by corresponding data structures. Upon instantiation, the network-object reads two datafiles supplied by the user.

The first file to be read defines the set of network nodes by providing various parameters for each individual node. The minimum information necessary to define the existence of a particular node is its location in terms of x & y coordinates in a virtual two-dimensional grid together with a node-id. Upon reading the information for a particular node, a node-object is instantiated with the corresponding parameter values. The new node-object is then stored in a node-list, thus making this node available to the simulation environment.

The second file read upon network instantiation contains a description of the communication links between nodes. For the purpose of our research, a link is defined simply by a tuple $< n_i, n_j >$, where n_i and n_j represent the source and destination, respectively. Upon reading link-information from the file, a link class object is instantiated and stored in a link-list. Link-specific parameters, such as bandwidth, are assumed to be uniform throughout the simulation environment. For other simulation models it may be necessary to supply this information as part of the tuple read from the datafile.

The network class provides functions for accessing individual network nodes and links. As these functions are *public* they can be invoked from anywhere within the simulation. Pointers to the nodes and links of the network are stored in *nodeList and *linkList, respectively. Both are declared as private data members of the network class so as to restrict access to the respective access functions. Both, *nodeList and *linkList are based upon

```
////////////////////////////////////////////////////////////////
//                    Network.h                             //
////////////////////////////////////////////////////////////////

#ifndef INC_NETWORK
#define INC_NETWORK

class Network
{
public :

   Network(Status *statusPtr);
   ~Network();
   void PrintNet();
Node *GetNodeById(int nid)   {return (Node *)(*nodeList)[nid];}
Link *GetLinkById(int lid)   {return (Link *)(*linkList)[lid];}
void  ExecuteEvent(EVENT_NUMBER num, List *el) ;
void  CheckPoint(void);

//private:

   int NodeCnt;
   int LinkCnt;
   Status *status;
   List *nodeList ;
   List *linkList ;

   int  ReadNodeList(void);
   int  ReadLinkList(void);
   void InitNet(void) ;

} ; //Network

#endif // INC_NETWORK
```

Figure 4: Network-Class header file

a commonly used implementation of a linked list. Access to individual list members is defined in the list class. The header file for the network class is shown in Figure 4.

3.3.2 Network Nodes

As our simulation aims at the investigation of routing algorithms which are executed in the network nodes, this module displays the highest degree of complexity. In addition to the basic functional components of a network node the node class must provide the various structures and mechanisms necessary

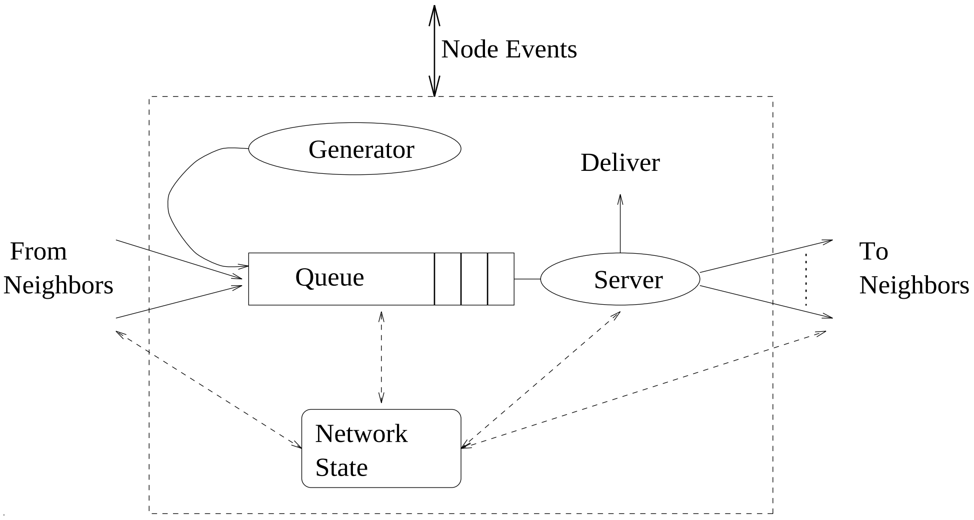

Figure 5: A Canonical Node Structure

for the various routing algorithms under investigation.

