Distributed Load-Sensitive Routing for Computationally-Constrained Flows

An-Cheng Huang and Peter Steenkiste
Carnegie Mellon University

***Abstract*— A network that provides not only connectivity but also computational resources to application flows will enable a new array of network services. For example, applications that require content adaptation can be deployed more easily in a network that provides integrated communication and computational resources. In this paper, we study the problem of finding a path for a flow that has both computational and bandwidth constraints. We present a distributed load-sensitive routing algorithm that generates precomputed routing information and optimizes the routing decisions for both applications and computational and communication resource providers. We show through simulations that our distributed approach performs comparably to a centralized algorithm and is more resilient to longer routing update intervals.**

I. Introduction

In today's network, many service providers are starting to deliver services that require "computational resources" in the network. For example, in a video streaming application, a transcoder may be needed to handle different codecs; when a handheld device accesses a web site, a proxy may be needed to rewrite the web pages. To provide such services, the service provider will have to route the application flows along paths that meet the communication constraints and also traverse network nodes with sufficient computational resources to run the service. We are interested in the problem of how to find a path that meets both the computational resource and the bandwidth constraints of an application flow. We will call this the QoS Routing with Computational Constraints (QRCC) problem, and it is an instance of the multi-constrained routing problem.

There are centralized algorithms that can solve the QRCC problem. However, centralized algorithms require that the global network state is available to the node running the algorithm. This may be a scalability problem. Therefore, our goal is to devise a distributed solution for the QRCC problem. Finding such a solution is difficult because a distributed algorithm needs to precompute routing information for all possible flow requests, the routing decisions need to be optimized to satisfy application constraints and achieve high resource efficiency, and the highly dynamic metrics in this problem may cause routing loops in a distributed environment.

In this paper, we present an extended distributed Bellman-Ford algorithm for the QRCC problem. Leveraging previous results in centralized QoS routing, we use a combination of two heuristics to handle the problem. The "discretization" heuristic is used on the computational resource metric to reduce the complexity of the problem, and a function of various metrics is used as the path cost function (PCF) for calculating the shortest paths so that the algorithm can adapt to the changing metrics. Our simulation results demonstrate that our heuristics are effective and that our distributed approach performs comparably to centralized source routing in many configurations.

In the remainder of this paper, we present the QRCC problem and challenges in Section II and describe how we address the challenges in Sections III through V. Our evaluation methodology and results are presented in Sections VI and VII. We then discuss the related work and conclude in Sections VIII and IX.

II. Problem definition and challenges

A. Problem statement

Given a network with computational resources on the network nodes, we are interested in the problem of finding a path for an application flow that requires one intermediate processing step, for example, a transcoder for a video streaming application. In addition, the path also needs to satisfy the bandwidth constraint of the application. Given these conditions, we define the QRCC problem as follows. Let the graph $G = (V, E)$ represent the network. V is the set of all nodes, and E is the set of all links. Each node $v \in V$ has computational resource $R(v)$, and each link $(v, v') \in E$ has available bandwidth $b(v, v')$. When the application starts, it issues a routing request that includes the sender node v_s, the receiver node v_r, the computational resource constraint C_c, and the bandwidth constraint C_b. Given this request, we need to find a path $p = \langle v_0, v_1, \dots, v_n \rangle$ that satisfies the following three conditions: (1) $v_0 = v_s$ and $v_n = v_r$. (2) $\exists 0 \leq t \leq n$ such that $R(v_t) \geq C_c$. This v_t is called the *target server*, i.e., the node that provides the computational resources needed by the application flow. (3) $\forall 0 \leq i \leq (n-1)$, $b(v_i, v_{i+1}) \geq C_b$.

The QRCC problem can be solved by centralized algorithms. For example, a centralized algorithm is proposed in [1] to solve a similar problem. However, a centralized algorithm must be applied to the global network state collected on a single node, and this may be a scalability problem. Therefore, our goal is to devise a distributed solution.

B. Challenges

(1) *Distributed algorithm*: We need a distributed routing algorithm that can handle both the bandwidth and computational resource metrics. The computational resource metric is a new type of routing metrics and is different from commonly

used ones such as end-to-end delay. In addition, it is much more difficult to use a distributed algorithm to handle the two metrics in the QRCC problem than using a centralized one.

(2) *Optimization*: When an application sends a flow request to a service provider, the provider needs to find a path that is "good" for both the application and the provider. For the application, the path should satisfy the application constraints and/or achieve good user-perceived performance. For the service provider, the goal is to achieve high resource efficiency. For example, if the routing algorithm routes many flows through one particular link, that link may become congested and cause further flow requests to be rejected, while other alternative links do not see much traffic. This leads to lower resource efficiency for the service provider. Therefore, the algorithm should be load-sensitive.

(3) *Highly dynamic metrics*: In the QRCC problem, the two metrics are highly dynamic (they change with every flow). Using these metrics in distributed load-sensitive routing may cause many routing loops, resulting in many flows being rejected.

In the next three sections, we will describe how we devise a distributed algorithm, optimize the routing decisions, and handle the dynamic metrics.

