Outdoor Wi-Fi Mesh Routing Protocol

Akshata Danivasa
Alphonse Hansel Anthony
Mani Pandian
Vikas Iyer
Vinesh Pallen

Contents

1. Abstract…… 3
2. Introduction…… 4
3. Related Work…… 5
4. Problem Formulation…… 6
5. MODEL…… 7
6. Final Design…… 9
7. Experimental Results…… 13
8. Analysis…… 15
9. Test Design…… 15
10. Performance Measurement…… 18
11. References…… 18

Abstract

Wireless mesh networks is a promising technology, which could help us realize the dream of connecting the world with “no strings attached”. These are infrastructure based multi hop networks, which have their own wireless distribution system. Each node has a number of interfaces, one of which is used as an access point. When data communication takes place in a mesh network, routing begins to take place between the nodes. The performance of routing is different for wired and wireless mesh networks. In case of a wireless mesh network, the path between two nodes tend to be much lossier than their wired counterparts. Thus, a shortest path based routing metric fails to give effective information about the path, and there is a need to look for other metrics to find out better routing paths. Such a metric should take into consideration, the other points of routing path comparison such as link quality, high throughput path, intra/ inter flow interference, etc.

To depict the above scenario, we have a wireless mesh testbed (also called as Meshbed), which has four nodes, with each node having four Madwifi cards and thus four interfaces. One from the four nodes is picked to be a control node, which acts as the gateway to the outside world. Our aim is to find the best routing paths calculated using link quality based metrics such as the 'Expected Transmission Time (ETT)'. One of the four interfaces acts as a communicator which hosts a number of processes. Each process has an independent function such as discovering neighbors, assigning channels, managing routing, etc. Once, a node gets the neighbor information, the nodes probe their neighbors to examine the link quality between them. The link quality information in the form of ETT is stored in a database at the control node, which is accessed by its routing manager to decide the best routing path.

This report talks about the algorithm used for metric calculation, our design to implement the formulated algorithm, the steps we undertook to modify the existing Madwifi driver to meet the needs of the Meshbed and the test cases to be used to examine our implementation.

Introduction

Current MeshBed implementation uses Hop count as a metric for routing in the wireless Mesh Network. Even though Hop Count is easy to compute, the disadvantage with using Hop Count is that it considers the wireless links between distant nodes, which could be slow or lossy leading to poor throughput. Using Hop count as a metric does not maximize the throughput of the flow. A two hop path over the fast links could be better than a one hop over a slow lossy link.

Thus, we need a metric that considers the **quality of the wireless links.** The main motive of this project is to implement a metric that always routes through a high throughput path. ETT is a link-quality metric that accounts for the link layer losses unlike the hop count metric. Owing to the bidirectional flow of information in the links between the nodes, the calculated link loss ratio would be asymmetrical had it been considered only in a single direction. ETT takes care of this by calculating the link loss ratios in both directions. The receiver calculates link loss using the packets coming from sender and vice versa.

The following are the benefits of the ETT metric:

- ETT is based on delivery ratios, which directly affect Throughput.
- ETT detects and appropriately handles asymmetry by incorporating loss ratios in each direction.
- Since we are obtaining the packet loss rates from the madwifi driver, ETT calculated can use precise link loss ratio measurements to make fine-grained decisions between routes.
- ETT penalizes routes with more hops, which have lower throughput due to interference between different hops of the same path
- ETT tends to minimize spectrum use, which should maximize overall system capacity. [1]

The ETT metric is particularly useful in case of a large network. As the paths get longer in this network, ETT ensures that the path chosen to route all the traffic is the one with the highest throughput. The nodes in the MeshBed have four channels, each allocated to a different radio based on the performance of the channel. This provides the best cost estimate for a particular path. The MeshBed will have higher throughput than the existing routes based on Hop Count.

Related Work

ETT & ETX

[1] explains the ETT calculation technique which was followed initially for calculating the forward and reverse path loss probability.
[6] describes the uses of more frequent broadcast probes to measure loss-rate, and very infrequent unicast probes to measure the bandwidth to each neighbor. This estimates the ETT without incurring too much overhead.
[5] describes the bit-rate selection algorithm used in the madwifi drivers which accurately calculates and maintains the transmission time to each neighbor interface including retransmission packets at the MAC level.

