# Creating an Object-Oriented Wireless Network Simulator

Skyler Nesheim and Luong Hoang
Math and Computer Science Department
Drake University
Des Moines, Iowa
scn003@drake.edu or lh0025@drake.edu

## Abstract

Wireless routing protocols are currently among the most actively researched areas in computer science. There are hundreds of such protocols already established and even more under development. Wireless network simulators however have not received as much attention. Currently, there are only a handful of these simulators which are powerful tools for analyzing routing protocols. But, altering their code can be tremendously cumbersome due to the great level of detail they were developed. Some of these systems were designed using the object-oriented programming languages but failed to take into account the various strengths of using such languages, such as inheritance and polymorphism furthermore making them less extensible.

As we have researched wireless routing at Drake University, we have found the need to develop our own wireless simulator as a test environment for our protocols [4]. The goal of this simulator was to improve the ability to add new and sometimes very different wireless protocols to the system and remove the overhead and complexity in the layers of wireless communication. We felt to develop such protocols and test features of interest to us we should not be required to have an in-depth knowledge of the networking layers.

# *1 Introduction*

Development of our network simulator began in the summer of 2006 and has continued throughout this academic year. The initial phase of development began with extensive UML modeling and consideration of design patterns. We believed the use of UML modeling and design patterns would offer us several advantages; one, it would guarantee we considered our system carefully before we ever began programming, and two, it would create an extensible and reusable simulation system [1].

Although this was a small, college research project in which time, money, and programming experience were all constraints we still attempted to closely follow the system development life cycle. The system development life cycle should pursue this pattern; design, followed by implementation, and then testing, and finally, code maintenance. Note that this process is a cycle and that at any point the process may return back to a previous stage. Our system has undergone a series of design reviews, coding changes, and testing.

# *2 Network Creation - System Design Phase*

Before we ever wrote a line of Java code we attempted to carefully consider all possible aspects of our system. During this process we used the Unified Modeling Language (UML) to extensively draw out what our system might look like. We created class diagrams representing the whole system and outlined the member data and functions each specific class would have. When considering the system's design we asked ourselves questions like, "Which parts of the system will be changing" and "Which pieces of the system will be the hardest to maintain [1]?"

It became clear that the network representation including nodes, connections, and routing protocols would be one aspect of the system that would be constantly changing. Our system represents a network as a set of nodes joined via a series of connections, which can send messages to each other using routing protocols. Potentially, each different protocol would require its own node and connection type. The process for creating these objects might also change. For example, in some situations it might be appropriate to randomly generate the locations of nodes in the network, but in others it would be important to allow nodes to be placed in specific locations.

These realizations led to our decision to incorporate object-oriented design patterns into the design of the network simulator. Using design patterns would enhance the maintainability and extendibility of the simulator. Considering the pieces of the system which could be in constant flux, two patterns immediately jumped out as candidates for our system.

The Builder Pattern, which separates the building of a complex object from its representation, seemed to be a practical choice for creating networks. Using the Builder Pattern enabled us to outline a step by step process for creating wireless networks. This

outline was then used to create many different types of Network Builders. So when the simulator wants to create a network the appropriate builder is instantiated and the program asks the builder to build a network without considering or caring exactly how the builder is creating the network object.

One possible issue we uncovered with the builder pattern is that there is no specific logic in place for which kind of nodes and connections to use when creating a network [1]. Two builders might want to create the same type of nodes, but use them in different ways. A potential class overload could occur if the simulator had to have a class for each type of builder, node, and connection. Under these conditions the system would have to have both a random network builder for GPSR nodes and one-way connections and also a builder that builds networks from file with GPSR nodes and one-way connections. The number of classes multiplies very quickly! Ideally, this situation would be avoided by our design.

The Abstract Factory Pattern was a reliable solution for this problem. In general, Abstract Factories are in charge of creating specific instances of objects for other classes that don't want to bother with how to create these objects [1]. Our system design called for two different categories of Abstract Factories; one for connections and one for nodes. These two types of Abstract Factories became the solution to our problem with Network Builders. In our system, when a builder object is created it is given both a node and a connection factory to aid in the creation of these objects.

