Project Status Report of Multipath Routing for Load Balancing and QoS in Computer Networks

Kyle Ebding
Computer Engineering Department
Jack Baskin School of Engineering
University of California at Santa Cruz
Email: kebding@ucsc.edu

***Abstract*—The Internet today largely uses single-path routing. Although this works, it results in central links becoming congested and other links being idle, which is not optimal for network traffic flow. One solution is to use multipath routing as a way to increase link utilization while decreasing congestion and improving quality-of-service (QoS). To do this requires the use of Multiprotocol Label Switching (MPLS) as a substitution for destination-based forwarding since there can and should be multiple paths to a given destination. To use the desired MPLS implementation for this project requires abandoning the end-to-end principle and adopting Software-Defined Networking. For this project the author is using the Ryu controller for SDN. This paper discusses the author's work on implementing a prototype of this multipath routing with MPLS and SDN multipath routing at the link layer in a local area network.**

***Index Terms*—Multipath Routing, MPLS, SDN, DSMR, Computer Networking, QoS, Ryu.**

I. Introduction and Background Knowledge

Currently, the Internet is run on a single-path routing model. This approach has been taken because it allows for distributed routing computations where an individual router does not necessarily need to know the entire network topology, only where its neighbors can reach and how good the path is (usually measured in hop count or latency). This approach requires that packets be forwarded by each router independently of every other router, which is called a *hop-by-hop* forwarding model. The problem with single-path routing and hop-by-hop forwarding is that it does not utililize network resources well because central links are used by nearly all packets, resulting in congestion that slows down the network, while leaving other links unused. This is where multipath routing comes into play. Dominant Set Multipath Routing (DSMR), described in I-A, computes a *set* of best paths rather than a single best path [1]. However, desetination-based forwarding does not support multipath routing since it assumes there is a single best path. This is where tag-switching mechanisms, such as Cisco's described in [2], will be helpful. Lastly, the tag-switching mechanism needs some kind of centralized computation to make set up the tags across the network. For this, software-defined networking (SDN) will be needed, so this paper will also look at [3].

A. Dominant Set Multipath Routing

The bulk of my work comes from the paper by Smith and Thurlow, "Practical Multipath Load Balancing with QoS." In this paper, a model of routing multiple best paths is proposed. Their work was basd on prior research but expanded on it to make it more suitable for load balancing. The basic idea of DSMR in their paper is that rather than computing a best path from each source to each destination, the computation returns a best set of paths from each source to each destination. The way they define a path as being in the best set is that it outperforms at least one other path in the best set in at least one category, and is not outperformed by another path in the best set in every category. For example, if a path P has higher bandwidth than some path in the best set, while also not having lower bandwidth and higher latency than another path in this set, then P will be an element of the best set. The paper includes some illuminating and useful graphics to help illustrate the point; they are included here.

The paper gives a detailed algorithm for the general practice of computing the dominant set of paths, including the use of what seems to me an unnecessary amount of symbols and operators mostly serve to make the paper look more formal and technical, but still help with conciseness and clarity once the symbols are understood by the reader.

The paper discusses how such a multipath routing scheme cannot simply use traditional destination-based forwarding since there can and should be more than one way to get to a given destination. It references the use of tag-switching as described by a Cisco document, which is discussed in I-B.

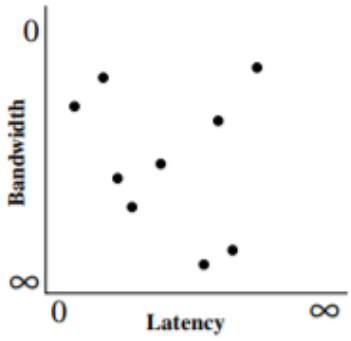

Fig. 1. A plot of the weights of the paths from a a given source to a given destination. Note that the closer to the origin the better the link is. Source: [1]

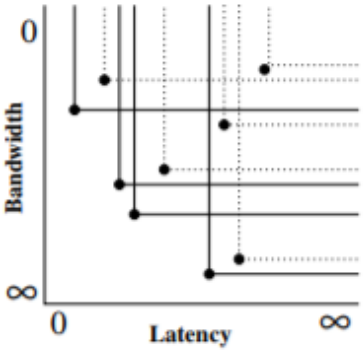

Fig. 2. A plot of the paths' QoS regions. In other words, everything upward *and* rightward of a point is considered to be inferior to the point, which means that the point is the best QoS that can be provided within that region. Source: [1]

B. Multiprotocol Label Switching

As mentioned in I-A, simple destination-based forwarding will not be sufficient for a network using multipath routing. This is where Multiprotocol Label Switching is needed. Originally called Tag-Switching when introduced by Cisco Systems, the protocol has since been renamed and given to the Internet Engineering Task Force.

