Distributed Multimedia Service Composition with Statistical QoS Assurances

Xiaohui Gu
IBM T. J. Watson Research Center
Hawthorne, NY 10532
{xiaohui}@us.ibm.com

Klara Nahrstedt
Univ. of Illinois at Urbana-Champaign
Urbana, IL 61801
{klara}@cs.uiuc.edu

Abstract

Service composition allows advanced distributed multimedia services to be automatically composed from atomic multimedia service components based on the user’s dynamic service requirements. However, existing service composition solutions lack user desired scalability, flexibility and multi-constrained quality-of-service (QoS) support. In this paper, we present a distributed multimedia service composition framework, called SpiderNet, to address the challenges. SpiderNet provides both statistical multi-constrained QoS assurances and efficient load balancing while composing services across wide-area networks. SpiderNet provides a fully decentralized QoS-aware service composition solution, which achieves scalability by eliminating expensive global states maintenance and centralized computation. To achieve flexibility, SpiderNet supports directed acyclic graph composition topologies. SpiderNet also explores exchangeable composition orders to enhance the QoS of composed services. We have implemented a prototype of SpiderNet and conducted extensive experiments using both large-scale simulations and wide-area network testbed. Our experimental results show the feasibility and efficiency of the SpiderNet service composition framework.

Index Quality of Service, Multimedia Middleware, Composable Multimedia Service Infrastructure, Service Composition, Multimedia Service Overlay Network

1 Introduction

Emerging advanced distributed multimedia services, such as voice-over-IP conferencing [27] 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 connects thousands of media servers and proxies into a service-oriented application-level

*This work was supported by the NASA grant under contract number NASA NAG 2-1406, NSF grant under contract number 9870736, 9970139, and EIA 99-72884EQ. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, NASA or U.S. Government.
overlay network called multimedia service overlay illustrated by Figure 1. Each media server or proxy can provide not only media content but also various media processing functions such as media transcoding, audio mixing, and information embedding as well as application-level data routing. The multimedia service overlay allows advanced distributed multimedia services to be automatically created from distributed multimedia service components. Compared to the conventional centralized approach, the multimedia service overlay has a number of salient advantages. First, it achieves better failure resilience by replicating service components at geographically dispersed locations. Second, it provides better quality-of-service (QoS) by dynamically select service instances that are closer to the clients. Third, it achieves better scalability by distributing multimedia service provisioning load onto different media servers and proxies.

We now describe some application examples to further motivate why a composable multimedia service overlay is desirable. First, we could use the multimedia service overlay to provide pervasive content distribution that often requires dynamic on-demand adaptations or transformations based on different computing context. For example, a Japanese person wants to listen to CNN sports news on his cell-phone. The system can automatically compose an html-to-speech service and English-to-Japanese language translation service to transform the original web content into the user desired content format, illustrated by the composed multimedia service between client A and client B.

The second application to support distributed sensor monitoring where thousands of sensors are deployed across wide-area networks. Service composition can provide various stream processing functions such as aggregation, filtering, and tracking to efficiently deliver useful information to the end-user such as client C in Figure 1. Another important application is to assist crisis action planning, which requires on-demand customized media streaming with specialized annotations during a crisis event such as September 11. The same media source needs to be augmented, annotated, and filtered based on the needs of different user roles, such as fire-fighters, medical people, and the governor.

A fundamental challenging problem in such a multimedia service overlay is to provide scalable QoS-aware
service composition, which can automatically compose distributed multimedia service components based on the user’s
dynamic function and QoS requirements. Recently, several research projects (e.g., SAHARA [37, 38], SPY-Net [51],
CANS [22], Infopipes [9]) have addressed the service composition problem. However, most existing solutions present
three major limitations. First, they all adopt a centralized approach to service composition, which presents serious
scalability problems. Second, they lack systematic QoS management that are especially important for distributed
multimedia applications. On the other hand, existing QoS management solutions (e.g., [16, 14, 47, 20, 19]) are not
readily applicable to service composition that needs to meet application-specific service requirements such as service
function constraints and inter-service dependency and commutation relations. Third, most existing solutions only
support linear service composition with fixed composition order, which greatly limits the applicability and efficiency
of service composition.

In this paper, we present a novel QoS-aware service composition framework called SpiderNet to address the
above challenges. SpiderNet proposes a bounded composition probing (BCP) scheme to achieve scalable QoS-aware
service composition in a fully distributed fashion. The BCP process simultaneously serves three purposes. First,
it discovers available service components based on the user’s service function requirements and composes dynam-
ically discovered service components based on the inter-service dependency and commutation relations. Second,
it performs on-demand selective states collection to eliminate the expensive periodical global states maintenance.
Third, it checks inter-component QoS consistencies [26] to assure a correct composed service. Fourth, it performs
intelligent parallel searching of multiple candidate service compositions to examine their QoS conditions and resource
availability. Compared to the conventional network probing [17, 12, 13], the composition probing considers not only
network connectivity and link states, but also inter-service dependency and commutation relations, QoS consistency
requirements, and various service states of candidate service components.

SpiderNet provides statistical multi-constrained QoS assurances for the composed distributed multimedia services.
Compared to deterministic QoS provisioning, statistical QoS provisioning can achieve better resource utilization [33]
and tolerate information dynamics in wide-area networks [31]. On the other hand, statistical QoS provisioning is
suitable for multimedia applications that are resilient to small QoS violations. In addition to satisfy individual QoS
requirements, SpiderNet also performs load balancing to optimize the overall resource utilization of the multimedia
service overlay.

In contrast to previous solutions that only support linear composition structure with fixed composition order, Spi-
derNet provides more expressive service composition interface. SpiderNet supports directed acyclic graph composition
topologies. Thus, the composed services can be more efficient with parallel execution of service functions instead of
strict pipelined chaining of service functions. Moreover, SpiderNet allows the application to specify exchangeable
composition orders that can be used for QoS enhancement. For example, in a composed media content delivery
service, distributed media objects can be composed in different permutation orders. These different composition
orders provide the same set of service functions to the end-user but can have different resource requirements and end-to-end QoS assurances. Thus, by exploring different composition orders, SpiderNet can improve the QoS provisioning and resource utilization in composed multimedia service delivery.

We have conducted a detailed experimental evaluation about the performance and scalability of the SpiderNet framework. We have implemented a prototype of SpiderNet and conducted extensive experiments by evaluating the prototype on both large-scale simulation testbed and wide-area network testbed PlanetLab [3]. The experimental results show that SpiderNet can achieve near-optimal QoS-aware service composition performance with low overhead. Compared to the centralized approach that requires global states maintenance, SpiderNet can reduce the system overhead by more than one order of magnitude. The rest of the paper is organized as follows. In Section 2, we present the SpiderNet system model. In Section 3, we present the distributed service composition design in detail. In Section 4, we present experimental results. We review related work in Section 5. Finally, the paper concludes in Section 6.

