sFlow: Towards Resource-Efficient and Agile Service Federation in Service Overlay Networks

Mea Wang, Baochun Li, Zongpeng Li
Department of Electrical and Computer Engineering
University of Toronto
{mea, bli, arcane}@eecg.toronto.edu

Abstract

Existing research work towards the composition of complex federated services has assumed that service requests and deliveries flow through a particular service path or tree. In this paper, we extend such a service model to a directed acyclic graph, allowing services to be delivered via parallel paths and interleaved with each other. Such an assumption of the service flow model has apparently introduced complexities towards the development of a distributed algorithm to federate existing services, as well as the provisioning of the required quality in the most resource-efficient fashion. To this end, we propose sFlow, a fully distributed algorithm to be executed on all service nodes, such that the federated service flow graph is resource efficient, performs well, and meets the demands of service consumers.

1. Introduction

Services have emerged as one of the main motivations to construct application-layer overlay networks. The concept of *services* in service overlay networks is not specific to certain categories, and is in fact quite generic. Nodes in overlay networks may process data (filtering, computation or media transcoding services), relay data (proxy or query forwarding services), store data (storage services) or search for data (peer-to-peer search engines). In most cases, the consumers demand complex services that require the *federation* or *composition* of multiple types and instances of services in overlay networks. The result of a service federation is referred to as a *federated service*. The most resource-efficient service federation minimizes the network and computing resources requirements.

Our original contribution is to propose *sFlow*, a fully distributed and application-independent algorithm to federate service instances in a resource efficient fashion and according to the needs of service consumers, even when the services are required to serve in a non-sequential order, and may be characterized by a directed acyclic graph. Using the directed acyclic graph as the model for service flow graphs has one additional benefit: it generally leads to superior performance, in terms of bandwidth and service latency. These benefits are validated by our extensive results in performance evaluations.

Towards the direction of end-to-end service federation, Gu *et al.* [1] and Xu *et al.* [5] have proposed QoS-aware algorithms to find point-to-point *service paths*. In particular, the algorithm proposed by Xu *et al.* is designed for the highly connected service mesh generated by cost-effective mesh augmentation methods. Beyond the concept of service paths that sequentially connect services, the recent concept of *service multicast trees* [3] makes it possible to create multiple paths with shared services merged, effectively forming a multicast tree. Instead of computing the service paths in a centralized fashion, Jin *et al.* [2] have proposed an distributed service federation algorithm. In this work, the service overlay network is first organized into a cluster network. The service path finding algorithm is then applied hierarchically in a divide-and-conquer fashion.

Nonetheless, these existing algorithms require services performing tasks consecutively to meet requirements with respect to resources. This type of service federation is only effective for a limited range of applications, such as multimedia transcoding and streaming. In more generic applications, as in our example, multiple services may perform tasks independent of each other, and services may be interconnected in a more arbitrary fashion. In this paper, we seek to address the more generic cases of federating services in service overlay networks, where the relationship among services may be characterized as a *directed acyclic graph* (DAG), referred henceforth as the *service flow graph*. In a service flow graph, federated services may perform tasks in either a sequential, parallel, or interleaved fashion as necessary.

The remainder of this paper is organized as follows. In Sec. 2 and 3, we present salient properties of service federation in a graph-theoretic setting, and provide theoretical insights to these problems. *sFlow*, our algorithm towards resource-efficient service federation, is presented in Sec. 4. Sec. 5 presents performance evaluation results of the dis-

Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)
1063-6927/04 $20.00 © 2004 **IEEE**

tributed algorithm. The paper is concluded in Sec. 6.

2. Towards Resource-Efficient and Agile Service Federation: Preliminaries

2.1. Service Requirements

Before we present the general model of service requirements, we first consider the example in Fig. 1, where the source service (Travel Engine) needs to send the requested data to Agency A via the Hotel service. This example illustrates the most primitive form of service requirements, where a single chain of services, referred to as the *service path* is specified.

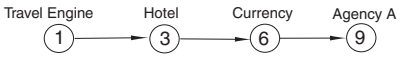

Figure 1. Service path: a basic form of service requirements

Service requirements may also be more flexible by allowing optional services, as the example illustrated in Fig. 2, which takes two alternative graphical representations. The source service (Travel Engine) may send the requested data to the Agency A service via a chain of the Attraction Service and either the Map or the Translator services. The topology of services that leads to better performance is preferably selected.

