How would you like your packets delivered? An SDN-enabled open platform for QoS routing

Chenfei Gao*, Vahid Rajabian-Schwar†, Weiyi Zhang*, Guoliang Xue‡, Jian Tang†,
* AT&T Labs Research, Bedminster, NJ, USA
† Syracuse University, Syracuse, NY, USA
‡ Arizona State University, Tempe, AZ, USA

***Abstract*—Traditional Internet routing is simple, scalable and robust, but cannot provide perfect QoS support due to the current completely distributed hop-by-hop routing architecture. Software defined networking (SDN) opens up the door to traffic engineering innovation and makes possible QoS routing with a broader picture of overall network resources. We further argue that SDN can provide more opportunity for the network users to make their own routing selections with network programmability. In this paper, we propose OpenMCR, a general framework for network users to make their own choice of routing given various requirements. OpenMCR provides routing subject to several additive QoS constraints, which is NP-hard when the number of constraints is two or more. By composing various necessary conditions with different path extension schemes, our platform can customize routing solutions for each network user based on their own requirements. Through experiments in an SDN emulated environment, we evaluate multiple aspects of OpenMCR, demonstrate its effectiveness compared with several baselines and validate our theoretical analysis.**

***Index Terms*—Traffic engineering, Software defined WAN, Multi-constrained QoS routing, Optimization framework, Efficient algorithms**

I. Introduction

The Internet design is based on end-to-end arguments [20] where network support is minimized and end hosts are responsible for most communication tasks. Such a design allows a unified best-effort service at the network layer for any type of data. Therefore, link-state routing protocols (e.g., OSPF and IS-IS) are widely used because they are scalable, robust, and based on simple abstractions. This type of architecture fits perfectly to data transmission where the primary requirement is reliability. Unfortunately, traditional routing schemes are relatively inflexible, since they direct all traffic over shortest paths. For many applications, which have stringent QoS requirements, cannot be guaranteed in the best-effort Internet. To provide Quality of Service (QoS), IETF has explored several QoS architectures, but none has been truly successful and globally implemented. This is because QoS architectures such as IntServ [3] and Diffserv [2] are built on top of the current Internet's completely distributed hop-by-hop routing architecture, lacking a broader picture of overall network resources. Even though MPLS [19] provides a partial solution via its ultra-fast switching capability, it lacks real-time reconfigurability and adaptivity. Network operators need flexible intra-domain routing to perform fine-grained Traffic Engineering (TE), provision backup paths, and steer traffic through middle-boxes. To do so, they cannot rely on distributed routing protocols which mandate all traffic to flow along the shortest paths. Instead, they typically use dedicated TE mechanisms, prominently MPLS RSVP-TE [1], [6]. Unfortunately, these mechanisms come with their own set of limitations [16]. Among others, the lack of coordination between routers can lead to a long convergence time. They also introduce control-plane and data-plane over-head due to signaling and encapsulation, respectively.

Software Defined Networking (SDN) offers fine-grained control over routing, at the expense of controller overhead, failover latency, and deployment challenges. SDN brings much more flexibility to network routing, along with improved performance and agility. With SDN, network operators can design routing applications to shape traffic from a central controller that reconfigures network switches in real-time. This could include prioritizing or even blocking different packets when necessary and allowing administrators to base routing decisions on external systems and events, such as a maintenance window. It would be much more efficient than current static configurations and would let network operators easily reroute network traffic to make networks run optimally.

Benefits for Enterprise Customers: Most SDN studies/implementations focus on benefits for the network providers, such as Opex (better network design) and Capex (cheaper commodity hardware) savings. At its core, most people view SDN as a tool that enables network providers to reconfigure networks on the fly to adapt to different subscriber usage patterns or application needs. SDN puts control into the hands of network engineers and administrators by letting them respond to changing business requirements in real-time. But for the *enterprise customers* of network providers, *what are the benefits from SDN?* Little research has been done to investigate this question. *In this paper, we argue that, to the network users, some level of control of on-the-fly changes can be provided.* Essentially, in a SDN network, control is directly programmable and gives administrators more functionality than ever before. Programmability offers the addition or expansion of features, as well as the ability to change flows dynamically and even pass management up to higher-level orchestration tools. A great example is QoS control, which allows unique flows to be programmed for different data types. Management of the network flows can be designed on a case-by-case basis, while

This research was supported by NSF under grants #1421685, #1704092 and #1443966.

still running on the same physical topology. Separate customers (internal or external) can be defined with separate routing based on need, budget or otherwise. *In this work, we propose a general routing platform that takes advantage of SDN technology and enables users to find customized routing strategies based on their preferred time/budget/reliability requirements. Users can select one or more QoS criteria, such as delay, jitter, cost, etc., and their own bound for these constraints. Based on the user QoS inputs, OpenMCR will compose a customized routing solution for each routing request.* To the best of our knowledge, this is one of the first studies that investigates how to use SDN to benefit network users, not just network operators. We provide a customized platform, OpenMCR, for multiconstrained QoS routing, which gives users the flexibility to choose their own routing based on time/budget/precision and other concerns.

II. Multi-constrained QoS routing

In this work, we consider the case that Enterprise customers need network connections with dynamic requests and on the fly solutions. For example, each customer must have various QoS requirements, such as budget, connection delay, jitter, and others. How to provide satisfiable network connection to each customer considering multiple QoS requirements at the same time? We formulate such practice as the following problem:

Definition 1 (Multi-Constrained Path (MCP) problem)**.** Consider a network $G(V, E)$. Each link $(u,v) \in E$ is associated with K additive weights $\omega_i(u,v) \geq 0$, $(i = 1,\ldots,K)$. Given K constraints W_i, $(i = 1,\ldots,K)$, the problem is to find a path $\mathcal{P}$ from source node s to destination node d such that:

$$D_i(\mathcal{P}) = \sum_{(u,v)\in\mathcal{P}} \omega_i(u,v) \leq W_i \quad i = 1, 2, \ldots, K. \qquad (2.1)$$

A path which satisfies the above condition is a *feasible* path. MCP is known to be NP-hard. There are extensive studies on finding approximated solutions [15], [22], [23]. While each solution has its own merit, the realization of these solutions are not practically feasible with the traditional distributed Internet architecture. More importantly, none of the previous work can provide a universal solution for various MCR requests. For example, an Enterprise customer could need one connection which is more delay tolerant with cheaper cost, but at the same, may need another connection for a delay and jitter sensitive service. How to provide solutions that adapt to dynamic customer requests? Most previous research could not answer this question. In this work, we propose a general framework that can be tailored to all types of customers. Moreover, our platform can actually be compatible with most existing approaches.

