AD-HOC NETWORK ROUTING BASED ON AODV AND K-SPR SETS

Tiffany Meredith, Jennifer Ehrlich and Michael Rieck
Department of Mathematics and Computer Sciences
Drake University
2507 University Avenue
Des Moines, Iowa 50311
michael.rieck@drake.edu

Abstract

This research explores AODV / *k*-SPR routing strategies by implementing these in Rice University's Monarch Project (wireless network) portion of U. C. Berkeley's NS-2 network simulator. Supported by a CREU grant, the authors of the paper being submitted have made a considerable effort at learning to alter the C++ and Tcl code for NS-2, and have completed the implementation for their basic routing strategy. The goal now is to collect and analyze data concerning this new strategy. The new algorithm will be contrasted with ordinary AODV in terms of issues such as throughput and robustness.

1 Introduction

Supported by a CREU grant from the Computer Research Association, we have spent the academic year of 2004-2005 exploring ad hoc wireless networks. In particular, we have installed and learned to use the NS-2 network simulation software, primarily developed at the University of California at Berkeley. In addition, we have learned about a variety of routing protocols for ad hoc wireless networks. We have particularly focused on Ad hoc On-demand Distance Vector routing (AODV) as described in [15] and [16], and on the notion of a *k*-SPR set as defined in [10] and [14], and the usage of these as routers in ad hoc networks. Ultimately our goal was to adapt AODV so as to incorporate features of *k*-SPR-set routing. This we have achieved.

AODV over the past couple years has become a popular way of sending messages from one location to another due in large part to its scalability. Our goal was the analyze AODV and also *k*-SPR sets and come up with a better version of sending a message from point A to point B in a wireless system, one that incorporates desirable features of both methods. We were able to expand our knowledge of networks, NS-2 and routing protocols, and did research into ideas for finding better ways to send messages in a wireless network. We studied the source code for NS-2 and altered it so that we could simulate sending messages around an ad hoc wireless network in a manner that we could control. We were able to allow Tcl scripts and the core NS-2 C++ code to communicate with one another, primarily through the "god" object. Substantial changes were made to the god.cc file to add features in support of our goals. Also, we made a copy of the aodv.cc file which we called kspr.cc. We made a great deal of alterations to this file in order to implement a version of AODV that incorporates *k*-SPR sets. We call this new protocol AODV / *k*-SPR.

We have also implemented a Java program that exhibits visually the behavior of our altered NS-2 simulator. It displays a random ad hoc network, highlighting the routers in the network and the propagation of route request messages based on AODV / *k*-SPR. This has helped us to check the correctness of our implementation. We plan to begin collecting and analyzing data, comparing our protocol with ordinary AODV.

2 Installing NS-2

Throughout the project, Ehrlich and Meredith were able to work successfully with the portions of NS-2 program (the "Monarch Project" from Rice University) that is specific to wireless simulation, although this generally demanded a considerable effort to learn new material at each stage. The wireless portion was the only part of the NS-2 system that required significant adaptation in order to suit our needs.

After finding the NS-2 program on the Internet at http://nile.wpi.edu/NS/, Ehrlich and Rieck began to install the system. During the first download, it seemed as if nothing had gone wrong and we were ready to start working on the project. However, a few days into the understanding and changing the NS-2 code, it was discovered that not much information existed on the ns.2.27 code. This presented a problem because no one involved in our project knew all of the answers to the questions that were being asked. Even consulting the Internet was not initially helpful. Therefore ns.2.27 was removed and ns.2.26, a slightly older version with fewer bugs and more Internet information on it, was installed. Thus began our project. Ehrlich and Meredith were immediately able to start investigating the code and understand it in substantial detail.