2 System Model

The SpiderNet system is designed as a distributed middleware infrastructure illustrated by Figure 2, which consists of three conceptual layers. The top layer represents the user’s composite service request. The middle layer represents the instantiated distributed service called ServFlow, which satisfies the user’s service request. The bottom layer represents the multimedia service overlay, which consists of thousands of media servers and proxies interconnected.
into a service-oriented overlay network layered on top of the IP-layer network. In this section, we first present the detailed system models for the above three layers. Then, we formally define the QoS-aware service composition problem, followed by the key assumptions made by the SpiderNet system. We summarize the notations in Table 1 for later references.

### 2.1 Composite Service Request

The user can specify a composite service request using a function graph and a QoS requirement vector, which is denoted by \( \Upsilon = (\xi, Q^{target}) \), where \( \xi \) represents the function graph and \( Q^{target} \) represents the user’s QoS requirements. The function graph specifies required service functions \( (F_i) \) and the inter-service dependency and commutation relations, which is illustrated by the top tier in Figure 2. 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 \( F_1 \xrightarrow{} F_2 \). The commutation relation between \( F_1 \) to \( F_2 \) means that the composition order of \( F_1 \) and \( F_2 \) can be exchanged, which will not affect the aggregated function of the composed service delivered to the end-user. However, the QoS provisioning or resource requirements of the composed service can be affected. We formally define the function graph as follows,

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

We use \( Q^{target} = [(C_1^q, P_1^q), \ldots, (C_m^q, P_m^q)] \) to define the user’s statistical QoS requirements for the composed service, where \( (C_i^q, P_i^q) \) specifies the bound \( C_i^q \) and the satisfaction probability\(^1\) \( P_i^q \) for the metric \( q_i \) that

\[ q_i \]

\[^1\]The satisfaction probability is defined as the probability when the random variable \( q_i \) is less or equal to \( C_i^q \), assuming \( q_i \) is minimum-optimal.
represents a QoS metric such as service delay and data loss rate. For example, a composite service request for a mobile video streaming service can be specified as follows, \( \Upsilon = \{(\{F_1 = VideoServer, F_2 = EmbedCaption, F_3 = ImageScaler, F_4 = ColorFilter, F_5 = VideoPlayer\}, \{F_1 \rightarrow F_2, F_2 \rightarrow F_3, F_4 \rightarrow F_5\}, \{F_3 \sim F_4\}), \langle [10\, ms, 98\%]_{\text{Delay}}, [5\%, 99\%]_{\text{LossRate}} \rangle \). Users can either directly specify the composite service request using extensible markup language (XML) or use available visual specification environment such as QoSTalk [30, 48].

### 2.2 Multimedia Service Overlay

A multimedia service overlay network consists of a set of overlay nodes \( (v_i) \) inter-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 neighbor nodes via overlay links. The neighbors of each node can be randomly selected or selected based on pre-defined metrics such as round trip delay [40], or based on the node’s own benefits [25]. However, the design of the SpiderNet service composition framework is orthogonal to the underlying overlay network topology. We have also conducted experiments on different overlay topologies, which will be presented in Section 4. We formally define the multimedia service overlay as follows,

**Definition 2.2.** A multimedia service overlay is described by a weighted directed graph \( G = (V, E) \), where \( V \) represents the set of \( |V| \) peers, denoted by \( v_i, 1 \leq i \leq |V| \), and \( E \) represents the set of \( |E| \) overlay links, denoted by \( e_j, 1 \leq j \leq |E| \).

Application-level data relaying [7] is required between two overlay nodes that are not directly connected. For example, in Figure 2, the data sent by \( v_1 \) to \( v_4 \) needs to be relayed by \( v_2 \). Similar to the overlay network topology, the overlay data routing is also orthogonal to the design of the SpiderNet service composition framework. The current design of SpiderNet uses shortest path routing algorithm using network delay metric for overlay data routing. 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.) \( \rho_r \) 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 \( [x - \frac{dx}{2}, x + \frac{dx}{2}] \) is \( Y \), then we have \( \rho_r(x) \approx \frac{Y}{Z} \). Each node \( v_i \) also maintains statistical bandwidth availability \( bw^\ell_j \) for its adjacent overlay links \( \ell_j \). For scalability, each node maintains the above histograms locally, which are not disseminated to other overlay nodes.

---

\[2\] Note that bandwidth requirement is regarded as a resource requirement metric, which will be described in Section 2.3.
2.3 Distributed Multimedia Service

A multimedia service component \( s_i \) is a self-contained multimedia application unit providing a certain functionality (e.g., media transcoding), which is illustrated by Figure 3 (a). Each service component has several input and output ports for 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 \( Q^{in} = [q_{in}^1, ..., q_{in}^d] \), and output quality properties of the service component, denoted by \( Q^{out} = [q_{out}^1, ..., q_{out}^d] \); and (3) adaptation policies \( \Gamma = \{\gamma_1, ..., \gamma_l\} \), where \( \gamma_i \) is expressed by an if-condition-then-action construct. The dynamic meta-data of a service component describe its fluctuating performance conditions, such as current service delay. We use statistical QoS vector \( Q^s = [q_1^s, ..., q_m^s] \) 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 \( \rho_{q_k} \). We use \( Pr(q_k \leq C_{q_k}) \) to define the satisfaction probability that the dynamic value of \( q_k \) is no larger than the required upper bound \( C_{q_k} \). We formally define the service component as follows,

**Definition 2.3.** A service component is defined as \( s_i = (F_i, \text{Code}, SMD, DMD) \), where \( F_i \) represents the provided service function, \( \text{Code} \) defines the service implementation, \( SMD \) represents static meta-data, and \( DMD \) represents dynamic meta-data.
When we compose two service components, we need to address two key issues, illustrated by Figure 3 (b). First, we need to check the QoS consistencies [26] 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 \( Q_{\text{in}} \) and \( Q_{\text{out}} \) of the two composed service components are consistent. Second, we check whether 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 model the dynamic QoS metrics statistically, the accumulation of statistical QoS metrics means convolution [43] between them.