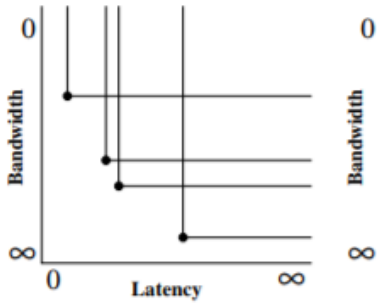

Fig. 3. A plot of the set of best paths. This represents all paths that are not dominated by another path that has both higher bandwidth and lower latency. Source: [1]

The idea of MPLS is that instead of selecting the next hop in the network based on a forwarding table that maps destination IP address ranges to which interface to forward the traffic out of, the next hop is selected by referencing a forwarding table that maps *labels* to interfaces. The difference is that because the next hop is based on a label rather than the destination address, there can be multiple labels whose forwardings will eventually lead to the same destination. The Cisco document describes that the control component of labels can be bound arbitrarily based on the needs of the network that is running MPLS: "At one extreme a tag could be associated (bound) to a group of routes (more specifically to the Network Layer Reachability Information of the routes in the group). At the other extreme a tag could be bound to an individual application flow (e.g., an RSVP flow)" [2].

The label on a packet is not static as a packet traverses the network. In label-switched forwarding, the process called *label-swapping* happens at every node a packet passes through in the network. Label-swapping involved reading the label on the incoming packet, as well as extra information included with the label, compares it to a table originally called the Tag Information Database, and replaces information according to the rules of the entry in the table. In my project, the extra information will be the *next-hop label*, which tells the node what outgoing label it should have. However, this leaves the complication of nodes knowing the correct next-hop label to put on outgoing packets. For this to work, there has to be some central processing in the network to coordinate the nodes. For this to happen, we must break the end-to-end principle (the idea that the hosts should be smart and the network should be a dumb, best-effort service) by using Software-Defined Networking.

C. Software-Defined Networking

Software-Defined Networking, at its core, is the idea of having a central controller that manages a network. Basically, the controller is connected to every node in the network and can issue rules to the nodes telling them how to behave.

The most commonly used SDN platform is OpenFlow. OpenFlow is a protocol that runs on the network nodes whose primary function is the operation of implementing and managing *flow tables*. These flow tables contain *flow entries*, which are just the rules governing the behavior of the node [3]. When an incoming packet reaches an OpenFlow node, the node checks packet information (e.g. MAC destination, IP source, ingress port, etc.) and looks for a match in the flow table, starting at the highest-priority flow entry and ending at the lowest-priority entry. A flow can only match with a single flow entry per flow table; in order to use more complex processing rule that involve multiple flows, a node must use multiple tables and have the flow entries redirect matching packets to the desired table.

Software-Defined Networking helps with the implementation of my project because it allows the controller to do the computation of determining MPLS labels, which in turn means that the next-hop labels can be calculated centrally. After being calculated the controller can send this information to the network nodes as needed so that they are able to handle traffic based on the needs of the packet.

D. Conclusion of Background Research

Based on the research of Smith and Thurlow, I found that I can use Dominant Set Multipath Routing to optimize network utilization and reduce link congestion, all while improving QoS of the network. Smith and Thurlow give a detailed explanation of how to find the best set of paths using DSMR, but do not explicitly give a way to implement routing in a DSMR network. To this end, they referenced Multiprotocol Label Switching (previously known as Tag Switching), which can enable a network to route traffic to a given destination via multiple possible routes, in constrast to destination-based forwarding that assumes a single best route. However, my implementation of MPLS requires the use of Software-Defined Networking to manage the network nodes to handle the MPLS interactions. OpenFlow is the SDN platform most widely used, so I am using it for my project; the specification proposed by McKeown et al. gives a general idea of how OpenFlow works. With all this information, I have a solid foundation for the implementation of DSMR networks.

II. Development Environment

In order to develop this project there has to be a development environment. The project is being developed in Python because it is fast to implement compared to other languages such as C and Java and because many of its built-in features are very handy. The environment must be able to represent and, later on, simulate networks. To this end, the project uses the NetworkX library to represent graphs of the simple virtual networks that will be used for testing. For simulation the project uses Mininet, a small-scale network simulation program created by Stanford University and freely available (see V). Lastly, the project is using the Ryu SDN controller because it is in Python and was easy to learn to use compared to other controllers (see VI).

