On Composing Stream Applications in Peer-to-Peer Environments

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

Abstract—Stream processing has become increasingly important as many emerging applications call for continuous real-time processing over data streams, such as voice-over-IP telephony, security surveillance, and sensor data analysis. In this paper, we propose a composable stream processing system for cooperative peer-to-peer environments. The system can dynamically select and compose stream processing elements located on different peers into user desired applications. We investigate multiple alternative approaches to composing stream applications: 1) *global-state-based centralized* versus *local-state-based distributed* algorithms for initially composing stream applications at setup phase. The centralized algorithm performs periodical global state maintenance while the distributed algorithm performs on-demand state collection. 2) *Reactive* versus *proactive* failure recovery schemes for maintaining composed stream applications during runtime. The reactive failure recovery algorithm dynamically recomposes a new stream application upon failures while the proactive approach maintains a number of backup compositions for failure recovery. We conduct both theoretical analysis and experimental evaluations to study the properties of different approaches. Our study illustrates the performance and overhead trade-offs among different design alternatives, which can provide important guidance for selecting proper algorithms to compose stream applications in cooperative peer-to-peer environments.

Index Terms—Peer-to-peer, stream processing, service composition, resource management, quality-of-service.

1 Introduction

1.1 Background

With the popularity of peer-to-peer (P2P) file sharing systems [1], P2P systems have drawn much research attention. One salient advantage of P2P systems is that they can aggregate a tremendous amount of resources in a failure-resilient and cost-efficient fashion. Previous work has addressed the problems of scalable data lookup (e.g., [28], [31], [26]), incentive engineering (e.g., [22]), and anonymity preservation (e.g., [35]) for providing efficient P2P data sharing. Inspired by P2P file sharing systems, researchers have proposed other P2P applications such as P2P streaming systems (e.g., [33], [17], [20], [8]) and P2P storage systems (e.g., [4]). In this paper, we focus on studying the problem of providing stream processing applications in cooperative P2P environments [8]. A cooperative P2P system consists of responsible peer users who honestly share resources with each other for the common good of everyone, such as enterpise P2P systems. The P2P streaming system can effectively support many stream applications such as voice-over-IP (VoIP) telephony, sensor data analysis, and security surveillance. Fig. 1 shows a VoIP application where the speaker's audio stream is processed by a language translation service and a speech transcription service before reaching the receiver. In this paper, a service is a self-contained application unit providing a certain stream processing function.

- *X. Gu is with IBM T.J. Watson Research Center, Hawthorne, New York. E-mail: xiaohui@us.ibm.com.*
- *K. Nahrstedt is with Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801. E-mail: klara@cs.uiuc.edu.*

Manuscript received 8 July 2004; revised 17 Mar. 2005; accepted 20 July 2005; published online 26 July 2006.
Recommended for acceptance by R. Schlichting.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0168-0704.

Although streaming systems have been studied in conventional distributed environments, P2P streaming systems must meet new challenges: 1) *Heterogeneity*: P2P systems consist of heterogeneous end-hosts, which implies that stream applications should be adaptive to fill the gap between senders and receivers. 2) *Decentralization*: P2P systems are fully decentralized where services can have multiple instances dispersed on different peers. 3) *Churn*: P2P systems allow peers to arbitrarily leave or join the systems, which makes long-lived stream applications prone to failures. To address these challenges, we propose a *composable* streaming system that can dynamically compose stream applications from selected service instances based on the user's function, resource, and quality-of-service (QoS) requirements. Although composable service infrastructures have been proposed under different research context (e.g., [25], [10], [21], [23], [13], [7]), it is still an open problem to provide an efficient composable streaming system that can meet the new challenges of P2P environments.

1.2 Our Contributions

We identify two key problems in composable P2P streaming systems: 1) *initial service composition* for initially composing stream applications at service setup phases, and 2) *dynamic failure recovery* for maintaining stream applications during service runtime phases. We first formulate both problems into constrained optimal graph mapping problems and prove them to be NP-hard [11]. Then, we propose multiple polynomial approximation algorithms for both problems. We also conduct theoretical analysis and simulation experiments to show the tradeoffs among different design alternatives. Our study provides important insights for designing P2P composable stream processing systems. Specifically, we make the following contributions:

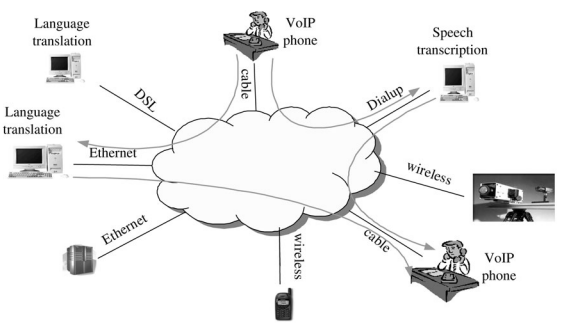

Fig. 1. Peer-to-peer composed stream applications.

- We present two polynomial approximation algorithms for the initial service composition problem: 1) *global-state-based centralized* (GC) algorithm, and 2) *local-state-based distributed* (LD) algorithm. In the former approach, each peer performs a centralized computation based on the global state information for the initial service composition. The GC algorithm uses a novel *adaptive composite metric* to achieve balanced load distribution subject to the user's multiconstrained QoS requirements. In contrast, the LD algorithm is a fully distributed and localized algorithm that employs a novel composition probing algorithm to find the initial service composition. We conduct theoretical analysis to compare the GC algorithm and the LD algorithm in terms of complexity and overhead. Our analysis reveals the key factors that decide the trade-offs between the GC algorithm and the LD algorithm.
- We present *reactive failure recovery* (RFR) and *proactive failure recovery* (PFR) schemes for maintaining the availability and QoS of composed stream applications during service runtime. The RFR algorithm dynamically recomposes a new stream application based on the global state information when the currently composed stream application fails. To achieve fast failure recovery, the RFR algorithm recomposes a new stream application that has the largest number of common service instances with the failed stream application subject to the condition that the new stream application satisfies the user's requirements. In contrast, the PFR algorithm maintains a number of backup service compositions in advance for each stream session. Thus, the PFR algorithm can quickly recover stream application failures if one of the backup service compositions can satisfy the user's requirements. We also conduct theoretical analysis to compare the RFR algorithm and the PFR algorithm in terms of complexity and overhead.
- We implement all proposed algorithms and evaluate their performance and overhead using extensive simulations. The simulator runs the GC, LD, RFR, and PFR algorithms on top of a simulated P2P overlay network. The simulator emulates dynamic resource allocations and shortest-path data routing at three different layers (i.e., IP network layer, overlay network layer, and composed stream application layer). Our results show that both GC and LD algorithms can achieve much better performance than other common heuristic algorithms, and that our algorithms are robust for different workloads and system conditions. We also conduct failure recovery experiments in dynamic P2P systems where a number of peers randomly fail. The results show that both RFR and PFR algorithms are much more efficient than the brute-force approach.

The rest of the paper is organized as follows: Section 2 introduces the system model. Section 3 presents the initial service composition algorithms. Section 4 presents the dynamic failure recovery schemes. Section 5 presents experimental results. Finally, we conclude the paper in Section 6.

2 System Model

2.1 Composite Stream Application Model

Composite stream applications extend conventional stream applications with interposed services that can provide various stream processing functions (e.g., transformation, aggregation, and correlation). To achieve QoS awareness and resource efficiency, interposed services are dynamically selected based on the user's requirements. The composable streaming system provides session-oriented application model that includes three different phases:

1. The *session setup phase* invokes the initial service composition algorithm to dynamically create a stream application satisfying the user's requirements. If the initial service composition is successful, the system allocates resources for the session and creates a session record for later references.
2. The *session runtime phase* performs streaming of application data units (ADUs) using source routing where each ADU carries the list of all service instances it needs to visit before it reaches the destination. During runtime, the composed stream application may experience failures due to peer departures or failures. The system can repair the broken composed stream application using dynamic failure recovery algorithms. If the failure recovery is successful, the streaming session is resumed from the last checkpoint.
3. The *session closing phase* tears down the streaming session when the application finishes its task. The resources used by the application session are released and the corresponding session record is deleted.

2.2 Layered System Architecture

We propose a three-layer system architecture to support the composed stream application model, illustrated by Fig. 2. The rationale behind this layered architecture is to achieve QoS-aware and resource-efficient service composition by decoupling the concepts of service functions, service instances, and physical hosts. Each service function can be mapped to different service instances located on distributed peer hosts, illustrated by Fig. 3. Instead of requiring the user to directly specify the service instance, the composable streaming

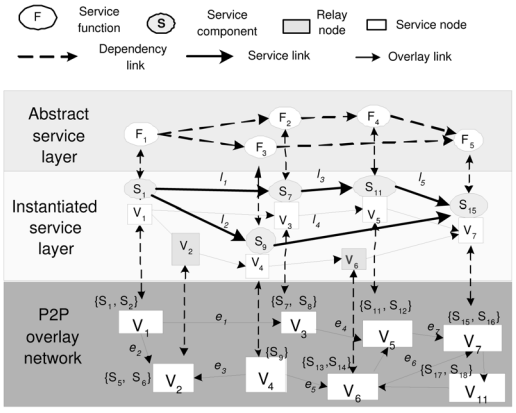

Fig. 2. P2P composable streaming system architecture.