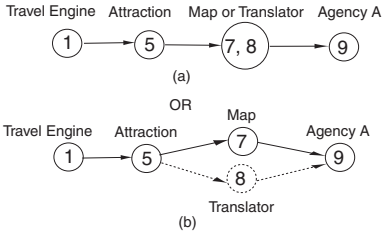

Figure 2. Optional services: an enhanced form of service requirements

It is rather restrictive to require that services be chained sequentially. We further improve the model of service requirements to allow a number of *disjoint service paths*, which execute service tasks concurrently. These disjoint service paths do not share any particular service except the source and the sink services. An example of disjoint service paths is shown in Fig. 3, where the source service (Travel Engine) sends airline, hotel, and attraction information to the Agency A service in three disjoint paths.

Nonetheless, the results of a particular service may be required by more than one downstream services, while it may also be required to integrate the results of existing services in order to provide a specific downstream service. In the previous example, service results (*e.g.,* price and location) of the Hotel service may feed into both the currency and map services before delivering to Agency A. We seek to model service requirements beyond simple service paths, leading to the model of *directed acyclic graphs*, the most generic and realistic form. In subsequent sections, we discuss this model in depth, which is one of the main contributions of our paper.

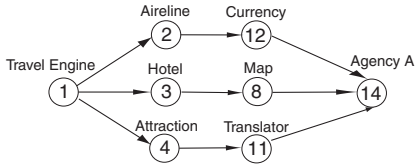

Figure 3. Disjoint service paths in service requirements

2.2. Service Federation: Preliminaries

In our model, we assign each node in the underlying network a unique *node identifier* (NID). Instead of distinguishing services by their names, we assign each service a *service identifier* (SID). A service may have multiple *service instances*. For example, Delta Airlines and Northwest Airlines are two service instances of the Airline service.

A service overlay network may be presented as an *overlay graph* $G(V, E)$, which reflects the topologies of services in the overlay network. In such an overlay graph, each node represents a *service node*, and is denoted by V_i for $i = 1, \ldots, n$ and $n = |V|$. Two services are *compatible* if the output produced by one service matches the input requirements of the other service. Two service nodes can be linked by an edge if they are compatible and there exists a path between them in the underlying network. This edge is called a *service link* and is denoted by E_i for $i = 1, \ldots, m$ and $m = |E|$. The direction of the service link indicates the direction of the *service flow*, also referred to as the *service stream*. In an overlay graph, a service consumes the output produced by its *upstream service* in order to provide input for its *downstream service* or the end users.

Fig. 4 illustrates an *overlay graph* over a typical underlying network. Each node in the underlying network is labeled with a NID. Each node in the overlay graph is a service instance of a particular service and is labeled with an appropriate SID. In this paper, we will use the terms *service node* and *service instance* interchangeably. Service nodes of the same service may share the same SID, and they are distinguished by their NIDs. An edge exists between each pair of compatible service nodes and is labeled with its character-

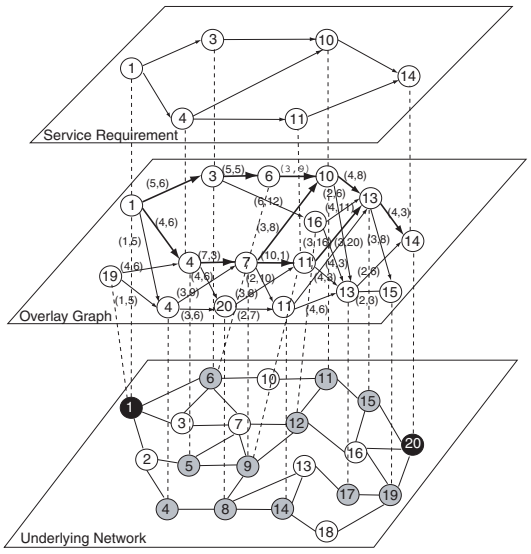

Figure 4. Modeling service overlay graphs in application overlay networks

istic performance metrics, such as bandwidth and latency.

The simplest form of service federation is the *service path*, in which services perform tasks one after another sequentially in delivering the desired service to the end users. In Fig. 4, the federated service from service 1 to service 14, via services 3, 6, 10, and 13, is an example of a service path. Each service, except the source and sink service, has exactly one upstream service and one downstream service.

A slightly more general form of service federation is the service multicast tree. In Fig. 4, the result of service 1 may reaches two groups of end users via service 14 and 15. A multicast tree may be constructed by merging multiple service paths that share a subset of common services. The root of the tree is the source service; and the leaf nodes are the sink services. Each intermediate service may have more than one downstream service, but only one upstream service.

