Hierarchical Routing using k-SPR

Skyler Nesheim and Matt Smith
Department of Mathematics and Computer Sciences
Drake University
2507 University Avenue
Des Moines, Iowa 50311
michael.rieck@drake.edu

Abstract

This research explores hierarchical routing strategies in AODV/ k-SPR by implementing them in U. C. Berkeley's NS-2 network simulator. Creating a hierarchical system of routing necessitated altering of the C++ code associated with AODV protocols and also the generating of C++ code to do a number of tasks connected with creating such a system. More data must be collected and analyzed in order to determine the most efficient method of applying these new protocols for routing.

Last year's research

Last year, two Drake students, Tiffany Meredith and Jennifer Ehrlich started working with the NS-2 simulator to create graphs and perform actions on these graphs. Their research yielded several interesting results including the development of a new wireless routing protocol which they named k-SPR.

Ehrlich and Meredith's version of k-SPR used a router selection algorithm called k-SPR-I to select routers and passed messages across a network using a hybrid of the k-SPR algorithm and the AODV algorithm. They also used a particular C++ program in the NS-2 simulator named God.cc along with the existing AODV code to help them implement k-SPR/AODV.

Our research begins where theirs left off.

Introduction

Supported by a grant from the Maytag Corporation, which is in conjunction with the Iowa College Foundation (ICF), we have spent the 2005-2006 academic year researching various wireless routing protocols and have even implemented some of our own.

Our research began in the fall when we worked with a network simulator designed by Eun Jik "Jake" Kim of San Jose State University. We used Jake's Simulator to gain a basic understanding of router selecting algorithms such as k-SPR-I, k-SPR-C, and the Greedy Algorithm. Our goal was to create a graphical Java program that would display the data created by Jake's Simulator. After completing this we focused on upgrading Jakes code to account for a weighted edge graph. This proved to be a challenging process, and we were unable to complete the work with Jake's Simulator, since we needed to focus on the NS-2 simulator.

After exhausting our research in Jake's Simulator we moved to working with the NS-2 simulator developed at the University of California in Berkley. We began by researching the code developed by two students from last year's wireless researching project, Jennifer Ehrlich and Tiffany Meredith. Our goal was to work with their code and create our own wireless routing protocol based on a paper entitled *Hierarchical Routing in Sensor Networks Using k-Dominating Sets* [3]. We devoted the remainder of the Fall 2005 Semester to learning some of the basics behind NS-2 and researching last years changes to the code.

When we returned from Winter Break we were ready to start coding in the NS-2 environment. We devoted our Spring 2006 semester to developing our own wireless routing algorithm which we named k-SPR-C-HIER. Basically this stands for the k-SPR algorithm using 'covering numbers,' a notion related to k-SPR sets, and implementing it in a hierarchical fashion. To do this we had to add several operations to the current NS-2 code. We were especially interested in developing hierarchical version of the various protocols. First we needed an operation to find the covering numbers of all the nodes in the graph. We also created a method for finding the covering numbers of routers. After coding both of the covering number algorithms we were able to choose router and super routers based on their covering numbers.

Working with Jake's Simulator

Before beginning work on NS-2, Nesheim and Smith first focused on a program designed by Eun Jik ("Jake") Kim. Jake's simulator was written in Java and basically created a graph with a given number of nodes. The simulator then applied a series of router selecting algorithms on these nodes and printed information about how the algorithms performed to an output file. In the Fall of 2005 Nesheim and Smith spent several weeks analyzing the Java code for Jake's Simulator.

Professor Rieck asked Nesheim and Smith to incorporate the DrawGraph Java program designed by Ehrlich and Meredith last year with Jakes Simulator. This proved to be a pretty easy task. First, the students added Java Code to Jake's Program that printed the necessary output for the DrawGraph program. Three files were needed in order for DrawGraph to work; Adjacency.dat, Coordinates.dat, and Routers.dat. Because Jake's Simulator provided for no animation that portion of the previous DrawGraph program was deleted. The new Java Program was named DrawGraphForJake.

Nesheim and Smith next set their sights on adding weights to the edges of the nodes in Jakes Simulator. Previously, Jakes Simulator assumed a weight of one for each edge. First Nesheim and Smith assigned values to each edge in the graph and incorporated this into the DrawGraphForJake simulator.