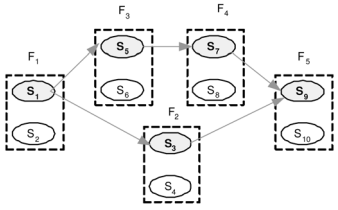

Fig. 3. Stream application composition model.

system can automatically select the best service instance for each required service function. During runtime, the mapping from the service function to the service instance can dynamically change to adapt to system changes (e.g., peer failures/departures).[1] Table 1 summarizes our notations. We now describe each layer as follows.

2.2.1 Abstract Service Layer

This layer represents the interface of the composable streaming system presented to the user, illustrated by the top layer in Fig. 2. The user is provided with various composite service templates described by function graphs (ξ_i). Each node in the function graph represents an atomic service function (F_i) such as the language translation service. The dependency link between two function nodes, denoted by $F_i \rightsquigarrow F_j$, represents that the output of F_i will be used as the input of F_j. The user request can also specify multiconstrained QoS requirements $Q^{req} = [q_1^{req}, \ldots, q_m^{req}]$, where q_i^{req} defines the user's requirement for the ith QoS metric (e.g., delay and loss rate). In this paper, we assume additive and deterministic QoS metrics. Multiplicative QoS metrics (e.g., loss rate) can be transformed into additive QoS metrics using logarithmic functions. Our solution can also be extended to support statistical QoS metrics [12]. For example, we can specify the VoIP application example illustrated by Fig. 1 as follows:

$$\begin{aligned}\xi_i = (&F_1 = AudioRecording, F_2 = LanguageTranslation,\\ &F_3 = SpeechTranscription, F_4 = AudioPlayback,\\ &F_1 \rightsquigarrow F_2, F_2 \rightsquigarrow F_4, F_1 \rightsquigarrow F_3, F_3 \rightsquigarrow F_4),\\ &Q^{req} = [q_1^{req}(delay) = 500ms, q_2^{req}(LossRate) = 5\%].\end{aligned}$$

1. Some service functions such as audio playback can only be instantiated on certain hosts (receiving host).

TABLE 1
Notations

notation	meaning
F	service function
s	service instance
$Q = [q_1, ..., q_m]$	QoS vector
ξ	function path
τ	service path
$\lambda \triangleq \lambda_s \cup \lambda_l$	service graph
e	overlay link
v	peer node
l	service link
$\wp$	overlay path
$RA = [ra_1, ..., ra_n]$	end-system resource availability
$R = [r_1, ..., r_n]$	end-system resource requirement
ba	bandwidth availability
b	bandwidth requirement

Different from conventional stream applications, composed stream applications have *nonuniform* resource requirements: 1) service instances, denoted by s_i, can have different end-system resource requirements (e.g., CPU and memory) due to function and implementation variations and 2) inter-service connections called service links, denoted by l_i, can have different bandwidth requirements since interposed services (e.g., speech-to-text) can change the data content. Thus, we associate an end-system resource requirement vector $R^{s_i} = [r_1^{s_i}, \ldots, r_n^{s_i}]$ with each service instance s_i, and a bandwidth requirement b^{l_i} with each service link l_i. The resource requirements are also related to the size of the ADU packet and the stream rate, which will be factored into the R^{s_i} and b^{l_i} during runtime.

2.2.2 Instantiated Service Layer

This layer consists of instantiated stream applications illustrated by the middle tier in Fig. 2. Each stream application is described by a service graph (λ) that includes a service instance set (λ_s) and a service link set (λ_l). Each service instance s_i is associated with a QoS vector $Q^{s_i} = [q_1^{s_i}, \ldots, q_m^{s_i}]$, where $q_i^{s_i}, 1 \leq i \leq m$ denotes the ith QoS value of s_i. The service link (l_i) represents the connection between service instances in the service graph. We use $Q^{l_i} = [q_1^{l_i}, \ldots, q_m^{l_i}]$ to denote the QoS values of the service link l_i. Each service link is instantiated into an overlay path ($\wp$) consisting of a list of overlay links e_i. For example, in Fig. 2, the service link l_2 between s_1 and s_9 is instantiated into the overlay path: $\wp \triangleq v_1 \rightarrow v_2 \rightarrow v_4$ since v_1 and v_4 are not directly connected. We call an overlay node a *service peer* if it provides any service instance for the stream application. We call an overlay node a *relay peer* if it only provides application-level data forwarding for the stream application. A service graph can be a directed acyclic graph (DAG) consisting of multiple service paths (τ_i). For example, in Fig. 2, the service graph includes two service paths. The QoS values of a service path, denoted by Q^{τ_i}, are defined as the accumulated QoS values of its constituent service instances and service links. The QoS values of a service graph, denoted by Q^{λ}, are defined as the worst QoS values of its service paths (i.e., the largest delay or worst loss rate).

2.2.3 P2P Overlay Network Layer

This layer consists of a collection of peer hosts v_i connected via application-level connections called overlay links (e_i), illustrated by the bottom tier in Fig. 2. Each peer provides a set of service instances denoted by $\{s_i, \ldots, s_j\}$. If two peers are not directly connected, the data stream between them has to flow through an overlay path $\wp$. For example, in Fig. 2, the data from v_1 to v_4 has to flow through the overlay path $\wp \stackrel{\triangle}{=} v_1 \rightarrow v_2 \rightarrow v_4$. The overlay network layer provides a resilient data routing infrastructure [5] for composed stream applications. The overlay topology [27] can be constructed in different ways in terms of topology type, node degree, and neighbor selections. We adopt the mesh topology that is commonly used by overlay streaming systems [9], [30], [20]. Each overlay link, e_i, is associated with a QoS vector $Q^{e_i} = [q_1^{e_i}, \ldots, q_m^{e_i}]$. Each peer v_i is associated with an end-system resource vector $RA^{v_i} = [ra_1^{v_i}, \ldots, ra_n^{v_i}]$, which describes the current available end-system resources (e.g., CPU and memory) on the peer host v_i. Each overlay link e_i is associated with a bandwidth availability metric ba^{e_i}. Each peer host maintains a local state including the QoS/resource states of its neighbor peers and adjacent overlay links in the overlay network. Currently, our system implements overlay data routing using the delay-based shortest path routing algorithm.

2.3 Problem Descriptions

We formulate the *initial service composition* (ISC) problem into a constrained optimization problem: mapping a function graph into the best service graph. First, the composed stream application should satisfy the user's function, QoS, and resource requirements. Second, the composed stream application should be instantiated on least-loaded nodes and overlay links for balanced load distribution. Formally, the ISC problem can be defined as follows.

Definition 1: Initial service composition (ISC) problem. *Given a P2P overlay network G = (V,E) where V denotes the set of $|V|$ nodes (v_i) and E denotes the set of $|E|$ overlay links (e_i). Let $s_i.F$ denote the function provided by the service instance s_i, s_i/v_j denote the service instance s_i provided by the peer host v_j, and $l_i/\wp_j$ denote the service link l_i instantiated on the overlay path $\wp_j$. We use $\omega_k, 0 \leq \omega_k \leq 1, 1 \leq k \leq n+1$ to denote the importance of the ith resource type, where $\sum_{k=1}^{n+1} \omega_k = 1$. Given a user request $\langle \xi, Q^{req}, R^{req} \rangle$, the ISC problem is to find the best service graph $\lambda \stackrel{\triangle}{=} \{\lambda_s, \lambda_l\}$ such that*

$$\textbf{min} \sum_{s_i/v_j \in \lambda_s} \sum_{k=1}^{n} \omega_k \cdot \frac{r_k^{s_i}}{ra_k^{v_j}} + \omega_{n+1} \cdot \sum_{l_i/\wp_j \in \lambda_l} \frac{b^{l_i}}{ba^{\wp_j}}, \quad (1)$$

$$\textbf{subject to } \forall F_k \in \xi, \exists s_i \in \lambda_s, s_i.F = F_k, \quad (2)$$

$$q_i^{\lambda} \leq q_i^{req},\ 1 \leq i \leq m, \quad (3)$$

$$\forall s_i/v_j \in \lambda_s, r_k^{s_i} \leq ra^{v_j}, 1 \leq k \leq n \wedge \forall l_i/\wp_j \in \lambda_l, b^{l_i} \leq ba^{\wp_j}. \quad (4)$$

Compared to conventional distributed systems, P2P systems present much more dynamics (e.g., arbitrary peer arrivals/departures and service instance deletions/additions). Thus, a composed stream application is prone to failures, which makes dynamic failure recovery necessary for providing failure-resilient composed streaming services. We also formulate the dynamic failure recovery (DFR) problem into a constrained optimization problem. First, the new stream application should satisfy the user's requirements and does not include any failed service instances or overlay links. Second, the new service graph should have the largest number of common service instances with the current service graph so as to minimize failure recovery disruption. Formally, the DFR problem can be defined as follows.