III. Multipath Routing

The first, and arguably most critical, step in this project is to create an algorithm for DSMR. The requirements for this algorithm are that it must return some representation of a *set* of best paths in a network, as described in I-A.

A. Multi-Path Dijkstra's Algorithm

Since there is no need to write a routing algorithm from scratch, my multipath routing algorithm is a variation of the widely-used Dijkstra's algorithm for single path routing, and runs as follows to find paths from a given source to a given destination:

1) Create an empty list of paths. These paths are represented as tuples of (hopCount, bandwidth, nextHop).
2) Initialize a queue. Its entries are tuples of (hopCount, bandwidth, currentNode, path) where *path* is a tuple whose elements are the nodes along the path from the source to currentNode. Add a dummy entry of (hopCount=0, bandwidth=∞, currentNode=source, path=()).
3) While the queue is not empty, perform the following loop:
    a) Pop the next entry from the queue.
    b) Compare the values of this entry to the values of a known best path. If the hopCount of this entry is lower than the hopCount of the best path, or if the bandwidth of this entry is higher than the bandwidth of the best path,

or if there is no path yet, proceed. Otherwise, return to A.

c) Append currentNode to path. Check if currentNode == destination, and if so, add the entry tuple (hopCount, bandwidth, nextHop) to the list of paths, then return to a). Otherwise, proceed to d).

d) Check the neighbors of currentNode. If the path through currentNode to a neighbor is undominated, add that path to the queue. Return to a).

This differs from Dijkstra's algorithm in a few key ways. Firstly, it relies on more than one metric. Traditional single-path routing typically only checks hop count, whereas this algorithm checks bandwidth as well. Secondly, there is no *unseen set*. This is because even once a path to the destination has been found we continue to look through the queue since there may be multiple paths where no path dominates all the others, and stopping after finding one path would prevent us from finding more than one path. Furthermore, the unseen set prevents us from examining a node more than once, for good reason in single-path routing; however, multi-path routing could result in two paths that intersect or join at some node, which means the node would have to be examined more than once. Despite the removal of the unseen set, the algorithm still terminates even in cyclic graphs because of the domination property.

B. Algorithm Input and Output

The input to the multipath routing algorithm is a NetworkX graph and the name of a source node. Once the algorithm has been run, we have a set of best paths from the given source to *every* destination because the algorithm runs in a loop that calculates the best set of paths for every destination in the network.

The format for storing the destinations is a Python dictionary. The keys of the dictionary are sources. The values of the dictionary are dictionaries. These inner dictionaries' keys are destinations in the network, and their values are lists whose elements are tuples. These tuples are of the format `(hopCount, bandwidth, nextHop)`. They serve to represent the parameters of a path as well as the next hop that must be taken to follow that path.

C. Running the Algorithm

This function is in the file `multipath_dijkstra.py`. It can be imported by other files, such as `multipath_labelSwap.py` (see IV), or run directly via the command line. If it is run directly it takes an input command-line argument pointing to an edgelist file describing a graph and an optional second command line argument naming the source node. If no second argument is given the program will run the algorithm for all source nodes. If run directly, instead of returning the data it prints the to the standard output.

IV. Multiprotocol Label Switching

Destination-based forwarding is unsuitable for a multipath routed network because it does not support the ability to have multiple next hop options for a given destination. In order to be able to forward packets along specified paths, a new mechanism is needed. This project uses Multiprotocol Label Switching (MPLS) to achieve such forwarding.

MPLS works by adding extra data called *labels* to packet headers. Each node in the network can examine the labels in a packet header to determine what to do with the packet, rather than having to read its IP address and consult a long forwarding table. In this project the only MPLS functionality used is two labels, which are assigned in a two-pass iteration of the list of all paths in the network.

First, the program adds an arbitrary label to each path by appending it to the tuple. For the second pass, we consider the nextHop value of a given path and find the cost to reach that nextHop from the source of the path. Then, we iterate through every path from the nextHop to the destination of the path and compare the parameters to determine which one is the continuation of the path we are considering from the source. Once we know this, we use the label for the path from the nextHop to the destination as the *nextHopLabel* for the path from the source to the destination. Since this is complicated, here is an example:

Suppose source=A, nextHop=B, and destination=C. We want to create a nextHopLabel for a path from A to C. We find the cost to travel from A to B, which we will call `cost(A,B)`. Then, we look at every path from B to C; when `cost(A,B)+cost(B,C)=cost(A,C)` for a particular path, we know that this path from B to C is the continuation of the path from A to C. We now use the label of the path from B to C as the nextHopLabel for the path from A to C.