We describe a composite distributed multimedia service using a directed acyclic graph called ServFlow \((\lambda)\), illustrated by the middle tier in Figure 2. 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 \((\ell_i)\) is mapped to an overlay path \((\varphi_i = e_1 \rightarrow e_2 \ldots \rightarrow e_k)\) by the overlay data routing layer\(^4\). We formally define the ServFlow as follows,

**Definition 2.4.** A ServFlow is defined as \(\langle S, L \rangle\), \(S = \{s_k | 1 \leq k \leq |S|\}\), \(L = \{\ell_k | \ell_k = s_i \rightarrow s_j, 1 \leq k \leq |L|\}\).

For example, the ServFlow shown in Figure 2 can be described as \(\lambda = \langle \{s_1, s_7, s_9, s_11\}, \{\ell_1/\varphi_1, \ell_2/\varphi_2, \ell_3/\varphi_3, \ell_4/\varphi_4, \ell_5/\varphi_5\}\rangle\), where \(\varphi_1 = e_1, \varphi_2 = e_2 \rightarrow e_3, \varphi_3 = e_4, \varphi_4 = e_5 \rightarrow e_6, \) and \(\varphi_5 = e_7\). If an overlay node contributes multimedia services on a ServFlow \(\lambda\), it is called a service node. If an overlay node only performs application-level data relaying on \(\lambda\), it is called a relay node. For example, in Figure 2, \(v_1\) is a service node and \(v_2\) is a relay node.

We define the QoS of a ServFlow as the accumulations of the QoS metrics of the ServFlow’s constituent service components and service links.

We define the resource cost aggregation metric to describe the load balancing property of an instantiated ServFlow. The resource cost aggregation metric, denoted by \(\psi^\lambda\), is the weighted sum of ratios between resource requirements of the service components/service links between resource availabilities on the corresponding overlay nodes/overlay paths. We define \(C_{s_i}^{\text{rs}}\) and \(P_{s_i}^{\text{rs}}\) 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 \(C_{\ell_{bw}}^{\text{rs}}\) and \(P_{\ell_{bw}}^{\text{rs}}\) to denote the required threshold and satisfaction probability for the network bandwidth on the service link \(\ell_i\), respectively. The resource requirements of a service component depend on its implementations and the communication volume between them. 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

---

\(^3\)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. Although we could have used a more sophisticated model that accounts for all the correlation more accurately, we decided to use a simpler model to avoid over-complicated computations. Our experimental results (to be presented in Section 4) show that the simplified model can suffice our QoS-aware service composition goals.

\(^4\)In the current implementation of SpiderNet, the overlay data routing layer uses shortest path routing algorithm considering network delay metric only.
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 $M_{v_j}^{r_i}$ to denote mean availability of $i^{th}$ end-system resource type on the overlay node $v_j$. We use $M_{bw}^{P_i}$ to denote the mean availability of the bandwidth on the overlay path $P_i$, which is defined as the minimum mean available bandwidth among all overlay links $e_i \in P$. The mean values can be calculated from the p.d.f.’s of the corresponding statistical metrics. Hence, the resource cost aggregation metric $\psi^\lambda$ is defined as follows,

$$\psi^\lambda = \sum_{s_i/v_j \in \lambda} \sum_{k=1}^{n} w_k \cdot \frac{C_{s_i}^{r_k}}{M_{r_k}^{v_j}} + w_{n+1} \cdot \sum_{\ell_i/P_i \in \lambda} \frac{C_{bw}^{P_i}}{M_{bw}^{P_i}}, \sum_{k=1}^{n+1} w_k = 1, 0 \leq w_k \leq 1, 1 \leq k \leq n + 1 \quad (1)$$

$w_k, 1 \leq k \leq n + 1$ represents the importance of different resource types. We can customize $\psi^\lambda$ 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.

### 2.4 Problem Description

We formulate the quality-aware service composition (QSC) problem in P2P service overlay as a two dimensional graph mapping problem, which is illustrated by Figure 4. In one dimension, we can derive different composition patterns from the original function graph by considering the commutation links. In the other dimension, we can map each service function into different functionally duplicated service components because of the inherent redundancy property of a large-scale multimedia service overlay. 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.,

---

5 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).
CPU, memory). For example, in Figure 4, function $F_1$ can be mapped to two duplicated service components $s_1$ and $s_2$.

Thus, we can derive different service graphs from the function graph by considering the above two dimensions. The QSC problem is to find the best mapping from the function graph to the best qualified service graph that satisfies the user’s multi-constrained QoS requirements $Q^\text{req}$ and achieves best load balancing in the current multimedia service overlay. We formally define the QSC problem as follows,

**Definition 2.5. QoS-aware service composition (QSC) problem.** Given a composite service request $\Upsilon = (\xi, Q^\text{target})$ and a multimedia service overlay $G = (V, E)$, the QSC problem is to map $\xi$ into the best qualified ServFlow $\lambda$, such that $\lambda$ minimizes $\psi^\lambda$ subject to two constraints:

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

$$Pr(r^\lambda_{v_j} \geq C^{r_{e_k}}) \geq P^{r_{e_k}}, \land Pr(bw^\lambda_i \geq C^{bw_i}) \geq P^{bw_i}, \forall k, 1 \leq k \leq n, \forall s_i/v_j \in \lambda, \forall \ell_i/\ell_j \in \lambda \tag{3}$$

However, the QSC problem is NP-hard, which is proved as follows. Thus, our goal is to provide efficient, fully decentralized service composition solution that is suitable for large-scale multimedia service overlay.

**Theorem 2.1. QSC problem is NP-hard.**

**Proof.** We prove this by showing that the multi-constrained path selection (MCP) problem, which is known to be NP-hard [24], maps directly to a special case of the QSC problem. The MCP problem is identical to the following special case of the QSC problem. Let us assume: (1) each overlay node provides the same set of service components; (2) all overlay nodes/links have infinite resources; and (3) the function graph has a path structure. Thus, finding the best qualified ServFlow $\lambda$ in the multimedia service overlay $G$ is identical to finding a multi-constrained path in $G$.

Thus, the QSC problem is also NP-hard.

2.5 Assumption

First, we assume that overlay nodes are cooperative and trust-worthy. Although the current implementation of SpiderNet does not provide security support, we could associate a cryptographically-protected credential with each composition probe [23]. The SpiderNet node can verify the probe according to its carried credentials. The probe is processed only if the security check is successful. Second, we assume that the function graph is given by the user to the SpiderNet. We have implemented a visual programming tool that allows the user to specify the function graph easily. The function graph can also be automatically generated by the function planning tools such as SWORD [36] based on the user’s composite function requirements. Third, we assume that the resource requirements of each service instance are known to its provisioning SON node through off-line or on-line profiling [5, 46]. Fourth, we assume that each overlay node can monitor both network and end-system resource availabilities using passive or proactive measurement
tools [32]. Finally, we assume that all service providers adopt a standardized service component architecture and specification, which can be based on the OMG component model [4] and OGS A service infrastructure [2].