Definition 2: Dynamic failure recovery (DFR) problem. *Let λ^{old} and λ^{new} denote the broken service graph and the new service graph, respectively. Let $|\lambda_s^{old} \cap \lambda_s^{new}|$ denote the number of common service instances between the old service graph and the new service graph. Given the P2P system $G = (V, E)$ and the user request $\langle \xi, Q^{req}, R^{req} \rangle$, the DFR problem is to find a new service graph λ^{new} such that* **maximize** *$|\lambda_s^{old} \cap \lambda_s^{new}|$,* **subject to** *$\lambda^{new}$ satisfies (2), (3), and (4).*

We prove that finding the optimal solutions for both ISC and DFR problems are computationally intractable, which is indicated by the following theorem. Proofs for all theorems and corollaries in this paper can be found in the Appendix. Thus, our goal is to explore efficient approximation algorithms for both problems.

Theorem 1. *Both ISC and DFR problems are NP-hard.*

2.4 Assumptions

First, we assume cooperative managed P2P environments [8] such as enterprise P2P systems. Second, service instances are either stateless or contain only soft states that can be recovered by software. Third, we assume that service instances are described using high-level specification languages (e.g., [29], [24], [15]) based on a common ontology. Fourth, we assume that there exists a translator (e.g., [3]) that can map the application-level QoS specifications into low-level resource requirements (e.g., CPU, memory, and network bandwidth). Fifth, we assume that each peer can monitor available network bandwidth on its adjacent overlay links using measurement tools (e.g., [19], [18]). Finally, we assume that a service discovery system [14] is available to find candidate service instances matching a required service function.

3 Initial Service Composition Alternatives

This section presents and compares two different polynomial algorithms for the initial service composition problem: 1) global-state-based centralized algorithm and 2) local-state-based distributed algorithm.

3.1 Global-State-Based Centralized Algorithm

The global-state-based centralized (*GC*) algorithm requires each peer to maintain a global state information and executes a centralized algorithm to find the best service graph. The global state information consists of all peers' local states: the QoS vectors of all service instances/overlay links and the resource vectors of all peer hosts/overlay

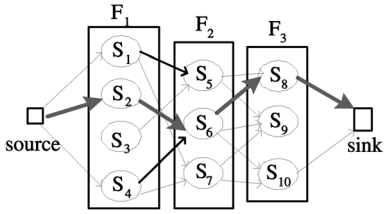

Fig. 4. Centralized service path composition.

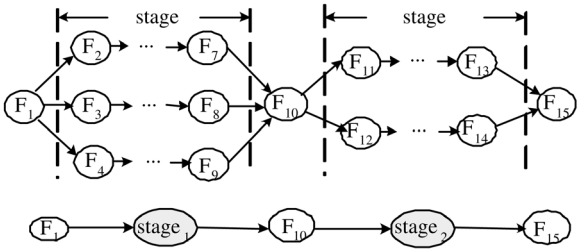

Fig. 5. Service graph composition using reduction.

links. The centralized algorithm requires global state information since a service composition request is allowed to use any service function. As we mentioned in Section 2, each peer will, by default, maintain local state information about its neighbors. The GC algorithm requires each peer to periodically disseminate its local state information to all other peers so that each peer can have update-to-date global state information.

We first describe the baseline GC algorithm for composing service paths illustrated by Fig. 4. The algorithm is adapted from the canonical Dijkstra shortest path algorithm. First, we construct a candidate graph consisting of all candidate service instances for all required service functions. The ith column in the candidate graph includes all candidate service instances for the ith required service function F_i in the function graph. The goal of this step is to satisfy the user's function requirements while reducing the service path searching range. Second, we perform a consistency check between every two dependent service instances. Two dependent service instances s_i and s_j are consistent if the output properties (e.g., media format, resolution, and stream rate) of s_i satisfy the input requirements of s_j. If two service instances are consistent, we add a link between them in the candidate graph. Third, we assign a cost value to each candidate graph link $\mathcal{L}_{ij} = s_i/v_i \rightarrow s_j/v_j$. The cost value is calculated based on an adaptive composite cost function, denoted by $\mathcal{F}(\mathcal{L}_{ij})$, that will be described in the next paragraph. Finally, we run a modified Dijkstra algorithm to find the shortest path from the source peer to the destination peer, which is returned as the best service path by the GC algorithm.

The cost function $\mathcal{F}(\mathcal{L}_{ij} = s_i/v_i \rightarrow s_j/v_j)$ is designed to meet the multiconstrained optimization goal of the initial service composition (Definition 1). We use $[q_1^{s_j}, \ldots, q_m^{s_j}]$ and $[q_1^{l_i}, \ldots, q_m^{l_i}]$ to denote the QoS values of the service instance s_j and the service link l_i from s_i to s_j. To aggregate different QoS metrics using normalization, we use $[q_1^{max}, \ldots, q_m^{max}]$ to denote the maximum values of the QoS metrics. To meet the load balancing goal (1), we consider the end-system resource "congestion" ratio $\frac{r_k^{s_j}}{ra_k^{v_j}}, 1 \leq k \leq n$, where $r_k^{s_j}$ denotes the requirement of s_j for r_k and $ra_k^{v_j}$ denotes the availability of r_k on the provisioning host v_j. We also consider the bandwidth "congestion" ratio $\frac{b^{l_i}}{ba^{\wp_i}}$, where b^{l_i} denotes the bandwidth requirement of l_i and $ba^{\wp_i}$ denotes the available bandwidth of the corresponding overlay path $\wp_i$. We set $\mathcal{F}(\mathcal{L}_{ij}) = \infty$, if $\exists k, 1 \leq k \leq n$, $r_k^{s_j} > ra_k^{v_j}$, or $b^{l_j} > ba^{\wp_j}$, which means that the resource requirements of s_i or l_j cannot be satisfied. Thus, we define $\mathcal{F}(\mathcal{L}_{ij})$ associated with a candidate graph link $\mathcal{L}_{ij} = s_i/v_i \rightarrow s_j/v_j$ as follows:

$$\mathcal{F}(\mathcal{L}_{ij}) = \sum_{k=1}^{m} w_k \left(\cdot \frac{q_k^{s_j}}{q_k^{max}} + \frac{q_k^{l_i}}{q_k^{max}} \right) + w_{m+1} \cdot \left(\sum_{z=1}^{n} \omega_z \cdot \frac{r_k^{s_j}}{ra_k^{v_j}} + \omega_{n+1} \cdot \frac{b^{l_j}}{ba^{\wp_j}} \right). \quad (5)$$

The weight $w_k, 1 \leq k \leq m+1$ represent the importance of different QoS or resource metrics during service path finding, where $\sum_{k=1}^{m+1} w_k = 1$. The weight $\omega_z, 1 \leq z \leq n+1$ represent the importance of different resource metrics during load balancing, where $\sum_{z=1}^{n+1} \omega_z = 1$. Higher weight value represents that the corresponding metric will be given higher priority in service instance selection. To satisfy multiple QoS constraints, we modify the Djikstra algorithm by adaptively adjusting the importance weights based on the "violation risk" of different QoS metrics. We want to minimize the maximum violation risk among all QoS metrics. To be specific, suppose s_j/v_j is the current chosen node by the Extract_Min operation in the Dijkstra algorithm, whose shortest path from the source s_0 is just determined, denoted by $s_0 \mapsto s_j$. We calculate the violation risk of the q_k metric using the ratio $P_{q_k} = \frac{q_k^{s_0 \mapsto s_j}}{q_k^{req}}$, where $q_k^{s_0 \mapsto s_j}$ is the value of q_k for the partial service path $s_0 \mapsto s_j$ and q_k^{req} is the required constraint. The higher the ratio P_{q_k} is, the higher violation risk the q_k metric has since its accumulated value is closer to its constraint. Based on the violation risk of different QoS metrics, we calculate the importance weights using $w_k = \frac{P_{q_k}}{\sum_{j=1}^{m} P_{q_j}} \cdot (1 - w_{m+1})$, which means that we give higher priority to the metric with higher violation risk. We do not claim that our GC algorithm is the best heuristic for the initial service composition problem. But, our design provides important insight that adaptation and aggregation are important for solving the problem. Later, in Section 5, we will show that the GC algorithm can achieve better performance than other common heuristics.

The above baseline GC algorithm solves the *service path finding* problem. We now extend the above algorithm to solve the problem of *service graph finding* using standard divide-and-conquer strategy, illustrated by Fig. 5. We first divide a function graph into several stages using "branch" nodes that have multiple inputs or outputs, e.g., F_1, F_{10}, and F_{15} in Fig. 5. Each stage consists of *disjoint* function paths. We can use the baseline GC algorithm to instantiate all disjoint function paths into service paths. The QoS values of a stage are defined

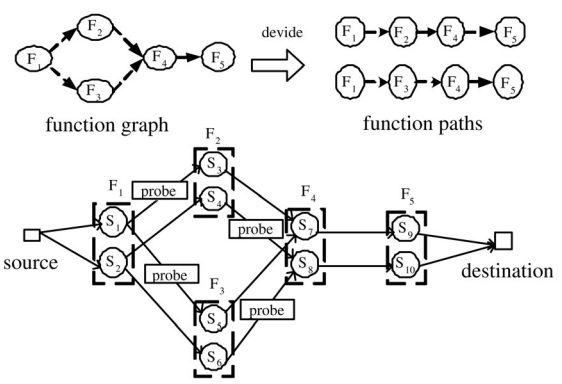

Fig. 6. Distributed composition using on-demand probing.

as the bottleneck values (e.g., longest delay) among the stage's constituent service paths. Then, we can use a virtual node to represent each stage to reduce the problem into the case of finding a service path. For example, in Fig. 5, we first instantiate the two stages and then replace them with two virtual nodes. The QoS values of a service graph are defined as the aggregation of its constituent stages. For more complex function graphs, we may need to recursively apply the above strategy since stages may include substages.