3 Initial Simulations & Learning to Use NS-2

The first experiments were done with very little knowledge of what would happen and how things would work. However, it was by this method that a great deal of information was used and learned that ultimately helped Ehrlich and Meredith understand the code. By using the “nam” animation system, and looking at actual simulation runs that were presented, we discovered that one can right click and set a flag to follow a packet path through the simulation. Each flag setting was able to give additional information on the nodes and this is when we began to understand the network-related acronyms used. Each packet showed information about itself that indicated if it was an "ack" (acknowledgment packet to a previous packet that was sent or if it was a "CBR" (constant bit rate) or "TCP" packet, which are data messages that are being sent to another node. These flags and the animator in general helped in our understanding of how the simulation worked.

Also, in watching the simulation, it was seen that some packets would fall off or be dropped off a queue that was created after it passed through one node on its way to its final destination. This is simply because some queues that are called Drop Tail Queues would allow for only a certain amount of information to be held in the queue. If there were too much information to queue, the Drop Tail Queue would begin to create a queue after the node that is simply passed through. If too many packets began to be queued at this point, then some would begin to fall off. However, if the packet remains on the queue during the time it will eventually reach what is called a "TCP sink." This sink, as it is named catches all of the TCP packets that are sent to. Upon discovering that the sink existed, it was also figured out that a node that has a sink is the only nodes that have a capability to send acknowledgment packets to the sending node. This means that when a packet from a node is sent, it will either be stored into the receiving nodes memory tables, or it will be dropped into the so called sink. If it is dropped in the sink, the information will be stored in the memory tables, but also an acknowledgment packet is sent back to where the original came from to inform the other node that the packet had been received. This acknowledgment packet is used to ensure that data was received by a node. When a packet is dropped, there will be no acknowledgment packet sent back saying that it was lost, so the acknowledgment is always a good thing to have.

There are also other nodes that can be set up in general to merely receive the information and keep it there. This also does not allow for a return packet, but the exact reasons why are still unknown, but this shows us that not all nodes have the ability to send acknowledgement packets. This is why in the routing protocols, the RREQs become increasingly important. It seems it would be because the packets are received and immediately stored into a node's data. Instead, with the sink, it seems that the node will look at each packet and send back an acknowledge message to the original node that sent the information to tell it the packet was received. The sink does not merely just add the information to the receiving node, but will instead responds to the information that was just sent.

Ehrlich and Meredith were able to use other Internet sites to help explain what was happening, particularly Power Point slide and online tutorials. Finally, before beginning to break the NS-2 code down and study its individual components, Meredith and Ehrlich were able to add nodes to the simulation that was already created. This was at first mainly done through copy, pasting and altering the Tcl code, but slowly grew into the actual writing of the code. This ended the first stage of the project in which Ehrlich and Meredith attempted to look at the code in its entirety and to focus on portions of special importance to our project.

4 Initial Modifications

Once the basic understanding of NS-2 was reached, the project was broken down into tasks related to the separate languages of C++ and Tcl. Ehrlich began to work on the C++ portions attempting to recreate the methods that are performed by the god files. God (general operations directive) was a class which enabled us to have the Tcl code and the C++ code interact with one another. It also knew about everything that was going on in the program and where all of the nodes were located at any given moment. The first main area that was tackled was adding a number of methods to enable our Tcl scripts to talk with god.cc. We were able to use DOxygen to be able to navigate through the code extremely easily. DOxygen allows the user to click on different functions and data types and then bring them to where they are defined. We discovered that by using the god::command function that we were able to have Tcl call different functions in the god.cc code. Ehrlich and Rieck started to try to decipher the Tcl code that communicated with the god.cc code. We decided to make god.cc more powerful than the previous developers. We added the functions:

- *who-neighbors:* that enabled one to send in a node and determine all of the node's neighbors (the nodes to which it was connected), returning this list of the neighbors as a string
- *get-dist:* that allow a node to discover the distance between the two nodes that you passed into it. It returns the hop count between the two nodes.
- *all-min-hops:* returns the entire min_hops array as a string.
- *all-routers:* returns a list of the routers that are on the specific graph.
- *k-SPR-I:* invokes k_SPR-I algorithm to select all the routers nodes.
- *get_next_hop_list:* allows the Tcl code to get an array with the list of the next hops that are one step closer to a destination nodes.
- *j_hop_neighbors:* returns an array that describes all the nodes that are

within j hops of the node that is specified.
- *is-within-j-hops:* calls isWithinJHops(int x, int y, int j) to see that if two nodes are within j hops of one another.
- *j_hop_routers:* finds all of the routers within k hops of the node that you are currently at. It calls jHopRters(int x, int j) to actually run the function.
- *j_hop_special_routers:* finds all of the special routers. A special router is a router that is within j hops, and is such that moving towards it using any shortest path, you do not go through any other routers. It does this by calling jHopsSpecRters (int x, int j) or jHopsSpecRters (int x, int y, int j) depending on how many parameters are used. The difference between the two calls is that when you call it with only two parameters then it finds all of the special routers whereas with three parameters it only returns special routers that are further away from y than is x.
- *buildNextHopList:* enables the program to return a string that contains all of the places that the message can be sent to from its current node via k_SPR_I(int k).

Many of the functions called using the Tcl code call other functions in god.cc.

5 Understanding the Tcl code

Meredith began to work independently learning as much as possible about Tcl. Due to the lack of resources at Drake University for this language, the Internet once again began a great resource for the development of understanding the language. There are multiple websites that allowed for the understanding and even some had tutorials to help diagram how the language worked. After a few days of working on understanding the code, Meredith and Rieck were able to begin inserting new methods in the Tcl code that would correspond with the C++ code. The main idea was that the Tcl needs to communicate to the C++ through an object called the "my agent." This concept allows the two languages to be able to communicate with one another. Most of the coding was done in the C++, and Tcl portions only needed to control the C++ methods.

Rieck began to work on extracting the data that was sent from the C++ to the Tcl code in order to understand the information. All information came back from C++ in the form of a string that had spaces in between the different pieces of information. Each string then needed to be separated so that it could be analyzed as to which node it belonged to. This was done using the method of looking at all information that comes before the first white space in the string. Once a white space was found, the information preceding it was removed from that string and printed. This would continue on until all of the information was parsed and displayed in a meaningful manner. In order to put the results in a matrix form, the code will force a new line after a specific number of characters have been parsed from the original string and printed. This code was done on the information about the 1-hop and 3-hop neighbors and the router list. Meredith then began adding code that would explicitly state what was being dumped on to the screen after each time the simulation was ran in order to analyze what was happening. This diagnostic testing helped to pinpoint where there were problems within the Tcl and C++ code. However, upon testing this idea, it took a great deal of time to have the information print out on the screen using Tcl, especially for graphs large enough (several dozen nodes) to be practical use in our experiment.

In order to render a suitable visual presentation, a graphical program (written in Java) was implemented. This program reads all of the data that a C++ command created during the simulation, and prints the information in an orderly manner to allow for the user to understand the details of the simulation.

6 Using a Java Program

The aforementioned Java program is a simple program that is designed to read from several data files, and then use a string tokenizer and Java graphics to read the information that is produced by the Tcl code and create a graph to represent the network and its behavior. The combination of the C++ and Tcl code working together creates information about each individual node that states where they are placed, their hops away from other nodes (using the Floyd-Warhsall algorithm), and the closest routers to them. First the Tcl code will create a file that is called “coordinates.dat” that will be used to explicitly state where each node is placed. Within our Tcl script “kspr.tcl”, the code is designed to generate the coordinates within a loop, and the only time that the loop can finish is once all nodes have coordinates and a connected graph is created. If the graph is not connected, the Tcl script discovers this fact and terminates, since we wanted to avoid isolated portions of the graph. Then, the C++ code creates a file that is called “adjacency.dat” and gives a listing of all nodes that are within one hop of each other. Using this list, edges are created between nodes, and a visual graph has been formed. After completing this and ensuring that it ran, two more files were created to enhance the information that is seen on the graph. The first is the file “routers.dat.” This file is created inside of the C++ code as well, and is a listing of all nodes that were chosen by one of the *k*-SPR algorithms to be routers. The ordinary nodes are then painted one color on the graph, where the routers are painted in red to make them stand out. The second file is called “route_requests.dat” and is being used in a Java Applet to see how the requests are sent between the nodes based on the information in the file. Using this we are able to observe the propagation of route request messages by highlighting the nodes involved. This is also a file that is found in the C++ code. In the end, the graph is a very useful piece of information. It is mainly being used as a diagnostics testing tool, but it is very helpful in explaining the procedures of the Tcl and C++ code. The graph seems to almost prove that the code is properly working in the way that we want it to.