III. OpenMCR Architecture

The architecture of OpenMCR is illustrated in Fig.1. The reasons we call it open are (1) SDN applications are built at northbound interfaces (e.g. GUI) which allow customers to dynamically specify routing requests (instead of static SLAs). (2) Customers have the ability to use MCP specifications (SDN application) to customize routing requests, with specific constraints, objective and other service level agreements (SLAs)

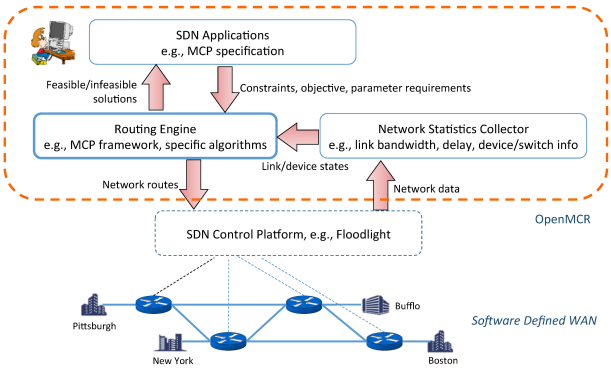


Fig. 1: OpenMCR architecture

(e.g. optimality, maximum running time, specified routing algorithm) in a dynamic way. (3) OpenFlow is the underlying protocol to manipulate forwarding behavior at each switch for every traffic flow.

The inputs are parsed and handed over to the routing engine, the key component of OpenMCR. The role of the routing engine is to compute a feasible/optimal path for each request based upon all routing requirements. The result, whether or not a feasible path is found, is fed back to the SDN application to notify the client. For each request, if a feasible path is found, the feasible path is translated to low level OpenFlow entries that are installed at each corresponding switch along the path in the network by the SDN control platform (e.g. Floodlight , OpenDaylight). Meanwhile, a network statistics collector provides real time network status information (e.g. available link bandwidth, link delay, jitter) to the routing engine for real time decision making. A closed loop is formed in OpenMCR, involving the routing engine, SDN control platform and network statistics collector. In the following section, we discuss the design of OpenMCR in detail with focus on the routing engine. In Sec.V, we unveil an implementation of OpenMCR and expand on the details of the MCP specification, OpenFlow compiler and network statistics collector. We present a performance evaluation of OpenMCR in Sec.VI, and follow it with a discussion on related work in Sec.VII and a conclusion in Sec.VIII.

IV. OpenMCR Detailed Design

In this section, we discuss the details of OpenMCR, particularly the routing engine, which is the key component of OpenMCR. There are two major factors which can affect the routing engine design: the time to find a solution, and the quality of the solution. The challenge here is how to dynamically find a solution to the customized MCP based upon real time network state adapting to the deciding factors.The theory behind the OpenMCR is:

- Use effective and efficient *necessary condition checks*, which will be described in Section IV-A, to reduce the solution space, and consequently improve the efficiency. In fact, most of the previous work, such as [13], [22], can be categorized as sort of necessary condition check. Based on the requirement for the running time or solution quality, various algorithms can be applied. So the platform

is generic in the ways that it can accommodate various algorithms regarding to dynamic requirements.

- Then pick a good candidate from the solution space which is reduced in the previous step, which will be described in Section IV-C. Many related work, such as [14], can be used as the algorithm to find a better candidate. Given the reduced search space, these solutions, though could be complicated by its nature, can be efficient for this purpose.

A. Necessary Condition Check

Assuming that we are looking for a feasible path from source node s to destination node d, and we already have a subpath from s to node u, denoted by $P_{s\to u}$, and now we need to choose the next hop. For any neighbor node A of u, there is a path $s \to A$: $P_{s\to u} \cup (u, A)$ with path weight c_i w.r.t. link weight ω_i. Now if from node A, we can find a path p to d which satisfies:

$$D_i(p) = \sum_{e\in p} \omega_i(e) \le W_i - c_i, i = 1, 2, \ldots, K(*)$$

then we know there is a feasible path through node A. On the other hand, if any constraint i in (*) is violated for *all* paths from A, then we can claim that no feasible path through node A exists given the chosen path $P_{s\to u}$. We say A is a *unqualified node*, and *exclude* it from the search space of the solution, referred as *solution space* in the rest of the chapter.

In the rest of the paper, we will use u to represent the current node, A the being checked neighbor node, and $W_i'(= W_i - c_i)$ to represent the modified i^{th} weight constraint.

Definition 2 (necessary condition)**.** A necessary condition is one or a set of testing conditions which can be used to decide the qualification of a node A. It must have the property that *if a node A does not pass the necessary condition check, we can claim that A must be an unqualified node*.

In the following, we will introduce few necessary conditions, and then show how to evaluate their effectiveness.

1) Variant Path ∞-norm Function: Holder's q-vector norm [7], which is listed in below,

$$D_q(p) = (\sum_{i=1}^{K} [\frac{D_i(p)}{W_i}]^q)^{\frac{1}{q}}$$

when $q \to \infty$, the q-norm (called ∞-norm) function will be satisfied if and only if there is a feasible path for the MCP problem from A to d.

$$\frac{D_i(p)}{W_i} \le 1, \quad i = 1, 2, \ldots, K$$

However, we do not have any polynomial time solution for checking the ∞-norm function. Therefore, we use a variant of ∞-norm function to be the necessary condition. Define $D(sp_i) = \sum_{e\in sp_i} \omega_i(e)$, where sp_i is a shortest path from A to d w.r.t. link weight ω_i. We want to check the following conditions:

$$\frac{D(sp_i)}{W_i'} \le 1, \quad i = 1, 2, \ldots, K \tag{4.1}$$

Corollary 1. If node A cannot satisfy some condition in (4.1), then A must be excluded from the solution space. [1] □

[1]Due to page limit, please refer to our technical report [25] for the proofs of all the Corollaries.

2) Edge 1-norm Function: Similar to the *path* q-norm function, we define the *edge q-norm* as:

$$(\sum_{i}^{K} (\frac{\omega_i(e)}{W_i})^q)^{\frac{1}{q}}$$

and use it as the new edge weight metric, we have $\omega_{nln}(e) = \sum_{i=1}^{K} \frac{\omega_i(e)}{W_i'}$. Given p^{opt} is the shortest path w.r.t. edge vector ω_{nln}, the necessary condition is

$$D_{nln}(p^{opt}) = \sum_{e\in p^{opt}} \omega_{nln}(e) = \sum_{e\in p^{opt}} \sum_{i=1}^{K} \frac{\omega_i(e)}{W_i'} \le K \tag{4.2}$$

Corollary 2. If A does not satisfy the necessary condition (4.2), A must be be excluded from the solution space. □

3) Weighted Edge 1-norm Function: We propose another new edge weight metric as $\omega_{wln}(e) = \sum_{i=1}^{K} \omega_i(e)$. Assume p^{opt} is the shortest path w.r.t. edge vector ω_{wln}, the necessary condition is

$$D_{wln}(p^{opt}) = \sum_{e\in p^{opt}} \omega_{wln}(e) \le \sum_{i=1}^{K} W_i' \tag{4.3}$$

Corollary 3. If at node A, the condition (4.3) cannot be satisfied, A must be excluded from the solution space. □

4) Edge ∞-norm Function: We give another modified edge weight function

$$\omega_{max}(e) = \max\{\frac{\omega_i(e)}{W_i'}, i = 1, 2, \ldots, K\}$$

Corollary 4. If for the shortest path p^{opt} from A to d, w.r.t. edge weight $\omega_{max}(e)$, $D_{max}(p^{opt}) = \sum_{e\in p^{opt}} \omega_{max}(e) > K$, then we can claim that A cannot be in the solution space. □

5) Guarantee Algorithm Check: Polynomial time approximations (PTAS) for MCP have been proposed in [5], [15], [22]. These PTAS share the following property:
Given a small real number ϵ, and weight constraints W_1,W_2, $\ldots, W_K$, PTAS can guarantee to find a feasible path p if there exists a path q such that:

$$D_1(q) \le W_1,\ D_i(q) \le (1-\epsilon)\cdot W_i, \quad i = 2, \ldots, K.$$

We call these algorithms *guarantee algorithms*, and have the following necessary checking corollary.

