Distributed Multimedia Service Composition with Statistical QoS Assurances

Xiaohui Gu, *Member, IEEE,* Klara Nahrstedt, *Member, IEEE*

***Abstract*— Service composition allows future multimedia services to be automatically composed from atomic service components based on user's dynamic service requirements. Previous work falls short for distributed multimedia service composition in terms of scalability, flexibility and quality-of-service (QoS) management. In this paper, we present a fully decentralized service composition framework, called SpiderNet, to address the challenges. SpiderNet provides statistical multi-constrained QoS assurances and load balancing for service composition. SpiderNet supports directed acyclic graph composition topologies and exchangeable composition orders. We have implemented a prototype of SpiderNet and conducted experiments on both wide-area networks and simulation testbed. Our experimental results show the feasibility and efficiency of the SpiderNet service composition framework.**

***Index Terms*— Service Composition, Service Overlay Network, Middleware, Quality-of-Service**

I. Introduction

Emerging advanced distributed multimedia services, such as voice-over-IP conferencing [9] and ubiquitous multimedia streaming, demands a scalable, robust, and adaptive multimedia service infrastructure. The conventional client-server system model has become inadequate for next-generation multimedia service provisioning due to its poor scalability, customizability, manageability, and reliability. Thus, we propose a *compositional* approach to multimedia service provisioning, which can dynamically create multimedia services using distributed service components. To support efficient service composition, we propose a service overlay network model where thousands of media servers and proxies are connected into an application-level overlay network.

Recently, several research projects (e.g., [13], [14], [20], [3]) have addressed the service composition problem. However, existing solutions present three major limitations. First, they all adopt a centralized approach to service composition, which has scalability limitation. Second, they lack systematic quality-of-service (QoS) management that are especially important for multimedia applications. On the other hand, existing QoS solutions (e.g., [5], [4], [18], [7], [6]) are not readily applicable to service composition due to its application-specific service requirements (e.g., service function requirements, inter-service dependency constraints). Third, previous work only supports linear composition topology and fixed composition order, which greatly limits the applicability and efficiency of service composition.

In this paper, we present a QoS-aware service composition framework called SpiderNet to address the above challenges. SpiderNet provides a novel *bounded composition probing* (BCP) scheme to achieve QoS-aware service composition in a scalable and efficient fashion. The basic idea of BCP is to intelligently examine a small subset of *good* candidate compositions according to user's service requirements and current system conditions. The BCP scheme executes a hop-by-hop distributed composition protocol to achieve three goals. First, it discovers available service components that match the user's service function requirements (e.g., transcoding, image scaling, caption embedding). Second, it checks statistical QoS condition (e.g., mean service time), resource availability, and inter-service dependency/commutative relations to select qualified service components. Third, it collects statistical QoS and resource state information from selected candidate service components. Finally, the best service composition is selected based on the probing results, user's QoS/resource requirements, and global load balancing objective function.

The SpiderNet service composition approach has three unique features. First, it provides multi-constrained statistical QoS assurances [12] for the composed distributed multimedia services. Second, SpiderNet achieves good load balancing in the SON to improve overall resource utilization. Third, SpiderNet supports directed acyclic graph (DAG) composition topologies and exchangeable composition orders. Thus, the composed services can be more efficient with parallel execution of service functions instead of strict pipelined chaining of service functions. By exploring exchangeable composition orders, SpiderNet can improve the QoS provisioning and resource utilization in service composition. We have implemented a prototype of SpiderNet and conducted extensive experiments by evaluating the prototype on both

Manuscript received October 8th, 2003; revised May 8th, 2004, accepted November 27th, 2004. Xiaohui Gu is with IBM T. J. Watson Research Center, Hawthorne, New York. Her email address is xiaohui@us.ibm.com. Klara Nahrstedt is with Department of Computer Science, University of Illinois at Urbana-Champaign. Her email address is klara @ cs.uiuc.edu. This work was mainly supported by the NASA grant and NSF grant.

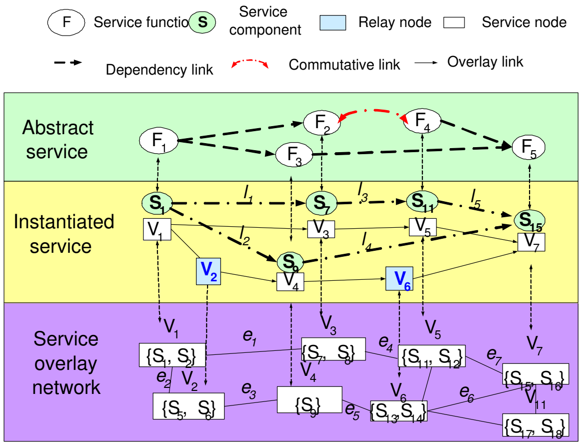

Fig. 1. SpiderNet system architecture.

large-scale simulation testbed and wide-area network testbed PlanetLab [1]. The experimental results show that SpiderNet can achieve near-optimal QoS-aware service composition performance with low overhead.

The rest of the paper is organized as follows. In Section II, we present the SpiderNet system model. In Section III, we present the distributed service composition design in detail. In Section IV, we present experimental results. Finally, the paper concludes in Section V.

II. System Model

In this section, we first introduce the SpiderNet three-tier system model illustrated by Figure 1. Then, we formally define the QoS-aware service composition problem. We summarize the notations in Table I for later references.

A. Abstract Service Layer

The abstract service layer consists of user's composite service requirements (e.g., secure mobile video-on-demand). The user can specify a composite service request using a function graph (ξ) and a QoS requirement vector (Q^{target}). The function graph specifies required service functions (F_i) and inter-service dependency and commutative relations, which is illustrated by the top tier in Figure 1. The dependency relation from F_1 to F_2 means that the output of F_1 will be used as the input by F_2, which is denoted by $F_1 \rightsquigarrow F_2$. The commutative relation between F_1 to F_2 means that the composition order of F_1 and F_2 can be exchanged without affecting the composite service's function. We formally define the function graph as follows,

Definition 2.1: A function graph is defined as $\xi = (F, DR, PR)$, $\mathcal{F} = \{F_i | 1 \leq i \leq |\mathcal{F}|\}$, $DR = \{dr_i | dr_i \triangleq F_i \rightsquigarrow F_j | 1 \leq i \leq |DR|\}$, $PR = \{pr_i | pr_i \triangleq F_i \sim F_j | 1 \leq i \leq |PR|\}$, where $|\mathcal{F}|$, $|DR|$, and $|PR|$ represent the cardinalities of the set $\mathcal{F}$, DR, and PR, respectively.

notation	meaning
F	service function
s	service component
ξ	function graph
Q^{target}	QoS requirements
λ	ServFlow
ℓ	service link
e	overlay link
$Q^{in} = [q_1^{in}, ..., q_d^{in}]$	static input QoS requirements
v	overlay node
$Q^{out} = [q_1^{out}, ..., q_d^{out}]$	static output QoS properties
$[r_1, ..., r_n]$	end-system resource availability
$Q^p = [q_1^p, ..., q_m^p]$	dynamic QoS metrics
γ	adaptation policy
$\gamma(Q^{in})$ or $\gamma(Q^{out})$	Q^{in} or Q^{out} after adaptation
Γ	adaptation policy set
π	adaptation condition hypercube
R^λ	resource requirements of λ
$\wp$	overlay path
β	probing budget
α	probing quota
ψ^λ	aggregate resource cost metric
$\langle \mathcal{C}^{q_i}, \mathcal{P}^{q_i} \rangle$	statistical requirement for q_i
$\mathcal{M}_{r_k}$	mean value of r_k
$\mathcal{M}_{bw}$	mean value of bw
bw	available bandwidth
$\langle \mathcal{C}_{bw}, \mathcal{P}_{bw} \rangle$	statistical bandwidth requirement
P	probing message
$\langle \mathcal{C}_{r_k}, \mathcal{P}_{r_k} \rangle$	statistical requirement of r_k

TABLE I

Notations.