7 Routing Protocols

7.1 DSDV Routing

Originally, we considered working with the routing protocol DSR (Dynamic

Source Routing), but this was not considered relevant to the project. The first routing protocol that was considered was DSDV (Dynamically Sequenced Distance Vectors). DSDV is a routing protocol that uses sequence numbers to identify packets that allow for the sending, receiving and storing of messages. When a network is initially created, the network nodes do not know about any neighboring nodes or any routes available to get to other nodes. In order to discover routes, a "hello" message is broadcast with a sequence number of one. This sequence number will inform all other nodes the age of the message being sent. Because the "hello" message has a sequence number of 1, it is the newest. All hello messages inform the sender of the hello about all other nodes that are directly next to it that can receive the hello. The other widely seen messages that are used are called RREQs (route requests) and will have a systematically chosen sequence number by a counter that is inherent in the routing. The RREQs are messages that are sent to request information about how to get to a node that is not adjacent to the sending node. Each node that receives a RREQ will store the information in the RREQ message in a routing table.

The information will consist of the sequence number, the node that sent the message, and the number of hops that it took the message to get to that node. The node will then look at the lifetime (maximum hop count) that is stored within the RREQ. If the node that has just received the message is not the one being requested, it will forward the RREQ long at there is maximum hop count in the life time has been exceeded. If the lifetime expires and therefore the message gets dropped or if a RREP (route reply) is not received within a certain amount of time at the original node making the request, then another RREQ is sent out. The RREPs in this protocol will act very similar the RREQs, but instead will have information about the sequence number, hop count, and path to the requested node within it, and propagate back along the route that the RREQ original came from. It is a direct reply back to the original node that the RREQ was sent from saying that the request for information has been received. The new RREQ will have a longer life timer that will allow it to reach more nodes. Usually the lifetime is incremented by one. This routing algorithm will allow nodes to learn about the quickest and most efficient paths to get from a node to another node. In DSDV, the RREQs are usually sent out every few seconds within the simulation, in order to discover if there have been new nodes added. Each time a RREQ is transmitted, the sequence number will be incremented by one. The sequence number will allow all receiving nodes to store the information with either the higher sequence number or if the sequence number is the same, then with the smallest hope count. In doing this, the information is able to stay current and all routes are the most efficient.

