Design and Implementation of Fisheye Routing Protocol for Mobile Ad Hoc Networks

by

Allen C. Sun

Submitted to the Department of Electrical and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 14, 2000

Author____

Department of Electrical Engineering and Computer Science
May 14, 2000

Certified by____

Amar Gupta
Thesis Supervisor

Accepted by____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Design and Implementation of a Fisheye Routing Protocol
For Mobile Wireless Ad Hoc Networks
by Allen C. Sun

Submitted to the
Department of Electrical Engineering and Computer Science

May 14, 2000

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Wireless networking is an emerging technology that will allow users to access information and services regardless of their geographic position. In contrast to infrastructure based networks, in wireless ad hoc networks, all nodes are mobile and can be connected dynamically in an arbitrary manner. All nodes of these networks behave as routers and take part in discovery and maintenance of routes to other nodes in the network. This feature presents a great challenge to the design of a routing scheme since link bandwidth is very limited and the network topology changes as users roam. This thesis investigates the behavior of existing traditional routing algorithms and proposes and implements a new routing approach for ad hoc wireless networks: Fisheye Routing. Fisheye Routing is similar to Link State routing, but uses a fisheye technique to reduce the consumption of bandwidth by control overhead.

Thesis Supervisor: Amar Gupta
Title: Co-Director, Productivity from Information Technology(PROFIT) Initiative

Table of Contents

1 Introduction

1.1 Background

Wireless networking is an emerging technology that will allow users to access information and services electronically, regardless of their geographic position. The use of wireless communication between mobile users has become increasingly popular due to recent performance advancements in computer and wireless technologies. This has led to lower prices and higher data rates, which are the two main reasons why mobile computing is expected to see increasingly widespread use and applications.

There are two distinct approaches for enabling wireless communications between mobile hosts. The first approach is to use a fixed network infrastructure that provides wireless access points. In this network, a mobile host communicates to the network through an access point within its communication radius. When it goes out of range of one access point, it connects with a new access point within its range and starts communicating through it. An example of this type of network is the cellular network infrastructure. A major problem of this approach is *handoff*, which tries to handle the situation when a connection should be smoothly handed over from one access point to another access point without noticeable delay or packet loss Another issue is that networks based on a fixed infrastructure are limited to places where there exists such network infrastructure.

The second approach is to form an ad-hoc network among users wanting to communicate with each other. This means that all nodes of these networks behave as routers and take part in discovery and maintenance of routes to other nodes in the network. This form of networking is limited in range by the individual nodes transmission ranges and is typically smaller compared to the range of cellular systems. However, ad-hoc networks have several advantages compared to traditional cellular systems. The advantages include 'on-demand' setup, fault tolerance, and unconstrained connectivity.

A key feature that sets ad-hoc wireless networks apart from the more traditional cellular radio systems is the ability to operate without a fixed wired communications infrastructure and can therefore be deployed in places with no infrastructure. This is useful in disaster recovery, military situations, and places with non-existing or damaged communication infrastructure where rapid deployment of a communication network is needed.

A fundamental assumption in ad-hoc networks is that any node can be used to forward packets between arbitrary sources and destinations. Some sort of routing protocol is needed to make the routing decisions. A wireless ad-hoc environment introduces many problems such as mobility and limited bandwidth which makes routing difficult.

This thesis researches existing traditional routing protocols, examines current proposed mobile ad-hoc routing protocols, and then designs and implements a functional link-state routing protocol employing a novel "fish-eye" updating mechanism specific for a wireless infrastructure. This mechanism is then analyzed to evaluate its effectiveness and the advantages it can offer.

1.2 Scope of Research

The objective for this master thesis was to design and implement a routing protocol for wireless ad-hoc networks. In this environment, the routing strategy must scale well to large populations and handle mobility. In addition, the routing protocol must perform well in terms of fast convergence, low routing delay, and low control overhead traffic.

This first involved researching existing routing protocols to determine their strengths and weaknesses. Using this analysis, a new mechanism was developed that enhances routing in the mobile ad-hoc environment. This mechanism was then implemented in a software environment. Once the implementation was completed, simulation and analysis of the protocol was performed to evaluate the advantages of the new mechanism.

The goals are of this thesis are as follows:

- Get a general understanding of ad-hoc networks
- Study existing and proposed routing protocols.
- Develop a new mechanism that offer advantages for routing in wireless ad-hoc networks.
- Implement the routing protocol.
- Analyze the protocol theoretically and through simulation.

Section 2 explains ad-hoc networks and routing in general. Section 3 describes current proposed ad hoc routing protocols. Section 4 describes the design and implementation of a new routing protocol and performance analysis. Section 5 gives summary and conclusions. Section 6 includes references used and section 7 (Appendices) include the listing for the code.

2 Routing in Wireless Networks- General Concepts

2.1 Wireless Ad-Hoc Networks

A wireless ad-hoc network is a collection of mobile nodes with no pre-established infrastructure. Each of the nodes has a wireless interface and communicates with others over either radio or infrared channels. Laptop computers and personal digital assistants that communicate directly with each other are some examples of nodes in an ad-hoc network. Nodes in the ad-hoc network are often mobile, but can also consist of stationary nodes.

Figure 1 shows a simple ad-hoc network with three nodes. The outermost nodes are not within reception range of each other and thus cannot communicate directly. However, the middle node can be used to forward packets between the outermost nodes. This enables all three nodes to share information and results in an ad-hoc network.

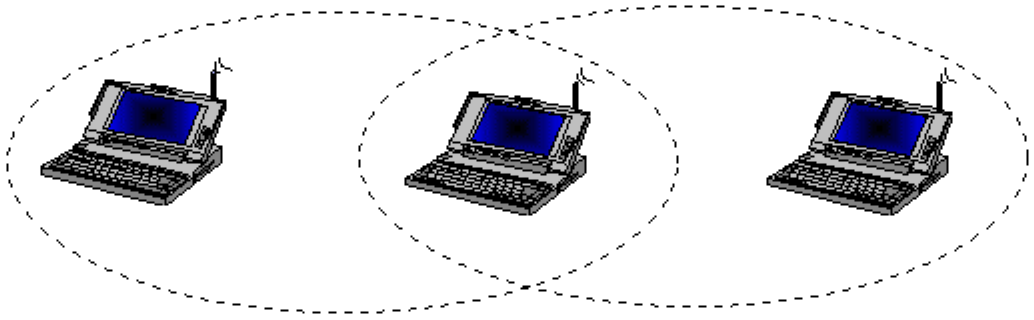


Figure 1: Example of simple ad-hoc network.

An ad-hoc network uses no centralized administration. This ensures that the network will not cease functioning just because one of the mobile nodes moves out of the range of the others. Nodes should be able to enter and leave the network as they wish. Because of the limited transmitter range of the nodes, multiple hops are generally needed to reach other nodes. Every node in an ad-hoc network must be willing to forward packets for other nodes. Thus every node acts both as a host and as a router.

The topology of ad-hoc networks varies with time as nodes move, join, or leave the network. This topological instability requires a routing protocol to run on each node to create and maintain routes among the nodes.

2.1.1 Usage

There is a plethora of applications for wireless ad-hoc networks. Wireless ad-hoc networks can be deployed in areas where a wired network infrastructure may be undesirable due to reasons such as cost or convenience. It can be rapidly deployed to support emergency requirements, short-term needs, and coverage in undeveloped areas. Examples of such situations include disaster recovery, search and rescues, or military applications [RT99].

Other usage includes convenience, such as allowing conference members or business associates to exchange documents, or accessing the Internet or resources such as printers. The applications are boundless.

2.1.2 Characteristics

Ad-hoc networks are often characterized by a dynamic topology due to the fact that nodes change their physical location by moving around. Another characteristic is that a node have limited CPU capacity, storage capacity, battery power, and bandwidth. This means that power usage must be limited thus leading to a limited transmitter range.

The access medium, usually a radio environment, also has special characteristics that must be considered when designing protocols for ad-hoc networks. One example of this may be uni-directional links. These links arise when two nodes have different strengths on their transmitters (allowing only one of the host to hear the other) or from disturbances from the surroundings. Multi-hop in a radio environment may result in an overall transmit capacity gain and power gain due to the squared relation between coverage and required output power. By using multi-hop, nodes can transmit the packets with much lower output power (by transmitting to closer neighbors).

2.2 Routing

Routing is a function in the network layer which determines the path from a source to a destination for the traffic flow. A routing protocol is needed because it may be necessary to traverse several nodes (multi-hops) before a packet reaches the destination. The routing protocol's main functions are the selection of routes for various source-destination pairs and the delivery of messages to their correct destination. In wireless networks, due to host mobility, network topology may change from time to time. It is critical for the routing protocol to deliver packets efficiently between source and destination. Routing protocols can be divided based on when and how the routes are discovered into two categories: Table-Driven and On-Demand routing [RT99].

In table-driven routing protocols, each node maintains one or more tables containing routing information to every other node in the network. All nodes update these tables so as to maintain a consistent and up-to-date view of the network. When the network topology changes the nodes propagate update messages throughout the network in order to maintain a consistent and up-to-date routing information about the whole network. Routing protocols in this category differ in the method by which the topology change information is distributed across the network and the number of necessary routing-related tables. The two main types of table-driven routing are: Distance Vector and Link State [PB96].

In on-demand routing, all up-to-date routes are not maintained at every node, instead the routes are created when required. When a source wants to send to a destination, it invokes a route discovery mechanisms to find the path to the destination. The route remains valid util the destination is unreachable or until the route is no longer needed. A typical type of on-demand routing is Source Routing [BJ98].

2.2.1 Distance Vector

Distance vector routing is sometimes referred to as Bellman-Ford, after the people who invented the algorithm. In the distributed Bellman-Ford algorithm [Per00], every node ***i*** maintains a routing table which is a matrix containing distance and successor information for every destination ***j***, where distance is the length of the shortest distance from ***i*** to ***j*** and successor is a node that is next to ***i*** on the shortest path to ***j***. To keep the shortest path information up to date, each node periodically exchanges its routing table with neighbors. Based on the routing table received with respect to its neighbors, node ***i*** learns the shortest distances to all destination from its neighbors. Thus, for each destination ***j***, node ***i*** selects a node ***k*** from its neighbor as the successor to this destination(or the next hop) such that the distance from ***i*** through ***k*** to ***j*** will be the minimum. This newly computed information will then be stored in node ***i***'s routing table and will be exchanged in the next routing update cycle.

Figure 2 shows an example of Distance Vector Routing. This example focuses on everyone's distance to destination D. D transmits its distance vector (next(D)=D) with cost 0 (dist(D)=0) to node 1. Now, Node 1 calculates its distance vector to D as one (dist(D)=1) through D (next(D)=D) and transmits this information to nodes 2 and 3. This process continues until all the nodes have a cost and next hop information to D.

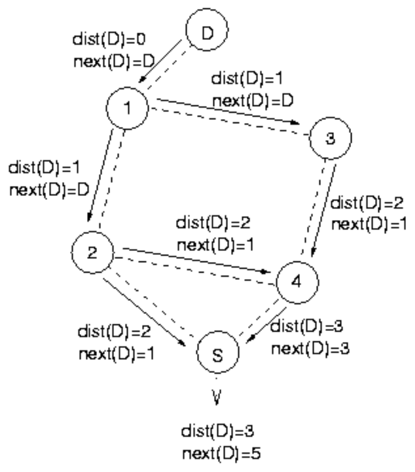


Figure 2: Example of Distance Vector Routing

The advantages of Distance Vector are its simplicity and computation. However, the chief problem with distance vector routing is its slow convergence when topology changes [BG87]. The primary reason for this is that the nodes choose their next-hops in a completely distributed manner based on information that can be stale. While routing information has only partially propagated through the network, routing can be seriously disrupted. This may lead to formations of both short-lived and long-lived routing loops [Per00].

An example of a routing loop is shown in Figure 3. This example will focus on everyone's distance to destination C. B calculates its distance to C as 1 (Dist(C)=1) and A calculates its distance to C as 2 (Dist(C)=2) through B (Next(C)=B). When the link between B and C breaks, B must recalculate its distance vector to C. Unfortunately, B does not conclude at this point that C is unreachable. Instead, B decides that it is 3 from C, based on distance vector information from A. Because B's distance vector has now changed, it transmits the changed vector back to A. A receives this modified distance vector from B and concludes that C is now 4 away. Both A and B conclude that the best path to C is through the other node and continue this process until they count to infinity.

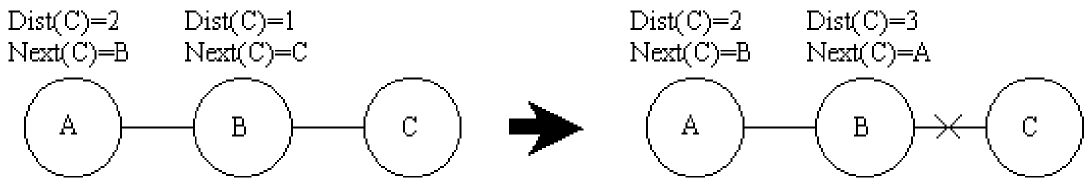


Figure 3: Example of Routing Loop in Distance Vector Routing

Partial remedies for these routing loops have been developed such as *poison-reverse* and *split-horizon* [Per00]. Poison-reverse means reporting a value of infinity to explicitly report that you can't reach a node rather than simply not mentioning the node. It is usually used together with split horizon. The rule in split horizon is that if A forwards traffic to destination C through B, then A reports to B that A's distance to C is infinity. Because A is routing traffic to C through B, A's real distance to B cannot possibly matter to B. B's distance to C cannot depend on A's distance to C

However, split horizon does not work in some cases. Consider the topology in Figure 4.

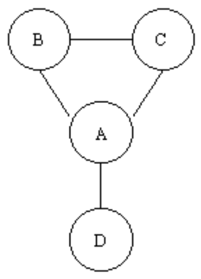


Figure 4: Count-to-infinity with split horizon

When the Link to D breaks, A concludes that D is unreachable because both B and C have reported to A that D is unreachable because of the split horizon rule. A reports D's unreachability to B and C. However, when B receives A's report that D is unreachable, B concludes that the best path to D is now through C. B concludes that 1) it is now 3 from D, 2)reports D as being unreachable to C because of split horizon, and 3) reports D as being reachable to A at cost 3. A now thinks that D is reachable through B at a cost of 4. The counting to infinity problem still exists.

2.2.2 Link State

Another class of algorithms that is also widely used in many existing routing protocols, such as OSPF [Moy93], is link-state routing. The main difference between link-state and distance vector is that in link-state, paths are computed based on the global network topology as opposed to the abstracted network view reported by neighboring nodes.

In link-state routing, each node maintains a complete view of the network topology with a cost for each link. To keep these costs consistent, each node (periodically and when it detects a link change) broadcasts the link costs of its outgoing links to all other nodes through special Link State Packets(LSP). These LSPs are flooded to all the other nodes in the network. As each node receives this information, it updates its view of the network and applies a shortest path algorithm(such as Djikstra's[Sed83] shortest path) to choose the next-hop for each destination.