The students then aimed to teach the routing algorithms how to incorporate weights into their process. Unlike the creation of the DrawGraphForJake program, this proved to be very challenging. Nesheim and Smith were not able to perform this task due to memory issues inherently programmed into Jakes Simulator.

There were however some very positive outcomes of the early stages of the project for us. First of all, we developed a new version of the DrawGraph program which proved to be very useful later when we were debugging our additions to the NS-2 simulator. Second, we were able to gain hands on knowledge of router choosing algorithms early in our research which saved us time when we actually started coding in NS-2.

Installing the NS-2 Simulator

Similarly to the experience with last year's project, Nesheim, Smith, and Rieck had trouble installing the NS-2 simulator. Specifically, there were issues in installing tcl and tk on the computers in the Sheppard Computer Lab at Drake. In their seemed that NS-2 was not compatible with the version of Linux installed on these machines, Fedora Core 3.

During this time Nesheim and Smith were deeply involved in Jake's Simulator so Professor Rieck took it upon himself to find a way to install NS-2. His solution was to install the program on a Windows XP computer in the lab. Frankly, he was left pulling his hair out over this issue as Windows just doesn't have the ease of compiling C++ code like Linux does.

After failing to revamp Jake's Simulator, Nesheim found a bootable version of Knoppix which had NS-2.26 pre-installed on it. Knoppix is a version of Linux which fits onto one CD and is considered to be a "live CD." Essentially a "live CD" can be booted when the computer is turned on without affecting the current operating system. The version used by Nesheim, Smith, and Rieck is called ZouriX v2.0 and was found at www.infotech.tu-chemnitz.de/~knoll/NS2/ZouriX.php.

There was only one issue to this solution. In order to change the C++ code incorporated into NS-2, this version of Knoppix would need to be installed on a computer in the lab. It turned out that ZouriX also provided a simple hard drive install script, which installed the current system onto the hard drive of the computer. Nesheim installed the operating system on a Pentium 3 computer in the lab. Professor Rieck was then able to transfer last year's changes to this computer and recompile NS-2. Because of ZouriX, Nesheim, Smith, and Rieck were able to start investigating the NS-2 code with almost no delay.

Diagnostics and Debugging

After installing NS-2 Nesheim and Smith immediately began examining the code developed by Meredith and Ehrlich. The first thing they determined was that the animation in DrawGraph, developed by Meredith and Ehrlich, was inadequate when showing the route of a message through the graph. Instead of using an animated GUI, Nesheim and Smith opted for a GUI that was simply a picture and added more diagnostic messages to the NS-2 code to further test last years results.

As mentioned early in this paper, the adapted Java program developed by Nesheim and Smith to work with Jake's Code became very useful later in the project to debug NS-2 coding changes. The Java program however, had to be updated. First off, we added scroll panes on the side and bottom of our GUI (graphical user interface) so that we could see the picture on any size monitor. We then updated the code so it would display ID numbers for each node in the graph. We also changed the color scheme to make the picture more readable. In our current DrawGraph program the background is grey, the ID numbers are black, the regular nodes are black, the lines connecting the nodes are black, the routers are red, and the super routers (we'll talk about these later) are cyan.

To better understand k-SPR/AODV we added many diagnostic messages to the NS-2 simulator. We used pre-processor definitions in the k-SPR C++ code to display text in the terminal window explaining what was happening with the nodes in the graph. For instance we might have printed a message that said, "Node 23: I received an AODV message." Using the draw graph and the diagnostic messages we were able to retrace the steps of an AODV route request through the graph.

Some Important Definitions

It is important to understand the various naming standards we use before continuing in this paper. Several times in the following pages we will speak of k, k-SPR, route requests, route replies, routing tables, God.cc, and AODV. This section aims to define these important elements of our research.

A k-SPR set, SPR standing for shortest path routing, is a set of nodes such that if any two nodes are more than k hops apart, there is some other node in the set such that it is along the k+ hop path between those two nodes. Every node in a k-SPR set knows all the nodes within k hops of itself. Two nodes are referred to as a 'pair' if they are exactly k+1 hops apart. Pairs become relevant when selecting what is called a 'router.' A router is a node that has a larger view of the graph than a regular node; however, a router may still act as a regular node. A node 'covers' a pair if it lies on a shortest path between the nodes of that pair, however, the nodes of the pair are not considered to cover their own pair. The 'covering number' of a node is the number of pairs that node covers.

