

Service Composition for Advanced Multimedia Applications

Jin Liang and Klara Nahrstedt
University of Illinois at Urbana-Champaign
{jinliang, klara}@cs.uiuc.edu

ABSTRACT

By composing distributed, autonomous services dynamically to provide new functionalities, service composition provides an attractive way for customized multimedia content production and delivery. Previous research work has addressed various aspects of service composition such as composibility, QoS-awareness, and load balancing. However, most of the work has focused on applications where data flow from a single source is processed by intermediate services and then delivered to a single destination. In this paper, we address the service composition problem for advanced multimedia applications where data flows from multiple content sources are processed and aggregated into a composite flow, which is then delivered to one or more destinations, possibly after being customized for each receiver. We formally define the problem and prove its NP hardness. We also design a heuristic algorithm to solve the problem. Our algorithm has the following attractive features: (1) it is effective at finding low cost composition solutions; (2) it has the ability to trade off computation overhead for better results; (3) it is efficient and can scale to relatively large number of network nodes and component services.

Keywords: Service composition, Quality of Service (QoS), algorithm, simulation

1. INTRODUCTION

The Internet is becoming increasingly service oriented, with various application services being deployed by different providers. As a result, service composition, the process of dynamically composing existing services to provide new functionalities, has emerged as a cost-effective way to deploy new Internet services.

Service composition not only enables the reuse of existing services, but provides an attractive way for *dynamic production* and *customized delivery* of multimedia contents. On today's Internet, on the one hand, multimedia contents are gaining increasing popularity. On the other hand, users are accessing the contents using widely different client devices (e.g., laptops or cell phones) and under different network connectivities (e.g., wired or wireless). To allow multimedia contents to be properly viewed by the user, some intermediate services must be applied to customize the contents based on the particular client device and network connectivity.

Figure 1(a) shows an example where a user with a handheld PC wants to stream a video from a server. Due to the mismatch of the video source and the client capacity (e.g., different codecs supported, limited client bandwidth or processing power), a transcoding service must be applied to the video stream before it is delivered to the client. Thus, the video streaming service has been composed with the transcoding service to provide a customized video streaming service. Figure 1(b) and 1(c) show two applications that are more interesting. In Figure 1(b), the video streams from two different servers are mixed into one. Next the translated news from a third server is embedded into the mixed video stream, which is then delivered to the user. In Figure 1(c), the composite stream is produced in a similar way, but the stream is delivered to more than one client. Due to the heterogeneity of the client devices, some transcoding service may again be applied to customize the composite stream for the individual receivers. Clearly for all applications shown in Figure 1, the ability to select the appropriate service instances and compose them dynamically, according to user requirements, is crucial for the success of these applications.

Previous research work has addressed various aspects of service composition such as composibility,¹ QoS-awareness^{2,3} and load balancing.⁴ However, most of the work has focused on application scenarios similar to Figure 1(a), and the results are not readily applicable to the other more interesting scenarios. In this paper, we address the service composition problem for advanced multimedia applications such as those shown in Figure 1(b) and 1(c). We focus on the QoS aware service selection and routing aspect of the composition problem, namely, given the set of services available in the network, some of which may be replicated at different places, how to

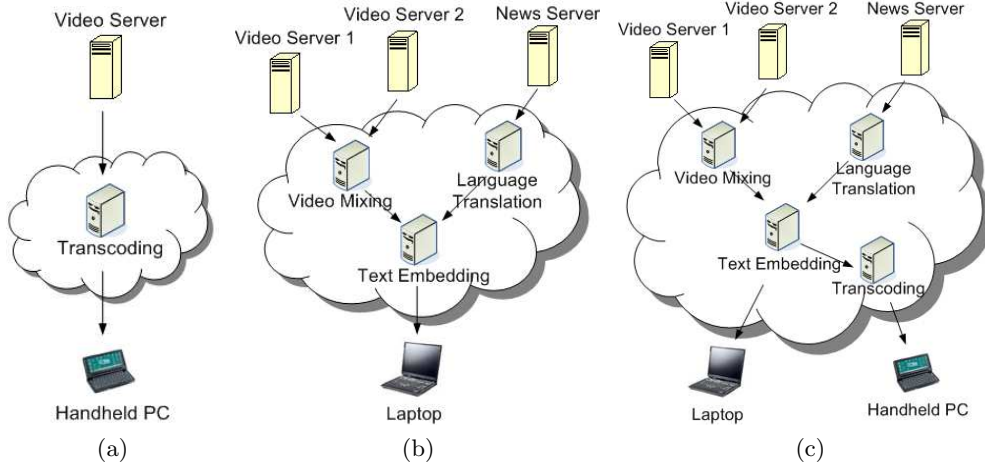


Figure 1. Example advanced multimedia applications

select the required service instances and route the data flows through the instances, so that the QoS requirements of the applications are satisfied, and the “cost” of the composed service is minimized. We formulate the problem as finding the smallest cost mapping from the functional service specification graph to the service overlay network and prove that it is NP hard. We also design a heuristic algorithm to solve the problem. Our algorithm has the following attractive features. (1) It is effective at finding low cost service composition solutions under normal network conditions. (2) It is able to trade off computation overhead for better results when network resources are scarce. (3) It is efficient and can scale to large number of network nodes and component services. These are verified by running the algorithm on different synthetic network topologies.

In the rest of the paper, we first formulate the service composition problem for advanced multimedia applications and prove its NP hardness in Section 2. We then provide a heuristic algorithm to solve the problem in Section 3. Section 4 presents the simulation results. Section 5 discusses research related to our work and concludes the paper.

2. THE SERVICE COMPOSITION PROBLEM

2.1. Service overlay network model

We adopt a service overlay network model similar to previous work on service composition.^{2,4} Component services are deployed in different *sites*. Each site can be a single or a cluster of computers. The sites connect with each other and form a service level overlay network. Each service can be *replicated*, which means multiple instances of the same service could be deployed on different sites. In our model, we use a directed graph $G = (V, E)$ to represent the service overlay network. Each service site is represented by an overlay node. V is the set of overlay nodes and E is the set of overlay links between the nodes. $nn = |V|$ is the number of nodes and $ne = |E|$ is the number of links. We assume each node keeps track of the services deployed on the node. If component service q is deployed on node v , we say service q is *available* on node v . To allow services not adjacent to each other in the overlay network to be composed, we assume there is a *no-op* service on each node similar to previous work.⁴ This service does not change the data stream. It just forwards data from an upstream node to a downstream node.