We use $Q^{target} = [\langle \mathcal{C}^{q_1}, \mathcal{P}^{q_1} \rangle, ..., \langle \mathcal{C}^{q_m}, \mathcal{P}^{q_m} \rangle]$ to define the user's statistical QoS requirements for the composed service, where $\langle \mathcal{C}^{q_i}, \mathcal{P}^{q_i} \rangle$ specifies the bound $\mathcal{C}^{q_i}$ and the satisfaction probability[1] $\mathcal{P}^{q_i}$ for the metric q_i that represents a QoS metric such as delay and loss rate. Users can either directly specify the composite service request using extensible markup language (XML) or use visual specification tools [11].

B. Instantiated Service Layer

The instantiated service layer consists of distributed multimedia services that are dynamically composed from existing service components. A multimedia service component (s_i) is a self-contained multimedia application unit providing a certain functionality (e.g., media transocoding), which is illustrated by Figure 2 (a). Each service component has several input and output ports for

[1] The satisfaction probability is defined as the probability when the random variable q_i is less or equal to $\mathcal{C}^{q_i}$, assuming q_i is minimum-optimal.

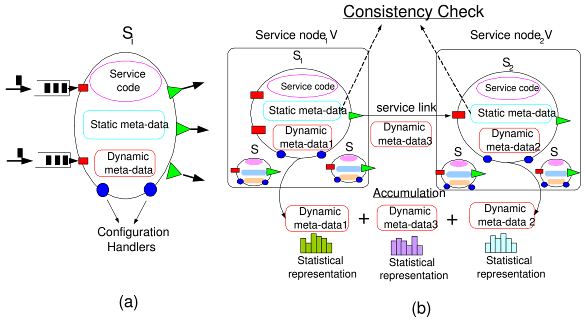

Fig. 2. Service composition model.

receiving input messages and sending output messages, respectively. Each input port is associated with a message queue for asynchronous communication between service components.

Each service component consists of four items, (1) *function name* describing the service function provided by the service component, (2) *service code* representing the service implementation, (3) *static meta-data*, and (4) *dynamic meta-data*. The static meta-data of a service component s_i consists of three parts: (1) the location of s_i; (2) input quality requirements of the service component such as media format, frame rate, which is denoted by $Q^{in} = [q_1^{in}, ..., q_d^{in}]$, and output quality properties of the service component, denoted by $Q^{out} = [q_1^{out}, ..., q_d^{out}]$; and (3) adaptation policies $\Gamma = \{\gamma_1, ..., \gamma_l\}$, where γ_i is expressed by an *if-condition-then-action* construct. The dynamic meta-data of a service component describe its fluctuating performance conditions, such as recent service delay. We use statistical QoS vector $Q^{s_i} = [q_1^{s_i}, ..., q_m^{s_i}]$ to characterize the dynamic QoS metrics of the service component. Each QoS metric $q_k, 1 \leq k \leq m$ is represented by a random variable, whose histogram is constructed from a number of recent sample values. Based on the histogram, we can estimate the probability density function (p.d.f.) of q_k, denoted by ρ_{q_k}. We use $Pr(q_k \leq \mathcal{C}^{q_k})$ to define the satisfaction probability that the dynamic value of q_k is no larger than the required upper bound $\mathcal{C}^{q_k}$.

When we compose two service components, we need to address two key issues, illustrated by Figure 2 (b). First, we need to check the QoS consistencies between two different service components since they can be developed by different third-party service providers. The QoS consistency includes two aspects. First, we check whether Q^{in} and Q^{out} of the two composed service components are consistent. Second, we check wether the adaptation policies of the two service components conflict with each other. The second issue is to derive the dynamic QoS values of the composed service from those of constituent service components and the network connection between them. Because we use statistical QoS metrics, the accumulation of QoS metrics means convolution [17] between them[2].

We describe a composite distributed multimedia service using a DAG called *ServFlow* (λ), illustrated by the middle tier in Figure 1. The nodes in the ServFlow represent the service components and the links in the ServFlow represent application-level connections called service link. Each service link is mapped to an overlay path by the overlay data routing layer. For example, in Figure 1, the service link ℓ_i is mapped to the overlay path $\wp_i = e_1 \rightarrow e_2 ... \rightarrow e_k$. We formally define the ServFlow as follows,

Definition 2.2: A ServFlow is defined as $\lambda = \langle S, L \rangle$, $S = \{s_k | 1 \leq k \leq |S|\}$, $L = \{\ell_k | \ell_k = s_i \rightarrow s_j, 1 \leq k \leq |L|\}$.

For example, the ServFlow shown in Figure 1 can be described as $\lambda = (\{s_1, s_7, s_9, s_{11}\}, \{\ell_1/\wp_1, \ell_2/\wp_2, \ell_3/\wp_3, \ell_4/\wp_4, \ell_5/\wp_5\})$, where $\wp_1 = e_1$, $\wp_2 = e_2 \rightarrow e_3$, $\wp_3 = e_4$, $\wp_4 = e_5 \rightarrow e_6$, and $\wp_5 = e_7$. If an overlay node contributes multimedia services on a ServFlow λ, it is called a *service node*. If an overlay node only performs application-level data relaying on λ, it is called a *relay node*.

To quantify the load balancing property of an instantiated ServFlow, we define a resource cost aggregation metric, denoted by ψ^{λ}, which is the weighted sum of ratios between resource requirements of the service components/service links between resource availabilities on the hosting overlay nodes/overlay paths. We use $\mathcal{C}_{r_i}^{s_i}$ and $\mathcal{P}_{r_i}^{s_i}$ to represent the resource requirement threshold and satisfaction probability of the service component s_i for the $i'th$ end-system resource type (e.g., CPU, memory, disk storage), respectively. Similarly, we use $\mathcal{C}_{bw}^{\ell_i}$ and $\mathcal{P}_{bw}^{\ell_i}$ to denote the required threshold and satisfaction probability for the network bandwidth on the service link ℓ_i, respectively.

The resource requirements of a service component depend on its implementations and the current workload. In contrast to the conventional data routing path, the resource requirements along a ServFlow are no longer uniform due to the non-uniform service functionalities on the ServFlow. Different service components can have different resource requirements due to heterogeneous functions and implementations. The bandwidth requirements also vary among different service links since the value-added service instances can change the original media content (e.g., image scaling, color filter, information embedding). We use $\mathcal{M}_{r_i}^{v_j}$ to denote *mean* availability

[2] For simplicity, we assume that all QoS metrics are additive since a multiplicative metric (e.g., loss rate) can be transformed into additive parameters using logarithmic function. We also assume that all QoS metrics of different service components and network links are independent.

of $i'th$ end-system resource type on the overlay node v_j.