Using this design we successfully eliminated the possibility of class explosion in our system. Network creation classes in the simulator would be limited to a builder for each network building process, a node factory for each type of node, and a connection factory for each type of connection.

## *2.1 Network Creation – Implementation Phase*

After placing weeks of effort into designing our system we were ready to start coding. It seemed practical for us to break the implementation into pieces as we realized that if at any point we came across a problem we would have to re-visit the system design phase.

The first logical piece of the coding to tackle was the network creation component of the software. This involved creating the network object, the network builders, a graphical user interface for user input, and the network simulation controlling class. As stated earlier the network object is comprised of a list of nodes and connections between these nodes. Each node has its own instance of a routing protocol which it uses to communicate with other nodes.

During this stage of development we also created a Network File Descriptor and Network File Descriptor Reader class. These classes were assigned the responsibility of reading user input and using that to instantiate the appropriate network builder with the correct node and connection factories attached. The program is kicked off when the Network

Simulator Controller asks the user for the appropriate Network Descriptor information. Next, an instance of the builder is created and stored in the Network Simulator Controller. Then the Network Simulator Controller class asks the builder to create a network which is stored as an instance variable in the Network Simulator Controller.

To test our implementation of basic network creation we created a simple Java Swing class called Draw Graph. Draw Graph takes a network object and uses the Java Graphics to draw a representation of the network nodes and connections on the screen. After a network object is created we pass it to Draw Graph to verify the network appears as expected.

# *3 Routing Protocols – Planning and Implementation*

After we agreed upon the initial design of the simulator and implemented it, the basic pieces were in place for nodes to begin communicating. Until this point the nodes did not have any functions or activeness as in a real network. The next logical step in our project's implementation would be to add message routing.

Message routing, in short, refers to a way of selecting paths in a computer network along which data will be sent. There are many different routing protocols. Each of these offers its own strengths and weaknesses, but generally they have goals of minimizing messages, maximizing efficiency, avoiding too much traffic, and not causing lag or loss of data.

There are a few main categories into which each routing protocol may be classified:

1. Pro-active (Table-driven: maintaining a list of destinations and routing tables)
2. Reactive (On demand route request when there's a packet outstanding)
3. Hybrid (Pro-Active / Reactive)
4. Hierarchical
5. Geographical

One of our ultimate goals is to create a network simulator with many routing protocols giving first priority to the system's extendibility, flexibility, and ease of use. To accomplish this goal it is important that new protocols can easily be added to the system. In the future we would like to have a wide variety of protocols implemented in our system from all the routing categories. Accomplishing this task we demonstrate the creation of a powerful piece of software. Currently, we have two different routing algorithms implemented, one of which is classified under geographical, and the other one as reactive.

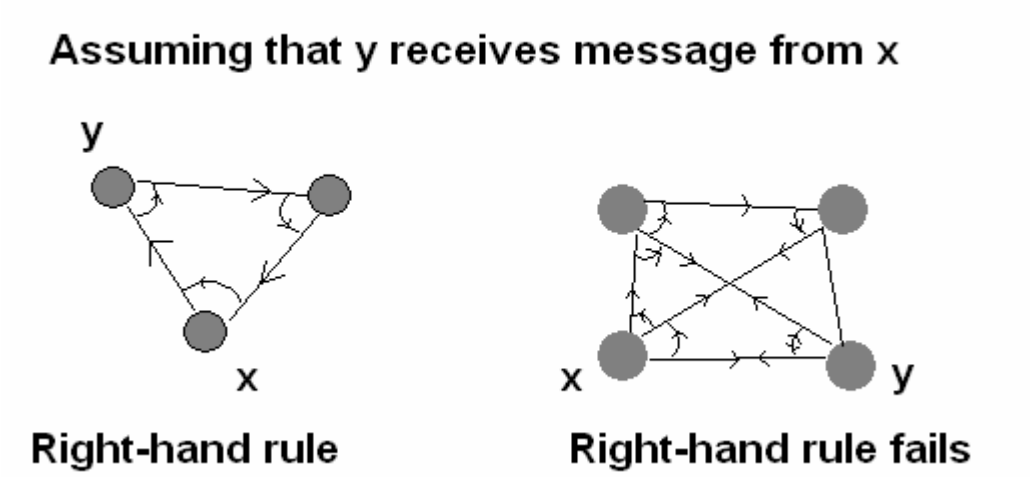
## *3.1 GPSR (Greedy Perimeter Stateless Routing) - A Brief Introduction*