Figure 5 shows an example of link state routing. Each node broadcasts to every other node its immediate neighbors and an associated cost(to keep things simple, in this example, the cost metric is just hops so all the costs are the same). Node 1 will broadcast its neighbors {D,2,3}, node 2 will broadcast its neighbors {S,1,4}, and so on.

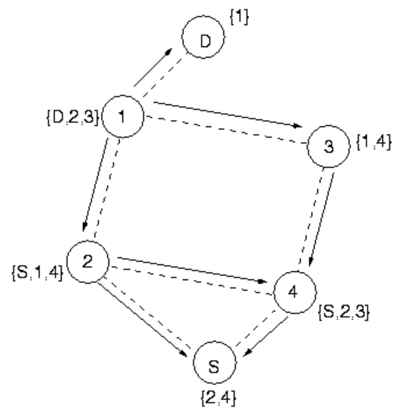


Figure 5: Example of Link State Routing

Inconsistent topology views can arise due to the delay in delivering an updated LSP across the entire network. Such inconsistent network topology views can lead to formation of routing loops. However, these loops are short-lived, because they disappear in the time it takes a message to traverse the diameter of the network [Jaf86].

2.2.3 Source Routing

In source routing, a node builds up a route by flooding a query to all nodes in the network for a given destination. The query packet stores the information of the intermediate nodes in a path field. On detecting the destination or any other node who has already learned the path to the destination, answers the query by sending a "source routed" response packet back to the sender. Since multiple responses may be produced, multiple paths may be computed and maintained. After the paths are computed, any link failure will trigger another query/response so the routing can always be kept up to date.

An example of Source routing is shown is Figure 6. This example focuses on node S finding a route to node D. S floods the network with a query to destination D. As each node receives the query, it stores the information of the intermediate nodes. Once D receives the query, it sends a "source routed" reply(reply(D)) back to S.

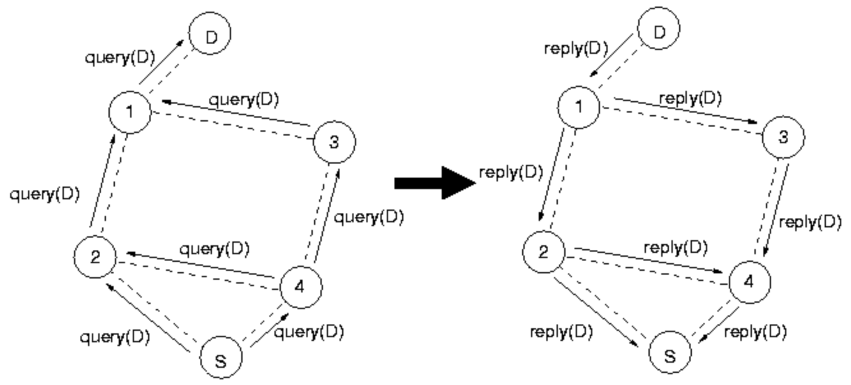


Figure 6: Example of On-Demand Source Routing

The advantage of this approach is that it minimizes overhead routing traffic as only routes that are needed are maintained. The disadvantage is that each packet requires an overhead containing the source route of the packet. This overhead grows when the packet has to go through more hops to reach the destination. In addition, every new destination from a source receives a latency hit as the route is discovered or needed. This does not scale well with mobility when topology changes and more route requests are generated as links break.

2.3 Summary

Conventional routing protocols for wired networks are ill-suited for ad-hoc wireless networks. Table-driven routing such as distance vector and link state have high overhead traffic caused by periodic exchange of control messages. On-Demand Routing suffers high initial latency hits and adds size to each packet to contain the source route. It does not scale well with traffic source/destination pair density and mobility. Clearly, new mechanisms must be introduced for effective ad hoc wireless routing.

3 Ad Hoc Routing Protocols

Since the advent of the Defense Advanced Research Projects Agency(DARPA) packet radio networks in the early 1970s [JT87], numerous protocols have been developed for ad hoc mobile networks. These routing protocols must deal with the typical limitations of these networks, which include low bandwidth and high error rates.

The following is a list of desirable properties for an ad hoc wireless routing protocol:

- **Distributed operation:** The protocol should be distributed, meaning that it should not be dependent on a centralized controlling node. This makes the system more robust to failure and growth.
- **Fast convergence:** Routes should be quickly determined in the presence of network changes. This means that when topology changes occur, the protocol should be able to quickly determine optimal new routes.
- **Loop free:** To have good overall performance, the routing protocol should supply routes that are loop-free. This avoids wasting bandwidth and CPU resources.
- **Optimal routes:** It is important for the protocol to find routes with the least number of hops. This reduces bandwidth and CPU consumption. In addition, it leads to lower overall routing delay.
- **Low overhead control traffic:** Bandwidth in a wireless network is a limited resource. The protocol should minimize the amount of overhead control messages for routing.

There are many research groups both in industry and in academia that are attempting to provide solutions to routing in ad hoc wireless networks. Much of the research is brought together by the Internet Engineering Task Force (IETF), which is a large open international community of designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. IETF has a working group named MANET (Mobile Ad-hoc Networks) that is working in the field of ad-hoc networks [Man00]. MANET and independent research groups have produced many different ad hoc routing protocols. Among them, the following proposed protocols will be analyzed in the next sections:

- Dynamic Destination-Sequenced Distance Vector (DSDV)
- Wireless Routing Protocol (WRP)
- Clusterhead Gateway Switch Routing (CGSR)
- Zone-based Hierarchical Link State (ZHLS)
- Ad Hoc On Demand Distance Vector (AODV)
- Temporally-Ordered Routing Algorithm(TORA)
- Dynamic Source Routing (DSR)
- Associativity-Based Routing (ABR)
- Signal Stability Routing (SSR)

These protocols have been selected to be analyzed because they constitute a good representation of current proposed Table-Drive and On-Demand techniques as applied to mobile ad hoc networks. DSDV, WRP, CGSR, and ZHLS are table driven routing protocols and AODV, TORA, DSR, ABR, and SSR are on-demand routing protocols.

3.1 Destination Sequenced Distance Vector – DSDV

3.1.1 Description

DSDV[Per94] is a hop-by-hop distance vector routing protocol where each node has a routing table that stores next-hop and number of hops for all reachable destinations. Like distance-vector, DSDV requires that each node periodically broadcast routing updates. The advantage with DSDV over traditional distance vector protocols is that DSDV guarantees loop-free routes.

To guarantee loop-free routes, DSDV uses a sequence number to tag each route. The sequence number shows the freshness of a route and routes with higher sequence numbers are favorable. A route R is considered more favorable than S if R has a greater sequence number, or, if the routes have the same sequence number, but R has a lower hop count. The sequence number is increased when node A detects that a route destination D has broken. So the next time node A advertises its routes, it will advertise the route to D with an infinite hop-count and a sequence number that is larger than before.

DSDV basically is distance vector with small adjustments to make it better suited for ad-hoc networks. These adjustments consist of triggered updates that will take care of topology changes in the time between broadcasts. To reduce the amount of information in these packets, there are two types of update messages defined: full and incremental. The full broadcast carries all available routing information and the incremental broadcast only carries the information that has changed since the last broadcast.

3.1.2 Properties

Because DSDV is dependent on periodic broadcasts, it needs some time to converge before a route can be used. This convergence time can probably be considered negligible in a static wired network, where topology changes are infrequent. However, in an ad-hoc network, the topology is expected to be very dynamic, thus causing a slow convergence of routes as packets are dropped and nodes move about. Periodic and triggered broadcasts also add a large amount of overhead into the network, especially when there is high mobility in the network.

3.2 The Wireless Routing Protocol- WRP

3.2.1 Description

The Wireless Routing Protocol (WRP) [MG96] is a table-based distance-vector routing protocol. Each node in the network maintains a Distance table, a Routing table, a Link-Cost table and a Message Retransmission list.

The Distance table of a node x contains the distance of each destination node y via each neighbor z of x. It also contains the downstream neighbor of z through which this path is realized. The Routing table of node x contains the distance of each destination node y from node x, the predecessor and the successor of node x on this path. It also contains a tag to identify if the entry is a simple path, a loop or invalid. Storing predecessor and successor in the table is beneficial in detecting loops and avoiding counting-to-infinity problems. The Link-Cost table contains cost of link to each neighbor of the node and the number of timeouts since an error-free message was received from that neighbor. The Message Retransmission list (MRL) contains information to let a node know which of its neighbor has not acknowledged its update message and to retransmit update message to that neighbor.

Node exchange routing tables with their neighbors using update messages periodically as well as on link changes. The nodes present on the response list of update message (formed using MRL) are required to acknowledge the receipt of update message.

If there is no change in routing table since last update, the node is required to send an idle Hello message to ensure connectivity. On receiving an update message, the node modifies its distance table and looks for better paths using new information. Any new path so found is relayed back to the original nodes so that they can update their tables. The node also updates its routing table if the new path is better than the existing path. On receiving an ACK, the mode updates its MRL. A unique feature of this algorithm is that it checks the consistency of all its neighbors every time it detects a change in link of any of its neighbors.

3.2.2 Properties

Part of the novelty of WRP stems from the way in which it achieves loop freedom. In WRP, routing nodes communicate the distance and second-to-last hop information for each destination in the wireless networks. It avoids the "count-to-infinity" problem by forcing each node to perform consistency checks of predecessor information reported by all its neighbors. This eliminates looping situations and provides faster route convergence when a link failure event occurs. However, to achieve this loop freedom, WRP suffers from high overhead control traffic caused by the periodic and triggered exchange of routing tables and the reliance on ACK and HELLO responses (caused by spurious retransmission of route tables if ACKs or HELLOs are lost).

3.3 Clusterhead Gateway Switching Routing- CGSR

3.3.1 Description

Clusterhead Gateway Switch Routing (CGSR) [Chi97] uses as basis the DSDV Routing algorithm described in the previous section. The protocol differs in the type of addressing and network organization scheme employed. Instead of a “flat” network, CGSR is a clustered multihop mobile wireless network. It routes traffic from source to destination using a hierarchical cluster-head-to-gateway routing approach. Mobile nodes are aggregated into clusters and a cluster-head is elected. All nodes that are in the communication range of the cluster-head belong to its cluster. A gateway node is a node that is in the communication range of two or more cluster-heads.

A packet sent by a node is first routed to its cluster head, and then the packet is routed from the cluster head to a gateway to another cluster head, and so on until the cluster head of the destination node is reached. The packet is then transmitted to the destination.

Figure 7 illustrates an example of this routing scheme. Using this method, each node must keep a “cluster member table” where it stores the destination cluster head for each mobile node in the network. These cluster member tables are broadcast by each node periodically using the DSDV algorithm. Nodes update their cluster member tables on reception of such a table from a neighbor.

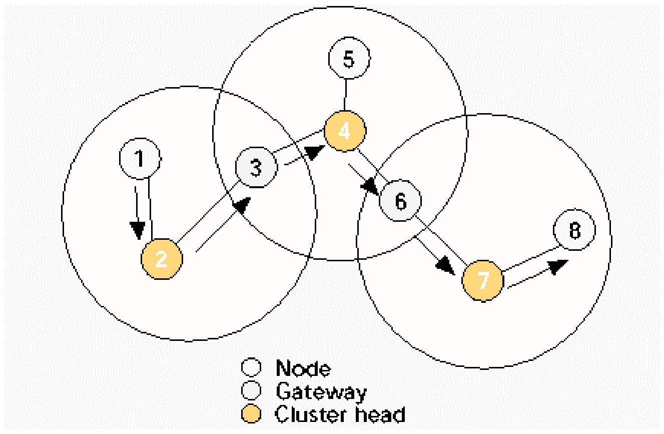


Figure 7: CGSR: Routing from node 1 to node 8.

In addition to the cluster member table, each node must also maintain a routing table which is used to determine the next hop in order to reach the destination. On receiving a packet, a node will consult its cluster member table and routing table to determine the nearest cluster head along the route to the destination. Next, the node will check its routing table to determine the next hop used to reach the selected cluster head. It then transmits the packet to this node.

3.3.2 Properties

CGSR achieves a framework among clusters for code separation, channel access, routing, and bandwidth by having a cluster head controlling a group of ad hoc nodes. This is a good approach when dealing with large ad-hoc networks. It is very scalable because it uses the clustering approach that limits the number of messages that need to be sent. However, the cluster design is vulnerable to point failures. If a cluster head goes down, then routing in the entire cluster is disturbed. Frequent cluster head changes can adversely affect routing protocol performance since nodes are busy in cluster head selection rather than packet relaying. In addition, routes between nodes in different clusters do not result in shortest hop paths.

3.4 Zone-based Hierarchical Link State- ZHLS

3.4.1 Description

In Zone-based Hierarchical Link State [NL99], the network is divided into non-overlapping zones. ZHLS defines two levels of topologies: 1) node level and 2) zone level. A node level topology describes how nodes of a zone are connected to each other physically. A virtual link between two zones exists if at least one node of a zone is physically connected to some node of the other zone. Zone level topology tells how zones are connected together. Unlike other hierarchical protocols, there are no zone heads. The zone level topological information is distributed to all nodes.

There are two types of Link State Packets (LSP) as well: node LSP and zone LSP. A node LSP of a node contains its neighbor node information and is propagated within the zone whereas a zone LSP contains the zone information and is propagated globally. Each node only knows the node connectivity within its zone and the zone connectivity of the whole network. So given the zone id and the node id of a destination, the packet is routed based on the zone id till it reaches the correct zone. Then in that zone, it is routed based on node id. A <zone id, node id> of the destination is sufficient for routing so it is adaptable to changing topologies.

3.4.2 Properties

ZHLS can be adjusted of its operation to the current network operational conditions (ie. change the routing zone radius). However this is not done dynamically, but instead the zone radius is set by the administrator of the network. The performance of this protocol depends greatly on this parameter.

ZHLS also limits the propagation of information about topological changes to the zone of the change(as opposed to flooding the entire network). This causes a reduction of overhead control traffic, however, at an expense of creating unoptimal routes (routes between zones are not necessarily minimum cost paths).

In the hierarchical approach, ZHLS mitigates traffic bottleneck and avoids single point failures by avoiding cluster heads. However, because of this, a node has to keep track of its physical location continuously in order to determine its affiliate zone. This requires some a complicated geo-location algorithm and device for each node.

3.5 Ad Hoc On Demand Distance Vector- AODV

3.5.1 Description

Ad hoc On-demand Distance Vector Routing (AODV) [PR98, PR99] is an improvement on the DSDV algorithm. AODV minimizes the number of broadcasts by creating routes on-demand as opposed to DSDV that maintains the list of all the routes.

To find a path to the destination, the source broadcasts a route request (RREQ) packet. The neighbors in turn broadcast the packet to their neighbors until it reaches an intermediate node that has a recent route information about the destination or until it reaches the destination. A node discards a route request packet that it has already seen. The route request packet uses sequence numbers to ensure that the routes are loop free and that the intermediate node replies to route requests are the most recent.