We use $\mathcal{M}^{\wp_i}_{bw}$ to denote the *mean* availability of the bandwidth on the overlay path $\wp_i$, which is defined as the minimum mean available bandwidth among all overlay links $e_i \in \wp$. The mean values can be calculated from the p.d.f.'s of the corresponding statistical metrics. Hence, the resource cost aggregation metric ψ^λ is defined as follows,

$$\psi^\lambda = \sum_{s_i/v_j \in \lambda} \sum_{k=1}^{n} w_k \frac{\mathcal{C}^{s_i}_{r_k}}{\mathcal{M}^{v_j}_{r_k}} + w_{n+1} \sum_{\ell_i/\wp_i \in \lambda} \frac{\mathcal{C}^{\ell_i}_{bw}}{\mathcal{M}^{\wp_i}_{bw}} \tag{1}$$

$$\sum_{k=1}^{n+1} w_k = 1, 0 \leq w_k \leq 1, 1 \leq k \leq n+1$$

$w_k, 1 \leq k \leq n+1$ represents the importance of different resource types[3]. We can customize ψ^λ by assigning higher weights to more critical resource types. The ServFlow with smaller resource cost aggregation value has better load balancing property because the resource availabilities exceed the resource requirements by a larger margin.

C. Service Overlay Network Layer

The substrate tier of the SpiderNet system is a service overlay network that consists of distributed overlay nodes (v_i) connected by application-level connections called overlay links (e_i). Each overlay node can provide one or more multimedia service components. The overlay network topology can be formed by connecting each overlay node with a number of other nodes called neighbors via overlay links. Application-level data relaying [2] is required between two overlay nodes that are not directly connected. For example, in Figure 1, the data sent by v_1 to v_4 needs to be relayed by v_2. The QoS-aware service composition is then performed on top of the overlay data routing layer.

Each node v_i is associated with a statistical resource availability vector $[r^{v_i}_1, ..., r^{v_i}_n]$, where $r^{v_i}_k$ is a random variable describing the statistical availability for the $k'th$ end-system resource type (e.g. CPU, memory, disk storage). We use a histogram to estimate the probability density function (p.d.f.) ρ_r of the random variable r. Thus, we do not need to make any assumption about the distribution of the random variable. The histogram is constructed from a number of recent sample values. For example, if the total sample size of the histogram is Z and the number of sample values in the bin $[x - \frac{\triangle x}{2}, x + \frac{\triangle x}{2}]$ is Y, then we have $\rho_r(x) \approx \frac{Y}{Z}$. Each node v_i also maintains statistical bandwidth availability bw^{ℓ_j} for its adjacent overlay links ℓ_j. For scalability, each node maintains the above histograms *locally*, which are not disseminated to other overlay nodes.

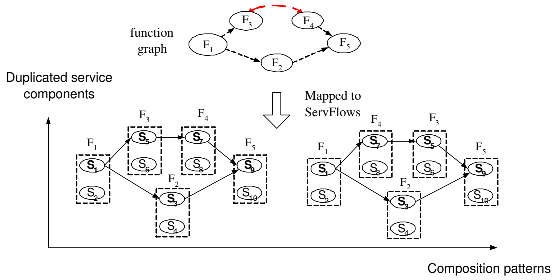

Fig. 3. QoS-Aware service composition problem illustration.

D. Problem Description

We formulate QoS-aware service composition as a two dimensional graph mapping problem illustrated by Figure 3. In one dimension, we can derive different composition patterns from the original function graph by considering the commutative links. In the other dimension, we can map each service function into different duplicated service components in the SON. These duplicated service components provide the same functionality but can have different QoS properties (e.g., service time) and available resources on the local peer host (e.g., CPU, memory). For example, in Figure 3, function F_1 can be mapped to two duplicated service components s_1 and s_2. We can derive different ServFlows from the function graph by considering the above two dimensions. Thus, the QoS-aware service composition problem (QSC) is to find the best mapping from the function graph to the best qualified ServFlow that satisfies the user's multi-constrained QoS requirements Q^{req} and achieves best load balancing in the current multimedia service overlay. We formally define the QSC problem as follows,

Definition 2.3: **QoS-aware service composition (QSC) problem**. Given a composite service request $\Upsilon = \langle \xi, Q^{target} \rangle$ and a multimedia service overlay G = (V, E), the QSC problem is to map ξ into the best qualified ServFlow λ, such that λ minimizes ψ^λ subject to the following constraints:

$$Pr(q^\lambda_k \leq \mathcal{C}^{q_k}) \geq \mathcal{P}^{q_k}, \forall k, 1 \leq k \leq m \tag{2}$$

$$Pr(r^{v_j}_k \geq \mathcal{C}^{s_i}_{r_k}) \geq \mathcal{P}^{s_i}_{r_k},\ \forall k, 1 \leq k \leq n, \forall s_i/v_j \in \lambda \tag{3}$$

$$Pr(bw^{\wp_i} \geq \mathcal{C}^{\ell_i}_{bw}) \geq \mathcal{P}^{\ell_i}_{bw}, \forall \ell_i/\wp_j \in \lambda \tag{4}$$

Theorem 2.1: QSC problem is NP-hard.
Readers are referred to [8] for the proof details.

III. System Design

In this section, we present the bounded composition probing protocol, concepts of probing budget and prob-

[3] Some services are computationally intensive (e.g., image analysis), which require low network bandwidth. However, some services require high network bandwidth and low CPU (e.g., forwarding service).

ing quota, per-hop probe processing algorithm, and optimal service composition selection. Finally, we discuss several enhancements to the basic distributed service composition scheme.

A. Bounded Composition Probing Protocol

SpiderNet executes a bounded composition probing (BCP) protocol to perform distributed service composition. Given a service composition request, the source node invokes the BCP protocol[4], which includes four major steps:

Step 1. Initialize the probe. The source first generates a composition probing message, called probe. The probe carries the information of function graph and the user's QoS/resource requirements. The probe can spawn new probes in order to concurrently examine multiple next-hop choices. To control the number of spawned probes, the probe carries a *probing budget* (β) that defines how many probes we could use for a composition request. We will introduce the probing budget in more detail in Section III-B.

Step 2. Hop-by-hop probe processing. Each peer processes a probe independently using only local information. The goal of hop-by-hop distributed probe processing is to collect needed information and perform intelligent parallel searching of multiple candidate ServFlows. We will describe this step in detail in Section III-D.

Step 3. Optimal composition selection. The destination collects the probes for a request with certain timeout period. It then selects the best qualified ServFlow based on the resource and QoS states collected by the probes. We will discuss this step in detail in Section III-E.

Step 4. Setup service session. Finally, the destination sends an acknowledge message along the reversed selected ServFlow to confirm resource allocations and initialize service components at each intermediate peer. Then the application sender starts to send application data stream along the selected ServFlow. If no qualified ServFlow is found, the destination returns a failure message to the source directly. Figure 4 shows the pseudo-code of distributed service composition algorithm.

