QoS-Aware Service Management for Component-Based Distributed Applications

Jingwen Jin and Klara Nahrstedt
Department of Computer Science, University of Illinois at Urbana-Champaign

Component-based software development has evolved from a tightly coupled style to a loosely coupled style in the recent few years. The paradigm shift will eventually allow heterogeneous systems to interoperate in open networks such as the Internet and will make software development more of a management task than a development task. Envisioning that future applications may comprise dynamically aggregated component services possibly distributed widely, we develop a Quality of Service (QoS)-aware service management framework in the middleware layer to make the component services infrastructure transparent to the applications. Specifically, we manage services not only as individuals, but more importantly as meaningful aggregated entities based on the logical compositional needs coming from the applications, by composing services properly according to QoS requirements at application setup time, and performing continuous maintenance at application runtime seamlessly. Our service management framework is scalable in two dimensions: network size and application’s client population size. Specifically, the framework employs a decentralized management solution that scales to large network size, and explores resource sharing in one-to-many group-based applications by means of multicasting mechanisms. Moreover, it incorporates local adaptation operations and distributed failure detection, reporting, and recovery mechanisms to deal with runtime resource fluctuations and failures.

1. INTRODUCTION

Due to their tight coupleness, standard object-based technologies such as CORBA, Java RMI, and DCOM have evolved into the more loosely coupled Service-Oriented Architecture (SOA) based ones [Wikipedia ] in the recent few years. While the object-based component models are suitable in closed systems, SOA has been designed for arbitrary systems to interoperate. The consequence of this paradigm shift is that independently developed and deployed software components will have a chance to interoperate in open systems such as the Internet. Based on this trend, we envision future applications will comprise multiple component services running on top of widely distributed Internet nodes.

To fully support this, it is fundamental that the distributed component service
infrastructure stay totally transparent to the application layer through a service management middleware layer. This layer should be responsible for managing services, both their functional and QoS aspects, not only as individuals, but more importantly as aggregated entities based on the logical compositional needs from the applications. Research in service composition, coordination, and orchestration has been triggered for this purpose.

While work such as [OASIS 2005; Stefan Tai, Rania Khalaf, and Thomas Mikalsen 2004] concentrated mostly on processes of service composition and coordination, we want to specifically focus on performance aspects of service composition in this paper. In the performance respect, the management layer needs to ensure that the application as a whole runs efficiently and robustly, with the following goals:

—**QoS-Awareness and Efficiency**: Since resources, including network bandwidth and machine computational resources, are limited in the physical world, composing services (each with multiple instances in the networked environment) for resource-demanding applications requires QoS-awareness. From the perspective of a single application, all service instances related to the application must run on top of a physical infrastructure with abundant resources, satisfying certain QoS requirements. From the perspective of the networks as a whole, it is fundamental to achieve global resource usage optimization, by wisely allocating limited resources so that the same can sustain more requests simultaneously.

—**Scalability**: While management can be done centrally in small-scale networks and on an end-to-end basis, we want this task to scale to large networks and to large group-based applications due to the potential of SOA.

—**Adaptivity**: During the lifetime of an application, which essentially comprises a path of component services, resource conditions may fluctuate. Properly reflecting resource fluctuations in the existing paths so that QoS is still satisfied or optimal, is important.

—**Robustness**: During the application runtime, services and resources may fail, but such failures should be seen transparently by the applications; i.e., applications must be self-healing and be able to recovery from failures seamlessly.

Although quite some work has been done to concentrate on the performance aspects of the problem, the above important issues have not been addressed or have been only partially addressed.

In [Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, Quan Z. Sheng 2003; Sumi Choi, Jonathan Turner, and Tilman Wolf 2001; Jingwen Jin and Klara Nahrstedt 2004b], global routing state (including QoS and service availability information) of the network is maintained at a single network node, so that computation of service paths can be performed in a centralized manner. Such an approach does not scale to network size. More scalable solution can be found in [Jingwen Jin and Klara Nahrstedt 2003a] by introducing hierarchies into the network and in [Xiaohui Gu, Klara Nahrstedt 2002] by decentralizing routing decision into a hop-by-hop manner. However, these solutions do not fully optimize application-specific QoS. In addition, they do not consider the global resource usage optimization problem. The global resource usage optimization problem was dealt with in [Jingwen Jin and Klara Nahrstedt 2003b; Zhichen Xu, Chunqiang
by introducing the service-added multicast routing mechanism in one-to-many application scenarios. However, service tree construction in [Jingwen Jin and Klara Nahrstedt 2003b] takes a source-based approach, which is simple but has constrained scalability in network size. Service tree construction in [Zhichen Xu, Chunqiang Tang, Sujata Banerjee, Sung-Ju Lee 2003] is slightly more scalable, by having a DHT infrastructure maintain the global routing state as well as the service tree information. This approach imposes extra burden onto the DHT infrastructure. Furthermore, the landmark-vector-based clustering is not precise in predicting Internet distances among participating nodes. From the descriptions in [Zhichen Xu, Chunqiang Tang, Sujata Banerjee, Sung-Ju Lee 2003], it is not clear how the approach maintains a service tree. In our previous work [Jingwen Jin and Klara Nahrstedt 2004a], we provided a decentralized service composition solution. However, maintenance issues such as adaptation and failure recovery have not been dealt with. In terms of maintenance, previous works [B. Raman and R. H. Katz 2003; Xiaohui Gu, Klara Nahrstedt 2002] have simply used a secondary service path as an alternative when the primary service path fails. Computation of secondary path is the same as that of the primary path. However, as we will see, such a protection-based failure recovery mechanism is prohibitively expensive and inadequate for multicast scenarios.