What is SampleRate?

SampleRate algorithm [5] is a bit-rate selection technique to maximize throughput over wireless links that are capable of multiple bit-rates. SampleRate sends most data packets at the bit-rate it believes will provide the highest throughput. SampleRate periodically sends a data packet at some other bit-rate in order to update a record of that bit-rate's loss rate. SampleRate switches to a different bit-rate if the throughput estimate based on the other bit-rate's recorded loss rate is higher than the current bit-rate's throughput. Measuring the loss rate of each supported bit-rate would be inefficient because sending packets at lower bit-rates could waste transmission time, and because successive unicast losses are time-consuming for bit-rates that do not work. SampleRate addresses this problem by only sampling at bit-rates whose lossless throughput is better than the current bit-rate's throughput. SampleRate also stops probing at a bit-rate if it experiences several successive losses. SampleRate performs better than other algorithms on links where all bit-rates suffer from significant loss.

The SampleRate Algorithm

- All packets are classified into three different buckets each of size less than 250, 1600, 3000.
- The information collected are per destination mac id.

Whenever the autorate is enabled , the madwifi ensures that for each packet the following takes place

1) Tries to send the packet at the rate specified as per SampleRate's estimated best rate with 11 retries

2) If it fails, then tries to send the packet at the same rate with 3 retries

3) Even if step 2 fails, then it tries to send the packet at 1Mbps with 3 retries.

4) Finally, the hardware gives up and notifies the driver regarding its status.

Problem Formulation

In order to calculate ETT we need to probe the link periodically to determine the quality of the link. We need to find the loss ratio of each link in the forward direction as well as in the reverse direction intermittently since this could change with the traffic flows.

In order to calculate ETT for each link we need to find the ETX for each link first. ETX is calculated based on forward and reverse ratios of the link. The forward delivery ratio, df , is the measured probability that a data packet successfully arrives at the recipient; the reverse delivery ratio, dr, is the probability that the ACK packet is successfully received. These delivery ratios can be measured as described below. The expected probability that a transmission is successfully received and acknowledged is $d f \times dr$. Because each attempt to transmit a packet can be considered a Bernoulli trial, the expected number of transmissions is:

$ETX = (1/ d f \times dr)$ *[1]*

The delivery ratios $d f$ and dr are measured using dedicated link probe packets.

- Each node broadcasts link probes of a fixed size, at an average period τ (one second in the implementation). Because the probes are broadcast, 802.11b does not acknowledge or retransmit them.
- Every node remembers the probes it receives during the last w seconds (ten seconds in our implementation), allowing it to calculate the delivery ratio from the sender at any time t as:

  $r(t) = \text{count}(t - w, t) / w/\tau$

  Count($t-w$, t) is the number of probes received during the window w, and w/τ is the number of probes that should have been received.

- In the case of the link $X \rightarrow Y$, this technique allows X to measure dr , and Y to measure $d f$. Because Y knows it should receive a probe from X every τ seconds, Y can correctly calculate the current loss ratio even if no probes arrive from X.

- Calculation of a link's ETX requires both $d f$ and dr . Each probe sent by a node X contains the number of probe packets received by X from each of its neighbors during the last w seconds. This allows each neighbor to calculate the $d f$ to X whenever it receives a probe from X. [1]

ETTi=ETXi*(S/Bi)

where,

S – Packet size used over the link i (this is a fixed quantity ,common for all links)

B_i - Bandwidth used over the link i (based on the channel assigned)

ETX_i - # of transmission required to successfully transmit a packet over the link i.

Model

We came up with a few designs implementing these ideas and incorporating them into the existing MeshBed code.

Design 1

The Neighbor Discovery Module in the MeshBed sends Hello Packets periodically to discover each of its neighbours. The Hello request module sends out a broadcast Hello packet every ten seconds. Any neighbor who receives this request (All the neighbours in range) will reply with a Hello Reply indicating the sequence number for the request it got. The reply is a unicast reply back to the sender from which it received broadcast Hello Packets. The Sender has a Hello Process module which listens for incoming acknowledgements and keeps a list of neighbours' IP addresses from which it received a reply.