There is a cost associated with each overlay node and link in the graph. The cost represents how well a particular node or link is suited for a service request. For example, a heavily loaded node or a wide area link may have a large cost, so that a composed service can avoid using such nodes and links. We use $c_n(v)$ and $c_l(u, v)$ to denote the cost for using node v and link (u, v) , respectively. The goal of our service composition is to minimize the cost of a composed service, so that efficient network resource usage is achieved. Each node and link in the overlay network also has an availability value, which represents the amount of computation/communication resources that is available on the node/link. We use $a_n(v)$ and $a_l(u, v)$ to denote the available resources on

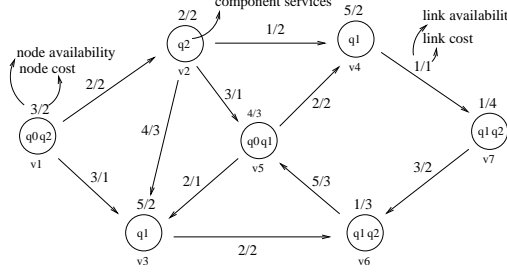


Figure 2. Example service overlay network topology

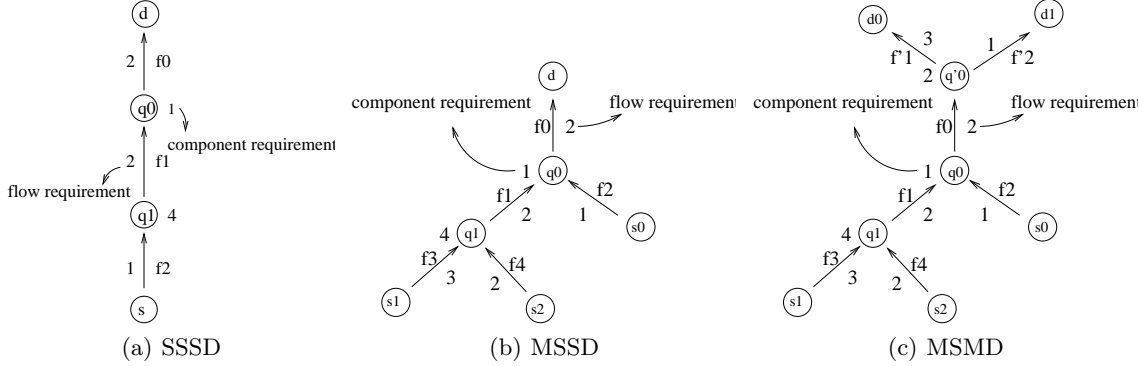


Figure 3. Example service specification graph

node v and link (u, v) , respectively. We assume the nodes in the overlay network can exchange information with their neighbors using a link state style protocol, so that the network topology, the deployed services on each node, and the cost and availability information are known to all the nodes. Figure 2 shows an example service overlay network annotated with the node and link availability and cost values. Each overlay node also shows the services that are available on the node.

2.2. Service request specification

To compose a new service, the user (or one of the users if there is more than one) must provide a *specification* of the required service. The specification is represented using a directed graph $H = (S \cup Q \cup D, F)$. $S \subset V$ is the set of content sources in the overlay network, Q is the set of component services that need to be included, $D \subset V$ is the set of receivers (destinations), and F is the set of data flows between the component services. $nq = |Q|$ is the number of required component services and $nf = |F|$ is the number of data flows. Since we are considering multimedia applications, each service component and data flow has certain QoS requirement. We assume this requirement can be translated into the amount of computation and communication resources (processing bandwidth and network bandwidth) needed for the particular service component or data flow. So long as the resource requirement of each service service component and data flow is satisfied, the QoS of the composed service is guaranteed. We use $r_q(q_i)$ and $r_f(f_i)$ to denote the processing bandwidth required by service component q_i and the network bandwidth required by data flow f_i , respectively.

The topology of a service specification graph H is important to the service composition algorithm. In this paper we consider three classes of service specification graphs (corresponding to the three application scenarios in Figure 1).

- 1 Single source, single destination (SSSD). The service specification graph consists of a chain of component services that need to be applied to a single source flow one after another. The final flow is delivered to a single destination. Suppose there are nq components, the first to be applied is denoted q_{nq-1} , the last one to be applied is denoted q_0 . The output data flow of component q_i is denoted f_i , and the source data flow

is denoted f_{nq} . If f_j is an input flow of q_i , q_i is called the *next service component* of f_j , and f_i is called the *next flow* of f_j . Figure 3(a) shows a service specification graph for the SSSD case.

- 2 Multiple source, single destination (MSSD). The specification graph H is a tree. The root of the tree is the destination. The leaves are the content sources, and the internal nodes are the component services that are needed. The output flow of component q_i is denoted f_i . The output flow of source s_i is denoted f_{nq+i} . If f_j is an input flow of q_i , then q_i is called the *parent service component* of f_j , and f_i is called the *parent flow* of f_j . Figure 3(b) shows an example of MSSD service specification.
- 3 Multiple source, multiple destination (MSMD). The service specification graph H can be thought of as the concatenation of two trees, the *aggregation tree* and the *dissemination tree*. The leaves of the aggregation tree are the content sources. The leaves of the dissemination tree are the destinations. Data flows from the sources are first processed and aggregated by the aggregation services in the aggregation tree, the composite data flow is then processed by the dissemination services in the dissemination tree and then delivered to the destinations. The output flow of an aggregation service q_i is denoted f_i . The input flow of a dissemination service q'_i is denoted f'_i . f_0 and f'_0 are the same flow. If f_j is an input flow of an aggregation service q_i , q_i is called the *parent service component* of f_j , and f_i is called the *parent flow* of f_j . If f'_j is the output flow of a dissemination service q'_i , q'_i is called the *parent service component* of f'_j , and f'_i is called the *parent flow* of f'_j . An example MSMD service specification graph is shown in Figure 3(c).

2.3. Problem definition

Given the service overlay network graph $G = (V, E)$ and the service specification graph $H = (S \cup Q \cup D, F)$, we define a *composition solution* to be a mapping from H to G so that (1) each required component service $q \in Q$ is mapped to an overlay node $v \in V$. q is said to be *initiated* on node v in the composed service. Of course to initiate service q on node v , q must be available on node v ; (2) each flow $f \in F$ is mapped to a path P in the overlay network. If f is from a service q_i to q_j , which are initiated on node v_i and v_j , respectively, then P is a path from v_i to v_j in the overlay network. If f is from a source s to q_i (i.e., a source flow), then P is a path from s to v_i in the overlay network. If f is from q_j to a destination d , then P is a path from v_j to d in the overlay network. If f is mapped to path P , f is said to *pass through* P in the composed service.