In this paper, we provide a solution framework that addresses the above performance goals in an integral way. When goals such as global resource usage optimization get introduced, the solution complexities will increase significantly, incurring more complex service composition and maintenance (including adaptivity and robustness) schemes. In a nutshell, our solution framework has the following features:

We perceive scalability in two dimensions: network size and application’s client population size, and our solution targets to achieve both. In terms of network size - when the network becomes too large such that centralized service management relying on global knowledge of the network becomes impractical, we achieve scalability by investigating a decentralized approach. In this approach, nodes only maintain local knowledge (each node may know the QoS and service states of a very limited set of neighbors), and service paths are computed in a hop-by-hop manner. However, a hop-by-hop path finding approach usually suffers from myopic effects, as local-based heuristics do not attempt to account for optimization of additive metrics (e.g., end-to-end delay) [Xiaohui Gu, Klara Nahrstedt 2002]; i.e., local-optimality property does not lead to global (such as total delay) path optimality. Our solution is to incorporate the easily maintainable geometric location information into the service management framework by which hop-by-hop service composition can be guided to achieve optimal end-to-end delay performance. In terms of application’s client population size - when the application involves one-to-many group communication, we achieve scalability by exploring resource sharing (in terms of network bandwidth and computational resources) by proposing two types of multicasting: service multicast and hybrid multicast. In the one-to-many application domain, the problem of computing an optimal multicast tree (Steiner tree) in traditional routing has been recognized as an NP-complete problem. Having multiple functionalities in network nodes makes multicasting even more complicated. We build service multicast trees on an incremental basis to naturally cope with dynamic
membership features in a multicast group. Realizing that service multicasting has not fully explored network bandwidth sharing, we further propose to integrate data multicasting into service multicasting, thus providing a hybrid type of multicasting solution. Our main purpose for proposing multicasting solutions is to achieve global optimization of resource usage by means of sharing.

At runtime, in order to keep the incrementally constructed service multicast tree near-optimal for the current membership, we devise distributed local performance monitoring and adaptation schemes; we let on-tree nodes monitor their local performances, and trigger local adaptations when needed to gradually improve the global tree performance. Failure recovery in service multicast is made more difficult than it is in traditional multicast, because a single network node failure may incur multiple logical service failures (which may or may not span multiple branches) of the tree. We devise distributed failure detection, reporting, and recovery mechanisms to effectively deal with failures.

This paper targets to solve service management (service composition plus runtime service maintenance) issues in a reasonably stable (no mobility and low network failure rate) proxy network for applications with high demands on resources such as computational power and network bandwidth. However, although our discussions are revolved around multimedia streaming applications, the solutions apply to general component-service-based applications requiring QoS treatment.

The paper will be structured as follows. We describe some motivating application scenarios that will benefit from the component services model in Section 2. Section 3 introduces notations and terminologies to be used in the main text. We then briefly present SOA as well as its implications on middleware support in Section 4. We describe our service management framework in Section 5 with three sub-sections. In Section 5.1 we describe our Geometric-Location-Guided decentralized hop-by-hop service composition solution that scales to large networks. In Section 5.2 we go on to explore scalability in terms of application’s client population size by means of service multicasting (Section 5.2.1) and hybrid multicasting (Section 5.2.2). Runtime maintenance issues are described in Section 5.3. We validate our QoS-aware service management framework in the well-known network simulator ns-2 and provide our performance results in Section 6. Section 7 concludes the paper.

As in certain respects, service composition resembles the traditional QoS routing problem with an additional requirement for paths/trees to be functionally correct (i.e., paths and trees need to encompass required services in required dependences), later on we will use the terms service-added routing and service composition intermittently.

2. MOTIVATING SCENARIOS

The Internet has long been recognized as an environment with heterogeneities everywhere, happening in every aspect. The heterogeneity problem has been further exacerbated with the increasing popular uses of small devices connecting to the Internet through wireless links in recent years. With a diverse spectrum of devices, ranging from powerful desktops, to less powerful and energy-sensitive laptops, hand-held computers, PDAs, and mobile phones, communicating over networks with varied bandwidths by using different protocols, there is a strong need...
Fig. 1. Two scenarios that make use of composite services: (a) A mobile phone user retrieves a Web document written in Latin and listens to its summary in English; (b) news video from CNN or Yahoo server is customized within a service network according to end users’ network and machine capacities.

to perform protocol and content translations between communicating parties to bridge the gaps. Value-added, transformational services have been proposed for such purposes [Rakesh Mohan, John R. Smith and Chung-Sheng Li 1999; Surendar Chandra, Carla Schlatter Ellis, and Amin Vahdat 2000]. However, given the range of diversities involved in the Internet, developing monolithic transformational services to bridge all conceivable end-to-end heterogeneities would be, if not totally impossible, some task that requires tremendous amount of effort.

With the emerging component service model described above, we can rely on complex transformational services to be dynamically aggregated from primitive ones, thus achieving dynamic customization and reusability [S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao 2001; A. Ivan, J. Harman, M. Allen, and V. Karamcheti 2002; Shankar R. Ponnekanti and Armando Fox 2002; Stefan Tai, Rania Khalaf, and Thomas Mikalsen 2004]. We depict two interesting and useful Internet applications below.

— One-to-one scenario: Figure 1(a) depicts a one-to-one scenario where a cell phone user who understands only English wants to listen to a summary of a written document in Latin. The document can first be translated from Latin to English, then summarized, and finally converted to speech. We call an end-to-end network path comprising a sequence of primitive service instances in a one-to-one scenario a service path.

— One-to-many scenario: Figure 1(b) depicts a one-to-many Web-based multimedia application where news videos from a CNN or Yahoo server get customized (e.g.,
transcoded, filtered) according to end users’ network and machine capacities. Multiple end users may be interested in receiving the same source data, with the same or different requirements on customizations.

