

A Flexible Object-Oriented Design of an Event-Driven Wireless Network Simulator

Imad Jawhar

College of Information Technology

UAE University, Alain, UAE

Email: ijawhar@uaeu.ac.ae

Abstract—Different simulation tools are available to the research community which can be used to validate, test and evaluate networking protocols. Such simulation software packages include OPNET, ns-2 and others. These programs are capable of performing relatively accurate simulations of networking architectures including wireless, ad hoc, and sensor networks. Although they are very useful for a wide range of experiments which involve performance evaluation of existing protocols, selecting them to verify and test the performance of new platforms and protocols that are not implemented in these programs tend to impose considerable restrictions and limitations. Therefore, implementing these protocols in existing simulator packages could be relatively difficult and time consuming. This paper presents an object-oriented framework to design an event-driven network simulator that can be used to evaluate different wireless networking architectures and protocols. The design of the simulator allows for a considerable increase in flexibility for the researcher who is able to include particular aspects and implementations of the corresponding protocols. The different classes that constitute the simulator application along with the main methods, and relationships are presented. In addition, design strategies for simulating node mobility using the popular Waypoint model are also presented.

Keywords: Wireless ad hoc and sensor network, network simulation, simulator design, object-oriented design.

I. INTRODUCTION

In response to the continued evolution and technological advancements in the area of communication and networking, researchers are constantly designing new protocols and architectures in order to support emerging applications and platforms. An essential part of the research methodology is the simulation of newly created network models and protocols. Currently, there exist several network simulation packages such as OPNET [1], ns-2 [2], and others. These packages can be reasonably easily used to study the performance of many networking protocols and platforms they support. However, practice has shown that implementing new protocols which are significantly different from the ones which are supported using these simulation packages is very difficult. Consequently, many researchers are faced with the need to design their own simulator software in order to implement and study the performance of new network protocols and platforms which are significantly different from the ones supported by these off-the-shelf packages. This is a good alternative as long as the researcher is able to design this simulator in a

reasonably short period of time and use the proper design and validation techniques. This paper, presents architecture and design techniques of a typical event-driven wireless network simulator which uses the object-oriented design methodology.

A. The ns-2 network simulator

As mentioned earlier, one of the most popular network simulation packages is the ns-2 (network simulator, version 2) [2]. It is an open source software that is available on the Internet [2]. It can be used for simulation of different algorithms and protocols such as IP, TCP (Tahoe, Reno, Newreno, SACK, Vegas), UDP, Ethernet, ARP, 802.11, DSDV, AODV, TORA, DiffServ, IntServ, and mobile ad hoc networks [3]. It also supports different queueing techniques such as Drop tail, RED, FRED, REM, DRR, and SFQ. The wireless and mobility extensions to ns-2 are a result of the Monarch project. A good number of researchers have contributed code that is used with ns-2. Although this simulator is text-based and does not have GUI capabilities for creating topologies and interfaces, it has features that allow the graphic visualization of the network and node activities such as transmission and reception of packets. The ns-2 simulator contains two sets of languages OTcl (Object-oriented extension of the Tool Command Language) and C++ [2][4]. The OTcl language is used for setting up the simulation, configuring objects, and scripting simulation topology and events. Specifically, OTcl is used to do the following: (1) Allows the user to perform some simulations by varying different parameters and configurations. (2) Investigate different scenarios. (3) Determine the simulation model by setting parameters such as the number of iterations. On the other hand, the C++ language can be used for programming each object in the simulation structure and topology. Specifically, it can be used for the following reasons: (1) Create new objects that might be needed in some specialized simulations of new protocols that are not supported. (2) Increase the speed and efficiency of the simulation process. (3) Perform byte manipulation, packet processing, algorithm implementation, or change the actions performed by different modules.

The ns-2 simulator provides some advantages due to its existence as an open source code [5]. It allows building upon existing work, and allows others to use the researcher's work. However, it also has the disadvantage of being huge and complicated. In addition, it has relatively poorly maintained documentation and the possibility of sometimes having missing

This work was supported in part by UAE Research grant 03-03-9-11/08.

components and unexpected bugs which can be significantly time consuming to discover, isolate, and fix.

