ABSTRACT

Based on the distributed and composable service model, the QoS service routing problem for supporting multimedia applications has emerged. Different from the conventional QoS data routing, QoS service routing presents additional challenges caused by the service availability, service dependency, and resource requirement heterogeneity issues, which make solutions for QoS data routing inapplicable to QoS service routing. Existing solutions for addressing this problem are either not generic enough or not integrated (service configuration selection is done separately from service path finding), so that they either become inapplicable to new environments/metrics or the computed paths are sub-optimal. This paper presents a generic and integrated approach for computing optimal service paths, and shows an aggregate performance function $\mathcal{F}$ that optimizes several routing metrics at the same time. We adopt a global-view approach, so that service paths can be computed quickly in a single node, without flooding the network. The simulation performances show that $\mathcal{F}$ is superior, and integrating service configuration selection with service path finding is desirable.

**keywords:** QoS, service routing, overlay networks

1. INTRODUCTION

Internet multimedia applications involve three aspects of heterogeneity: machine capacity, network connectivity, and user preference, which, together, make end-to-end customization of multimedia content more and more desirable. End-to-end customization is possible, or at least becomes easier and more scalable, with the introduction of media proxies at strategic locations, to intercept and transform data delivered from source to destination whenever necessary. Given the range of Internet heterogeneity, instead of developing monolithic services to cater to every special need, the recent trend is to build composable services such that multiple services can be applied in a chain to deliver a more complex one [1, 2]. Based on this, and assuming that services are widely distributed in media

---

This work was supported by the DARPA grant, F30602-97-2-0121, NSF grants CCR-9988199 and EIA 99-72884 EQ, and NASA grant NAG2-1406. Jingwen Jin is also supported by CAPES/Brazil.
proxies over the Internet, the service routing problem, which is to find a path that contains all the necessary services in a determined order, has emerged. Multimedia applications such as video conferencing and video on demand often have stringent QoS requirements. Therefore, service routing needs to be QoS-aware.

QoS service routing needs special attention because it differs from the conventional QoS (data) routing in several aspects:

• **service availability and service dependency** - While data routing is solely based on network connectivities, service routing depends on, in addition to network connectivities, service availability and service dependencies\(^1\). Later on, we will refer to these two issues as service requirements.

• **resource heterogeneity** - While in QoS data routing, resource requirement throughout a single data path is homogeneous, in QoS service routing, resource requirement throughout a service path is heterogeneous, because different services may have different requirements for machine capacity and I/O bandwidths\(^2\).

Given a proxy/service topology, a service request and a certain routing metric - \(m\), the QoS service routing problem is to find a path such that it satisfies the service and resource requirements and that its performance in terms of \(m\) is optimized. Stating in another way, the problem is how to map the service request onto the proxy overlay network so that the resulting service path is optimum in \(m\). While solutions such as [3, 4] exist for QoS service routing, they present certain limitations. In [3], the authors apply an extended Dijkstra’s algorithm on top of the proxy topology to compute the best service-to-proxy mapping in terms of their resource safety function, under the assumptions that all services are installed on all proxies (thus the service availability issue did not need to be concerned) and that a service request has linear dependencies. However, the solution becomes unsuitable for different routing metrics or different assumptions (e.g., if not all services are available on all proxies, or if service requests have non-linear dependencies). In [4], a two-step solution was presented: first, a resource-shortest service configuration (the one with minimum aggregate resource requirement) is selected from the service graph, then for this linear service configuration, a service path is sought in a distributed fashion. However, decoupling service configuration selection from service path finding may yield sub-optimal solutions. Assuming the two service configurations - \(c_1\) and \(c_2\) - in Figure 1(a): using the “minimum aggregate resource requirement” selection criterion, \(c_2\) is the preferred configuration. However, if we check the real service availability in the proxy network (Figure 1(b)), we see choosing \(c_1\) would be actually more advantageous, because CPU bandwidths are usually much larger than those of wide area network links (ratios in magnitude of 100 can be assumed). Therefore, the bandwidth requirement between \(s_1\) and \(s_2\) becomes negligible compared to CPU bandwidth, since both services can be discovered in a single proxy. In conclusion, whether or not a bandwidth (or proxy capacity) requirement is high can not be seen off-line from its absolute value; instead, it should be compared against the available one.