From the scenarios depicted in Figure 1, we see how the emerging SOA model can achieve content customization on-the-fly, thus bridging disparate data between sources and destinations and leveraging the Internet heterogeneity problem. Based on the motivating scenarios described above we assume all transformational component services reside in a proxy network such as Akamai’s caching proxy network or any network of proxies provided by commercial ISPs and/or other organizations. Therefore, it can be assumed the end hosts on which component services are run form a reasonably stable overlay service network with no mobility and low network failure rate. For the purpose of illustration, we use multimedia transformational services (e.g., language translators, document format translators, image/video quality converters) as examples of component services in this paper, but all of the general concepts in this paper could also apply to other types of component-service-based applications (e.g., business applications such as travel planning and on-line merchandize ordering).

3. NOTATIONS AND TERMINOLOGY

A service component is associated with an input data QoS - $Q_{in}$, and an output data QoS - $Q_{out}$, where $Q_{in}$ and $Q_{out}$ are QoS vectors of multiple application-level QoS parameters such as image size, image resolution, video frame rate. Each service $s$ has its resource usage function defined as $R_s: Q_{in} \times Q_{out} \rightarrow R$, that computes the amount of resources needed to deliver an output QoS $Q_{out}$ when $Q_{in}$ is the input QoS. For notational conveniences, later on we will only use $s_x$ to denote a component service, as opposed to exemplifying with a complex service name such as MPEG2H263 video converter or Latin2English language translator.

Since server- and client-side applications rely on third-party entities (machines that are deployed with special services) to do transformations, we call these third-party entities proxies. We denote the functional part of a service request as $SR = (p_s, s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots, p_d)$. The request is for finding a service path between the source $p_s$ and the destination $p_d$ containing $s_1$, $s_2$, and $s_3$... in sequence. A concrete service path actually represents a mapping from the service request to overlay nodes that are capable of providing the requested services at the required QoS level, and will be denoted as $SP = (p_s \rightarrow s_1/p_\alpha \rightarrow s_2/p_\beta \rightarrow s_3/p_\gamma \rightarrow \ldots \rightarrow p_d)$ ($s_i/p_\theta$, which we call a service node, means service $s_i$ is provided by or mapped onto proxy $p_\theta$).

We define service neighbor of a service $s_i$ as $s_i$’s next service in service graphs. For instance, if $SG_1 = s_1 \rightarrow s_2 \rightarrow s_3$ and $SG_2 = s_1 \rightarrow s_4$, then $s_1$’s service neighbor can be either $s_2$ or $s_4$, depending on which service graph is in use. We also define next service hop of a service node $s_{nx}$ to be an instance of $s_{nx}$’s next logical service in the request. Thus, if $SP = (p_s \rightarrow s_1/p_\alpha \rightarrow s_2/p_\beta \rightarrow s_3/p_\gamma \rightarrow \ldots \rightarrow p_d)$, then $p_s$’s next service hop is $s_1/p_\alpha$, and $s_1/p_\alpha$’s next service hop is $s_2/p_\beta$ and so forth.
4. SERVICE ORIENTED ARCHITECTURE AND ITS IMPLICATIONS ON MIDDLEWARE SUPPORT

In the SOA model, a service is a self-contained, stateless business function that accepts one or more requests and returns one or more responses through a well-defined, standard interface. Loose coupling in SOA means two things: self-containedness of individual services and loose coupling in service instantiation. In SOA, services are supposed to be independently developed, deployed and discovered. To achieve loose coupling in service instantiation, SOA incorporates a service discovery system, whose main task is to discover services and maintain their meta-data (including functional specification and QoS specification) for future service requests from service consumers. A discovered service is bound between the service provider and the service consumer via a service contract and subsequently executed. This “find, bind, and execute” paradigm not only provides transparency in service locations, but also enables late binding between service providers and service consumers.

To make the SOA concept widely accepted, there has to be some middleware support which can make the component service infrastructure transparent to end users. A user should be unaware of the service component infrastructure as well as all or most related negotiations. Assuming the scenarios depicted in Figure 1, to achieve simplicity at the end user side, we can delegate all management-related tasks to a nearby user proxy. The client may first contact its nearby proxy, which can then negotiate the QoS specifications between the server and client, to derive the Service Graph (SG) that is needed to bridge the gaps between the communicating parties. Once obtaining the SG, the proxy could further contact a service discovery agent to learn where instances of services are located. With this knowledge, depending on how much performance-related state information the proxy has about the network, the proxy can initiate service path computation. Below we depict the roles in more detail.

—QoS compilation: QoS compilation refers to a process of obtaining a Service Graph (SG) that is needed to bridge content and protocol gaps between two communicating ends based on their QoS specifications. In such an SG (Figure 2), all resource requirements (e.g., in terms of CPU, memory, network bandwidth) are defined. Work in [Duangdao Wichadakul 2003] describes a meta-data QoS programming and compilation framework. The work proposes a set of translation models and their compilations that enable semantic interoperability among interoperable components’ specific QoS requirements and provisions.

—service description and discovery: Before a developed service component is deployed, it needs to be associated with an unambiguous name and/or an interface describing the component’s inputs and outputs. To promote interoperability, the W3 Consortium and OASIS together have defined several standards (e.g., WSDL, UDDI, and SOAP) for Web Service Architecture (WSA), which can be seen as an instance of SOA. WSDL (Web Service Description Language) is an XML-based language for describing Web services. Service components need to be published and later on discovered before being composed. UDDI (Universal Description, Discovery and Integration) creates a standard interoperable platform that enables companies and applications to publish and find Web services. Scalable ways of performing service discovery have been also investigated in peer-to-peer networks.
Fig. 2. A service request with a linear service graph (SG): between the source and the destination, the gap is filled by a service graph $s_1 \rightarrow s_2 \rightarrow s_3$.