3.2 Local-State-Based Distributed Algorithm

The *local-state-based distributed* (LD) algorithm, illustrated by Fig. 6, executes a composition probing protocol that employs a set of probing messages, called probes, to perform distributed service graph finding. Different from the GC algorithm, the LD algorithm does not require global state information and each peer only maintains local state information about its neighbors. The major task of the composition probing is to *collect* distributed state information on-demand according to the user's request. Different from the network probing that performs on-demand *state measurement*, the composition probing performs on-demand *state collection*. The best service graph is then derived based on the state information collected by the composition probing. The LD algorithm can explicitly control the number of probes for a user request, which is called *probing budget* (β). The probing budget is a configurable system parameter controlling the tradeoff between performance (i.e., composition success rate) and overhead. We can adaptively set the probing budget as the minimum number of needed probes to achieve a performance target under current system conditions [16].

The LD algorithm includes four major phases. First, the source peer decomposes the function graph into Y function paths. For example, in Fig. 6, we decompose the function graph into two function paths. The source then generates a probe for each function path. Each probe is initialized with its designated function path, user's QoS/resource requirements, and the probing budget $\lfloor \beta/Y \rfloor$. Then, the algorithm enters the second phase: hop-by-hop probe processing. At each hop, the probe collects local state information from the visited peer host. The probe can spawn new probes to visit multiple next-hop service instances under the probing budget constraint. All spawned probes will arrive at the destination peer after traversing different candidate service paths.

In the third phase, the destination peer merges probed service paths into complete candidate service graphs. We briefly describe the merging algorithm as follows: 1) Classify all service paths into Y sets according to their provisioned service functions. All service paths in one set provide the same set of service functions. 2) Merge every Y combinable service paths, one from each of the Y sets, into a complete service graph. Two service paths are mergable if their common service functions are performed by the same service instance. Next, we select qualified service graphs based on the user's QoS and resource requirements (3), (4). One nice property of the LD algorithm is that it can find multiple qualified service graphs using one round of composition probing. These redundant qualified service graphs can be conveniently used as backup service graphs, which will be described in the next section. The best service graph is then selected among the qualified service graphs according to the load balancing goal (1). Finally, the destination peer sends a confirmation message to the source peer containing the selected service graph.

The per-hop probe processing at each service peer includes four major steps. First, the peer checks whether the service path traversed by the probe already violates the user's QoS and resource requirements. If the service path cannot meet the requirements, the peer drops the probe to prune unqualified searching branches as early as possible. Otherwise, the peer performs *soft* resource allocation to temporarily reserve required resources on the local node and the overlay link to the next-hop peer. Thus, we can ensure that required resources are still available at the end of the probing process. By soft, we mean that resource allocation will be cancelled after a certain timeout period if the peer does not receive a confirmation message. Since two service links l_i and l_j can share the common overlay link e_k, soft resource allocation can address the problem of resource overbooking on the same overlay link by two different service links. We can apply the same approach to the problem of shared physical link between two overlay links if the underlying physical network topology is available.

Third, the peer selects candidate service instances to probe for the next-hop service function. Let β_k denote the probing budget for the next-hop function F_k. Let Z_k denote the number of all candidate service instances for F_k. If $\beta_k \geq Z_k$, the peer has enough probing budget to probe all candidate service instances. The peer spawns Z_k probes from the received probe and each new probe has the probing budget $\lfloor \frac{\beta_k}{Z_k} \rfloor$. However, if $\beta_k < Z_k$, the peer needs to select a subset of service instances to probe. If the information about those candidate service instances is not available in the peer's local state, the peer performs random selection. The peer can also perform greedy selection based on available external information such as geographical locations and network delay provided by the overlay data routing layer. Fourth, each new probe inherits the state information of the old probe and collects local state information from the current peer. Finally, the current peer delivers all new probes to the overlay data routing layer that will route the probes to their designated next-hop service instances. During the routing process, probes will also collect QoS and available bandwidth information about the overlay paths from the current service peer to the next-hop service peers. Different from service peer, the relay peer only performs overlay data routing for the stream application. Thus, when a relay peer receives a probe, it

does not spawn new probes but only updates the probe with its local resource and QoS states.

3.3 Analytical Comparison

3.3.1 Computational Complexity

Both GC and LD algorithms are polynomial algorithms whose computational complexities are stated by the following theorems.

Theorem 2. *Let K, L, and m denote the number of candidate service instances, the number of candidate service links, and the dimension of QoS constraints. The computational complexity of the GC algorithm is $O(KlogK + Km + LlogK)$.*

Theorem 3. *Let K denote the number of candidate service instances, L denote the number of candidate service links, $|V|$ denote the number of all peers, β denote the probing budget, Y denote the number of function paths decomposed from the function graph, and m denote the dimension of QoS constraints. The computational complexity of the LD algorithm is $O(log|V| + (L + (\frac{\beta}{Y})^Y(Y(log\beta - logY) + m))/K)$.*

Compared to the GC algorithm that performs centralized computation at the source peer, the LD algorithm performs distributed computation at different peers. Thus, the LD algorithm imposes less computation load on each peer. However, the LD algorithm may require longer execution time since probes have to travel across wide-area networks to collect state information. We have implemented a prototype of the LD algorithm and measured its execution time on the PlanetLab testbed [2]. The LD algorithm takes about a few seconds to finish the initial service composition in a P2P system using 102 Planetlab hosts distributed across the US and Europe [14], which is acceptable for long-lived stream applications that often last tens of minutes or hours.

3.3.2 Space Complexity

Let $|V|$ denote the number of all peers. The space complexity of the GC algorithm is $O(|V|^2)$ for recording the global state information. Thus, in a large-scale P2P system (i.e., $|V|$ is large), the GC algorithm has large memory requirement for all peers. In contrast, the space complexity of the LD algorithm is a constant with regard to $|V|$ because each peer only maintains local state information.

3.3.3 Algorithm Overhead

The overhead of the GC algorithm mainly comes from the global state maintenance. To keep up-to-date global state information at each peer, we need to perform periodical dissemination of each peer's states to all other peers. Each peer first constructs a message containing the local state information measured by itself and then sends this message to all other peers. We define the overhead of the GC algorithm as the number of state update messages generated per time unit in the P2P system.

Theorem 4. *Let T denote the state update period and $|V|$ denote the number of peers. Assuming each peer needs at least one message to update its local state information with any other peer host, the overhead lower-bound of the GC algorithm is $\lfloor\frac{|V|(|V|-1)}{T}\rfloor$.*

The overhead of the LD algorithm includes both probing overhead and local state maintenance overhead, which are defined as the total number of probing messages and local state update messages generated per time unit in the whole P2P system.

Theorem 5. *Let Z denote the average number of user requests per time unit, β denote the probing budget, T denote the state update period, and σ denote the average node degree in the P2P overlay network. In a P2P system $G = (V, E)$, the overhead upper-bound of the LD algorithm is $Z \cdot \beta + \lfloor\frac{|V|\cdot\sigma}{T}\rfloor$.*

Based on Theorem 3 and Theorem 4, the GC algorithm has larger overhead than the LD algorithm if

$$\left\lfloor\frac{|V|(|V|-1)}{T}\right\rfloor > Z \cdot \beta + \left\lfloor\frac{|V|\cdot\sigma}{T}\right\rfloor. \quad (6)$$

In a large P2P system where $|V|^2 \gg |V|$, we can simplify the above inequality as follows:

$$Z < \frac{|V|(|V|-\sigma)}{\beta \cdot T}. \quad (7)$$

The above inequality indicates that the overhead comparison between the GC algorithm and the LD algorithm depends on a number of factors including:

1. the scale of the P2P system $|V|$,
2. the user request rate Z,
3. the probing budget β,
4. the state update period T, and
5. the overlay network's node degree σ.

P2P systems can include tens of, hundreds of, or thousands of nodes. The user request rate can also vary from time to time. The probing budget is adaptively configured by the system based on the system's performance target and resource conditions. The state update period depends on the dynamics of the P2P system. In a highly dynamic P2P system, the state update period should be short in order to keep track of frequent peer changes. The value of σ depends on the overlay network's topology. Our analysis provides important guidance for selecting proper algorithms under different system conditions. For example, if the peer number $|V|$ is 1,000, average node degree σ is 100, the probing budget β is 200, the state update period T is two time units, and the average request rate Z is 2,000 requests per time unit, the LD algorithm has lower overhead than the GC algorithm.

4 Dynamic Failure Recovery Alternatives

In this section, we present and compare two different schemes for the *dynamic failure recovery* (DFR) problem (Definition 2): 1) *reactive failure recovery* and 2) *proactive failure recovery*. The failures of composed stream applications can be caused by dynamic peer departures/failures or software failures of service instances.