3 System Design

In this section, we present the distributed service composition design details. First, we present the composition protocol that is used by SpiderNet to perform distributed service composition. Second, we introduce the concepts of probing budget and probing quota to achieve efficient controllable composition probing. Third, we describe the per-hop probe processing algorithm. Fourth, we present the optimal service composition selection algorithm. Finally, we discuss several enhancements to the basic distributed service composition solution.

3.1 Distributed Service Composition Protocol

SpiderNet executes a bounded composition probing (BCP) protocol to perform distributed service composition, which aims at addressing three key challenges: (1) scalability, where the system should scale well in the presence of a large number of overlay nodes; (2) multi-dimensional constraint satisfaction, where the system should be able to find a qualified ServFlow satisfying multi-dimensional constraints including resource constraints, QoS constraints, and service functionality constraints; and (3) flexibility, where the system should be able to provide expressive composition interface such as supporting directed acyclic graph composition topologies and commutable composition orders. Given a service composition request, the application sender\(^6\) invokes the BCP protocol, which is illustrated by Figure 5. The BCP protocol includes four major steps:

**Step 1. Initialize the probe.** The source first generates a composition probing message, called probe, which is illustrated by Figure 5 (a). 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** \((\beta)\) that defines how many probes we could

---

\(^6\)For simplicity, we use a unicast streaming application as an example.
Input: composite service request \(\langle \xi, Q^{\text{target}} \rangle\), probing budget \(\beta\);
Output: best qualified ServFlow from the source to the destination.

A. The source node generates a probe \(P\)
   (A.1) Initialize \(P\) with \(\xi, \beta, Q^{\text{target}}\)
   (A.2) Derive first-hop service components \(s_1, \ldots, s_k\)
   (A.3) Spawn \(k\) new probes from \(P\)
   (A.4) Route them toward the first-hop candidate service components

B. Per-hop probe processing at an intermediate node \(v_i\), next service hop \(s_t\) at \(v_t\)
   (B.1) if \(v_i\) is a relay node
      (B.1.1) Derive next data routing hop \(v_k\) towards \(v_t\) using local overlay data routing table
      (B.1.2) Update \(P\) with its local QoS and resource info. about the overlay link \(e(v_i, v_k)\)
      (B.1.3) Forward \(P\) to \(v_k\)
   (B.2) if \(v_i\) is a service node, providing service component \(s_i\)
      (B.2.1) Check Resource/QoS conformance
      (B.2.2) if ServFlow is qualified
         (B.2.2.1) Perform soft resource allocation
         (B.2.2.2) Derive next-hop service functions with their keywords using \(\xi\)
         (B.2.2.3) Check QoS consistency between \(s_i\) and candidate service components
         (B.2.2.4) Spawn new probes and allocate probing budgets for new probes
         (B.2.2.5) Updates new probes with local resource and QoS info. of \(s_i\)
         (B.2.2.6) Updates new probes and route them in the same way as step (B.1)
      (B.2.3) else Drop the received probe \(P\)

C. The destination node selects the best qualified ServFlow

Figure 6: Distributed service composition algorithm.

use for a composition request. We will introduce the probing budget in more detail in Section 3.2.

**Step 2. Hop-by-hop probe processing.** Each peer processes a probe independently using only local information until the probe arrives at the destination, illustrated by Figure 5 (b). The goal of hop-by-hop distributed probe processing is to collect needed information and perform intelligent parallel searching of multiple candidate service graphs. We will describe this step in detail in Section 3.3.

**Step 3. Optimal composition selection.** The destination collects the probes for a request with certain timeout period, illustrated by Figure 5 (c). It then selects the best qualified service graph based on the resource and QoS states collected by the probes. We will discuss this step in detail in Section 3.4.

**Step 4. Setup service session.** Finally, the destination sends an acknowledge message along the reversed selected service graph to confirm resource allocations and initialize service components at each intermediate peer, illustrated by Figure 5 (d). Then the application sender starts to stream application data units along the selected service graph. If no qualified service graph is found, the destination returns a failure message to the source directly. Figure 6 shows the pseudo-code of distributed service composition algorithm.
3.2 Efficient Composition Probing

To achieve efficient composition probing, we first introduce the concept of probing budget that allows us to precisely control the number of probes we could use for 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 service graphs, which allows us to find a better qualified service graph. Thus, our solution 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. We can also adaptively adjust the probing budget based on the user feedbacks and historical information.

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 each service function, dividing $\beta_0$ among all candidate service components can quickly use up the probing budget at early stage of the composition probing. To address the problem, we associate a probing quota ($\alpha_i$) with each service function $F_i$ to limit the number of service components that can be probed 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 3.5. Let us assume that we are given a function graph $\xi$ includes $k$ branch paths $\tau_1, ..., \tau_k$, each of which includes $L_i$ service functions with $Z_i \geq 1, 1 \leq i \leq k$ different permutations (i.e., composition patterns). If each service function is associated with the same probing quota $\alpha$, then the total probes generated on the branch path with $L_i$ service functions and $Z_i$ permutations is $Z_i \cdot \alpha^{L_i}$. Thus, we can derive $\alpha$ based on the following inequalities:

$$Z_i \cdot \alpha^{L_i} \leq \frac{\beta_0}{k}, \quad 1 \leq i \leq k.$$  \hspace{1cm} (4)

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

3.3 Per-hop Probe Processing

We now describe the details of the per-hop probe processing algorithm at a service node (i.e. Step 2 in the BCP protocol), which is illustrated by Figure 7. The per-hop probe processing mainly includes six sub-steps:

**Step 2.1: 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 service graph already violate the user’s requirements. If the accumulated QoS and resource values already violate the user’s requirements, the probe is dropped immediately.
Figure 7: Per-hop probe processing at a service node.

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.