Fig. 3. The service management substrate resides between the application layer and the service discovery/QoS compilation layer to make component services transparent to the application layer.

—Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan 2001; Sylvia Ratnasamy, Pau Francis, Mark Handley, Richard Karp, Scott Shenker 2001]. SOAP is a transport layer for sending messages between service consumer and service provider. Thus, a service consumer can search for a service in the UDDI registry, get the descriptions of the service written in WSDL, and invoke the service using SOAP messaging.

—service management: Since a service discovery system’s task is only to locate instances of single services, and a QoS compiler’s task is only to obtain a system-independent meta service graph, there needs to be a service management layer that resides above these tasks and that can choose appropriate service instances (returned by a discovery system) for the logical components in a service request (returned by a QoS compiler), so that users at the application layer will perceive the application as an integrated service, rather than separate components (Figure 3). The management substrate should also make sure that the selected service paths run continuously despite resource fluctuations and failures. This management substrate will be the focus of our study.

ACM Transactions on Internet Technology, Vol. ?, No. ?, to appear
5. SERVICE MANAGEMENT FRAMEWORK

In this section, we describe the major features of our service management framework.

Before describing our service management framework further, we first present an enhanced service discovery system based on which our service management framework will be built. A service discovery system’s task is to return service instances’ locations, typically the IP addresses of the hosts in which instances are resided. However, with only the IP address information, it is hard to estimate how far away service instances are located from each other, thus making hop-by-hop routing decisions also hard if communication delay is a concern. We address this weakness by associating each Internet host with geometric coordinates (which is easily maintainable and can be retrieved by enhanced service discovery engines) and using it to estimate Internet distances (communication delays) between hosts. As mentioned earlier, the relative geometric coordinates of nodes in a large network can be efficiently obtained by the Global Network Positioning (GNP) approach [T. S. Eugene Ng, Hui Zhang 2002]. As will be clear later, the added geometric location information in the service discovery system will serve us as guidance for finding more delay-efficient service paths/trees.

Our service management framework includes three different stages: pre-stage (status collection), path finding stage (composition), and runtime stage (maintenance). The pre-stage refers to collection of routing states related to individual services, including their functional specifications and QoS states, such as failure rate, availability, monetary cost, amount of available machine resources, and bandwidth, before any composition is performed. In our case, routing states are measured on-demand; i.e., upon receiving a request for routing, a network node initiates certain probing activities to learn about the resource conditions of its associated neighbor as well as the link conditions (e.g., end-to-end available bandwidth) in between. Methods for measuring end-to-end available bandwidths can be found in [B. Melander, M. Bjorkman, and P. Gunningberg 2000; Manish Jain, Constantinos Dovrolis 2002]. While the pre-stage is only briefly presented, we will detail descriptions of service composition and service maintenance stages in the following subsections.

To maximize routing efficiencies at the overlay layer, we do not set network topology constraints. That is, for data delivery, the initial network is a fully connected, unstructured topology\(^1\), and a service path/tree is built for each application scenario. However, while service paths/trees are built on top of an unstructured overlay topology, we maintain another structured mesh topology for general control message communication. Basically, this mesh topology will be used for nodes to find

---
\(^1\)Overlay network routing can be performed either on top of structured topologies [Y. Chu, S. G. Rao and H. Zhang 2000; Jingwen Jin and Klara Nahrstedt 2003b] or on top of unstructured topologies [Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, Quan Z. Sheng 2003]. The former approach views the overlay network topology as a partial mesh, so that the same routing protocols designed for the physical network, such as OSPF, MOSPF, and DVMRP, can be directly employed at the overlay layer. In the latter approach, hosts are considered fully connected, and for each application, a special topology, e.g., a multicast tree, is built and maintained.
nearby on-tree nodes in multicast scenarios by sending requests toward the source node and hoping that the requests will hit some on-tree nodes before reaching the source. Note that the tree and the mesh are employed for different purposes: the former is used for content distribution and the latter is used for control messages.

For communication efficiency, we connect the overlay network nodes (for control message purposes) into a Delaunay triangulation [J. Liebeherr, and M. Nahas 2001], because Delaunay triangulation is a spanner graph in which a path has length bound by a constant times the straight-line distance between the endpoints of the path. Methods of incremental construction of Delaunay triangulation network have been derived [J. Liebeherr, and M. Nahas 2001]. Since the proxy network is relatively stable, maintaining such a topology requires very low overhead. By using such a geometric topology, control messages can be routed by using an on-line, local routing method, such as the compass routing approach [Evangelos Kranakis, Harvinder Singh, Jorge Urrutia]. Note that the use of Delaunay Triangulation and on-line compass routing for control message routing is only our choice. An alternative could be to link the nodes into a mesh and then have nodes maintain routing tables by using, e.g., distance vector or link state protocols. However, since we are dealing with very large network sizes, such an approach is inefficient.

5.1 Service Composition: Scalability in Network Size

At a high level, composition equals to establishing a QoS-satisfied service path based on certain QoS requirements. For instance, the application may require that all related service instances run on machines with sufficient machine resources including memory space and CPU cycles, that the end-to-end service path contain sufficient bandwidth and be delay-optimal (or shortest).

When the network is small so that network nodes can afford to maintain global state of the system, service composition (as well as service maintenance) may be done in a centralized manner [Jingwen Jin, Klara Nahrstedt 2002]. However, centralized solution relying on maintenance of global system state becomes infeasible when the network size grows to a certain point. Two rules of thumb applied in networking and distributed systems areas to achieve scalability are to adopt hierarchical or distributed solutions. A hierarchical-based approach has been investigated in [Jingwen Jin and Klara Nahrstedt 2003a]. In this paper, we choose to adopt the second solution, by letting nodes obtain the QoS state of a limited number of neighbors, and then performing a hop-by-hop based service selection at the time of composing services. Similar approach has been used in traditional QoS routing.