The quality of the federated service required by the user may include the specification of a set of elementary or primitive services, as well as the service relationships among them. More formally, a *service requirement* is a graph $R(V^R, E^R)$ consisting of all the required services, which includes one source service node, at least one sink service node, and a set of intermediate service nodes. The edges in a service requirement specify the sequence that the services should be performed during the federation process. The direction of each edge indicates the direction of the service flow. Fig. 4 shows a complete illustration from the service requirement to the underlying network.

Given our model of service overlay networks, the fundamental and open problem with respect to the construction of service overlay graphs is as follows. *We wish to select one particular node for each required service (uniquely identified by the service identifiers), such that the selected nodes form a high-quality topology that satisfy the service requirement.* Such a service topology is henceforth referred to as the *service flow graph*. For example, there exist 8 different paths from service 4 to service 11 in Fig. 4. We choose node 5 over node 4 for service 4, and node 9 over node 14 for service 11, because they offer a service flow graph with higher overall bandwidth and shorter end-to-end latency. In Fig. 4, the selected service flow graph is highlighted by the darker lines in the overlay graph.

In this paper, we consider two resource metrics: bandwidth and latency. We adopt the distributed algorithm proposed by Wang and Crowcroft [4] based on link states that finds the *shortest widest* paths. With this algorithm, the *widest* path, or the path with the highest end-to-end bandwidth, is selected; and if there are more than one widest paths, the *shortest*, or the one with the lowest end-to-end latency, is then selected.

3. sFlow: Theoretical Insights

In this paper, we study a generic form of service federation, the *service flow graph*, which includes but is not limited to service paths and service trees. Services may perform tasks in a sequential, parallel, or interleaved fashion. For instance, in Fig. 5, it is not necessary for the Translator service to wait for the Map and the Currency services to be completed before translating the attraction information. In this section, we present the concept of the service flow graph, and investigate the problem of constructing the service flow graph from a theoretical point of view.

3.1. The service flow model based on directed acyclic graphs

In practical cases of service federation, end users should have the freedom to request a federated service of any form. Therefore, a service requirement should not be restricted to combinations of service paths, as have been shown. Instead, multiple service streams may be *merged* at a *merging service node*; and a particular service stream may also be *split* into multiple paths at a *splitting service node*. In Fig. 5, we show one example of such service requirements.

We now present the formal definition of the *service flow graph*, $G'(V', E')$, as a subgraph of the given overlay graph $G(V, E)$. The service flow graph takes the form of a *directed acyclic graph*, and contains one source service V_s, at least one sink service V_d, and a number of intermediate services V_i, for $i = 1, \ldots, n$ and $i \neq s, d$. In contrast to service paths and trees, the service flow graphs allow each intermediate service to have multiple upstream services and multiple downstream services. This provides greater flexi-

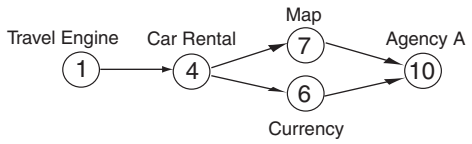

Figure 5. Generic service requirements: an example

bility in the service federation process, in that services may be federated in any fashion, and may assist in the optimization of performance, such as throughput and latency.

In an overlay graph, there are multiple service instances corresponding to each service specified in a service requirement. The service flow graph must consist of exactly one instance of each required service specified in the service requirement. It may also need to include other service instances that bridge two required services. Here, we define a *service abstract graph* to connect the service requirement to the overlay graph. Each node in the service abstract graph, known as a *service abstract node*, represents a required service in the service requirement. This node is further populated with service instances of its corresponding service. Two service instances are linked by an edge whenever there is an edge between the corresponding required services in the service requirement. As an result, service instances of one service abstract node are fully connected to the those of another service abstract node, if there exists an edge between these two services in the requirement.

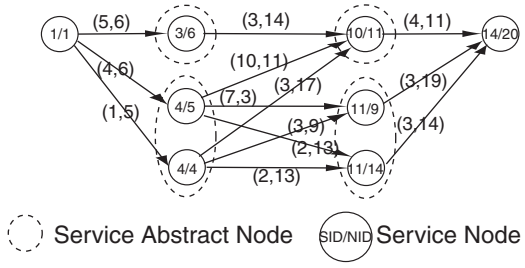

Figure 6. Service abstract graph: an example

Fig. 6 presents the service abstract graph of our previous example shown in Fig. 4. All edges in the service abstract graph are labeled with bandwidth and latency of the shortest-widest path between two service instances in the overlay graph, which can be easily computed using the Wang-Crowcroft algorithm [4] based on known weights in the overlay graph.

With the introduction of the service abstract graph, we may further investigate the complexity of constructing optimal service flow graphs with respect to their performances.