We incorporated the calculation of ETX with the Neighbour Discovery Module initially.

- Each node X sends out a broadcast probe packet every 1 second similar to the Neighbour Request module
- Once the neighbor node Y receives a packet with a new sequence number, it waits for a window (10 seconds) to send a reply or an ack back to the sender. The receiver node keeps a count for the number of packets received from the sender node IP during that time interval.
- After the interval (10 seconds) the Reply module of Y sends the acknowledgement back to the sender IP. In this reply packet it includes the number of packets received from the sender IP X. This is the df for the sender node X.
- The sender node receives these acknowledgements and uses the value to store df for the link between the particular sender X and receiver Y (X -> Y) . It keeps a counter for the number of acknowledgements received for the next 10 seconds (*τ seconds*) . This is used to calculate dr for link Y -> X.
- The value of ETX is calculated and stored in a file every t seconds (10)
- This value is reported to the control node which updates the database (dbxml) from which they are picked up by the routing module which uses ETX as a cost metric for routing.

Design Flaw

- df was calculated based on broadcast packets and dr was calculated based on unicast acknowledgements received. Thus the ETX value calculated did not have symmetric values for df and dr.
- The acknowledgements for the dr were unicast and used retransmissions whereas the df was calculated on broadcast packets which does not have retransmissions.
- The broadcast and unicast packet could be sent at different data rates (broadcast is always sent at the lowest rate). This brings discrepancies to the accurate calculation of ETT.

Design 2

In order to make the df and dr calculation uniform, we made the following changes

- Use, the existing Neighbour Discovery module without any changes made to it
- Create a new module for calculation of ETX. This module spawned three threads , one for sending the broadcast probe packets, one for receiving the probe packets from the other nodes and one thread for the calculation of ETX every t seconds (10 seconds)
- The sending thread sends a broadcast probe packet every second in a continuous loop. Each broadcast packet has the information for each of the neighbor IP's (list of IP's from which I received broadcast probes) and how many packets I received from each of those neighbours in the last t seconds.
- The receiving thread listens to the broadcast packets from all the neighbours and searches for its IP in the packet. This value is updated as df for the link X-> Y.
- The receiving thread also keeps count on the number of such broadcast packets received from each neighbor Y. This is the dr for the link Y->X
- The thread for the calculation of ETX wakes up every w seconds (w is size of the window) and for each IP calculates the ETX using the formula

  $$ETX = 1/ df * dr$$

  The df and dr are the most recent values that are calculated.
- This value of ETX is stored in a file as the metric for that link towards the neighbor IP.
- This value is reported to the control node which updates its database.

Design Flaw

- For higher data rates the broadcast packet range is much higher than unicast packet range.
- The time intervals at which each neighbor calculates the df and dr's are skewed

  Thus while calculation of the ETX, the ETX takes df and dr values which might not correspond to the same time instances. Thus the calculation of an accurate value for ETX is also erroneous.

Final Design

Madwifi Driver Modification:

The Atheros chipset is accessed by the using the madwifi driver implementation. Since this is open source project the source code is available for modification. The madwifi has three different algorithm implementation to find out the best optimum rate to send packets to a specified destination. This is known as autorate.

The three implemented algorithms are

1) ONOE

2) AMRR

3) SampleRate

By default madiwifi uses the SampleRate algorithm to support the autorate feature and hence we have modified the driver code to export the required arguments as per the needs of the WiFi Meshbed.

Changes to the Sample Rate Algorithm:

The major difference between the algorithm in the Thesis presented by John Bicket[5] and the implementation in madwifi driver are

1) Instead of recalculating the average transmission time for the bit-rate based on the sum of transmission times and the number of successful packets sent at that bit-rate, the implementation uses Exponential Weighted Mean Average(EWMA) to calculate the Autorate. To avoid using stale information, the original SampleRate algorithm only calculates the average transmission time over packets that were sent within the last averaging window(10 seconds).

2) Instead of using every 10th packet to select a random rate for probing, the implementation does the same only when the estimated time is less than ath_sampe_rate% of the average time calculated for that packet size to avoid choppiness.