7.2 AODV Routing

Because DSDV will request information about the nodes with RREQs at set intervals, there can be a great deal of network overhead which can make the protocol run much slower and inefficiently. So the upgraded version of DSDV was investigated, AODV (Ad-hoc On Demand Distance Vector). This routing protocol is slightly more advanced in that the RREQs are not sent out unless the user within the simulation asks for them to be broadcast. Also, in AODV, the sequence numbers are much easier to calculate. In DSDV they were dynamically allocated, but in AODV they are continuously incremented each time a new RREQ needs to be sent from a node. This helps because it will eliminate the possibility that a routing loop could occur. In AODV there are also RERRs (router errors) that can occur when a RREQ is not properly matching or finding what it needs. RERRs are usually seen when an existing route has broken its link and the node that once was able to reach it can no longer do so. Upon this happening, the route's hop count will be stored as "infinity" until such time as the node can once again be reached. However, this only happens in wireless networks because the nodes can move in and out of range of the other nodes. This can create confusion, but as long as the RREQs and RREPs are properly used, everything will be systematically stored and made much more useful. The format of a RREQs message is as follows:

```
     0                   1                   2
3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |     Type      |J|R|G|        Reserved         |  Hop Count|
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                         Broadcast ID                          |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                    Destination IP Address                     |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                  Destination Sequence Number                  |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                       Source IP Address                       |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                    Source Sequence Number                     |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1 (Perkins, Royer, Das 24 November 2000)

AODV sends a RREQ out every time that it wants to find from point A to B. Once it finds B or a node that knows a route to B, that node will then send a message back to with the path. Each time this information moves back from one node to another it adds its node ID onto the routing table so that when A receives the information, it will know that exact path that was taken. This will help when wanting to reach that node again and not wanting to search through all other nodes for the fastest or best way there. Inside the nodes are two counters that AODV uses. The first is the sequence number that is used when a link is created. It will help remember how fresh the information is and when new information is received, if the new information is better than the old. This is only done when we are sending out new information to discover. The second counter would then be the ID_broadcast and this is updated for every new RREQ that is sent out.

The RREPs will keep the same ID_broadcast since that is how each node knows what is responding too. The ID_broadcast is once section of the RREP. It is the second part (the first part is the IP address). It is counter that will be incremented so that we know how far apart the node are. AODV is a great way of setting up a network because within each node there is a routing table that will store information about its adjacent nodes. However, in the wireless arena, there is the ability for the node to time out that will wipe away information after awhile. This helps to clear all information that is old and to ensure that new information is stored within the node. When a node becomes unreachable an infinity emblem is placed in the routing table, showing that there is no path to the nodes any more.

The AODV code will also use hello messages and send them to nodes one hop away, in order to learn their directly adjacent neighbors. Each time a hello message is received then, the node that had just received the message will attach its IP address before sending the hello message on. This will help to determine who was in the path as the hello messages were sent out. The major advantage to this process is that each node will learn about its k+1 hop local neighborhood.

7.3 *k*-SPR Routing

Once the decision was made to use the AODV routing algorithm, we began to plan a sensible way of incorporating *k*-SPR sets into this algorithm. In AODV, all of their nodes are treated equally. However in our approach, certain nodes will be singled out to serve as routers. These nodes form a "*k*-SPR set". In previous investigations into *k*-SPR sets, it was understood that router nodes actively exchange global information with one another in order to facilitate the global routing of arbitrary nodes. However, this is very expensive. In our adaptation of AODV, the routers and other nodes will only learn about routes in a reactive manner. The routers will however be given a special status in connection with route discovery. Our decision about how to integrate AODV with *k*-SPR sets will be explained in the next section. In this section, we will simply give a sense of the underlying notion of a *k*-SPR set, as well as basic algorithms for computing such a set of router nodes, as described in [10] and [14].