When a node forwards a route request packet to its neighbors, it also records in its tables the node from which the first copy of the request came. This information is used to construct the reverse path for the route reply packet. AODV uses only symmetric links because the route reply packet follows the reverse path of route request packet. As the route reply packet traverses back to the source, the nodes along the path enter the forward route into their tables.

If the source moves then it can reinitiate route discovery to the destination. If one of the intermediate nodes move then the moved nodes neighbor realizes the link failure and sends a link failure notification to its upstream neighbors and so on until it reaches the source upon which the source can reinitiate route discovery if needed.

The protocol also uses HELLO messages that are broadcast periodically to the immediate neighbors. These HELLO messages are local advertisements for the continued presence of the node to its neighbors. If HELLO messages stop coming from a particular node, the neighbor can assume that the node has moved away and notify the affected set of nodes by sending them a link failure notification.

3.5.2 Properties

The advantage with AODV compared to classical routing protocols like distance vector and link-state is that AODV has greatly reduced the number of routing messages in the network. AODV achieves this by using a reactive approach.

AODV only supports one route for each destination. This causes a node to reinitiate a route request query when it's only route breaks. This does not scale well as the number of route requests increase as mobility increases(topology changes in the network and links break).

AODV also does not support unidirectional links. When a node receives a RREQ, it will setup a reverse route to the source by using the node that forwarded the RREQ as the next hop. This means that the route reply is unicasted back the same way the route request used.

3.6 Temporally-Ordered Routing Algorithm- TORA

3.6.1 Description

Temporally Ordered Routing Algorithm (TORA) is a distributed source-initiated on-demand routing protocol [PC97]. The basic underlying algorithm is one in a family referred to as link reversal algorithms. TORA is designed to minimize reaction to topological changes. A key concept in this design is that control messages are typically localized to a very small set of nodes. It guarantees that all routes are loop-free (although temporary loops may form), and typically provides multiple routes for any source/destination pair.

TORA can be separated into three basic functions: 1) creating routes, 2) marinating routes, and 3) erasing routes. The creation of routes basically assigns directions to links in an unidirected network or portion of the network, building a directed acyclic graph (DAG) rooted at the destination.

TORA associates a height with each node in the network. All messages in the network flow downstream, from a node with higher height to a node with lower height. Routes are discovered using Query (QRY) and Update (UPD) packets. When a node with no downstream links needs a route to a destination, it will broadcast a QRY packet. This QRY packet will propagate through the network until it reaches a node that has a route or the destination itself. Such a node will then broadcast a UPD packet that contains the node height. Every node receiving this UPD packet will set its own height to a larger height than specified in the UPD message. The node will then broadcast its own UPD packet. This will result in a number of directed links from the originator of the QRY packet to the destination. This process can result in multiple routes.

Maintaining routes refers to reacting to topological changes in the network in a manner such that routes to destination are re-established within a finite time, meaning that its directed portions return to a destination-oriented graph within a finite time. Upon detection of a network partition, all links in the portion of the network that has become partitioned from the destination are marked as undirected to erase invalid routes. The erasure of routes is done using clear (CLR) messages.

3.6.2 Properties

The protocols underlying link reversal algorithm will react to link changes through a simple localized single pass of the distributed algorithm. However, there is potential for oscillations to occur, especially when multiple sets of coordinating nodes are concurrently detecting partitions, erasing routes, and building new routes based on each other. Because TORA uses internodal coordination, its instability problem is similar to the *count-to-infinity* problem in distance vector routing protocols, except that such oscillations are temporary and route convergence will eventually occur.

There are situations where multiple routes are possible from the source to the destination, but only one route will be discovered. This is caused because the graph is rooted at the destination(which has the lowest height) but the source originating the QRY does not necessarily have the highest height. The reason for this is that the height is initially based on the distance in number of hops from the destination.

3.7 Dynamic Source Routing- DSR

3.7.1 Properties

Dynamic Source Routing (DSR) [JM98, JM99] is a source-routed on-demand routing protocol. Source routing means that each packet in its header carries the complete ordered list of nodes through which the packet must pass. DSR uses no periodic routing messages (ie. no router advertisements), thereby reducing network bandwidth overhead and avoiding large routing updates throughout the ad-hoc network. Instead, DSR maintains a route cache, containing the source routes that it is aware of. It updates entries in the routes cache when it learns about new routes. The two basic modes of operation in DSR are 1) route discovery and 2) route maintenance.

When the source node wants to send a packet to a destination, it looks up its route cache to determine if it already contains a route to the destination. If it finds that an unexpired route to the destination exists, then it uses this route to send the packet. But if the node does not have such a route, then it initiates the route discovery process by broadcasting a route request packet. The route request packet contains the address of the source and the destination and a unique identification number. Each intermediate node checks whether it knows of a route to the destination. If it does not, it appends its address to the route record of the packet and forwards the packet to its neighbors. To limit the number of route requests propagated, a node processes the route request packet only if it has not already seen the packet and it's address is not present in the route record of the packet.

A route reply is generated when either the destination or an intermediate node with current information about the destination receives the route request packet. A route request packet reaching a such node contains the sequence of hops taken from the source to this node in its route record.

As the route request packet propagates through the network, the route record is formed. If the route reply is generated by the destination then it places the route record from route request packet into the route reply packet. On the other hand, if the node generating the route reply is an intermediate node then it appends its cached route for the destination to the route record of route request packet and puts that into the route reply packet. To send the route reply packet, the responding node must have a route to the source. If it has a route to the source in its route cache, it can use that route. The reverse of route record can be used if symmetric links are supported. In case symmetric links are not supported, the node can initiate route discovery to source and piggyback the route reply on this new route request.

DSRP uses two types of packets for route maintenance: Route Error packet and Acknowledgements. When a node encounters a fatal transmission problem at its data link layer, it generates a Route Error packet. When a node receives a route error packet, it removes the hop in error from it's route cache. All routes that contain the hop in error are truncated at that point. Acknowledgment packets are used to verify the correct operation of the route links. This also includes passive acknowledgments in which a node hears the next hop forwarding the packet along the route.

3.7.2 Properties

DSR uses the key advantage of source routing. Nodes do not need to maintain a complete view of the network in order to route the packets they forward. There is also no need for periodic routing advertisement messages, which will lead to reduced routing overhead.

This protocol has the advantage of learning routes by scanning for information in packets that are received. A route from A to C through B means that A learns the route to C, but also that it will learn the route to B. The source route will also mean that B learns the route to A and C and that C learns the route to A and B. This form of active learning is very good and reduces overhead in the network.

However, each packet carries an overhead containing the source route of the packet. This overhead grows when the packet has to go through more hops to reach the destination. This does not scale well to large networks when packets have to traverse through many hops to reach a destination. In addition, nodes only hold one route to their destination. When the route becomes invalid (due to topology change), a new route must be discovered. This does not scale well to mobility as overhead (due to route requests) and latency (due to finding new routes) increase as mobility increases.

3.8 Associativity-Based Routing- ABR

3.8.1 Description

The Associativity Based Routing (ABR) protocol is a new approach for routing proposed in [Toh96,Toh99]. ABR defines a new metric for routing known as the degree of association stability. It is free from loops, deadlock, and packet duplicates. In ABR, a route is selected based on associativity states of nodes. The routes thus selected are liked to be long-lived. All node generate periodic beacons to signify its existence. When a neighbor node receives a beacon, it updates its associativity tables. For every beacon received, a node increments its associativity tick with respect to the node from which it received the beacon. Association stability means connection stability of one node with respect to another node over time and space. A high value of associativity tick with respect to a node indicates a low state of node mobility, while a low value of associativity tick may indicate a high state of node mobility. Associativity ticks are reset when the neighbors of a node or the node itself move out of proximity. The fundamental objective of ABR is to find longer-lived routes for ad hoc mobile networks. The three phases of ABR are Route discovery, Route reconstruction (RRC) and Route deletion.

The route discovery phase is a broadcast query and await-reply (BQ-REPLY) cycle. The source node broadcasts a BQ message in search of nodes that have a route to the destination. A node does not forward a BQ request more than once. On receiving a BQ message, an intermediate node appends its address and its associativity ticks to the query packet. The next succeeding node erases its upstream node neighbors' associativity tick entries and retains only the entry concerned with itself and its upstream node. Each packet arriving at the destination will contain the associativity ticks of the nodes along the route from source to the destination. The destination can now select the best route by examining the associativity ticks along each of the paths. If multiple paths have the same overall degree of association stability, the route with the minimum number of hops is selected. Once a path has been chosen, the destination sends a REPLY packet back to the source along this path. The nodes on the path that the REPLY packet follows mark their routes as valid. All other routes remain inactive, thus avoiding the chance of duplicate packets arriving at the destination.

RRC phase consists of partial route discovery, invalid route erasure, valid route updates, and new route discovery, depending on which node(s) along the route move. Source node movement results in a new BQ-REPLY process because the routing protocol is source-initiated. The route notification (RN) message is used to erase the route entries associated with downstream nodes. When the destination moves, the destination's immediate upstream node erases its route. A localized query (LQ [H]) process, where H refers to the hop count from the upstream node to the destination, is initiated to determine if the node is still reachable. If the destination receives the LQ packet, it selects the best partial route and REPLYs; otherwise, the initiating node times out and backtracks to the next upstream node. An RN message is sent to the next upstream node to erase the invalid route and inform this node that it should invoke the LQ [H] process. If this process results in backtracking more than halfway to the source, the LQ process is discontinued and the source initiates a new BQ process.

When a discovered route is no longer needed, the source node initiates a route delete (RD) broadcast. All nodes along the route delete the route entry from their routing tables. The RD message is propagated by a full broadcast, as opposed to a directed broadcast, because the source node may not be aware of any route node changes that occurred during RRCs.

3.8.2 Properties

ABR is a compromise between broadcast and point-to-point routing, and uses the connection-oriented packet forwarding approach. Route selection is primarily based on the aggregate associativity ticks of nodes along the path. Although this may not produce shortest hop routes, the path tends to be longer-lived. Long lived routes result in fewer route reconstructions and therefore yield higher throughput. However, to maintain the associativity of a path, ABR relies on the fact that each node is beaconing periodically. This beaconing creates additional routing overhead.

3.9 Signal Stability-Based Routing- SSR

3.9.1 Description

Signal Stability-Based Routing protocol (SSR) presented in [Dub97] is an on-demand routing protocol that selects routes based on the signal strength between nodes and a node's location stability. This route selection criterion has the effect of choosing routes that have "stronger" connectivity. SSR comprises of two cooperative protocols: the Dynamic Routing Protocol (DRP) and the Static Routing Protocol (SRP).

The DRP maintains the Signal Stability Table (SST) and Routing Table (RT). The SST stores the signal strength of neighboring nodes obtained by periodic beacons from the link layer of each neighboring node. Signal strength is either recorded as a strong or weak channel. All transmissions are received by DRP and processed. After updating the appropriate table entries, the DRP passes the packet to the SRP.

The SRP passes the packet up the stack if it is the intended receiver. If not, it looks up the destination in the RT and forwards the packet. If there is no entry for the destination in the RT, it initiates a route-search process to find a route. Route-request packets are forwarded to the next hop only if they are received over strong channels and have not been previously processed (to avoid looping). The destination chooses the first arriving route-search packet to send back as it is highly likely that the packet arrived over the shortest and/or least congested path. The DRP reverses the selected route and sends a route-reply message back to the initiator of route-request. The DRP of the nodes along the path update their RTs accordingly.

Route-search packets arriving at the destination have chosen the path of strongest signal stability because the packets arriving over a weak channel are dropped at intermediate nodes. If the source times out before receiving a reply then it changes the

PREF field in the header to indicate that weak channels are acceptable, since these may be the only links over which the packet can be propagated.

When a link failure is detected within the network, the intermediate nodes send an error message to the source indicating which channel has failed. The source then sends an erase message to notify all nodes of the broken link and initiates a new route-search process to find a new path to the destination.

3.9.2 Properties

SSR selects routes based on the signal strength and location stability of nodes along the path. While the paths selected by this algorithm are not necessarily shortest in hop count, they do tend to be more stable and longer-lived. One of the drawbacks of SSR is that intermediate nodes cannot reply to route requests sent toward a destination. No attempt is made to use partial route recovery to allow intermediate nodes to attempt to rebuild the routes themselves. This may lead to longer route reconstruction times since link failures cannot be resolved locally.

3.10 Summary and Comparison

The routing protocols can be generally categorized into two groups: Table-Driven and On-Demand. DSDV, WRP, CGSR, and ZHLS utilize Table-Driven routing. AODV, TORA, DSR, ABR, and SSR utilized On-Demand routing.

DSDV routing is essentially a modification of the basic Bellman-Ford routing algorithm. DSDV provides one path to any given destination and selects the shortest path based on the number of hops to the destination. However, DSDV is inefficient because of the requirement of periodic update transmissions, regardless of the number of changes in the network topology.

In CGSR, DSDV is used as the underlying routing protocol. Routing in CGSR occurs over cluster heads and gateways. One advantage of CGSR is that several heuristic methods can be employed to improve the protocol's performance. These methods include priority token scheduling, gateway code scheduling, and path reservation[XXX]. However, CGSR is vulnerable to point failures and cluster head assignment is difficult to do.

ZHLS is a very interesting proposal that divides the network into several zones. This approach is probably a very good solution for large networks as it reduces overhead control traffic by limiting topology updates within each zone. However it produces unoptimal (routes that are not shortest hop) for nodes between zones. In addition, there is overhead in maintaining the status of the zone a node is in.

WRP protocol avoids the problem of creating temporary routing loops through the verification of predecessor information. This requires each node to maintain four routing tables, which can lead to substantial memory requirements, especially when number of nodes in the network is large. In addition, the use of HELLO packets whenever there are no recent packet transmissions from a given node consumes bandwidth.

Of the reactive on-demand protocols, AODV and DSR are similar in that they have a route discovery mode that uses request messages to find new routes. The difference is that DSR is based on source routing and will learn more routes than AODV. DSR also has the advantage that it supports unidirectional links. DSR has the major drawback that the source route must be carried in each packet. The can be quite costly, especially with network size becomes very large. TORA uses a link-reversal algorithm to minimize reaction to topological changes. However, it suffers slow route convergence due to oscillations.

ABR and SSR create routes based on route stability instead of shortest number of hops. The idea is that long-lived routes require fewer route reconstructions, therefore yielding higher overall throughput. ABR and SSR differ on how route stability is measured. ABR route selection is primarily based on the aggregate associativity ticks of nodes along a path, whereas SSR selects routes based on the signal strengths and location stability of nodes along the path. A drawback of these protocols is that because routes are selected based on an aggregate metric for route stability, when a link failure occurs along a path, the route discovery algorithm must be reinvoked from the source to find a new path to the destination. This may lead to longer route reconstruction times since link failures cannot be resolved locally. In addition, it remains to be seen whether creating routes that are longer-lived rather than shortest hop produces better performance.