The canonical structure of a network node is based upon a single queue, a service function, and a message generator (see figure 5). Arriving messages are added to the queue and upon invocation by the event dispatcher, the service function removes the first element (message) from the queue and performs service in the form of routing. A *generate-message* event causes the message generator to create a message to a randomly chosen destination node. This message is then added to the node's queue. The structure of messages used in the simulator is discussed below. Message generation and service represent the fundamental events that are scheduled and executed by a network node. The rates at which messages are generated and serviced by the node is determined by the frequency at which these events are scheduled. This, in turn, is defined by the generation and service rates that are set upon instantiating a network node. The values for these rates are provided by the status class as described above.

Upon generating a new message, node n_i randomly determines the time δt until the next generation. A new *generate message* event is then scheduled at $T + \delta t$, where T represents the current simulation time. The time interval δt is exponentially distributed with a mean of $1/\lambda_g$ [6, 7, 16, 17]. Hence, the generation of messages is Poisson distributed with mean λ_g:

$$P_n(\Delta t) = \frac{(\lambda_g \Delta t)^n}{n!} e^{-\lambda_g \Delta t} \quad \Delta t \geq 0, \quad n = 0, 1, 2... \tag{1}$$

A node's service rate μ (in k $bits/sec$) remains constant for all messages that are serviced at this node. The time spent to service an individual message depends therefore on the size of the message. In order to simulate exponential service times, messages sizes are exponentially distributed. Thus, a network nodes displays the behavior of a single M/M/1 queue. Our approach to simulate an exponentially distributed service rate by the means of exponentially distributed packet size is motivated by the need to preserve the sequence of random numbers used for the simulation. This approach will be further discussed below.

Upon servicing a message, node n_i determines the time instant at which the next message is to be removed from its queue. In the current implementation, *post-scheduling* is used for scheduling the next service in a node n_i. The concepts of pre- and post-scheduling are discussed below.

Communication links to and from neighbor nodes are accessible through a list of pointers, each of which uniquely identifies a particular communication link. A node n_i can communicate with a neighbor node n_j through the corresponding outgoing or incoming links. Depending on the routing mechanism used, different types of network state information may be propagated along communication links in addition to the actual messages.

To determine the current load ρ (or queue utilization), a node must be able to measure the current arrival rate λ to the queue. Using its service rate μ, the node can derive the current load as:

$$\rho = \frac{\lambda}{\mu} \tag{2}$$

Clearly, different routing algorithm require different data structures in order to represent the state of the network upon which routing decisions can then be based. While random routing does not require any acquisition and propagation of network state information, other routing algorithms may require data structures that grow linearly with the size of the network. Distance vector and link state routing are examples of the latter.

One of the routing strategies used in our research is minimum distance routing. The network is based on a regular grid topology which enables nodes to determine their distance to the destination. A routing decision made by node n_i with respect to a message with destination node n_d would result in the selection of a neighbor node n_j which minimizes the remaining distance to n_d. Different distance metrics can lead to different routing decisions. Hence we have implemented both, euclidean as well as non-euclidean distance measures. Depending in the simulated network topology, some distance measures may be more applicable than others and can be implemented as necessary. An example of a non-euclidean distance measure is the Manhattan or city block distance

$$M_{i,j} = \mid x_i - x_j \mid + \mid y_i - y_j \mid \tag{3}$$

which is suitable for a regular grid topology.

The implementation of node class provides various functions to access private data members, including the necessary *set-* and *get-functions* to modify the rates at which node events are scheduled. These functions are used to dynamically alter the behavior of network nodes as a simulation experiment is conducted.

3.3.3 Communication Links

Like network nodes, communication links are also modeled as a single queue. The service provided by a communication links consists of the propagation of messages from its internal queue to the corresponding network node. A node n_i can forward a message via link $l_{i,j}$ to neighbor node n_j. If no other messages are currently queued, the time until $l_{i,j}$ delivers the message to n_j is determined by its corresponding link-bandwidth. If required, the queue in a link $l_{i,j}$ can be implemented so as to limit the available buffer space to a constant k. The $k+1$ message appended by n_i is then dropped. As the link bandwidth is assumed to be constant in $bits/sec$, the actual time spent to service a message depends on its size.