Through this mechanism we now have path tuples of the format (hopCount, bandwidth, nextHop, label, nextHopLabel). When a packet is received by a node in the network, the node reads the nextHopLabel on the incoming packet and finds which path has the value of the incoming nextHopLabel in its label field. It then sends the packet along the this path, where its label is the old nextHopLabel and its nextHopLabel is from the data of the path. Here is an example:

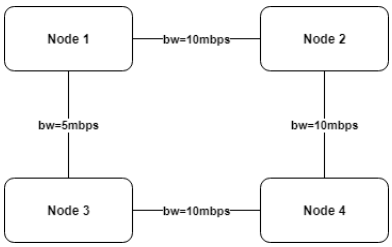

Fig. 4. An example topology. There will be two paths from 1 to 3 because of the differences in bandwidth and hop count.

Remeber that the tuple format is (hopCount, bandwidth, nextHop, label, nextHopLabel). Suppose the tuple for the path from Node2 to Node3 via Node4 is (2, 10, Node4, 4, 3) and that the tuple for the path from Node4 to Node3 is (1, 10, Node3, 3, 3). When a packet is sent from Node2 to Node3 along this path it has label=4 and nextHopLabel=3 when it arrives at Node4. Node4 sees the incoming packet's nextHopLabel=3, so it checks the labels of all paths for the one with label=3 and finds the tuple (1, 10, Node3, 3, 3). It then forwards the packet along this path with label=3 and nextHopLabel=3.

A. Running the Algorithm

The code for label-swapping is in `multipath_labelSwap.py` in the function `compute_MPLS_labels`, which takes an input NetworkX graph, calls `multipath_dijkstra` for every node in the graph, computes the MPLS labels, and returns the dictionary structure described above. If `multipath_labelSwap.py` is run directly it takes a command line argument that points to an edgelist file describing the input graph, and instead of returning the data it prints the labels to the standard output.

V. Mininet

With algorithms written for multipath routing and assigning MPLS labels, the next step was to implement a simulation using these algorithms. This project is using Mininet, a free program that "creates a real virtual network, running real kernel, switch and application code, on a single machine" [4]. However, what really makes Mininet the best tool for this project is that Mininet can run Open vSwitch, a virtual switch that supports the OpenFlow protocol [5]. This means that Mininet is sufficient for testing the project's algorithms.

In this project, a Python script called `square_topo.py` is used to define a network topology for Mininet to use. When Mininet is launched it is provided command line arguments to use the square topology.

VI. Ryu

This project is using the Ryu, "a component-based software defined networking framework" [6]. Ryu runs in Python so it is quick to implement, and it has a well-written tutorial. The basis for the controller in this project came from the simple learning switch example in the tutorial.

A. Ryu Handlers

Ryu is based around the idea of responding to network events through objects called "event handlers." Event handlers are bound to specific types of events, such as `EventOFPPacketIn`. The events handled in this project are `EventOFPSwitchFeatures`, `EventOFPPacketIn`, `EventOFPPortStatus`, and `EventOFPSwitchEnter`. OFP in this context stands for "OpenFlow Protocol" because the project is using OpenFlow for the SDN environment.

1) EventOFPSwitchEnter: As the name suggests, this event occurs when a switch enters the network and establishes communication with the controller. When this event occurs, the controller for this project updates an internal NetworkX graph describing the known topology of the network. This relies on Ryu's `topology` module.

2) EventOFPSwitchFeatures: After a switch connects to the controller, the controller sends a features request message to the switch automatically and the switch replies with a switch features message. When the switch features message reaches the controller, this event occurs. The controller for this project handles the switch features event by just adding a table-miss flow entry so that the switch knows to send packets that do not match any flows to the controller.

3) EventOFPPortStatus: This event occurs when a port on a switch is brought online, modified, or goes offline. The controller uses this event just for logging purposes and may be removed at a later date.

4) EventOFPPacketIn: This event occurs when the controller receives a packet from a switch because the packet did not match any flow table entries on the switch. The bulk of the work on the controller is done in this event handler.

First, the controller checks if the packet is an IPv6 multicast packet, and if so, drops the packet. Then, the controller adds the packet's source to its internal mac-to-port dictionary entry for the switch if it is not already in the dictionary, much like a traditional learning switch has but instead stored in the controller. Then the controller adds a flow to the switch to forward traffic to the source out of the packet's ingress port, much like a traditional learning switch behaves. Next, the controller adds a flow to the switch that prevents network loops from leading to infinitely-flooding packets, as described in VII. Then, the controller adds the source of the packet to its internal NetworkX graph if the source was not already in the graph. Lastly, the controller uses the information about the packet to determine what the switch should do with it and future packets with the same destination.