These protocols offer different solutions for routing in the ad hoc mobile network, however, they also come with drawbacks. By studying current proposed routing protocols, a good understanding of tradeoffs in routing in ad hoc mobile networks is achieved. By weighing the advantages and disadvantages of certain routing protocol features, a new protocol will be presented in the next section that provides good routing performance, and at the same time, mitigates the drawbacks incurred to achieve such performance.

4 Fisheye Wireless Routing Protocol

4.1 Protocol Overview

In this chapter, a new routing scheme for ad-hoc wireless networks is presented. The goal is to provide an accurate routing solution while the control overhead is kept low. The proposed scheme is named "Fisheye Routing" due to the novel 'fisheye' updating mechanism. Similar to Link State Routing, Fisheye Routing generates accurate routing decisions by taking advantage of the global network information. However, this information is disseminated in a method to reduce overhead control traffic caused by traditional flooding. Instead, it exchanges information about closer nodes more frequently than it does about farther nodes. So, each node gets accurate information about neighbors and the detail and accuracy of information decreases as the distance from the node increases.

4.2 Table-Driven Design

Fisheye Routing determines routing decisions using a table-driven routing mechanism similar to link state. The table-driven ad hoc routing approach uses a connectionless approach of forwarding packets, with no regard to when and how frequently such routes are desired. It relies on an underlying routing table update mechanism that involves the constant propagation of routing information. A table-driven mechanism was selected over an on-demand mechanism based on the following properties:

- On-Demand routing protocols on the average create longer routes than table driven routing protocols [ICP99].
- On-Demand routing protocols are more sensitive to traffic load than Table-Driven in that routing overhead traffic and latency increase as data traffic source/destination pairs increase.
- On-Demand Routing incurs higher average packet delay than Table Driving routing which results from latency caused by route discovery from new destinations and less optimal routes.
- Table-Driven routing accuracy is less sensitive to topology changes. Since every node has a 'view' of the entire network, routes are less disrupted when there is link breakage (route reconstruction can be resolved locally).
- Table-Driven protocols are easier to debug and to account for routes since the entire network topology and route tables are stored at each node, whereas On-Demand routing only contain routes that are source initiated and these routes are difficult to track over time.

For these reasons, a table driven scheme for the ad hoc routing protocol was chosen. Link state was chosen over distance vector because of faster speed of convergence and shorter-lived routing loops [ZA91]. Link state topology information is disseminated in special link-state packets where each node receives a global view of the network rather than the view seen by each node's neighbor. Fisheye routing takes advantage of this feature by implementing a novel updating mechanism to reduce control overhead traffic. The algorithm for Fisheye routing is described in the next sections.

4.3 Algorithm

There are 3 main tasks in the routing protocol:

1) **Neighbor Discovery**: responsible for establishing and maintaining neighbor relationships.
2) **Information Dissemination**: responsible for disseminating Link State Packets(LSP), which contain neighbor link information, to other nodes in the network.
3) **Route Computation**: responsible for computing routes to each destination using the information of the LSPs.

Each node initially starts with an empty neighbor list and an empty topology table. After its local variables are initialized, it invokes the *Neighbor Discovery* mechanism to acquire neighbors and maintain current neighbor relationships. LSPs in the network are distributed using the *Information Dissemination* mechanism. Each node has a database consisting of the collection of LSPs originated by each node in the network. From this database, the node uses the *Route Computation* mechanism to yield a routing table for the protocol. This process is periodically repeated.

4.3.1 Neighbor discovery

This mechanism is responsible for establishing and maintaining neighbor relationships. Neighbors can meet each other simply by transmitting a special packet(a HELLO packet) over the broadcast medium. In the wireless network, HELLO packets are periodically broadcasted and nodes within the transmission range of the sending node will hear these special packets and record them as neighbors. Each node associates a TIMEOUT value in the node's database for each neighbor. When it does not hear a HELLO packet from a particular neighbor within the TIMEOUT period, it will remove that neighbor from the neighbor list. TIMEOUT values are reset when a HELLO message is heard.

HELLO Packets also contain the list of routers whose HELLO Packets have been seen recently. Nodes can use this information to detect the presence of uni-directional or bi-directional links by checking if it sees itself listed in the neighbor's HELLO Packets.

4.3.2 Information Dissemination

This mechanism is responsible for distributing LSPs to the nodes in the network. It's two main functions are to handle the LSP integrity and updating interval.

LSP Integrity

After the router generates a new LSP, the new LSP must be transmitted to all the other routers. A simple scheme is flooding, in which each packet received is transmitted to each neighbor except the one from which the packet was received. Because each router retains the most recently generated LSP from other nodes, the router can recognize when it is receiving a duplicate LSP and refrain from flooding the packet more than once.

The problem with this flooding is that a router cannot assume that the LSP most recently received is the one most recently generated by that node. Two LSPs could travel along different paths and might not be received in the order in which they were generated. A solution to this is to use a scheme involving a combination of a sequence number and an estimated age for each LSP.

A sequence number is a counter. Each router keeps track of the sequence number it used the last time it generated an LSP and uses the next sequence number when it needs to generate a new LSP. When a router receives a LSP, it compares the sequence number of the received LSP with the one stored in memory (for that originating node) and only accepts the LSP if it has a higher sequence number. The higher the sequence number, the more recently generated.

However, a sequence number alone is not sufficient. The sequence number approach has various problems:

1) The sequence number field is of finite size. A problem arises when a node creates a LSP to case the field to reach the maximum value. Making the sequence number field wrap around is not a good idea because it causes ambiguity on the relation of the sequence numbers.
2) Sequence number on an LSP becomes corrupt. If the sequence field is corrupted to a very large sequence number, it will prevent valid, newer LSPs (with smaller sequence numbers) to be accepted.
3) Sequence number is reset. When a router goes down or forgets the sequence number it was using, newer LSPs cannot be distinguished from older LSPs

To solve the preceding problems, an age field is added to each LSP. It starts at some value and is decremented by routers as it is held in memory. When an LSP's age reaches 0, the LSP can be considered too old and an LSP with a nonzero age is accepted as new, regardless of its sequence number.

Update Interval

The key difference between fisheye and traditional Link-state is the interval in which the routing information is disseminated. In Link State, the link state packets are generated and flooded into the network whenever a node detects topology changes. Fisheye uses a new approach to reduce the number of LSP messages.

In [KS71], Kleinroch and Stevens proposed the fisheye technique to reduce the size of information required to represent graphical data. The original idea of fisheye was to maintain high resolution information within a range of a certain point of interest and lower resolution further away from the point of interest. For routing, this fisheye approach can be interpreted as maintaining a highly accurate network information about the immediate neighborhood of a node and becomes progressively less detailed as it moves away from the node.

Figure 8 illustrates the application of fisheye in a mobile wireless network. The figure defines the scope of fisheye for the center node. The scope is defined in terms of the nodes that can be reached in a certain number of hops. The center node has most accurate information about all nodes in the first circle, and becomes less accurate with each outer circle. Even though a node does not have accurate information about distance nodes, the packets are routed correctly because the route information becomes more and more accurate as the packet moves closer to the destination.

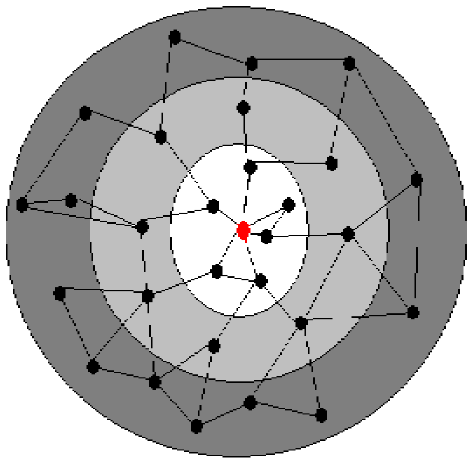


Figure 8: Application of fisheye in a network.

The reduction of routing messages is achieved by updating the network information for nearby nodes at a higher frequency and remote nodes at a lower frequency. As a result, considerable amount of LSPs are suppressed. When a node receives a LSP, it calculates a time to wait before sending out the LSP from the following equation:

UpdateInterval = ConstantTime * hopcount^alpha

ConstantTime is the user defined default refresh period to send out LSPs(in the first scope), *hopcount* is the number of hops the LSP has traversed, alpha is a parameter that determines how much effect each scope has on the *UpdateInterval*. Values for alpha are zero(same as no fisheye) and greater than or equal to one(fisheye). A maximum value of *UpdateInterval* is established to prevent an effective complete suppression of LSP messages(when calculated *UpdateInterval* is too large).

When a router accepts a LSP from a faraway node, and has not yet sent out the LSP in memory, the next time it will send out the LSP will be the minimum of the time left to wait in memory and the new calculated *UpdateInterval* based on the new LSP:

UpdateInterval(new) = MIN(UpdateInterval(memory), UpdateInterval(LSP))

This is to prevent a router from waiting indefinitely to send out a LSP when a new LSP arrives before the one in memory is sent out for that node.

4.3.3 Route Computation

Once the router has a database of LSPs, it computes the routes based on the Djikstra's [Sed83] algorithm which computes all shortest paths from a single vertex. The link metric used for path cost is the hop count. The algorithm uses 3 databases:

1) Link State Database- Contains the LSPs the node received.

2) PATH- contains ID, path cost, forwarding direction tuples. Holds the best path found.
3) TENT- contains ID, path cost, forwarding direction tuples. Holds possible best paths.

The Djikstra algorithm is as follows:

1) Start with "self" as the root of a tree by putting (myID, 0, 0) in PATH.
2) For node N just place in PATH, examine N's LSP. For each of N's neighbors, add the total path cost at N to the cost path of each neighbor. If the new total path of the node is better than the value for that node in PATH or TENT, put into TENT.
3) If TENT is empty, terminate the algorithm. Otherwise, find the minimal cost in TENT, move into PATH, and go to Step 2.

One the algorithm completes, PATH now contains the shortest next-hop information for each destination. The protocol can now use the PATH database as a routing table to forward packets toward their destinations.

4.4 Implementation

The Fisheye routing protocol was implemented in the Composable Network Software(CNS) environment developed by the Digital Communication Networks Group at MITRE Corporation. CNS is a scalable design environment for network systems. Since most network systems are being built using a layered approach similar to the OSI layer network architecture, CNS uses the same approach. Modules can be built between the different simulation layers. This will allow rapid integration of models developed at different layers by different people. The protocol stack supports models for the channel, radio, MAC, network, transport, and application layers.

CNS is programmed in C++ to take advantage of the Object Orientated Programming paradigm. Each module developed in the CNS environment has a well-defined interface to pass data between modules. Modules in CNS can also be used to generate traffic, setup network topologies, introduce link/transmission characteristics, debugging, or any other function.

The composable network stack for the Fisheye routing protocol is shown in Figure 9.

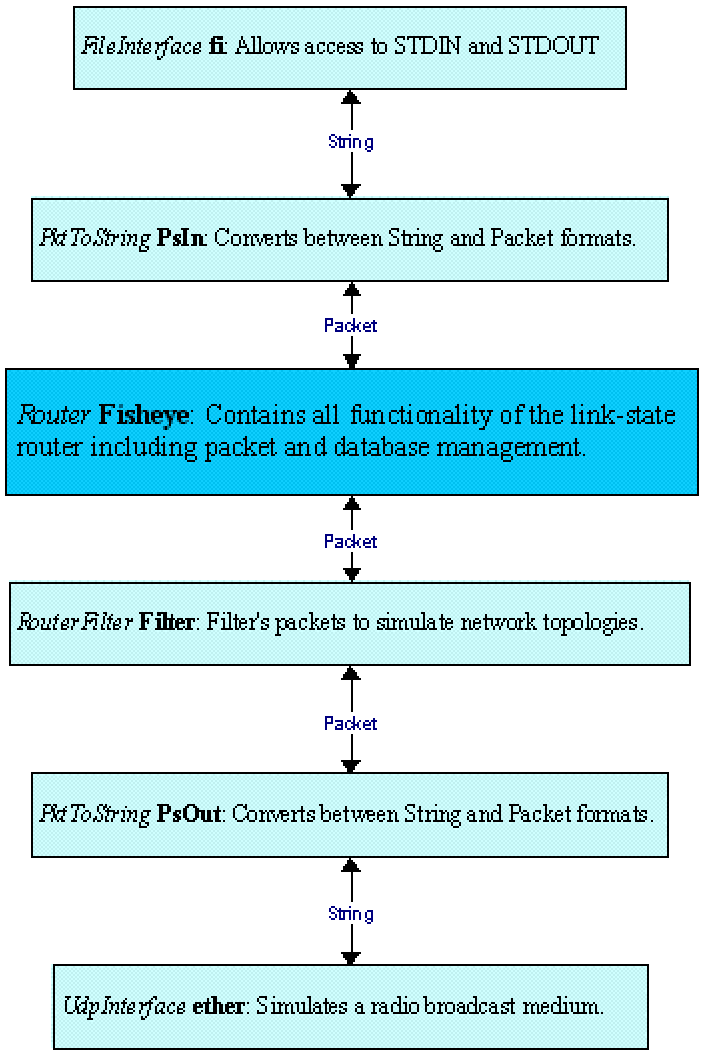
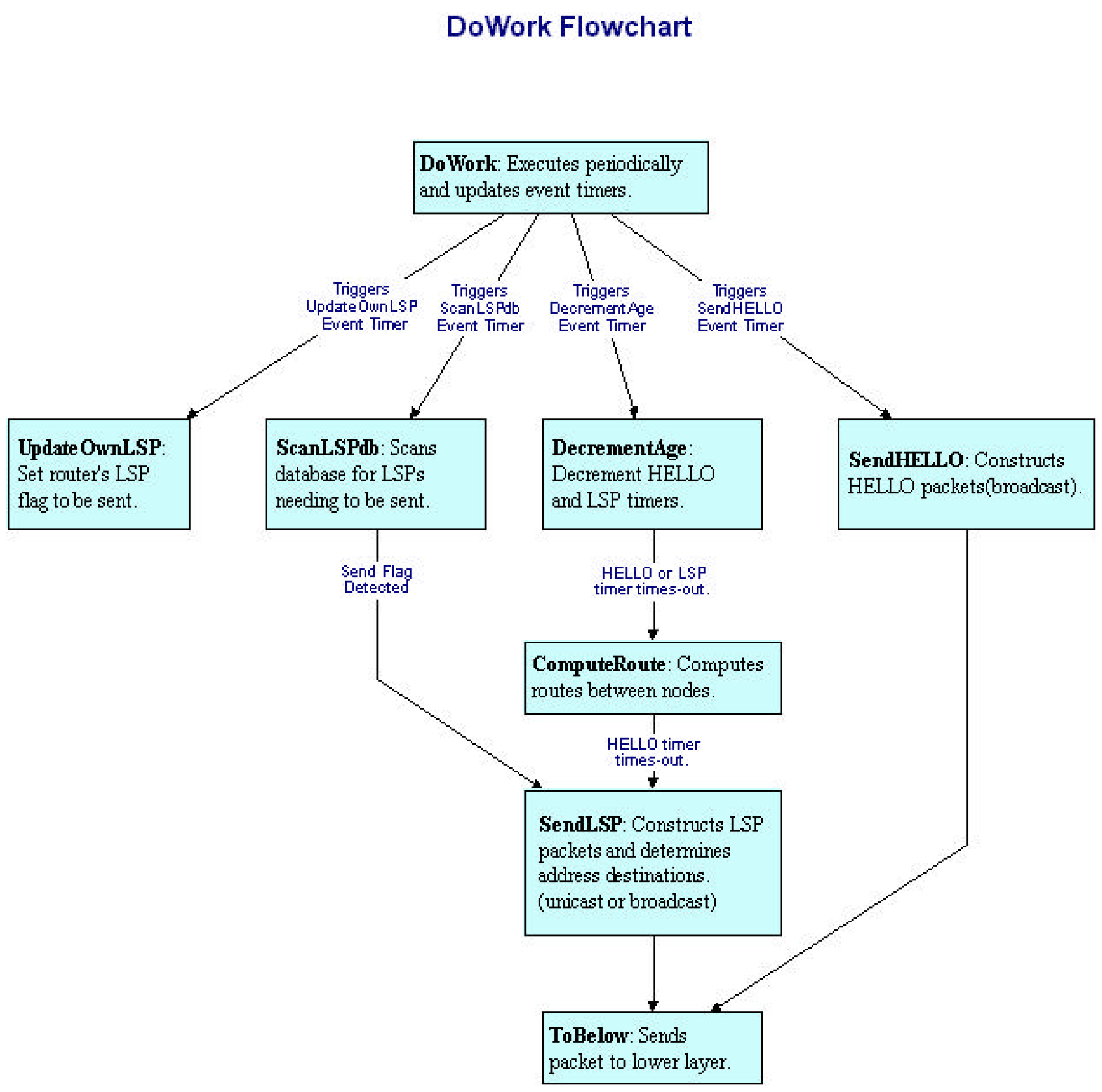