Link failure can easily be simulated by either setting the bandwidth to 0, or by removing the link entry in the corresponding node. In the first case, all messages will be lost (i.e., queued indefinately). In the second case, link $l_{i,j}$ will not be considered for routing, thereby removing node n_j from n_i's neighbor set H_i. In the current implementation of our model we have chosen the second approach. The burden to deal with link failure is thus given to the network nodes.

As for service events in network nodes, post-scheduling (see Section 3.4) is used to schedule the removal of messages from the link queue.

3.3.4 Messages

A message in our network model can potentially contain any number of information fields needed to execute the simulation and allow for an efficient statistical evaluation of the model. The values of these fields can either be determined at the time of message creation or be acquired and altered as the message propagates among nodes towards its destination. Clearly, messages used in the simulation model are an abstraction and should provide for the analysis of the model under test. The structure of the message class used for simulating traffic in a network environment does not necessarily coincide with the various protocol fields in actual messages.

The minimum information that should be available in a message is:

- Its identifier (ID), used to refer to different messages in the system;
- Sender and Receiver ID, used to make routing decisions;
- The message size, used to determine the service time at various queues;

Depending on the statistical variables to be analyzed other measures, such as hop-count and queueing time, may be accumulated as the message moves through the network.

3.4 Pre vs. Post Scheduling

Whenever a message arrives at a network node or link, it is appended to the central queue of the corresponding entity. In general, it is assumed that the message remains in the queue until service is completed. This implies that the message is serviced while it allocates the first position in the queue. Clearly, the completion of service, signified by the removal (or departure) of the message from the queue, constitutes an event in our OODES that must be scheduled. This *departure event* can be scheduled either upon message arrival or when the message enters service (i.e. enters the first position in the queue). We refer to the two strategies as *pre-* and *post-scheduling* respectively.

Let t_s^M be the service time for a message M. For pre-scheduling, the completion of service is scheduled as soon as the message is appended to the queue. Pre-scheduling requires queues to maintain information about the earliest possible time of service for a newly arriving message. Let T be the current simulation time and let t_{queue} be the earliest time an arriving message can be serviced. The completion and thus the removal of this message is then scheduled as an event for time t_e where

$$t_e = t_s^M + max(t_{queue}, T) \tag{4}$$

A node or link may determine not to schedule the event e for message M if $t_e \geq T_{max}$ (where T_{max} is the maximum simulation duration), as the event e is not going to be dispatched during the lifetime of the simulation. For the same reason, message M can be removed from the queue and thus memory resources are freed.

Pre-scheduling may be necessary when the corresponding server does not serve customers in strict first in first out (FIFO) order but instead interleaves the service on different customers. An example of such a system is a car repair service where customers are given a date and time when repair work on their car will be completed. The service provider, however, may choose to work on various cars in parallel, interrupt work, or postpone service on a car until the latest possible time. The earliest possible time for uninterrupted service t_{queue} might hence depend on a complex set of parameters $\vec{\Upsilon}$, representing the state of the system and possibly external events; i.e.,

$$t_{queue} = F(\vec{\Upsilon}) \tag{5}$$

Post-scheduling on the other hand does not require the service completion to be scheduled upon message arrival at a queue. Instead, the completion event is scheduled at the time when message M moves to the first position in the queue ready to be serviced. With the respective service time t_s^M the completion event is scheduled at:

$$t_e = t_s^M + T \tag{6}$$

Post-scheduling clearly reduces the number of events on the event list since it contains at most one completion event for any queue in the network. However, knowledge about whether or not an arriving message will ultimately be serviced during the simulation is not available. Hence, resources cannot be freed as with pre-scheduling. However, all messages in a queue can be deleted as soon as a completion event is scheduled at $t_e \geq T_{max}$.

We have chosen the post-scheduling approach in our simulation environment, as the primary objective is to simulate a large network environment with hundreds or even thousands of queues. The following example demonstrates the advantage of using post-scheduling vs. pre-scheduling in such an environment.