B. Probing Budget

We introduce the concept of *probing budget* that allows us to precisely control the number of probes used for each composition request. The probing budget represents the trade-off between the probing overhead and composition optimality. Larger probing budget allows us to examine more candidate ServFlows, which allows us to find a better qualified ServFlow. Different from previous work, SpiderNet can provide an adaptive composition solution with tunable performance by properly adjusting the probing budget. For example, we can use larger probing budget for the request with (1) higher priority, (2) stricter QoS constraints, or (3) more complex function graph. We can also adaptively adjust the probing budget based on user feedbacks and historical information.

```
Input: request $\langle \xi, Q^{target} \rangle$, probing budget $\beta$;
Output: best qualified ServFlow $\lambda$.

The source node generates a probe $P$
  Initialize $P$ with $\xi$, $\beta$, $Q^{target}$
  Derive first-hop services $s_1, ..., s_k$
  Spawn k probes from $P$
  Send new probes to next hop
Per-hop probe processing at an intermediate node $v_i$
  if $v_i$ is a relay node
      Derive next data routing hop $v_k$
      Update $P$ with its local info. of $e(v_i, v_k)$
      Forward $P$ to $v_k$
  if $v_i$ is a service node providing $s_i$
      Check Resource/QoS conformance
      if ServFlow is qualified
          Perform soft resource allocation
          Derive next-hop service functions using $\xi$
          Check QoS consistency
          Spawn new probes
          Updates new probes with local info. of $s_i$
          Send new probes to next-hop
      else Drop the received probe P
The sink node selects the best ServFlow
```

Fig. 4. Distributed service composition algorithm.

C. Probing Quota

Although the probing budget could control the total probing overhead, it cannot guarantee the fair sharing of the probing budget among different service functions. If there are many candidate service components for a service function, dividing β_0 among *all* candidate service components can quickly use up the probing budget. To address the problem, we associate a *probing quota* (α_i) with each service function F_i to limit the number of probes used for F_i. In the basic distributed service composition algorithm, we assume that all service functions are equally important. We will describe the differentiated probing quota allocation in Section III-F. Let us assume that we are given a function graph ξ includes k branch paths $\tau_1, ..., \tau_k$, each of which includes L_i service functions with Z_i $(Z_i \geq 1, 1 \leq i \leq k)$ alternative permutations (i.e., composition patterns). If each service function is associated with the same probing quota α, the total probes generated on the branch path with L_i

[4] The BCP initiator can also be the destination or a third-party overlay node.

service functions and Z_i permutations is $Z_i \cdot \alpha^{L_i}$. Thus, we can derive α based on the following inequalities:

$$Z_i \cdot \alpha^{L_i} \leq \beta_0/k, \quad 1 \leq i \leq k. \tag{5}$$

For example, in Figure 3, the function graph includes two branch paths. The first branch has two permutations that can generate $2\alpha^4$ probes if BCP uses α probes for each function. The second branch can generate α^4 probes. According to Equation 5, we have $2\alpha^4 \leq \frac{\beta_0}{2}$ and $\alpha^4 \leq \frac{\beta_0}{2}$. Thus, the upper-bound of α is $\lfloor \sqrt[4]{\frac{\beta_0}{4}} \rfloor$.

D. Per-hop Probe Processing

We now describe the details of the per-hop probe processing steps that mainly includes six steps:

(a) Resource/QoS check and soft resource allocation. When a service node receives a probe, it first check whether the QoS and resource values of the probed ServFlow already violate the user's requirements. If the accumulated QoS and resource values already violate the user's requirements, the probe is dropped immediately. Otherwise, the peer will temporarily allocate required resources to the expected application session. However, the resource allocation is *soft* since it will be cancelled after certain timeout period if the peer does not receive a confirmation message. The purpose of this soft resource allocation is to avoid conflicted resource admission caused by concurrent probe processing. Thus, we can guarantee that probed available resources are still available at the end of the probing process.

(b) Derive next-hop service functions. Next, the service node derives the next-hop service functions according to the dependency and commutative relations in the function graph. All the functions dependent on the current function are considered as next-hop functions. For each next-hop function F_k derived above, if there is a exchange link between F_k and F_l, F_l is also a possible next-hop function. The probing budget is proportionally distributed among next-hop functions according to their probing quotas. To avoid incorrect loops in the composition probing, we modify the functions graphs in the new probes destined to the two exchangeable service functions F_k and F_l. In the probe for F_k, we modify its function graph by replacing $F_k \sim F_l$ with $F_k \rightsquigarrow F_l$. In the probe for F_l, we modify its function graph by first replacing $F_k \sim F_l$ with $F_l \rightsquigarrow F_k$, and then letting F_k inherit all the relations of F_l. More details can be found in [8].

Step 2.3: Discover candidate service components. The service node discovers candidate service components for all next-hop functions derived above. For scalability, we implement a decentralized service discovery based on the distributed hash table (DHT) system [16]. Readers are referred to [10] for detailed description.

Step 2.4: Check QoS consistency. Based on the service discovery results, the service node then performs QoS consistency check between the current service component and next-hop candidate service components. The QoS consistency check includes two aspects: (1) the consistencies between output QoS parameters Q^{out} of the current service component and input QoS parameters Q^{out} of the next-hop service component; and (2) the compatibility between the adaptation policies of two connected service components. Unlike the IP-layer network where all routers provide a uniform data forwarding service, the node in the multimedia service overlay can provide different multimedia services, which makes it necessary to perform QoS consistency check between two connected service components. We first check the parametric consistency based on the following definitions,

Definition 3.1: Parametric consistency relation $(Q^{out}_{s_a} \preceq Q^{in}_{s_b})$. Given two service components s_a and s_b, $Q^{out}_{s_a} \preceq Q^{in}_{s_b}$ if and only if $\forall i, 1 \leq i \leq d$, $\exists j, 1 \leq j \leq d$, (1) $q^{out}_{a_j} = q^{in}_{b_i}$, if $q^{in}_{b_i}$ is a single value, and (2) $q^{out}_{a_j} \subseteq q^{in}_{b_i}$, if $q^{in}_{b_i}$ is a range value.