B. The OPNET network simulator

OPNET (Optimized Network Engineering Tool) [1] is another available network simulation package. It is one of the leading existing network simulators in the market. The OPNET simulation software costs in the order of several thousand dollars for a single license. Although free licenses are available for educational purposes, it provides a graphical user interface that allows the user to create networking models using a drag and drop approach. It is designed using a discrete event driven simulation approach and applies an object-oriented design methodology [6]. In addition, OPNET uses a hierarchical modelling strategy that allows the user to choose the desired specifications and protocols including physical layer components such as transceivers, antennas, queue management, nodes with process modules, and a networking model that is used to connect them. In addition, packet formats that are to be used with the communication protocols can be programmed into the model. The hierarchical model building has the following modules [7]: (1) Network Editor which allows the user to build the network topology, (2) Node Editor which is used to define data flow models, (3) Process Editor which is used to specify control flow models. Through these editors, the user can specify models at three different levels: the network domain, the node domain, and the process domain [8]. Modelling in the network domain can be used to hide the complex structure of the lower level component which would be invisible to the user. For running simulations, it has a simulation tool to define and run the simulation, and a debugging tool. For results analysis, it has a probe editor to specify the points where data needs to be collected, an analysis tool, data filtering tool, and an animation viewer to display the behavior of the network in case that is needed. OPNET operates at the packet level, and has a large library of models for existing network hardware and software protocols. It allows the running of external code components through its External System Domain (ESD) tool.

C. Other network simulators

In addition to the simulations mentioned above, there are other more specialized network simulators such as REAL [9][7], which is used to study congestion control schemes in packet switched data networks, INSANE [10], which is used to simulate IP-over-ATM algorithms with traffic loads that are derived from real traffic measurements, NetSim [7], which is used for simulation of Ethernet networks, and Maisie [11] which uses a C-based language for using parallel discrete event simulations. In Maisie, a logical process is used to model one or more physical processes. The events in the physical system are implemented using message exchanges among the corresponding logical processes in the model. Additionally, Parsec and MOOSE are object-oriented extensions to Maisie.

The rest of the paper is organized as follows. Section II presents an overview of the event-driven simulator design. Section III discusses each of the classes that are used. Section

TABLE I
PARAMETERS FOR THE RACE-FREE PROTOCOL SIMULATION

Parameter	Value
Network Area	$600 \times 600 m^2$
Number of Nodes	30
Transmission Range	150 m
Data Packet Size	512 bytes
Number of Sessions	50
Maximum Session Time	4000 s

IV presents the simulation process and its phases, and the last section provides the conclusion.

II. EVENT-DRIVEN SIMULATOR DESIGN OVERVIEW

It is important to note that the simulator design presented in this paper was implemented using the C++ object-oriented language in order to evaluate and test two quality of service (QoS) routing protocols for mobile ad hoc networks with omnidirectional and directional antenna environments and the corresponding results were published in related papers [12][13]. The simulator is event-driven and is implemented using an object-oriented design approach. It starts by creating the required simulation objects including the area, nodes, event priority queue (EPQ), and graph objects. It loads the EPQ with the initial data messages with each message having source, destination nodes, arrival time, and message length as the main parameters. It then starts the simulation process by deleting events from the EPQ according to their time of occurrence, executing them by updating the required objects and keeping the required statistics, and inserting any newly generated events into the EPQ. This process continues until the EPQ is empty. Then, the program outputs the simulation results to the user.

A. A case study: the simulation of a typical wireless ad hoc and sensor network

In order to illustrate the design of the event-driven simulator, a simplified case study is presented. The case study involves the simulation of a DSR-Based routing algorithm and the study of its performance in a mobile ad hoc network [13][14][15]. An ad hoc network is a wireless network where the nodes do not have a pre-existing infrastructure. It has a variable topology where the nodes join, or leave the network anytime [12][16][17][18]. Also, nodes can be mobile which means that the connectivity of the node with its neighbors as well as the topology of the network is variable. New communication links between nodes are created and existing ones are deleted as nodes move from one location to another. Each node in the network plays the role of an end point as well as a router in the data transmission process. Communication between source and destination nodes is established using multi-hop routes passing through intermediate nodes.