B. Layer 2 Routing in Ryu

Working with SDN gives us the ability to route traffic even at the link layer, which is something this project does. When a packet is sent to the controller, the EventOFPPacketIn handler routes the packet. This is a multi-step process:

1) Check if there is already a known path through this switch to the packet's destination. If so, send the packet along this path.
2) If there is not already a path, try computing a path. As of now the simulation is still stuck on single-path routing, so it uses NetworkX's built-in Dijkstra function to route. Later on the project will use the multipath variant of Dijkstra's algorithm described in III.
3) If no path is found, as may happen due to the topology not loading in correctly, then flood the packet and hope it can reach the destination via a link unknown to the controller.

If the packet is not being flooded, a flow entry is added to the switch to treat all future packets with the same destination in the same way.

VII. Preventing Network Loops

Multipath routing relies on the presence of multiple links from a given source to a given destination, which inherently means that the network must by cyclic. In order to prevent flooded packets from clogging the network forever, traditional networks use Spanning Tree Protocol (STP) to block certain ports on certain switches to cause the network to become a tree rather than a cyclical network. The same issue must be overcome in this project but must also *not* block ports because we still want to have multiple paths.

This project accomplishes this challenge by creating a source-based tree at each switch for each source. When a switch forwards a packet to the controller to learn what to do with it, the controller adds two flows to the switch. The first flow has a high priority and tells the switch that if it receives a multicast or broadcast packet from this source on this same ingress port to flood the packet. The second flow has a low priority and tells the switch that if it receives a multicast or broadcast packet from this source to drop the packet. This pair of flows has the effect that the switch will only flood multicast and broadcast packets from this source if the packet was received on the ingress port the source was first seen from. Since these flows are installed on every switch for every source as traffic comes in, effectively each source is the head of a tree such that multicast and broadcast traffic from the source only flows down the tree, preventing network loops from resulting in network failure.

VIII. Current Issues

The biggest roadblock in the project as of now is that Ryu's topology module has issues discovering the entire topology of the network as it comes online. Specifically, it non-deterministically fails to learn of the links between the switches. It is unknown if this is due to a problem with Ryu's source code, a problem with Mininet, or a problem with some interaction between the two or between Ryu and NetworkX. It is also possible that this is the result of running the project in a virtual machine.

The reason this is problematic is because without the links between the switches being known, it becomes impossible to route traffic and the network relies on flooding, which is a huge waste of network resources and is much less efficient. Furthemore, because of how the code is written, flooding like this means that every packet must be sent to the controller, which must then go through all the steps on the PacketIn handler to determine that the best course of action is to flood, which means the network performs much worse when just flooding packets.

IX. Moving Forward

The next step in this project is to begin using the multi-path routing algorithm in the Ryu controller and Mininet environment. Once that is working, the following step is to switch from the prototype's current destination-based forwarding to the goal of label-switched forwarding using MPLS. This entails learning how Ryu interfaces with OpenFlow v1.3's MPLS stack functionality. Assuming all works as intended, that will fulfill the project's original goals. From there it can stop or it can continue on with expanded functionality, depending on availability and motivation of the author.

Acknowledgements

The author would like to thank Professor Brad Smith of the University of California at Santa Cruz for providing the basis of the project, guiding the project, helping overcome problems with implementation, and securing funding to pay for hours worked on the project.

The author would also like to thank Shivani Vaidya, a peer at UC Santa Cruz, for her help with learning how to use Ryu.

Furthermore, the author would like to thank Professor Katia Obraczka of UC Santa Cruz for introducing the author to computer networking and SDN, inspiring interest in research on the topic.

Lastly, the author would like to thank Avirudh Kaushik, a graduate student at UC Santa Cruz, for working with Professor Smith on figuring out how to use MPLS in Ryu, which will certainly help down the road in the project.

References

[1] B. R. Smith and L. Thurlow. Practical multipath load balancing with qos. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 937–943, Jan 2013.

[2] Y. Rekhter, B. Davie, D. Katz, E. Rosen, and G. Swallow. Cisco systems' tag switching architecture overview, 1997.

[3] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[4] Mininet Team. Mininet: An instant virtual network on your laptop (or other pc). http://mininet.org, 2018.

[5] Sue Marek, Linda Hardesty, Dan Meyer, Jessica Lyons Hardcastle, and Ali Longwell. What is open vswitch (ovs)? https://www.sdxcentral.com/cloud/open-source/definitions/what-is-open-vswitch/, 2018.

[6] Ryu SDN Framework Community. Ryu sdn framework. https://osrg.github.io/ryu/, 2017.