Since we are considering multimedia applications, we require each composition solution to satisfy the resource constraints of the network nodes and links. Let Q_v denote the set of component services that are initiated on node v , and $F_{u,v}$ denote the set of flows that pass through link (u, v) . The resource constraints of the network nodes and links are satisfied if for each node v in the network $\sum_{q \in Q_v} r_q(q) \leq a_n(v)$, and for each link (u, v) in the network $\sum_{f \in F_{u,v}} r_f(f) \leq a_l(u, v)$. This means the processing requirement of the set of services initiated on node v should not exceed the available processing bandwidth on node v , and the network bandwidth requirement of the set of flows that pass through link (u, v) should not exceed the available bandwidth on link (u, v) .

If a service q is initiated on node v , the cost incurred by q is $c_n(v)$. If a flow f is mapped to a path P , the cost incurred by f is $\sum_{(u,v) \in P} c_l(u, v)$. The cost of a composition solution is the sum of the costs incurred by all the component services and the data flows.

Now we can define the *service composition (SC) problem* as given the network topology G and service specification graph H , find the composition solution that has the smallest cost among all composition solutions.

2.4. NP hardness of the service composition problem

We have the following theorem about the service composition problem.

Theorem 1. *The service composition problem is NP hard.*

Proof. Note that the service specification graph for the SSSD case is a chain, which is a special case of the MSSD and MSMD scenarios. Therefore we only need to prove that the SSSD service composition problem is NP hard.

To prove the SSSD service composition problem is NP hard, we consider the corresponding decision problem. Given the service overlay network G , the service specification H , and an integer n , the decision problem asks if there is a composition solution that has a cost no more than n .

We show that the decision problem is NP hard by reducing the NP complete bin packing problem to it. The bin pack problem is as follows. Suppose we have a number of bins each has a capacity of C , we also have n objects each has a size of $s(i)$ (for $i = 1, 2, \dots, n$). Given an integer k , the bin packing problem asks if the n objects can be packed into k bins. Given an instance of the bin packing problem, we construct an instance of the SSSD service composition problem as follows. We create a complete graph with k nodes as the service overlay network. The available processing bandwidth of each node is C . The available network bandwidth of each link is infinity. The cost of each node is 1 and the cost of each link is 0. The service specification graph H consists of a chain of n component services, each of which is replicated on every node. Component service i has a required processing bandwidth of $s(i)$. The network bandwidth requirement of each flow is 0 (or a small number). The source and destination nodes are arbitrarily chosen from the k nodes. And we ask if there is a composition solution with cost at most n . The construction can be done in polynomial time.

Clearly if there is a composition solution to the service composition problem with cost n , it means the n component services have been successfully mapped to the k nodes, which means the n objects can be packed into the k bins in the bin packing problem. Conversely, if the n objects can be packed into k bins, the corresponding mapping of component services to nodes constitute a composition solution with cost n in the service composition problem. This shows that the service composition problem is NP hard. \square

3. HEURISTIC ALGORITHM

In this section, we present a heuristic algorithm to solve the service composition problem. We first describe the algorithm for the SSSD service specification in Section 3.1 and analyze the algorithm in Section 3.1.1. We then extend the algorithm to the MSSD and MSMD cases in Section 3.2 and Section 3.3, respectively.

3.1. Heuristic algorithm for the SSSD case

We use a greedy heuristic algorithm similar to the Dijkstra's shortest path algorithm to solve the SSSD service composition problem.

From Section 2.3, we know the composition solution for an SSSD composition problem is a path (not necessarily a simple path), because it is the concatenation of the path segments to which the data flows are mapped. Therefore a composition solution to the SSSD service composition problem is called a *solution path*.

To represent a solution path, and more importantly the prefix of a solution path (which is the partial mapping of the service specification graph H to the overlay network G), we define a *labeled partial path* $P(v)$ to be a path $\{v_0(=s)^{l_0}, v_1^{l_1}, \dots, v_k(=v)^{l_k}\}$ from source s to node v . Each node v_i in the path is associated with an *outgoing flow* whose flow number is l_i . It means the data flow that leaves node v_i is f_{l_i} . Alternatively, it means component services $q_{nq-1}, q_{nq-2}, \dots, q_{l_i}$ have been initiated along the path from s to v_i . Apparently we should have $l_i \geq l_{i+1}$ for $0 \leq i < k$. If $l_i > l_{i+1}$, it means the components services $q_{l_i-1}, q_{l_i-2}, \dots, q_{l_{i+1}}$ are initiated on node v_{i+1} . Similar to the solution path, we require a partial path to satisfy both service and resource constraints, it means a service is only initiated on nodes where it is available, and the resource capacity of the network nodes and links are not exceeded.

If the outgoing flow of the last node in the partial path $P(v)$ is f , we may write the partial path as $P(v, f)$ and say that it ends at node v with an outgoing flow f . If a partial path ends at the destination node d with the final flow f_0 as its outgoing flow, then it is a solution path.

A partial path may not be a simple path, because a node can be visited multiple times, each time for a different service. However, it is meaningless if a node appears twice in a partial path with the same outgoing flow, because this corresponds to a loop in the overlay network for the same data flow. This means a node can appear at most nf times in the partial path, and the length of a partial path is at most $nm \cdot nf$.

For each node u and flow f , there could be multiple partial paths that end at u with an outgoing flow f , each represents a different partial mapping of H to G , and each may have a different cost. Therefore, we use a *label* $lb(u, f)$ to keep track of the smallest cost partial path that ends at u with an outgoing flow f . Initially the value of $lb(u, f)$ is infinity and its state is *inactive*. When a partial path $P(u, f)$ is first discovered, the value of the label is set to the cost of $P(u, f)$ and its state set to *active*. If more partial paths are discovered that end at u with an outgoing flow f , $lb(u, f)$ is updated to reflect the one with the smallest cost.

```

GreedySolve
  active_Labels.Initialize()
  while active_Labels.IsNotEmpty()
    lb = active_Labels.GetMinLabel()
    lb.Finalize()
    if IsDone(lb) return lb.PartialPath
    UpdateLocal(lb.u, lb)
    for each neighbor v of lb.u
      UpdateNeighbor(lb.u, v, lb, lb.c + c_l(lb.u, v))
    end for
  end while
  return NULL

```

Figure 4. Greedy heuristic algorithm

A new partial path is discovered by “growing” existing partial paths. There are two ways that a partial path can be grown. Suppose we have a partial path $P(u, f_i)$, if we let the data flow pass through a link (u, v) to a neighbor v , we may be able to grow the partial path “horizontally” to $P(v, f_i)$. If the next component service q_{i-1} is available on node u , we may also initiate the service on node u and grow the partial path “upward” to $P(u, f_{i-1})$. Of course to grow the partial path to $P(v, f_i)$, there must be enough network bandwidth on link (u, v) . And to grow the partial path to $P(u, f_{i-1})$, there must be enough processing bandwidth on node u . Since node u and link (u, v) may have been used in $P(u, f_i)$ for previous services and data flows, such resource usage must be accounted when growing a partial path.