FileInterface **fi**: Allows access to STDIN and STDOUT
String
PktToString **PsIn**: Converts between String and Packet formats.
Packet
Router **Fisheye**: Contains all functionality of the link-state router including packet and database management.
Packet
RouterFilter **Filter**: Filter's packets to simulate network topologies.
Packet
PktToString **PsOut**: Converts between String and Packet formats.
String
UdpInterface **ether**: Simulates a radio broadcast medium.

Figure 9: Composable Network Stack for Fisheye Routing Protocol

The routing protocol is entirely implemented in the ***Router Fisheye*** module. The other modules offer a simulated network environment to test the functionality of the routing protocol. ***FileInterface fi*** allows user to insert packets into the Router module(such as test and data packets). ***RouterFilter Filter*** simulates a network topology. ***UdpInterface ether*** simulates a radio broadcast transmission medium. ***PktToString PsOut*** and ***PsIn*** allows interface between different modules.

In the Router Fisheye module, there are two top level main functions that are called to handle routing. They are ***DoWork()*** and ***DoConsume()***. ***DoWork()*** is called periodically based on a series of event timers. ***DoConsume()*** is invoked when the router receives a packet on it's interface. The flowchart for these functions will be shown in the next two sections.

4.4.1 DoWork() Flowchart


DoWork Flowchart
DoWork: Executes periodically and updates event timers.
Triggers UpdateOwnLSP Event Timer
Triggers ScanLSPdb Event Timer
Triggers DecrementAge Event Timer
Triggers SendHELLO Event Timer
UpdateOwnLSP: Set router's LSP flag to be sent.
ScanLSPdb: Scans database for LSPs needing to be sent.
DecrementAge: Decrement HELLO and LSP timers.
SendHELLO: Constructs HELLO packets(broadcast).
Send Flag Detected
HELLO or LSP timer times-out.
ComputeRoute: Computes routes between nodes.
HELLO timer times-out.
SendLSP: Constructs LSP packets and determines address destinations. (unicast or broadcast)
ToBelow: Sends packet to lower layer.

The DoWork() function is called periodically based on four event timers:

1) UpdateOwnLSP

2) ScanLSPdb

3) DecrementAge

4) SendHELLO.

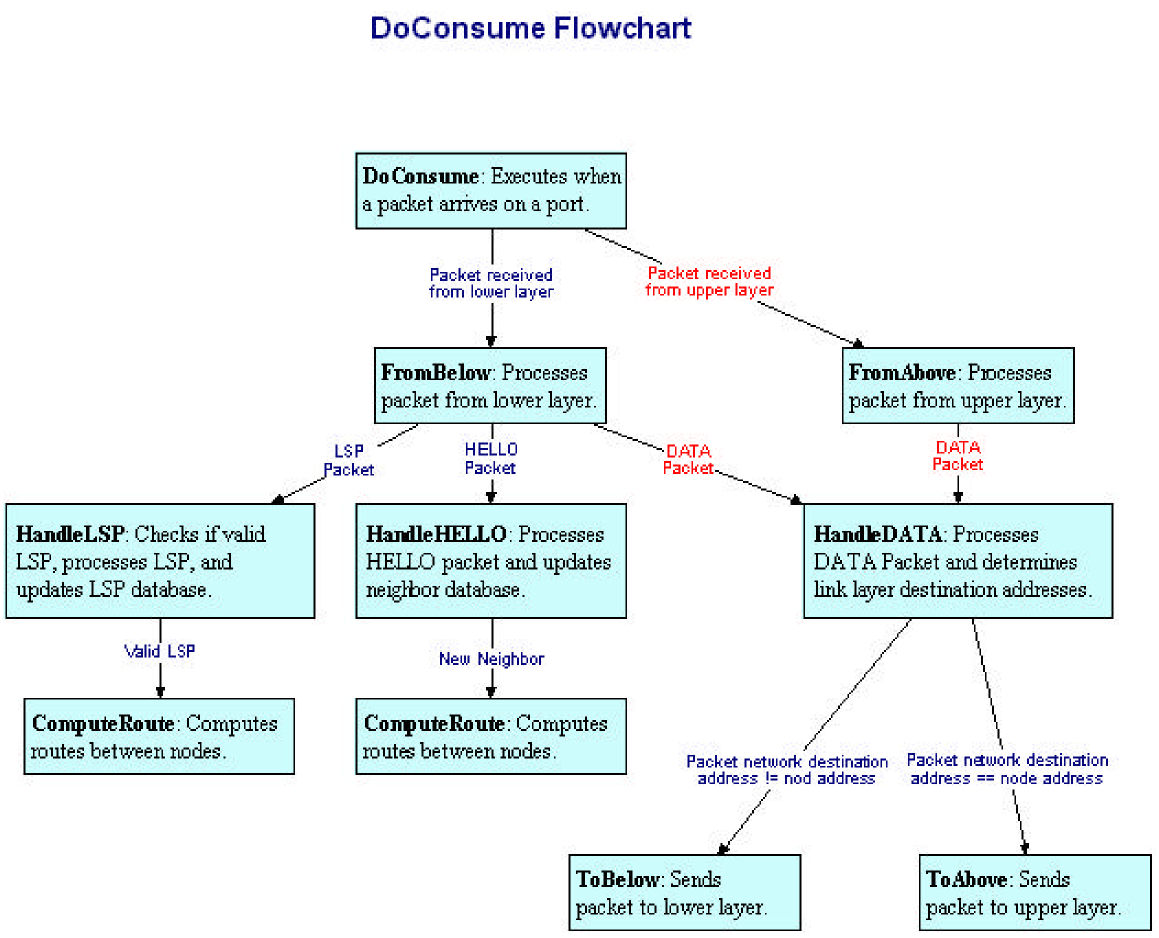
UpdateOwnLSP event timer controls when that node should send out its *own* LSP to neighbor nodes. This is necessary to propagate the current node's LSP to the other nodes in the network.

ScanLSPdb event timer scans the LSP database to check when a LSP received from *another* node should be sent out. This is necessary for the Fisheye update mechanism since LSPs received at different scopes will have different times to send. It checks the *UpdateInterval* (as described in section 4.3.2) values associated with each LSP and only sends the LSP out if the *UpdateInterval* for that LSP has been exceeded.

DecrementAge event timer decrements the HELLO and LSP timeout timers. HELLO timeout timers are needed to check if node x is still a valid neighbor of node y. Node y needs to periodically hear HELLO messages within a certain period of time from node x for node x to be considered a valid neighbor of node y. The LSP timeout timers are to decrement the AGE field of the LSP while it sits in memory to insure LSP integrity. When a neighbor becomes invalid, it will invoke *ComputeRoutes()* to compute a new routing table because it has detected a topology change.

SendHELLO event timer is used to periodically send HELLO messages to neighboring nodes. This is needed for other nodes to detect and maintain the presence of neighboring nodes.

4.4.2 DoCosume() Flowchart


DoConsume Flowchart
DoConsume: Executes when a packet arrives on a port.
Packet received from lower layer
Packet received from upper layer
FromBelow: Processes packet from lower layer.
FromAbove: Processes packet from upper layer.
LSP Packet
HELLO Packet
DATA Packet
DATA Packet
HandleLSP: Checks if valid LSP, processes LSP, and updates LSP database.
HandleHELLO: Processes HELLO packet and updates neighbor database.
HandleDATA: Processes DATA Packet and determines link layer destination addresses.
Valid LSP
New Neighbor
ComputeRoute: Computes routes between nodes.
ComputeRoute: Computes routes between nodes.
Packet network destination address != nod address
Packet network destination address == node address
ToBelow: Sends packet to lower layer.
ToAbove: Sends packet to upper layer.

DoConsume() is called whenever a packet is received at it's interface. It utilizes the CNS environment's network stack to determine which interface a packet was received on. Packets received from a the upper layer are handled by *FromAbove()*. These packets are usually data packets from a application or debug packets. Packets received the lower layer are handled by *FromBelow()*. These packets can both be routing packets and data packets.

All data packets are processed by *HandleData()* with determines where the packet should be forwarded to based on the protocol's routing table. If the packet is intended for the current node, it forwards the packet toward the upper (application) layer to be processed. If the packet is intended for another destination, it forwards the packet toward that node.

. Routing packets are processed by *HandleLSP()* or *HandleHELLO()* based upon the type of packet. *HandleLSP()* processes LSPs. It checks to see if the fields in the LSP are valid, determines if the node should accept the LSP, and records the relevant information of the LSP into the LSP database including calculating the *UpdateInterval* time. *HandleHELLO()* processes HELLO packets received from neighboring nodes and it updates the node's neighbor list in the database. If either a new LSP or HELLO packet with a new neighbor is accepted, then *ComputeRoutes()* is invoked which calculates a new routing table. This is needed to maintain an up-to-date routing table when it detects changes in the network topology.

4.5 Performance Analysis

The Fisheye routing protocol was simulated in a mobile environment to determine the connectivity among mobile hosts. The simulator for evaluating the protocol is the Global Mobile Simulation (GloMoSim) environment [ZBG98] from UCLA. GloMoSim is designed using the Parallel Simulation Environment for Complex Systems (PARSEC) [Bag98] to provide a discrete-even simulation environment for wireless network systems.

4.5.1 Simulation Model

The simulation models a network of 30 mobile nodes migrating within a 20m x 20m spaces with a transmission radius of 5 meters. Every node in the network moves in a Random-waypoint fashion. In Random-waypoint, each node calculates a random destination and moves towards it at a fixed rate. Once the destination has been reached, it selects another random location and repeats the process. The raw channel wireless capacity is 2Mbits/sec. A traffic generator was developed to simulate constant bit rate sources between two nodes. Simulation runs of 200,000,000,000 simulation ticks(equal to 200 seconds of simulated time) were performed multiple times and the results averaged.

4.5.2 Simulation Results

Different values of *alpha* were tested to compare their relative effects on (a) Control Overhead and (b) Successful Packet Deliveries. *Alpha* affects the interval of LSP updates at different scopes. As previously stated, the update interval when propagating LSPs at each node is calculated as: UpdateInterval = ConstantTime * hopcount^alpha. As *alpha* increases, the *UpdateInterval* increases (at each hop). *ConstantTime* in these simulations was set at 3 seconds.

The effect of latency was considered but did not yield good information. This is because the computation of average latency is hindered by packet drops. If packet drops are excluded from the computation, then average latency appears to decrease as packet loss increases. This is because the dropped packets are most likely the ones going through greater number of hops. If dropped packets are included in the computation, they must be assigned an arbitrarily large constant delay. Unfortunately, this arbitrary constant skews the average latency and prevents one from knowing the average latency of packets that were not dropped.

Figure 10 shows the control overhead incurred by different values of *alpha* as a function of mobility. As one would expect, control overhead goes down as the value of alpha increases. This occurs because nodes wait a longer time before transmitting LSPs it received from other nodes at each successive scope. This results in lower control overhead traffic. The reduction of overhead traffic at higher values of alpha are very significant.

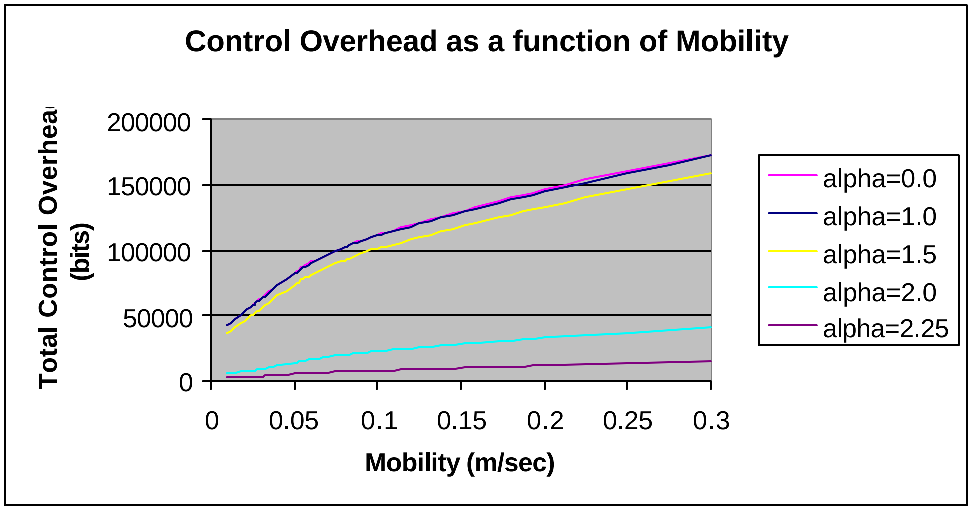


Figure 10: Overhead as a function of Mobility

Figure 11 shows the number of successful packet delivery over different *alphas* as a function of mobility. Overall, higher mobility causes a decrease in successful packet deliveries for all values of alpha. However, as this figure shows, there are more successfully delivered packets at lower values of alpha. This is because LSPs are refreshed more frequently and therefore route tables are more reflective of the actual network topology, thus producing greater number of valid routes.