B. A simplified overview of a DSR-based routing process

The routing protocol that is used in this case study is based on the Dynamic Source Routing (DSR) protocol [14]. It is on-demand which makes it more scalable. Nodes do not need to

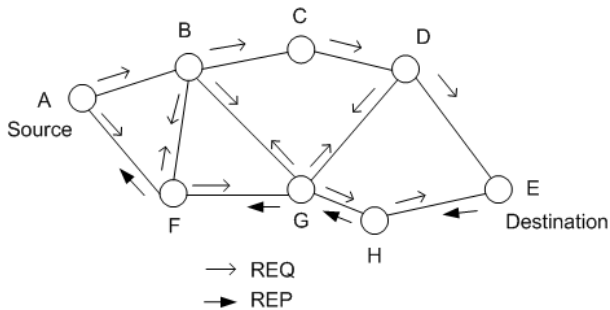


Fig. 1. A simplified example of the route discovery process.

keep information about the entire topology, and routes are only discovered as the need arises. When a source node s wants to send data to another destination node d which is not within its transmission range, it will try to discover a multi-hop path to it. In order to do that, node s broadcasts a request (REQ) message to all of its neighbors. Each of the neighbors adds its ID to the accumulating path in the message and in turn forwards it to all of its neighbors. This process continues until the REQ message reaches the destination, which then unicasts a reply (REP) message back to the source. Upon receiving the reply message, the source updates its routing table and starts the data transmission process.

A simplified example of the route discovery process is shown in Figure 1. In the figure, node A is a source node that needs to send data to a destination node E. Node A checks its routing table and realizes that it does not have a path to E. Consequently, node A starts a route discovery process by broadcasting a REQ message to its neighbors B and F. Nodes B and F, not having processed a REQ message with the same (s, d, ID) tuple, check the information in the REQ message and realize they are not the destination and that they do not have a path to E. Each of them forwards the REQ message to its neighbors except for the node from which they received the request (i.e. A). This process continues until the REQ message reaches the destination E. One or more REQ messages might reach the destination. If more than one REQ message reaches the destination, a certain criteria might be applied by the destination in order to select one of the discovered paths. In our case, the destination responds only to the first REQ message it receives, thereby choosing the earliest discovered paths. Assuming that the REQ message propagating along nodes A, F, G, H, and E arrives first, the destination unicasts a REP message with the discovered path $A - F - G - H - E$ to A. Subsequently, the destination simply drops any subsequent REQ messages that might arrive later through different paths. Upon receiving the REP message, node A updates its routing table and starts the data transmission process along the discovered path.

C. A sample of simulation parameters

Table I shows a sample of simulation parameters for the case study network. The number of nodes, n , is 30. The nodes are contained in an area of $600 \times 600 \text{ m}^2$. The range of each node is set to 150 m. The message length is randomly selected

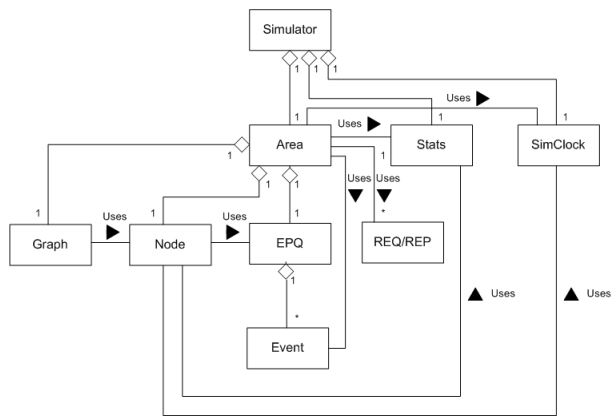


Fig. 2. A structure chart of the simulator.

according to a uniform distribution with a range from 0 to 1000 Mbytes corresponding to a session length range of 0 to 4000 sec. In some simulation experiments, the traffic rate could be varied to study its impact on different network performance measures such as throughput and path acquisition delay. The session (or data message) arrival is a Poisson process with a mean which could be varied as needed.

III. THE CLASSES

Figure 2 shows a structure chart of the simulator design which includes all of the classes and their relationships. The main class is the *simulator* class which includes the *main()* method. Also, it includes one of each of the following objects: an area, a stats, and a simClock object. In turn, the *area* class includes a graph, an array of nodes, and an *event priority queue (EPQ)* class. The *EPQ* class includes an array of events objects. The chart also includes the *REQ/REP* class which is used by the *area* class. All of the classes in the simulator, along with their functions, and relationships are discussed in this section.