Since a new partial path is discovered by growing existing partial paths, if a label $lb(u, f)$ has the smallest value among all active labels, we know its value can not be further updated and hence it is *finalized*.

Our algorithm is shown in Figure 4. The algorithm maintains a set of active labels. Initially the set contains only one label corresponding to the source node and source flow. After the initialization, the algorithm repeatedly selects the active label with the smallest value, finalizes it and grows the associated partial path. This is continued until either the label $lb(d, f_0)$ is finalized (in this case, *IsDone()* evaluates to true and the corresponding partial path $P(d, f_0)$ is returned as the solution path), or the set of active labels becomes empty (in this case, the algorithm declares failure to find a solution).

Readers familiar with the Dijkstra’s shortest path algorithm may have seen the similarity between it and our algorithm. However, our algorithm is different from the shortest path algorithm in several aspects. First, each node maintains nf labels, instead of just one. Second, when a label is finalized, we need to call *UpdateLocal()* to try to initiate the next service on the local node, and update the corresponding label. This is not present in the shortest path algorithm. Third, both *UpdateLocal()* and *UpdateNeighbor()* need to check resource availability before updating a label on the local node or on a neighbor node (the resource check is straightforward to do and is not detailed here). Fourth, as we will see in the next subsection, the “subpath optimality” of the shortest path algorithm no longer holds for our problem, therefore, for each node u and flow f , our algorithm actually maintains more than one candidate partial solution, in order to trade off computation overhead for better composition results.

3.1.1. Analysis of the algorithm

Clearly if our algorithm returns a partial path, then it is a solution path, because the resource constraints of the partial paths are maintained all the time. However, it is possible that the returned solution path is not the optimal one, or there exists a solution path but the algorithm failed to find it. This is because for each node u and flow f , we only keep the smallest cost partial path $P(u, f)$. However, it is possible that a non-optimal partial path $P'(u, f)$ can lead to the optimal solution path, while $P(u, f)$ cannot. For example, for the service specification in Figure 3(a) and the network topology in Figure 2 (suppose v_1 is the source and v_4 is the destination), $P(v_5, f_1) = \{v_1^2, v_2^2, v_5^2, v_5^1\}$ has a smaller cost ($c_l(v_1, v_2) + c_l(v_2, v_5) + c_n(v_5) = 6$) than

$P'(v_5, f_1) = \{v_1^2, v_3^2, v_3^1, v_6^1, v_5^1\}$ ($c_l(v_1, v_3) + c_n(v_3) + c_l(v_3, v_6) + c_l(v_6, v_5) = 8$). However, the former cannot lead to any solution path, while the latter can lead to the optimal solution path.

Based on the above observation, it may be desirable to keep more than one candidate partial path (label) for a given node u and flow f , because a non-optimal partial path may later lead to an optimal solution path. Since there could be an exponential number of partial paths for a given (node, flow) pair, it is impossible to keep all of them. Therefore, our algorithm keeps the nc smallest partial paths discovered thus far. The number of candidate partial paths nc provides a way to tradeoff the computation needed for better results. Every time a new partial path is discovered, it is accepted if it belongs to the nc smallest partial paths discovered thus far.

Now we analyze the time complexity of the algorithm. The time complexity depends on the data structure for the active label set. There are three functions that operate on the active label set. In the main body of the algorithm in Figure 4, $GetMinLabel()$ is called to extract the smallest label in the active label set. In $UpdateLocal()$ and $UpdateNeighbor()$, when a new partial path is discovered, a new label needs to be added to the label set, and an old label needs to be removed from the label set. In our implementation, we have used a double linked list to store the active labels. As a result, to remove a label from the set takes $O(1)$ time. To add a label to the set, we just append the new label at the end of the list, so it also takes $O(1)$ time. However, since the list is unordered, for $GetMinLabel()$ we must search through the list to find the label with the smallest cost.

Suppose there are nn nodes in the overlay network, nf flows in the application topology graph, and we maintain nc candidate partial paths for each (node, flow) pair, then there could be at most $nn \cdot nf \cdot nc$ labels. As a result, the while loop of the main algorithm may execute $O(nn \cdot nf \cdot nc)$ times.

For each execution, $GetMinLabel()$ takes $O(nn \cdot nf \cdot nc)$ time, because it must traverse the whole list of active labels. $UpdateLocal()$ takes $O(nn \cdot nf)$ time because it needs to check the resource usage of a partial path to determine the available resource on a node (remember the length of a partial path is bounded by $O(nn \cdot nf)$). $UpdateNeighbor()$ also takes $O(nn \cdot nf)$ time because it needs to check the resource available on a link, and this is done by examining the resource usage of the partial path at hand. Suppose there are ne links and nn nodes in the service overlay network, then each node has ne/nn neighbors on average. Therefore each time a label is finalized, $UpdateNeighbor()$ may be called ne/nn times. As a result, the time complexity of the whole algorithm is

$$O(nn \cdot nf \cdot nc \cdot (nn \cdot nf \cdot nc + nn \cdot nf + \frac{ne}{nn} \cdot nn \cdot nf)) = O((nn \cdot nf \cdot nc)^2 + nn \cdot ne \cdot nf^2 \cdot nc).$$

If nc is fixed to a small constant, and the average node degree in the overlay network is constant, then the complexity of the algorithm is $O(nn^2 \cdot nf^2)$.

3.2. Heuristic algorithm for the MSSD case

We use the same heuristic algorithm shown in Figure 4 to solve the MSSD service composition problem, but there are several changes that must be made.

Since the service specification graph H is a tree for MSSD, we can thought of the composition solution as a tree (called a *solution tree*), where the network nodes to which the service components are mapped are internal nodes of the tree, and the network paths to which the flows are mapped connect these internal nodes.