3.2. Towards optimal service flow graphs

In general, we show the unfortunate result that, constructing the most high-performance service flow graph for a specific service requirement is a NP-Complete problem. We prove the NP-Completeness property of this problem by reducing a well-known NP-complete problem, the *SAT Problem*, to a special case of our problem, the *Maximum Service Flow Graph Problem*. By the notion of *maximum service flow graphs*, we mean that the overall bandwidth of the service flow graph is maximized, or bounded by an upper limit. It is a well known fact that the overall throughput is equivalent to the bandwidth on the bottleneck link, since the bottleneck provides pressure for flow control towards both upstream and downstream directions. By the definition of shortest-widest paths, bandwidth takes precedence over latency when evaluating the quality of links. Therefore, proving the NP-completeness of the *Maximum Service Flow Graph Problem* is sufficient to show that finding the optimal-quality service flow graph is NP-complete.

Definition 1: Given a directed acyclic graph $G = (V, E)$ where $V = \{v_1, v_2, \cdots, v_n\}$ and $v_i = s_i = \{v_i^1, v_i^2, \cdots, v_i^{m_i}\}$ for all $1 \leq i \leq n$ and $m_i = |v_i|$, $e = \{v_i^a, v_j^b | 1 \leq a \leq m_i, 1 \leq b \leq m_j\} \in E$ for $1 \leq i, j \leq n$ and $i \neq j$, weight $w(e) \in Z^+$, and a positive integer K, the **Maximum Service Flow Graph Problem** is to find a service flow graph $G' = (V', E')$ in this graph with minimum weight among all edges $\min(w(e_1), w(e_2), \ldots, w(e_{|E'|})) \geq K$, where $V' = \{v_1^{i_1}, v_2^{i_2}, \cdots, v_n^{i_n} | 1 \leq i_1 \leq m_i, 1 \leq i_2 \leq m_2, \cdots, 1 \leq i_n \leq m_n\}$.

Theorem 1: The Maximum Service Flow Graph Problem is NP-complete.

Proof: It is easy to see that the Maximum Service Flow Graph Problem $\in$ NP. We shall show that the *SAT Problem* $\propto$ the *Maximum Service Flow Graph Problem*.

Given an instance of the *SAT Problem*, collection $C = \{c_1, c_2, \cdots, c_n\}$ of clauses on a finite set $U = \{u_1, u_2, \cdots, u_m\}$, we shall construct an instance of the *Maximum Service Flow Graph Problem*.

Let each clause, c_i, corresponds to v_i in the directed acyclic graph G. The construction of graph $G(V, E)$ involves the local replacement of the basic element v_i, for $1 \leq i \leq n$. We replace each node v_i by a set of nodes $\{v_i^1 ... v_i^{m_i}\}$ where v_i^k corresponds to the kth literal in clause c_i and $m_i = |c_i|$. Every pair of nodes with the form $\{v_i^a, v_j^b\}$, where $i \neq j$, $1 \leq a \leq |c_i|$, and $1 \leq b \leq |c_j|$, is connected by an edge. This step connects all nodes to each other, except the nodes corresponding to the literals in the same clause. Now we label each edge $e = \{v_i^a, v_j^b\}$ with $w(e) = 1$ if node v_i^a and node v_j^b correspond to p and $\overline{p}$, otherwise, let $w(e) \geq 2$. At the end, we add directions to each edge. Starting at v_1, we make v_1 the source/root node, such that all edges are outgoing edges. We then make

each undirected edge of v_2 an outgoing edge. Repeat this step from v_2 to v_n. This makes v_1 the source node and v_n the sink node. Let $K = 2$. It is easy to see that this transformation takes polynomial time. Fig. 7 visualizes the transformation process with an example, $U = \{x, y, z, w\}$ and $C = \{\{x, y, \overline{z}, \overline{w}\}, \{\overline{x}, \overline{y}, \overline{z}\}, \{x, \overline{y}, \overline{w}\}, \{\overline{y}, z\}\}$. The weight of each edge is indicated by its darkness. The darker edges have $w(e) = 1$, and the normal edges have $w(e) \geq 2$.