In the first place, a *k*-SPR set is a *k*-dominating set of nodes. This means that each node in the network must be within k hops of some router node (i.e. some node in the *k*-SPR set). A *k*-SPR set must also be *k*-hop connected, and actually to have the following desirable property: Given any two nodes x and y, there is some shortest path connecting x and y such that router nodes are regularly encountered, in fact a router node will occur within k hops of any point along the path. A number of algorithms have previously been developed for obtaining reasonably small *k*-SPR sets. Naturally, obtaining a *k*-SPR set whose size is very close to the smallest possible size is prohibitively expensive in terms of overhead communication between nodes. Yet the reasonable algorithms introduced in [10] and [14] can lead to acceptable set sizes.

The first of these algorithms is known as *k*-SPR-I. Each node *x* examines its *k*-hop local view and identifies pairs of nodes, *y* and *z*, that are exactly *k*+1 hops apart, and for which *x* lies along a shortest path connecting them. It also identifies all of the nodes that lie strictly between y and z, along some shortest path. Among all of these, it chooses the one with the highest ID and "elects" this node to "cover" the pair *y* and *z*. If a node observes that it has elected itself to cover some pair, then that node becomes a router, i.e. a member of the chosen *k*-SPR set. The "I" in *k*-SPR-I indicates that the decision as to which node to elect to cover *y* and *z* is simply based on the ID of the node.

A slightly more involved algorithm is an alteration of *k*-SPR-I that uses the nodes' "covering numbers". The covering number of a node *x* is defined to be the number of pairs of nodes, *y* and *z*, that are *k*+1 hops apart and for which *x* lies strictly between *y* and *z* along a shortest path. Each node is capable of computing its own covering number based on its local view. Once this is done, each node broadcasts this number to each of its *k*-hop neighbors. Only after this is accomplished, does the election process take place. In addition, the elections are now based on covering numbers rather than node IDs. Experiments have shown that this algorithm, known as *k*-SPR-C, results in a substantially small *k*-SPR set than does the *k*-SPR-I algorithm.

A third algorithm called *k*-SPR-G generally results in a much smaller *k*-SPR set than the sets produced by *k*-SPR-I and *k*-SPR-C. Naturally this comes at the expense of additional computations. The *k*-SPR-G is based on a "greedy algorithm" and a detailed description of it is beyond the scope of this article. See [10] for details.

7.4 AODV / *k*-SPR

As already mentioned, both AODV and *k*-SPR are concerned with finding shortest possible paths between nodes that need to communicate with each other. We wanted to design something new that was a little like AODV and a little like *k-SPR* , and which leveraged the best features of these methods. This was the thinking that led to the design of the AODV / *k*-SPR protocol. Both AODV and KSPR are based on shortest path routing and hope to control the amount of RREQs by using routers to control the propagation of RREQs. In our code we used special packets called RRREQs, but in reality these are just RREQs, handled a little differently as far as the way in which they are forwarded. In particular, our RRREQs are unicast from node to node, in contrast with AODV's protocol of broadcasting RREQs. In addition, we use routers from the *k*-SPR set to help control the propagation of the RRREQs.

Following the *k*-SPR approach, each node x sends out an "extended hello" (EHELLO) message to its neighbors. Each neighbor y of x receiving this message will attach its ID to the header, and then propagate the message out for an additional k hops. In this way, nodes within k+1 hops of x will learn about both x and y and the link between them. In this way, each node is ultimately able to construct a data structure to represent its "(*k*+1)-hop local view". Each node can then use and apply one of the three *k*-SPR algorithms to this data structure to decide whether or not it should become a router. If so, then it announces this by means of another message that propagates to a distance k. Thus each node learns all the routers within k hops of itself. This aspect of AODV / *k*-SPR is identical with the *k*-SPR algorithms.

RRREQ messages are restricted to propagate no further than *k* hops which is sufficient to send them from a router to another router, using local routing based on the local views. When a router receives an RRREQ, it checks to see if it has a suitable route in its routing table. If so, then it initiates a reply, identical to the AODV reply message RREP.