Corollary 5. With modified weight constraints$\{W_1, \frac{1}{1-\epsilon}\cdot W_i', i = 2, \ldots, K\}$, if a guarantee algorithm cannot find a feasible solution from A to d, then we claim that A must be excluded from solution space. □

B. The Comparison of Necessary Conditions

Previous section only presented few necessary conditions. Many more necessary conditions can be applied for MCP, and it is not practical to implement and use all of them at once. Our design assumes that time and quality are two deciding factor for selecting right necessary conditions. So it is desirable to use most efficient and effective ones. The question then is how to judge which necessary conditions are more effective or efficient than others? In this section, we develop schemes to evaluate the necessary conditions in order to provide more effective and efficient necessary condition checks.

1) **Exact Comparison Between Necessary Conditions:**

Definition 3 (domination relation)**.** For any two necessary conditions, R_1 and R_2, we say R_1 is dominated by R_2 if either of the following cases is satisfied:

1) R_2 finds the same unqualified nodes set as $R1$, but has less running time than R_1 has.
2) The set of unqualified nodes found by $R1$ is a subset of the set of unqualified nodes found by $R2$.

Case 1 is defined from the implementation perspective, while case 2 is defined from algorithmic perspective. Based on the definition of domination relation, we can easily claim that [22] dominates [5] and [23] because it has faster running time than the other two. In the rest of this section, we want to provide more evaluations for the presented necessary conditions.

Theorem 1. Edge ∞-norm function is dominated by edge 1-norm function.

PROOF: We use p_M to denote the shortest path w.r.t the edge weight ω_{max}, p_S to denote the shortest path w.r.t. ω_{nln}. If *weighted ∞-norm function* can exclude some node A, we have $D_{max}(p_M) > K$. Moreover, because p_M is the shortest path with ω_{max}, we know that $D_{max}(p_S) \geq D_{max}(p_M) > K$.

Meanwhile, we have $D_{n1n}(p_S) = \sum_{e \in p_S} \sum_{i=1}^{K} \frac{\omega_i(e)}{W_i'}$. Resulted from the fact that

$$\sum_{i=1}^{K} \frac{\omega_i(e)}{W_i'} > \max\{\frac{\omega_i(e)}{W_i'}, \quad \forall i \in \{1, 2, \ldots, K\}\} = \omega_{max}(e),$$

we can find that

$$D_{n1n}(p_S) = \sum_{e \in p_S} \sum_{i=1}^{K} \frac{\omega_i(e)}{W_i'} > \sum_{e \in p_S} \omega_{max}(e) = D_{max}(p_S) > K$$

On the other hand, we cannot find such proof for the reverse direction. Therefore, *weighted ∞-norm function* is dominated by 1*-norm function*. □

It worth noting that edge 1-norm function always dominates edge q-norm function ($q \geq 1$). Thus, we will **pick *edge 1-norm function* over the *edge* q*-norm function* for the necessary condition check**.

2) **The Expectation of Necessary Condition:** Some necessary conditions do not have an exact domination relation between each other. For example, the edge 1-norm function and the weighted edge 1-norm function. When all $W_i, i = 1, 2, \ldots, K$ are same, these two functions are exactly same. In this section, we study the case that $W_i, i = 1, 2, \ldots, K$ are not all same.

For those conditions, we introduce the following definition and propose another evaluation scheme.

Definition 4 (expectation of necessary condition)**.** The size of the set of total possible paths which can be excluded by a necessary condition is the expectation of a necessary condition.

Let us use Fig. 2 for explanation. When K=2, for any path p that would be excluded by the 1-norm function condition, it has $\sum_{e \in p}(\frac{\omega_1(e)}{W_1} + \frac{\omega_2(e)}{W_2}) > K$. If we use x to represent $\sum_{e \in p} \frac{\omega_1(e)}{W_1}$, and y to represent $\sum_{e \in p} \frac{\omega_2(e)}{W_2}$. Then the 1-norm function condition is presented as $\frac{x}{W_1} + \frac{y}{W_2} \leq 2$. In Fig. 2(a), the line represents $\frac{x}{W_1} + \frac{y}{W_2} = 2$. For any path p that does not pass the 1-norm check, it corresponds to a point which is above the line ($D_{n1n}(p) > 2$). All the paths that are excluded by 1-norm function compose the blue shaded region above the line, which is called *edge 1-norm excluded space*. While the area under the line is the *edge 1-norm solution space*.

Similarly, the weighted 1-norm function condition corresponds to $x + y \leq W_1 + W_2$ in Fig. 2(b). From the figures, it is worth noting that 1-norm function and weighted 1-norm function are not dominated by each other because the two corresponding excluded regions are overlapped. Then if and how we can pick between these necessary checks?

Theorem 2. *For $K \geq 2$, the expectation of 1-norm function is better than the expectation of weighted 1-norm function.*

PROOF: To evaluate the expectations of edge 1-norm function and weighted edge 1-norm function, we need to compare the sizes of *1-norm excluded space* and *weighted 1-norm excluded space*. The larger size of the region represents the better expectation of the condition.

We use R_o and R_w to denote the solution space of edge 1-norm function and edge weighted 1-norm function, respectively.

In Figs. 2(c) and 2(d), when K=3, we use x to denote$\sum_{e \in p} \omega_1(e)$, y to represent $\sum_{e \in p} \omega_2(e)$, and z to denote $\sum_{e \in p} \omega_3(e)$. Then for *1-norm function*, its solution space, R_o, is represented by:

$$\frac{x}{W_1} + \frac{y}{W_2} + \frac{z}{W_3} \leq 3, x \geq 0, y \geq 0, z \geq 0$$

which is the area under the plane in Fig. 2(c). Similarly, the solution space for weighted 1-norm function, R_w, is:

$$x + y + x \leq W_1 + W_2 + W_3, x \geq 0, y \geq 0, z \geq 0$$

which is the space under the plane in Fig. 2(d).

$V(R_o)$ (the volume of R_o) = $\frac{1}{6} \cdot (3W_1) \cdot (3W_2) \cdot (3W_3)$. $V(R_w)$ (the volume of R_w) = $\frac{1}{6} \cdot (W_1 + W_2 + W_3)^3$. It is evident that R_o has smaller solution space, and hence has a better expectation when K=3.