III. Devising a distributed algorithm

To devise a distributed algorithm for the QRCC problem, we leverage existing work on multi-constrained QoS routing. Traditional QoS routing metrics are of two main types. (1) *Minimum additive* metrics such as end-to-end delay: the sum of the delay of all links along the path must be lower than the application constraint. (2) *Maximum concave* metrics such as bandwidth: the bandwidth on the bottleneck link (i.e., the link with the lowest bandwidth) along the path must be higher than the application constraint. We identify the computational resource metric as *maximum convex*, i.e., the computational resources on the "richest" node (i.e., the node with the most resources) along the path must be higher than the application constraint. Therefore, the QRCC problem becomes a QoS routing problem with constraints on one maximum convex metric (computational resource) and one maximum concave metric (bandwidth).

It is much more difficult to use a distributed algorithm to handle the two metrics in the QRCC problem than using a centralized algorithm. The reason is that distributed routing algorithms need to "precompute" routes for all possible flows. The common technique is to precompute the optimal path, for example, when end-to-end delay is used as the routing metric, a distributed routing algorithm can precompute the "shortest path" between two nodes, which can be used for all flows between those nodes. However, with two metrics, the number of potentially optimal paths between two nodes can grow exponentially [2], [3]. Therefore, the technique of precomputing the optimal path no longer works.

A. Our approach

We start with the basic distributed Bellman-Ford algorithm [4], which is a shortest path algorithm and deals with one minimum additive metric. Therefore, we need to extend the algorithm to handle the maximum concave (available bandwidth) and the maximum convex (computational resources) metrics in the QRCC problem.

We first look at how previous studies deal with multiple metrics in a centralized QoS routing algorithm. One heuristic is to define precedence among the metrics, for example, the shortest-widest path algorithm (presented in [2] as a distributed algorithm, but used as a centralized one in other studies such as [5]). Another heuristic is to use a function of different metrics as the path cost function (PCF), and find the path with the lowest cost. Examples include the approximate algorithm in [6] and the shortest-distance algorithm in [5]. "Discretization" (or "limited granularity" [3]) is another heuristic to reduce the complexity of problems with constraints on multiple metrics. For example, the EDSP and EBF algorithms in [7] discretize one metric and find the optimal paths based on the second metric.

Leveraging these previous results, we combine two heuristics in our precomputed/distributed approach to deal with the QRCC problem. First, we define a PCF as a function of various metrics such that a "better" path will have a lower cost (e.g., a path with higher available bandwidth should have a lower cost). Given a source and a destination, the algorithm computes the path with the lowest cost. Therefore, we need to choose a PCF so that the cost of a path will reflect both the available bandwidth and the computational resources along the path. Section IV will describe how we choose such PCFs to optimize the routing decisions. For simplicity, we will use the sum of link costs as the PCF in the description of our algorithm. The second heuristic is to discretize the computational resource metric into k discrete levels. For each pair of source and destination and each resource level i ($0 \leq i \leq (k-1)$), the algorithm computes the shortest path whose target server has computational resource level i.

B. Extended Bellman-Ford algorithm

Now we present our extended Bellman-Ford algorithm. The network is represented by a graph $G = (V, E)$; V is the set of all nodes, and E is the set of all links. $R(v)$ denotes the computational resources on node $v \in V$, and the algorithm discretizes the computational resource metric R into a number of discrete levels $(0, 1, 2, \ldots, K)$ using a discretization function D. Let S denotes the set of the discrete levels ($S = \{0, 1, 2, \ldots, K\}$), and $r(v) \in S$ denotes the computational resource level at node $v \in V$. In other words, for all $v \in V$, $r(v) = D(R(v))$. $w(u, v)$ is the link cost of $(u, v) \in E$ (note that the link cost can be a function of different metrics such as bandwidth, latency, etc.). For simplicity, in the algorithm description, we use the sum of the link costs as the PCF. Therefore, the shortest path computed by the algorithm is the path with the lowest cost. Given the destination node $d \in V$, for all $v \in V$ ($v \neq d$) and $k \in S$, we define the following:

- $n[v][k]$: The next hop on the current shortest path p (from v to d) that traverses a node with resource level k.

```
INITIALIZE($d$)
1   for each node $v \in V$
2   do {
3       for each resource level $k \in S$
4       do {
5           $c[v][k] \leftarrow \infty$
6           $n[v][k] \leftarrow$ NIL
7           $t[v][k] \leftarrow$ NIL
8       }
9   }
10  $c[d][r(d)] \leftarrow 0$
11  $t[d][r(d)] \leftarrow d$

RELAX($u, v, k$)
1   if $c[v][k] > c[u][k] + w(u,v)$
2   then {
3       $c[v][k] \leftarrow c[u][k] + w(u,v)$
4       $n[v][k] \leftarrow u$
5       $t[v][k] \leftarrow t[u][k]$
6   }
7   if $c[v][r(v)] \geq c[u][k] + w(u,v)$
8   then {
9       $c[v][r(v)] \leftarrow c[u][k] + w(u,v)$
10      $n[v][r(v)] \leftarrow u$
11      $t[v][r(v)] \leftarrow v$
12  }

EXTENDED-BELLMAN-FORD($d$)
1   INITIALIZE($d$)
2   for $i \leftarrow 1$ to $2(|V| - 1)$
3   do {
4       for each edge $(u, v) \in E$
5       do {
6           for each resource level $k \in S$
7           do {
8               RELAX($u, v, k$)
9           }
10      }
11  }
```

Fig. 1. The extended Bellman-Ford algorithm.

- $c[v][k]$: The estimated cost of the current shortest path p (from v to d) that traverses a node with resource level k.
- $t[v][k]$: The node with resource level k on the current shortest path p (from v to d) of level k. This is the target server on path p.