A. Simulator class

This class has the *main()* method. The *main()* method instantiates the other objects in the simulator such as the nodes, EPQ, graph, and so on. It also has the method which loads the EPQ with the initial events. In most simulation cases, these events consist of the initial data transmission messages which occur at certain times according to the desired random distribution function. As indicated earlier, they are typically randomly generated according to a Poisson arrival distribution with an average arrival rate.

B. Area Class

One area object is created from this class. It holds most of the other objects in the simulation including the array of nodes, an event priority queue, and graph.

C. Node Class

The array of node objects is instantiated from this class. Each node contains the basic attributes such as: node ID, x and y geographic location coordinates of the node, and all tables that each node might need. Depending on the simulation, and network protocol requirements, these tables might include the routing table, cached information tables, tables that might hold status of different node communication parameters such as slot reservation status in a time division multiple access (TDMA) environment, and so on.

D. Event Priority Queue (EPQ) Class

This is a priority queue of events. The key field in each event that is used for ordering the events in the queue is the time. The time is an integer number which indicates the time of occurrence of the event. The main interface methods in the *EPQ* class are the *insertEvent()* and *deleteEvent()*. In order to increase the efficiency of the process of inserting and deleting events, a heap structure is used to implement the queue [19]. This makes the time complexity for insertion and deletion as well as most other queue processing functions $O(\log(n))$.

E. Event Class

The event objects that are instantiated from this class are the events of the event-driven simulation. Each event has the following main attributes:

- 1) **Event time:** An integer number which is the time stamp of each event.
- 2) **Event type:** An integer number which represents the event type. Examples of event types are: *send request message*, *receive request message*, *send data message*, *receive data message*, etc.
- 3) **Source node ID:** This is an integer number representing the ID of the source node that is sending a route request or data message for example.
- 4) **Destination node ID:** This is an integer number representing the ID of the destination node with which a particular source node wants to communicate.
- 5) **Session ID:** This is an integer number representing the message or session ID. For example, each node can have a running counter which generates the message IDs of the messages it generates.
- 6) **Other attributes:** Additional attributes are typically added, which are specific to the particular networking simulation that is required.

F. Stats Class

This class is used to keep track of the statistics and results of the simulation. It contains attributes such as number of successfully transmitted messages, number of unsuccessfully transmitted messages, number of dropped packets, number of collisions, and so on. Each attribute has accessor and mutator methods to read and update the corresponding variable. The *stats* class also contains a *printResults()* method which prints the required results at the end of the simulation experiment.

G. Graph Class

Typically, one graph object is instantiated from this class. It is a graph data structure. The structure could be using an adjacency matrix, or an adjacency list [20][21][22]. This graph has the nodes in the network as vertices, and the one-hop communication links between the nodes as edges. The graph is constructed by first making each node in the network a vertex in the graph. Then, the communication range, which is a simulation constant, along with the geographic coordinates of the nodes are used to determine the edges between the vertices in the graph. Edges are added between the nodes that are within range of each other. Specifically, an edge is added if the Euclidian distance between two nodes is less than or equal than the corresponding communication range. Later, in the simulation, the graph is used to determine the neighbors of each node by calling the *getNeighborList(x)* method which returns a list of the neighbors of a node *x*. This information is used to propagate route requests, route replies, data transmission messages, as well as any other types of communication between the nodes.

H. REQ/REP Class

Depending on the kind of network and simulation that is being designed, this type of classes might be needed. For example, in the case study presented earlier, which involves the simulation of the routing and data communication protocols in a wireless ad hoc and sensor network, a request (REQ) message is broadcast by a source node to all of its one-hop neighbors to initiate the route discovery process. The request message propagates through the network until it reaches the desired destination node which then responds by unicasting a reply (REP) message back to the source. When the source receives the reply message, it updates its routing table with the discovered path and starts the data transmission process. In this kind of networks, each one of the *REQ* and *REP* classes would contain attributes such as the source ID, destination ID, message ID, and a linked list of the IDs of the accumulated nodes along the discovered path.

I. SimClock Class

This is a very simple class which is used to keep track of the current time in the event-driven simulation. It has only one integer attribute named *time*. It also mainly has two methods: *getTime()* which is used to return the value of the time integer, and *setTime()* which is used to set the value of the same variable. Initially the simulation starts at time 0. As the next earliest event is retrieved from the event priority queue, the time of the simulation clock is set to the time stamp (time of occurrence) of that event. Note that because this is an event-driven simulation, this is the only way the simulation time is advanced. This is one of the main powerful features of an event-driven simulation compared to a time-driven simulation. Only instances of time concerned with the occurrence of an event are simulated as opposed to simulating each tick of the clock. Consequently, the simulation time can advance in large steps skipping all of the time when no events happen. This

method of handling time significantly increases the efficiency of the simulation process.