\(^1\)A service may be available only in certain proxies (the availability issue), and a service path request may require the found service instances to be applied in a determined order (the dependency issue).

\(^2\)Different services are likely to require different amounts of machine resources (CPU, memory etc), and the fact that services are transformational often makes the output data rate be different from the input data rate.
Unlike [3, 4], the approach to be presented in this paper intends to solve the QoS service routing problem in a generic and integrated way, so that it can adapt easily to new routing metrics and new environments, and the computed paths are optimal. Overall, the solution has the following features:

1. **generic** - The solution is generic enough to be applicable to any environment assumptions an any routing metrics.

2. **integrated** - Service configuration selection is integrated with service path finding, so that the computed paths are optimal.

3. **aggregated** - We identify several routing metrics most relevant to application-level service routing and show how they can be aggregated to optimize the computed service paths; i.e., we present a performance function that aggregates and optimizes several routing metrics at the same time.

4. **global-view based** - Like [3], our solution also adopts a global-view based approach, by maintaining full routing information in each proxy. Compared to distributed approaches, global-view based ones are preferred for: (a) high speed in establishing service paths, which is an important factor in most of the multimedia application scenarios; (b) not flooding the network to seek paths. Since service path finding can be done in any proxy, there is no problem of single point of failure in this case.

The remainder of this paper will be structured as the following. In Section 2, we describe the assumptions we will make throughout this paper. In Section 3, we present our aggregate routing function - $\mathcal{F}$ - which is composed by several metrics that we consider most relevant to QoS service routing in a proxy overlay network. In Section 4, our solution to QoS service routing is presented, and its complexity analyzed. In Section 5, we conduct simulation tests to: (1) measure the performance of $\mathcal{F}$ in terms of its individual metrics; (2) compare the performances of our integrated approach against those of the two-phase approach in [4]. Section 6 gives some conclusions.
2. ASSUMPTIONS

2.1. Service Model

An individual service component is associated with an input QoS - $Q_{in}$, and an output QoS - $Q_{out}$, where $Q_{in}$ and $Q_{out}$ are QoS vectors of multiple application-level QoS parameters such as image size, image resolution, video frame rate [5]. Each service $s$ has its resource usage function defined as $r_s : Q_{in} \times Q_{out} \to R$, that computes the amount of resources needed to deliver an output QoS $Q_{out}$ when $Q_{in}$ is the input QoS. A generic QoS-resource model is depicted in Figure 3.

We assume a distributed, composable service model, where multiple services can be applied in a determined order (e.g., $s_i \to s_j \to s_k$) to deliver a more complex service. The notation “$s_i \to s_j$” is used to indicate that service $s_i$ is followed by service $s_j$. When two services, $s_i$ and $s_j$, are to be composed, then the output quality of $s_i$ should be equal to the input quality of $s_j$. A QoS compiler [6] is able to translate a user’s request (e.g., get secure Video-on-Demand service with viewing quality of 30fps and in format of JPEG) into a possibly non-linear service graph - SG, where a path that leads from any source service to any sink service is a viable configuration, and any such single configuration would satisfy the end-to-end service requirements. In such an SG, all services have their input qualities and output qualities defined, according to the user’s requirement. These application-level quality parameters will be further interpreted by the QoS compiler into concrete resource requirements (e.g., CPU, memory, network bandwidth). Hereafter we will only consider service graphs at resource level. Figure 2(a) shows an example of SG (together with the services’ resource requirements) that has three viable configurations: $s_1 \to s_2 \to s_4$, $s_1 \to s_3 \to s_4$, and $s_1 \to s_3 \to s_5$. A service request consists of a service graph (SG), and a pair of source proxy $p_s$ and destination proxy $p_d$, as shown in Figure 2(b). The goal of this paper is to find a service
path from $p_s$ to $p_d$ that satisfies any configuration in the SG.