In addition, we also check whether the adaptation policies of the two service components are compatible with each other. Generally, we can express an adaptation policy using an *if-condition-then-action* construct. For example, a video tracking service can have the following adaptation policy, *if CPU is below 40% and bandwidth is below 100kbps, then use RGB8_color*. We say two adaptation policies are compatible if their actions will not cause any parametric in-consistency. For example, an adaptation policy of a service component specifies that the service component changes output media format from MPEGII to JPEG when the available CPU is lower than 40%. If the component's successor specifies that the required input media format must be MPEGII, then the adaptation policy will potentially cause parametric in-consistency between the two service components. We use hyper-cube π to model adaptation conditions, where each condition attribute (e.g., CPU and bandwidth in the visual tracking example) represents one dimension of the hyper-cube. We check the compatibility of two adaptation policies based on the relations of their condition hypercubes (e.g., equal, disjoint, overlapping), which is formally defined as follows,

Definition 3.2: Adaptation Rule Set Compatibility Relation $(\Gamma_{s_a} \bowtie \Gamma_{s_b})$. We use $\gamma(SQ^{in})$ and $\gamma(SQ^{out})$ to represent the new SQ^{in} and SQ^{out} after the service component is changed by its adaptation policy γ. Given two adaptation policy sets $\Gamma_{s_a} = \{\gamma_{a_1}, ..., \gamma_{a_A}\}$ and $\Gamma_{s_b} = \{\gamma_{b_1}, ..., \gamma_{b_B}\}$, we define that two adaptation policy sets are compatible, denoted by $\Gamma_{s_a} \bowtie \Gamma_{s_b}$, if and only if $\forall \gamma_{a_i} \in \Gamma_{s_a}$, $\forall \gamma_{b_j} \in \Gamma_{s_b}$, (1) $\pi_a = \pi_b \Rightarrow \gamma_{a_i}(SQ^{out}_{s_a}) \preceq \gamma_{b_j}(SQ^{in}_{s_b})$; (2) $\pi_a \cap \pi_b = \emptyset \Rightarrow$

$\gamma_{a_i}(SQ^{out}_{s_a}) \preceq SQ^{in}_{s_b} \wedge SQ^{out}_{s_a} \preceq \gamma_{b_j}(SQ^{in}_{s_b})$; (3) $\pi_a \cap \pi_b \neq \emptyset \wedge \pi_a \nsubseteq \pi_b \wedge \pi_b \nsubseteq \pi_a \Rightarrow \gamma_{a_i}(SQ^{out}_{s_a}) \preceq SQ^{in}_{s_b} \wedge SQ^{out}_{s_a} \preceq \gamma_{b_j}(SQ^{in}_{s_b}) \wedge \gamma_{a_i}(SQ^{out}_{s_a}) \preceq \gamma_{b_j}(SQ^{in}_{s_b})$, (4) $\pi_a \subset \pi_b \Rightarrow \gamma_{a_i}(SQ^{out}_{s_a}) \preceq \gamma_{b_j}(SQ^{in}_{s_b}) \wedge SQ^{out}_{s_a} \preceq \gamma_{b_i}(SQ^{in}_{s_b})$; (5) $\pi_b \subset \pi_a \Rightarrow \gamma_{a_i}(SQ^{out}_{s_a}) \preceq SQ^{in}_{s_b} \wedge \gamma_{a_i}(SQ^{out}_{s_a}) \preceq \gamma_{b_j}(SQ^{in}_{s_b})$.
Based on the above two definitions, we define the inter-component QoS consistency relation as follows,

Definition 3.3: Inter-component QoS consistency ($s_a \Leftrightarrow s_b$). Given two service components s_a and s_b, their static meta-data items $(SQ^{in}_{s_a}, SQ^{out}_{s_a}, \Gamma_{s_a})$ and $(SQ^{in}_{s_b}, SQ^{out}_{s_b}, \Gamma_{s_b})$. We define that s_a is QoS consistent with s_b ($s_a \Leftrightarrow s_b$), if and only if (1) $SQ^{out}_{s_a} \preceq SQ^{in}_{s_b}$ and (2) $\Gamma_a \bowtie \Gamma_b$.

In SpiderNet, static meta-data items are described using the XML-based markup language HQML [11]. SpiderNet check the QoS consistency between two service components using the HQML syntactic and semantic parsers [11] according to the above Inter-component QoS consistency definitions. The computation complexity of the parametric consistency check is $O(d)$, where d is the dimension of the vectors Q^{in} and Q^{out}. If the adaptation condition requires a K-dimensional space, then we can decide the relation of two condition hypercubes in $O(K^2)$. Thus, the computation complexity of checking the compatibility of two adaptation policy sets is $O(ABdK^2)$, where A defines the size of rule set of s_a, B defines the size of rule set of s_b. The computation complexity of the complete inter-component QoS consistency check algorithm is $O(ABdK^2)$.

Step 2.5: Select next-hop service components. Due to the probing budget and probing quota constraints, the service node v_i may not be able to probe all the qualified next-hop service components. Thus, v_i selects a subset of most promising next-hop candidate service components to probe. Suppose we find K qualified candidate service components for the next-hop function F_k. Let β_k denote the available probing budget for F_k decided by step 2.2. Let α_k define the probing quota for F_k. Thus, the number of probes that can be used by F_k is $I = min(\beta_k, \alpha_k)$. If $I \geq K$, then the service node spawns K new probes from the received probe to examine all K candidate service components. Each new probe has a probing budget $\lfloor \frac{\beta_k}{K} \rfloor$. However, if $I \langle K$, then we do not have enough probing budget to probe all the K candidate service components. In this case, v_i selects I most promising next-hop service components from the K candidates based on the local available information. To meet our multi-constrained QoS and resource management goals, v_i selects best next-hop service components based on a combined metric that comprehensively considers all local states information such as the network delays retrieved from the overlay data routing layer and average service delays of the candidate service component retrieved from the static meta-data. Finally, v_i spawns a new probe for each selected next-hop service component. Each new probe has a probing budget $\lfloor \frac{\beta_k}{I} \rfloor$.

Step 2.6: Update probe with statistical local states. In the last step, the service node v_i sets the content of each new probe P^{new} based on the content of the received probe P and its local statistical information. First, v_i updates the p.d.f.'s of the accumulated QoS values Q^{λ} of the probed ServFlow λ using the convolution between its old values recorded in P and Q^{s_i} of the current service component s_i as follows,

$$\rho^{\lambda}_{q_i}(u) = \int_{-\infty}^{\infty} \rho^{\lambda}_{q_i}(x) \cdot \rho_{q_i^{s_i}}(u - x)dx, \; 1 \leq i \leq m \quad (6)$$

Second, v_i updates the resource requirements of the probed ServFlow λ with the CPU resource requirement of s_i $(\mathcal{C}^{s_i}_{cpu}, \mathcal{P}^{s_i}_{cpu})$ and the bandwidth requirement $(\mathcal{C}^{\ell_k}_{bw}, \mathcal{P}^{\ell_k}_{bw})$ for the service link ℓ_k between s_i and the selected next-hop service instance in P^{new}. Third, v_i calculates the mean resource availability value $\mathcal{M}_{r_k}$ and the resource satisfaction probability $Pr(r_k \geq \mathcal{C}^{s_i}_{r_k})$ for the end-system resource r_k as follows,

$$\mathcal{M}_{r_k} = \int_{-\infty}^{+\infty} x\rho_{r_k}(x)dx, \quad 1 \leq k \leq n \quad (7)$$

$$Pr(r_k \geq \mathcal{C}^{s_i}_{r_k}) = \int_{\mathcal{C}^{s_i}_{r_k}}^{+\infty} \rho_{r_k}(x)dx, \quad 1 \leq k \leq n \quad (8)$$