The algorithm is shown in Figure 1. The path cost function (PCF) it uses to compute the shortest path is the sum of the link costs ($w(u,v)$) along the path. If we want to use a PCF that involves more than the link costs (for example, a PCF that involves the computational resources on the target server), we need to modify the path cost calculation in the algorithm. We will have a more detailed description of the PCFs when we discuss optimizing the routing decisions in Section IV.

The main difference between this algorithm and the original Bellman-Ford algorithm (see [8]) is the extra **if** statement in RELAX() (lines 7 to 12). In addition, $c[v][k]$ and $n[v][k]$ have an extra dimension in our algorithm, since it precomputes the shortest path for each computational resource level. Our algorithm also keeps track of the target server of a path ($t[v][k]$), since we need to know which node can provide the requested computational resources after a path is found for a flow. Like the original Bellman-Ford algorithm, our algorithm can be executed in a distributed and asynchronous fashion by exchanging information between neighboring nodes and executing the relaxations on different nodes in parallel (see [4]).

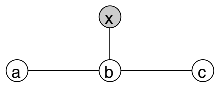


Fig. 2. An example of non-simple paths.

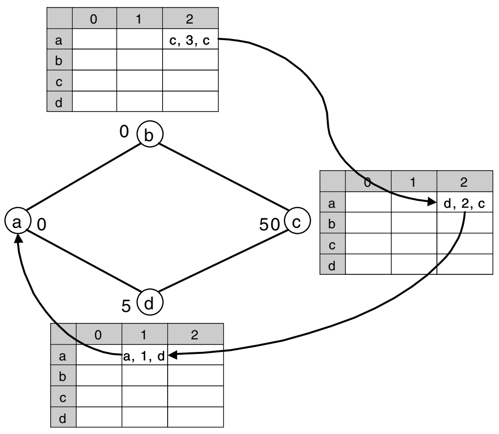


Fig. 3. Finding a path.

After the routing tables (the $n[v][k]$ entries) are computed, the path for a flow can be found by following the appropriate entries. Note that paths found by our algorithm may not be "simple paths". For example, in Figure 2, node x has sufficient computational resources for a flow from source node a to destination c. Our algorithm will find the path "$abxbc$", which is not a simple path.

Let us use an example to illustrate how to find a path for a request after the algorithm computes all the information. Figure 3 shows a simple network in which nodes a and b have no computational resources, node d has 5 units of computational resources, and node c has 50 units. We use the following discretization function:

$$r(v) = D(R(v)) = \begin{cases} 0 & \text{if } R(v) = 0 \\ 1 & \text{if } 0 < R(v) < 10 \\ 2 & \text{if } R(v) \geq 10 \end{cases}$$

Therefore, $r(a) = r(b) = 0$, $r(d) = 1$, and $r(c) = 2$. Each node constructs its routing table using the information computed by the algorithm above (for simplicity, most entries are not shown, and the PCF in this example is the hop count). Assume that a flow from node b to node a requires 9 units of computational resources, which corresponds to level 1. Therefore, we need to find a target server with computational resource level 2. We can find the target server and the path as depicted in Figure 3. Let rt$[b][a][2]$ denotes node b's routing table entry for destination a and computational resource level 2. We start with rt$[b][a][2]$, which is $(c, 3, c)$, i.e., the next hop is node c, the path cost is 3, and the target server is node c. Therefore, we follow it to node c and look at rt$[c][a][2]$, which is $(d, 2, c)$, so we follow it to node d. However, now that we have already traversed the target server, we will not use rt$[d][a][2]$. Instead, we will look at rt$[d][a][1]$. The reason is that since we have already traversed the target server, we do not need to consider the computational resource constraint any more. Since we know node d has computational resource level 1, rt$[d][a][1]$ implies the shortest path to the destination node a. In other words, given destination x and computational resource constraint level k, the path can be found by following rt$[v][x][k]$ for each node v "before" the target server and then following rt$[v'][x][r(v')]$ for each node v' "after" the target server. This path-finding phase may be integrated with the resource reservation mechanism so that resources can be reserved along the path. If we reach a link that does not have sufficient bandwidth, or if we reach the target server and discover that it does not have sufficient computational resources, then the flow is rejected.

IV. Optimizing the routing decisions

There are two goals for optimizing routing decisions. First, the computed paths should satisfy the application constraints. Since we are using a heuristic algorithm, the paths found by the algorithm may not actually satisfy the application constraints, so we want to maximize the likelihood that a path found by the algorithm actually works. Second, the routing decisions should result in high resource efficiency, so we want the algorithm to avoid creating "hotspots", for example, avoid routing all flows through the same node/link. We approach these goals by defining the path cost function (PCF) as a function of various metrics such that a "better" path (according to the above description) will have a lower cost. Therefore, two simple heuristics are used in defining the PCF. First, a link with higher available bandwidth should have a lower link cost. Second, a path whose target server has more available computational resources should have a lower cost. Based on these two heuristics, we define a family of PCFs as follows.