The first protocol which we attempted to implement was GPSR, or Greedy Perimeter Stateless Routing. GPSR allows for the network to use the positions of the nodes to make

forwarding decisions. The routing protocol assumes that each node or computer has a GPS (Global Positioning Satellite) antenna which it can use to determine its location. Thus in GPSR, geographical information about the network is readily available. Another assumption is also made such that when a node wants to send a message to another node it will attach the location of that destination node to the message. Using the geographic information a node gathers about itself, its neighbors, and the destination, each nodes can make decisions on how to send messages across the network [3].

If one of the node's neighbors is closer to the destination than the current node, the message should be forwarded to that neighbor. If there are multiple neighbors that are closer the current node should choose the one that is closest to the destination as the next node to forward to. This state of forwarding is called greedy forwarding, where a node can "*greedily*" forward a packet to a closer neighbor in order to minimize time and cost, given that such a neighbor exists.

However, this neighbor will not always exist. When a node is itself geographically the closest node to destination, greedy forwarding is at a stalemate and therefore improved algorithms are needed. In such a case, a packet will be forwarded around the "*perimeter*" of the network, specifically according to the right-hand rule. Under this rule, when a packet encounters a dead-end, it will be forwarded to the node which is first in counter-clockwise order. The counter-clockwise order is determined based on the edge the packet came from or, in the case that this is the source node, the edge between the source and the destination. This method may result in a loop where the graph has crossing links, resulting in failure of sending the packets. In order to ensure the stability of the algorithm, several other solutions were suggested to remove nodes with crossing edges in the network.




The *Relative Neighborhood Graph (RNG)* and *Gabriel Graph (GG)* are two examples of such [3]. RNG and GG provide ways to remove crossing links in a graph and guarantee that the resulting network has no two overlapping connections. For the purpose of this paper, we will only discuss the method of RNG graph. To create a "planarized" graph, we proceed with the following rule: an edge exists between two nodes only if the distance between them is less than or equal to the maximal distance between each of the two and

every other node in the network. With the help of RNG graph, GPSR guarantees that if the graph is connected, i.e. there is a path from each node to every other node in the network, a message can be delivered.

### *3.1.1 GPSR - Design Phase*

When implementing routing protocols we again found it necessary to carefully consider the design of our software. We have put in efforts to understand GPSR's underlying algorithms, and to design all structures so as to guarantee the flexibility and extendibility of our system. Regular meetings were held where we discussed our own opinions and understanding of the algorithm, providing suggestions on different aspects of the project concerning the efficiency and accuracy of our program.

While studying GPSR we realized the need for a change to our system design. The original plan was to use routing protocols from the Network Simulator Controller level and only instantiate one protocol for the entire network. When we started actually writing protocols it became immediately evident that Routing Protocol should be implemented at the node level as a component of each node. This model more accurately depicts how things might work in a real network where nodes themselves are expected to make decisions regarding message forwarding.

Because of our thorough planning at the early stage of the code's development, implementing the first routing protocol did not require much work. In fact, we were pleased with the system's overall compatibility and flexibility.

### *3.1.2 GPSR - Implementation Phase*

After consulting different sources to ensure our understanding of GPSR, we began to implement it. The process of greedy forwarding is very straight forward. Each node maintains it's own x-y position on the graph. These nodes also contain a list of neighbors and their x-y positions. To determine if a message can be forwarded *"greedily"* a node must compute the distances between each of its neighbors and destination of the message.