4.1 Reactive Failure Recovery

The main idea of the reactive failure recovery (RFR) algorithm is to dynamically recompose a new service graph when the current service graph fails, illustrated by Fig. 7.

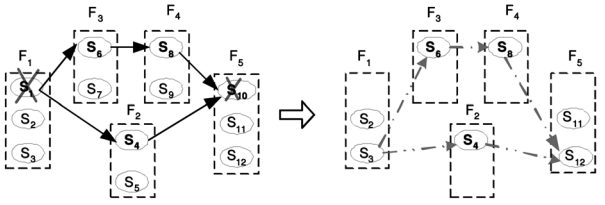

Fig. 7. Reactive failure recovery using recomposition.

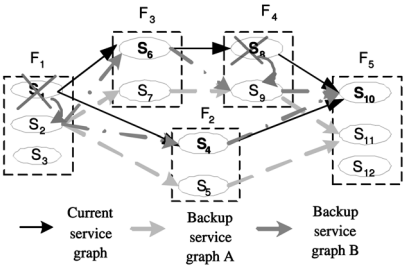

Fig. 8. Proactive failure recovery using backup compositions.

The new service graph should continue to satisfy the user's function, QoS and resource requirements. To achieve fast failure recovery, the new service graph should incur minimum service instance changes for reducing the user perceived disruptions. Similar to the GC algorithm, the RFR algorithm requires global state information and performs centralized computation to find a new service graph. First, we generate an updated candidate graph from the function graph including the update-to-date QoS/resource states and the failure information about all candidate service instances and service links. In P2P systems, the service instance failure can be caused by peer departures, peer host failures, or service instance deletions. We modify the candidate graph by removing those failed service instances. The service link failure will be handled by the overlay data routing layer (e.g., finding a new overlay path). For example, in Fig. 7, two service instances s_1 and s_{10} failed on the current service graph. Thus, we remove s_1 and s_{10} from the candidate service instance list.

To minimize service instance changes, we further modify the candidate graph by removing the other candidate service instances in the column where the old service instance is good. For example, in Fig. 7, the service instance s_6 is performing well for providing the service function F_3 on the current service graph. We modify the candidate graph by removing the other candidate service instance s_7 that also provides F_{10}. Thus, the new service graph is forced to reuse the same service instance s_6 in the current service graph. In Fig. 7, we applied the above modification on the other two good service instances s_4 providing F_2 and s_8 providing F_4. We then use the same adaptive Dijkstra algorithm as the GC scheme to find a new service graph in the modified candidate graph. If a new qualified service graph can be found, the interrupted stream session can be resumed by switching from the broken service graph to the new service graph. For example, in Fig. 7, the RFR algorithm found a new service graph shown in dash-dotted lines in the modified candidate graph to recover the failures of the current service graph shown in solid lines.

4.2 Proactive Failure Recovery

The main idea of the proactive failure recovery (PFR) algorithm is to maintain a number of backup service graphs for each active stream session, illustrated by Fig. 8. When the current service graph fails, the stream session can be resumed if one of the backup service graphs can recover the failure. For example, in Fig. 8, the source peer maintains two backup service graphs that are partially disjoint with the current service graph. When the current service graph (shown in solid lines) failed, we can recover the failure by switching to one of the backup service graphs (shown in dashed lines). In contrast to the RFR scheme, the PFR algorithm does not recompose a new service graph on-the-fly. Instead, the PFR algorithm needs to maintain the quality of backup service graphs using periodical measurements, which is denoted as "maintenance overhead." When the source peer detects that one of the backup service graphs fails or becomes unqualified, the PFR algorithm will select another qualified service graph as the backup service graph.

To achieve efficient proactive failure recovery, we need to answer two key questions: 1) How many backup service graphs should be maintained for a stream session? 2) Which qualified service graphs should be selected as backup service graphs? The number of maintained backup service graphs represents the trade-off between failure resilience and maintenance overhead. The more backup service graphs we maintain, the better failure resilience we can achieve. However, the maintenance overhead also becomes larger. Thus, the PFR algorithm adaptively decides the number of backup service graphs based on the condition of the current service graph and the user's requirements. In particular, the failure probability of the composed stream application is decided by the provisioning peers' availability [6]. Intuitively, if the QoS values and failure probability of the current service graph are much better than the user's requirements, we can maintain a small number of backup service graphs since the failure probability of the current service graph is small. Otherwise, we need to maintain more backup service graphs to achieve user desired QoS assurances and failure resilience. Let $Q^\lambda = [q_1^\lambda, \ldots, q_m^\lambda]$ and f^λ denote the QoS values and failure probability of the current service graph λ. Let $Q^{req} = [q_1^{req}, \ldots, q_m^{req}]$ and f^{req} denote the user required QoS and failure probability. Let Γ denote the system's upper-bound for the backup service graph number, and Θ denote the number of redundant qualified service graphs found by the initial service composition. The PFR algorithm calculates the number of backup service graph γ as follows:

$$\gamma = \mathbf{min}\left(\left\lfloor \Gamma \cdot \left(\sum_{i=1}^{m} \frac{q_i^\lambda}{q_i^{req}} + \frac{f^\lambda}{f^{req}}\right)\right\rfloor, \Theta\right). \quad (8)$$

Given the number of backup service graphs, the PFR algorithm needs to decide which qualified service graphs should be selected as backup service graphs. On one hand, failure resilience implies that backup service graphs should be disjoint with the current service graph. On the other hand, fast failure recovery implies that the backup service graph should be overlapped with the current service graph.

To accommodate both requirements, the PFR algorithm selects the backup service graphs as follows: For each service instance s_i on the current service graph, PFR selects a qualified service graph that does not include s_i but has the largest number of common service instances with the current service graph. Hence, if s_i fails, PFR can quickly recover the failure by switching to the above backup service graph with minimum service instance changes. In order to handle concurrent service instance failures, the PFR algorithm continues to select backup service graphs that do not include every two service instances, every three service instances, and so forth. However, under the constraint of backup service graph number γ, PFR may not be able to maintain all desired backup service graphs. Thus, PFR first maintains backup service graphs for most unreliable service instances.

4.3 Analytical Comparison

4.3.1 Computational Complexity

The RFR algorithm recomposes a new service graph based on the modified candidate service graph. Thus, the computational complexity of the RFR algorithm is the same as the GC algorithm (Theorem 2). In contrast, the PFR algorithm does not recompose a new service graph upon failure, but replaces the broken service graph with a backup service graph. The PFR algorithm selects a number of backup service graphs from the qualified service graphs found by the initial service composition algorithm. Let Γ denote the upper-bound of backup service graph number for each session. It takes the PFR algorithm at most $O(\Gamma)$ time to find a proper backup service graph to replace the broken service graph.

4.3.2 Space Complexity

The space complexity of the RFR algorithm is the same as the GC algorithm since both of them require global state information. If we use the RFR algorithm together with the GC algorithm, we do not need extra memory since the global state is already available. In contrast, the space complexity of the PFR algorithm is the requirement for storing backup service graphs. Let M denote the number of active sessions maintained by the peer and Γ denote the upper-bound on the number of backup service graphs for each session. The space complexity of the proactive failure recovery is $O(M \cdot \Gamma)$.

4.3.3 Algorithm Overhead

The overhead of the RFR algorithm comes from the requirement of the global state. If we use the RFR algorithm together with the GC algorithm, we pay no extra overhead for failure recovery. In contrast, the PFR algorithm does not require global state information. However, the PFR algorithm has to pay the backup service graph maintenance overhead. We define the overhead of the PFR algorithm as the number of maintenance messages generated per time unit. We have the following theorem about the PFR algorithm overhead.

Theorem 6. *Let Z denote the average user request rate, D denote the average application session duration, Γ denote the upper-bound of backup service graph number, and T denote the backup maintenance period. In a P2P system $G = (V, E)$, the overhead upper-bound of the proactive failure recovery algorithm is $\lfloor \frac{Z \cdot D \cdot \Gamma}{T} \rfloor$.*

TABLE 2
Simulation Parameters

# of physical hosts	3200
IP link delay	U[10,50]ms
IP link loss rate	[1,5%]
IP link bandwidth	[2,100]Mbps
Host CPU capacity	[100,800]
Host memory capacity	[200,1000] MB
# of overlay nodes ($\|V\|$)	[200, 1000]
average # of node degree (σ)	$0.1 \cdot \|V\|$
# of service functions	100
replication number	[5,25]
stream rate	[5, 50] ADU/second
ADU size	[160, 2000] Byte
CPU requirement (per ADU)	[5, 50]
memory requirement (per ADU)	[200, 4000] Byte
service time (per ADU)	[1,5] ms
state update period (T)	1 minute
# of function paths per ξ	[1,2]
function path length	[2,6]
request rate (Z)	[10, 100] requests/minute
session duration (D)	[5, 20] minutes
delay constraint.	[1000, 3000] ms
loss rate constraint	[3%, 10%]

5 Experimental Evaluation

5.1 Evaluation Methodology

We implement all proposed algorithms and conduct extensive experiments using a P2P streaming simulation testbed. The simulator implements the three-layer system architecture illustrated by Fig. 2a. We first use the degree-based Internet topology generator Inet-3.0 [34] to generate a 3,200 node power-law graph as the IP-layer network topology. We then randomly select $|V|$ nodes as peer hosts and connect them into an overlay network. We use the mesh topology where each peer is connected with $0.1 \cdot |V|$ neighbors via overlay links. The simulator emulates delay-based shortest path data routing and dynamic resource allocations at both IP network layer and overlay network layer. Table 2 summarizes all parameters used by the simulator.