To represent a (partial) solution tree, we define a *labeled partial tree* $T(v, f)$ to be a tree in the overlay network rooted at v . Each node in the tree is labeled with a flow number, and the flow is called the *outgoing flow* of the node. The outgoing flow of the root node (f) is also called the outgoing flow of the partial tree.

If node w is the parent of node v in $T(u, f)$, the outgoing flow of w is f_i and the outgoing flow of v is f_j , then either $f_i = f_j$, or f_i is the parent flow of f_j . In the latter case, it means the parent component service of f_j is initiated on node w . Similar to the SSSD case, we require a partial tree to satisfy the service and resource constraints, which means a service is only initiated on overlay nodes where it is available, and the available resource on a node or link is not exceeded. Just as the partial path for the SSSD case may not be a simple path, the partial tree may not be a tree on the overlay network, because a node may be visited multiple times, each time for a different service. However, it is meaningless if a node appears twice in the partial tree with the same outgoing flow, because it corresponds to a loop in the overlay network for the same data flow. This means a node can appear at most nf times and therefore there are at most $nn \cdot nf$ nodes in a partial tree.

Similar to the SSSD case, we maintain a set of active labels, each label $lb(v, f)$ tracks the smallest cost partial tree that is rooted at node v with an outgoing flow f . We keep growing the partial trees and updating the labels until either a solution tree is finalized, or the active label set becomes empty.

There are two main differences from the SSSD case. First, since there are multiple sources, initially there are multiple partial trees, each rooted at a source node, with the corresponding source flow being the outgoing flow. Second, when we want to grow a partial tree “upward” by initiating a service q_i on node u (assume q_i is available on u), If q_i has more than one input flow, then we must make sure that for each input flow f_j , a partial tree $T(u, f_j)$ exists on the node and the corresponding label $lb(u, f_j)$ is finalized. Further, we must make sure that the resource usage of these partial trees are compatible with each other, which means when they are combined into a partial tree $T(u, f_i)$, the resource constraints are still satisfied. This involves checking resource usage on every node and link, not just node u . The above changes are implemented by modifying *Initialize()* and *UpdateLocal()*.

The time complexity of *UpdateLocal()* is affected by the above changes. However, if we assume that the average node degree in the network is a constant, then the time complexity is still $O(nn \cdot nf)$, which means the time complexity for the whole algorithm is still $O(nn^2 \cdot nf^2)$.

3.3. Heuristic algorithm for the MSMD case

Since the service specification graph H for the MSMD case can be thought of as the concatenation of two trees, the aggregation tree and the dissemination tree, we leverage the algorithm for the MSSD case to solve the MSMD service composition problem. In fact, if we reverse the direction of the flows in the dissemination tree, we may have two separate MSSD composition problem, except that the two trees must meet at some rendezvous point.

In our algorithm, we maintain both *partial aggregation trees* and *partial dissemination trees*. The partial aggregation trees originate from the sources and are combined by initiating aggregation service components. The partial dissemination trees originate from the destinations and are combined by initiating dissemination service components. If a partial aggregation tree and a partial dissemination tree are finalized on the same node, both have the composite flow as their outgoing flow, and their resource usages are compatible with each other, then the two partial trees are returned as the final solution.

Since the MSSD service composition problem is solved as two MSSD problems, the time complexity is the same as the MSSD problem.

3.4. Illustration of the heuristic algorithm

In this subsection, we use an example to show how the heuristic algorithm works. We use the service overlay network as shown in Figure 2 and the service specification graph as shown in Figure 3(b). We assume the source nodes s_0, s_1 and s_2 are v_6, v_1 and v_2 , respectively, the destination is v_5 . We set $nc = 1$, which means we only keep one candidate partial tree for each (node, flow) pair.

Initially, we have three active labels, $lb(v_6, f_2)$, $lb(v_1, f_3)$ and $lb(v_2, f_4)$, each has a value of 0. The labels are shown in Figure 5(a). The corresponding partial trees are shown under the labels.

After the initialization, the algorithm selects the label with the smallest value. Suppose $lb(v_6, f_2)$ is selected, the algorithm finalizes the label, and calls *UpdateLocal()* to attempt to deploy the parent service component of f_2 . Since the parent service component q_0 is not available on node v_6 , no new partial trees are discovered. The algorithm will also call *UpdateNeighbor()* for each neighbor. Since v_5 is the only neighbor of v_6 , and (v_6, v_5) has enough bandwidth for flow f_2 , the partial tree $T(v_6, f_2)$ is grown to $T(v_5, f_2)$, which creates a new label $lb(v_5, f_2)$ with a value of 3 (the cost for using link (v_6, v_5)). After the three labels in Figure 5(a) are finalized, the active labels are as shown in Figure 5(b). Note since link (v_1, v_2) does not have enough bandwidth for flow f_3 , the partial tree $T(v_1, f_3)$ cannot grow to v_2 . Similarly $T(v_2, f_4)$ cannot grow to v_4 .

Next, from the active label set shown in Figure 5(b), label $lb(v_3, f_3)$ is selected and finalized. Since link (v_3, v_6) does not have enough bandwidth for flow f_3 , no neighbors are updated. The parent component service q_1 is not initiated, because it has another input flow f_4 , and the label $lb(v_3, f_4)$ has not been finalized. Next, label $lb(v_5, f_4)$ is selected and finalized, this creates a new label $lb(v_4, f_4)$ and updates an old label $lb(v_3, f_4)$ (the value is changed from 3 to 2). Now the active labels are as shown in Figure 5(c).