3.4.1 Example

Consider an 1024-node network arranged as a $m \times n$ grid with $n = 32$ and $m = 32$. With a regular grid topology, this network contains

$$2 \times [(m-1) \times n + m \times (n-1)] = 3968 \tag{7}$$

unidirectional communication links. Assuming that each of the network nodes and each of the links are modeled as single M/M/1 queues, this model thus contains 4992 queues.

If we assume an average queue utilization ρ of 0.66, the average number of messages in each of the queues is

$$\frac{\rho}{1-\rho} \approx 2 \tag{8}$$

If post-scheduling is in effect, there is at most one completion event for each queue on the event list. Thus, at most 4992 events need to be scheduled in addition to events such as message generation and state computation events.

With pre-scheduling, one completion event must be scheduled for each message in the system. Hence, in average there are 9984 completion events on the event list. As ρ grows larger, the total number of events generated by all entities clearly becomes prohibitive, as the overhead associated with creating events and searching for their appropriate place in the event list degrades the performance of the simulation testbed.

3.5 A Priori Service Time

Our simulation testbed has been designed to execute on a single PC or workstation. In such a computing environment there is generally only a single random number stream $RAND$ available. Message inter-arrival rates, service times, and message destinations are generally based on random variate generations with individual calls to $RAND$. In general, the number of calls to $RAND$ for an individual message to be created and to propagate through the network until it is delivered to its destination node depends on the number of queues encountered. This, however, is a function of the routing algorithm used. As our research requires the trace of an individual message and the comparison of results with various parameter settings, it is imperative that the same sequence of random numbers (obtained from $RAND$) is maintained. In general, this cannot be achieved if random variates are generated to determine the service time of a message M at every queue visited by M while propagating through the network.

In order to preserve a random sequence across various experiments, we have chosen to simulate exponential service at every queue, by assigning a deterministic service rate in $k\ bits/sec$, and to vary message sizes according to an exponentially distributed random variate. However, the disadvantage of this approach is the possible magnification of error in the approximation of the exponentially distributed random sequence. This error may nevertheless be acceptable if the experiment does not rely on the precision of this approximation.

A second possible approach to preserve the random sequence is to preassign each message an exponentially distributed service time upon message generation. This approach, however, deprives the user of the possibility to model queues with different service rates.

In either approach, all necessary calls to $RAND$ are thereby moved to the time of message creation thus preserving the same random sequence among experiments with different parameter settings.

3.6 Verification of the Simulation Environment

Various experiments have been conducted in order to verify the behavior of our implementation of the OODES and to identify its limitations. Since network nodes and links are based upon the implementation of a single queue, it is necessary to validate the behavior of these basic elements. Among other experiments, we have simulated the behavior of single M/M/1 queue and a single M/D/1 queue [2, 6, 7, 16].

A 1024-node network model is used to investigate the limitations and intricacies of simulating a large communication environment. These experiments as well as their results are described below.

3.6.1 Simulating M/M/1 and M/D/1 Queues

The simulation of a single queue in our OODES is achieved by simulating a communication network containing a single node. The message generator in this node can provide the necessary Poisson arrival of messages. The server can be manipulated to either determine the service time by using an exponentially distributed random variate (for M/M/1) on a per message basis, or by using an apiori assigned service time (for M/D/1) for all messages serviced.

Let $\overline{T}_m$ be the time at which message M enters the queue and let $\underline{T}_M$ be the time at which service on m is completed and M is removed from the queue. The total delay experienced by M is then given by:

$$D_M = \underline{T}_M - \overline{T}_M \tag{9}$$

Upon completing service on M, D_M is recorded by the corresponding function in the STATS-class of our OODES. After T_{max} seconds of simulation time, the average delay $\overline{D}$ over all messages that are recorded during the interval $[0.1\, T_{max}, 0.9\, T_{max}]$ is reported. From queueing theory it is known that the mean delay depends on the utilization ρ of the queue, i.e., $\overline{D} = f(\rho)$. From Equation (2) it is apparent that the load or utilization ρ depends on the mean message arrival rate of λ and the mean service time, $1/\mu$.