We refer to k-SPR quite a bit in this paper. However, there are several reasons we may mention it. On its highest level k-SPR is the wireless routing protocol suggested in *Distributed Routing Schemes for AdHoc Networks Using d-SPR Sets* [1]. We also refer to the “router selection process,” “router election process,” or “router selection algorithm” using k-SPR-I or k-SPR-C. There is a major difference between a wireless routing protocol and a router selection algorithm. The routing protocol mainly deals with how the nodes in a particular graph are able to pass messages to each other. The router selection algorithms aim to select or elect a set of routers from a set of nodes in the graph.

There are several types of messages we might like to send throughout a graph. We inherit the names for these messages for AODV. A route request (RREQ) is necessary if one node wants to talk to another node, but doesn’t know a path between them. A RREQ has, among others, a source and a destination field. A route reply (RREP) is the message a node sends back to another node if it is the destination of a route request or if it knows a route to the destination.

The routing table is another feature that the k-SPR protocol inherits from AODV. Basically the routing table contains information about all the paths to other nodes that a particular node knows about. Of particular interest, this table can be used to determine if a route is up to date.

When explaining how the k-SPR protocol works we will discuss an important feature of the NS-2 simulator God.cc. God.cc is an all knowing portion of the C++ program which allows us to cheat when we implement k-SPR. God.cc has functions which return information about the graph such as the number of hops from one node to another or a list of a node’s k-hop neighbors. What do we mean by cheat? By definition, in the k-SPR protocol each node must maintain a list of its neighbors in its k-hop neighborhood. In a real network an all knowing node or program would not be implemented. So far however, the initialized phase of a real network has not been simulated. Instead, a node simply asks God.cc for a list of his k-hop neighbors or other information it can't compute on its own. So we aren't really cheating, but we are implementing k-SPR in a way that allows us to skip the initialization for the time being.

Much later in this analysis we will speak of AODV, which stands for Ad Hoc On Demand Vector Routing Protocol. AODV is defined in *Ad-hoc On- Demand Distance Vector Routing* [2]. We will speak more as to how k-SPR is designed with some of the features of AODV, we however did not code AODV in the NS-2 network simulator, it came pre-coded.

Understanding the Previous Protocol

The k-SPR routing protocol was originally designed to work with a set of routers. Previously, the path of a route request through the network was fairly simple. First a node, Bill, decided that he wanted to send a message to another node, Ted. The problem is that Bill doesn't know where Ted is in the graph. First, Ted asks God.cc for a list of all his k-hop neighbors. If Ted is in that set, then Bill simply forwards his message to him, but if Ted is not in that set, Bill sends a route request to all the routers in his local k-hop neighborhood. Once a router, Jane, in Bill's k-hop neighborhood receives the node she checks to see if Ted is in her k-hop neighborhood. If he is, Jane can forward the route request to him; otherwise Jane will pass the route request to all the routers in her k-hop neighborhood. This process continues until Ted receives the message.

Once Ted receives the route request from bill he sends a route reply message back to Bill along the same path that the route request was sent to him. When Bill gets Ted's reply message he stores the path to Ted in his node information, this means that Ted's route is up to date by Bill's information. Bill now knows the path along which to send a message to Ted.

Plan for Spring 2006 Semester

After returning from our winter break, Nesheim and Smith felt rested and ready to begin coding in the NS-2 Simulator. It became evident that we would need a plan for attacking our goals in the Spring of 2006. First we wanted to implement an algorithm that would return the covering numbers for all the nodes in the graph. Next, we wanted to implement the k-SPR-C version of the router election process using our new covering number algorithm. Finally, our official goal for the spring semester was to implement a hierarchical version of the k-SPR wireless routing protocol using the NS-2 simulator and AODV as a backbone.

How to choose a set of routers

For our purposes routers are chosen in one of two ways. The first version is based on the ID of the nodes and is called k-SPR-I. This version of the algorithm was implemented by

Ehrlich and Meredith last year. The other version is called k-SPR-C and is based on the number of pairs a router covers or blocks.