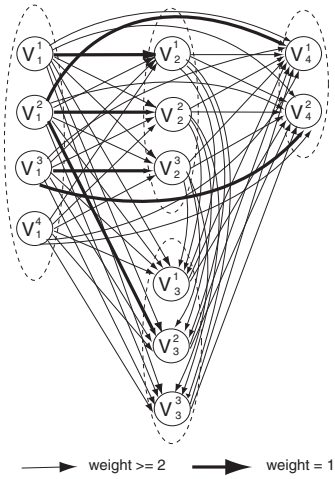

Figure 7. An example of transformation

Now, consider the graph $G(V, E)$ and the integer K we have just obtained. A service flow graph with a minimum edge weight greater than or equal to K must have one node from each v_i. According to the transformation described above, each v_i corresponds to a clause, and each node in v_i corresponds to a literal in that clause. We set the literals corresponding to the chosen node to true, that is, setting exactly one literal in each clause to true, and then set the rest of the variables randomly to true or false. Since the minimum edge weight is greater than or equal to K, only the edges with weight greater than or equal to 2 can be selected. In other words, nodes corresponding to p and $\overline{p}$ must not be selected at the same time. Since this assignment guarantees that at least one literal in each clause is true, the assignment must be a satisfying truth assignment in the *SAT problem*.

Conversely, if there exists a satisfying truth assignment in the *SAT problem*, at least one literal in each clause is set to true. We randomly choose one of the truth literals from each clause, and select the corresponding nodes in the graph G. These nodes form a service flow graph with a minimum weight equal to K, since a literal and its complement should never exist in the truth assignment at the same time, that is, all bottleneck links are avoided. Therefore, the Maximum Service Flow Graph Problem is NP-complete. □

Given that the Maximum Service Flow Graph Problem is NP-complete, an algorithm that targets the optimal solution of the shortest-widest service flow graph may not do much better than the brute-force exhaustive search. However, there exists a polynomial time algorithm for a special case of the problem, which serves as the baseline algorithm that leads to the fully distributed *sFlow* algorithm. We first present such a baseline algorithm to solve the problem in the special case, and then present intuitions towards heuristics that incorporates such baseline algorithms to solve more complex problems.

3.3. The Baseline Algorithm

In the special case where the service requirement is a single service path, we show that there still exists a polynomial time algorithm for finding the optimal-quality service flow graph. Given a single-path service requirement, our *baseline algorithm* that computes the optimal service flow graph is shown in Table 1.

Baseline algorithm	
1	Compute the all-pairs shortest-widest path by computing the shortest-widest path from each node to every other nodes in the overlay graph using the Wang-Crowcroft algorithm.
2	Construct the service abstract graph using the service requirement, as shown in Sec. 3.1;
3	Compute the shortest-widest abstract path from the source service to each sink service in the service abstract graph;
4	Replace each edge in the service abstract path with the actual shortest-widest path between two consecutive service instances in the service abstract graph.

Table 1. The baseline algorithm

Since the time complexity of the Wang-Crowcroft algorithm to compute shortest-widest paths [4] is shown in [4] to be $O(N^2)$, the all-pairs shortest-widest paths computation takes $O(N^3)$ time. Step 2, 3 and 4 takes at most $O(N)$ time. Therefore, the time complexity of the baseline algorithm is also $O(N^3)$, where N is the number of nodes in the overlay graph.

3.4. Heuristics

Based on the baseline algorithm presented in Table 1, we seek to develop heuristics to construct optimal service flow graphs satisfying generic service requirements. Towards this objective, we attempt to *reduce* complex service requirements to more primitive service requirements. Such reductions may hopefully lead to single service paths in the service requirement, which may be solved by applying the polynomial-time baseline algorithm.

3.4.1. Path Reduction Strategy Even for service requirements with a slightly more complex topology, we may still be able to reduce them to simple service paths. As an example, the service requirement in Fig. 8(a) may be split into three requirements, shown in Fig. 8(b). After such a reduction, we can certainly find the local optimal service flow graph for the two single-path requirements using our baseline algorithm. The remainder of the requirement topology requires another heuristic — the *Split-and-Merge Reduction*, that we will introduce next. Since the baseline algorithm guarantees the optimality of delivering main service streams in each single-path service requirement, the overall service flow graph for the given service requirement has the best quality within an acceptable degree of approximation. Since each part of the service flow graph is of optimal quality corresponding to a disjoint service path in the service requirement, the merged graph is also an optimal quality service flow graph.

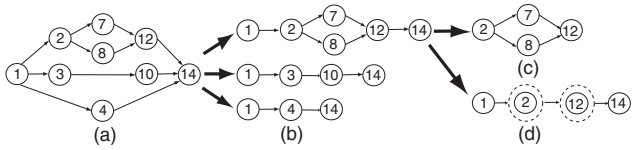

Figure 8. Path reduction: an example.