2.2. Proxy/Service Overlay Topology

Different from the network-layer data routing, where the network consists of physically connected routers and hosts, in the proxy/service overlay model, the proxy network is constructed on top of the physical network. As long as the underlying network does not partition, a proxy $p_r$ can always reach a proxy $p_y$; thus logically, proxies in a proxy overlay can be considered fully connected. However, due to the routing information measurement/maintenance cost issue, application-level networks are usually configured as partially connected topologies such as meshes, trees, and hypercubes [7, 8]. These kinds of topologies are mostly suitable for data routing. In service routing, due to the fact that any pair of services has probability of becoming service neighbors, and that the service neighboring issue is not resolved until a service request is actually made by a client, meshes or trees configured before that time usually cannot reveal the run-time service dependency needs. Figure 4(a) shows a mesh proxy topology. If we are to seek a service path for $s_1 \rightarrow s_2$ from $p_1$ to $p_2$, then either $p_3$ or $p_4$ needs to be included into the path to act as a relay. It is easy to see that the more the relays, the longer the service paths, and that service path lengths are inversely proportional to the in/out degrees of the proxies. The maximum service path efficiency is only achieved if the proxy overlay topology is fully connected. It is easy to see only the fully connected topology (Figure 4(b)) would be able to support all kinds of service dependencies without involving any relay nodes. A disadvantage associated with fully connected topologies is that they incur more routing information maintenance cost. On the other hand, mesh topologies incur less routing information maintenance cost, but longer service paths. In this paper, we leave the topology issue open, because the solution is applicable to all kinds of topologies.

The solution is also flexible with the service availability model; we may assume a single proxy is installed with all kinds of services, or a single proxy has only a partial set of services. However, we assume that services are not dynamically downloadable (active services). The reason that we do not consider active services is that to get them widely accepted is not easy, because most system administrators may not allow dynamic installations of software on their systems due to security concerns in the first place. Without the active service assumption, service availability becomes a problem in service routing.

The only thing that we enforce is that proxies maintain full service routing information for the centralized approach to work. We assume that periodically, each proxy monitors the performance values such as its own machine capacity/volatility, the communication delay/bandwidth from itself to other neighbors, and distributes this information to others either directly (if the topology is fully connected), or via flooding by using the link state protocol (if the topology is a mesh).

\footnote{The delay between two proxies can be measured by using a simple ping protocol. Methods for measuring end-to-end available bandwidths can be found in [9, 10].}
3. ROUTING PERFORMANCE METRICS

For service routing at the application level, we should consider several metrics such as service path delay, proxy hop count, proxy network bandwidth, proxy capacity, and proxy volatility. Ideally, the goal is to find a service path that satisfies the service requirements and that is optimal in all aspects. However, optimizing all of them at the same time is impossible, because they may conflict with each other. In this section, we present a single routing function - $F$ - that aggregates the above metrics, so that optimizing the value of $F$ more or less tends to optimize the individual metrics. Before presenting $F$, we first discuss the individual routing performance metrics.

3.1. Individual Performance Metrics

Individual performance metrics can be classified into three categories: additive, concave, and multiplicative. Let $sp = \langle n_0, n_1, \ldots, n_k \rangle$ be a service path, and let $m(n_i, n_j)$ denote the performance metric for service link $(n_i, n_j)$:

- **additive performance metric**: A metric $m$ is additive if $m(sp) = m(n_0, n_1) + m(n_1, n_2) + \ldots + m(n_{k-1}, n_k)$.

- **concave performance metric**: A metric $m$ is concave if $m(sp) = \min[m(n_0, n_1), m(n_1, n_2), \ldots, m(n_{k-1}, n_k)]$.

- **multiplicative performance metric**: A metric $m$ is multiplicative if $m(sp) = m(n_0, n_1) \times m(n_1, n_2) \times \ldots \times m(n_{k-1}, n_k)$.