Mathematically, a node z covers a pair of nodes x and y if distance(x,z) + distance(y,z) = distance(x,y). This property must be satisfied for a node to cover a pair of nodes.

When selecting routers using k-SPR-I we examine all the pairs and determine which nodes cover them. Then we choose a router based on which one has the highest ID number.

To choose routers based on covering number, as in k-SPR-C, we first loop over all the nodes and determine how many pairs they cover or block. Then we loop over all the pairs and choose the node which covers a given pair, and has the highest covering number.

Determining the Covering Numbers of Routers

In principle, routers' covering numbers are computed in much the same way as a node's covering number is determined. The essential difference comes in consideration of the graph that is used in order to determine those numbers, which slightly changes the rule by which pairs are selected. Here, we should make it clear that the covering number of a router is different from the covering number of a node. Though a router is also a node and thus has a covering number as a node, it *also* has a covering number as a router, which is entirely independent of its covering number as a node. The router covering number is based upon what are called 'router pairs,' which we will define presently.

A pair of routers may not necessarily be a specific number of hops from each other, as a node pair is, but they are still considered a pair because they are within a certain range of each other. Much like with node pairs, a router pair must be a distance apart that is significant to the functioning of the graph in a routing manner, however, since all routers are already about k hops apart, k cannot be the standard. So the condition of being at least k^2 hops apart becomes a useful criteria since we are , essentially, looking at a graph of routers as though it were a graph of nodes and k^2 hops is equivalent to k 'router hops,' which is considered to be k hops for simplicity.

The maximum distance between router pairs must also be limited so as to ensure that covering numbers are not inflated by routers that are very far apart hop-wise. In keeping with the idea that the graph of routers is similar in principle to that of the nodes, the range of a router pair should be about k 'router hops' apart, no more, no less. We have already examined what it means to be at least k router hops apart, now we must make sure we are not more than k router hops apart. Since a router hop is about k hops, the range from the minimum of k^2 hops, which is k router hops, should be at most k additional hops, thus yielding the maximum range of a router pair, that is k^2 + k hops.

From this criteria, it is necessary to further confine what a router pair is, attaining what will be hereafter referred to as a minimal router pair. If it should happen that there is a router along the shortest path between a router pair such that that router also satisfies the criteria of a router pair in conjunction with one of the routers in the original router pair, then the original router pair is not a minimal router pair. This does not mean that the second router pair is minimal, only that it is closer to minimal than the original router pair. If it so happens that the router pair in question has no routers along the shortest path between its two routers such that it pairs with either router of the router pair, then that router pair is minimal.

Minimal router pairs are most pertinent because they represent those pairs of routers which most closely form the ideal graph for routing. We select the minimal router pair because it does not inflate the covering number of the routers superficially. Were we to select any router pair that satisfies router pair criteria, the same paths would be run over multiple times, only with one or two additional routers in them, in calculating the covering numbers of the routers, thus overstating the use of some routers in connecting the network.

With the set of minimal router pairs, the covering numbers are calculated almost exactly as they are in the case of nodes. Starting with each minimal router pair, every router is tested to see if it lies on the shortest path between the minimal router pair, ignoring those routers that are actually in the pair. If a router lies on the shortest path between the minimal router pair, its covering number is incremented. After this has been performed for each minimal router pair, the covering number for each router has been determined.

Applying k-SPR-C to Choose a Set of Super Routers

To select super routers, we must first find the minimal router pairs, as described heretofore. By then interrogating every router, excluding those in the minimal router pair, we learn whether or not it is along the shortest path between the minimal router pair and also it’s covering number. The id and covering number of the first router that is along the shortest path of the minimal router pair in question are stored for the remainder of the loop.

Whenever another router is encountered that is also along the shortest path of the minimal router pair, its covering number is compared to the stored value. If its covering number is less than the stored value, the loop goes on to look at the next router in queue. If its covering number is greater than the stored value, its id and covering number are stored, overwriting the previous information. If its covering number is the same as the stored value, the id of the router is then compared to the stored id, and if the router's id is greater, its id is stored, overwriting the previous value.