In traditional QoS routing, hop-by-hop routing can be classified into two categories: single-path routing (SPR) and multiple-path routing (MPR). In SPR, one single path is probed for QoS, while in MPR, multiple candidate paths are probed, and then among the candidate paths, the best one is selected [S. Chen, K. Nahrstedt 1998]. Usually MPR is done by multiplicative probe messages at outgoing links as the probing proceeds. To control probing overhead, special rules or mechanisms have to be adopted to constrain the number of probes multiplicated at outgoing links. MPR may find better paths than SPR, but at the cost of more message overhead. To minimize the overhead spent on probing, we adopt an SPR approach in this paper. However, we set guidance for SPR so that the probed path is likely to be a good one.
In QoS (data) routing, starting from one end, the shortest network path towards the other end is usually probed for QoS. If, at certain point, insufficiency of resources is detected, the probe will detour to other neighboring links/nodes [Shigang Chen, Klara Nahrstedt, Yuval Shavitt 2000]. In data routing there is always the shortest network path (maintained by, e.g., the distance vector or link state protocol) that serves as a guide for hop-by-hop QoS path finding so that the computed QoS-satisfied path is not unnecessarily long. However, in service routing, due to the unexpected functional dependency relations among services, no similar shortest service paths can be easily maintained as to allow a node to quickly lookup for the next service hop along the shortest service path to destination. Due to this reason, hop-by-hop-based service composition will have to mostly rely on resource conditions that are probed on-demand.

Our hop-by-hop approach to computing service paths will be based on routing states obtained by on-demand resource probing as well as the geometric location information of the service instances. Generally speaking, starting from the source node, we gradually add to the path those instances of required services as we route toward the destination. The source may first discover the locations of all requested services’ instances by invoking a geometric-location-enhanced service discovery system. After that, a service path can be resolved in a hop-by-hop manner as follows. Each hop sends QoS probe messages to all instances of its service neighbor, and then among the instances that satisfy resource requirements, the current hop will select the one that has largest amount of available resource and that is on the way to destination.

Below we describe previous work on hop-by-hop service composition as well as our approach in more detail.

5.1.1 Local-Heuristics-Based (LHB) Approach. Existing SPR-based hop-by-hop unicast QoS service composition approach [Xiaohui Gu, Klara Nahrstedt 2002] works as follows: starting from the source, the current node selects, among many probed service neighbors, the one whose aggregate value of available bandwidth, machine resources and machine’s up time is optimum. We name this approach Local-Heuristics-Based approach, because routing decisions are based on heuristics obtained within one hop of distance. The local heuristics alone, however, would only potentially optimize the path’s overall concave or multiplicative metrics (e.g., the path’s bottleneck bandwidth or robustness) and may help balance the network and machine loads, but does not pose any constraint on the length of the overall service path, which is an additive metric that requires special planning. Without planning of any sort for optimizing path lengths, service paths computed hop-by-hop by adopting local heuristics tend to be long, and inevitably consume more network resources.

A simple example is illustrated in Figure 4(a). Suppose we want to find a good instance of \( s \) between \( p_s \) and \( p_d \), and suppose \( p_s \) detects that both instances of \( s (s/p_i \text{ and } s/p_j) \) are equally good in terms of network bandwidth and machine capacity, for being unaware of the locations of the two service instances relative to the source and destination, \( p_s \) may choose \( s/p_i \), which is an instance off the way to destination.
5.1.2 **LHB Enhanced with Geometric Location Guidance.** The weakness of LHB can be remedied if we enhanced the service discovery system and let it return also the geometric location information of the queried service instances. As stated, geometric coordinates of network nodes can be efficiently obtained from the GNP approach [T. S. Eugene Ng, Hui Zhang 2002]. By doing so, we can let the current node select the service instance, among those satisfying all resource requirements, that lies on the shortest service path (estimated by the hosts’ geometric locations) from current node to destination. This is illustrated in Figure 4(b): for being aware of the geometric positions of the service instances, $p_s$ may be able to choose $s/p_j$ as $p_j$ lies on the shortest path from $p_s$ to $p_d$; in other words, $s/p_j$ is “on the way to destination”.

Whether or not a service node lies on the way to destination can be computed as shown in the following example. For simplicity, the example focuses on optimizing overall path length by means of geometric location guidance, and we call this approach Geometric Location Guided (GLG). In Figure 5, we want to find a path between the source $p_s$ and the destination $p_d$, with $SG = s_1 \rightarrow s_2 \rightarrow s_3$. Before starting the hop-by-hop routing, $p_s$ invokes an enhanced service discovery system to learn about the locations (including IP address and geometric location) of candidate instances of all services in $SG$. In Figure 5(a), knowing $p_1$ and $p_2$ are hosts in which $s_1$ resides, $p_s$ probes available end-to-end bandwidth and delay information to $p_1$ and to $p_2$, as well as available machine capacities of $p_1$ and $p_2$. Based on the probing results, $p_s$ derives the correspondent overlay map of service instances - a DAG (Directed Acyclic Graph) where nodes represent service instances and links represent dependency relations among the instances. First-hop nodes and links that do not meet resource requirements or are failed are excluded (represented in the figures in dashed circles or lines). At $p_1$, suppose both instances have sufficient resources, $p_s$ then applies a shortest paths algorithm [Jingwen Jin, Klara Nahrstedt 2002] on top of the DAG to obtain a shortest service path (shown in bold lines). The first hop along the shortest path will be chosen as our next hop, as it is most probably on the way to future service instances and the destination (Figure 5(a’)).