Thus, delay and hop count are additive; bandwidth and proxy capacity are concave; and proxy volatility is multiplicative. A real service path may have the form: $sp = \langle -/p_0, s_1/p_1, \ldots, s_n/p_n, -/p_{n+1} \rangle$, where $p_0$ and $p_{n+1}$ are source and destination proxies, respectively, and $s_i/p_j$ means that service $s_i$ is mapped onto proxy $p_j$ (note that $-/p_i$ means that no service is mapped onto $p_i$).

**delay and hop count (additive)**: Delay $d$ of the service path $sp$ - $d(sp)$ is the time necessary for data to get through $sp$, which includes transmission delay and service execution delay; i.e., $d(sp) = \sum_{i=0}^{n} d(p_i, p_{i+1}) = \sum_{i=0}^{n} \text{trans}(p_i, p_{i+1}) + \sum_{i=1}^{n} \text{exec}(s_i/p_i)$. We consider hop count at the proxy granularity, and the number of proxy hops is the number of times the service path needs to switch proxies (if a proxy is visited twice for two
different services, then it is counted twice). The goal of service routing is to find a path such that the aggregate service path delay and/or the total number of hops is minimized.

**bandwidth and proxy capacity (concave):** While in data routing, bandwidth requirement in a single data path is homogeneous, and bandwidth optimization is achieved by seeking the widest path [11], in service routing, due to the heterogeneous resource requirement issue, selecting the widest-path in absolute value is no longer appropriate. For instance, in Figure 5(a), using the widest-path criterion, service $s$ would be routed through proxy $p_j$, leaving the residual bandwidth from $p_j$ to $p_d$ zero. While the main objective of the widest-path selection in data routing is to balance traffic on the Internet links, we see that traffic balance is not achieved with the genuine widest-path selection in service routing. To achieve traffic balancing, the residual bandwidth needs to be normalized based on the bandwidth requirement. We define the normalized bandwidth to be the ratio of residual bandwidth to required bandwidth: $bw_{\text{norm}} = \frac{bw_{\text{res}}}{bw_{\text{req}}}$. After the normalization process, the widest-path selection criterion can again help to achieve better traffic balance. Using the normalized widest-path criterion, $s$ should be routed through $p_i$ (shown in Figure 5(b)). Note that the bandwidth we talk about here is service bandwidth (bandwidth between two services) in that, when two services are located in two separate proxies, then the service bandwidth is the network bandwidth between the two proxies, and when two proxies are located in one single proxy, then the bandwidth is that proxy’s CPU bandwidth. Normalization should also be applied to proxy capacity for proxy load balancing purposes. Proxy capacity refers to several machine items, e.g., CPU and memory, and can be usually represented in an $n$-tuple vetor[5]. For simplicity, we assume the overall machine capacity can be represented by a single numerical value. Let $bw_{\text{norm}} (p_i, p_j)$ denote the normalized residual bandwidth from $p_i$ to $p_j$, and let $pc_{\text{norm}} (p_i)$ denote the normalized residual proxy capacity of $p_i$, then:

$$bw_{\text{norm}} (sp) = \min [bw_{\text{norm}} (p_0, p_1), bw_{\text{norm}} (p_1, p_2), \ldots, bw_{\text{norm}} (p_{n}, p_{n+1})]$$

$$pc_{\text{norm}} (sp) = \min [pc_{\text{norm}} (p_0), pc_{\text{norm}} (p_2), \ldots, pc_{\text{norm}} (p_{n+1})]$$

With traffic and proxy load balancing in mind, the goal of service routing is to find a path so that the bottleneck normalized bandwidth or the bottleneck normalized proxy capacity is the widest; i.e., $bw_{\text{norm}} (sp)$ or $pc_{\text{norm}} (sp)$ is maximized.