**Step 2.2: Derive next-hop service functions.** Next, the service node derives the next-hop service functions according to the dependency and commutation relations in the function graph. All the functions dependent on the current function are considered as next-hop functions. For example, in Figure 7, the current function $F_1$ has two dependent next-hop functions $F_2$ and $F_3$. For each next-hop function $F_k$ derived above, if there is an exchange link between $F_k$ and $F_1$, then $F_1$ is also considered as a possible next-hop function. For example, in Figure 7, since $F_3$ can be commutated with $F_1$, $F_4$ is also considered as the next-hop of $F_1$. 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 $F_1$. In the probe for $F_k$, we modify its function graph by replacing $F_k \sim F_1$ with $F_k \sim F_l$. In the probe for $F_1$, we modify its function graph by first replacing $F_k \sim F_1$ with $F_l \sim F_k$, and then letting $F_k$ inherit all the relations of $F_1$. For example, in Figure 7, in the probe for $F_2$, its function graph is modified by replacing $F_2 \sim F_3$ with $F_2 \sim F_3$. In the probe for $F_3$, we modify its function graph by replacing $F_2 \sim F_3$ with $F_3 \sim F_2$, and replacing $F_3 \sim F_5$ with $F_2 \sim F_3$.

**Step 2.3: Discover candidate service components.** The service node then discovers candidate service components for all the next-hop service functions derived above. For example, in Figure 7, the service node discovers four...
service components for function $F_2$. To achieve scalable fully decentralized service discovery, we implement the dynamic service component lookup by retrieving the service components’ static meta-data (e.g., locations, $Q^{in}$, $Q^{out}$) from a decentralized service directory built on top of the distributed hash table systems [41, 44, 39]. When a service node $v_k$ wants to provide a service component $s_k$ with function name $F_k$, SpiderNet registers the service component by storing its static meta-data including locations and input/output QoS properties into the DHT system. The static meta-data is inserted into or retrieved from DHT using a unique identifier, called key. The key is routed by the DHT system to a designated node that is responsible for storing the static meta-data. To implement the one-to-many mapping required by the function name based service components discovery, SpiderNet adds another indirection into the DHT system. To store a static meta-data item, the node $v_k$ provisioning $s_k$ first generates a key by applying a secure hash function on the service function name $F_k$. Then, $v_k$ stores the service component’s static meta-data into the DHT using the key. The DHT system will then route the static meta-data of duplicated service components with the same function name to the same designated node $v_r$. When another overlay node, such as $v_i$, wants to retrieve the static meta-data of duplicated service components with a specific function name $F_k$, it issues a query to the DHT system using the key generated from the same secure hash function on $F_k$. The DHT system routes the query message to the responsible node $v_r$, which returns the static meta-data items to the requesting node $v_i$.

**Step 2.4: Check QoS consistency.** Based on the service discovery results, the service node then performs QoS consistency check [26] 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 define the parametric consistency relation as follows,

**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.

The single value static QoS parameters include media format (JPEG, MPEG, etc.), resolution (1024*768 pixels) and others. The range value static QoS parameters include frame rate ([10fps,30fps]), tracking precision ([0,100%]) and others. We can then check the parametric consistency between $Q^{out}$ of the current service component and $Q^{in}$ of the next-hop service component based on the above definition.

Besides to check the parametric consistencies, we also need to 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 $\pi$ 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. If the condition hypercubes of the two adaptation policies are equal, illustrated by Figure 8 (a), then the two adaptation policies will be triggered simultaneously. Thus, we need to check whether the two service components still satisfy the parametric consistency relation after applying the two adaptation policies, respectively. If two adaptation policies’ condition hypercubes do not have any intersection, illustrated by Figure 8 (b), then the two adaptation policies are never triggered at the same time. Thus, we need to check whether the new $SQ^{out}$ of $s_a$ after adaptation still satisfies the old $SQ^{in}$ of $s_b$ or the old $SQ^{out}$ of $s_a$ satisfies the new $SQ^{in}$ of $s_b$ after the adaptation. If the two hypercubes are partially overlapped, illustrated by Figure 8 (c), then we need to consider three possible scenarios, namely adaptation applied on $s_a$ only, or adaptation applied on $s_b$ only, or adaptations applied on both. Finally, we need to consider the cases where one hypercube is a subset of the other, illustrated by Figure 8 (d) and (e). Under these circumstances, we need to check upon two possible scenarios, namely adaptations happen on both service components or adaptation happens on the superset node only. We formally define the adaptation policy set compatibility relation 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 $\gamma$. Given two adaptation policy sets $\Gamma_{s_a} = \{\gamma_{a_1}, \ldots, \gamma_{a_A}\}$ and $\Gamma_{s_b} = \{\gamma_{b_1}, \ldots, \gamma_{b_B}\}$, 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) $\pi_a = \pi_b \Rightarrow \gamma_{a_i}(SQ_{s_a}^{out}) \leq \gamma_{b_j}(SQ_{s_b}^{in})$; (2)
\(\pi_a \cap \pi_b = \emptyset \Rightarrow \gamma_i(SQ_{s_a}^{out}) \leq SQ_{s_b}^{in} \wedge SQ_{s_a}^{out} \leq \gamma_j(SQ_{s_b}^{in});\) (3) \(\pi_a \cup \pi_b \neq \emptyset \wedge \pi_a \nsubseteq \pi_b \wedge \pi_b \nsubseteq \pi_a \Rightarrow \gamma_i(SQ_{s_a}^{out}) \leq SQ_{s_b}^{in} \wedge SQ_{s_a}^{out} \leq \gamma_j(SQ_{s_b}^{in})\) and \(SQ_{s_a}^{out} \leq \gamma_b(SQ_{s_b}^{in});\) (4) \(\pi_b \subseteq \pi_a \Rightarrow \gamma_a(SQ_{s_a}^{out}) \leq SQ_{s_b}^{in} \wedge \gamma_a(SQ_{s_a}^{out}) \leq \gamma_b(SQ_{s_b}^{in})\).

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 \((SQ_{s_a}^{in}, SQ_{s_a}^{out}, \Gamma_{s_a})\) and \((SQ_{s_b}^{in}, SQ_{s_b}^{out}, \Gamma_{s_b})\). We define that \(s_a\) is QoS consistent with \(s_b\) \((s_a \Leftrightarrow s_b)\), if and only if (1) \(SQ_{s_a}^{out} \leq SQ_{s_b}^{in}\) and (2) \(\Gamma_a \cap \Gamma_b\).

In SpiderNet, static meta-data items are described using the XML-based markup language HQML [30]. SpiderNet check the QoS consistency between two service components using the HQML syntactic and semantic parsers [30] according to the above Inter-component QoS consistency definitions. For example, in Figure 7, the service node finds three qualified next-hop service components (i.e., \(s_3\), \(s_5\), \(s_6\)) for the function \(F_2\). 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 \(\beta_k\) denote the available probing budget for \(F_k\) decided by step 2.2. Let \(\alpha_k\) 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 \([\frac{\beta_k}{K}]\). However, if \(I/K\), 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 \([\frac{\beta_k}{K}]\). For example, in Figure 7, the service node selects two service components for each next-hop service function.