Implementation of Routing using Sample rate

As part of the project requirement, we wanted to implement the ETT (Expected Transmission Time) metric[4]. It specifies the amount of time taken to send a packet over the radio, comparing the values obtained over possible radio channels. This provides a metric to determine the shortest path for routing.

SampleRate algorithm maintains a variable "average_tx_time" as part of its calculation which is the EWMA variable that stores the time taken to send a packet on a specific rate. This is maintained for each of the possible rates supported by the hardware and in our case, 12 possible values.

The transmission time for each sample(probe packet) is calculated as follows:

Applications Places System mesh2 Sun Apr 19, 8:13 PM

mesh2@overdrive:~

File Edit View Terminal Tabs Help

```
calling tx_complete from ath_tx_processq in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
Inside the function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
ath_rate_tx_complete()ts_status :OK
Inside the function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
update_stats():The total time taken for this run 7042 and the smoothened value :18658
Exiting function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
Exiting function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
start of function ath_tx_start in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
calling find rate from ath_tx_start in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
Inside the function ath_rate_findrate in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
Auto rate is set and the index value in try 2 mrr 1 ATH_TXMAXTRY 11
switch rate 00:02:6f:4b:0f:3c from 96 to 96
The index values of the new rate(00:02:6f:4b:0f:3c): 11/12
Inside the function ath_rate_setupxtxdesc in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sampl
ath_rate_setupxtxdesc():For 00:02:6f:4b:0f:3c=>(8,3):(27,3):(0,0)
Exiting function ath_rate_setupxtxdesc in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
calling tx_complete from ath_tx_processq in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
Inside the function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
ath_rate_tx_complete()ts_status :OK
Inside the function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
update_stats():The total time taken for this run 3216 and the smoothened value :20639
Exiting function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
Exiting function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
start of function ath_tx_start in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
start of function ath_tx_start in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
calling tx_complete from ath_tx_processq in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
Inside the function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
ath_rate_tx_complete()ts_status :OK
Inside the function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
update_stats():The total time taken for this run 7042 and the smoothened value :21966
Exiting function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
Exiting function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
calling tx_complete from ath_tx_processq in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
Inside the function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
ath_rate_tx_complete()ts_status :OK
Inside the function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
update_stats():The total time taken for this run 3216 and the smoothened value :22063
Exiting function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
Exiting function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
start of function ath_tx_start in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
calling tx_complete from ath_tx_processq in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
Inside the function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
ath_rate_tx_complete()ts_status :OK
Inside the function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
update_stats():The total time taken for this run 32730 and the smoothened value :7897
Exiting function update_stats in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
Exiting function ath_rate_tx_complete in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
start of function ath_tx_start in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
calling find rate from ath_tx_start in file /home/mesh2/mesh/madwifi-0.9.4/ath/if_ath.c
Inside the function ath_rate_findrate in file /home/mesh2/mesh/madwifi-0.9.4/ath_rate/sample/sample.c
                                                                                    1391,21        95%
```

Entry Point into the Driver

Finds the best possible Rate

Instructs the Hardware

Exit point from the Driver

root@overdrive:/ho... [Submission-Alphonse] mesh2@overdrive:~

1) Calculate the total time taken for the first step as specified above

a) Convert the time taken into microseconds by considering the # of retries, the CIFS,DIFS and the slots used for this probe packet.

2) If it had failed for the first step,find the time taken for step two and add it to the previous result in step 1.

3) Similar to step 2 ,add the time taken to the previous result,if step 2 had failed.

4) Similar to step 3.

The total time taken is considered and smoothed out to avoid choppiness. This is the EWMA value used.