IV. THE SIMULATION PROCESS AND ITS PHASES

This section describes the main steps and functions used in the simulation process. Basically, the simulator starts by generating an area with certain dimensions and randomly places a predetermined number of nodes in the area. Generating the area object automatically instantiates all of the objects inside it including the event priority queue and the array of nodes. Then, the *shuffleNodes()* method of the *area* class is used in order to randomly shuffle the geographic locations of the nodes (x- and y-coordinates) within the designated geographic area. As indicated earlier, the nodes have a predetermined transmission range. From the placement of the nodes, a graph is generated which includes the nodes as vertices. Edges in the graph are constructed between the nodes/vertices that are within range of each other. Then the simulator generates a number of data messages with a certain length for each message. The length of the messages is generated randomly according to a certain desired distribution function. Each message has a randomly generated source and destination pair. The arrival times of the messages is according to a Poisson process with a certain mean inter-arrival time. As the messages are generated, the corresponding events with the *send data message* type are inserted in the event priority queue. When the *send data message* event is retrieved from the queue according to its arrival time, the corresponding source node initiates the route discovery process by sending a REQ message to all of its neighbors as indicated earlier. This is done by inserting corresponding events into the queue with a *receive request message* type and a time stamp equal to the current time plus the total request message transmission time. The REQ message is propagated in this manner through the intermediate nodes until it reaches the destination node. Then, similarly a REP message is propagated back to the source along the discovered path. Each node has a routing table as well as all of the tables needed for the routing algorithm. When the source receives the REP message, it updates its routing table with the discovered path and starts the data transmission process. All of the indicated activities are done by insertion and extraction of the corresponding events into and from the event priority queue. Stated in other words, the simulation starts with the initial events in the event priority queue. Events are extracted according to their time stamp. As each event is processed it might generate additional events which are inserted into the queue to be processed at a later time. This process of extracting, processing, and inserting events continues while updating the corresponding parameters and statistics in the stats object, until all of the events are processed, and the priority queue is empty. At this point, the simulation is ended, and the program outputs the results to the user.

Consequently, the main loop in this simulation is the one that extracts events from the EPQ and processes them one by one. This while-loop, which is considered the event processing engine of the simulation, is in the *main()* method in the *simulator* class. Algorithm 1 presents the main simulation

Algorithm 1 The main simulation algorithm

```

instantiate area object. Automatically instantiates EPQ, graph,
and array of nodes objects.
instantiate simClock object and initialize clock to 0.
load EPQ with initial SEND_DATA_MESSAGE events.
call area.shuffleNodes() method to shuffle node locations.
call area.generateGraph() method to construct new graph
from node locations and ranges.
while EPQ is not empty do
    retrieve next earliest event e from EPQ.
    eType = e.getEventType()
    if eType = SEND_DATA_MESSAGE_FIRST_TRY
    then
        insert event SEND_DATA_MESSAGE_TRIES to at-
        tempt to discover path in several tries.
    else if eType = SEND_DATA_MESSAGE_TRIES
    then
        Call e.curNode.checkRoutingTable(s, d) to see if node s
        already has path to destination node d.
        if path exists then
            insert event SEND_DATA_MESSAGE.
        else
            try to discover path by inserting event
            REC_REQ_MESSAGE for each neighbor node.
            insert another event SEND_DATA_MESSAGE_TRIES
            to try again later after path is discovered and inserted
            into routing table.
        end if
    else if eType = SEND_DATA_PACKETS then
        insert event REC_DATA_PACKETS for next node in
        path.
    else if eType = REC_DATA_PACKETS then
        if current node is destination then
            increment number of successful receptions statistics vari-
            able.
        else
            insert event SEND_DATA_PACKETS for next node
            in path.
        end if
    else if eType = REC_REQ_MESSAGE then
        if current node is destination then
            insert event REC_REP_MESSAGE for last node be-
            fore destination in path.
        else
            insert event REC_REQ_MESSAGE for next node in
            path.
        end if
    else if eType = REC_REP_MESSAGE then
        if current node is source then
            update routing table for current node.
        else
            insert event REC_REP_MESSAGE for node before
            current node in path.
        end if
    else if eType = MOBILITY_UPDATE then
        call area.updateAllNodePos() method to update all node
        positions with new locations.
        insert event MOBILITY_UPDATE with
        time stamp equal to (simClock.getTime() +
        MOBILITY_TICK_TIME).
        call area.generateGraph() method to construct new graph
        from new node locations and ranges.
    else
        print error "Unknown event type"
    end if
end while