**Step 2.6: Set probe content with statistical information.** 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 $Q^\lambda$ of the probed ServFlow $\lambda$ using the convolution between its old values recorded in $P$ and $Q^{s_i}$ of the current service component $s_i$ as follows,

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

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

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

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

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 $Q^\lambda$ using the convolution between old $Q^\lambda$ values and $Q^{e_k}$,

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

$v_i$ then updates mean value and satisfaction probability for the available bandwidth on the overlay path $\ell_k$ to the selected next-hop service instance as follows,

$$
M_{bw}^{\ell_k} = \min(M_{bw}^{\ell_k}, \int_{-\infty}^{+\infty} x \rho_{bw}^{\ell_k}(x) \, dx)
$$

$$
Pr(bw_{\ell_k} \geq C_{bw}^{\ell_k}) = Pr(bw_{\ell_k} \geq C_{bw}^{\ell_k}) \cdot \int_{C_{bw}^{\ell_k}}^{+\infty} \rho_{bw}^{\ell_k}(x) \, dx
$$

where $M_{bw}^{\ell_k}$ and $Pr(bw_{\ell_k} \geq C_{bw}^{\ell_k})$ are initialized as $+\infty$ and 1, respectively.

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. Thus, 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 $e_k$ towards the next-hop service node specified in $P$. The overlay link $e_k$ is derived according to the local overlay data routing table. The relay node first updates the accumulated QoS values $Q^\lambda$ of the probed ServFlow $\lambda$ using Equation 8. Next, the relay node updates the mean available bandwidth and bandwidth satisfaction probability in $P$ using Equation 9 and Equation 10, respectively. Finally, the relay node forwards the probe along $e_k$ towards the next-hop service node.
3.4 Optimal Service Composition Selection

At the destination node, SpiderNet selects the best qualified service graph 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 9, each probe traverses either the branch path performing service functions $F_1 \to F_2 \to F_4 \to F_5$ or the branch path performing service functions $F_1 \to F_3 \to F_4 \to F_5$. 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 9, 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 9, 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., $\mathcal{M}_{r_{i_k}}$, $\mathcal{M}_{r_{j_k}}$) of the DAG ServFlow are the union of its constituent branch paths. The statistical QoS values $Q^\lambda$ of the DAG ServFlow is defined as the worse value between the two branch paths. Thus, we calculate $\rho_{q_i}^\lambda$ for the DAG ServFlow as follows,

$$
\rho_{q_i}^\lambda(u) = \rho_{q_1}^\lambda(u) \int_{-\infty}^{u} \rho_{q_1}(x) dx + \rho_{q_2}^\lambda(u) \int_{-\infty}^{u} \rho_{q_2}(x) dx, \quad 1 \leq i \leq m
$$

(11)

where $\rho_{q_1}^\lambda(u)$ and $\rho_{q_2}^\lambda(u)$ represent the p.d.f.'s of $q_i$ for the two branch paths, respectively. If the non-linear ServFlow

---

7Assuming all QoS metrics are minimum optimal (e.g., delay, loss rate), we calculate the QoS values of the ServFlow as the larger values of the two branches. If the QoS metric are maximum optimal (e.g., throughput), we can transform the metric into minimum optimal by applying the negation or inversion function.
includes more than two branch paths, the statistical QoS values are calculated recursively between every two branch paths. Using the above derived statistical resource and QoS states, \( v_d \) first selects all qualified ServFlows according the user’s QoS requirements (i.e., Equation 2 in Section 2) and resource requirements (i.e., Equation 3 in Section 2). Then, \( v_d \) sorts all the qualified ServFlows in the increasing order of the resource cost aggregation metric \( \psi^\lambda \) (i.e., Equation 1 in Section 2). The qualified ServFlow with the smallest \( \psi^\lambda \) value is regarded as the best qualified ServFlow.

### 3.5 Enhanced Distributed Service Composition

We can improve the above basic distributed service composition solution in three aspects: (1) caching, (2) pruning, and (3) differentiated probing quota allocation, which are described as follows,

**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\(^8\) 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\(^9\). Specifically, the overlay node drops a probe if: (1) \( \Pr(q_k^\lambda \leq C_{q_k^\lambda}) < P_{q_k^\lambda}, \forall k, 1 \leq k \leq m \); or (2) \( \Pr(r_j^\psi \geq C_{r_j^\psi}) < P_{r_j^\psi} \); or (3) \( \Pr(bw^\ell_i \geq C_{bw^\ell_i}) < P_{bw^\ell_i} \). 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 3.2, 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 probing quota ratios among different service functions. Suppose the function graph \( \xi \) 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 \( \nu_i \) is the probing quota weight associated with the service function \( F_i \). We can decide

\(^8\)If the ServFlow has \( k \) branch paths, then we need \( k \) composition probes to examine the ServFlow.

\(^9\)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.
the value of $\nu_i$ 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 $\sigma_i$ different service instances. We can calculate $\nu_i$ 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 $\nu_i$ to more critical service functions. We can achieve more efficient consumption of the probing budget $\beta_0$ by partitioning $\beta_0$ 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 $\tau_i$, where $\varpi_i$ represents the weight assigned to the branch path $\tau_i$. Suppose the function graph $\xi$ includes $B$ branch paths $\tau_1, \ldots, \tau_B$, $B \geq 1$. Each branch path $\tau_i$ includes $L_i$ service functions $\{F_{i_1}, \ldots, F_{i_{L_i}}\}$ 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 $\alpha$ 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 3.3, 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 $\beta$ and there are $T$ next-hop service functions $F_1, \ldots, F_T$, $v_i$ proportionally divides $\beta$ 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 $\tau_{i_1}, \ldots, \tau_{i_{k_i}}$. Then, the probing budget allocated to $F_i$ is decided by $\beta_i = \left\lfloor \left( \sum_{j=1}^{k_i} \varpi_{i_j} / \sum_{i=1}^{T} \sum_{j=1}^{k_i} \varpi_{i_j} \right) \cdot \beta \right\rfloor$, 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 $F_1, \ldots, F_T$. We now prove that the above proportional probing budget partition scheme can guarantee that each branch path $\tau_i$ receives its share of probing budget $\varpi_i \cdot \beta_0$.