Forwarding in non-greedy situations was more challenging. While implementing this process we encountered some difficulties with GPSR which were not related to how our system was designed, but rather to a technical aspect of the protocol. At the stage where greedy forwarding fails, in order for right-hand rule to work correctly, we needed to find out the first edge counter clockwise from the previous to forward the packet. After some brainstorming, we found good use of a built-in method atan() in the Java Math Library which computes the angle between two lines [2]. We also came across other problems such as how to forward the packet in its perimeter state, mostly because of our confusion to how the original algorithm actually worked.

### *3.1.3 GPSR - Testing Phase*

Testing became a very important step in the process of creating and debugging the GPSR routing protocol. Generally, our tests consisted of creating random networks and sending messages between nodes. However, if there was a network that had particularly interesting dimensions, we saved it and used it for later tests. There were errors, bugs emerging as we advanced to the testing phase. At times packets might be forwarded around an endless loop or stuck in certain locations. In order to help our testing, we printed out a file of the network information, connections and nodes' locations as well as the current locations of the packets. We also used the debugging utility in Eclipse to step through the code line by line as messages were sent. To date, it seems our system maintains a successful implementation of GPSR.

## *3.2 AODV (Ad-hoc On-demand Distance Vector) – A Brief Introduction*

AODV was the second routing protocol we implemented in our system. This protocol is one of the most popular wireless routing protocols and is in use on some network cards today. AODV is a reactive protocol meaning that routes to other nodes are determined only when needed [5]. Unlike in a GPSR network, AODV nodes do not know their geographic location. The only information an AODV node has about the network is a list of its one hop neighbors. In a true AODV system this list is preserved by periodically sending hello messages to every node in broadcast range.

AODV nodes also maintain a routing table which stores routes to nodes in the network. If an AODV node wishes to send a message to another node, but does not know a route to that node, a route request must be sent to find a route. This route request, or RREQ, is forwarded to all of the nodes neighbors, who check their neighbors for the destination, and forward the RREQ if they do not "know" the neighbor. Each intermediate node will store the route to the source node in its routing table in order to reply to the source node if and when the destination is found [5].

When a node receives a RREQ, it checks whether it knows a route to the destination, if it does then a route reply, or RREP, is generated and sent back to the source using the information stored in the routing table. When a node receives a RREP it stores the route to the destination in its routing table, and if that node is the source of the RREQ it can begin sending information to the destination.

Like a real wireless network, a computer's position may change and thus a link between two computers may be broken. Because AODV is a reactive protocol and paths are maintained it is important to account for this situation. If a node attempts to send a packet to another node which is in its routing table, but finds that the link is broken a RERR is sent along a path back to the source of the packet. Along this path, each intermediate node erases the path to the unreachable node from its routing table [5].

### *3.2.1 AODV - Design Phase*

AODV is a much different protocol compared to GPSR, and in a sense much more complicated. GPSR is very iterative in nature and also has the advantage of knowing geographical information. The difficulty in implementing AODV lies in correctly and effectively maintaining a node's routing table. Sending packets in an AODV network is not nearly as iterative as sending packets in a GPSR network. This realization led us to rethink our system's design. It seemed that upgrading our application to use Java Threads would appropriately solve the problem. Adding multi-threading capabilities to the simulator would allow nodes to send several messages simultaneously.

Multi-threading also made our system more complex so we had to carefully design our structures and algorithms for the AODV protocol in such a way that when confronted with a large sample network, the efficiency and accuracy was consistent. At this point the decision was made to create new threads at the message level. Thus, when writing a new protocol for the system, programmers could ignore dealing with threads and focus on simulating how the protocol actually worked.

Our careful system planning at the early stage of the code's development again became a great ally to us. We were able to quickly add all the classes and setup the AODV protocol. Here, as with GPSR it became the semantics of the network simulation which held up our development of the simulator.

### *3.2.2 AODV - Implementation Phase*

After we have decided on our design and ideas, we began coding AODV. The first step was to implement the multi-threading capabilities. We created a new class called Message Scheduler which nodes could use to send messages to other nodes. The Message Scheduler also used another class called Message Sender which would initialize and thread for each message sent. After a message was effectively dead the thread would also complete so as not to tax the Java Virtual machine.