Let $p = \langle v_0, v_1, \ldots, v_n \rangle$ be a path that goes from v_0 to v_n, and v_t is the "target server" on p. $R(v_t)$ is the computational resource on the target server (note that this is the "real" resource value, not the discrete level). We assume each link $(v_i, v_{i+1}) (0 \leq i < n)$ has a link distance $s(v_i, v_{i+1})$ and available bandwidth $b(v_i, v_{i+1})$. The family of PCFs is defined as follows.

$$\text{Cost}(p, k, m) = \frac{\sum_{i=0}^{n-1} \frac{s(v_i, v_{i+1})}{b(v_i, v_{i+1})^k}}{R(v_t)^m}$$

By changing k and m, we can adjust the impact of available bandwidth and computational resources on the path cost. For example, when we increase k, we reduce the costs of paths with higher-bandwidth links. Similarly, when m is increased, the costs of paths whose target servers have more computational resources become lower. When k and m are set to 0, if $s(u, v)$ is the delay on link (u, v), the PCF becomes the end-to-end delay, and if $s(u, v) = 1$ for all links, the PCF is simply the hop count. In this paper, we will consider the following four PCFs: $\text{pcf}_0(p) = \text{Cost}(p, 0, 2)$, $\text{pcf}_1(p) = \text{Cost}(p, 0, 1)$, $\text{pcf}_2(p) = \text{Cost}(p, 0.5, 2)$, and $\text{pcf}_3(p) = \text{Cost}(p, 0.5, 1)$.

Note that since these PCFs involve the computational resources on the target server, $R(v_t)$, the algorithm in Figure 1 needs to be changed slightly so that when calculating the cumulative path cost, it uses the correct computational resource value. Specifically, at line 7 in the Relax function, $c[v][r(v)]$ is the cost of the path whose target server is v, while $c[u][k]$ is the cost of the path whose target server is $t[u][k]$. Therefore, we need to convert $c[u][k]$ to a new cost based on the resources on v so that we can do the comparison at line 7.

V. Handling highly dynamic metrics

Routing algorithms based on the distributed Bellman-Ford algorithm have two inherent problems: "count-to-infinity" ("bouncing effect") [4], [9] and "routing-table loops" [9]. For example, routing loops occur when the routing tables of two or more nodes contain conflicting next hop information, e.g., node a points to b while b points to a. These problems get worse in the QRCC problem because the metrics used in the PCF described above are highly dynamic. Since each node only exchanges network state with its neighboring nodes, many nodes will be using stale network state to compute the next hop information. Therefore, the routing tables may include many routing loops, resulting in many flows being rejected.

We observe that there is a trade-off between the optimality of the routing decisions and how dynamic the PCF is. For example, if we use a static PCF in the algorithm, routing loops are eliminated since all nodes will always use the same static network state to compute the paths, which in fact are always the same. The problem is of course that the routing decisions will be sub-optimal since the algorithm does not consider the available bandwidth and computational resources, i.e., not load-sensitive. On the other hand, using a dynamic PCF produces better paths, but some flows will be rejected due to routing loops.

Based on this observation, we enhance our algorithm with a "fallback" mechanism: in addition to the routing table entries computed using the dynamic PCFs in Section IV, the enhanced algorithm also uses a "less dynamic" PCF to compute a "fallback path" between each pair of source and destination, i.e., there will be an extra column in each routing table in Figure 3. This less dynamic PCF is called the "fallback PCF" (FPCF). Since the FPCF is less dynamic, paths computed using the FPCF will be less likely to contain routing loops. When we try to find a path for an application flow, we first try to use the path computed using the regular, more dynamic PCF. This path is constructed by looking up the routing tables as described in Section III-B, i.e., first look up the target server, then construct the path segment from the source to the target server, and finally construct the path segment from the target server to the destination. If a loop is encountered when we construct either segment, instead of rejecting the application flow, we can use the fallback path computed using the FPCF. Therefore, we are able to reduce the number of application flows that are rejected due to routing loops.

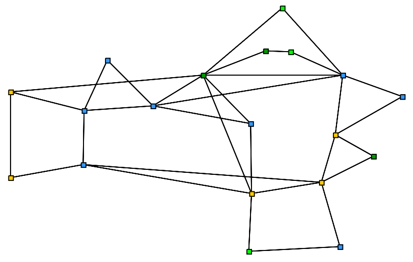


Fig. 4. Simulation topology.

In this paper, we use the static $\text{Cost}(p, 0, 0)$ as the FPCF in most of our simulations. As a result, the fallback paths are static and contain no routing loops, In Section VII-E, we will present some preliminary results on the effectiveness of using FPCFs that are not static.

VI. Evaluation

In this section, we evaluate the effectiveness of our heuristics and the different PCFs and the efficiency of different network resource provisioning schemes, and we also want to compare our precomputed/distributed approach with an on-demand source routing approach. We decide to use "rejection rate" (connection blocking rate) as our performance metric: from an application's perspective, a lower rejection rate means better service, and from a service provider's perspective, given the same amount of resources, an approach with a lower rejection rate achieves higher resource efficiency. We define two types of rejections: "resource rejections" (our algorithm finds a path that does not have sufficient computational resources for a flow) and "bandwidth rejections" (our algorithm finds a path that does not have sufficient bandwidth).