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

**Proof.** Proof by induction on $L$, the length of $\xi$, which is defined as the maximum number of service functions (except $F_{-1}$ and $F_{-2}$) on all the branch paths of $\xi$. When $L = 1$, the fan-out service function can only be $F_{-1}$, where the proportional probing budget partition scheme allocates $(\varpi_i / \sum_{i=1}^{B} \varpi_i) \cdot \beta_0$ to the branch path $\tau_i$. Because $\sum_{i=1}^{B} \varpi_i = 1$, $\tau_i$ is allocated with $\varpi_i \cdot \beta_0$ probing budget, $1 \leq i \leq B$. The theorem holds. Suppose when $L = k$, the theorem holds. When $L = k + 1$, let us consider the first fan-out service function $F_g$. Suppose $F_g$ has $T$ next-hop service functions
centering

Figure 10: SpiderNet node software architecture.

Let $F_{1}^{next}, \ldots, F_{T}^{next}$. We use $\zeta_i$ to denote the sub-graph of $\zeta$, which is rooted at $F_{i}^{next}$. Suppose $\zeta_i$ includes $k_i$ branch paths, which are denoted by $\tau_{i_1}, \ldots, \tau_{i_{k_i}}$. Since $F_g$ is first fan-out service function, the total probing budget allocated to $F_g$ is $\beta_0$. $F_g$ is the root of all the $B$ branch paths included in $\zeta$. Thus, we have $\sum_{i=1}^{T} \sum_{j=1}^{k_i} \rho_{i_j} = \sum_{i=1}^{B} \rho_i = 1$. According to the proportional probing budget partition scheme, the probing budget allocated to $F_{i}^{next}$ is $(\sum_{j=1}^{k_i} \rho_{i_j} / \sum_{i=1}^{B} \rho_i) \cdot \beta_0 = \sum_{j=1}^{k_i} \rho_{i_j} \cdot \beta_0$. Since the length of $\zeta_i$ is at most $k$, the induction hypothesis applies to $\zeta_i$. Each branch path $\tau_{i_j}, 1 \leq j \leq k_i$ included in $\zeta'$ is allocated with probing budget $w_{i_j}' \cdot \beta_0', w_{i_j}' = \sum_{j=1}^{k_i} \rho_{i_j} / \sum_{i=1}^{B} \rho_i$ and $\beta_0' = \sum_{j=1}^{k_i} \rho_{i_j} \cdot \beta_0$. Thus, the probing budget allocated to $\tau_{i_j}$ is $(\sum_{j=1}^{k_i} \rho_{i_j} / \sum_{i=1}^{B} \rho_i) \cdot (\sum_{j=1}^{k_i} \rho_{i_j} \cdot \beta_0') = \rho_{i_j} \cdot \beta_0$. Thus, the theorem holds.

4 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 [3]. We first describe the prototype implementation.

4.1 Prototype Implementation and Evaluation

We have implemented a prototype of the SpiderNet system. Each SpiderNet node software is a multi-threaded running system written in about 13K lines of java code. Figure 10 illustrates the software architecture for one SpiderNet node. 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 [41]; (2) service graph generator module performs initial QoS-aware service composition finding and runtime service maintenance; (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. We deploy one service component on each SpiderNet node, which is randomly selected from the above six multimedia service components. Our experiments use 102 Planetlab hosts that are distributed across U.S. and Europe. Thus, the average replication degree of each multimedia service is $102/6 = 17$.

We then 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 P2P video streaming with desired transformations and enriched content. We have deployed and evaluated the SpiderNet system with the video streaming application on the wide-area network testbed PlanetLab [3]. The end-application on each node periodically submits random service composition requests to the SpiderNet system.

First, we measure the service session setup time in the wide-area network, which includes (1) decentralized service discovery time, (2) initial service graph finding time using the bounded composition probing protocol, and (3) service session initialization time. Figure 11 illustrates the average service session setup time using more than 500 requests generated from 102 different PlanetLab hosts. The current prototype of the SpiderNet system can setup a 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 implementation implements and tunings.

Second, we compare the QoS provisioning performance of SpiderNet with the random and optimal algorithms. We consider service composition requiring three different functions. We ask different approaches to find the best qualified service composition that has minimum end-to-end service delay. Because each service function has on average 17
instances, the average number of probes required by the optimal algorithm is $17^3 = 4913$. As shown by Figure 12, the average service delay of the service graphs discovered by the SpiderNet reduces with a growing probing budget. When the probing budget is very low, SpiderNet degenerates into the random algorithm, so the overhead is low, but the service quality is not satisfactory. When larger probing budget is allowed, the service graph quality improves, and when the probing budget reaches a certain threshold, it asymptotically approaches the optimal performance. However, SpiderNet can achieve near optimal performance with much lower overhead (i.e., $200/4913 = 4\%$) than the unbounded flooding scheme performing exhaustive searching.

4.2 Simulation Results

We have implemented an event-driven multimedia service overlay simulator using C++. The simulator first uses the degree-based Internet topology generator Inet-3.0 [50] to generate a power-law graph with 3200 nodes to represent the Internet topology. To construct the multimedia service overlay, we randomly select certain number nodes as overlay nodes and connect them into certain overlay network. We have used two different overlay topologies in our experiments: (1) mesh topology, where each node has an equal node degree $\theta$, and (2) power-law graph topology, where node degrees follow a power-law distribution. We use $\theta = |V| \cdot 10\%$ in our experiments. Once the node degrees are chosen, the nodes are connected into a topologically-aware overlay network using the Short-Long algorithm presented in [40].

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) contract 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. The SpiderNet distributed service composition is then performed on top of the overlay data routing. Each overlay node provides two service components. Each service component performs a service function that is randomly selected from $\lceil |V|/5 \rceil$ service functions. Thus, the average service duplication ratio is $2 \cdot |V| / |V| / 5 = 10$, which conforms to our assumption that a service function can be mapped to a limited number of service instances. The function graph $\xi$ of the request is randomly selected from 200 pre-defined templates, which include two to five service functions with one or two branch paths. The statistical resource and QoS requirements are uniformly distributed. During each time unit, certain number of composite service requests are generated. Each service session lasts 5 to 15 time units. A QoS-aware service composition is said to be successful, if and only if the composed service graph (1) satisfies the function graph requirements, (2) satisfies the user’s resource

---

10 We choose Inet-3.0 instead of other Internet topology generator because recent research [45] has shown that the degree-based Internet topology generator can most accurately capture the current Internet structure.

11 Recent study [42] has shown that real-world peer-to-peer overlay networks, such as Gnutella [1], have the power-law graph topology.

12 If one ranks all nodes from the most connected to the least connected, then the $i^{th}$ most connected node has $\omega/\omega$ neighbors, where $\alpha$ and $\omega$ are constants. We set $\omega = 3 \cdot \alpha$ and $\omega = 1.5$ to make its average node degree equal to $\theta$ for the purpose of comparison.
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 \( \frac{\text{SuccessNumber}}{\text{RequestNumber}} \).