Fourth, v_i derives the first overlay link e_k to the selected next-hop service instance according to the local overlay data routing table. Then, v_i updates the values of Q^{λ} using the convolution between old Q^{λ} values and Q^{e_k},

$$\rho^{\lambda}_{q_i}(u) = \int_{-\infty}^{\infty} \rho^{\lambda}_{q_i}(x) \cdot \rho_{q_i^{e_k}}(u - x)dx, \; 1 \leq i \leq m \quad (9)$$

v_i then updates average available bandwidth on the overlay path $\wp_i$ to the next-hop as follows,

$$\mathcal{M}^{\wp_i}_{bw} = min[\mathcal{M}^{\wp_i}_{bw}, \int_{-\infty}^{+\infty} x\rho_{bw^{e_k}}(x)dx] \quad (10)$$

The bandwidth satisfaction probability $Pr'_{bw} = Pr(bw_{\wp_i} \geq \mathcal{C}^{\ell_i}_{bw})$ is updated as follows,

$$Pr'_{bw} = Pr_{bw} \cdot \int_{\mathcal{C}^{\ell_i}_{bw}}^{+\infty} \rho_{bw^{e_k}}(x)dx \quad (11)$$

We have presented the per-hop probe processing algorithm at a service node. In contrast, the per-hop probe processing at a relay node is much simpler since it does not provide any service components but only performs application-level data forwarding in the overlay network. The relay node does not spawn new probes. It only updates the content of the received probe P with the local statistical information about the overlay link e_k

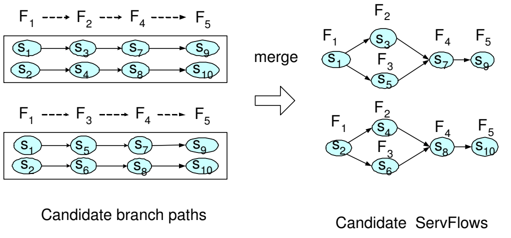

Fig. 5. Merge branch paths into DAG ServFlows.

towards the next-hop service node specified in P.

E. Optimal Service Composition Selection

At the destination node, SpiderNet selects the best qualified ServFlow based on the information collected by the received probes. If the function graph has a linear path structure, each probe records a complete service composition. However, if the function graph has a directed acyclic graph (DAG) structure, each probe only collects the information for one composition branch. For example, in Figure 5, each probe traverses either the branch path $F_1 \rightarrow F_2 \rightarrow F_4 \rightarrow F_5$ or $F_1 \rightarrow F_3 \rightarrow F_4 \rightarrow F_5$. Hence, we need to first merge the examined branch paths into complete DAG ServFlows. We briefly describe the merging algorithm as follows. First, we classify all the received branch paths into Y sets according to their provisioned service functions. All branch paths within one set should include the same set of service functions. For example, in Figure 5, we classify the four received branch paths into two sets. Second, we merge every Y combinable branch paths, one from each of the Y sets, into a complete DAG ServFlow. Two branch paths are combinable if and only if their common service functions are performed by the same service component. For example, in Figure 5, we can derive two candidate DAG ServFlows from the received four branch paths.

When we merge two branch paths, we need to calculate the statistical resource and QoS values of the DAG ServFlow from its constituent branch paths. The mean values of resource availabilities (i.e., $\mathcal{M}^{s_i}_{r_k}$, $\mathcal{M}^{\ell_j}_{bw}$) of the DAG ServFlow are the union of its constituent branch paths. The statistical QoS values Q^λ of the DAG ServFlow is defined as the “bottleneck” value between the two branch paths (e.g., longer delay). If the nonlinear ServFlow includes more than two branch paths, the statistical QoS values are calculated recursively between every two branch paths. More details about the QoS calculation for the DAG ServFlow can be found in [8].

Using the aggregated statistical resource and QoS states, SpiderNet first selects all qualified ServFlows according the user's QoS requirements (i.e., Equation 2 in Section II) and resource requirements (i.e., Equation 3 in Section II). Then, SpiderNet sorts all the qualified ServFlows in the increasing order of the resource cost aggregation metric ψ^λ (i.e., Equation 1 in Section II). The qualified ServFlow with the smallest ψ^λ value is regarded as the best qualified ServFlow.

F. Algorithm Enhancements

We now present several enhancements to the basic distributed service composition scheme.

Caching composition probing results. Each overlay node can cache the qualified ServFlows found by recent composition probing operations. When a node v_i receives a composite service request with the same abstract function, it can avoid invoking the composition probing to find a new ServFlow if the cached ServFlow can satisfy the user's QoS constraints. Each cached ServFlow is only kept for a short period of time to assure the validity and optimality of the cached ServFlow. Moreover, before we use the cached ServFlow, we can send a single composition probe[5] along the cached ServFlow to validate whether it is still qualified. Thus, we can greatly reduce the composition probing overhead by eliminating unnecessary composition probing operations.

Pruning unqualified candidate ServFlows. We now describe how to reduce the probing overhead by pruning the searching branches along unqualified candidate ServFlows. When an overlay node receives a probe, it compares the current accumulated QoS and resource metric values with the user required QoS and resource constraints. If the satisfaction probabilities of the accumulated QoS metrics or the resource metrics already violate the user's requirements, the probe is dropped immediately[6]. Specifically, the overlay node drops a probe if: (1) $Pr(q^\lambda_k \leq \mathcal{C}^{q_k}) < \mathcal{P}^{q_k}, \forall k, 1 \leq k \leq m$; or (2) $Pr(r^{v_j}_k \geq \mathcal{C}^{s_i}_{r_k}) < \mathcal{P}^{s_i}_{r_k}$; or (3) $Pr(bw^{\wp_i} \geq \mathcal{C}^{\ell_i}_{bw}) < \mathcal{P}^{\ell_i}_{bw}$. Thus, we can greatly reduce the probing overhead by cutting off probe forwarding and spawning along those unqualified searching branches. If all probes are dropped during BCP, the probing source will automatically timeout and assume no qualified ServFlow can be found to satisfy the user's composite service request.

Differentiated probing quota allocation. In section III-B, we have described the uniform probing quota allocation scheme, which assumes that all service functions and branch paths are equally important. We now describe a differentiated probing quota allocation scheme, which considers the differences among service functions and branch paths in the function graph. First, we decide the

[5] If the ServFlow has k branch paths, then we need k composition probes to examine the ServFlow.

[6] Because the composition probing follows the function constraints specified by the function graph and QoS constraints are minimum optimal, the satisfaction probabilities will not be increased by further accumulations.

probing quota ratios among different service functions. Suppose the function graph ξ includes L service functions, $\{F_1, ..., F_L\}$. We use $\nu_i \cdot \alpha, 1 \leq i \leq L$, to represent the probing quota allocated to the service function F_i, where ν_i is the probing quota weight associated with the service function F_i. We can decide the value of ν_i based on different policies. For example, we can assign a higher weight to the service function that has more candidate service instances since it needs more probes to search different alternatives. Suppose a service function F_i can be mapped to σ_i different service instances. We can calculate ν_i as $\nu_i = \sigma_i / \sum_{j=1}^{L} \sigma_j$. If all service functions have the same number of duplicated service instances, we can assign larger ν_i to more critical service functions. We can achieve more efficient consumption of the probing budget β_0 by partitioning β_0 differentially among various service functions.