**proxy volatility (multiplicative):** Another metric relevant to QoS service routing is proxy volatility - probability of a proxy being down. Given that proxies may vary greatly in this respect, the objective is to find a service
path whose aggregate volatility is the lowest, so that the transmission will be most likely successful. Let $v(p_i)$ denote the volatility of proxy $p_i$, then:

$$v(sp) = 1 - \prod_{i=1}^{n}(1 - v(p_i))$$

Note that each proxy $p_i$ in $sp$ is counted only once for $sp$’s volatility, even if $p_i$ is visited more than once for different services. Minimizing $v(sp)$ amounts to maximizing $\prod_{i=1}^{n}(1 - v(p_i))$ (the probability of successful transmission), which follows the multiplicative composition rule.

### 3.2. Aggregate Performance Metric - $\mathcal{F}$

Many multimedia applications require that multiple performance metrics, instead of just one, to be optimized at the same time. A common approach to achieving multiple-metric optimization is to define a function and generate a single metric from multiple parameters. Let: (1) $p_{i-1}$ and $p_i$ denote two proxies onto which two consecutive services are mapped; (2) $d(p_{i-1}, p_i)$, $h(p_{i-1}, p_i)$, $bw_{norm}(p_{i-1}, p_i)$ denote, respectively, the delay, hop count, and normalized bandwidth between nodes $p_{i-1}$ and $p_i$; (3) and $v(p_i)$ and $pc_{norm}(p_i)$ denote, respectively, the volatility and normalized proxy capacity of $p_i$. We define our aggregate performance function as follows:

$$\mathcal{F}(p_{i-1}, p_i) = \frac{d(p_{i-1}, p_i) \cdot h(p_{i-1}, p_i) \cdot v(p_i)}{\alpha \cdot bw_{norm}(p_{i-1}, p_i) + (1 - \alpha) \cdot pc_{norm}(p_i)}$$

Note that if two services are located in one single proxy, i.e., $p_{i-1} = p_i$, then $h$ is always zero, otherwise, it’s one if the proxy topology is fully connected, or it’s the number of hops that separate $p_{i-1}$ and $p_i$ if the topology is a mesh or any non-fully connected ones. The reasoning behind the function $\mathcal{F}$ is: (1) delay, hop, and volatility are all metrics that we want to minimize, thus they are put at the upper part of the fraction, and since they are incomparable to each other, the most meaningful operation among them is multiplication; (2) normalized bandwidth and normalized proxy capacity are something that we want to maximize, thus they are put at the lower part of the fraction. They are comparable, because both are normalized values, thus summation can be a meaningful operation between them. The terms $\alpha$ ($0 \leq \alpha \leq 1$) and $(1 - \alpha)$ are used to adjust the weight of each metric in the aggregate function. The soundness of this function will be confirmed through simulations in Section 5. $\mathcal{F}$ is additive, thus $\mathcal{F}(sp) = \sum_{i=0}^{n}\mathcal{F}(p_i, p_{i+1})$. Our goal is to find a service path that minimizes $\mathcal{F}(sp)$.

### 4. QoS SERVICE ROUTING SOLUTION

In data routing, given a network topology, then a classic graph algorithm, such as the Dijkstra’s algorithm and its variants, can be applied to find an optimal network path between two nodes. However, given a proxy/service topology (a graph) and a service request (another graph), none of the existing graph algorithms can be applied directly to compute an optimal service path between two nodes, due to the service availability and dependency issues. The task of computing an optimal service goal can be accomplished by first completing a mapping process, which takes the proxy/service topology and the service request, and maps them into a directed acyclic graph (service DAG). In such a service DAG, any path that goes from the source node to the sink node satisfies the
service availability and dependency requirements, thus reducing the original complexity greatly. Once obtained
the service DAG with the correspondent node and link resource values, algorithms similar to those for QoS data
routing can be applied, albeit with a more complex resource checking process, to compute an optimal QoS service
path in terms of a given performance metric \( m \). We now proceed to describe these steps in detail.

4.1. Mapping

The mapping process takes two pieces of information, proxy/service topology and service request, and maps them
into a service DAG as shown in Figure 6. Detailed procedures of mapping are as follows (see also example in
Figure 6):

1. **instance finding** - Find, for each requested service, instances of it in the proxy overlay. For example, service
   \( s_1 \) of the service request has instances in three different proxies, \( p_1, p_2, \) and \( p_3 \). These instances are named
   \( s_1/p_1, s_1/p_2, \) and \( s_1/p_3 \). Recall that the notation “\( s_i/p_j \)” means “service \( s_i \) at proxy \( p_j \)”.