```

steps which consist of the initialization phase, followed by the event processing loop, and the printing of the results.

A. Simulation initialization

As illustrated in Algorithm 1, the simulation starts by creating the area object which includes the event priority queue (EPQ), and array of node objects. Also, the simulation clock is created and initialized. Then, the EPQ is loaded with the initial events which consist mainly of a number of *send data message* events. Each message has a *source*, a *destination*, a *message ID*, an *arrival time*, and *message length*. The *source node ID*, *destination node ID*, and *message length* are randomly created from a uniform distribution. The arrival time is also a random number generated according to an exponential distribution with a given average arrival rate. This creates Poisson arrivals which is typically the most realistic assumption for the message arrival process, and is used in most simulations. Then, the *area.shuffleNodes()* method is called to assign random geographic coordinates for each node within the area dimensions. The *area.clearAndGenerateGraph()* method is called to generate the graph from the node locations and node transmission range. Nodes are represented by the vertices of the graph and edges are created between nodes that are within range of each other.

B. Event processing

After the initialization phase is completed, the main while-loop, which constitutes the event processing engine is executed by extracting, and processing the event from the event priority queue. Events are retrieved according to their time stamp starting with the earliest event.

C. Printing of the results

As the simulation is performed, different simulation and performance measures are maintained in the corresponding variables in the *stats* class. These variables are updated constantly during the running of the program and are printed at the end of the simulation.

D. Mobility simulation

In some networking applications such as mobile ad hoc networks, node mobility might be required. Node mobility can be simulated using different mobility models. In the research, the Waypoint model is the most popular model [23]. In this model, each node is assigned a particular random speed between 0 and *MAX_SPEED* in meters/sec. Each node is also assigned randomly generated starting and ending geographic coordinates within the area. Then, each node starts moving from its starting geographic position towards its ending position at its assigned speed. When the node arrives at its ending position, it pauses for a *PAUSE_TIME* seconds. The *PAUSE_TIME* is a simulation parameter chosen by the researchers to characterize the mobility of the nodes. A low value for this parameter along with a high value for the *MAX_SPEED* parameters indicate a highly mobile environment. After the expiration of the pause time, the node is assigned a new random speed as well as

a new geographic destination. Then, the node starts to move again at the new speed towards the new destination. When it arrives at the destination, it pauses and starts moving again, and so on. This process is repeated for each node until the end of the simulation experiment.

In the simulator design, a *MOBILITY_UPDATE* event is executed every *MOBILITY_TICK_TIME* seconds. During the initialization phase of the simulation, a *MOBILITY_UPDATE* event is inserted into the EPQ. When this event is executed, it inserts another *MOBILITY_UPDATE* event at the current time + *MOBILITY_TICK_TIME*. Also, new node location coordinates are calculated for each node using the current geographic coordinations, node speed, and direction. The nodes are then placed at their new locations, and a new graph is generated again using the node locations and transmission ranges. This mobility update results in new links being formed or deleted between neighboring nodes. The size of the *MOBILITY_TICK_TIME* should be selected in a manner that is appropriate for the required movement precision considering the value of *MAX_SPEED*. Obviously, a smaller *MOBILITY_TICK_TIME* value produces higher precision in the node mobility simulation. However, this increase of the precision in the mobility simulation comes at a cost of increasing the total simulation running time.

V. CONCLUSION

Currently, there exist several network simulation tools such as OPNET, ns-2, and others which are available to researchers to validate, test and optimize networking protocols. These simulators are useful for a wide range of simulation experiments which involve performance evaluation of existing protocols they support. However, selecting them to verify and test the performance of new platforms and architectures which are not implemented in these packages may impose considerable restrictions and limitations on the researcher. This paper presented the architecture and design of a typical object-oriented event-driven network simulator. The main classes, methods, and algorithms that are used in the simulator were offered along with a case study which illustrates a typical implementation of the presented model. Researchers can use this framework as a guide to design a network simulator which can be used to easily and accurately implement, validate, evaluate, and optimize new networking architectures, and protocols which may not be supported by the present simulation tools.