We designed and implemented an event-driven simulator to evaluate the performance of our approach. Events such as flow requests and routing updates trigger the simulator to find a path for a flow and execute one iteration of the distributed algorithm. In our simulations, we used the MCI backbone topology (Figure 4). The link distance $s(u, v)$ in the cost function is generated with a continuous uniform distribution between 0.5 and 2. The computational resource constraint of a flow is either 1 unit or 2 units. We use 4 different levels of bandwidth load: 0.2/0.4 (i.e., flows with computational constraint 1 have bandwidth constraint 0.2, and flows with computational constraint 2 have bandwidth constraint 0.4), 0.4/0.8, 0.6/1.2, and 0.8/1.6. In other words, the bandwidth constraint of a flow is proportional to its computational resource constraint. For convenience, we denote these 4 bandwidth load levels using the average, i.e., average bandwidth 0.3, 0.6, 0.9, and 1.2.

The duration of each flow is generated using a bounded Pareto distribution (lower bound 0.2, upper bound 1000, and $\alpha = 1.5$). The inter-arrival time between flows is generated using an exponential distribution, and we use 5 different average inter-arrival times from 0.0011 to 0.0018. The flow inter-arrival time determines the computation load in the network, for example, if the average inter-arrival time is 0.0011, and all flows are admitted, on average 90% of all computational resources in the network are being used by application flows, i.e., the computation load is 0.9. Therefore, we have 5 different levels of computation load from 0.9 to 0.55.

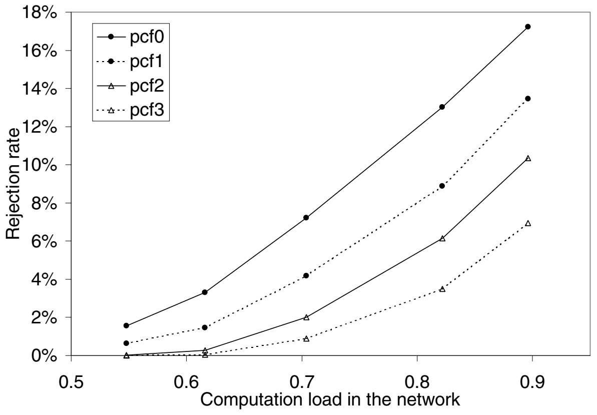


Fig. 5. Algorithm performance with different PCFs.

We use three computational resource distributions: "3X300" (three random nodes with 300 units on each), "6X150", and "9X100". The nodes that have computational resources are called "resource nodes", and a link is called a "resource link" if at least one of its two ends is a resource node. In most of our simulations, we use one of two bandwidth capacity allocation schemes. The "Flat" allocation scheme means that every link has 100 units of bandwidth. The "Planned" scheme means that every resource link has 150 units of bandwidth, and every non-resource link has 50 units of bandwidth. Finally, we use routing update intervals from 5 flows to 250 flows (average number of flow requests between updates).

VII. Simulation results

A. Algorithm performance

In this set of simulations, the computational resource distribution is 6X150, the bandwidth capacity allocation is Flat, and the routing update interval is 100 flows. Figure 5 shows the rejection rates of the four different PCFs (Section IV) as a function of the computation load in the network. The average bandwidth is 0.6. We can see that the rejection rates of pcf_2 and pcf_3 are lower than those of pcf_0 and pcf_1, respectively. This shows that using the available bandwidth metric in the PCF (i.e., use $k > 0$ in $\text{Cost}(p, k, m)$) can help balance the load and achieve lower rejection rates. In addition, pcf_1 and pcf_3 perform better than pcf_0 and pcf_2, respectively. This shows that if m is too high in the PCF, the computational resource metric is over-emphasized, and too many flows will be directed to the same node(s), causing worse load-balancing.

Next we look at the contributions of different types of rejections. Figure 6 shows the contributions of bandwidth and resource rejections with the four PCFs under computation load 0.822. When the average bandwidth is 0.6, almost all rejections are "resource rejections". When the average bandwidth is 0.9, resource rejections are reduced dramatically with the use of "better" PCFs, but bandwidth rejections remain fairly constant. The reason is likely that we use a static FPCF to compute

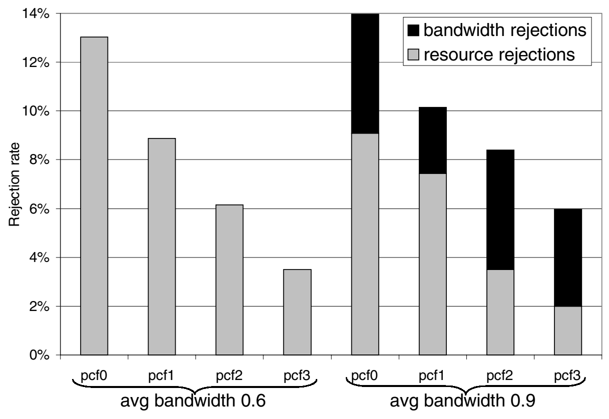


Fig. 6. Rejection sources.