The existing Sample algorithm exports the statistics collected through the /proc filesystem. To make parsing easy for the RoutingManager we have added additional driver code to export variables specific to

our implementation to a new /proc entry called autorate. This is created per interface and the variables are classified into three different packet sizes and per destination mac ID. To access the /proc entry for autorate execute the following command

 # cat /proc/net/madwifi/ath0/autorate

```
[mesh2@overdrive ~]$ cat /proc/net/madwifi/ath0/autorate | head -10
mac_id=00:02:6f:4b:0f:3c pkt_size=250 average_tx_time=492 successive_failures=0 tries=710 total_packets=688 packets_acked=683 perfect_tx_tim
e=492 last_tx=155786660 rate=24
mac_id=00:02:6f:4b:0f:3c pkt_size=1600 average_tx_time=764 successive_failures=0 tries=40 total_packets=20 packets_acked=11 perfect_tx_time=
764 last_tx=155642039 rate=36
mac_id=00:02:6f:4b:0f:3c pkt_size=3000 average_tx_time=1072 successive_failures=9 tries=29 total_packets=9 packets_acked=0 perfect_tx_time=1
072 last_tx=153386172 rate=36
mac_id=00:02:6f:4b:0f:31 pkt_size=250 average_tx_time=469 successive_failures=0 tries=586 total_packets=527 packets_acked=508 perfect_tx_tim
e=444 last_tx=155787652 rate=54
mac_id=00:02:6f:4b:0f:31 pkt_size=1600 average_tx_time=703 successive_failures=0 tries=420 total_packets=301 packets_acked=261 perfect_tx_ti
me=672 last_tx=155727667 rate=48
mac_id=00:02:6f:4b:0f:31 pkt_size=3000 average_tx_time=1072 successive_failures=27 tries=99 total_packets=27 packets_acked=0 perfect_tx_time
=1072 last_tx=155778659 rate=36
mac_id=00:13:e8:54:d2:a9 pkt_size=250 average_tx_time=464 successive_failures=0 tries=0 total_packets=0 packets_acked=0 perfect_tx_time=464
last_tx=0 rate=36
mac_id=00:13:e8:54:d2:a9 pkt_size=1600 average_tx_time=764 successive_failures=0 tries=0 total_packets=0 packets_acked=0 perfect_tx_time=764
 last_tx=0 rate=36
mac_id=00:13:e8:54:d2:a9 pkt_size=3000 average_tx_time=1072 successive_failures=0 tries=0 total_packets=0 packets_acked=0 perfect_tx_time=10
72 last_tx=0 rate=36
mac_id=00:18:f3:4d:20:72 pkt_size=250 average_tx_time=464 successive_failures=0 tries=0 total_packets=0 packets_acked=0 perfect_tx_time=464
last_tx=0 rate=36
[mesh2@overdrive ~]$
```

Autorate from the Driver

Estimated Time Taken per packet (smoothened)

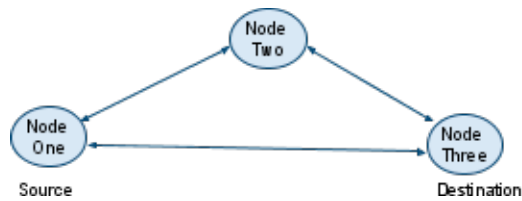
ICMP redirect

ICMP messages are control messages which are sent by gateways/destination hosts back to the source to express errors during IP datagram processing/routing. ICMP is a network layer protocol encapsulated within IP datagram to relay information.No ICMP messages are sent about the ICMP messages and it is always for the fragment zero of the fragmented IP datagram[2].

What is ICMP redirect[3]?

Whenever a better alternate route is available to a destination ,the intermediate routers pass on this information to the source host by sending ICMP messages which are redirect messages and hence ICMP redirect messages.

A brief explanation based on our topology in MeshBed and all the nodes specified below are within the same subnet.



In the above case two possible paths are available to send packets from Node One to Three. One is a direct connection and the other one is via a hop through Node Two. If Node one is configured to send packets to Three via Two, the Node two would send a ICMP redirect message back to Node One specifying that Node three is directly accessible and that the new path is better compared to routing via a hop on Node Two.

Why do we need to disable ICMP redirect in the MeshBed?

In the Mesh bed routing tables are populated based on ETT, the estimated time required to send a packet over a link. This is based on recording the retransmits required to send a packet successfully at the physical layer over a link between two nodes. Thus taking the example from the above scenario, the routing table would be populated in Node one to route all packets to three via two based on this measurement, which would be the best possible shortest path considering the costs related to retransmissions, channel quality etc. If Node two sends a ICMP redirect and Node one follows that instructions, it would represent a complete break in the calculated routing topology. Thus to avoid this, ICMP redirect needs to be disabled in all nodes participating in the Meshbed network.