Maintaining each node's routing table was also a bit tricky. For AODV to work correctly we needed to guarantee that each node's routing table was being correctly updated. Each time a node receives a RREQ, RREP or RERR; it should update its routing table performing any adding or removing as necessary. There may be several different routes between a source and a destination. It is important that AODV nodes maintain the freshest and shortest routes in their routing tables. The creators of AODV provided a mechanism for keeping track of freshest routes called the sequence number, which we took full advantage of in our implementation [5].

In addition to the necessary functions of AODV, we also added mode to our network simulator which allowed us to visualize simulations of messages being sent. This functionality made it easier for us to track system bugs and test. When a message is being sent from one node to another, the nodes and the link between them is highlighted in

different colors. By implementing this functionality we effectively added a visualize mode to GPSR and any other protocol we should wish to implement.

#### *3.2.3 AODV - Testing Phase*

As mentioned above, the testing phase was supported by the visualize mode added to the network simulator. We also printed messages to standard output regarding the AODV message sending process. Using these two tools we were able to ensure that the AODV protocol we implemented was working correctly and not flooding the network with un-needed messages or destroying any node's routing table.

## *4.0 Current & Future Work*

We are currently researching the AODV/k-SPR routing protocol which is of particular interest to us at Drake University as one of our professors, Dr. Rieck was heavily involved in creating the concept of this protocol [6]. k-SPR itself is not an actual routing protocol but rather a k-dominating set of nodes which allow for more efficient message transferring by giving some nodes advanced responsibility as network routers. We aim to integrate our AODV system with the k-SPR set, thus creating the AODV/k-SPR protocol.

A k-SPR set must be k-hop connected meaning that all of its elements are within (k+1) hops of each other. It must also satisfy the property that given any two nodes **x** and **y** in the graph, there exists a shortest path connecting them along which "routers" are encountered [6]. Each router in an AODV/k-SPR network acts as an intermediary for sending packets from one node to another, improving speed and efficiency of messages forwarding. Therefore a smaller sized k-SPR set is more desirable. It is proven that finding such a set with minimal size is an NP-complete problem; hence there are many algorithms which attempt to reach the closest to optimal size set [6]. We call these algorithms router selection algorithms and these will need to be designed into our system in order to leverage the development of AODV/k-SPR. To date Drake students have implemented three such routing protocols: k-SPR-I, k-SPR-C, and k-SPR-G [4].

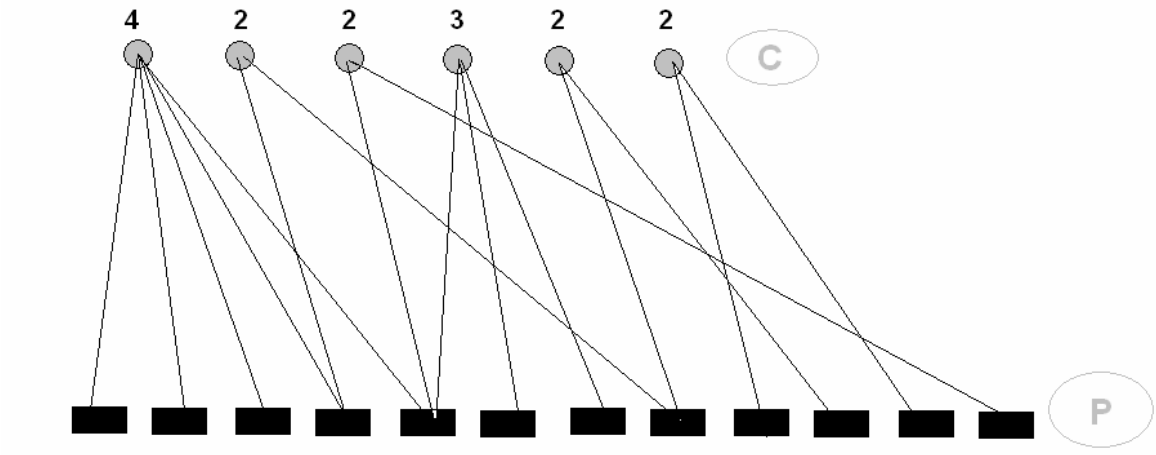
<u>k-SPR-I:</u>

In this algorithm routers are chosen based on the ID numbers of the nodes. Specifically, each node **x** examines its k-hop local network and identifies all pairs **y** and **z** that are exactly k+1 hops apart for which **x** lies along the shortest path connecting them. **X** also identifies all the nodes that lie strictly between **y** and **z** along some shortest path. Among all of these nodes, we choose the one with the highest ID number and say that it "*covers*" that pair. Thus if a node covers some pair, then it becomes a router or an element of the k-SPR set.