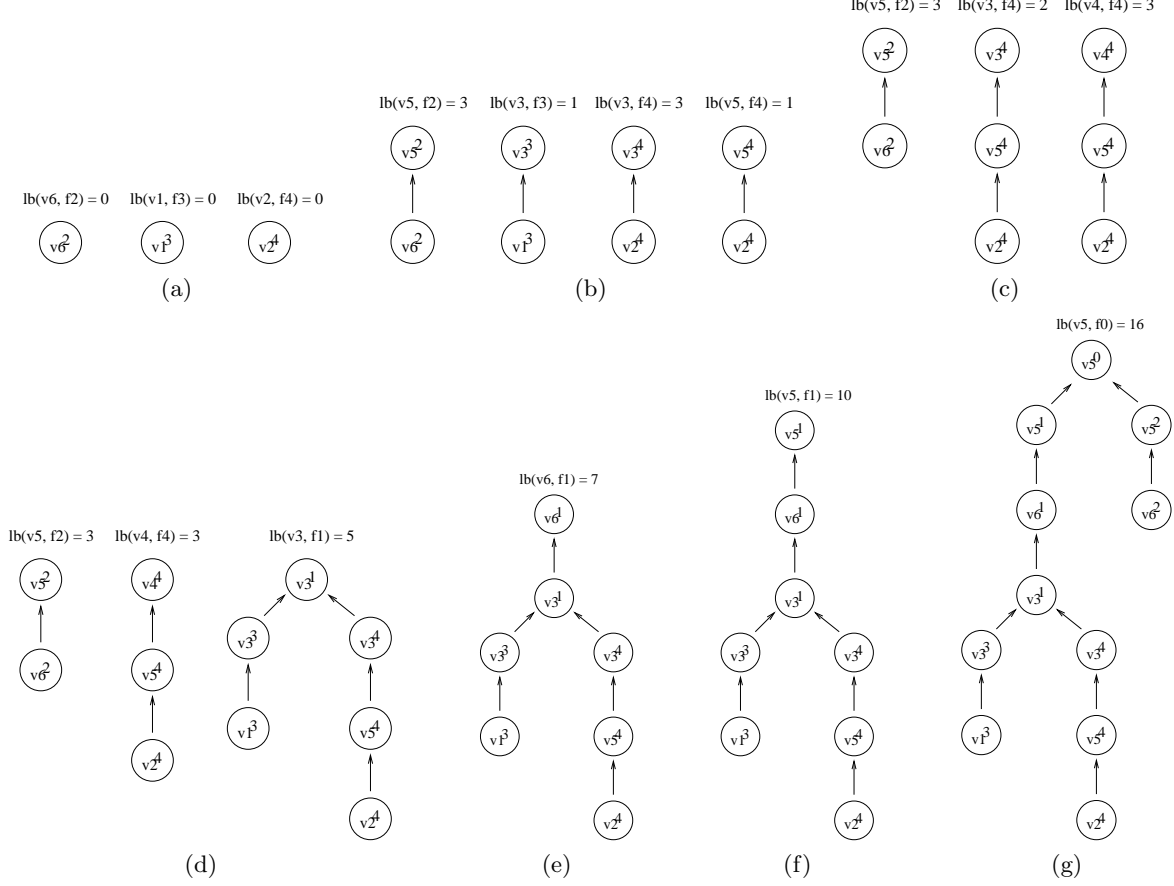


Figure 5. Illustration of how the heuristic algorithm works. Each figure shows the set of active labels and the corresponding partial trees at different stages of the algorithm.

Continuing Figure 5(c), label $lb(v_3, f_4)$ is finalized. This causes the parent component service q_1 to be initiated. As a result, a new label $lb(v_3, f_1)$ is created, and the corresponding partial tree is the combination of $T(v_3, f_3)$ and $T(v_3, f_4)$. And the value of $lb(v_3, f_1)$ is the sum of the cost of $T(v_3, f_3)$, $T(v_3, f_4)$, and the cost for using node v_3 , which is $1 + 2 + 2 = 5$. The set of active labels now are as shown in Figure 5(d).

Next, labels $lb(v_5, f_2)$ and $lb(v_4, f_4)$ are finalized, without any new partial paths found. $lb(v_3, f_1)$ is then finalized, creating a new label $lb(v_6, f_1)$, which is the only active label in the system and is shown in Figure 5(e).

When $lb(v_6, f_1)$ is finalized, a new label $lb(v_5, f_1)$ is created, which is shown in Figure 5(f). When $lb(v_5, f_1)$ is finalized, the parent component service q_0 is initiated, which combines two partial trees $T(v_5, f_1)$ and $T(v_5, f_2)$ and creates the label $lb(v_5, f_0)$. Note the link (v_6, v_5) is used in both trees. However, since the combined bandwidth requirement of f_1 and f_2 does not exceed the capacity of link (v_6, v_5) , the resource usage of the two partial trees are compatible with each other. Now the new label $lb(v_5, f_0)$ is the only active label and is shown in Figure 5(g).

When $lb(v_5, f_0)$ is finalized, since v_5 is the destination and f_0 is the final flow, the algorithm stops and the corresponding partial tree $T(v_5, f_0)$ is returned as the solution. The total cost of the solution tree is 16.

4. PERFORMANCE EVALUATION

4.1. Topology generators

There has been no consensus in the network community as to how to generate realistic network topology graphs. Therefore, we use several different topology generators to evaluate the heuristic algorithm.

- Waxman model⁵: nodes are randomly distributed into a square region. Suppose d is the Euclidean distance between node u and v , then the two nodes are connected with probability $p(u, v) = \alpha e^{-d/(\beta L)}$, where $0 < \alpha, \beta < 1$ and L is the maximum distance between any two nodes.
- Locality model: this is a variant of the locality model in the GT-ITM topology generator.⁶ The nodes are randomly distributed into ng groups. If two nodes belong to the same group, they are connected with a high probability p_h . Otherwise they are connected with a low probability p_l .
- Power law random graph: nodes are first assigned degrees that follow power law distribution, then connected with each other randomly until they reach their assigned degree.

The above models only generate the network connectivity graph. For our evaluation we also need to generate node and link availability values $a_n(v)$ and $a_l(u, v)$, and node and link cost values $c_n(v)$ and $c_l(u, v)$. In our experiments, unless otherwise specified, $a_n(v)$ and $a_l(u, v)$ are randomly generated from the range $[40, 80]$, and $c_n(v)$ is randomly generated from the range $[10, 30]$. For link cost $c_l(u, v)$, a base value is first generated randomly from the range $[10, 30]$. A model dependent quantity is then added to each link. For the Waxman model, this quantity is proportional to the distance between two nodes. For the locality model, the quantity is 0 for intra group nodes and a constant for inter group nodes. For the power law model, the quantity is proportional to the average degree of the two nodes.

In addition to the overlay topology, we also need to generate the service specification graph H . For the SSSD case, H is just a chain of components. For MSSD, the service specification graph is a tree. We assume the graph is a binary tree and generate a random binary tree as follows. Suppose we want to generate a binary tree with n internal nodes (including the root) and l leaves, we use one internal node as the root, and randomly divide the remaining internal nodes and leaves into left and right subtrees, and generate the subtrees recursively. If a tree has only one internal node, then it is the root and all the leaves are its children. For MSMD, we just generate two random trees for the aggregation and dissemination trees. The resource requirements of service components and data flows are both randomly generated from the range $[20, 30]$.