In Figure 5(b), once at $p_1$, $p_1$ probes the resource conditions of three instances of next service in the request - $s_2/p_3$, $s_2/p_4$, and $s_2/p_5$. Figure 5(b’) shows how $p_1$ chooses the most delay-efficient and QoS-satisfied next service hop. Assuming in this case the probed bandwidth between $p_1$ and $p_3$ does not meet the requirement,
Fig. 5. Finding a QoS-satisfied and potentially shortest service path hop-by-hop from $p_s$ to $p_d$ that satisfies the service graph $s_1 \rightarrow s_2 \rightarrow s_3$.

the correspondent link becomes deleted (shown in dashed line) from the service DAG. Such a hop-by-hop process continues until all of the services in the request have been resolved.

Combining LHB and GLG, selection of next hop can follow one of the following approaches: (a) **LHB-GLG**: applies a shortest paths algorithm [Jingwen Jin, Klara Nahrstedt 2002] on top of the DAG to obtain the shortest service paths, identifies the next service hops that potentially lead to shortest service paths, and then among the potential next hops that lie on the shortest paths select the one that is best in terms of available resources; (b) **GLG-LHB**: among the potential next
hops that are best in terms of resources, select the one that potentially leads to shortest path. LHB-GLG and GLG-LHB actually resemble the \textit{widest-shortest} and \textit{shortest-widest} approaches in traditional QoS routing respectively. An additional advantage of LHB-GLG is that it would also reduce probing overhead, as only the hops along the shortest paths are probed for resource conditions.

5.1.3 Routing Backtracking. SPR-based hop-by-hop routing may end up in an unsuccessful state even if there exists a qualified path elsewhere. For improved success rate, routing should backtrack to other unexplored branches if the current probe yields a dead end (e.g., when resource conditions of the candidate node-branches are not satisfactory; or when probes yield no responses because of node/link failures). In this paper, we consider back-tracking to immediately-previous node if the current node/link yields unsatisfactory performance quality. Future routing trials will exclude the detected failed nodes and links to avoid repetitions.

5.2 Service Composition: Scalability in Application’s Client Population Size

In this section, we propose two multicast mechanisms that are suitable in an SOA environment: service multicast and hybrid multicast.

5.2.1 Service Multicast. When a multimedia stream is delivered to a group of users that demand different transformational rules on the stream, then instead of having the stream transformed and delivered through multiple independent service paths, we propose to construct a single service multicast tree for transformation and delivery purposes to explore resource sharing. We call this delivery mode service multicast, to differentiate it from traditional (data) multicast\textsuperscript{2}. In the same way that data multicast [Steve Deering 1988] helps to achieve scalability in group-based communications model by means of bandwidth sharing, service multicast achieves the similar goal by means of bandwidth and computational resource sharing. Computing a globally optimal tree is known as the Steiner tree problem that is NP-hard. Traditional multicast tree construction has adopted heuristic solutions that usually branches out from an existing node to cover one member at a time, until all members have been included in the tree. Such a heuristics-based incremental solution also naturally supports the dynamic membership feature required by many applications. Our design is based on the same idea as we build a service multicast tree based on the service unicast solution.

While source-based (pure) service multicast has been proposed and studied in our previous work [Jingwen Jin and Klara Nahrstedt 2003b] for small service networks, at the same time that scalability in application’s client population size is being explored in this paper, we also consider scalability in network size. Therefore, we devise a fully distributed approach for service multicast. By distributed, we mean not only service path/tree construction, but also multicast group management as well as tree maintenance (including adaptation and failure recovery), will be performed distributively.

\textbf{Graftable Node:} A key issue in incremental multicast tree building is to find a point of attachment (\textit{graftable on-tree node}) for the new joining member. In

\textsuperscript{2}A service multicast tree is a tree where nodes have different functionalities, while a data multicast tree is a tree where all nodes act as simple relay nodes.
traditional data multicast, every on-tree node can be such a point because the original data from the root get forwarded as is by all on-tree nodes. The problem is only in how to find a good one. In the Protocol Independent Multicast (PIM) protocol, a newly joining member \( p_m \)'s request is forwarded towards the source along the shortest path, and the first on-tree node \( p_g \) hit by the request becomes the \textit{graftable on-tree node} for \( p_m \). Unlike the conventional data multicast, where every on-tree node functionally qualifies as a graftable node for all other group members, in service multicast, not all on-tree nodes functionally qualify as graftable nodes for other joining members. In fact, due to the functionality issues, an on-tree node \( p_g \) only qualifies as a graftable node for a member \( p_m \) (whose service request is \( SR \)) if \( p_g \)'s up-tree service path (the service path from the root to \( p_g \)) is a prefix of \( SR \). Let \( SP = (p_s \to s_1/p_s \to s_2/p_3 \to s_3/p_7 \to s_4/p_4 \to \ldots \to p_{d1}) \) denote a service path, and let \( SR = (p_s, s_1 \to s_2 \to s_3 \to s_5 \to \ldots, p_{d2}) \) denote a service request, then several nodes \((p_s, s_1/p_3, s_2/p_3, p_3/p_5)\) in \( SP \) qualify as functionally graftable service node for \( SR \). To maximize service sharing, we use the \textit{longest match} (prefix) [Jingwen Jin and Klara Nahrstedt 2003b] criterion when selecting a graftable service node. We call the graftable service node selected by the longest prefix criterion the \textit{best functionally graftable service node}. In this case, \( s_3/p_8 \) is the best functionally graftable service node, because \( s_1 \to s_2 \to s_3 \) is the longest prefix of \( SP \) and \( SR \) and its last service - \( s_3 \) - is mapped onto \( p_7 \).