When this loop has exhausted the list of routers, the id that was last stored is marked as being a super router using a boolean variable in an array called is_super_router. Then the loop starts again for the next minimal pair, again marking the super router selected. It is possible that the same router will be marked as a super router several times, however, since this is stored via a boolean variable, this does not present a problem. That this happens is not only beneficial to the routing, but also necessary since there are likely to be more minimal router pairs than routers and there cannot be more super routers than routers. This system for selecting super routers ensures that any router will be within about k^2 hops of a super router by creating a super router between any two routers that satisfy the criteria for a minimal router pair.

Teaching NS-2 to Work With Super Routers

As stated earlier, the k-SPR routing protocol was designed to work with a set of routers. In order to complete our goal for the semester our final task would be to re-engineer NS-2 and the k-SPR protocol to work with a set of super routers.

The path of a route request in a graph with super routers is a bit different. We want to be very sure that we don't miss the destination we are trying to send the message to. We also want to make use of the super routers and avoid fanning a message throughout the graph. In general, our goal with k-SPR is to minimize the number of messages send across the network, not increase them.

Again we will use the nodes Bill and Ted as an example, but this time Bill and Ted are in the super router world. Bill wants to talk to Ted, but he doesn't know a route to him. So, Bill checks his local k-hop neighborhood, if Ted lives there, Bill simply forwards the request to him, but if Ted doesn't live in Bill's neighborhood, Bill must forward it to all the surrounding routers. This is similar to how things were done in the past, but now when a router, Jane, receives a message she looks for Ted in her extended neighborhood. An extended neighbor hood is equal to K (k*(k+1)+1). So, if Ted isn't in Jane's extended neighborhood she will forward the request to all the super routers in her K neighborhood. To forward this message Jane will first send it to the closest router on the shortest path to a super router, Mike. The message is passed between the routers along the path to Mike until he receives it. When Mike receives it he checks his local K neighborhood to see if Ted lives there. If not Mike passes the message to all the super routers in his K neighborhood.

This process for sending a message across the network is much more complex than using the non-hierarchical version of k-SPR. It is easy to explain the process of sending a message across the network on paper, but some very important questions arise at the implementation level. How does a router or super router know about all the nodes in their K-hop neighborhood? How is the path to the next super router determined? How is the next router along this path found?

The answer is by using God.cc to cheat. God.cc can provide a list of neighbor nodes for a given node within any arbitrary number of hops. The path to the next router can be used by implementing God.cc's next-router algorithm in conjunction with the next-hop method. The next router returns the next router on the shortest path between any kind of node and a super router. Then using that next router the next-hop method returns the next node on the path to a particular router, which is on the path to a particular super router.

Information about the next node, router, or super router is maintained in the header information of the message as it is sent. Each message has three types of headers an IP header, an AODV header, and a RREQ header. The IP header contains information about who the next router is and who the previous router was, the AODV header contains the previous super router and next super router fields, and the RREQ header contains the previous node and next node hop information. Using these headers a node can determine whether it is the next node, next router, or next super router.

Conclusion

From our tests we found that simulating a network that would need super routers takes up a great amount of memory very quickly. The reason for this becomes clear as we look at the criteria used for selecting super routers. Recall that in order for a super router to be selected, it must be covering a router pair, which is a pair of routers whose shortest path is k^2 hops and at most k*(k+1) hops. Even for a small k, the number of hops this necessitates for a router pair to exist makes it somewhat unlikely to find such a pair, and hence to select a super router.

For an explanation of this, we look to the principle behind “six degrees of separation.” Since that theory deals with social-networking, and is based in mathematical principle, it correlates almost directly to our work. The idea is that every person, or node, on average knows, or is within one hop of, x people, each of which also knows about x people, so the person knows about x^2 people (about is used because the person who is known also knows the person who knows him or her) by 2 degree of separation, or 2 hops. So, given that every node in our network is one hop away from about x other nodes, it follows that in order for two nodes to be at most k hops apart would require the presence of at least x^k nodes.

When we start to look for a router, we should have at least x^k nodes, which is not hard to accomplish if k is 2 or 3, a common choice in our testing, however, when we begin the search for super routers, we see that it is necessary to have at least x^(k^2) nodes in order to *guarantee* that there will be a router pair and thus a super router. If k is 3, as might often be the case, and we assume that each node is one hop apart from at least 3 nodes, which is a reasonable, if not low, estimate, then there must be at least 3^9, or 19,683, nodes to ensure that a super router can be selected. This is because the router pair must be at least 9 hops apart and at most 12 hops apart.