When $K > 3$, in a K-dimensional space, the solution space of 1-norm function can be formulated as:

$$\frac{X_1}{W_1} + \ldots + \frac{X_K}{W_K} \leq K, X_i \geq 0, i = 1, 2, \ldots, K$$

corresponding to the space between the origin and a *(K-1)-simplex*. Its volume, which is the size of the solution space, is $\frac{1}{K!} \cdot \begin{pmatrix} A^{11} & A^{11} & \ldots & A^{1K} \\ A^{21} & A^{22} & \ldots & A^{2K} \\ \vdots & \vdots & \ddots & \vdots \\ A^{K1} & A^{K2} & \ldots & A^{KK} \end{pmatrix}$

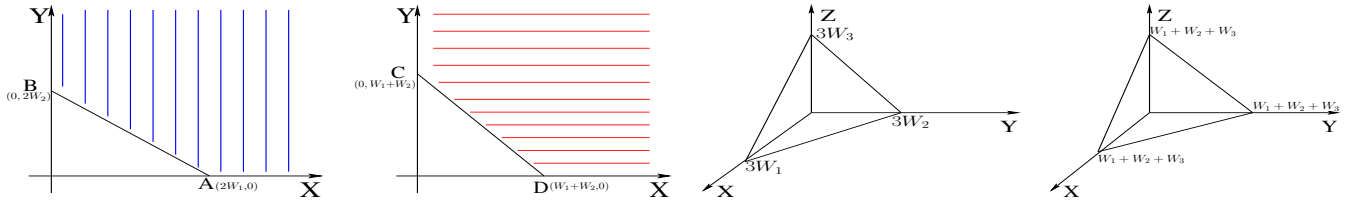
Because it is a diagonal matrix, the value is $\frac{1}{K!} \cdot (K \cdot W_1) \cdots (K \cdot W_K)$.

Similarly, in K-dimensional space, the solution space of weighted edge 1-norm function also composes of the origin and a *(K-1)-simplex*,

$$X_1 + \ldots + X_K \leq W_1 + \ldots + W_K, X_i \geq 0, i = 1, 2, \ldots, K$$

The size of the solution space is $\frac{1}{K!} \cdot (W_1 + W_2 + \ldots + W_K)^K$.

Because $(W_1 \cdot W_2 \cdot \ldots \cdot W_K)^{\frac{1}{K}} < \frac{W_1 + W_2 + \ldots + W_K}{K}$ (when W_i are not all same), we can claim that the solution space of 1-norm is



(a) 1-norm function (*K=2*) (b) Weighted 1-norm (*K=2*) (c) 1-norm function (*K=3*) (d) Weighted 1-norm (*K=3*)

Fig. 2: Expectation of necessary condition

smaller. Therefore, we find that for $K \geq 2$, the expectation of edge 1-norm function is better than the expectation of weighted edge 1-norm function. And we should expect better overall performance from using edge 1-norm as the necessary condition check. □

3) The Comparison between FAST and Edge 1-norm Function: According to our guaranteed algorithm check, we can find that the solution space of FAST algorithm [22] is

$$X_1 \leq W_1, X_i \leq \frac{1}{1-\epsilon} \cdot W_i, i = 2, 3, \ldots, K$$

So its solution space volume is $(\frac{1}{1-\epsilon})^{K-1} \cdot \prod_{i=1}^{K} W_i$. Compared to the solution space volume of edge 1-norm function $\frac{K^K}{K!} \prod_{i=1}^{K} W_i$, if $(\frac{1}{1-\epsilon})^{K-1} < \frac{K^K}{K!}$, then FAST algorithm can achieve better expectation. According to the Stirling's Approximation, we have $K! = K^K e^{-K} \sqrt{2\pi K}$, then we find that generally even with $\epsilon = 0.6$, FAST can have better expectation.

C. Path Extension Scheme

Necessary condition checks help to vastly reduce the solution space, but we still have to pick the next hop to extend the path towards to the destination. In fact, most of the MCP algorithms can be used for evaluating and selecting next hops. In this section, we introduce three different path extension schemes to meet various routing requirements. For each scheme, OpenMCR stores multiple paths in a heap. The number of paths stored in heap is limited, denoted as Q, which is set it to 100 in our experiments. During the process, we maintain the heap by removing the path with the least number of hops if the limit Q is reached.

At each iteration, path extension schemes check each next hop candidate stored in heap with sufficient condition checks, and then determine one with the most potential as the next hop. Next, we discuss each of the schemes in more details.

• **Random Pick** We treat all of the possible next hop candidates equally and randomly pick up one of them. Thus, the sufficient condition here is to assign equal potential to every next hop candidate.

• **Further Look**

In this scheme, for each candidate node A in the solution space, we further check its own solution space. In other words, assuming A is chosen, we check how many qualified next hop nodes A has. Our surmise is that the more qualified next hops A has, the more likely it can provide a path to the destination.

Therefore, in this scheme, we select the candidate node which has the most qualified children in the heap to be the next hop.

• **Greedy Optimal** We propose a new metric *path length* here to evaluate the quality of path. Let p be any path in G, we define path length of p, denoted by $l(p)$, as:

$$l(p) = \max_{1 \leq k \leq K} \frac{\omega_k(p)}{W_k} \tag{4.4}$$

Naturally, path length reflects how close the path is to each additive constraint bound. Thus, considering all additive constraints, we can compare and determine that, one path is better than another if it has a lower path length. Naturally, lower path length presents more flexibility for the path to extend. Hence, we pick the candidate node which has the lowest path length to be the next hop.

Algorithm 1: General Framework for MCP

Input : G, s, d, Alg, Obj, t, ϵ, $\mathbf{W} =< W_i >$, $\mathbf{\Omega} =< \omega_i >$, $\mathbf{B} =< b_j >$
Output: $p_{s \to d}$

```
<necessary condition, sufficient condition, path extension scheme, algorithm>:= SLA Analyzer (G, Alg, Obj, t, ε);
if algorithm ≠ null then
    p_{s→d} := algorithm (G, s, d, Obj, ε, W, Ω, B);
    return p_{s→d};
if necessary condition = null then
    return SLA may not be satisfied in the worst case;
Q := p_{s→s};
while π[d] = null and Q ≠ ∅ do
    for each path p_{s→q} ∈ Q do
        if any b_j ∈ B cannot be satisfied on p_{s→q} then
            Q := Q/p_{s→q};
            Continue;
        if q = d then
            return p_{s→q};
        Let H_q be the set of all adjacent nodes to q except π[q];
        for each node u ∈ H_q do
            Check qualification of u by necessary condition;
            if u is unqualified or u is marked grey then
                H_q := H_q/u;
            else
                Check the potential of u by sufficient condition;
        if H_q = ∅ then
            Remove p_{s→q} from Q;
    Extend a path p_{s→q} ∈ Q to p_{s→q→u} according to selected path extension scheme;
    Mark u grey; Q := Q ∪ p_{s→q→u}; π[u] := q;
return No feasible path found;
```

D. MCP Framework

One key observation is that all existing solutions to the MCP problem either use specific necessary condition check or provide a sufficient condition scheme or both. In fact, if we look from a broader view, *most of the previous solutions can be applied either as the necessary condition check or as the path extension scheme or both.* In that sense, we want to connect the previous seemingly unrelated solutions, combine them and present as a general framework for the MCP problem.