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 service graphs to find the best qualified service graph. 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.

First, we compare the performance of different algorithms. Figure 13 illustrates the composition success rate achieved by different algorithms under different workload conditions. We used two variations of our scheme, “probing-0.2” and “probing-0.1”, which uses 20% and 10% of the probes required by the optimal algorithm, respectively. Each round of simulation lasts 2000 time units. Each success rate is averaged over all the requests generated during 2000 time units simulation duration. We observe that SpiderNet can achieve near-optimal performance with much lower overhead, which is also much better than the random and static algorithms. We also conduct the same experiments on a 500 node multimedia service overlay with power-law graph topology. The results also show the similar trend.

Second, we evaluate how the performance and overhead of SpiderNet scale with the sizes of the multimedia service overlay. We show the results on the mesh overlay topology only. The results on the power-law graph topology demonstrate the similar trend. We use three different multimedia service overlays with 200, 500, and 1000 nodes, respectively. Figure 14 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.

Third, we evaluate the overhead of the SpiderNet system. Figure 15 illustrates the probing overhead comparison between the optimal algorithm and the SpiderNet algorithm on the three different multimedia service overlays. The probing overhead is measured by the total number of probing messages generated in the whole service overlay during each time unit. The results show that SpiderNet has much lower probing 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 the service overlay. Figure 16 illustrates the total system overhead comparison between SpiderNet and the conventional centralized approach. The total system overhead of the centralized approach is calculated by $|V|^2$, $|V| = 200, 500, 1000$, which represents the system overhead lower-bound required by the global states update assuming the states update can be finished in one time unit. The total system overhead of SpiderNet includes two parts: local states update and probing overhead. The local states update overhead is calculated as $|V|^2 \cdot 10\%$ since each node has $|V| \cdot 10\%$ neighbors. The results demonstrate that SpiderNet has much lower overhead than the centralized approach. The overhead reduction becomes larger as we increase the overlay size. Thus, SpiderNet has much better scaling property than the centralized approach.

5 Related Work

Composable multimedia service framework has been proposed by much previous work. For example, Amir et al. [6] proposed the active service framework and applied it on a media transcoding gateway service. However, the
active service framework did not address the QoS-aware service composition problem. In [9], the authors proposed a component-based abstraction, called Infopipes, to reify the communication between connected multimedia components. The problems addressed by SpiderNet is orthogonal to those addressed by the Infopipes framework and can be beneficial to each other. In [35], the authors proposed a framework to decompose a computation into sub-computations and assign them to multiple gateways. In contrast, SpiderNet supports the on-demand composition of existing atomic multimedia services into composite multimedia services.

**Service composition** has been addressed by several research projects. For example, the SAHARA project [37, 38] addressed the fault resilience and load balancing problems in wide-area service composition. The SPY-Net framework [51] addressed the problem of resource contention for finding multimedia service paths in small-scale media proxy networks. The CANS project [22] provided an on-line function composition framework, which can dynamically compose a list of service functions during runtime to adapt to heterogeneous end-system and network conditions. In [26] and [28], we proposed two centralized service composition solutions for smart rooms and enterprise service overlay networks, respectively. SpiderNet differs from the above work by providing a fully distributed solution to support flexible and expressive service composition with statistical QoS assurances. In [29], we have presented the overall framework of the SpiderNet system. This paper extends the above work by integrating statistical QoS provisioning and inter-component consistency check into the service composition system. We also describe several enhancements to the basic service composition design.

**QoS provisioning** in both IP-layer network and end-system has been extensively studied over the past decade. The IP-layer network packet delivery QoS architecture includes Intserv [10, 11], and Diffserv [21]. Various QoS routing algorithms [16] have been proposed. Service composition is different from conventional QoS routing in that the goal of QoS routing is to find qualified *data paths* while the goal of service composition is to find qualified service-level paths or graphs, which requires the service composition framework to address a range of new constraints, namely service constraints. The end-system QoS provisioning has been addressed by both operating system kernel solutions (e.g., [8]) and middleware solutions (e.g., [14, 47, 20, 49]). Because composed distributed multimedia services require both network and end-system resource and QoS provisioning, we argue that these existing QoS solutions must be extended and integrated in order to provide QoS-aware service composition. SpiderNet is also related to the reflective middleware work (e.g., [19]) where meta-data is used to adjust the middleware operations. However, SpiderNet is a decentralized middleware framework to specifically support scalable QoS-aware service composition.

**Overlay networks** have been proposed for providing services that are hard to be implemented or deployed in the IP-layer network. For example, End System Multicast [18] provided application-layer multicast using overlay networks. The resilient overlay network [7] project provided an overlay network solution for quickly detecting and recovering from Inter-domain network failures by routing through intermediate overlay nodes. SpiderNet differs from the above work by focusing on composed multimedia service provisioning rather than merely providing data
transport. In terms of application-specified resource and QoS provisioning, SpiderNet is related to the Darwin project [15], which provides integrated customizable resource management via virtual meshes. However, the Darwin project did not address the service composition problem. In terms of dynamic creation of composed services on top of SON, SpiderNet is also related to the Genesis Kernel system [34], which can create distinct virtual network architectures on-demand. Different from the Genesis Kernel system that focuses on the network service provisioning, SpiderNet focuses on the dynamic provisioning of distributed multimedia services.

6 Conclusion

In this paper, we have presented SpiderNet, a fully decentralized QoS-aware service composition framework for multimedia service overlays. SpiderNet allows advanced distributed multimedia services to be dynamically and cost-effectively created from atomic multimedia service components that can be distributed in wide-area networks. SpiderNet makes four major contributions. First, SpiderNet integrates statistical multi-constrained QoS provisioning and automatic load balancing into the distributed service composition operation. Thus, SpiderNet can achieve both statistical QoS assurances for the composed distributed multimedia services and efficient resource utilization in the multimedia service overlay. Second, SpiderNet provides a fully distributed QoS-aware service composition solution to address the scalability challenge. The key idea is a bounded composition probing scheme, which assists the automatic mapping from an function graph to an instantiated composed service by performing on-demand, selective states collection and intelligent parallel searching of multiple candidate compositions. Third, SpiderNet provides an expressive service composition interface by supporting complex composition topologies and exchangeable composition orders. We have implemented a prototype of SpiderNet and conducted extensive experiments using both large-scale simulations and wide-area network testbed. Our results show that SpiderNet can achieve near-optimal QoS-aware service composition with low overhead.

References