2. **connecting** - Let \( n_i \) be a node in the service DAG, we use \( o(n_i) \) to denote its original node in the service
   request. Thus \( o(-/p_0) = p_0 \), and \( o(s_1/p_1) = o(s_1/p_2) = o(s_1/p_3) = s_1 \). Connect node \( n_i \) to node \( n_j \)
   of the service DAG if \( o(n_i) \) is connected to \( o(n_j) \) in the service request. This is so because we assume the
   proxy topology is not partitioned, thus any proxy should be able to reach any other proxy either directly or
   indirectly.

3. **labeling and resource screening** - Label all related nodes/links with measured performance values available
   in the proxy/service topology, such as delay, hop count, residual bandwidth, residual proxy capacity, and
   proxy volatility, and screen out those nodes and links whose residual bandwidth and residual proxy capacity
   are less than those required by the services. When a node is screened out, then all of its in/out links should be
   also screened out. For clarity purposes, we only label the residual bandwidths and residual proxy capacities
   in Figure 6. As mentioned in Section 3, bandwidth could be CPU bandwidth or network bandwidth de-
   pending on whether or not two consecutive services are mapped onto a single proxy. In the figure, the CPU
   bandwidths are labels with value of 1000. If the proxy topology is fully connected, then the bandwidth from
   \( n_i \) to \( n_j \) is that of the logical link connecting \( \text{proxy}(n_i) \) to \( \text{proxy}(n_j) \), which is measured directly [9, 10].
   If the topology is not fully connected, the bandwidth from \( n_i \) to \( n_j \) should be the bottleneck bandwidth
   of the shortest path from \( \text{proxy}(n_i) \) to \( \text{proxy}(n_j) \). In parallel of labeling, nodes and links whose residual
   resources are lower than the required values should be immediately screened out. In Figure 6, nodes and
   links that do not meet the requirements are shown in dashed circles and dashed lines respectively.

---

\(^4\)Let \( n_i = s_j/p_k \), we say that \( \text{proxy}(n_i) = p_k \) and \( \text{service}(n_i) = s_j \).
Figure 6. The mapping process.
4.2. Routing Computation

Once the service DAG has been obtained, if it isn’t because of the *loop* problem (explained later), then conventional graph algorithms, such as the Dijkstra’s algorithm or DAG-Shortest-Paths algorithm, can be directly adopted to compute paths that optimize certain routing metric.

However, different from data routing, where paths are loop free, in service routing, loops are possible, because a single proxy may be visited recurrently for different services. Therefore, the resource screening step done in the mapping process does not guarantee that a path computed by using a conventional graph algorithm such as the Dijkstra’s algorithm, would contain sufficient resources. Taking Figure 7(a) as an example: if the final computed service path is \((\ldots, s_1/p_1, s_2/p_1, \ldots)\), then just knowing that \(p_1\) had enough resources at the mapping stage is not sufficient, because the resource checking was only done isolatedly; at \(s_1/p_1\), it was checked only whether or not the resources were sufficient to serve \(s_1\), and at \(s_2/p_1\), it was checked only whether or not the resources were sufficient to serve \(s_2\). However, in practice, either or both services can be mapped onto \(p_1\), and the residual resources may fail to meet the requirements in the latter case: when both services are mapped onto \(p_1\).

This indicates that the resource screening at the mapping stage is not sufficient. We thereby need to modify a base algorithm, such as DAG-Shortest-Paths algorithm\(^5\), to further perform a *resource checking* test each time a new node/link is relaxed. Taking *delay* as the routing metric, and \(u\) and \(v\) as two connecting nodes in the service DAG, we only perform \(\text{relax}(u, v, \text{delay})\) after verifying that \(v\)’s updated residual proxy capacity and \((u, v)\)’s updated residual bandwidth are both non-negative. The node \(v\)’s updated proxy capacity is calculated by backtracking to \(u\) and all \(u\)’s predecessors, and subtracting from \(v\)’s current proxy capacity, the amount of proxy resources that has been consumed by \(u\) and \(u\)’s predecessors. Similar procedure is done for bandwidth checking.