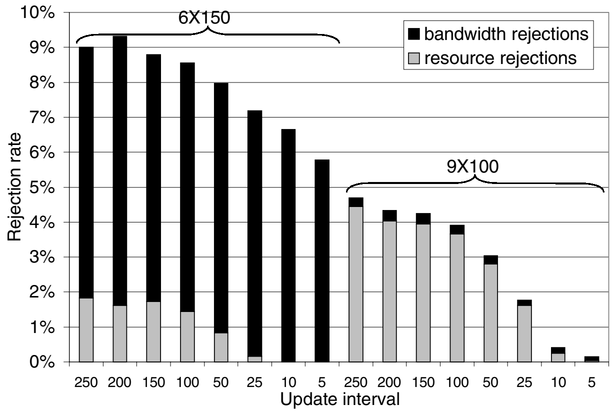
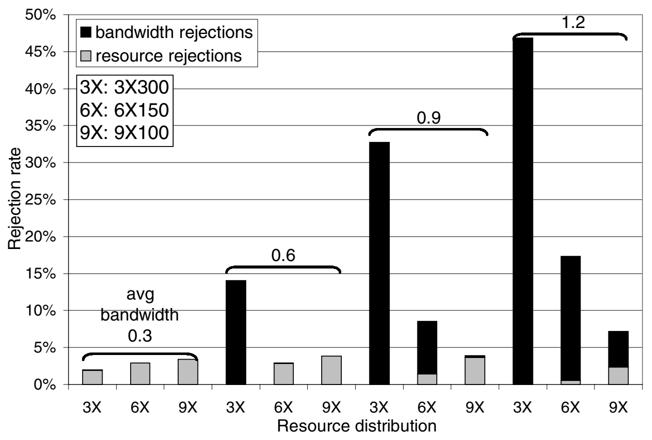
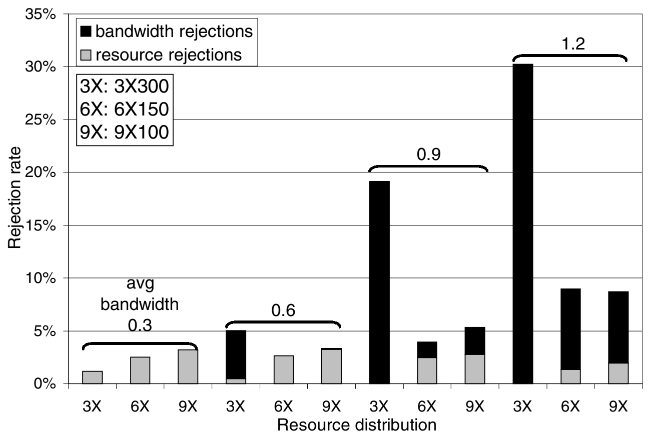


Fig. 7. Effect of routing update intervals.



(a) Flat



(b) Planned

Fig. 8. Effect of network resource provisioning.

fallback paths, which are not sensitive to bandwidth load, as described in Section V.

B. Effect of routing update interval

To evaluate the impact of the routing update interval, we use the following configuration: the algorithm uses pcf$_3$, the average bandwidth is 0.9, the bandwidth capacity allocation is Flat, and the computation load is 0.822. Figure 7 shows the contributions of the two sources of rejections under two different resource distributions, 6X150 and 9X100. Under 6X150, it is clear that the "bottleneck" is bandwidth, and shortening the update interval does not reduce bandwidth rejections very much. On the other hand, when the resource distribution is 9X100, bandwidth rejections are insignificant, and frequent updates can greatly reduce resource rejections. Overall, the update interval does not have a significant effect on bandwidth rejections. Again, this is probably due to the use of the static FPCF.

C. Network resource provisioning

Next, we look at how the computational resource distribution affects the rejection rates of our algorithm. The simulations use the following configuration: the algorithm uses pcf$_3$, the computation load is 0.822, the bandwidth capacity allocation is Flat, and the routing update interval is 100 flows. We compared 3 computational resource distributions (3X300, 6X150, and 9X100) under 4 different average bandwidth (0.3, 0.6, 0.9, and 1.2). Figure 8(a) shows the result. We can see that, as the bandwidth load increases, the overall rejection rates of the three resource distributions increase at different speeds. The rejection rate of 3X300 increases very quickly: at average bandwidth 0.3 it is the lowest, but at 0.6 it becomes the highest. 6X150 increases more slowly, surpassing 9X100 at average bandwidth 0.9, and 9X100 only increases slightly across the graph. Now if we look at the sources of rejections, we see that when the average bandwidth is 0.3, almost all rejections are resource rejections, and 3X300 has the lowest rejection rate. This suggests that when computational resources are the main constraint, a more "concentrated" resource distribution is better. When the average bandwidth becomes 0.6, for 3X300 the bandwidth rejections grow dramatically, while 6X150 and 9X100 remain roughly the same. Similarly, when the average bandwidth is increased to 0.9, bandwidth rejections become dominant for 6X150, and its overall rejection rate becomes significantly higher than that of 9X100. The reason is that with a more concentrated resource distribution like 3X300, each resource node will need to "serve" more flows. As a result, the resource links are overloaded more quickly when the bandwidth load is increased. Therefore, a more "dispersed"

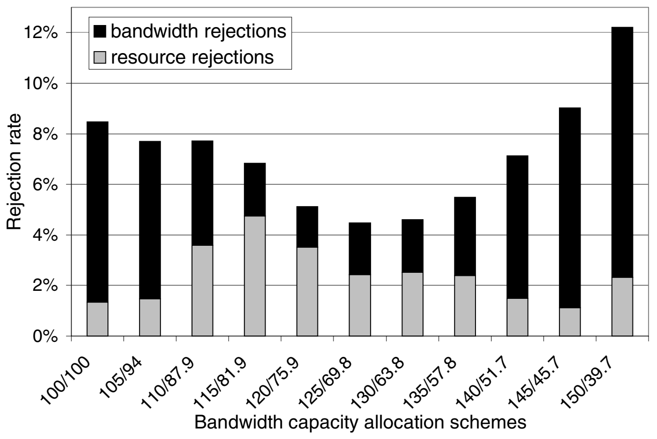


Fig. 9. Effect of different bandwidth capacity allocation schemes.