Using the theoretical results from queueing theory we compute the mean delay for M/M/1 and M/D/1 as:

$$E\left[D\right] = \frac{1/\mu}{1-\rho} \qquad (M/M/1) \tag{10}$$

$$E\left[D\right] = 1/\mu + \frac{\rho/\mu}{2*(1-\rho)} \qquad (M/D/1) \tag{11}$$

Our experiments involve the simulation of a single queue for 3600 seconds. The mean service rate is set to $10 msgs/sec$. In order to alter the queue utilization, ρ, the message generation rate λ (effectively the arival rate) is gradually increased from $1.0 msgs/sec$ to $9.5 msgs/sec$ in steps of $0.5 msgs/sec$. The mean message delay, $\overline{D}$ is computed for the various values of ρ, and compared to the theoretical derived expected message delay, $E\left[D\right]$. For each ρ, $\overline{D}$ is based on 50 model executions, each using a different random sequence. Tables 1 and 2 summarize and contrast the results of a single M/M/1 queue and a single M/D/1 queue, respectively.

ρ	$E[D]$	$\overline{D}$	$\frac{\lvert E[D]-\overline{D}\rvert}{E[D]}$
0.100000	0.111111	0.110758	0.003174
0.150000	0.117647	0.117730	0.000702
0.200000	0.125000	0.125196	0.001565
0.250000	0.133333	0.133354	0.000152
0.300000	0.142857	0.143110	0.001769
0.350000	0.153846	0.154131	0.001852
0.400000	0.166667	0.167745	0.006472
0.450000	0.181818	0.182129	0.001711
0.500000	0.200000	0.200482	0.002409
0.550000	0.222222	0.222451	0.001028
0.600000	0.250000	0.249536	0.001858
0.650000	0.285714	0.283663	0.007179
0.700000	0.333333	0.333133	0.000600
0.750000	0.400000	0.400757	0.001892
0.800000	0.500000	0.494867	0.010267
0.850000	0.666667	0.651828	0.022258
0.900000	1.000000	0.993463	0.006537
0.950000	2.000000	2.018213	0.009106

Table 1: Expected vs. actual message delay in an M/M/1 queue for different values of ρ

ρ	$E[D]$	$\overline{D}$	$\frac{\lvert E[D]-\overline{D}\rvert}{E[D]}$
0.100000	0.105556	0.105550	0.000053
0.150000	0.108824	0.108790	0.000309
0.200000	0.112500	0.112466	0.000305
0.250000	0.116667	0.116682	0.000133
0.300000	0.121429	0.121420	0.000074
0.350000	0.126923	0.127021	0.000768
0.400000	0.133333	0.133403	0.000525
0.450000	0.140909	0.140854	0.000394
0.500000	0.150000	0.150190	0.001269
0.550000	0.161111	0.161318	0.001282
0.600000	0.175000	0.175446	0.002550
0.650000	0.192857	0.192996	0.000720
0.700000	0.216667	0.216208	0.002117
0.750000	0.250000	0.248646	0.005417
0.800000	0.300000	0.298541	0.004862
0.850000	0.383333	0.378362	0.012969
0.900000	0.550000	0.544446	0.010098
0.950000	1.050000	1.035018	0.014268

Table 2: Expected vs. actual message delay in an M/D/1 queue for different values of ρ

The model is simulated for 3600 seconds for different values of ρ. The mean service rate for each queue remained at $\mu = 10 msgs/s$ throughout the experiment. It follows that the number of messages from which $\overline{D}$ is derived increases as ρ increases. The number of messages recorded in the 80%-interval $[0.1T_{max}..0.9T_{max}]$ is at least 2800. For both, M/M/1 and M/D/1 queues, the actual message delay is within 2.5% relative error,

$$\frac{\mid E[D] - \overline{D} \mid}{E[D]}, \tag{12}$$

despite the fact that the exponential random variate only approximates the true exponential distribution. The behavior of an individual queue is thus deemed sufficiently precise to be used for the construction of larger network model.