In the figure, mobility has a greater affect on higher values of alpha. As mobility increases from 0 to 0.5 m/s, packets are dropped at a higher rate at higher values of alpha. This is a result of less accurate routing tables at each node because network topology changes are not propagated as frequently. At higher values of alpha, when refresh rates are less frequent, routing table accuracy is more sensitive to network topology changes.

One can also notice that the successful packet deliveries over different values of alpha seem to converge at high values of mobility. This is caused by a second order effect where the nodes are moving so fast, that only the minimal hop(1 hop) routes are present, regardless on how fast the routing tables are updated.

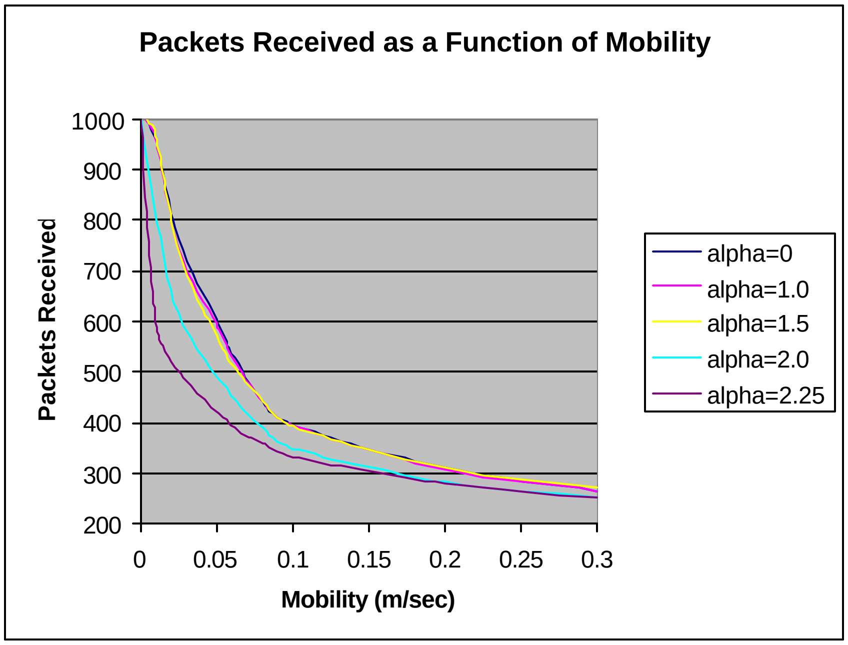


Figure 11: Successful Packet Received as a Function of Mobility

4.5.3 Simulation Summary

The simulations of the Fisheye protocol implementation has shown that fisheye does work in routing packets and reducing overhead traffic in a mobile environment. By increasing the update interval time of LSPs at different scopes, tremendous amounts of overhead control traffic can be suppressed. However, there is a trade-off. As the update interval time increases, the routing tables at each node become less accurate, causing a reduction of the number of successful packet deliveries. One must balance mobility, routing accuracy, and overhead traffic to achieve an optimal value for the update interval(*alpha*).

4.6 Comparison with other Ad Hoc Routing Protocols

This section provides comparisons of previously described routing algorithms (section 3) with Fisheye routing. Table 1 summarizes and compares properties of the ad hoc routing protocols.

	Fisheye	DSDV	WRP	CGSR	ZHLS
Loop-free	Yes	Yes	Yes, but not instantaneous	Yes	Yes
Distributed	Yes	Yes	Yes	Yes	Yes
Routing Philosophy	Table-Driven	Table-Driven	Table-Driven	Table-Driven	Table-Driven
Periodic Broadcasts	Varying over scopes	Periodic	Periodic and triggered	Periodic	Different by zone level
Topology Philosophy	Flat	Flat	Flat	Hierarchical	Hierarchical
Critical Nodes	No	No	No	Yes	Yes
Routing Metric	Shortest path	Shortest path	Shortest path	Shortest path	Shortest path

	AODV	TORA	DSR	ABR	SSR
Loop-free	Yes	No, short lived loops	Yes	Yes	Yes
Distributed	Yes	Yes	Yes	Yes	Yes
Routing Philosophy	On-Demand	On-Demand	On-Demand	On-Demand	On-Demand
Periodic Broadcasts	Periodic and when needed	Periodic	No	Periodic on associativity	No
Topology Philosophy	Flat	Flat	Flat	Flat	Flat
Critical Nodes	No	No	No	No	No
Routing Metric	Freshest and shortest path	Shortest path	Shortest path	Associativity/ route stability	Signal strength stability

Table 1: Comparison between ad-hoc protocols.

Among the table driven protocols, Fisheye, DSDV, and WRP use ‘flat’ network addressing. Because Fisheye uses link-state, it has the advantage over DSDV in terms of faster route convergence. However, Link-state requires more computation complexity than Distance-vector, in that Link-state requires more computation steps for a node to perform routing computations from the update messages [Per00]. WRP uses consistency checks of predecessor information to avoid routing loops. This requires that it maintain several routing tables which lead to much higher memory requirements than Fisheye. Fisheye also has the advantage over DSDV and WRP in lower overhead control traffic resulting from the periodic broadcast of routing messages. However, the suppression of routing messages at successive scopes used by Fisheye may degrade routing accuracy.

CGSR and ZHLS differ among the other table-driven protocols, in that they use a hierarchical addressing scheme such that nodes are grouped into clusters (or zones). Nodes can be localized for channel access, routing, bandwidth allocation separation among clusters. This has the advantage that it can scale well to high network sizes. However, this relies on critical nodes to control routing between regions and to maintain node association. This is a difficult problem to solve, but may be necessary for large networks. While flat addressing schemes may be less complicated and easier to use, there are doubts as to its scalability [Dal97]. Fisheye partially circumvents the problem of scalability of flat addressing schemes by using different updating scopes. This has the effect of localizing routing messages to nodes that are close to each other.

Among the on-demand routing schemes, AODV, DSR, and TORA find shortest-hop routes only when routes to new destinations are desired. ABR and SSR are on-demand routing schemes that find routes that are longer-lived (which are not necessarily shortest hop) based on some metric. It is uncertain weather shortest hop routes or longer-lived routes are better. Since longer-lived routes do not necessarily result in smallest number of hops, it may incur higher latency. However, longer-lived routes require fewer route reconstruction and therefore may yield higher throughput. In addition, network conditions will affect the performance of each method. Long-lived routes will be favored in presence of high mobility when there are higher number of link changes, and shortest-hop routes will be favored when there is low mobility. Thus, it remains to be seen whether longer-lived routes are more optimal than shortest-hop routes.

On-demand routing schemes have an advantage over the table-driven fisheye scheme in that they do not rely on an underlying routing table update mechanism that involves the constant propagation of routing information. Routing information in Fisheye is constantly propagated, and a route to every other node in the network is available. This feature incurs substantial signaling traffic. However, in on-demand routing, routing traffic grows with increasing mobility of active routes and with increasing source/destination traffic pairs. Thus, in a large dense network with high number of traffic pairs, on-demand routing may incur higher overhead traffic than the fisheye scheme. Since network conditions are not known a priori, it is favorable to have a mechanism that is insensitive to traffic conditions.

4.7 Summary

Each of the proposed schemes have certain features to deal with certain problems of routing in ad hoc networks. Because inherently, network conditions (such as traffic density, network size, and mobility) in an ad hoc network are not known, it is preferable to design a protocol that is not sensitive to network conditions. This is why Fisheye opted for a table-driven approach. The table-driven approach makes the protocol insensitive to traffic source/destination pair density. The fisheye update mechanism significantly reduces the overhead traffic that plagues conventional and proposed table-driven schemes. This results in good scalability to network size. However, routing performance is affected by the update interval between scopes, which is partially determined based on the mobility of the system. In the presence of high mobility, routing updates must be propagated more frequently to reflect the current network topology. However, when there is low mobility, routing updates do not need to be propagated as frequently as the topology does not change as much. The trade-off is between overhead traffic verses routing accuracy.

5 Conclusions

The ad hoc wireless network presents many challenges in routing protocol design. The goal of this thesis is to study traditional routing schemes and design and implement a new routing approach for ad hoc wireless networks.

5.1 Contributions

A new routing scheme using a link-state foundation and employing a novel fisheye updating mechanism was designed and implemented. Called Fisheye routing, this mechanism reduces the control overhead by disseminating topology information using the fisheye technique, where routing information is updated at different rates depending on the distance from the source.

Through simulation, Fisheye routing has exhibited good performance in reducing overhead control traffic. It also performs well in terms of successful packet delivery in the presence of low mobility. Proper selection of the update interval time is necessary for good successful packet delivery in the presence of high mobility.

This thesis has given insight into the problems that arise when designing routing protocols in an ad hoc wireless network, shown correct implementation functionality, and demonstrated functionality and performance of the Fisheye Routing Protocol

5.2 Future Work

Current ad hoc routing approaches have introduced several new paradigms, such as exploiting user demand, the use of location, association parameters, and updating mechanisms. However, it is not clear that any particular algorithm or class of algorithm is the best for all scenarios, each protocol has definite advantages and disadvantages, and is well suited for certain situations. A key characteristic to the success of widespread use of a ad hoc wireless routing protocol is flexibility. A flexible ad hoc routing protocol could responsively invoke table-driven and/or on-demand approaches based on situations and communication requirements. The "toggle" between these two approaches may not be trivial since concerned nodes must be "in sync" with the toggling. Coexistence of both approaches may also exist in spatially clustered ad hoc groups, with intracluster employing table-driven approach and intercluster employing the demand-driven approach, or vice-versa. Further work is necessary to investigate the feasibility and performance of hybrid ad hoc routing approaches.

Other features of ad hoc networks that can be examined not addressed in this research are 1) Multicast routing [GCZ98] and 2) Quality of Service(QoS) support. Multicast is desirable to support multiparty wireless communications. Since the multicast tree is no longer static, the multicast routing protocol must be able to copy with mobility, including multicast membership dynamics. In terms of QoS, given the problems associated with the dynamics of nodes, hidden terminals, and fluctuating link characteristics, support end-to-end QoS is a nontrivial issue that requires in-dept investigation.

The field of ad hoc mobile networks is rapidly growing and changing, and while there are still many challenges that need to be met, it is likely that such networks will see widespread use within the next few years.

6 References

[Bak97] D. Baker, et al., "Flat vs. Hierarchical Network Control Architecture," ARP/DARPA Workshop on Mobile Ad-Hoc Networking, March 1997.

[Bag98] R. Bagrodia and et. al, "Parsec: A Parallel Simulation Environment for Complex Systems", Computer, Vol. 31, October 1998, pp. 78-85.

[BJM98] J. Broch, D. Johnson, and D. Maltz, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks," IETF Internet draft, Dec. 1998.

[BG87] D. Bertsekas and R. Gallager. *Routing in Data Networks,* chapter 5., Prentice Hall, second edition, 1987.

[BJ98] J. Broch, D. John Johnson, and D. Maltz, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks", IETF Internet-Draft, draft-ietf-manet-dsr-00.txt, Mar. 1998

[Chi97] C. Chiang, "Routing in Clustered Multihop, Mobile Wireless Networks with Fading Channel," Proceedings of IEEE SICON '97, April 1997, pp. 197-211.

[Dub97] R. Dube, "Signal Stability based Adaptive Routing for Ad-Hoc Mobile Networks," IEEE Personal Communication, Feb. 1997, pp. 36-45.

[GCZ98] M. Gerla, C. Chiang, and L. Zhang, "Tree Multicast Strategies in Mobile, Multihop Wireless Networks," ACM Mobile Networks and Applications, January 1998.

[HP98] Z. Haas and M. Pearlman, "The Performance of Query Control Schemes for the Zone Routing Protocol", ACM SIGCOMM '98.

[ICP99] A. Iwata, C.-C. Chiang, G. Pei, M. Gerla, and T.-W. Chen, "Scalable Routing Strategies for Ad Hoc Wireless Networks", IEEE Journal on Selected Areas in Communications, Aug. 1999, pp. 1369-79.

[Jaf86] J.M. Jaffer and et al. "Subtle Design Issues in the implementation of Distributed, Dynamic Routing Algorithms", *Computer Networks and ISDN systems*, 1986, pp. 147-68

[JLT99] M. Jiang, J. Li, and Y. Tay, "Cluster Based Routing Protocol", August 1999, IETF Internet-Draft.

[JM96] D. Johnson and D. Maltz, "Dynamic Source Routing in Ad Hoc Networks", *Mobile Computing*, Kulwer, 1996, pp. 152-81.

[JM99] D. Johnson and D. Maltz, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks", October 1999 IETF Internet-Draft.

[JN99] M. Joa-Ng and I. Lu, "A Peer-to-Peer zone-based two-level link state routing for mobile Ad Hoc Networks," IEEE Journal on Selected Areas in Communications, Special Issue on Ad-Hoc Networks, Aug. 1999, pp.1415-25.

[JT87] J. Jubin and J. Tornow, "The DARPA Packet Radio Network Protocols," Proceedings of IEEE, vol. 75, no. 1, 1987, pp. 21-32

[KS71] L. Kleinrock and K. Stevens, "Fisheye: A Lenslike Computer Display Transformation," Computer Science Department, UCLA, CA Tech. Report, 1971.

[Man00] Mobile Ad-hoc Networks (MANET). URL:www.ietf.org/html.charters/manet-charter.html. February 2000. Work in progress.

[MG96] S. Murthy and J.J. Garcia-Luna-Aceves, "An Efficient Routing Protocol for Wireless Networks," ACM Mobile Networks and Applications, Routing in Mobile Communication Networks, Oct. 1996, pp. 183-97.

[Mis99] P. Misra, "Routing Protocols for Ad Hoc Mobile Wireless Networks", Computer Science Department, Ohio State University, 1999.

[Moy98] J. Moy, *OSPF: Anatomy of an Internet Routing Protocol.* Reading, Massachusetts, Addison Wesley Longman, Inc., 1998.

[PB94] C. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing(DSDV) for Mobile Computers", Computer Communication Review, October 1994, pp.234-244.

[PB96] L. Peterson and B. Davie, *Computer Networks – A Systems Approach.* San Francisco, Morgan Kaufmann Publishers Inc., 1996.

[PC97] V. Park and M. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," Proceedings of INFOCOM '97, Apr. 1997.

[PR99] C.E. Perkins and E.M. Royer, "Ad-hoc On-Demand Distance Vector Routing," Proceedings of 2nd IEEE Workshop of Mobile Computer Systems and Applications, Feb. 1999, pp. 90-100.

[PC97] V. Park and M. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks", Proceeedings of INFOCOM '97, April 1997.

[Per00] R. Perlman, *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols- 2nd Edition*. Reading, Massachusetts, Addison Wesley Longman, Inc., 2000.

[PR98] C. Perkins and E. Royer, "Ad Hoc On Demand Distance Vector(AODV) Routing," IETF Internet draft, Nov. 1998.

[PRD99] C. Perkins, E. Royer, and S. Das, "Ad Hoc On-demand Distance Vector Routing", October 1999 IETF Internet-Draft.