Similar to previous work (e.g., [9]), we randomly assign QoS values (e.g., delay and loss rate) to each IP network link since Inet-3.0 does not provide QoS values for each network link. To simulate dynamics in the real world, the simulator first sets mean QoS values for each IP link. The instant QoS values then follow uniform or Gaussian distribution with certain deviation range. The mean delays of IP links are uniformly distributed within the interval [10, 50] ms. The mean loss rates of IP links are randomly set in the range of $[1\%, 5\%]$. The initial bandwidth capacity of each IP link is set in the range of [2, 100] Mbps. The CPU capacity of each physical host is uniformly distributed in the range of [100, 800] units. Different values reflect the heterogeneity of P2P systems. In the GC algorithm, the system performs global state update within the whole network every minute.

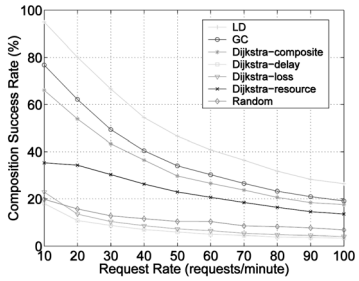

Fig. 9. Initial composition performance with different request rates.

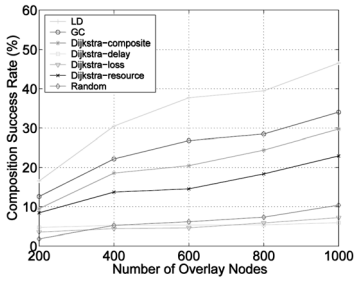

Fig. 10. Initial composition performance with different overlay networks.

The QoS and resource values of each overlay link can be derived from the values of its underlying IP-layer network path. The number of service instances for each service function is uniformly distributed in the range of [5, 25]. The source stream rate is randomly set in the range of [5, 50] ADUs per second. The size of one ADU is uniformly distributed in the range of [160, 2,000] Byte. The CPU and memory requirements for a service instance to process one ADU are uniformly distributed in the range of [5, 50] units and [200, 4,000] Byte, respectively. The processing time for a service instance to process one ADU is randomly distributed in the range of [1, 5] ms. For example, in the VoIP application, the source audio stream rate is 64Kbps (i.e., stream rate = 50 ADUs per second, the size of each ADU = 176 Byte). The amortized processing time and memory requirement for processing one ADU in the VoIP application is 2 ms and 400 Byte, respectively.

A number of user requests are randomly generated from different peers during each minute, which is defined by the *request rate*. The user request randomly chooses one of 30 predefined function graphs. Each function graph consists of one or two function paths. Each function path consists of [2, 6] function nodes. The total delay constraint and loss rate constraint for the composed stream application are uniformly distributed in the range of [1,000, 3,000] ms and $[3\%, 10\%]$, respectively. Each stream session lasts [5, 20] minutes.

For comparison, we also implement several alternative heuristic algorithms for the ISC problem:

1. *Dijkstra-composite* algorithm that uses standard Dijkstra algorithm with the composite metric (5) with fixed importance weights $w_k = \frac{1}{m+1}, 1 \leq k \leq m+1$,
2. *Dijkstra-delay* algorithm that uses standard Dijkstra algorithm with delay metric only,
3. *Dijkstra-loss* algorithm that uses standard Dijkstra algorithm with loss rate metric only,
4. *Dijkstra-resource* algorithm that uses standard Dijkstra algorithm with resource metrics only,
5. *Brute-force probing* algorithm that exhaustively probes all candidate service graphs, and
6. *Random* algorithm that randomly selects service instances in service composition.

To evaluate the performance of the proposed failure recovery algorithms, we also implement a *random recovery* (RR) algorithm for the DFR problem that randomly recomposes a new service graph to recover failures.

We define the following comparison metrics:

1. *composition success rate*, which is defined by $\frac{SuccessNum}{ReqNum}$ where $SuccessNum$ denotes the number of successful service compositions and $ReqNum$ denotes the number of user requests;
2. *composition overhead*, which is defined as the number of algorithm overhead messages (e.g., state update messages, probing messages, and backup maintenance messages) generated per minute; and
3. *composition failure rate*, which is defined by $\frac{FailureNum}{SessionNum}$, where $FailureNum$ denotes the number of failed stream sessions and $SessionNum$ denotes the number of established stream sessions.

5.2 Results and Analysis

We first evaluate the GC and LD algorithms for the initial service composition problem. We use a 1,000-node P2P overlay network. Fig. 9 shows the performance comparison results under different request rates. Each success rate value is averaged over all requests generated during a 1,000 minute simulation. We empirically set the probing budget as $\beta = min(0.3 \cdot \beta^{max}, 500)$, where β^{max} denotes the number of probes required by the *brute-force probing* algorithm. Each service function has 10 candidate service instances. We observe that both GC and LD algorithms consistently achieve better performance than other heuristic algorithms. Compared to the random algorithm, the GC algorithm can achieve three times better composition performance, and the LD algorithm can be four times better. The GC algorithm also performs better than the Dijkstra-composite algorithm by as much as 20 percent, which indicates the adaptation effectiveness of the GC algorithm. Compared to the single-metric-based Dijkstra algorithms Dijkstra-delay and Dijkstra-loss, the GC algorithm can have as much as 250 percent improvement, which shows the effectiveness of the composite metric. The Dijkstra-delay and Dijkstra-loss algorithms can even perform worse than the random algorithm under heavy workload since they do not consider load conditions of different peers.

Fig. 12 shows the overhead of different algorithms in the above experiments. The overhead of the GC algorithm is not affected by the request rate. The results show that the GC algorithm can have one magnitude larger overhead than the LD algorithm in the 1,000-node P2P system due to its global state requirement. The overhead of the GC algorithm is solely decided by the size of the overlay network (i.e., $|V| = 1,000$). In contrast, the overhead of the brute-force probing algorithm increases linearly with the request rate. However, the LD algorithm has much slower overhead increase since it performs bounded and pruned probing.

We conduct the second set of experiments to evaluate the scalability of the GC and LD algorithms. Fig. 10 shows the performance of different algorithms on different sizes of P2P overlay networks under the same workload (i.e., 50 requests

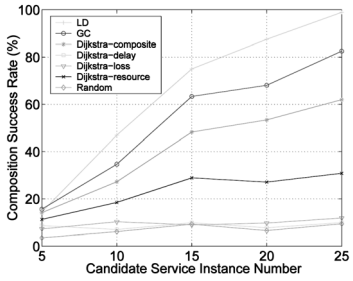

Fig. 11. Initial composition performance with different instance numbers.

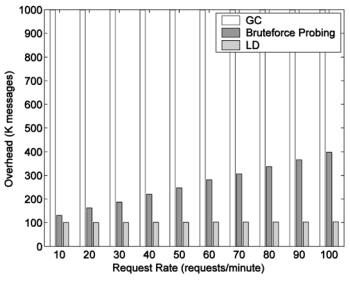

Fig. 12. Composition overhead with different request rates.

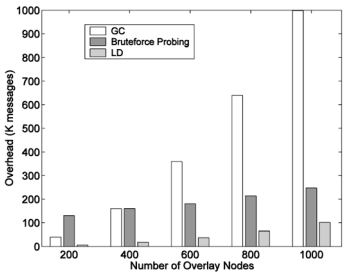

Fig. 13. Composition overhead with different overlay networks.

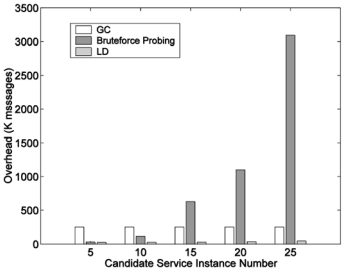

Fig. 14. Composition overhead with different instance numbers.

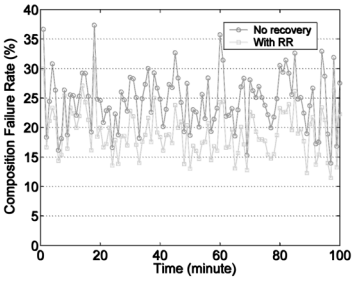

Fig. 15. Random failure recovery with a failure rate of 10 percent peers per minute.

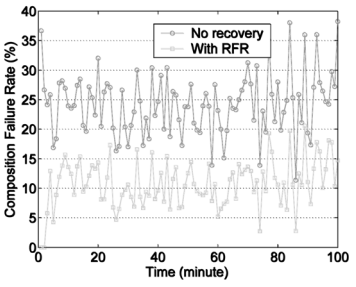

Fig. 16. Reactive failure recovery with a failure rate of 10 percent peers per minute.

per minute). The overall processing capacity of the P2P system increases as more nodes are added into the system. We observe that both LD and GC algorithms perform consistently better than other alternative schemes. Moreover, the LD and GC algorithms show faster performance increases than other heuristic algorithms. Fig. 13 shows the overhead of the GC, LD, and brute-force probing algorithms. The results show that both GC and brute-force probing algorithms require much larger overhead than the LD algorithm. Particularly, the overhead of the GC algorithm quickly increases as more nodes join the P2P system.