**Incremental Service Tree Construction:** Construction of our service multicast tree will take the following procedures. Each member joining the multicast group sends its request \( SR \) towards the source through the structured overlay network topology (in our case the Delaunay triangulation) by using compass routing. For each overlay node \( p_i \) that is hit by the request, it is verified if \( p_i \) is an on-tree node. If it is not, then \( p_i \) simply forwards the original request to the next hop (computed by compass routing) towards the source, and if it is, it tries to match \( SR \) with the local copy of functional service tree \( T_f \) (management of \( T_f \) will be discussed further) to identify the best functionally graftable service node \( p_g \). The current node \( p_i \) then forwards the request to \( p_g \) if \( p_g \neq p_i \). With a prefix of \( SR \) satisfied at point \( p_g \), \( p_g \) calculates the suffix of \( SR \), and starts a hop-by-hop routing process (by using a unicast service routing solution described in Section 5.1) towards destination \( p_m \) for the suffix of \( SR \).

**Tree Management:** We now briefly describe the tree management issue. In data multicast, routers express their join/leave interests through IGMP (Internet Group Management Protocol). Since all routers have one single function - to forward data as is, they basically need to be only aware of their children in the multicast tree. However, the same information is insufficient in service multicast due to additional service functionality constraints. In service multicast, in order to be able to identify graftable service nodes for new requests, an on-tree node must know the functional tree information of the multicast group. This implies that whenever the functional aspect of the service tree has been modified, tree state needs to be updated in all current on-tree proxy nodes.

**An Example:** Figure 6 depicts an example of how a service multicast tree is built and managed. In Figure 6(a), assume \( p_{d1} \) is the first group member. After \( p_{d1} \) has joined, the on-tree proxy nodes \( p_s, p_1, p_4, p_7 \), and \( p_{d1} \) will obtain a copy of
Fig. 6. (a) A service request message is sent from the newly joining member \( p_{d2} \) towards the source by using compass routing. The request hit an on-tree node \( p_1 \) before it reaches \( p_s \). Since every on-tree node maintains \( T_f \), \( p_1 \) found that \( p_4 \) is the best graftable node for the current request, thus forwarding the request to \( p_4 \). (b) A service branch satisfying the suffix of the original request is established hop-by-hop from the graftable node \( p_4 \) to \( p_{d2} \).

the functional service tree - \( T_f \) - depicted on the right side of Figure 6(a). When \( p_{d2} \) joins, a service request \( SR_2 = (p_s, s_1 \rightarrow s_2 \rightarrow s_4, p_{d2}) \) is sent from \( p_{d2} \) towards the source by using compass routing. The request hits an on-tree node \( p_1 \) before it reaches \( p_s \). Since \( p_1 \) has a copy of \( T_f \), it finds that \( p_4 \) is the best functionally graftable node for the current request, thus forwarding the request to \( p_4 \). In Figure 6(b), a service branch is established hop-by-hop from the graftable node \( p_4 \) to \( p_{d2} \). Since the graftable node \( p_4 \) has already satisfied a prefix of \( SR_2 \), only the correspondent suffix needs to be satisfied by the new service branch from \( p_4 \) to \( p_{d2} \).

After finishing the join operation, \( p_{d2} \) broadcasts adequate message to on-tree nodes so that they incorporate the new functional branch into the old \( T_f \). The functional service tree \( T_f \) maintained by all on-tree nodes will thus become that on the right-side figure of Figure 6(b). Note that \( T_f \) only needs to be updated if the service tree has been modified functionally. As an example, if a third join request has the form \( SR_3 = (p_s, s_1 \rightarrow s_2 \rightarrow s_3, p_{d3}) \), then \( p_{d3} \) can get attached to \( p_r \) without functionally changing the service tree. Therefore no updates are needed.

It is easy to see that service multicast definitely helps to save machine resources because each service in the functional service tree gets executed only once. It should also reduce network bandwidth consumption compared to service unicast, as in most cases, we can expect the length of a service branch satisfying only the

\[^3\text{An algorithm for effectively computing an on-line multicast tree based on nodes’ geometric loca-}\]

\[^3\text{tions can be found in [Kai Chen, Klara Nahrstedt 2002].}\]

ACM Transactions on Internet Technology, Vol. ?, No. ?, to appear
suffix of the request to be shorter than an individually built service path that needs to satisfy the whole request.

5.2.2 Hybrid Multicast. In pure service multicast, each service branch gets directly attached to its best functionally graftable node. However, in doing so, bandwidth usage may not have been optimized. An example is illustrated in Figure 7(a): the proxy providing the MPEG2H261 transcoding service needs to send four separate copies of transformed data to its downstream nodes. Likewise, the node of quality filter will send two separate copies of filtered data to the downstream nodes. The scenario illustrates that data delivery in those sub-groups are sub-optimal. First, it is expensive to do so, because bandwidths need to be separately allocated. Second, after a node’s (e.g., the one offering MPEG2H261) outbound network bandwidth usage reaches its limitation, then no new service branches can be created starting from that point.

We address these weaknesses by further employing data multicast in the local sub-groups. Although IP-layer multicast would be a solution, in this research, we will only exploit data multicast at the application layer because, different from the IP-layer multicast, application-layer multicasting does not require support from the infrastructural level. Our target is, taking Figure 7(a) as an example, to build a hybrid multicasting scenario that explores, in addition to service multicast, data multicast in the subgroups 1 and 2, as shown in Figure 7(b). In addition to boosting the overall cost efficiency of the service tree, exploring data multicast would also increase possibility of finding successful service branches when resources are scarce.