[RT99] R. Royer and C. Toh, "A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks", IEEE Personal Communications, Vol. 6, No.2, pp.46-55, April 1999.

[Sed83] R. Sedgewick. *Weighted Graphs,* chapter 31. Addison-Wesley, 1983.

[Toh96] C. Toh, "A Novel Distributed Routing Protocol to Support Ad-Hoc Mobile Computing," Proceedings of 1996 IEEE 15th Annual International Conference on Computing and Communication, Mar. 1996, pp. 480-86.

[Toh97] C. Toh, "Associativity-Based Routing for Ad-Hoc Mobile Networks," Wireless Personal Communication, vol. 4, no.2, Mar. 1997, pp. 1-36.

[ZA91] W. Zaumen and J. Aceves, "Dynamics of Distributed Shortest-path Routing Algorithms", Proceedings on Communication Architecture and Protocols, September 1991, pp 31-42.

[ZBG98] X.Zeng, R. Bagrodia, and M.Gerla, "GloMoSim: A library for the parallel simulation of large-scale wireless networks," in Proc. 12th Workshop Parallel and Distributed Simulations- PADS'98, pp. 154-161.

7 Appendix- Code Listing

7.1 Router.C

```
#include <Router.h>

#include <IntfUdp.h>

Router::Router(const String& address) : MultiPort("Router", 2)
{
  // Initialization
  cNextTime_SCAN_DB = Time::Now()      + EV_TIMER_SCAN_DB;
  cNextTime_UPDATE_LSP = Time::Now()   + EV_TIMER_UPDATE_LSP;
  cNextTime_SEND_HELLO = Time::Now()   + EV_TIMER_SEND_HELLO;
  cNextTime_DECREMENT_AGE = Time::Now()    + EV_TIMER_DECREMENT_AGE;
  cNextTime_FORWARD_DB = Time::Now() + EV_TIMER_FORWARD_DB;
  cLocalNode = address;

  // Init self in LSP Database
  Router_DB_Entry index;
  index.cSrc = cLocalNode;
  index.cHops = 0;
  index.cSequence = 0;
  index.cAge = COUNTER_AGE;
  index.cValid = 1;
  index.cSend_Flag = 0;
  index.cIs_Neighbor = 0;
  cDB[cLocalNode] = index;
}

void Router::PrintEntry(String s) {

  Router_DB_Entry entry;
  entry = cDB[s];


  cout << "------>FOR ENTRY: "<< s << "<--------\n";
  cout << "cSrc: " << entry.cSrc <<endl;
  cout << "cHops: " << entry.cHops <<endl;
  cout << "cSequence: " << entry.cSequence <<endl;
  cout << "cAge: " << entry.cAge <<endl;
  cout << "cValid: " << entry.cValid <<endl;
  cout << "cNeighbor_list:\n";
  typedef map<String, int>::const_iterator CI;
  for (CI p =
entry.cNeighbor_list.begin();p!=entry.cNeighbor_list.end();
       ++p) {
    cout << "Node:"<< p->first << "\tCost:" << p->second << endl;
  }

  cout << "cTTS: " << entry.cTTS <<endl;
  cout << "cSend_Flag: " << entry.cSend_Flag <<endl;
  cout << "cHELLO_timer: " << entry.cHELLO_timer <<endl;
  cout << "cIs_Neighbor: " << entry.cIs_Neighbor <<endl;
  cout << "--------------------------------------\n";
}

void Router::ComputeRoute(void) {

  if (Debug("Router.ComputeRoute")) {
       cout << "ComputeRoute: Running at Time:" << Time::Now()<<endl;
  }

  map<String, Route_info> PATH;
  map<String, Route_info> TENT;
  map<String, String> temp_DB;
  Route_info index;
  String current_node;
  int path_cost;
  String node;
  int cost;
  typedef map<String, int>::const_iterator CI;
  typedef map<String, Route_info>::const_iterator RI;

  // Flags
  int store_tent = 0;
  int empty_tent = 0;
  int min_cost = 0;
  String lowest_node;

  // Begin with 'self' as root
  index.cost = 0;
  index.forw = cLocalNode;
  PATH[cLocalNode] = index;
  current_node = cLocalNode;


  do {
    /* Look at PATH's LSPs and store better paths into TENT */
    if (Debug("Router.ComputeRoute")) {
      cout << "Current Node: " << current_node <<endl;
    }
    if (cDB[current_node].cValid == 1) {
      map<String, int> &neighbors = cDB[current_node].cNeighbor_list;
      for (CI p = neighbors.begin(); p!=neighbors.end(); ++p) {
      node = p->first;
      cost = p->second;
      path_cost = PATH[current_node].cost+cost;
      // Check if cost is shorter than value stored in TENT
      store_tent = 0;
      if (PATH.find(node) != PATH.end()) store_tent = 0;
      else {
        if (TENT.find(node) == TENT.end()) store_tent = 1;
        else {
          if (path_cost < TENT[node].cost) store_tent = 1;
        }
      }
      // Store into TENT
      if (store_tent == 1) {
        index.cost = path_cost;
        if (PATH[current_node].forw ==cLocalNode) {
          index.forw = node;
        }  else index.forw = PATH[current_node].forw;
        if (Debug("Router.ComputeRoute")) {
          cout << "Store node into TENT: " << node << ", " <<
            "Forward: "<< index.forw <<endl;
        }
        TENT[node] = index;
      }
      }
    }

    // Find node in TENT with minimal cost and move to PATH
    empty_tent = 1;
    for (RI p = TENT.begin(); p!= TENT.end(); ++p) {
      empty_tent = 0;
      min_cost = 9999999;
      node = p->first;
      cost = p->second.cost;
      if (cost < min_cost) lowest_node = node;
    }
    // Move entry from TENT into PATH
    if (empty_tent == 0) {
      PATH[lowest_node] = TENT[lowest_node];
      if (Debug("Router.ComputeRoute")) {
      cout << "MOVING ENTRY FROM TENT TO PATH:"<< lowest_node <<
        ", FOWARD:" << TENT[lowest_node].forw << ", COST:" <<
        TENT[lowest_node].cost << endl;
      }
      TENT.erase(lowest_node);
      current_node = lowest_node;
    };
  } while (empty_tent == 0);

  // Copy over forwarding database
  for (RI p = PATH.begin(); p!= PATH.end(); ++p) {
    temp_DB[p->first] = p->second.forw;
    if (Debug("Router.ComputeRoute")) {
      cout << "Route: "<< p->first << ", Forward: "<< p->second.forw
<<endl;
    }
  }
  Forward_DB = temp_DB;

  if (Debug("Router.ComputeRoute")) {
    cout << "ComputeRoute: Ending at Time:" << Time::Now()<<endl;
  }

}




void Router::HandleLSP(const Packet & p) {

  if (Debug("Router.HandleLSP")) {
```

```
  cout << "HandleLSP: Running at Time:" << Time::Now()<<endl;
}

// Extract values
String source = p["src"];
String node = p["node"];
int hops = p["hops"].Convert((int*)0);
int age = p["age"].Convert((long*)0);
int sequence = p["sequence"].Convert((long*)0);
String payload = p["payload"];

String d = ":#:";
Router_DB_Entry index;
map<String, int> List;
String temp;
int cost;
int new_entry = 0;

// Need to de-serialize payload
do {
  temp = payload.Split(d,&payload);
  cost = payload.Split(d,&payload).Convert((int*)0);
  if (temp.length()) List[temp] = cost;
} while (temp.length());


if (cDB.find(node) == cDB.end()) { // New Entry
  new_entry = 1;
}

index = cDB[node];
if ((index.cSequence < sequence) || (index.cAge == 0) ||
    (new_entry == 1) ) {
  // Valid LSP, update into database
  index.cSrc = source;
  index.cHops = hops;
  index.cSequence = sequence;
  index.cAge = age;
  index.cValid = 1;
  index.cNeighbor_list = List;
  // Fisheye update calculation
  index.cTTS = Time::Now() + EV_TIMER_UPDATE_LSP * pow(hops, ALPHA);
  index.cSend_Flag = 1;
  if (new_entry == 1) {
    index.cHELLO_timer = 0;
    index.cIs_Neighbor = 0;
  }

  // Store back in database
  cDB[node] = index;

  if (Debug("Router.HandleLSP")) {
    cout << "Store Packet in Database:\n" ;
    PrintEntry(node);
  }

  // Recompute Routes
```

```
    ComputeRoute();
  } else {
      if (Debug("Router.HandleLSP")) {
      cout << "HandleLSP: LSP Rejected from node: " << node << endl;
      }
  }

}

void Router::HandleHELLO(const Packet & p) {

  int new_neighbor = 0;
  String Source = p["src"];

  if (Debug("Router.HandleHELLO")) {
       cout << "HandleHELLO: Running at Time:" << Time::Now()<<
       ", from node:"<< Source << endl;
  }



  if (cDB.find(Source) == cDB.end()) { // New Neighbor
    new_neighbor = 1;
  } else if (cDB[Source].cIs_Neighbor == 0) new_neighbor = 1;

  // Initialize entries
  if (new_neighbor == 1) {
    if (Debug("Router.HandleHELLO")) {
      cout << "HandleHELLO: New Neighbor: " << Source << endl;
    }

    cDB[Source].cSrc = "";
    cDB[Source].cHops = 0;
    cDB[Source].cSequence = 0;
    cDB[Source].cAge = 0;
    cDB[Source].cValid = 0;
    cDB[Source].cSend_Flag = 0;
    cDB[Source].cAck_Flag = 0;
  }

  // Reset HELLO timer and update into LSP Database
  cDB[Source].cIs_Neighbor = 1;
  cDB[Source].cHELLO_timer = COUNTER_HELLO;
  // Update neighbor list in LocalNode entry
  cDB[cLocalNode].cNeighbor_list[Source] = COST_ROUTE;

  if (new_neighbor == 1) {
    // Must send out new LSP for self
    // Currently suppress event triggered LSPs because of too much
overhead traffic.
    // SendLSP(cLocalNode);
    // Recompute Routes
    ComputeRoute();
  }
  if (Debug("Router.HandleHELLO")) {
    cout << "Store HELLO Information in Database:\n" ;
    PrintEntry(Source);
  }

}



void Router::HandleDATA(const Packet & q) {

  Packet p = q;
  String SRC = p["src"];
  String DST = p["dst"];
  String Net_SRC = p["net_src"];
  String Net_DST = p["net_dst"];
  String Data = p["data"];
  String Next_SRC;
  String Next_DST;

  if (Debug("Router.HandleDATA")) {
       cout << "HandleDATA: Running at Time:" << Time::Now()<<endl;
       cout << "Received DATA: SRC:"<<SRC<<", DST:"<<DST<<
       ", NET_SRC:"<<Net_SRC<<", NET_DST:"<<Net_DST<< endl;
  }

  if (cLocalNode == Net_DST) {  // Final Destination
    //    cout << "Data packet reached final destination:\n"<< p <<
endl;
    ToAbove(p);
  } else {  // Intermediate Node
    if (Forward_DB.find(Net_DST) == Forward_DB.end()) {
      cout << "Router does not have path to destination: "
         << Net_DST << "!!! Packet discarded!\n";
      return;
    } else {
      Next_SRC = cLocalNode;
      Next_DST = Forward_DB[Net_DST];
    }

    // Update fields
    p["src"] = Next_SRC;
    p["dst"] = Next_DST;

    // Pass it back down.
    if (Debug("Router.HandleDATA")) {
      cout << "Forwarding Packet to node: " << Next_DST << endl;
    }
    ToBelow(p);
  }
  return;
}

void Router::HandleACK(const Packet & p) {

  if (Debug("Router.HandleACK")) {
       cout << "HandleACK: Running at Time:" << Time::Now()<<endl;
  }
  // XXX Unfinished
  cout << "Received ACK packet.. yippy!\n";
  return;
}



void Router::SendLSP(String node) {

  if(Debug("Router.SendLSP")) {
    cout << "SendLSP: Node: "<< node << " at:" << Time::Now()<<endl;
  }

  // Sends out LSP Packet
  Router_DB_Entry index;
  Packet p;
  String ListNeighbors;
  String d = ":#:";
  String h = "";

  index = cDB[node];
  if (index.cValid == 0) {  // XXX node not valid in cDB
    cout << "FATAL ERROR SendLSP: Node not valid in LSP Database!\n";
    exit(0);
  }
  // Extract LSP information and turn into String
  typedef map<String, int>::const_iterator CI;
  for (CI
i=index.cNeighbor_list.begin();i!=index.cNeighbor_list.end();++i) {
    h = h + i->first + d + String::Convert(i->second) + d;
  }

  // Construct LSP Packet
  p["opcode"]        = OP_LSP;
  p["src"]           = cLocalNode;
  p["node"]          = node;
  p["hops"]          = String::Convert(index.cHops+1);
  p["sequence"]      = String::Convert(index.cSequence);
  p["age"]           = String::Convert(index.cAge);
  p["payload"]       = h;

  if (SWITCH_BROADCAST == 0) {  // Send unicast
    typedef map<String, Router_DB_Entry>::iterator IT;
    for (IT i = cDB.begin(); i != cDB.end(); ++i) {
      index = i->second;
      if ((index.cIs_Neighbor == 1) && (i->first != cDB[node].cSrc)) {
        p["dst"] = i->first;
        ToBelow(p);
      }
    }
  }

  if (SWITCH_BROADCAST == 1) {  // Send Broadcast
    p["dst"] = BROADCAST;
    ToBelow(p);
  }
}
```

```
void Router::SendHELLO(void) {
  Packet p;

  if (Debug("Router.SendHELLO")) {
    cout << "SendHELLO: Running at Time:" << Time::Now()<<endl;
  }

  p["opcode"]          = OP_HELLO;
  p["timestamp"]       = String::Convert(Time::Now());
  p["src"]             = cLocalNode;
  p["dst"]             = BROADCAST;

  ToBelow(p);
}

void Router::UpdateOwnLSP(void) {

  if (Debug("Router.UpdateOwnLSP")) {
    cout << "UpdateOwnLSP: Running at Time:" << Time::Now()<<endl;
  }

  // increment sequence number
  cDB[cLocalNode].cSequence++;
  // reset age
  cDB[cLocalNode].cAge = COUNTER_AGE;
  // Set send flag
  cDB[cLocalNode].cSend_Flag = 1;
  cDB[cLocalNode].cTTS = Time::Now();
  cDB[cLocalNode].cValid = 1;

}

void Router::DecrementAge(void) {

  Router_DB_Entry index;
  int need_ComputeRoute = 0;
  int need_SendLSP = 0;

  if (Debug("Router.DecrementAge")) {
       cout << "DecrementAge: Running at Time:" << Time::Now()<<endl;
  }

  typedef map<String, Router_DB_Entry>::iterator CI;
  for (CI i = cDB.begin(); i != cDB.end(); ++i) {
    index = i->second;
    // only look at valid entries for decrementing ages
    if (index.cValid == 1) {
      // Decrement Age
      index.cAge--;
      if (index.cAge <= 0) {
      if (Debug("Router.DecrementAge")) {
        cout << "DECREMENT_AGE: Entry:"<< i->first << " LSP age is
zero!\n";
      }
      index.cValid = 0;
      index.cSend_Flag = 0;
      index.cAge = 0;
```

```
        need_ComputeRoute = 1;
        }
      }
      //  Only look at Neighbors for decrementing hello timers
      if (index.cIs_Neighbor == 1) {
        index.cHELLO_timer--;
        if(index.cHELLO_timer <= 0) {
        if (Debug("Router.DecrementAge")) {
          cout << "DECREMENT_AGE: Entry:"<< i->first <<
            " HELLO age is zero!\n";
        }
        index.cIs_Neighbor = 0;
        index.cHELLO_timer = 0;
        // Remove Entry from Local Node neighbor_list entry
        cDB[cLocalNode].cNeighbor_list.erase(i->first);
        need_SendLSP = 1;
        need_ComputeRoute = 1;
        }
      }
      // Write back entry into Master database
      i->second = index;
    }

    // Print out
    if ((need_SendLSP == 1) || (need_ComputeRoute == 1)) {
      if (Debug("Router.DecrementAge")) {
        typedef map<String, Router_DB_Entry>::iterator CI;
        for (CI i = cDB.begin(); i != cDB.end(); ++i) {
        PrintEntry(i->first);
        }
      }
    }


    // Check if we need to recompute Routes
    if (need_ComputeRoute == 1) {
      if (Debug("Router.DecrementAge")) {
        cout << "DECREMENT_AGE: Recomputing Routes!\n";
      }
      ComputeRoute();
    }
    if (need_SendLSP == 1) {
      if (Debug("Router.DecrementAge")) {
        cout << "DECREMENT_AGE: Sending out LSP!\n";
      }
      // Send out own LSP because of neighbor change
      SendLSP(cLocalNode);
    }
  }


  void Router::ScanLSPdb(void) {

    Router_DB_Entry index;
    String Node;

    if (Debug("Router.ScanLSPdb")) {
```

```
    // cout << "ScanLSPdb: Running at Time:" << Time::Now()<<endl;
  }

  typedef map<String, Router_DB_Entry>::iterator CI;
  for (CI i = cDB.begin(); i != cDB.end(); ++i) {
    Node = i->first;
    index = i->second;

    if (index.cValid == 0) continue; // only look at valid entries
    // Check for SEND FLAG
    if (Debug("Router.ScanLSPdb")) {
      cout << "ScanLSPdb: Node:" << Node << ", TTS:" << index.cTTS <<
      ", TimeNow:" << Time::Now() << endl;
    }
    if (index.cSend_Flag == 1) {
      if (index.cTTS <= Time::Now()) {
      SendLSP(Node);
      index.cSend_Flag = 0;
      i->second = index;
      }
    }

    // Check for ACK FLAG
    if (index.cAck_Flag == 1) {
      // XXX No ACKS for now...
    }

  }
}


Time Router::DoWork() {


  if (Debug("Router.DoWork")) {
       cout << "DoWork: Running at Time:" << Time::Now()<<endl;
  }

  if (Debug("Router.PrintForwardDB")) {
    if(Time::Now() >= cNextTime_FORWARD_DB) {
      cNextTime_FORWARD_DB = Time::Now() + EV_TIMER_FORWARD_DB;
      typedef map<String, String>::const_iterator CI;
      cout <<"----------FORWARDING DATABASE-------------\n";
      for (CI p = Forward_DB.begin();p!=Forward_DB.end(); ++p) {
      cout << "Node:"<< p->first << "\tForward:" << p->second << endl;
      }
      cout <<"-----------------------------------------\n";
    }
  }

  if(Time::Now() >= cNextTime_UPDATE_LSP) {
    UpdateOwnLSP();
    cNextTime_UPDATE_LSP = Time::Now() + EV_TIMER_UPDATE_LSP;
  }
  if(Time::Now() >= cNextTime_SEND_HELLO) {
    // Send HELLO Messages
    SendHELLO();
    cNextTime_SEND_HELLO = Time::Now() + EV_TIMER_SEND_HELLO;
  }
  if(Time::Now() >= cNextTime_DECREMENT_AGE) {
    // 1) Decrement Age and 2) Check for neighbor livetime
    DecrementAge();
    cNextTime_DECREMENT_AGE = Time::Now() + EV_TIMER_DECREMENT_AGE;
    // Can do some type of trick here for power-saving features
  }
  if(Time::Now() >= cNextTime_SCAN_DB) {
    // LSP Check for send
    ScanLSPdb();
    cNextTime_SCAN_DB = Time::Now() + EV_TIMER_SCAN_DB;
    // Can do some type of trick here for power-saving features
  }
  return(Time::Now()-Time::Now() + 1);
}

bool Router::DoConsume(const Packet& p, size_t port) {

  switch(port) {
  case top: FromAbove(p); break;
  case bot:
    if ( ((p["dst"] == cLocalNode)||(p["dst"] == BROADCAST)) &&
       (p["src"] != cLocalNode) ) {
      if (Debug("Router.DoConsume")) {
      cout << "DoConsume: PACKET with OPCODE:"<< p["opcode"]
           << " received from SRC:" << p["src"] << endl;
      }
      FromBelow(p); break;
    } else {
      if (Debug("Router.DoConsume")) {
      cout << "DoConsume: SRC of Packet:"<< p["src"] <<". Ignored Packet
for node:" << p["dst"] << endl;
      }
    }
  default: break;
  }
  return(true);
}

void Router::FromAbove(const Packet& q) {
  Packet p = q;

  if (Debug("Router.FromAbove")) {
    cout << "FROM_ABOVE:\n" << p;
  }

    if (!p.Has("opcode")) { cout << "No OPCODE!\n"; }
  if (!p.Has("net_src")) p["net_src"] = cLocalNode;

  const String opcode = p["opcode"];
  if (opcode == OP_DATA) {
    HandleDATA(p);
  }


}
```

```
bool Router::ValidPacket(const Packet& p) {

  /* Verifies integrity of packet- checks for valid fields
     for now.  Could add checksum and other things later. */

  bool valid;

  if (!p.Has("opcode")) {
    cout << "ValidPacket(): Packet recieved has no OPCODE!\n";
    cout << p;
    return 0;
  }
  if (!p.Has("dst")) {
    cout << "ValidPacket(): Packet recieved has no LINK
DESTINATION(dst)!\n";
    cout << p;
    return 0;
  }


  const String opcode = p["opcode"];
  valid = 1;
  if (opcode == OP_LSP) {
    // Fields: src, dst, node, hops, sequence, age, payload
    if (!p.Has("node")) {
      cout << "ValidPacket(): LSP Packet recieved has no NODE(node)!\n";
      valid = 0;
    };
    if (!p.Has("sequence")) {
      cout << "ValidPacket(): LSP Packet recieved has no SEQUENCE
Number(sequence)!\n";
      valid = 0;
    };
    if (!p.Has("age")) {
      cout << "ValidPacket(): LSP Packet recieved has no AGE(age)!\n";
      valid = 0;
    };

    if (valid == 0) { cout << p; };
    return valid;
  }
  if (opcode == OP_HELLO) {
    // Fields: src, dst, timestamp
    return valid;
  }
  if (opcode == OP_DATA) {
    // Fields: src, dst, net_src, net_dst, data
    if (!p.Has("net_dst")) {
      cout << "ValidPacket(): DATA Packet recieved has no NETWORK
Destination(net_dst)\n";
      valid = 0;
    };
    if (valid == 0) { cout << p; };
    return valid;
  }
  cout << "Invalid OPCODE!\n" << p;
  return 0;
}



void Router::FromBelow(const Packet& q) {
  Packet p = q;

  // Error Checking Packets here for key fields

  if (!p.Has("opcode")) {
    cout << "FromBelow: Packet recieved has no OPCODE!\n";
    cout << p;
    return;
  }
  if (!p.Has("src")) {
    cout << "FromBelow: Packet recieved has no SRC!\n";
    cout << p;
    return;
  }

  const String opcode = p["opcode"];

  if (opcode == OP_LSP) {
    HandleLSP(p);
  }
  if (opcode == OP_HELLO) {
    HandleHELLO(p);
  }
  if (opcode == OP_DATA) {
    HandleDATA(p);
  }
  if (opcode == OP_ACK) {
    HandleACK(p);
  }
}

bool Router::ToBelow(const Packet & p) {

  if (Debug("Router.ToBelow")) {
    cout << "TO_BELOW EXECUTED!  Sending Packet:\n";
    cout << p;
  }
  return(DoProduce(p, bot));
  // return(true);
}

bool Router::ToAbove(const Packet & p) {

  if (Debug("Router.ToAbove")) {
    cout << "TO_ABOVE EXECUTED!  Sending Packet:\n";
    cout << p;
  }
  return(DoProduce(p, top));
  // return(true);
}
```

```
#include <IntfFile.h>
#include <IntfPipe.h>
#include <IntfTty.h>
#include <IntfUdp.h>
#include <IntfChild.h>
#include <UserCore.h>
#include <ProdTimer.h>
#include <MrrCore.h>
#include <RouterFilter.h>

int main(int argc, char* argv[]) {

  /* Handle command line options...
   * Valid options are:
   * -router=<String>
   * -pipe=<0:1>
   */

  String RouterOpt("-router=");
  String RouterName("Allen's Router");
  String PipeOpt("-pipe=");
  String Pipe("1");
  for (int i=1; i<argc; i++) {
    String a = argv[i];
    if(strncmp(RouterOpt.c_str(), a.c_str(), RouterOpt.length()) == 0) {
      RouterName = a.substr(RouterOpt.length());
    }
    else if (strncmp(PipeOpt.c_str(), a.c_str(), PipeOpt.length()) == 0)
{
      Pipe = a.substr(PipeOpt.length());
    }
    else {
      cout << "Invalid Options\n";
      return(1);
    }
  }



  Debug::Load("Debug-route.flags");

  FileInterface fi(STDIN_FILENO, STDOUT_FILENO);
  PktToString psIn;   // From STDIN to Router String->Packet
  PktToString psOut;  // From Router to Pipe  Packet-> String
  Router LSR(RouterName);
  RouterFilter Filter("Router-Network.filter", RouterName);
  String inpipe  = ((Pipe == "1") ? "/tmp/in"  : "/tmp/out");
  String outpipe = ((Pipe == "1") ? "/tmp/out" : "/tmp/in" );
  //  PipeInterface ether(inpipe, outpipe);


  int noCheck=false;
  int mcast = true;
  int mcastLoop = true;
  int broadcast = false;
  int reuse = true;
  UdpInterface ether("239.0.0.1:1024", "239.0.0.1:1024",
             noCheck,
             mcast,
             mcastLoop,
             broadcast,
             reuse);


  fi.ConnectTo(psIn.Port(String()));
  psIn.Port(Packet()).ConnectTo(LSR.Port(MrrLayer::top));
  LSR.Port(MrrLayer::bot).ConnectTo(Filter.Port(MrrLayer::top));
  Filter.Port(MrrLayer::bot).ConnectTo(psOut.Port(Packet()));

  //  LSR.Port(MrrLayer::bot).ConnectTo(psOut.Port(Packet()));

  ether.ConnectTo(psOut.Port(String()));

  // Set Manager and Run
  Manager m;
  m.BuildEmpire();            // Collect any strays...
  OrgChart chart;
  cout << chart.Build(m) << endl;
  cout << Worker::Payroll() << endl;
  m.TakeCharge();
}
```