k-SPR-C:

Another algorithm is to make decisions based on the covering numbers of each node. The covering number is computed by how many pairs of k+1 hop distance that a node **x** lies on the shortest path between. Each node broadcasts to its neighbors its covering number and based on the final information, the one with the highest covering number is chosen.

k-SPR-G:
k-SPR-G is a greedy algorithm to choose routers in a set. Note that "*greedy*" here has different meaning than that of the GPSR protocol.




In greedy router selection, we use the bipartite graph to select routers. This graph contains two sets C and P. P is the set of all ordered pairs (**x**, **y**) that a node **u** covers, and C is the set of all nodes that cover some pairs in P [6]. If a node covers a pair, we represent this by an edge between. So to select a router, we choose the one with most edges in the current graph. After it is selected, we remove it and all pairs that it covers to end up with a new bipartite graph and repeat the process.

## *5.0 Future System Design and Development Planning*

In the process of designing the AODV/k-SPR protocol, we have come across an issue that we are currently trying to resolve. It seems that we have cheated while implementing our routing protocols. We have provided a class called Network Helper which allows nodes to get a list of all their one-hop neighbors. In a real wireless network this process requires each node to send out a multicast message to all nodes in transmission range called a hello message. So cheating in this scenario isn't a huge blunder. But, k-SPR would expect that nodes can discover all of their k-hop neighbors, and this process requires many more messages thus affecting the efficiency of the algorithm. Essentially this sort of cheating would be OK, if we did not want to collect any data about protocols, but data collection about protocols is still a major goal of our project.

We have designed a solution to this issue under which every node has a process which it continually follows; send hello messages, discover neighbors, send any messages waiting to be sent, receive and handle any messages sent to you, and loop back to the beginning. This process more accurately follows the process of a real network but severely convolutes the design of our system. At this point we are still deciding whether we will use this solution, revert back to the old system, or design a new solution to the problem. One other possible solution would be to account for this hello messaging.

As with any other change we have made to the project we will continue designing and brainstorming our ideas so as to ensure the coherence of our system as well as its flexibility with any protocol which might be added.

## *6.0 Conclusion*

In this paper we have presented our development and design of a new wireless network simulator. Over the course of this ongoing project we have attempted to develop a simulator that has high adaptability and friendly user-interface. We have put in efforts to carefully design and implement our system. In doing this, we've gained not only the valuable knowledge of system design but also, to a certain extent, a deep understanding of Java applications. We aim to continue to strengthen our understanding of the various issues in our system and do our best to examine all possible mends.

In working with network routing protocols, we were exposed to one real life application of many emerging in our technology society. Network routing difficulties such as cost and efficiency have become backbones in our computing knowledge. Getting an insight into what current scientists are working on offers us a new experience and a worthy step into the professional world. Let alone the application, what we have learned is much richer than what we have programmed.

## References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, pages 87-106, 127-135, 1995 Addison-Wesley.

[2] Brad Karp. Geographic Routing for Wireless Networks, Ph.D Dissertation, Harvard University, Cambridge, MA, October, 2000.

[3] Brad Karp, K.T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. MobiCom 2000.

[4] Skyler Nesheim, Matt Smith. Hierarchical Routing using k-SPR. 2006 MICS Conference. April, 2006.

[5] Charles Perkins, Elizabeth Royer. Ad-hoc On-Demand Distance Vector Routing.

[6] Michael Rieck, Sukesh Pai, Subhankar Dhar. Distributed routing algorithms for multi-hop ad hoc networks using d-hop connected d-dominating sets, Computer Networks Journal, v. 47, no. 6, 785-799, Elsevier, April 2005.