Conditions to be satisfied for a successful ICMP redirect message:

1) The incoming packet should be forwarded out using the same interface.

2) Both the forward hop and the previous hop must be on the same subnet.

3) kernel should be enabled to send ICMP redirects.

4) The IP datagram should not be source routed.

Modifications in the MeshBed to avoid ICMP redirect[4]:

1) Set the following entries in the /etc/sysctl.conf file to be zero

net.ipv4.conf.all.accept_redirects = 0

net.ipv4.conf.all.send_redirects = 0

This ensures that all interfaces on the host machine do not accept/send redirects and also this configuration remains persistent between reboots.

Experimental Results

Node Configuration

Internet <=> M8 <=> M4 <=> M1

Routing table updates based on ETT metric:

Routing table on M8

```
[mesh2@localhost neighInfo]$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.2.1        10.0.2.4        255.255.255.255 UGH   0      0        0 ath1
10.0.4.1        10.0.4.4        255.255.255.255 UGH   0      0        0 ath3
10.0.3.1        10.0.3.4        255.255.255.255 UGH   0      0        0 ath2
10.0.1.1        10.0.1.4        255.255.255.255 UGH   0      0        0 ath0
10.0.4.0        0.0.0.0         255.255.255.0   U     0      0        0 ath3
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 ath0
10.0.2.0        0.0.0.0         255.255.255.0   U     0      0        0 ath1
10.0.3.0        0.0.0.0         255.255.255.0   U     0      0        0 ath2
[mesh2@localhost neighInfo]$
```

Routing table on M4

```
[mesh2@four neighInfo]$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.4.0        0.0.0.0         255.255.255.0   U     0      0        0 ath3
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 ath0
10.0.2.0        0.0.0.0         255.255.255.0   U     0      0        0 ath1
10.0.3.0        0.0.0.0         255.255.255.0   U     0      0        0 ath2
0.0.0.0         10.0.4.8        0.0.0.0         UG    0      0        0 ath3
0.0.0.0         10.0.3.8        0.0.0.0         UG    0      0        0 ath2
0.0.0.0         10.0.2.8        0.0.0.0         UG    0      0        0 ath1
0.0.0.0         10.0.1.8        0.0.0.0         UG    0      0        0 ath0
[mesh2@four neighInfo]$
```

Routing table on M1

```
[mesh2@overdrive neighInfo]$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.3.8        10.0.3.4        255.255.255.255 UGH   0      0        0 ath2
10.0.1.8        10.0.1.4        255.255.255.255 UGH   0      0        0 ath0
10.0.4.8        10.0.4.4        255.255.255.255 UGH   0      0        0 ath3
10.0.2.8        10.0.2.4        255.255.255.255 UGH   0      0        0 ath1
10.0.4.0        0.0.0.0         255.255.255.0   U     0      0        0 ath3
10.0.1.0        0.0.0.0         255.255.255.0   U     0      0        0 ath0
10.0.2.0        0.0.0.0         255.255.255.0   U     0      0        0 ath1
10.0.3.0        0.0.0.0         255.255.255.0   U     0      0        0 ath2
152.14.96.0     0.0.0.0         255.255.254.0   U     0      0        0 eth0
0.0.0.0         10.0.4.4        0.0.0.0         UG    0      0        0 ath3
0.0.0.0         10.0.3.4        0.0.0.0         UG    0      0        0 ath2
0.0.0.0         10.0.2.4        0.0.0.0         UG    0      0        0 ath1
0.0.0.0         10.0.1.4        0.0.0.0         UG    0      0        0 ath0
[mesh2@overdrive neighInfo]$
```