3.4.2. Split-and-Merge Reduction Strategy More complex service requirements allow the *splitting* and *merging* of service streams, where a service may feed into multiple downstream services, or consume outputs from multiple upstream services. We show that a split-and-merge topology may be identified and isolated from the rest of the service requirement, and once the performance metrics are determined from the splitting node to the merging node (possibly by further reductions), it can be replaced with one single edge from the splitting node to the merging node. Fig. 8 continues to show our example from Fig. 8. The split-and-merge topology in the original service requirement (Fig. 8(b)) can be isolated from the rest of the requirement, and replaced by an edge between the splitting node 2 and the merging node 12. After such an isolation and replacement process, the reduced topologies can be represented by both multiple disjoint paths (Fig. 8(c)) and a single service path (Fig. 8(d)). The former may be further reduced to two service paths by applying the path reduction strategy previously outlined. The optimality of each reduced requirement leads to the approximate optimality of the overall service flow graph.

It is clear that these reduction strategies are best-effort heuristics, and do not guarantee the successful reduction of arbitrary generic service requirements to single service paths. However, these heuristics lead to insights that we use in the design of the distributed sFlow algorithm, and they are applicable to a wide range of service requirements. In Sec. 4, we present the sFlow algorithm that computes service flow graphs in a fully distributed fashion.

4. sFlow: a Distributed Algorithm for Service Federation

In this section, we seek to design a fully distributed algorithm, referred to as *sFlow*, to construct service flow graphs that satisfy the provided service requirements, and optimize performance at the same time. For this purpose, we apply theoretical insights and heuristics developed in Sec. 3 to compute locally optimal service flow graphs on each service node, based on its knowledge of the local neighborhood in the overlay graph. We will use the example shown in Fig. 9 as a running example throughout our explanations in this section, which uses the same service requirement as our previous example in Fig. 4.

To mark the starting point of the service federation process, the user delivers the service requirement, to the source service node by sending a *sFederate* message. As the source service node receives the *sFederate* message and the embedded service requirement, it first seeks to analyze its local overlay graph and locate the immediate downstream service nodes in its overlay graph.

As an example, examine Fig. 9(a), where node 1, as the source service node, splits the service requirements into two simpler requirements, with two immediate downstream services, 3 and 4. Fig. 9(a) also shows the local overlay graph that node 1 has the knowledge of, assuming that all service nodes are aware of the portion of the overall overlay graph within a two-hop vicinity in the overlay graph. All service nodes in the local overlay graphs are labeled with their SID/NID tuples. Operating with such a local overlay graph, it is straightforward for node 1 to execute our baseline algorithm and reduction heuristics, in order to compute the most efficient partial service flow graph, as a subgraph of the local overlay graph. The result of such computation is shown at the bottom of Fig. 9(a).

As node 1 has obtained its locally computed partial service flow graph, it promptly forwards it to its immediate downstream nodes in the graph, embedded in new *sFederate* messages. In our example shown in Fig. 9(a), the two downstream service nodes are nodes 5 and 6, carrying services 3 and 4, respectively. Carried by the same *sFederate* message, node 1 also forwards the service requirement to its downstream service nodes. Compared to the original service requirement received at each node, the service requirement that it forwards to its downstreams does not include service on this node itself, since all computations involving node 1 has already been considered in the partial service flow graph it generates.

As the nodes 5 and 6 receive the *sFederate* message, they execute precisely the same sFlow algorithm, yet with a sim-