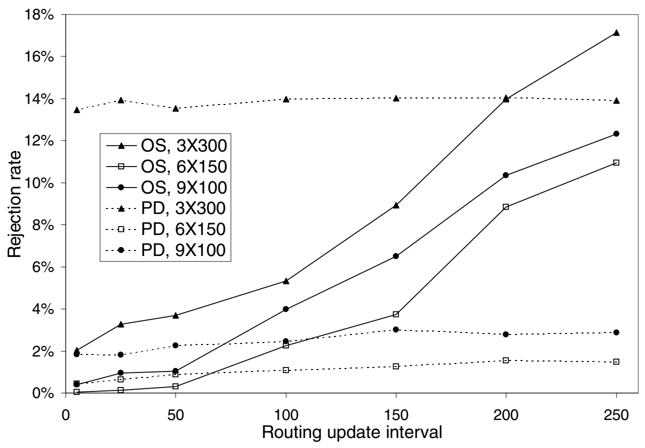


Fig. 10. Precomputed distributed (PD) vs. on-demand source routing (OS).

resource distribution is more resilient to higher bandwidth load.

Now let us compare the two bandwidth capacity allocation schemes: Flat (shown in Figure 8(a)) and Planned (shown in Figure 8(b)). We can see that in most cases the overall rejection rate under Planned is better than that under Flat, since bandwidth rejections are greatly decreased. This is what we expected. However, there are two exceptions: with 9X100 distribution under bandwidth loads 0.9 and 1.2, the bandwidth rejections under Planned are higher than Flat. This indicates that, in both cases, the resource links are no longer the bottleneck, and therefore using Planned allocation results in more bandwidth rejections since it reduces the bandwidth capacity of the non-resource links to 50.

These results suggest that allocating higher bandwidth capacity to the resource links helps, as long as the other links do not become the bottleneck. Therefore, one interesting question is how to find the "optimal" allocation of bandwidth capacity (given a fixed total amount of bandwidth capacity in the network). Intuitively, the optimal allocation will be the one that balances the load on resource links and the load on non-resource links. We conducted another set of simulations under the following configuration: the algorithm uses pcf_3, the computation load is 0.822, the resource distribution is 6X150, and the routing update interval is 100 flows. Each bandwidth capacity allocation scheme is denoted by "X/Y", where X is the bandwidth capacity assigned to every resource link, and Y is the capacity assigned to every non-resource link. All allocation schemes have roughly the same amount of total bandwidth capacity in the network. Figure 9 shows the rejection rates of these schemes. We see that bandwidth rejections decrease as we put more bandwidth on the resource links. However, when the bandwidth on the other links becomes too low, they become the bottleneck, and the rejection rate starts rising. The result roughly matches our intuition.

D. Precomputed distributed vs. on-demand source routing

We compare our precomputed distributed routing (PD) approach with an on-demand source routing (OS) approach that uses a brute-force algorithm. The OS approach has two advantages. First, it explicitly considers the bandwidth constraints by pruning links that do not have sufficient bandwidth. Second, it knows the computational resource constraint of a flow and can pick a node that has sufficient resources.

We evaluated how the two approaches perform under routing update intervals 5 to 250 flows. We need to keep in mind that "routing update" has different meanings for the two approaches. For the PD approach, a routing update means each node exchanges its "distance vector" with its neighboring nodes (i.e., one iteration in the Bellman-Ford algorithm). For the OS approach, a routing update means each node broadcasts/floods its link state to all other nodes in the network. We assume that both are completed before the next event in the simulation.

Figure 10 shows the performance of the two approaches. The PD approach uses pcf_3, the average computation load is 0.704, the average bandwidth is 0.9, and the bandwidth capacity allocation is Planned. We can see that when the routing update interval is short, the OS approach performs better. However, as the routing update interval increases, the rejection rate of the OS approach increases significantly, while the rejection rate of the PD approach only goes up slightly. The reason is likely that with the OS approach, all the nodes have a consistent picture of the global network state. As a result, all flows will be directed to use the nodes and links that are "good" at the time of the previous routing update. On the other hand, with the PD approach each node only exchanges information with its neighbors. Therefore, the PD approach introduces some randomness that helps balance the computation and bandwidth loads.

E. Trade-off between optimality and stability

So far we have been using the static $\text{Cost}(p, 0, 0)$ as the fallback PCF (FPCF) to compute fallback paths. Of course, these static fallback paths are far from optimal, e.g., they may not have sufficient bandwidth, resulting in bandwidth rejections. Therefore, we experimented with a slightly more dynamic FPCF, Cost(p,0.5,0). By using this FPCF, the computed fallback paths may contain loops, and therefore we can no longer guarantee that we can always find a loop-free path

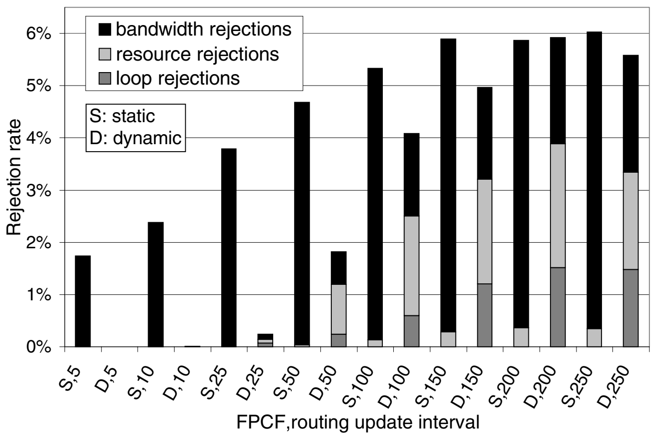


Fig. 11. Static and dynamic FPCFs.