We conduct the third set of experiments to evaluate the effect of candidate service instance number on the performance of different initial service composition algorithms. We use a 500-node P2P system and gradually increase the number of candidate service instances for each service function from 5 to 25. The request rate is set as 30 requests per minute. Fig. 11 shows the composition performance of different algorithms using different number of candidate service instances. We observe that the GC and LD algorithms consistently perform better than other alternative algorithms. Fig. 14 shows the overhead of the GC, LD, and brute-force probing algorithms. The overhead of the GC algorithm is not affected by the number of candidate service instances. In contrast, the overhead of the brute-force probing algorithm exponentially increases with the number of candidate service instances since the number of candidate service graphs exponentially increases as the number of candidate service instances. However, the overhead of the LD algorithm has limited overhead increase because of the probing budget constraint. This set of experiments indicate that the proposed algorithms are robust for different replications of service functions.

We now evaluate the performance of different failure recovery algorithms. The RFR algorithm is combined with the GC algorithm since the RFR algorithm requires the same global state as the GC algorithm. The PFR algorithm is combined with the LD algorithm. We simulate a dynamic 500-node P2P system where a portion of peers randomly fail during each minute. We evaluate the performance of different failure recovery algorithms by comparing the failure rate values with and without dynamic failure recovery. A failure recovery is said to be successful if we can replace the broken service graph with a new qualified service graph. The maximum backup service graph number (Γ) in the PFR algorithm is set as 5. Fig. 15, Fig. 16, and Fig. 17 show the failure rate results of different failure recovery algorithms in a dynamic P2P system with 10 percent peers randomly fail each minute. We observe that the random recovery (RR) algorithm can only reduce the failure rate by at most 25 percent, the RFR algorithm

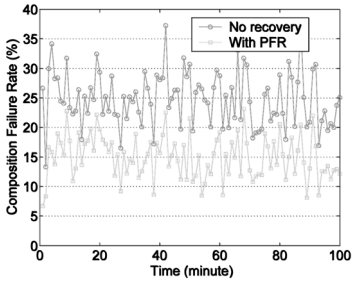

Fig. 17. Proactive failure recovery with a failure rate of 10 percent peers per minute.

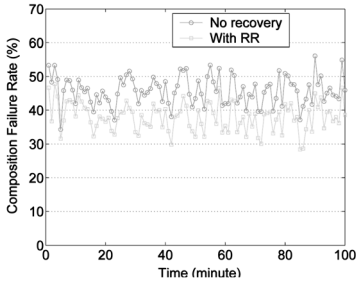

Fig. 18. Random failure recovery with a failure rate of 20 percent peers per minute.

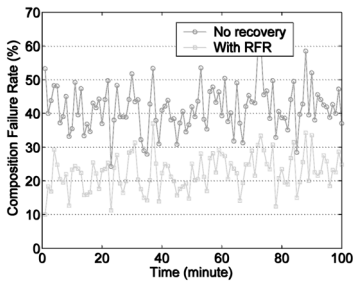

Fig. 19. Reactive failure recovery with a failure rate of 20 percent peers per minute.

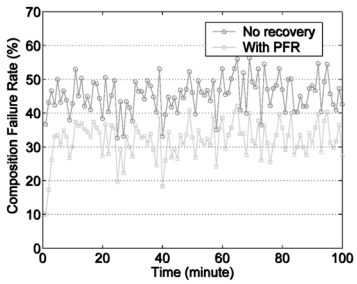

Fig. 20. Proactive failure recovery with a failure rate of 20 percent peers per minute.

can reduce the failure rate by as much as 70 percent, and the PFR algorithm can reduce the failure rate by as much as 65 percent by maintaining on average 4.5 backup service graphs. Fig. 18, Fig. 19, and Fig. 20 show the failure rate results of different failure recovery algorithms in a more dynamic P2P system with 20 percent peers randomly fail each minute. We observe that all algorithms experience higher failure rates since more peers experience failures. However, the RFR algorithm achieves on average 50 percent lower failure rate than the RR algorithm. By maintaining about four backup service graphs for each session, the PFR algorithm can achieve on average 25 percent lower failure rate than the RR algorithm. We can increase the number of backup service graphs to further reduce the failure rate.

6 Conclusion

This paper has presented a composable stream processing system for cooperative P2P environments. To the best of our knowledge, this is the first work that studied composing stream applications in P2P environments. Specifically, this paper proposes and analyzes a set of efficient algorithms for two key problems in composing stream applications: 1) the global-state-based centralized algorithm versus the local-state-based distributed algorithm for the initial service composition problem, and 2) the reactive failure recovery algorithm versus the proactive failure recovery algorithm for the dynamic failure recovery problem. While finding the optimal solutions for both problems is NP-hard, the proposed approximation algorithms can achieve much better performance than other alternative heuristics under different workloads and system conditions. Our comparison study also reveals the key factors that can affect the trade-offs among different design alternatives. Our results can provide important guidance for adaptively selecting proper algorithms to compose stream processing applications in dynamic P2P environments.

Appendix

Proof scratch of Theorem 1. We prove that both ISC and DFR problems are NP-hard by showing that the *multi-constrained path selection* (MCP) problem, which is known to be NP-hard [11], maps directly to the special cases of both problems. □

Proof of Theorem 2. Let us first consider the basic case where the function graph has a path structure. The first step of the GC algorithm (i.e., constructing the candidate graph) takes time $O(K)$. The time to perform the QoS consistency check is $O(L)$. In the third step, each EXTRACT-MIN operation takes time $O(logK)$ assuming that the priority queue in the Dijkstra algorithm is implemented by a binary heap. Each weight adjustment operation takes time $O(m)$. There are K such operations. Each RELAX operation takes time $O(logK)$. There are at most L such operations. Thus, the total running time is $O(KlogK + Km + LlogK)$. We now prove that the above conclusion also holds for generic function graphs. Suppose the function graph consists of B stages, each stage consists of U disjoint function paths. Let K_i and L_i, $1 \leq i \leq U$ denote candidate service instance number of the ith function path. According to the basic case, the computational complexity of finding the best service path for the function path is

$$O(K_i logK_i + K_i m + L_i logK_i).$$

Thus, the computational complexity of instantiating the whole stage is $O(\sum_{i=1}^{U} K_i logK_i + K_i m + L_i logK_i)$. Let $K'_j = \sum_{i=1}^{U} K_i, 1 \leq j \leq B$ denote the number of all service instances in the stage and $L'_j = \sum_{i=1}^{U} L_i$ denote the number of all service links in the stage. Then, we have

$$O\left(\sum_{i=1}^{U}(K_i logK_i + K_i m + L_i logK_i)\right) < O(K'_j logK'_j + K'_j m + L'_j logK'_j).$$

Thus, the running time to instantiate all stages is $O(\sum_{j=1}^{B}(K'_j logK'_j + K'_j m + L'_j logK'_j))$. Let $K' = \sum_{j=1}^{B} K'_j$ and $L' = \sum_{j=1}^{B} L'_j$. Then, we have

$$O\left(\sum_{j=1}^{B}(K'_j logK'_j + K'_j m + L'_j logK'_j)\right) < O(K' logK' + K' m + L' logK').$$

The last step is to replace all stages with virtual nodes and then run our modified Dijkstra algorithm. The running time of the last step is

$$O((K-K')log(K-K') + (K-K')m + (L-L')log(L-L')).$$

Thus, the total running time is

$$O(K' logK' + K' m + L' logK') + O((K-K')log(K-K') + (K-K')m + (L-L')log(K-K')) < O(KlogK + Km + LlogK).$$

□

Proof of Theorem 3. The first step to generate the root probe takes $O(1)$. In each per-hop probe processing, the step to derive the next-hop service task takes time $O(1)$. Each P2P service discovery takes time $O(log|V|)$ using the distributed hash table [28]. There are K service discovery operations. The rest of the per-hop probe processing operation takes $O(1)$ time. There are totally L operations. In the third step, to select the best service graph, there are at most β different service paths due to the probing budget constraint. The classification takes $O(\beta)$ time. Because the function graph is decomposed into Y function paths, the service paths can be classified into Y sets and each set includes β_i service paths satisfying $\sum_{i=1}^{Y} \beta_i = \beta$. Thus, the merging operation takes time $O(\prod_{i=1}^{Y} \beta_i) < O((\frac{\beta}{Y})^Y)$. The number of complete service graphs is at most $O((\frac{\beta}{Y})^Y)$. The qualified service path selection takes at most $O(m(\frac{\beta}{Y})^Y)$ time. The sorting takes $O((\frac{\beta}{Y})^Y Y(log\beta - logY)$ time. Thus, the total running time is $O(Klog|V| + L + (\frac{\beta}{Y})^Y(Y(log\beta - logY) + m))$. There are K service peers. Thus, the amortized running time at each peer is $O(log|V| + (L + (\frac{\beta}{Y})^Y(Y(log\beta - logY) + m))/K)$. □