4.2. Simulation results

We first evaluate how well our algorithm performs at finding low cost service composition solutions, and how well can the use of candidate partial solutions trade off computation overhead for better results.

To do this, we must compare the solution found by our algorithm with the optimal solution. The optimal solution to the service composition problem is difficult to compute. Therefore, we use a simple algorithm to compute a lower bound of the optimal solution. Given the number of replicas nr and the number of required service components nq , the simple algorithm enumerates all the nr^{nq} ways to map the components to the nodes. For each mapping that satisfies the node capacity constraint, the algorithm maps the flows to the shortest paths in the overlay network that connect the corresponding nodes. Apparently the mapping thus obtained may not satisfy the link resource constraints, because some links may be included in multiple shortest paths. However, if we enumerate all possible assignments, the one with the smallest total cost can be used as a lower bound of the optimal solution.

For the first set of experiments, for each class of service specification graph and network topology, we generate 500 instances of the service composition problem. For each instance, we solve the problem using different nc , and compute the ratio of the cost of the heuristic solutions to the lower bound. We set the network size to 40 and the number of service components to 3. For MSSD, the number of sources 4. For MSMD, the number of aggregation and dissemination services is 1 and 2, respectively. There are 2 sources 3 destinations. The topology generators are configured so that the average node degree is about 4, and each service is replicated on 5 nodes.

Figure 6(a) shows the performance of the heuristic algorithm for the SSSD case for different overlay topologies. We can see the cost of the heuristic solutions are very close to the lower bound. Even when $nc = 1$, the average cost of the heuristic solution is within 1% of the lower bound. When nc increases, the ratio would further decrease. This shows our heuristic algorithm is effective at finding low cost composition solutions, regardless of the network topologies. Figure 6(b) and 6(c) are for the MSSD and MSMD scenarios. We can see the heuristic solutions are also close to the lower bound, and the solution becomes better if we increase the number

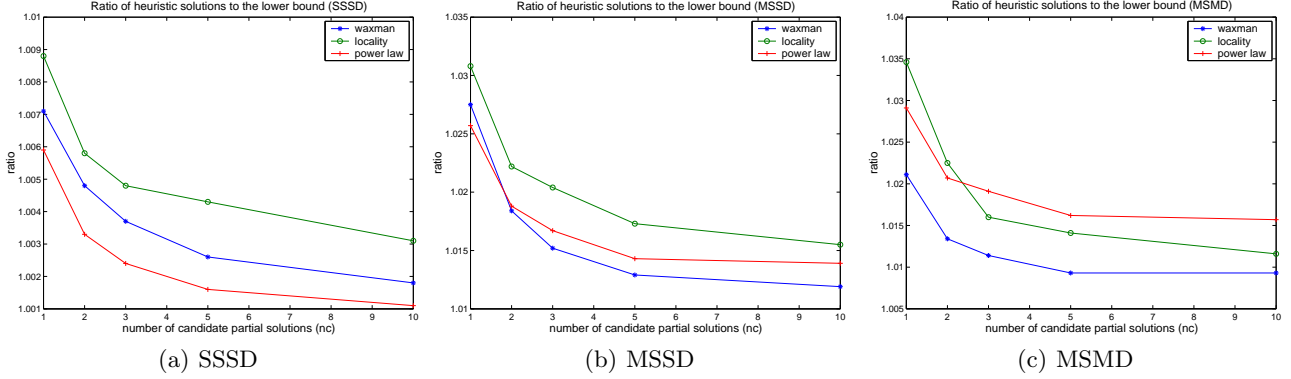


Figure 6. Ratio of heuristic solutions to the lower bound for different service specification graphs

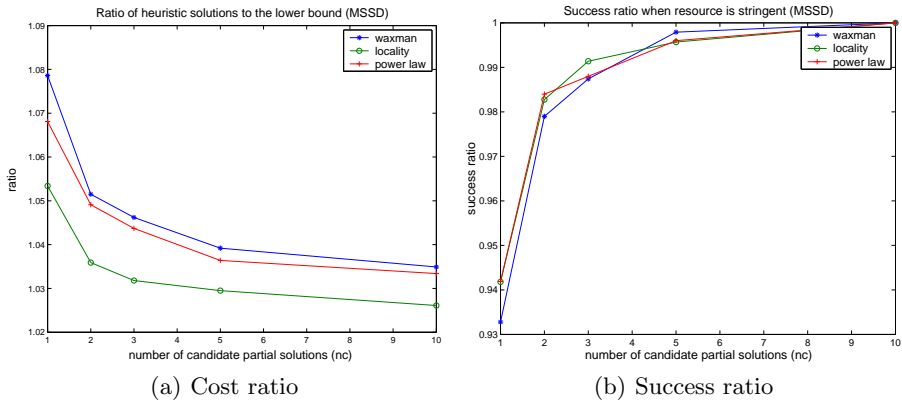


Figure 7. Performance of the algorithm when resource availability is low

of candidate solutions nc . The results for MSSD and MSMD cases are not as good as that for the SSSD case, this is because the MSSD and MSMD service graphs have more flows than the SSSD service graph. As a result, the lower bound returned by the simple algorithm is less likely to be a valid solution.

The previous experiments tested a relatively lightly loaded network, where the average resource availability on a node or link is twice the largest resource requirement of a service component or data flow. Figure 7(a) shows the performance of the algorithm for the MSSD case when both the node and link availability values are uniformly distributed between 30 and 60. The figure shows that when the network resource availability is low, the heuristic solutions when $nc = 1$ may not be satisfactory. For example, the heuristic solution for the Waxman model is about 8% worse than the lower bound when $nc = 1$. However, by increasing the value of nc , we can obtain solutions that are closer to the lower bound.

When the resource availability is low, having a small nc not only results in large cost of the heuristic solutions, but increases the likelihood that the algorithm fails to find a composition solution even when one exists. Figure 7(b) shows the success ratio of the heuristic algorithm when the resource availability is low. The success ratio is defined as the probability that the heuristic algorithm finds a composition when one exists, and we assume a composition solution exists if and only if the heuristic algorithm returns a solution when nc is set to 10. Figure 7(b) shows that when the resource availability is low and $nc = 1$, the success ratio of the algorithm is only about 94%. However, when nc is increased to 2, the success ratio improves to about 98%.

The previous results demonstrate the effectiveness of our algorithm to find near optimal solutions and its ability to trade off computation overhead for better solutions by employing the parameter nc . Next we study how well our algorithm scales to large service overlay networks and required services.