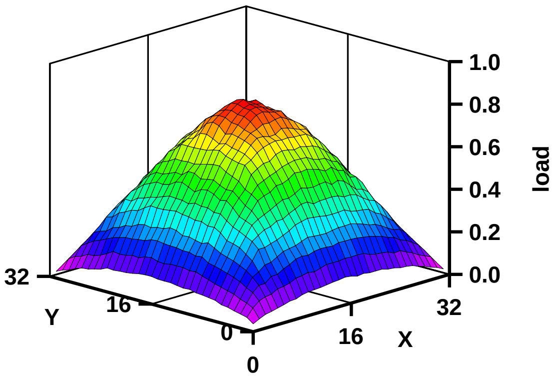

Figure 6: Load Distribution in a 1024 node grid network using Shortest Path Routing

3.6.2 Simulation of a Large Network

To assess the performance of the OODES when executing large, complex models, we have simulated 300 seconds of traffic in a 1024-node network [14]. The nodes are arranged in a 32×32 grid topology. In this model, each node is implemented as a single M/M/1 queue. Each node generates messages at a rate of $0.3 msgs/s$. The destination for each message is chosen at random, self-traffic, however, does not occur. The mean service rate for each node remains static at $20 msgs/s$ throughout the course of the simulation. It is further assumed that communication links between nodes have sufficient bandwidth so as to regard transmission delays as negligible. Message delays are thus assumed to be caused solely by queueing delays encountered in network nodes.

In order to propagate a message from its source to its destination, each node follows the same routing algorithm. The algorithm will select among all neighbor nodes the one which minimizes the remaining euclidean distance to the destination. The load distribution in the network is therefore a function of the routing algorithm used. The respective load landscape is shown in figure 6.

Depending on the system load, the simulation of 300 seconds of network traffic as described above is executed in approximately 4400 seconds on a

HP 712/80 workstation. The utilization of individual queues clearly depends on the routing algorithm used in the model. Hence, it is difficult to predict whether or not there will be queues that experience infinite queueing (i.e., $\rho > 1.0$). It may indeed be necessary to rely on the result of short time sample execution of the model or intermittent sampling of the nodes' utilization. The consequence of infinite queueing in multiple nodes can result in an unbounded resource requirement which may be impossible to be satisfied during the simulation time. However, it may be acceptable to permit a small set of nodes to enter the state of infinite queueing if the resource demand is bounded by the simulation time.

4 Summary

In the simulation of large communication networks there clearly exists a tradeoff between the size of the model that is to be simulated and the computational resources needed to run simulation experiments. As the size of the network under study grows, the resource requirements often become prohibitive. Hence it is necessary to enable the user to control the degree of detail at which various aspects of the communication network are modeled.

While off-the-shelf simulation packages are generally sophisticated and versatile enough to support modeling of a wide range of different communication environments, most of them do not provide the user with sufficient control over the degree of detail at which the individual entities are modeled. This severely limits the use of off-the-shelf simulation packages in the experimental study of very large communication networks containing hundreds or thousands of nodes.

Given the primary focus of our research on efficient adaptive heuristic routing algorithms for very large communication networks, it was necessary for us to design and implimient a modular, object oriented simulation toolbox to support our research. In the resulting toolbox which is the subject of this paper, modularity and inheritance enable users to tailor the individual entities that make up the system to incorporate varying amount of detail as needed.

Since event-driven simulation offers several major advantages in terms of efficiency over time-driven simulation in modeling the dynamic behavior of large systems in which events occur asynchronously, the toolbox was implemented using the discrete event-driven simulation paradigm. Further efficiency gains were obtained (in terms of the size of the event list to be maintained), by using post-scheduling as opposed to pre-scheduling.

Since the main purpose of our simulation toolbox was to support systematic experimental investigation of the dynamic behavior and performance of large communication networks, reliability and validity of the results of simulation was a primary concern that needed to be addressed. This task is further complicated by numerous calls to random the number generator, and the complex interactions among differeent individual entities within the system. We therefore relied on the verification of the individual components of the simulation under experimental conditions for which theoretical calculations can be used to validate the system. This, along with a careful checking of the interactions among entities was used to validate the simulation toolbox.