Algorithm 2: SLA Analyzer

Input : G, Alg, Obj, t, ϵ
Output: <*necessary condition*, *sufficient condition*, *path extension scheme*, *algorithm*>

1 **if** $Obj \neq null$ **then**
2 <*sufficient condition*, *path extension scheme*>:= Greedy Optimal;
3 **if** $\epsilon \neq null$ **then**
4 <*necessary condition*>:= FAST;
5 **else**
6 Select one <*necessary condition*> from VarPath∞-norm > Edge1-Norm while ensuring framework worst case running time $T \leq t$;
7 **if** *no* necessary condition is selected **then**
8 **return** <*null*, *null*, *null*, *null*>;
9 **else**
10 **if** $\epsilon \neq null$ **then**
11 <*necessary condition*>:= FAST;
12 Select one <*sufficient condition*, *path extension scheme*> from FurtherLook > GreedyOptimal > RandomPick while ensuring $T \leq t$;
13 **if** *no* sufficient condition *is selected* **then**
14 **return** <*null*, *null*, *null*, *null*>;
15 **else**
16 Select one <*necessary condition*> from VarPath∞-norm > Edge1-Norm and one <*sufficient condition*, *path extension scheme*> from FurtherLook > GreedyOptimal > RandomPick while ensuring $T \leq t$;
17 **if** *no* sufficient condition *is selected* **then**
18 **return** <*null*, *null*, *null*, *null*>;
19 **return** <necessary condition, sufficient condition, path extension scheme, *null*>;

Users can pick and choose different algorithms and compose their own solution adapting to various demands with different running time or quality. For example, if users need a fast solution, [13] and [8] can be combined as a solution. If customers want to spend more time to improve the possibility of finding a solution, they can select [22], [23] and [14] to form a solution. By choosing different necessary condition checks and path extension schemes, we can implement different solutions to specified MCP while satisfying SLA requirements. The engineering problem for building such a platform is *how to select the appropriate necessary condition and path extension scheme to best serve each request*. A specific set of SLA requirements along with MCP constraints and an objective function is mapped to each request. Thus, we need an SLA analyzer to process such information and help decision making in terms of the encoded logic. The output of SLA analyzer is a tuple of selected necessary condition, sufficient condition, path extension scheme, and possibly specific algorithm. Note that sufficient condition is mapped to path extension scheme. The procedure of our framework for MCP is listed in Algorithm 1. It invokes a subroutine of SLA analyzer which is listed in Algorithm 2.

It is worth noting that though we use encoded logic in the algorithm, users of OpenMCR are not restricted to it and can customize decision making logic for SLA Analyzer.

Given a necessary condition and path extension scheme, OpenMCR selects the next hop with the most potential in iterations to extend paths currently stored in a heap until a destination node has been reached (line 8-25 of Alg.1). Then the path will be returned as a solution (line 13-14 of Alg.1).

V. OpenMCR Implementation

We implement a proof-of-concept version of OpenMCR on a linux machine using Mininet and Floodlight. All OpenMCR related functionalities are implemented in Python. Detailed architecture is shown in Fig.3.

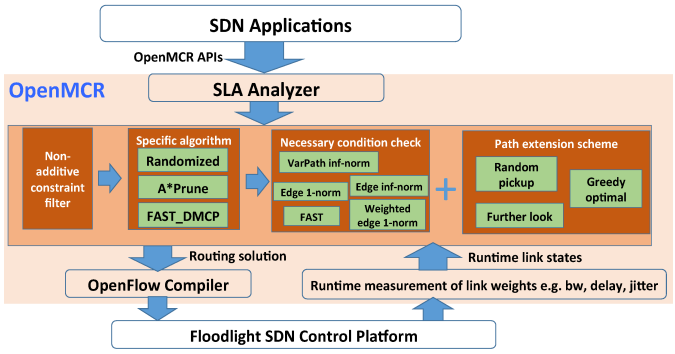


Fig. 3: OpenMCR Implementation

We developed a program to establish the network topology in Mininet, collect runtime state of links periodically (e.g. every 5 seconds), install flow entries along the path and generate real traffic flows according to the flow requirements using multiple threads. Additionally, we developed an optimizer program to implement the routing engine of OpenMCR. Our routing engine consists of 3 classic algorithms, 5 necessary conditions, 3 path extension schemes and the logic to select the combination of necessary condition and path extension scheme described in Algorithm 2.

We start 3 separate processes for Floodlight, Mininet program and optimizer program respectively. At the northbound interface, MCP requirements can be specified by invoking our OpenMCR APIs. A snippet of such APIs is shown in Fig.4.

```
MCPOptimizer.setTopology(topology)
MCPOptimizer.addFlow(source, destination)
MCPOptimizer.setObjective(objective)
MCPOptimizer.setAlgorithm(algorithm)
MCPOptimizer.addConstraint(LinkWeight, value, type)
MCPOptimizer.setEpsilon(value)
MCPOptimizer.setRuntimeSLA(value)
MCPOptimizer.setLinkState(link[], value[], type)
Path = MCPOptimizer.solve()
```

Fig. 4: A snippet of OpenMCR north-bound APIs

Runtime link statistics are measured and collected every 5 seconds by invoking another set of Floodlight REST APIs (i.e. topology, switch-stats, device, port stats) and sending probes (e.g. ping, iperf). Such runtime states are sent to the optimizer by invoking its API (i.e. setLinkState). Any found routing solution is sent to an OpenFlow compiler that maintains switch port to topology link mappings and IP addresses of each node. So the path solution can be efficiently translated into flow entries in each switch along the path. Floodlight REST APIs of static flow pusher are invoked for entry installation. After successful installation of flow entries, the optimizer sends a signal to the traffic generator we developed (using the Linux iperf tool for Mininet). Real traffic flows will be generated according to flow requirements (e.g. bandwidth, lifetime, source and destination).

VI. Evaluation

In this section, we evaluate the effectiveness of OpenMCR with multiple experiments. Each of them focuses on one property of OpenMCR. Table I below lists all the experiments conducted and summarizes their purpose.

Expr.	Description
1, 2	illustrate the dynamic routing feature
3,4	compare various necessary conditions
5,6	validate the satisfaction of user requirements
7,8	demonstrate the flexibility of framework
9	show the compatibility with arbitrary topology

TABLE I: Summary of experiments

We implement the routing engine of OpenMCR using NetworkX [27] and emulate OpenFlow based traffic transmission and network environment in Mininet [26], an SDN flavored network emulator widely used in SDN related research. All the experiments are run on a 2.6GHz Intel Core i7 Linux machine with 16GB of memory.

A. Environment setup

In this section we discuss the link capacity and delay settings used across our experiments in Mininet. We conduct most of experiments (i.e. expr $3-8$) using the public CORONET CONUS topology [24]. Since link delay in WAN transmission is dominated by propagation delay, we convert the physical distance to propagation delay and emulate it as the link delay in Mininet. We assign each link a capacity in the range of $[10, 20]$Mbps. We make this choice due to the fact that Mininet supports at most an aggregated 10Gbps shared among all links, our topology has 99 links, and we require 75 control-plane links to connect from an edge switch to each host. For experiments 1 and 2, we consider a simpler topology that facilitates the illustration by extracting 9 nodes and 9 links interconnected from CONUS topology to demonstrate the dynamic routing feature. Particularly for emulating this 9 nodes topology, we increase each link capacity to $[30, 40]$Mbps and retain their delays.