Second, we need to decide to how to share the probing budget among different branch paths in the non-linear ServFlow composition. We use $\varpi_i \cdot \beta_0, 1 \leq i \leq B$, to represent the probing budget allocated to the branch path τ_i, where ϖ_i represents the weight assigned to the branch path τ_i. Suppose the function graph ξ includes B branch paths $\tau_1, ..., \tau_B, B \geq 1$. Each branch path τ_i includes L_i service functions $\{F_{i_1}, ..., F_{i_{L_i}}\}$ with Z_i permutations. The number of probes spawned on each branch path is $Z_i \cdot \prod_{k=1}^{L_i} \nu_{i_k} \alpha^{L_i}$, which should be no larger than its probing budget share $\varpi_i \cdot \beta_0$. Thus, we can solve α and $\varpi_i, 1 \leq i \leq B$ based on one equation $\sum_{i=1}^{B} \varpi_i = 1$ and B inequalities: $Z_i \cdot \prod_{k=1}^{L_i} \nu_{i_k} \alpha^{L_i} \leq \varpi_i \cdot \beta_0, 1 \leq i \leq B$. Then, we can decide the probing quota allocated to each service function by $\nu_i \cdot \alpha$.

To implement the above differentiated probing quota allocation in the distributed service composition, we replace the *uniform probing budget partition* scheme, presented in Section III-D, with a *proportional probing budget partition* scheme, which is described as follows. When a service node v_i receives a probe whose probing budget is β and there are T next-hop service functions F_1,..., F_T, v_i proportionally divides β among T next-hop service functions as follows. Suppose there are k_i branch paths that are rooted at the service function F_i, which are denoted by τ_{i_1},..., $\tau_{i_{k_i}}$. Then, the probing budget allocated to F_i is decided by $\beta_i = \lfloor (\sum_{j=1}^{k_i} \varpi_{i_j} / \sum_{i=1}^{T} \sum_{j=1}^{k_i} \varpi_{i_j}) \cdot \beta \rfloor$, which means that the proportion of the probing budget allocated to F_i is decided by the ratio between weight sum of all branch paths rooted at F_i and the weight sum of all branch paths rooted at all next-hop service functions F_1,..., F_T. We now prove that the above proportional probing budget partition scheme can guarantee that each branch path τ_i receives its share of probing budget $\varpi_i \cdot \beta_0$.

Theorem 3.1: Suppose the function graph ξ includes B branch paths $\tau_1, ..., \tau_B, B > 1$. The proportional probing budget partition scheme can guarantee that each branch path τ_i is allocated with $\varpi_i \cdot \beta_0$ probing budget, $1 \leq i \leq B$.

Readers are referred to [8] for the proof details.

IV. Experimental Evaluation

In this section, we evaluate the performance of SpiderNet using both large-scale simulations and prototype running in wide-area network testbed PlanetLab [1].

A. Prototype Implementation and Evaluation

The SpiderNet prototype software is a multi-threaded running system written in about 13K lines of java code. There are 6 major modules: (1) *service lookup agent* is responsible for discovering the list of service instances, which is implemented on top of the Pastry software [16]; (2) *ServFlow generator* module executes the BCP protocol for QoS-aware service composition; (3) *session manager* maintains session information for current active sessions; (4) *data transmission* module is responsible for sending, receiving, and forwarding application data; (5) *overlay topology manager* maintains the neighbor set; (6) *monitoring* module is responsible for monitoring the network/service states of neighbors. As proof-of-concept, we also implemented a set of multimedia service components to populate our P2P service overlay. Each service component provide one of the following six functions: (1) embedding weather forecast ticker; (2) embedding stock ticker; (3) up-scaling video frames; (4) down-scaling video frames; (5) extracting sub-image; and (6) re-quantification of video frames.

Our experiments use 102 Planetlab hosts that are distributed across U.S. and Europe. The average replication degree of each multimedia service is about 15. We implement a customizable video streaming application on top of the SpiderNet service composition system. The customizable video streaming application allows the user to perform wide-area video streaming with desired transformations and enriched content. Figure 6 shows the screen shot of the customized video streaming application. Our experiments on the PlanetLab indicate that current SpiderNet prototype can setup a composite service session within several seconds, which is acceptable for long-lived streaming applications that usually lasts tens of minutes or several hours. The above service setup time can be reduced with further implementation improvement.

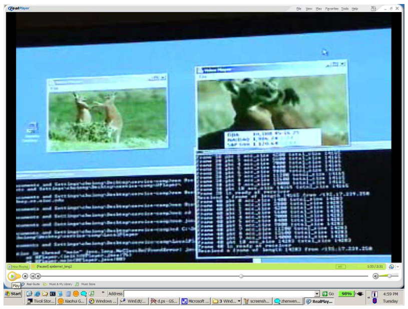

Fig. 6. Customizable video streaming using SpiderNet.

B. Simulation Results

In our simulation experiments, we first use the degree-based Internet topology generator Inet-3.0 [19] to generate a power-law graph with 3200 nodes to represent the Internet topology. We then randomly select a number of nodes as overlay nodes and connect them into a SON. The average node degree is $10\% \cdot |V|$. Once the node degrees are chosen, the nodes are connected into a topologically-aware overlay network using the Short-Long algorithm presented in [15]. To simulate the dynamic QoS values, we generate the dynamic QoS values using either uniform distribution function or normal distribution function. The histogram for each random variable includes 30 sample values and 10 bins. We choose the mean and deviation values based on real-world Internet service level agreement (SLA) contracts and the profiling results of fully implemented multimedia services. We simulate the IP-layer and overlay-layer data routing using the shortest path routing algorithm.

Each overlay node provides two service components. Each service component performs a service function that is randomly selected from $\lfloor \frac{|V|}{5} \rfloor$ service functions. Thus, the average service duplication ratio is $\frac{2 \cdot |V|}{|V|/5} = 10$, which conforms to our assumption that a service function can be mapped to a limited number of service instances. The function graph ξ of the request is randomly selected from 200 pre-defined templates, which include two to five service functions with one or two branches. The statistical resource and QoS requirements are uniformly distributed. The service session time is uniformly distributed within certain range. A QoS-aware service composition is said to be successful, if and only if the composed ServFlow (1) satisfies the function graph requirements, (2) satisfies the user's resource requirements (e.g., CPU, network bandwidth) , and (3) satisfies the user's QoS requirements (e.g., delay, data loss rate). The composition success rate is calculated by $\frac{SuccessNumber}{RequestNumber}$.

For comparison, we also implement three other common approaches: *optimal*, *random*, and *static* algorithms. The optimal algorithm uses unbounded network flooding, which exhaustively searches all candidate ServFlows to find the best qualified ServFlow. The random algorithm randomly selects a functionally qualified service component for each function node in the function graph. The static algorithm uses pre-defined service component for each service function in the function graph. Both random and static algorithms do not consider the user's QoS and resource requirements.

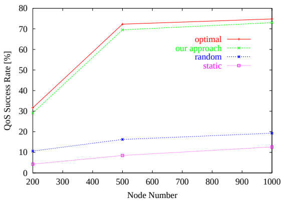

Fig. 7. Performance comparison.

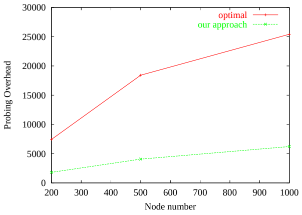

Fig. 8. Overhead comparison.