7.2 Router.h

```
#include <stdlib.h>
#include <iostream.h>
#include <UtString.h>
#include <UtTime.h>
#include <UtDebug.h>
#include <MrrPacket.h>
#include <MrrLayer.h>
#include <UserCore.h>
// Opcodes for packets
#define OP_LSP               "LSP"
#define OP_HELLO             "HELLO"
#define OP_DATA              "DATA"
#define OP_ACK               "ACK"
// Broadcast address
#define BROADCAST            "BROADCAST"
// Timer definitions: in seconds
#define EV_TIMER_SEND_HELLO        1
#define EV_TIMER_SCAN_DB           1
#define EV_TIMER_UPDATE_LSP        3
#define ALPHA                      1.0
#define EV_TIMER_DECREMENT_AGE     1
#define EV_TIMER_FORWARD_DB        3
// Router Switches
#define SWITCH_BROADCAST     0
```

```
// COUNTER VALUES
#define COUNTER_HELLO        60
#define COUNTER_AGE          60
// COSTS
#define COST_ROUTE           1

struct  Router_DB_Entry {
  String cSrc;
  int cHops;
  long cSequence;
  long cAge;
  int cValid;  // determined by age
  map<String, int> cNeighbor_list;
  Time cTTS;
  int cSend_Flag;
  int cAck_Flag;
  long cHELLO_timer;
  int cIs_Neighbor;
  long cTime_stamp;
  Router_DB_Entry() :  // Constuctor
    cSrc(""),
    cHops(0),
    cSequence(0),
    cAge(0),
    cValid(0),  // determined by age
    cNeighbor_list(),
    cTTS(0.0),
    cSend_Flag(0),
    cAck_Flag(0),
    cHELLO_timer(0),
    cIs_Neighbor(0),
    cTime_stamp(0)
    {};
};

  struct Route_info {
    int cost;
    String forw;
  };




class Router : public MultiPort<Packet> {

protected:
  map<String, Router_DB_Entry> cDB;
  map<String, String> Forward_DB;

  Time cNextTime_SCAN_DB;
  Time cNextTime_UPDATE_LSP;
  Time cNextTime_SEND_HELLO;
  Time cNextTime_DECREMENT_AGE;
  Time cNextTime_FORWARD_DB;

  String cLocalNode;
  // Bunch of other global type variables here
  /// DoConsume() - Place where all packets arrive...
  virtual bool DoConsume(const Packet& p, size_t port);

  /// FromAbove() - Handle packets from above
  virtual void FromAbove(const Packet& p);

  /// FromBelow() - Handle packets from below
  virtual void FromBelow(const Packet& p);

  /// ToAbove() - Send packet above
  virtual bool ToAbove(const Packet & p);

  /// ToBelow() - Send packet below
  virtual bool ToBelow(const Packet & p);

  /// Allows us to schedule timeouts
  virtual Time DoWork();

  virtual void SendHELLO(void);

  virtual void SendLSP(String);

  virtual void UpdateOwnLSP(void);

  virtual void DecrementAge(void);

  virtual void ScanLSPdb(void);

  virtual void HandleLSP(const Packet & p);

  virtual void HandleHELLO(const Packet & p);

  virtual void HandleDATA(const Packet & q);

  virtual void HandleACK(const Packet & p);

  virtual void ComputeRoute(void);

  virtual void PrintEntry(String s);

  virtual bool ValidPacket(const Packet &p);

public:
  enum { top,bot };
  Router(const String& address);
  virtual ~Router() {};
};
```