If we were to attempt to select a new type of router that is, in essence, a 'super router router', or a super duper router, as some may like to call it, we would have to find a pair of super routers that are at least k^3 hops apart and at most k*(k*(k+1) +1) hops apart. Looking at the previous example of k being 3 and each node being one hop apart from about 3 other nodes, the graph would have to have at least 3^27, or 7,625,597,484,987, nodes to guarantee the selection of a 'super duper router.'

Due to memory restrictions, we are unable at this time to simulate a graph of 20,000 nodes, however, we have simulated graphs with about 45 nodes in which k is 2 and we were able to select super routers. Since 3^4 is 81, assuming each node knows at least 3 other nodes, we are not guaranteed the selection of super routers, however, since not all neighbors of neighbors are necessarily distinct, the number of hops between any two given nodes is not ideal. In other words, not all graphs will be ideally connected for routing purposes, thus super routers will be selectable in many graphs so long as k does not get too large.

Furthermore, as the number of nodes in a graph and the diameter of that graph tend closer to one another, the purpose of routing becomes void. If the greatest number of hops between any two nodes in the graph, what is called the diameter of the graph, is very nearly equal to the number of nodes in the graph, it must be that the graph is, or is nearly, linear. Since the number of hops between two nodes is so great, it becomes much easier to select a super router, or even a super duper router, but neither of those would serve a purpose because all messages would have to travel linearly, and thus pass through those nodes anyways.

Noting this trend in graphs speaks to the point of making router selection a *generalized* hierarchical system as being moot. It could be argued that a graph that warrants super duper routers might easily come up, or some tier further along the hierarchy, but such a graph would likely be inefficient or monstrously large. In the case where it is monstrously large, it must still be finite and thus only in need of a finite hierarchy of routers, but adding one tier of routing generates a greatly increased criteria for selecting that tier of routers so that at some point the criteria will become too lofty. This does not, however, answer the question of how many tiers are appropriate for a standard system of ad-hoc wireless networking.

The main problem in creating a generalized hierarchical system of routing is selecting an appropriate criteria by which to stop generating tiers of routing. It seems that a tier of routers ought not be selected if the diameter of the tier before it, that is the greatest number of hops between any two routers in that tier, is sufficiently close to the number of routers in that tier. What constitutes 'sufficiently close,' is much more difficult to say and would require further research.

Our typical day during this research was divided into thirds. The first third we would normally spend writing pseudocode, working out kinks in our reasoning behind the structure of what we were doing. The second third was spent writing the actual code to ape the pseudocode. Finally, the last third was spent figuring out that what we had written was actually working how we thought it was. In truth, the final third was often more than half our time and once in a while involved feeling that the code did not work only to find out later that it was and that it simply had not been telling us so. Our research was often challenging and days would go by when we were convinced that much of what we had done up until that point had been useless, but in retrospect every one of those days helped us figure out the direction we needed to go. Incidentally, some of those days were fun, too. The more we think about this project, the more we want to see it realized and applied and we look forward to the day when ad-hoc wireless networking becomes an everyday practice in the computing world.

References

[1] S. Dhar, E.J. Kim, S. Pai and M.Q. Rieck. Distributed Routing Schemes for AdHoc Networks Using d-SPR Sets. Pages 1 to 10.

[2] C.E Perkins and E.M. Royer. Ad-hoc On- Demand Distance Vector Routing. Pages 2 to 11

[3] S. Dhar and M.Q. Rieck. Hierarchical Routing in Sensor Networks Using k-Dominating Sets.

[4] S. Dhar, S. Pai and M.Q. Rieck. Distributed Routing Algorithms for Multi-hop Ad Hoc Networks Using d-Hop Connected d-Dominating Sets. From *Computer Networks Journal.* April 2005.

[5] S. Dhar, E.J. Kim, S. Pai and M.Q. Rieck. Distributed Routing Schemes for Ad Hoc Networks Using d-SPR sets. From *Journal of Microprocessors and Microsystems.* October 2004.