Figure 7 show the composition success rate achieved by different algorithms on the three different multimedia service overlays. The results illustrate that SpiderNet can consistently achieve near-optimal performance (i.e., $> 95\%$ of the optimal performance) on the three different service overlays. Compared to the random and static algorithms, SpiderNet can achieve as much as 300% better performance than the random algorithm and 400% better performance than the static algorithm. Moreover, SpiderNet presents much better scaling property than the random and static algorithms. When we increase the service overlay size from 200 nodes to 500 nodes, SpiderNet can achieve as much as 130% performance improvements by efficiently utilizing added resources while the random and static algorithm can achieve at most 50% improvements. The improvements from 500 nodes to 1000 nodes are not too much since the system resources of 500 nodes already meet the resource requirements of the workload. Figure 8 illustrates the overhead comparison between the optimal algorithm and the SpiderNet algorithm in the above experiment. The probing overhead is measured by the total number of probing messages generated per time unit. The results show that SpiderNet has much lower overhead than the optimal algorithm. The overhead increasing rate of the SpiderNet algorithm is also slower than that of the optimal algorithm as we increase the size of SON. More simulation results can be found in [8].

V. Conclusion

In this paper, we have presented SpiderNet, a fully decentralized QoS-aware multimedia service composition framework. SpiderNet integrates statistical QoS provisioning and automatic load balancing into the distributed service composition operation. Moreover, SpiderNet achieves *expressive* service composition by supporting directed acyclic graph composition topologies and exchangeable composition orders. Our prototype implementation demonstrated the feasibility and efficiency of the SpiderNet framework. In the future, We intend to investigate the probing budget tuning scheme to realize self-adaptive QoS-aware service composition. We also plan to extend the SpiderNet framework to support more composition relationships such as conditional branch, exclusive or, and conditional loop.

REFERENCES

[1] the PlanetLab. *http://www.planet-lab.org/*.

[2] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. *Proc. of ACM SOSP 2001, Banff, Canada*, October 2001.

[3] A. P. Black, J. Huang, J. Walpole, and C. Pu. Infopipes: an Abstraction for Multimedia Streaming. *Multimedia Systems (special issue on Multimedia Middleware), 8(5), pp.406-419*, 2002.

[4] A. Campbell and G. Coulson. A QoS Adaptive Transport System: Design, Implementation and Experience. *Proc. of ACM International Conference on Multimedia (ACM Multimedia 96)*, November 1996.

[5] S. Chen and K. Nahrstedt. An Overview of Quality-of-Service Routing for the Next Generation High-Speed Networks: Problems and Solutions. *IEEE Network Magazine, Special Issue on Transmission and Distribution of Digital Video, 12(6), pp. 64-79*, 1998.

[6] G. Coulson, S. Baichoo, and O. Moonian. A Retrospective on the Design of the GOPI Middleware Platform. *Multimedia Systems (special issue on Multimedia Middleware), 8(5), pp.340-352*, 2002.

[7] D. Ecklund, V. Goebel, T. Plagemann, and E. F Ecklund. Dynamic End-to-End QoS Management Middleware for Distributed Multimedia Systems. *ACM Multimedia Systems*, November 2002.

[8] X. Gu and K. Nahrstedt. Distributed Multimedia Service Composition with Statistical QoS Assurances. *IBM Research Technical Report, http://www.research.ibm.com/people/x/xgu/tmm-long.pdf*, December 2004.

[9] X. Gu, K. Nahrstedt, R. Chang, and Z. Shae. An Overlay Based QoS-Aware Voice-Over-IP Conferencing System. *Proc. of IEEE International Conference on Multimedia and Expo 2004, Taipei, Taiwan*, 2004.

[10] X. Gu, K. Nahrstedt, and B. Yu. SpiderNet: An Integrated Peer-to-Peer Service Composition Framework. *Proc. of IEEE International Symposium on High-Performance Distributed Computing (HPDC-13), Honolulu, Hawaii*, 2004.

[11] X. Gu, K. Nahrstedt, W. Yuan, D. Wichadakul, and D. Xu. An XML-based QoS Enabling Language for the Web. *Journal of Visual Language and Computing (JVLC), Special Issue on Multimedia Language for the Web, 13(1), pp. 61-95*, February 2002.

[12] E. Knightly and N. B. Shroff. Admission Control for Statistical QoS: Theory and Practice. *IEEE Network, 13(2):20-29*, March 1999.

[13] B. Raman and R. H. Katz. An Architecture for Highly Available Wide-Area Service Composition. *Computer Communications Journal, special issue on "Recent Advances in Communication Networking*, May 2003.

[14] B. Raman and R. H. Katz. Load Balancing and Stability Issues in Algorithms for Service Composition. *Proc. of IEEE INFOCOM 2003, San Francisco, CA*, April 2003.

[15] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. *Proc. of Infocom 2002,New York City, N.Y.*, 2002.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[17] H. Stark and J. W. Woods. Probability, random process, and estimation theory for engineers (second edition). *Prentice Hall, Upper Saddle River, NJ 07458*, 1986.

[18] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. *Proc. of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), Springer*, 1998.

[19] J. Winick and S. Jamin. Inet3.0: Internet Topology Generator. *Tech Report UM-CSE-TR-456-02 (http://irl.eecs.umich.edu/jamin/)*, 2002.

[20] D. Xu and K. Nahrstedt. Finding Service Paths in a Media Service Proxy Network. *Proc. of SPIE/ACM Multimedia Computing and Networking Conference (MMCN'02), San Jose, CA*, January 2002.

PLACE PHOTO HERE

Xiaohui Gu Xiaohui Gu is currently a research staff member at IBM T. J. Watson Research Center, Hawthorne, New York. She received her PhD degree in 2004 and MS degree in 2001 from the Department of Computer Science, University of Illinois at Urbana-Champaign. She received her BS degree in computer science from Peking University, Beijing, P.R. China. Her general research interests include distributed systems, computer networks, and operating systems with a current focus on stream processing, pervasive computing, service computing, and Grid computing. She received ILLIAC fellowship, David J. Kuck Best Master Thesis Award, and Saburo Muroga Fellowship from University of Illinois at Urbana-Champaign.

PLACE PHOTO HERE

Klara Nahrstedt Klara Nahrstedt is an associate professor at the University of Illinois at Urbana-Champaign, Computer Science Department. Her research interests are directed towards multimedia middleware systems, quality of service (QoS), QoS routing, QoS-aware resource management in distributed multimedia systems, and multimedia security. She is the coauthor of the widely used multimedia book 'Multimedia: Computing, Communications and Applications' published by Prentice Hall, the recipient of the Early NSF Career Award, the Junior Xerox Award, and the IEEE Communication Society Leonard Abraham Award for Research Achievements. She is the editor-in-chief of ACM/Springer Multimedia Systems Journal, and the Ralph and Catherine Fisher Associate Professor. Klara Nahrstedt received her BA in mathematics from Humboldt University, Berlin, in 1984, and M.Sc. degree in numerical analysis from the same university in 1985. She was a research scientist in the Institute for Informatik in Berlin until 1990. In 1995 she received her PhD from the University of Pennsylvania in the department of Computer and Information Science.