Proof of Theorem 4. The global state information at each peer should include the states of all peers and all overlay links in the P2P overlay network. Since each state update message includes the states of at most one peer, each peer needs to receive at least $(|V|-1)$ state update messages to obtain the states of all other peers. Because each state is updated $\lfloor \frac{1}{T} \rfloor$ times per time unit, the number of messages received by each peer per time unit is at least $\lfloor \frac{(|V|-1)}{T} \rfloor$. Thus, the total number of messages received by all peers must be no less than $\lfloor \frac{|V|(|V|-1)}{T} \rfloor$. Because the number of messages generated should be no less than the messages received, the global state maintenance overhead is at least $\lfloor \frac{|V|(|V|-1)}{T} \rfloor$. Thus, the overhead lower-bound of the GC algorithm is at least $\lfloor \frac{|V|(|V|-1)}{T} \rfloor$. □

Proof of Theorem 5. Because the LD algorithm is triggered by user requests, the number of probing processes invoked in the P2P system is no more than Z times per time unit (i.e., request rate). Because the LD algorithm guarantees that the number of probes generated during a single probing process is no larger than the probing budget β, the total number of probes generated per time unit is at most $Z \cdot \beta$. Because each peer has on average σ neighbors and the state update period is T time units, the number of all local state update messages generated in the P2P system is $\lfloor \frac{|V| \cdot \sigma}{T} \rfloor$. Thus, the overhead upper-bound of the LD algorithm is $Z \cdot \beta + \lfloor \frac{|V| \cdot \sigma}{T} \rfloor$. □

Proof of Theorem 6. According to (8), the number of backup service graphs maintained per session is no more than $min(\Gamma, \Theta - 1)$. Because each candidate service graph needs at least one probe, the qualified service service graph number Θ won't exceed the probing budget β. Thus, $min(\Gamma, \Theta - 1)$ should be no more than $min(\Gamma, \beta)$. Because $\beta \gg \Gamma$, the number of backup service service graphs maintained per session is no more than Γ. Thus, each session generates at most $\lfloor \frac{\Gamma}{T} \rfloor$ maintenance messages per time unit assuming we use one maintenance message for each backup service graph. Given the request rate Z and maximum session duration D time units, the number of active sessions during each time unit is no more than $Z \cdot D$. Thus, the upper-bound of backup service graph maintenance overhead is $\lfloor \frac{Z \cdot D \cdot \Gamma}{T} \rfloor$. □

REFERENCES

[1] Gnutella, http://gnutella.wego.com/, 2006.

[2] PlanetLab, http://www.planet-lab.org/, 2006.

[3] T.F. Abdelzaher, "An Antomated Profiling Subsystem for QoS-Aware Services," *Proc. IEEE Real-Time Technology and Applications Symp.*, June 2000.

[4] A. Adya et al., "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," *Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI '02)*, Dec. 2002.

[5] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris, "Resilient Overlay Networks," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP)*, Oct. 2001.

[6] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G.M. Voelker, "TotalRecall: System Support for Automated Availability Management," *Proc. ACM/USENIX Symp. Networked Systems Design and Implementation (NSDI)*, Mar. 2004.

[7] A.P. Black, J. Huang, J. Walpole, and C. Pu, "Infopipes: An Abstraction for Multimedia Streaming," *Multimedia*, vol. 8, no. 5, pp. 406-419, 2002.

[8] M. Castro, P. Druschel, A-M Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-Bandwdith Multicast in Cooperative Environments," *Proc. ACM Symp. Operating Systems Principles (SOSP)*, Oct. 2003.

[9] Y. Chu, S.G. Rao, S. Seshan, and H. Zhang, "A Case for End System Multicast," *IEEE J. Selected Areas in Comm. (JSAC)*, 2002.

[10] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti, "CANS: Composable, Adaptive Network Services Infrastructure," *Proc. Third USENIX Symp. Internet Technologies and Systems*, Mar. 2001.

[11] M.R. Garey and D.S. Johnson, "Computers and Intractability," *A Guide to the Theory of NP-Completeness,* 1979.
[12] X. Gu and K. Nahrstedt, "Distributed Multimedia Service Composition with Statistical QoS Assurances," *IEEE Trans. Multimedia,* 2005.
[13] X. Gu, K. Nahrstedt, R.N. Chang, and C. Ward, "QoS-Assured Service Composition in Managed Service Overlay Networks," *Proc. IEEE 23rd Int'l Conf. Distributed Computing Systems (ICDCS '03),* 2003.
[14] X. Gu, K. Nahrstedt, and B. Yu, "SpiderNet: An Integrated Peer-to-Peer Service Composition Framework," *Proc. IEEE Int'l Symp. High-Performance Distributed Computing (HPDC-13),* June 2004.
[15] X. Gu, K. Nahrstedt, W. Yuan, D. Wichadakul, and D. Xu, "An XML-based QoS Enabling Language for the Web," *J. Visual Language and Computing,* 2001.
[16] X. Gu, P.S. Yu, and K. Nahrstedt, "Optimal Component Composition for Scalable Stream Processing," *Proc. IEEE 23rd Int'l Conf. Distributed Computing Systems (ICDCS '05),* 2005.
[17] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, "Promise: Peer-to-Peer Media Streaming Using Collectcast," *Proc. ACM Multimedia Conf.,* Nov. 2003.
[18] N. Hu and P. Steenkiste, "Evaluation and Characterization of Available Bandwidth Probing Techniques," *IEEE J. Special Areas in Comm.,* special issue on Internet and WWW measurement, mapping, and modeling, vol. 21, no. 6, Aug. 2003.
[19] M. Jain and C. Dovrolis, "End-to-End Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput," *Proc. ACM SIGCOMM Conf.,* Aug. 2002.
[20] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh," *Proc. ACM Symp. Operating Systems Principles (SOSP),* Oct. 2003.
[21] P.K. Mckinley, U.I. Padmanabhan, N. Ancha, and S.M. Sadjadi, "Composable Proxy Services to Support Collaboration on the Mobile Internet," *IEEE Trans. Computers,* vol. 52, no. 6, June 2003.
[22] T.-W. Ngan, D. Wallach, and P. Druschel, "Enforcing Fair Sharing of Peer-to-Peer Resources," *Proc. Second Int'l Workshop Peer-to-Peer Systems (IPTPS '03),* Feb. 2003.
[23] W.T. Ooi and R.V. Renesse, "Distributing Media Transformation over Multiple Media Gateways," *Proc. Ninth ACM Int'l Multimedia Conf.,* Sept. 2001.
[24] J.P. Loyall, D.E. Bakken, R.E. Schantz, J.A. Zinky, D.A. Karr, R. Vanegas, and K.R. Anderson, "QoS Aspect Languages and Their Runtime Integration," *Lecture Notes in Computer Science,* vol. 1511, May 1998.
[25] B. Raman and R.H. Katz, "An Architecture for Highly Available Wide-Area Service Composition," *Computer Comm. J.,* May 2003.
[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *Proc. ACM SIGCOMM 2001 Conf.,* 2001.
[27] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Topologically-Aware Overlay Construction and Server Selection," *Proc. Infocom 2002 Conf.,* 2002.
[28] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms (Middleware),* Nov. 2001.
[29] S. Frolund and J. Koistinen, "QML: A Language for Quality of Service Specification," Technical Report HPL-98-10, Feb. 1998.
[30] A.C. Snoeren, K. Conley, and D.K. Gifford, "Mesh Based Content Routing using XML," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP),* Oct. 2001.
[31] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. ACM SIGCOMM Conf.,* Aug. 2001.
[32] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger, "Network Topology Genrators: Degree-Based vs. Structural," *Proc. ACM SIGCOMM Conf.,* Aug. 2002.
[33] D. Tran, K. Hua, and T. Do, "Zigzag: An Efficient Peer-to-Peer Scheme for Media Streaming," *Proc. IEEE INFOCOM Conf.,* Apr. 2003.
[34] J. Winick and S. Jamin, "Inet3. 0: Internet Topology Generator," Technical Report UM-CSE-TR-456-02, http://irl.eecs.umich.edu/jamin/, 2002.
[35] L. Xiao, Z. Xu, and X. Zhang, "Low-Cost and Reliable Mutual Anonymity Protocols in Peer-to-Peer Networks," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 9, Sept. 2003.

Xiaohui Gu received the BS degree in computer science from Peking University, Beijing, P.R. China, the MS degree in 2001 from the Department of Computer Science, University of Illinois at Urbana-Champaign, and the PhD degree in 2004. She is a research staff member at the IBM TJ Watson Research Center, Hawthorne, New York. 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 the ILLIAC fellowship, the David J. Kuck Best Master Thesis Award, and the Saburo Muroga Fellowship from University of Illinois at Urbana-Champaign, and Invention Achievement award from IBM research. She is a member of the IEEE.

Klara Nahrstedt received the BA degree in mathematics from Humboldt University, Berlin, in 1984, and the MSc 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 the PhD degree from the University of Pennsylvania in the Department of Computer and Information Science. She is a full professor at the University of Illinois at Urbana-Champaign, Computer Science Department. Her research interests are directed toward 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* (Prentice Hall), the recipient of the Early US National Science Foundation 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. She is a member of the IEEE.