Figure 8 shows the scaling property of the algorithm for the MSSD service composition problem. The network

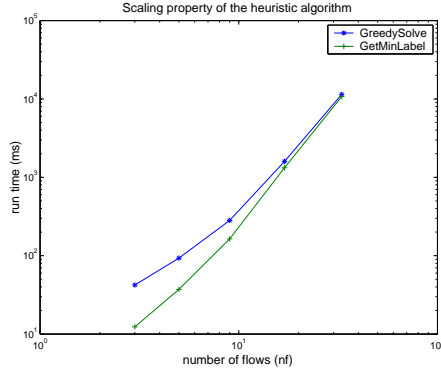


Figure 8. Scaling property of the algorithm

is generated using the locality model. There are 500 nodes in the network and the average node degree is about 10. Each service is replicated on 20 nodes. We set $nc = 2$ and plot the run time of the algorithm against the number of flows nf in log-log scale. *GreedySolve* is the run time of the whole algorithm, and *GetMinLabel* is the run time for extracting the smallest cost label.

Figure 8 shows that in a network of 500 nodes where each service is replicated on 20 nodes, our algorithm can compute the service composition solution for service requests with more than 10 flows using only several seconds*. Moreover, the figures show that *GetMinLabel* consumes significant portions of the total run time. For simplicity, we have implemented the active label set as an unordered double linked list, which takes $O(1)$ for insertion and deletion, but $O(n)$ for *GetMinLabel*. If we implement the label set as a min heap, *GetMinLabel* would take $O(1)$ and insertion and deletion only take $O(\log n)$. Therefore, we believe our algorithm can be implemented to scale to still larger overlay networks and service requests.

5. RELATED WORK

There is much previous work on service composition. The SWORD project¹ provides a toolkit for web service composition. The focus is on automatic generation of a service composition plan based on the functional requirements for the composed application. The SAHARA⁷ project addresses the trust and performance issues of service composition when the services are provided by independent providers. And Gutierrez-Nolasco⁸ et al. have considered the safety issues when composing two distributed services.

Our work assumes that the specification of the required service is given, the different services are cooperative with each other, and our goal is to achieve efficient network resource usage while guaranteeing the QoS of the service. Thus our work is more related to that of Choi⁹ et al., Raman⁴ et al., Gu^{2,10} et al. and Jin^{3,11} et al. Choi⁹ et al. formulate a network configuration problem similar to our service composition problem. They solve the problem by replicating the network graph and apply Dijkstra’s algorithm on the new graph. Raman⁴ et al. use the same algorithm to solve service composition problem, and they focus on defining a cost metric to achieve load balancing. Gu² et al. and Jin³ et al. have used a similar algorithm to first transform the service overlay network into a “staged” graph, then apply Dijkstra’s algorithm on the graph to obtain a service path that satisfies the service functional constraints. All this work focus on applications similar to the SSSD scenarios in our model. And except Jin³ et al., the network resource constraints are not explicitly considered.

Jin¹¹ et al. also considered “service multicast” applications that are similar to the dissemination tree in our multiple source, multiple destination scenario. However, the algorithm presented in the paper handles the receivers one at a time, making use of the “service unicast” algorithm. We believe the comprehensive consideration of all the receivers at the same time can produce better results.

The sFlow¹² and SpiderNet¹⁰ are two recent systems that solve service composition problem for generic DAG (directed acyclic graph) service graphs. However, the sFlow work did not consider the computing resources

*Our experiments were run on a Linux 2.4.21 machine with Intel Xeon 3.06GHz CPU.

required by component services, which we believe is important for QoS sensitive multimedia applications. The SpiderNet system uses constrained probing to collect QoS and resource states along different branches of the original service graph. The destination will merge different branches into candidate service graphs and select the best one according certain criteria. One drawback of the probing approach is that individual branches are probed independently. Thus two probes may pass through the same overlay link which does not have enough bandwidth. Such resource conflicts cannot be detected until both probes arrive at the destination. In contrast, our algorithm never grows a partial solution unless enough resources are present.

Both sFlow and SpiderNet are distributed algorithms. Our algorithm can be decentralized using techniques similar to the well known GHS¹³ distributed spanning tree algorithm. However, we have demonstrated that a centralized algorithm like ours can scale to relatively large service overlay networks and service requests.

REFERENCES

1. S. R. Ponnekanti and A. Fox, "SWORD: A developer toolkit for building composite web services," in *Proceedings of the eleventh World Wide Web Conference*, May 2002.
2. X. Gu, K. Nahrstedt, R. N. Chang, and C. Ward, "QoS-assured service composition in managed service overlay networks," in *the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS 2003)*, May 2003.
3. J. Jin and K. Nahrstedt, "Source-based qos service routing in distributed service networks," in *Proc. of IEEE International Conference on Communications 2004 (ICC2004)*, June 2004.
4. B. Raman and R. H. Katz, "Load balancing and stability issues in algorithms for service composition," in *IEEE INFOCOM 2003*, April 2003.
5. B. Waxman, "Routing of multipoint connections," *IEEE J. Selected Areas in Communication* **6**, 1988.
6. E. W. Zegura, K. Calvert, and M. J. Donahoo, "A quantitative comparison of graph-based models for internet topology," *IEEE/ACM Transactions on Networking* **5**, December 1997.
7. B. R. et al., "The SAHARA model for service composition across multiple providers," in *International Conference on Pervasive Computing*, August 2002.
8. S. Gutierrez-Nolasco and N. Venkatasubramanian, "Reachability snapshots in the presence of failures: An exercise in protocol-service composition," in *International Conference on Dependable Systems and Networks (DSN'02)*, June 2002.
9. S. Choi, J. Turner, and T. Wolf, "Configuring sessions in programmable networks," in *Proc. of IEEE INFOCOM*, April 2001.
10. X. Gu, K. Nahrstedt, and B. Yu, "SpiderNet: An integrated peer to peer service composition framework," in *the IEEE International Symposium on High-Performance Distributed Computing (HPDC-13)*, June 2004.
11. J. Jin and K. Nahrstedt, "QoS service routing in one-to-one and one-to-many scenarios in next-generation service-oriented networks," in *Proc. of the 23rd IEEE International Performance Computing and Communications Conference (IPCCC2004)*, April 2004.
12. M. Wang, B. Li, and Z. Li, "sFlow: Towards resource-efficient and agile service federation in service overlay networks," in *the 24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004)*, (Tokyo, Japan), March 2004.
13. R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Trans. on Programming Languages and Systems* **5**, pp. 66–77, 1983.