However, if a route to a destination is not discovered in the table, then the router looks for other routers to forward the RRREQ to as described below.

Suppose router *r* wants to find a shortest path from a node *x* to a node *y*, one that passes through *r*. Also suppose x is within k hops of r and hence is "seen" by r in its local view. Suppose too that r does not posses routing information to y. This situation could happen because r initiates a search, or because r receives an RRREQ from an ordinary node (the source) that it cannot immediately reply to, or similarly because r receives an RRREQ from some other router on behalf of the source. r then looks at its local view and identifies other routers s with these two properties: (1) $d(r,s) <= k$ and (2) $d(x,r) + d(r,s) = d(x,s)$. For each such router s, r sends a new (unicast) RRREQ message to s asking it where y is. When s receives this RRREQ, if it also does not know where x is, then it identifies router nodes to forward the request to and so forth. Naturally, when an RRREQ request can be handled, and so no further RRREQ messages need to be sent out, then corresponding reply messages are returned along the path back to the first source of the request.

If you are within *k*-hops from your destination node, then use the local routing information. If you are not within *k*-hops, the following will be used. When a message is sent from one node to another it goes through a series of processes. Imagine a string of nodes, A, B, C, D, all four of these nodes are connected in order and all within one hop of the next.

A---B---C---D

When A decides that it wants to send a message to D it first looks up D in its router table. The router table describes every node and what paths are available from one node to another. There is also a timer on each of the slots in the table. After an entry has been in the table for a certain amount of time it is erased because different nodes could have moved or they could have left the network. The lookup determines if there is a route that it already knows about to get to D. If there is nothing is this slot in the router table then it send out an RRREQ. It does this by calling NS-2's sendRequest function. By calling this function it sends an RRREQ from A to B. If B does not know where the final destination node is, then it alters the packet header and uses NS-2's "forward" function to send the message to all of its neighbors and also decrements the TTL, time to live, counter. If C knows where D is located then it does not have to forward the request, and instead uses sendReply to send a reply message back saying that he knows where D is and the path that should be used. C sends this reply back to B. B then updates its router table with the path to D and sends the reply back to A, who does the same thing then sends the actual message that wanted to be sent to D.

8 Message Format

Every message that is sent has a header component. When a message is sent locally via Ethernet and there is no connection to the Internet then each computer has a specific CMN number to use. When computers are connected to the Internet then they have 3 parts to their header, the CMN, the IP, and the KSPR, which only exists for the special routing packets means when you are sending something using the recvRRRequest, recvRRReply. Each part of the header, CMN, IP, KSPR, has a source and a destination. When you are sending a message then the KSPR source and destination are the original source and destination, in the example above the A would be the source and the D would be the destination. For the IP header portion when you are sending a regular message the source and destination are the final source, D, and the beginning destination, and the CMN source is the local source and local destination. This means that in the example above if you were sending a message from A to D and going though C it would have a local source of B, the node that it just sent it from, and the destination of C, the source that it is going to. If you are sending a special message, using the recvRRRequest, recvRRReply, then the KSPR_packet source and destination are still the same but the IP address, source is the past router and the destination is the next router while the CMN source is the previous hop and the destination is the next hop.

9 Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm is implemented in the original god.cc file in the NS-2 simulator and is used to find the "hop distance" between each pair of nodes in the network. The hop distance between two nodes is simply the minimum number of links required to reach one node from the other node. Beginning with an adjacency matrix that shows the connections between the nodes, the algorithm produces another matrix that contains the hop distances. It does this because it is an exhaustive search algorithm that is designed to check the path from the first node (denoted i) to the second node (denoted j). A three-way nested loop is used for this purpose. The outer-most loop is over the index i and the inner-most loop is over the index j. In between, there is a loop over an index k that also ranges over all node IDs.