The extended DAG-Shortest Paths algorithm, which adds one line (shown in bold font) in the original algorithm [12], is presented below (note that \(\text{pred}(u)\) denotes the predecessor node of \(u\)):

\(^5\)We choose DAG-Shortest-Paths algorithm instead of Dijkstra’s algorithm due to its lower complexity on top of directed acyclic graphs.
DAG-SHORTEST-PATHS*(G, delay, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u taken in topologically sorted order
do for each vertex v \in Adj[u]
    if POSITIVE-RESOURCE-CHECKING ((\ldots, pred(pred(u)), pred(u), u, v))
        do RELAX(u, v, delay)

bool POSITIVE-RESOURCE-CHECKING((n_1, n_2, \ldots, n_{k-1}, n_k))
if (pc_{res}(n_k) - \sum_{i=1}^{k} pc_{req}(n_i)prxy(n_i) = prxy(n_k) \geq 0)
and if (bw_{res}(n_{k-1}, n_k) - \sum_{i=1}^{k-1} bw_{req}(n_i, n_{i+1})prxy(n_i) = prxy(n_{k-1}), prxy(n_{i+1}) = prxy(n_k) \geq 0)
then return true
else return false

4.3. Complexity Analysis of the Solution

Mapping: Let \( n_p \) denote the number of proxies in the proxy overlay, and let \( n_n \) and \( n_l \) denote, respectively, the numbers of nodes and links in the service graph (SG), then for each node in the SG, up to \( n_p \) service instances can be found, and for each link in SG, up to \( n_p^2 \) links can be formed in the service DAG. Thus, the number of nodes in the service DAG - \( V \) - can be written as \( V = O(n_n \ast n_p) \), and the number of links in the service DAG - \( E \) - can be written as \( E = O(n_l \ast n_p^2) \). The complexity of labeling the resource values of the nodes and links depends on how the proxy topology is structured. If the topology is considered a fully connected graph, then performance values of, for instance, bandwidth and delay between two neighboring services, are directly measured and obtained. In this case, labeling a single link or node’s performance value takes constant time. However, in case the proxy topology is not fully connected (e.g., a mesh), then the delay between two neighboring services should be the aggregate delay.
of all links that make up the shortest path between the two nodes, and the bandwidth between two neighboring services should be the bottleneck bandwidth of all links on the shortest path. Both values can be derived using algorithms such as Dijkstra, Bellman-Ford, or Floyd-Warshall, whose performance is no larger than $O(n^3)$\(^6\). The worst case complexity for the service mapping process thus has complexity of $O(V + E + n^3_p)$.

**Routing Computation Using DAG-Shortest-Paths* Algorithm:** The complexity of applying the DAG-Shortest-Paths* algorithm on top of the service DAG, whose number of nodes is $V$ and whose number of links is $E$, is dominated by the for loops. Let $l$ denote the longest service configuration in SG, then the complexity of the back-tracking resource checking process inside the inner for loop is $O(l)$, (the resource checking can back track up to $l$ nodes/links). The total complexity of DAG-Shortest-Paths* is $O((V + E) * l)$.

## 5. PERFORMANCE EVALUATION

In this section, we do simulation tests to: (1) measure the performances of $\mathcal{F}$ in terms of its individual metrics; (2) compare the performances of the integrated approach against those of the two-phase approach in [4]. Simulations are done in ns2.

We assume a fully connected proxy topology, where each proxy monitors its own node and link conditions actively, and reports the results to other proxies in the system periodically. We consider an overlay topology of 20 proxies, where each proxy is assigned a random amount of capacity, and each link is assigned a random amount of bandwidth. Each proxy has a set of locally available services, and has certain volatility associated with it. We roughly assume the ratio of CPU bandwidth to network bandwidth to be 100.