for a flow. As a result, a new type of rejection, "loop rejection", is needed for this set of simulations. We use the following simulation configuration: the algorithm uses pcf_3, the resource distribution is 6X150, the bandwidth capacity allocation is Flat, the computation load is 0.704, and the average bandwidth is 1.2. The result is shown in Figure 11. "Static" means the FPCF is $\text{Cost}(p, 0, 0)$, and "dynamic" means the FPCF is $\text{Cost}(p, 0.5, 0)$. We see that with the static FPCF, there are no loop rejections. Using the dynamic FPCF does generate some loop rejections, but it is very effective in reducing bandwidth rejections. Overall, when the update interval is short, the dynamic FPCF is much better than the static one. When the update interval is long, although using the dynamic FPCF results in some loop rejections, the overall rejection rate is still very close to (or better than) using the static FPCF. Based on this preliminary result, the approach of using a dynamic FPCF looks promising.

VIII. Related work

The most relevant work to this paper was done by Choi et al. [1]. They studied the problem of finding a feasible path for an application session that requires computational resources. Their approach is to transform the problem into a shortest path problem by constructing a layered graph. This centralized algorithm requires the global network state, and maintaining the state on a central server or distributing the state to all nodes may be a scalability problem.

It has been shown that multi-constrained routing problems that involve constraints on two or more minimum additive metrics are NP-hard [2], [10]. Many previous studies focus on heuristic solutions to such problems [3], [6], [7], [11], [12]. Other studies looked at routing problems that involve the bandwidth constraint [2], [5]. Most of these previous studies on QoS routing use a centralized algorithm and assume the availability of accurate global network state. A probabilistic approach was used in [13] to deal with inaccurate network state. On the other hand, a number of studies presented distributed algorithms [2], [6], [14], but they did not consider problems such as routing-table loops caused by using highly dynamic metrics in the algorithms [4], [9].

IX. Conclusion

This paper addresses the problem of finding a path that satisfies both the computational and bandwidth constraints of an application flow in a network that provides both types of resources. We present a distributed load-sensitive routing algorithm that produces precomputed routing information based on these metrics. Our evaluation shows that our algorithm is able to achieve a rejection rate of less than five percent in many simulation configurations, and that using a function of both computational resource and bandwidth metrics in the algorithm can lead to better performance. On the effect of network resource provisioning, we demonstrate that computation and bandwidth loads should be considered when choosing a computational resource distribution, and that a good bandwidth capacity allocation scheme should balance the loads on resource links and non-resource links. When comparing our precomputed distributed approach with an on-demand source routing approach, our results indicate that, although the centralized approach performs well under very short routing update intervals (e.g., less than 50 flows), our distributed approach is more resilient to stale information caused by longer update intervals.

References

[1] S. Choi, J. Turner, and T. Wolf, "Configuring Sessions in Programmable Networks," in *Proc. IEEE INFOCOM 2001*, Apr. 2001.

[2] Z. Wang and J. Crowcroft, "Quality-of-Service Routing for Supporting Multimedia Applications," *IEEE JSAC*, vol. 14, no. 7, pp. 1228–1234, Sept. 1996.

[3] X. Yuan and X. Liu, "Heuristic Algorithms for Multi-Constrained Quality of Service Routing," in *Proc. IEEE INFOCOM 2001*, Apr. 2001.

[4] D. Bertsekas and R. Gallager, *Data Networks*, 2nd ed. Prentice Hall, 1992, ch. 5, pp. 404–410.

[5] Q. Ma, P. Steenkiste, and H. Zhang, "Routing High-bandwidth Traffic in Max-min Fair Share Networks," in *Proc. ACM SIGCOMM '96*, Aug. 1996, pp. 206–217.

[6] J. M. Jaffe, "Algorithms for Finding Paths with Multiple Constraints," *Networks*, vol. 14, pp. 95–116, 1984.

[7] S. Chen and K. Nahrstedt, "On Finding Multi-constrained Paths," in *Proc. IEEE International Conference on Communications '98*, June 1998.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. The MIT Press, 1990, ch. 25.

[9] C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves, "A Loop-Free Extended Bellman-Ford Routing Protocol without Bouncing Effect," in *Proc. ACM SIGCOMM '89*, Sept. 1989, pp. 224–236.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[11] A. Orda, "Routing with End-to-End QoS Guarantees in Broadband Networks," *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 365–374, June 1999.

[12] Q. Ma and P. Steenkiste, "Quality-of-Service Routing for Traffic with Performance Guarantees," in *Proc. Fifth International Workshop on Quality of Service*, May 1997, pp. 115–126.

[13] R. A. Guérin and A. Orda, "QoS Routing in Networks with Inaccurate Information: Theory and Algorithms," *IEEE/ACM Transactions on Networking*, vol. 7, no. 3, pp. 350–364, June 1999.

[14] D. S. Reeves and H. F. Salama, "A Distributed Algorithm for Delay-Constrained Unicast Routing," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2, pp. 239–250, Apr. 2000.