Inside these loops, an integer d_{ij}^{k} is computed as follows. As initial information, d_{ij}^{0} is set equal to one if nodes i and j are adjacent, but zero otherwise. Inside the loops, d_{ij}^{k} is set using the formula

$$d_{ij}^{k+1} = \min \{ d_{ij}^{k} , d_{ik}^{k} + d_{kj}^{k} \}$$

As a result, d_{ij}^{k} measures the hop distance from node i to node j restricted to paths that only use internal nodes whose ID does not exceed k. Once k has reached the largest node ID, the hop distance (without restriction) between nodes will have been obtained.

The NS-2 simulator uses this algorithm to obtain the hop distances for the network as a whole. However, in connection with routing via *k*-SPR sets, each node must begin with a more restricted graph, namely its "*k*-hop local view" which is a subgraph of the whole network graph. Once a node has determined its *k*-hop local view, it can apply the Floyd-Warshall algorithm to this in order to obtain the hop distances as perceived in this local view. As indicated in [14] some important hop distances associated with the *k*-SPR set election process as perceived in the local view agree with the corresponding actual hop distances in the network graph as a whole.

10 Conclusion

Throughout this project we were able to increase our knowledge about NS-2, AODV, *k*-SPR and many other routing protocols. We learned how a message would be sent from node A to node B and how it would decipher which node it should choose to send it to get there (the middle nodes). The AODV-*k*-SPR, has many advantages like performing tasks only when told to do so and finding the shortest distance between two points, which we believe help make sending a message from node to node to be more efficient and thus take less time.

After months of working on this project, we feel that we have a greater understanding of how a networking system would work, and the best possible way to have it set up. The newly developed AOVD / *k*-SPR routing protocol is the main reason that this project was so successful. At the beginning of the project, the routing protocol was merely an idea, but now at the end it seems to be fully functional.

We hope to continue our research collecting data having to do with the amount of over head involved in sending out route requests in connection with AODV *k*-SPR vs. AODV, also looking at throughput and time delays. It is our hope that someday we can make this process even more efficient than we have it now, to help allow people to send large messages at a faster rate.

References

[1] Tcl Built-In Commands – array (n).
http://www.astro.princeton.edu/~rhl/Tcl-Tk_docs/tcl18.0a1/array.n.html

[2] S. Park. NS-2 Tutorial. http://www.isi.edu/nsnam/ns

[3] A.A Abouzeid. Ns Tutorial.
http://www.isi.edu/nsnam/ns/ns-documentation.html

[4] A. Zang. NS2 Tutorial. http://dcl.ee.ncku.edu.tw.~aga/ns2.htm

[5] S. Wen. ns-2 Network Simulator.
http://protocols.netlabs.uky.edu/~suwen/paper/ns-short.ppt

[6] D. Anderson and X. Yang. ns-2 Tutorial, part 2.
http://www.nms.tcs.mit.edu/6.829_fol/ns_tutoials.ppg

[7] Y. Xu. NS Tutorial: mobile and wireless network simulation.

http://www.isi.edu/nsnam/ns/ns_tutorial/wireless.ppt

[8] P. Haldar. Wireless world in NS. http://wcc.iiita.ac.in/ns/wireless_world_in_ns.ppt

[9] X. Leining. How to Add a New Protocol in NS2. http://www.cs.tcl.ie/~htewari/papers/ns-extentx/ming.ppt

[10] M.Q. Rieck, S. Dhar, J. Kim, and S. Pai. Recent directions in wireless networking- Ad-Hoc routing strategies using k-dominating hop sets. *Power point presentation at IUCSC 2004 at Drake University.*

[11] Ad hoc On-Demand Distance Vector Routing. http://moment.cs.ucsb.edu/AODV/aodc.html

[12] R. Shorey. Multi-Hop Wireless Ad Hoc Networks. From *The 6th International Conference/Exhibition on High Performance Computing In Asia Pacific Region.* December 16-19, 2002. http://www.cdacindia.com/hpcasia2002

[13] L. Klein-Berndt. A Quick Guide to AODV Routing. *National Institute of Standards and Technology.* US Department of Commerce.

[14] S. Dhar, M.Q. Rieck , S. Pai and E.J. Kim. Distributed Routing Schemes for Ad Hoc Networks Using d-SPR Sets. *J. or Microprocessors and Microsystems: Special Issue on Resource Management in Wireless and Ad Hoc Mobile Networks*. Elsevier, v. 28, no. 8, October 2004. Pages 427-437.

[15] C.E. Perkins and E.M. Royer. Ad-hoc On-Demand Distance Vector Routing. *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*. New Orleans, LA, February 1999. Pages 90-100.

[16] S.J. Lee, C.E. Perkins, and E.M. Royer. Scalability study of the ad hoc on-demand distance vector routing protocol. *International Journal of Network Management*. 2003, v. 13. Pages 97-114.

[17] E.M. Royer. Routing in Ad hoc Mobile Networks: On-Demand and Hierarchical Strategies. *Ph.D dissertation*.

[18] Charles E. Perkins, Elizabeth M. Belding-Royer, and Ian Charkeres. Ad Hoc On Demand Distance Vectors (AODV) Routing. *IETF Internet draft*, draft-perkins-manet-aodvbis-00.txt, Oct 2003 (Work in Progress).