Traceroute from M1 to M8

```
[mesh2@overdrive neighInfo]$ traceroute 10.0.1.8
traceroute to 10.0.1.8 (10.0.1.8), 30 hops max, 40 byte packets
 1   (10.0.1.4)  0.425 ms  2.214 ms  2.096 ms
 2   (10.0.1.8)  5.653 ms  5.723 ms  5.761 ms
[mesh2@overdrive neighInfo]$ traceroute 10.0.1.4
traceroute to 10.0.1.4 (10.0.1.4), 30 hops max, 40 byte packets
 1   (10.0.1.4)  5.679 ms  5.091 ms  5.496 ms
[mesh2@overdrive neighInfo]$
```

Traceroute from M1 to M4

```
[mesh2@overdrive neighInfo]$ traceroute 10.0.1.8
traceroute to 10.0.1.8 (10.0.1.8), 30 hops max, 40 byte packets
 1   (10.0.1.4)  0.425 ms  2.214 ms  2.096 ms
 2   (10.0.1.8)  5.653 ms  5.723 ms  5.761 ms
[mesh2@overdrive neighInfo]$ traceroute 10.0.1.4
traceroute to 10.0.1.4 (10.0.1.4), 30 hops max, 40 byte packets
 1   (10.0.1.4)  5.679 ms  5.091 ms  5.496 ms
[mesh2@overdrive neighInfo]$
```

Traceroute from M4 to M8

```
[mesh2@four neighInfo]$ traceroute 10.0.1.8
traceroute to 10.0.1.8 (10.0.1.8), 30 hops max, 40 byte packets
 1   (10.0.1.8)  0.667 ms  1.537 ms  1.417 ms
[mesh2@four neighInfo]$
```

Analysis

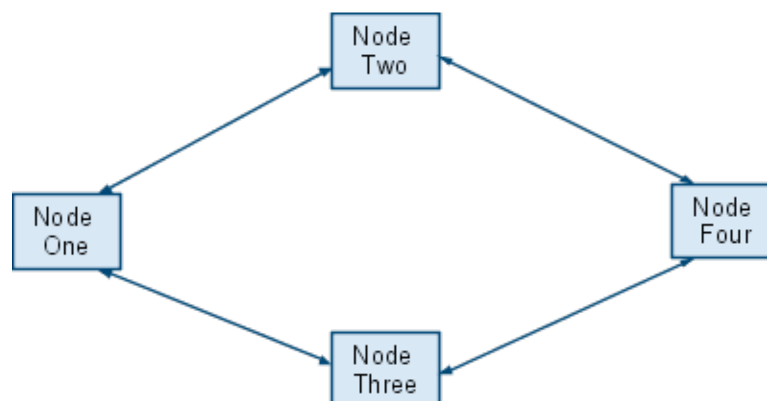
This will be provided along with the project demonstration after executing the performance tests.

Test Design

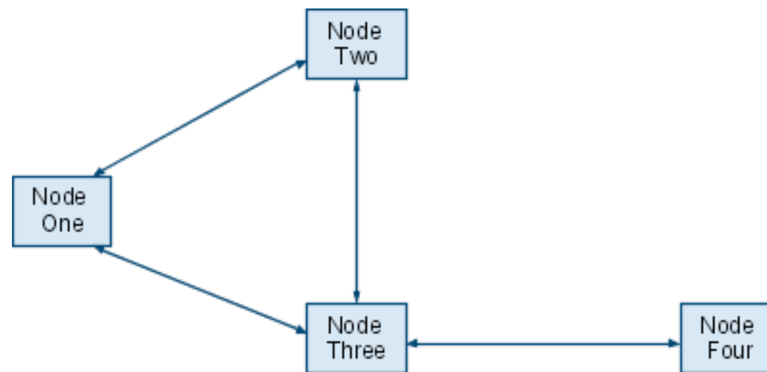
1. Test Plan:

To test our routing protocol, we will be considering two topologies. And we also test if we can successfully calculate the metrics at high loads.

Topology-1



Topology-2



Test plan for Topology-1:

a) Initiate a Connection from Node One to Node Four.
b) Load the connection between Node One & Node three.
c) Based on the routing metric path One->Two->four would be less costly compared to One->Three->Four.
d) Similarly the reverse can be verified.
e) The bandwidth would be varying parameter.