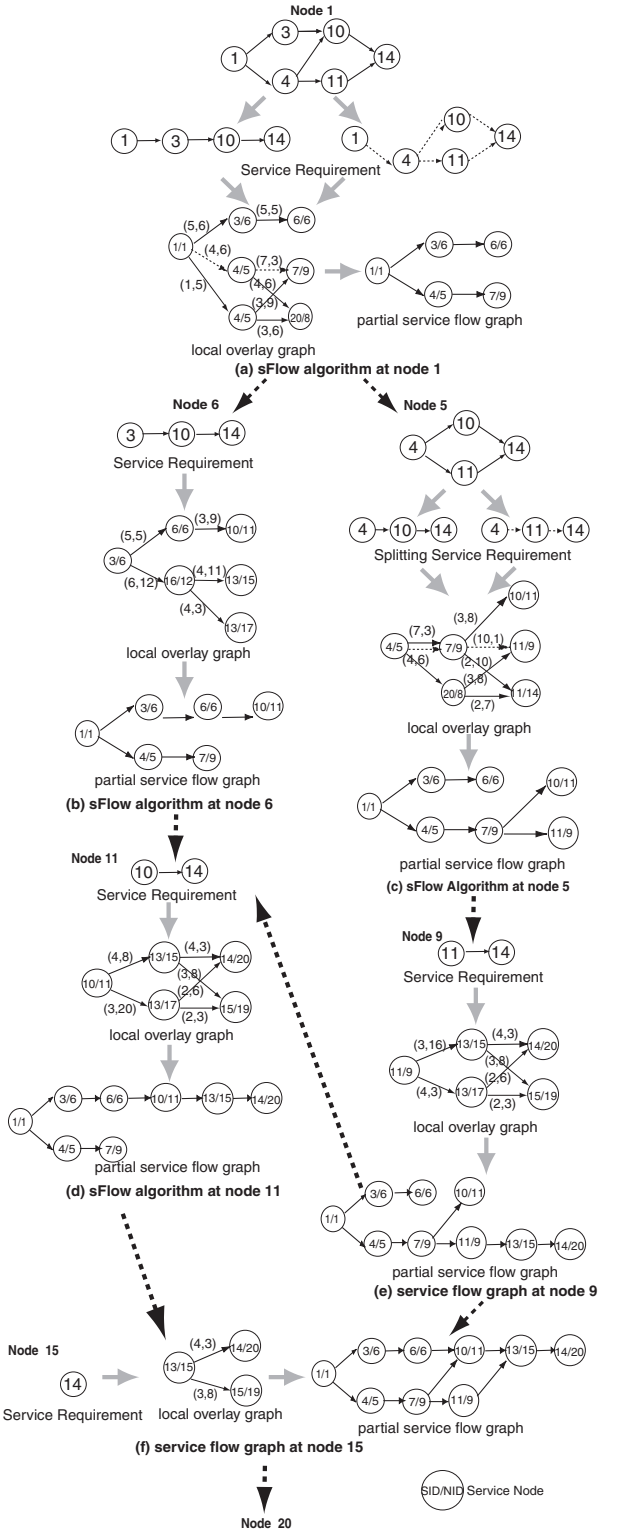

Figure 9. The sFlow algorithm in action: an example.

pler service requirement. For example, node 5 receives a single-path service requirement $3 \rightarrow 10 \rightarrow 14$, which can be easily solved with our baseline algorithm. Node 5, on the other hand, receives a slightly more complex service requirement, that can easily be solved by first applying the split-and-merge reduction strategy, and then the baseline algorithm. As the distributed sFlow algorithm progresses, more downstream service nodes will receive the *sFederate* message, and eventually the complete service flow graph will be computed and finalized on the sink service node, which is node 20 in our example.

We note that, even though the path reduction strategy is used explicitly on each service node to compute its own partial service flow graph, the split-and-merge strategy may also be applied implicitly by the distributed sFlow algorithm, since the tasks of computing optimal service flow graphs are generally assumed by the splitting node.

5. Evaluation

We have performed a simulation-based study to evaluate the effectiveness of sFlow. In the simulation, we have implemented the sFlow algorithm in C++ in a local host, while all network communications are simulated using the event-driven simulation methodology. For comparisons, we have implemented three alternative heuristic algorithms as control: the *random* algorithm, the *fixed* algorithm, as well as the single service path algorithm. The *random* algorithm randomly chooses a direct downstream in the local overlay graph that leads to the corresponding downstream required in the service requirement. The *fixed* algorithm always chooses the direct downstream with the highest available bandwidth that leads to the corresponding downstream service in the service requirement. The single service path algorithm is identical to the end-to-end service federation algorithm previously proposed by Gu *et al.* [1].

In our simulation, the sink service node creates service requirements of any type, and starts the service flow graph computation as described in Sec. 4. The overall service flow graph is collected at the source service node. We have executed the sFlow algorithm in networks of sizes 10, 20, 30, 40, and 50 respectively, to observe how the sFlow algorithm scales with network size. We also computed the global optimal resource-efficient service flow graph, and used it as the benchmark to evaluate the scalability and correctness of the sFlow algorithm.

To verify the correctness of the sFlow algorithm, we compare the service flow graph computed by all four service federation algorithms, with fixed global optimal resource-efficient service flow graph. We define the *correctness coefficient* as the ratio between the number of matching nodes in the two service flow graphs and the total number of nodes in the global optimal graph. The coefficient should be greater than zero and less than or equal to one. As the coefficient ap-

proaches one, the service flow graph is closer to the global optimal service flow graph.