B. Dynamic routing

We aim to illustrate the dynamic routing feature of OpenMCR in this section. Initially, we select 20 random pairs of source and target nodes and generate a routing request for each of them with a bandwidth requirement in the range of $[3, 5]$Mbps and a lifetime of 10mins to cover the length of the experiment. We deploy the initial state on the CORONET network using paths computed by OpenMCR. We consider bandwidth requirements as non-additive constraints, delay and hop as additive constraints in experiment 1 and 2 below. We mark real time measured link weights (e.g. bandwidth, delay) on each link in Fig.5-Fig.7. OpenMCR computes the path based on real time measurements of link weights. Next, we inject another flow with bandwidth requirement of 10Mbps, delay bound of 10ms and hop bound of 5 from Abilene to Houston. The default route computed by OpenMCR is marked red in Fig.5.

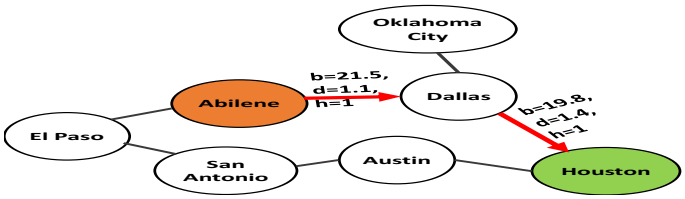


Fig. 5: Default route

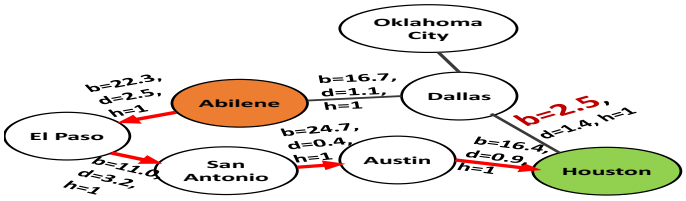


Fig. 6: Route changed due to link saturation

Experiment 1: Before injecting the flow from Abilene to Houston, we inject a big flow from Dallas to Houston in order to saturate the link. From Fig.6, we can see the available bandwidth on the link Dallas→Houston is only 2.5Mbps. The remaining available bandwidth is not enough to carry the flow Abilience→Houston. Thus, OpenMCR returns a different path Abilene→El Paso→San Antonio→Austin→Houston, which has been marked red in Fig.6. It has been demonstrated that OpenMCR will provide dynamic routing solutions based on real time network state.

Experiment 2: This is designed to illustrate the impact of link failure to routing path. With NetworkX we are able to maintain the real time and state based on runtime measurement. To emulate the link failure, we either remove the link Abilene→Dallas in real time topology or set the bandwidth to 0 and delay to ∞ for that link. We choose the latter option for illustration purpose. From Fig.7 we can see, OpenMCR provides an alternative path to route the flow given link Abilene→Dallas

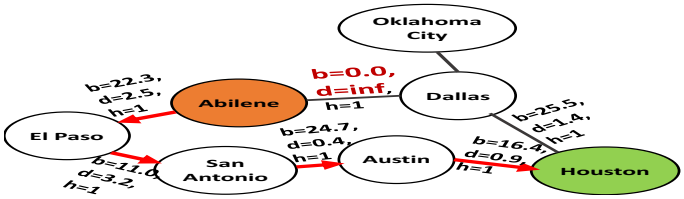


Fig. 7: Route changed due to link failure

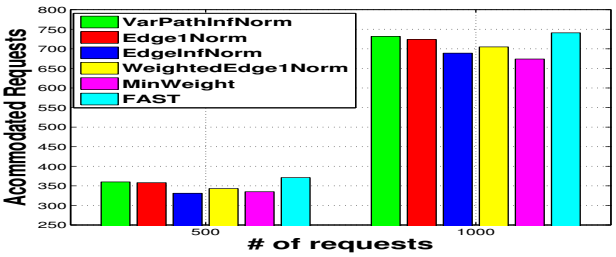


Fig. 8: Different NCs with RandomPickup

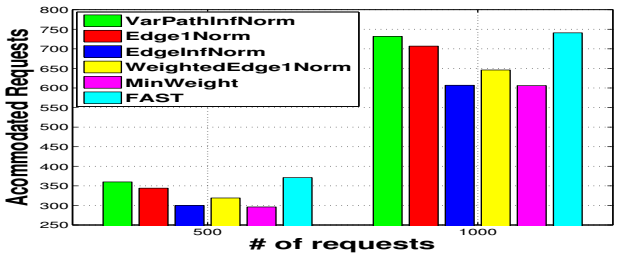


Fig. 9: Different NCs with GreedyOptimal

is disconnected. Once again, this demonstrates dynamic routing with OpenMCR.

C. Comparison of necessary conditions

We aim to compare the performance of various necessary conditions and validate theoretical analysis in Sec.IV-B. We conduct two experiments below. In each of the experiments, we randomly generate 500 and 1000 requests with common delay bound of 25ms and hop bound of 8. A request being accommodated means that OpenMCR has returned a feasible path that satisfies all the constraints for that request. The more accommodated requests are returned, the more effective the selected necessary condition is. Thus, we investigate the number of accommodated requests to validate the effectiveness of each different necessary condition. Note that we set $\epsilon = 0.5$ for the FAST necessary condition.

Experiment 3: We select Random Pickup as the path extension scheme in this experiment and combine it with each necessary condition. We run 500 and 1000 random source-target requests for each necessary condition and show the results in Fig.8. We can observe that, for both cases, FAST as the necessary condition leads to the most accommodated requests. Second is Variant Path ∞-Norm. Edge 1-Norm returned slightly fewer accommodated requests than Variant Path ∞-Norm. Minimum-weight performs worse than both Edge 1-Norm and Variant Path ∞-Norm. Edge 1-Norm outperforms Edge ∞-Norm. Weighted Edge 1-Norm is not as good as Edge 1-Norm. All these observations support the theoretical analysis in Sec.IV-B.

Experiment 4: We replicate experiment 3 but select Greedy Optimal as our path extension scheme. The results from this experiment are shown in Fig.9. Most observations are the same. But it is worth noting that for some necessary conditions (e.g. Edge 1-Norm, Edge ∞-Norm, Weighted Edge 1-Norm and Minimum Weight), Greedy Optimal does not perform as well as Random Pickup. The reason is that in each path extension iteration only the optimal node is greedily selected. Thus with a weak necessary condition, a greedy optimal node that is unqualified is more likely to be selected. With the best necessary conditions (e.g. FAST, Variant Path ∞-Norm), all three path extension schemes work consistently well.

D. SLA satisfaction

In this section, we aim to demonstrate that paths provided by OpenMCR can satisfy SLA requirements/constraints.

Experiment 5: We consider 5 DMCP routing requests from unique source to target nodes. Each of the requests has a random bandwidth requirement in the range of $[3, 5]$Mbps, delay constraint with bound of 15ms, hop constraint with bound of 10 and cost constraint with bound of 500. For each request we expect to have one feasible path returned that might differ from different routing algorithms/schemes. We randomly generate the cost of each link in the range of $[30, 50]$. This cost can be used to reflect the preference (e.g. location, temporal) of WAN provider to use that link for routing during particular time periods. In this experiment, we compare OpenMCR with some existing DMCP algorithms (e.g. Randomized, FAST_DMCP). We specify the running time requirement of each flow request to drive OpenMCR to select Variant Path ∞-Norm as the necessary condition and Greedy Optimal as the path extension scheme. For DMCP, path length is minimized by Greedy Optimal. We set $\epsilon = 0.5$ for FAST_DMCP.