Acknowledgement

The authors would like to thank Deepa Parthasarathy and Todd Campbell for their contribution to the design of C++ object classes used in the simulation toolbox discussed in this paper. Vasant Honavar would like to acknowledge the support of National Science Foundation (grant IRI-9409580) and the Iowa State University College of Liberal Arts and Sciences during this research.

References

[1] Alaettinnoglu, G., Shankar, A.U., Dussa-Zieger, K., et al. Design a nd Implementation of MaRS: a Routing Testbed. *Internetworking: Research a nd Experience* Vol. 5, (1994): 17-41.

[2] Bertsekas, D. and Gallager, R. *Data Networks*. 2nd ed. Prentice-Hall, 1992.

[3] Cassandras, C.G. *Discrete Event Systems - Modeling and Performan ce Analysis.* Richard D. Irwin, Inc., and Aksen Associates, Inc., Boston.1993 .

[4] Deitel, H.M. and Deitel, P.J.*How to Program C++.* Prentice-Hall, Englewood Cliffs. 1994.

[5] Fujimoto, R.M. Parallel Discrete Event Simulation. Communications of the ACM Vol.33, No.10 (1990):30-53

[6] Gross, D. and Harris, C.M. *Fundamentals of Queueing Theory.* 2nd ed. John Wiley & Sons, New York.1985.

[7] Jain, R. *The Art of Computer Systems Performance Analysis.* Wiley & Sons, Inc., New York: 1991.

[8] Lehmann, F., Seising, R., and Walther-Klaus, E. Simulation of Learning in Communication Networks. *Simulation Practice and Theory* 1, No.1 (1993): 41-48.

[9] Lippman, S.B. *C++ Primer.* 2nd ed. Addison-Wesley, Reading, MA.1991.

[10] Mikler, A.R., Honavar, V.G., and Wong, J.S.K. Simulating a Traveller: A Heuristic Approach to Routing in Large Communication Networks. *Proceedings of the European Simulation Symposium, ESS '92* November 1992 (1992): 297-301.

[11] Mikler, A.R., Wong, J.S.K., and Honavar, V.G. Quo Vadis – A Framework for Adaptive Routing in Very Large Communication Networks. *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications; (Alspector,J, Goodman,R ,Brown,T X ,Ed)* (1993): 196-202.

[12] Mikler, A.R., Wong, J.S.K., and Honavar, V.G. Quo Vadis - A Framework for Intelligent Traffic Management. *Proc of the ISMM International Conference on Intelligent Information Management Systems* (1994): 25-28.

[13] Mikler, A.R., Wong, J.S.K., and Honavar, V.G. Quo Vadis - A Framework for Intelligent Routing in Large Communication Networks. Technical Report TR94-24 Department of Computer Science, Iowa State University. December (1994). To appear in: *Journal of Systems and Software.*

[14] Mikler, A.R., Wong, J.S.K., and Honavar, V.G. Quo Vadis - Adaptive Heuristics for Routing in Large Communication Networks. *Proceedings of the 3rd International Conference on Telecommunication Systems, Modeling and Analysis* (1995): 66-75.

[15] Ni, L.M. and McKinley, P.K. A Survey of Wormhole Routing Techniques in Direct Networks. IEEE Computer Vol.26, No.2, (1993): 62-76

[16] Robertazzi, T.G. *Computer Networks and Systems: Queueing Theory and Performance Evaluation.* Springer Verlag, New York: 1990.

[17] Ross, S.M. *Introduction to Probability Models.* 4th ed. Academic Press, INC., San Diego.1989.

[18] Snyder, J.M. A Routing Architecture for Very Large Networks Undergoing Rapid Reconfigurations. *Proc ACM SIGCOMM 1989* (1989): 57-63.

[19] Tanenbaum, A.S. *Computer Networks.* Prentice-Hall, 1988.

[20] Tanir, O. and Sevinc, S. Defining Requirements for Standard Simulation Environment. *IEEE Computer* Vol.27, No.2, (1994): 28-34.