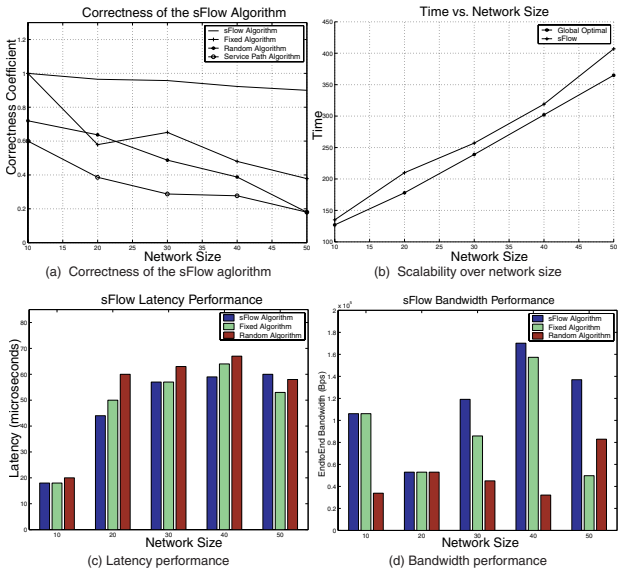

Figure 10. Scalability and performance

As shown in Fig. 5(a), the sFlow algorithm outperforms the other three algorithms. This figure shows that the correctness of the service flow graph is comparable with the global optimal service flow graph. The service path algorithm has the lowest success rate, since it can only handle the simplest service requirements. The success rate of the random algorithm is about 50%, since it always randomly choose one instances out of many candidates of the required services. The fixed algorithm has high success rates only when the optimal service flow graph contains all the links with the highest bandwidth. As the network size increases, each service node has less global overlay network information, which causes the service flow graph to fall below global optimality. Fig. 5(a) shows that the locally optimized service flow graph guarantees a correctness coefficient of 0.9 or larger.

We now verify the scalability of the sFlow algorithm. Since the complexity of the sFlow algorithm is $O(N^3)$, it is scalable in theory. We further verify the scalability by the computation time. The results are presented in Fig. 5(b). Since there is no polynomial time algorithm for finding the optimal service flow graph for non-simple service requirements, we use only simple requirements in order to make reasonable comparison between the sFlow algorithm and the global optimal algorithm. As the network size increases, the computation time increases gradually, as expected. Since the global optimal service flow graph is computed once at the sink node, its computation time is slightly less than that of the sFlow algorithm. The time difference between the two lines is due to re-computation time introduced at certain service nodes. Generally speaking, the distributed sFlow algorithm does not introduce significant amount of computation overhead, and performs in a scalable fashion.

The major advantage of the service flow graph approach over the traditional service path approach is that, the former allows services to be involved in any fashion. It turns out that, this flexibility also leads to better delay performance. In Fig. 5(c), we compare delay latency introduced in the service flow graph algorithm with that in the fixed and random algorithms. The sFlow algorithm supersedes the service path algorithm since the latter fails to consider the parallel processing cases. As indicated in Fig. 5(d), the sFlow algorithm consistently produces service flow graphs with higher end-to-end throughput, regardless of the network size. To conclude, among the different algorithms we have studied, the sFlow algorithm produces service flow graphs with the best quality.

6. Concluding Remarks

In this paper, we have presented a generic form of service federation that allows independent services to perform tasks in any appropriate fashion. We have proposed *sFlow*, a fully distributed, scalable, and flexible service federation algorithm for finding a resource efficient service flow graph in a service overlay network. Our simulation results have verified the correctness and the scalability of the *sFlow* algorithm over different network size. As we have expected, the end-to-end latency and overall bandwidth in the service flow graph is significantly better than the traditional service path. Given the results we have presented, we claim that the sFlow algorithm is a scalable, flexible, and reliable service federation algorithm for generic service requirements in any generic network settings, and is able to produce service flow graphs with high qualities.

References

[1] X. Gu and K. Nahrstedt. A Scalable QoS-Aware Service Aggregation Model for Peer-to-Peer Computing Grids. In *Proc. of 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[2] J. Jin and K. Nahrstedt. Large-Scale Service Overlay Networking with Distance-Based Clustering. In *Proc. of ACM/IFIP/USENIX International Middleware Conference (Middleware)*, June 2003.

[3] J. Jin and K. Nahrstedt. On Construction of Service Multicast Trees. In *Proc. of IEEE International Conference on Communications (ICC)*, May 2003.

[4] Z. Wang and J. Crowcroft. Quality-of-Service Routing for Supporting Multimedia Applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1228–1234, 1996.

[5] D. Xu and K. Nahrstedt. Finding Service Paths in an Overlay Media Service Proxy Network Dongyan Xu. In *Proc. of SPIE/ACM Conf. on Multimedia Computing and Networking (MMCN)*, January 2002.