REFERENCES

- [1] The official OPNET company website. <http://www.opnet.com>.
- [2] The network simulator - ns-2. <http://www.isi.edu/nsnam/ns/>.
- [3] Woongsup Lee. Network simulators: Opnet, ns-2. http://cnr.kaist.ac.kr/lecture/te523_2008/download/Lab08_TA_2008.ppt, Korea Advanced Institute of Science and Technology, 2008.
- [4] Network simulator (ns-2), internet technologies 60-375. <http://web2.uwindsor.ca/courses/cs/aggarwal/cs60375/NS/NS2.ppt>.
- [5] M. Welzl. The ns-2 network simulator. <http://www.welzl.at/research/tools/ns/welzl-ns-tutorial.ppt>, 2008.
- [6] Yi Pan. Introduction to opnet simulator. <http://netresearch.ics.uci.edu/ypan/>, 2008.
- [7] X. Chang. Network simulations with OPNET. *Proc. of the 1999 Winter Simulation Conference, P.A. Farrington, H.B. Nembhard, D.T. Sturrock, and G.W. Evans, eds.*, pages 307–314, 1999.

- [8] J. Prokkola. OPNET - network simulator. http://www.telecomlab.oulu.fi/kurssit/521365A.tietoliikennetekniikan_simuloinnit_ja_tyokalut/Opnet_esittely_07.pdf.
- [9] S. Keshav. Real 5.0 overview. Cornell University, Available as:<http://www.cs.cornell.edu/skeshav/real>.
- [10] Yi Pan. INSANE, *An Internet Simulated ATM Networking Environment*.
- [11] R. L. Bagrodia. Designing efficient simulations using maisie. *Proceedings of the 1991 Winter Simulation Conference, Phoenix, AZ, USA*, pages 243–247, December 1991.
- [12] I. Jawhar and J. Wu. Resource allocation in wireless networks using directional antennas. *The Fourth IEEE International Conference on Pervasive Computing and Communications (PerCom-06), Pisa, Italy*. Publisher IEEE Computer Society, pages 318–327, March 2006.
- [13] I. Jawhar and J. Wu. Race-free resource allocation for QoS support in wireless networks. *Ad Hoc and Sensor Wireless Networks: An International Journal*, 1(3):179–206, May 2005.
- [14] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad-hoc wireless networks. *Mobile Computing*, pages 153–181, 1996.
- [15] C. E. Perkins. *Ad Hoc Networking*. Addison-Wesley, Upper Saddle River, NJ, USA, 2001.
- [16] I. Jawhar and J. Wu. Quality of service routing in mobile ad hoc networks. *Resource Management in Wireless Networking*, M. Cardei, I. Cardei, and D. -Z. Du (eds.), Springer, *Network Theory and Applications*, 16:365–400, 2005.
- [17] I. Gerasimov and R. Simon. Performance analysis for ad hoc QoS routing protocols. *Mobility and Wireless Access Workshop, MobiWac 2002. International*, pages 87–94, 2002.
- [18] W.-H. Liao, Y.-C. Tseng, and K.-P. Shih. A TDMA-based bandwidth reservation protocol for QoS routing in a wireless mobile ad hoc network. *Communications, ICC 2002. IEEE International Conference on*, 5:3186–3190, 2002.
- [19] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, The MIT Press, Cambridge, Massachusetts, USA, 2001.
- [20] Frank M. Carrano. *Data Abstraction and Problem Solving with C++: Walls and Mirrors*. Addison Wesley, USA, 2004.
- [21] Larry Nyhoff. *ADTs, Data Structure, and Problem Solving with C++*. Prentice Hall, Upper Saddle River, NJ, USA, 2005.
- [22] Walter Savitch. *Problem Solving with C++, the Object of Programming*. Addison-Wesley, NY, USA, 2005.
- [23] A. Jardosh, E. M. Belding-Royer, K. C. Almeroth, and S. Suri. Towards realistic mobility models for mobile ad hoc networks. *In proceedings of ACM MobiCom*, pages 217–229, September 2003.