Test plan for Topology-2:

a) Initiate a connection from Node one to Node Four.
b) Two possible routes One->Three->Four or One->Two->Three->Four
c) Load the link between One->three and check that the path One->Two->Three->Four is selected.
d) Then load the link Two->Three such that, Load(one->three) < Load(Two->Three), implies that the path One->Three->Four should be selected.
These test displays the dynamism due to change in routing metric based on link load (loss due to probe packets).

<u>2. Test Automation:</u>

We run a tcl script inorder to carry out the test of our routing protocol and metric calculation for our meshbed.

The links are loaded using iperf, the network link is delimited by two hosts running Iperf.

To verify the routing path we use traceroute, it is the program that shows you the route over the network between two systems, listing all the intermediate routers a connection must pass through to get to its destination.

make and install of the applications.

setting up machines and initializing network neighborhood
0. kill all
1. Set ip address on each machine. = <node_id> <ctrl OR not>
2. Setup communicator on each node, select one out of 4 card as a management channel
<ath<id>>- the card which runs communicator.
run the above steps until success on both steps
3. Run the neighbor discovery - and metric calculation <interval>
5. If node == control node , run the topology manager on reception of neighbor and metric info
6. On this control node run routing, send this routing info to all nodes:and wait for ACK

for different topology - different sequence of link loading:
a. load links and do metric measurement
select topology and link to be loaded

for topology-1
    on node2&3: iperf -s
    on node1: iperf -c bw_test_high[i] to node2

check if node 3 gets the packet to be forwarded to node4 , Run traceroute from node one to node four

for topology-2
  on node2&3: iperf -s
  iperf -c bw on node1&2 to node2&3
  on node1: iperf -c bw_test_low[i] to node2
        : iperf -c bw_test_low[i] to node3

on node1: iperf -c bw_test_high[i] to node3
check if node 2 gets the packet to be forwarded to node3, run traceroute from node 1 to node 4

Check routing decisions, based on metric: by tracing teh path followed by the packet.

Performance Measurement

Evaluating the performance is possible by comparing the performance given by iperf for static routes and dynamic routes ie with the dynamism of routing protocol. Following can be measured using iperf.

- Routing path selection: can be measured using traceroute
- Latency (response time or RTT): can be measured with the ping command.
- Jitter (latency variation): can be measured with an Iperf UDP test.
- Datagram loss: can be measured with an Iperf UDP test.

References

[1]D. De Couto, D. Aguayo, J. Bicket, and R. Morris. High-throughput path metric for multi-hop wireless routing. In MOBICOM, 2003.

[2] http://www.ietf.org/rfc/rfc792.txt

[3] http://www.cisco.com/en/US/tech/tk365/technologies_tech_note09186a0080094702.shtml

[4] http://www.itsyourip.com/Security/how-to-disable-icmp-redirects-in-linux-for-security-redhatdebianubuntususe-tested/

[5] John Bicket. Bit-rate selection in wireless networks. Master's thesis, Massachusetts Institute of Technology, February 2005.

[6] R. Draves, J. Padhye, and B. Zill. Routing in Multi-Radio, Multi-Hop Wireless Mesh Networks. In MOBICOM, 2004.

Comments:

A few changes are made from the ECE 575 course project to the existing Mesh testbed.

1. For the ECE 575 project /24 subnet was used. Changes were made to convert it to a /16 subnet that is used currently. Changes were made to all the initial configuration script files to change the subnet mask.
2. DBXML was used for the database to store all the topology information and metrics for each link. This was changed to MYSQL which is currently used to hold the same.
3. Due to the conversion to a /16 subnet, all the interfaces were on the same subnet because of the way of addressing i.e.
   10.0. <interface id>. <nodeid>
   Hence a form of binding was done at the socket level in the communicator, to ensure the packets go out on the correct interface.
4. The implementation for computing the routes for each node changed in the Routing Manager. The logic of computation of routes remains the same. The control node acts as a root node of a tree. All nodes other than the control node have routes to and from the control node .A parent node has a route to all its children nodes and descendants.
   Only the implementation of the computation of routes changed, since now each node has a route to all of the interfaces of its neighbor node . Thus there are as many routes to each neighbor as there are number of interfaces . This has been taken care of for all the nodes as well as while computing the gateway (next hop) for each interface.