The results are listed in Table.II. Column 3, 5, 7 list the delay, hop and cost of the paths returned by OpenMCR, respectively. The last three columns show the path lengths (defined in Sec.IV-C) of the paths returned by OpenMCR, Randomized and FAST_DMCP, respectively. The actual delay values are real time measured in Mininet by transferring real traffic flows. We inject these 5 flows one by one. The lifetime of each flow is long enough to cover the experiment.

We can see from Table.II that all the paths are feasible, which means SLA requirements (i.e. delay, hop, cost) are all satisfied. In addition, OpenMCR outperforms DMCP baselines such as Randomized and FAST_DMCP in terms of path length. In other words, path length value of the path returned by OpenMCR is no greater than that from existing baselines. Our results prove the effectiveness of OpenMCR for DMCP in this experiment.

Experiment 6: We use the same settings of flow requests as in experiment 6. The difference here is we try to solve OMCP by minimizing the path delay. Thus, we compare OpenMCR with A*Prune. The results are listed in Table.III. We can observe that the optimal paths returned by OpenMCR and A*Prune are the same since delay values are almost identical. The slight deviation is due to error of real time measurement. Comparing the results of Table.II and Table.III, we see that the paths returned by OpenMCR might be different in terms of different SLAs even though the source and target are the same. Take the request Chicago→Houston in experiment 6 as an example. The path with minimized path length from Chicago to Houston has a 13.3ms delay (row 4, column 3). However, the path Chicago→Houston in this experiment has only a 7.68ms delay (row 4, column 7). It has been demonstrated that OpenMCR is designed to dynamically find feasible/optimal paths based upon the actual SLAs of a request.

Node A	Node Z	Delay	Delay Bound	Hop	Hop Bound	Cost	Cost Bound	OpenMCR	Randomized	FAST_DMCP
Albany	Detroit	3.6ms	15ms	6	10	233	500	0.60	1.00	0.60
Syracuse	Wilmington	1.9ms	15ms	3	10	114	500	0.3	0.7	0.3
Baltimore	Kansas City	6.5ms	15ms	5	10	217	500	0.50	1.00	0.80
Chicago	Houston	13.3ms	15ms	7	10	284	500	0.88	0.96	0.88
Seattle	San Francisco	4.5ms	15ms	4	10	167	500	0.40	0.95	0.82

TABLE II: Path length comparison of different DMCP solutions

Node A	Node Z	Hop	Hop Bound	Cost	Cost Bound	OpenMCR	A*Prune
Albany	Detroit	6	10	259	500	3.71ms	3.73ms
Syracuse	Wilmington	3	10	114	500	2.97ms	3.10ms
Baltimore	Kansas City	6	10	240	500	6.36ms	6.31ms
Chicago	Houston	7	10	292	500	7.68ms	7.72ms
Seattle	San Francisco	4	10	167	500	4.49ms	4.43ms

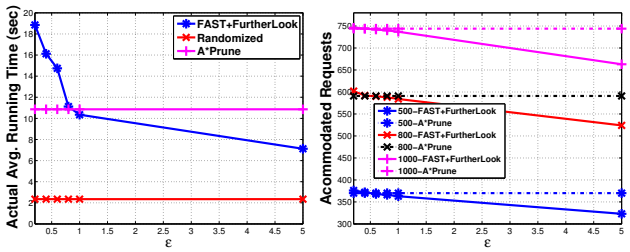
TABLE III: Delay comparison of different OMCP solutions with minimizing delay

E. Flexibility

In this section, we aim to show the flexibility of OpenMCR. That is to say, based on different SLAs of each request, OpenMCR can dynamically select the most appropriate necessary condition and sufficient condition to assemble the framework while ensuring SLAs are satisfied. SLAs consist of multiple factors. Experiment 8 is designed to satisfy an SLA with a specified epsilon value that reflects the tradeoff between the running time and performance (i.e. the number of accommodated requests). Experiment 9 is designed to meet the SLA with specified worst case running time requirement.

Experiment 7: As discussed in Sec.IV-D, if a client specifies an epsilon value, FAST will always be selected as necessary condition. In this experiment, we select Further Look as path extension scheme. We evaluate the tradeoff between actual average running time and the number of accommodated requests among 500, 800 and 1000 random requests. We increase the value of epsilon from 0.2 to 1.0 in increments of 0.2 and also collect the results with high epsilon value of 5.0. We compare OpenMCR with some baselines such as Randomized and A*Prune. The results are shown in Fig.10(a) and Fig.10(b). A trade-off can be observed that, although it takes less time to find a path as epsilon is increased, fewer requests are accommodated. That is to say, there will be more requests with no feasible paths found. Additionally, it can be observed that Randomized requires the least running time. A*Prune takes less time than OpenMCR when epsilon is close to 0. With respect to accommodated requests, OpenMCR has almost the same performance as A*Prune, especially when epsilon is no more than 1.0. It has been demonstrated that OpenMCR can provide as good solutions as classic DMCP/OMCP algorithms but it has more flexibility to support various SLAs.

Experiment 8: We consider the running time requirement of a SLA in this experiment. When a client specifies their running time requirement t, it means the path should be computed and proactively installed in the network by t. We guarantee that for any case, the running time should be no more than t. Thus, we take worst case running time of path computation into consideration. We generate 1000 random requests. Each request has a bandwidth requirement in the range of $[30, 50]$Kbps, a lifetime in the range of $[30, 120]$s, delay bound of 20ms and jitter bound of 5ms. This experiment lasts for 10mins. All the requests randomly arrive during that time period. We collect the results of actual average running times and total number of accommodated requests with respect to increased running time requirements for OpenMCR and two baselines - Randomized and A*Prune. The results are shown in Fig.11(a) and Fig.11(b). From both figures, we can see that the results of OpenMCR are a step function of running time requirement, which implies that different combinations of necessary and sufficient conditions are selected as running time requirement t is increased from 2s to 182s with step of 18s. According to our encoded logic, Edge1-Norm+GreedyOptimal is selected for $t = 2$s. VariantPath∞-Norm+GreedyOptimal is selected for $t \in [20, 74]$s. Edge1-Norm+FurtherLook is selected for $t \in [92, 146]$s. VariantPath∞-Norm+FurtherLook is selected for $t \geq 164$s. From Fig.11(a) and Fig.11(b), we can also see that randomized has the least running time but leads to the worst performance. OpenMCR has less running time than A*Prune and specially for $t = 2$s, A*Prune is infeasible since its worst case running time exceeds 2s. In some cases where $t \in [20, 74]$s, A*Prune outperforms OpenMCR while in the other cases where $t \geq 92$s, OpenMCR performs best. Thus, it has been demonstrated that OpenMCR can provide as good solutions as classic MCP algorithms or even better solutions and moreover, OpenMCR has more flexibility to support various SLAs.