### 5.1. Study 1: Performances of $\mathcal{F}$

In this study, we concentrate on evaluating the performance of our aggregate metric $\mathcal{F}$ in aspects of its individual metrics: delay, hop count, normalized bandwidth, normalized proxy capacity, and proxy volatility, by comparing with performances of two cases: best and random. In the best case, the path computation only seeks to optimize one of the single metrics (such as delay or hop count). In the random case, a service path is chosen randomly. Note that in all cases, the found paths always satisfy the service and resource requirements, the difference lies in their optimization metrics (in the random case, optimization metric is none). We run 20 test cases; with each test case consisting of 200 randomly generated client requests. All 20 test cases have the same environment settings, e.g., in terms of initial proxy capacities and bandwidths. Service requests arrive randomly with a maximum inter-arrival time of 60s, and each established service path may remain active for up to 30 minutes.

Figure 8 shows the performances of $\mathcal{F}$ in terms of individual metrics; the values for each test case are averaged over the results obtained for 200 client requests. We see that $\mathcal{F}$’s individual performances range between best and random. In most cases, they are closer to best than random, which indicate that $\mathcal{F}$ is a sound aggregate

---

\(^6\)Let $n$ and $e$ be the number of nodes and number of links, respectively, in the graph. Both Dijkstra and Bellman-Ford are single-source shortest paths algorithms, and takes $O(n^2)$ and $O(ne)$ of time. Floyd-Warshall is an all-pair shortest paths algorithm, and takes $O(n^3)$ of time.


Figure 8. The performance values of $F$ compared with those of the best and random cases.

function. Only in the figure of “normalized capacity” is $F$’s performance comparable to random. This is so because, by nature, $F$ tends to map consecutive services onto a single proxy in order to reduce hop count and network bandwidth usage. Since not all metrics can be optimized at the same time, the merit comes at a price of compromising proxy capacity balancing in these cases. We say that proxy capacity balancing is compromised in these cases because when two consecutive services have chances of mapping onto a single proxy ($h = 0$), then the value of $F$ becomes zero, independent of how large or how small the value of $pc_{norm}$ is at the lower fraction of $F$.

5.2. Study 2: Integrated Approach vs Two-Phase Approach

In Section 1, it was clear that decoupling resource configuration selection from service path finding may yield sub-optimal paths. In this section, we concentrate on showing the performance differences between the integrated approach and the two-phase approach. In our simulation, each service graph may contain somewhere between three and eight service configurations. In the two-phase approach, we first select a “resource-shortest” path out of the service graph according to [4], and then compute a service path that optimizes $F$. Figure 9 shows the performance results of the two approaches, where the $x$-axis shows the number of the test cases, and the $y$-axis shows the $F$ value averaged over 200 client requests. The integrated approach yields better overall performance than the separate approach. Note that in order to make the comparisons fair, in all of the simulation tests conducted so far, we made the available resources at proxies and networks sufficiently large, so that all path findings are successful.

We also compare the path finding success rates of two approaches in faces of resource scarcity. We set lower amounts of resources in the proxies and networks, and let service paths remain active for longer periods of time, so that resources may become used up at some times. We also ran 20 test cases, each consisting of 1000 client requests. Figure 10 shows that integrated approach clearly yields better path finding success rates. These simulation
results indicates that service configuration selection should not be decoupled from service path finding.

6. CONCLUSIONS

In this paper, we have presented an integrated, global-view approach for computing QoS service paths, which shows promising performance results. The main advantages of our global-view approach is that service path finding can be done quickly, without flooding the network. One important observation that we made is that selection of service configuration should not be decoupled from service path finding. The contributions of this paper is three-fold: (1) the paper presented a generic approach to solving the QoS service routing problem through an additional mapping process; (2) it presented a sound performance aggregate function that tries to optimize several routing metrics at the same time; and (3) it showed that service configuration selection and service path finding should be an integrated process.

References