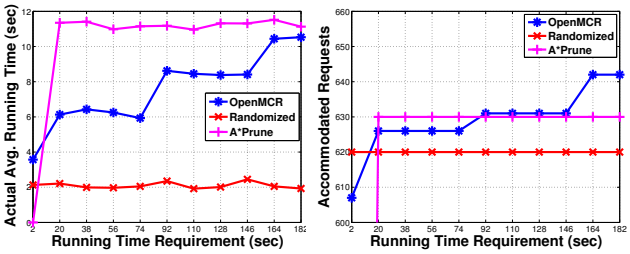


(a) Actual running time (b) Accommodated requests

Fig. 10: ϵ customization

F. Generality of topology

Experiment 9: We also evaluate OpenMCR over varying WAN topologies to validate our claim that OpenMCR can support arbitrary topologies. Due to page limit, we don't present the results here. Please refer to our technical report [25] for more details of this experiment.



(a) Actual running time (b) Accommodated requests

Fig. 11: Increasing running time requirement

VII. Related Work

Traffic engineering in SD-WAN has been the target of recent research [11], [12]. Google [12] deployed the first global software defined WAN interconnecting their datacenters. In [11], Hong et al., proposed a system SWAN to boost the utilization of inter-datacenter networks. While pioneering works in data centers well justify the benefits of SDN, such clean-slate designs cannot be directly applied to Carrier/ISP networks where unique requirements and properties create non-trivial challenges.

QoS routing has been extensively studied over decades for both wireline and wireless networks [4], [5]. In [8], [15], ϵ-optimal approximation algorithms were proposed. Other works [13], [23] provide randomized or limited granularity heuristics. [22] considered both *decision version* and *optimization version* of the MCP problem with improved time complexity. However, most research studies focus on a *specific* approach to get a feasible or approximate solution. More importantly, *these QoS routing protocols cannot be dynamically implemented due to the distributed nature of current Internet*. Our work provides a *general framework* on which most existing MCP schemes are compatible and realize it in SDN.

Optimization framework As SDN grows popular for simplifying network management, optimization upon SDN has attracted researchers' attention. Several optimization frameworks have been proposed [9], [10], [17], [18], [21]. A general, efficient framework SOL has been proposed in [10] for expressing and solving network optimizations. In [9], a centralized optimizer DEFO is proposed along with a two-layer architecture separating connectivity and optimization tasks. Authors of [21] proposed a framework Merlin for managing resources in SDN, specially for bandwidth allocations. [17] focuses on traffic steering for middleboxes in SDN, while [18] proposes a routing framework to strategically route traffic sub-populations over fixed monitors. However, none of these frameworks are compatible to solving multi-constrained routing problem.

VIII. Conclusion

In this paper we propose a general SDN routing platform, OpenMCR, that can provide multiconstrained QoS routing service to enterprise-level users. OpenMCR is backwards compatible with existing QoS routing approaches and it can also strategically implement a feasible/optimal routing solution for its users based upon their requirements. Via extensive experiments in an SDN emulated environment, we demonstrate its effectiveness compared to multiple baselines.

References

[1] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow, RSVP-TE: Extensions to RSVP for LSP Tunnels, *RFC 3209, Internet Engineering Task Force*, 2001.

[2] D. Black, Ed., P. Jones, Differentiated Services (Diffserv) and Real-Time Communication, *RFC 7657, Internet Engineering Task Force*, November 2015.

[3] R. Braden, D. Clark, and S. Shenker, Integrated services in the internet architecture: an overview, *RFC 1633, Internet Engineering Task Force*, June 1994.

[4] Yu Cheng, C. Zhou, and W. Zhuang, Minimizing end-to-end delay: a novel routing metric for multi-radio wireless mesh networks, *IEEE INFOCOM'2009*.

[5] S. Chen and K. Nahrstedt, On Finding Multi-Constrained Paths, *IEEE ICC'98*, Vol. 2, 1998, pp. 874-879.

[6] A. Farrel, J.-P. Vasseur, and J. Ash, A Path Computation Element (PCE)-Based Architecture, *RFC 4655*, 2006.

[7] G. H. Golub, C. F. Van Loan, Matrix Computations, Oxford, U.K., *North Oxford Academic*, 1983.

[8] R. Hassin, Approximation Schemes for the Restricted Shortest Path Problem, *Mathematics of Operations Research*, Vol. 17, No.1, 1992, pp. 36–42.

[9] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. *ACM SIGCOMM*, 2015.

[10] V. Heorhiadi, M. K. Reiter, and V. Sekar. Simplifying software-defined network optimization using SOL. *USENIX symposium on networked systems design and implementation*, 2016.

[11] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. *ACM SIGCOMM*, pages 15-26, 2013.

[12] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hlzle, S. Stuart and A. Vahdat, B4: Experience with a Globally-Deployed Software Defined WAN, *ACM SIGCOMM*, 2013.

[13] T. Korkmaz, M. Krunz, A Randomized Algorithm for Finding a Path Subject to Multiple QoS Constraints, *Computer Networks*, Vol. 36, No.23, 2001, pp. 251–268.

[14] G. Liu, K. G. Ramakrishnan, A*Prune: An Algorithm for Finding K shortest Paths Subject to Multiple Constraints, *IEEE Infocom 2001*, Vol. 2, April 2001, pp. 743–749.

[15] D. Lorenz, D. Raz, A simple efficient approximation scheme for the restricted shortest paths problem, *Operations Research Letters*, Vol. 28, 2001, pp. 213–219.

[16] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. A. Maltz, Latency inflation with MPLS-based traffic engineering, *Internet Measurement Conference*, 2011, pp. 463472.

[17] Z. Qazi, and C.-C Tu, L. Chiang, R. Miao, V. Sekar, M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. *ACM SIGCOMM*, 2013.

[18] S. Raza, G. Huang, C.-N. Chuah, S. Seetharaman, and J. P. Singh. MeasuRouting: a framework for routing assisted traffic monitoring. *ACM/IEEE Transactions on Networking*, 20(1):45-56, 2012.

[19] E. Rosen and Y. Rekhter, BGP/MPLS VPNs, *RFC 2547, Internet Engineering Task Force*, 1999.

[20] J. H. Saltzer, D. P. Reed, and D. Clark, End-to-end arguments in system design, *ACM Transactions on Computer Systems*, vol. 2, no. 4, Nov. 1984.

[21] R. Soule, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. *ACM CoNEXT*, 2014.

[22] G. Xue, W. Zhang, J. Tang, K. Thulasiraman, Plynomial Time Approximation Algorithms for Multi-Constrained QoS Routing, *IEEE/ACM Transactions on Networking*, Vol. 16 (2008), pp. 656-669.

[23] X. Yuan, Heuristic Algorithms for Multiconstrained Quality-of-Service Routing, *IEEE/ACM Transactions on Networking*, Vol. 10, No. 2, 2002, pp. 244–256.

[24] CORONET CONUS Topology, http://www.monarchna.com/topology.html

[25] https://www.dropbox.com/s/krldlbyb7s90tph/OpenMCR-TechReport.pdf?dl=0

[26] Mininet; http://mininet.org/

[27] NetworkX; https://networkx.github.io/