Tree Management: To realize such a hybrid multicast scenario, we make each on-tree (physical) proxy and (logical) service node to keep two trees respectively: the global functional service tree ($T_f$) and the local data distribution tree ($T_d$). Since two types of tree exist in the hybrid multicast case, we will call nodes on the functional tree $T_f$ on-functional-tree nodes to explicitly mean they are nodes providing specific functionalities, rather than nodes that only perform relaying of data. The same as in service multicast, each on-functional-tree proxy will keep an updated $T_f$, which is the functional service tree of the whole multicast group. In addition to $T_f$, each on-tree service node $n$ also keeps a $T_d$, whose root is itself, and whose lower-level members are its children in $T_f$ ($T_d$ should also maintain

![Fig. 7. (a) Pure service multicasting; (b) hybrid multicasting (service multicasting + data multicasting).](image)
the location information of its nodes, for some purpose that will be clear soon). While $T_f$ is global and its maintenance is still to enable on-functional-tree nodes to individually search for functionally graftable nodes for other joining requests, $T_d$ is local and is maintained for exploiting benefits of data multicast in subgroups.

**Parent Switching Protocol:** When a new service branch gets attached to a graftable node $p_g$, initially, $p_g$’s $T_d$ will have the branch’s first node (say $p_x$) attached to itself. However, as $p_g$ is aware of the geometric locations of its $T_d$’s nodes, it will be able to identify which nodes are closer to $p_x$ than itself. If there is any such node, then $p_g$ will initiate a *parent switching protocol*, so that at the end, $p_x$ gets attached to a closer parent with sufficient network bandwidth. Note that the parent switching protocol is only for switching parent in the local data distribution tree, it does not affect the global functional service tree.

The *parent switching protocol* works as follows. First, $p_g$ sends $p_x$ a list of nearby nodes in an increasing order of distance. Upon receiving the list, $p_x$ starts to probe the bandwidth conditions from itself to the listed nodes one by one in the increasing order of distance. Once it finds a node whose outbound bandwidth to $p_x$ is sufficient for supporting the data stream, $p_x$ sends a request of *parent switching* to $p_g$, so that $p_g$ will update $p_x$’s parent in its $T_d$. Different from $T_f$, which is maintained by every on-functional-tree proxy, a separate $T_d$ needs to be maintained by every on-functional-tree service node. This means that if a single proxy offers different services in the multicast group, then it needs to keep multiple data trees.

**An Example:** Figure 8 depicts what the global functional service tree and the local data distribution tree would look like in the scenarios. In Figure 8(a), right after $P_d1$ and $P_d2$ have successfully joined the multicast group, the functional service tree kept by all on-tree service nodes and the data distribution tree at $s_2/p_4$ are shown on the right side of Figure 8(a). Subsequently, inside the subgroup (circled), the *parent switching protocol* will take place. Suppose $p_7$ is closer to $p_8$ than $p_4$,
and suppose from \( p_7 \) to \( p_8 \) there is sufficient bandwidth to support the data stream, then \( p_8 \) will ask \( p_4 \) to switch parent, after which \( p_4 \)'s data distribution tree becomes the one shown on the right side of Figure 8(b).

With data multicasting in all subgroups, it can be expected that end-to-end service paths may become longer than in pure service multicast. However, such individual performance degradations would be justified by overall network bandwidth savings.

5.3 Composite Service Maintenance

Maintenance of paths/trees is called for due to network, traffic, and group dynamics. During the lifetime of a service path/tree, the on-path or on-tree nodes and links may have varying resource conditions, or may even fail completely. In parallel, new members may join the multicast group, and old members may leave, causing the tree structure to become “distorted” and its performance to degrade over time. Therefore, for the continuous operation of the application at a good QoS level, adaptation (e.g., service multicast tree rearrangement) and failure recovery are mechanisms that need to be incorporated into the service routing framework.

5.3.1 Tree Rearrangement. The natural consequence of constructing a multicast tree incrementally is that over time, as new branches are added and existing branches are pruned, the tree structure may become sub-optimal. To maintain a good tree structure, selections of service instances have to take currently covered members into account, which means that previously selected service nodes may need to be relocated.

A centralized tree rearrangement approach for traditional multicasting has been studied in [R. Sriram, G. Manimaran and C. Siva Ram Murthy 1999]. However, a centralized approach is inappropriate in our case in which the service multicast tree is constructed and managed in distributed manners. We adopt a distributed approach by distributing the task of tree rearrangement (including performance monitoring and rearrangement itself) to all on-tree nodes. The basic idea is as follows: on-tree nodes monitor their regional performances\(^4\) and trigger local tree rearrangement if necessary. We expect that the local rearrangements together would contribute to global tree improvement.

Although theoretically tree rearrangement can be performed every time a join or leave has occurred, in practice, the disturbance caused by excessive changes may be intolerable to the ongoing multicast sessions, as packets are constantly in flight within the tree. Replacement of a node with large number of downstream members may cause large disturbance (as all downstream members may perceive some data loss). On the other hand, the change will also benefit all downstream members (i.e., utility is large). Considering tradeoffs between performance gain and disruption of services, certain threshold needs to be maintained to suppress those adaptation operations whose performance gains are not significant enough.

Let \( sn_x \) be an on-tree service node providing service \( s \), and \( sn_y \) be a candidate service node that is capable of providing \( s \) but is not on-tree. If \( sn_y \) replaces \( sn_x \),

\(^4\)We define a region in a multicast tree to be the neighborhood of an on-tree node, including its upstream and downstream nodes as well as the links connecting those.
we define the potential performance improvement as $\gamma = \frac{c(s_{nx}) - c(s_{ny})}{c(s_{nx})}$. Disturbance can be measured as packet loss rate (at fine granularity) or the number of downstream members that perceive data loss (at coarse granularity), and utility can be defined as the fraction of benefited members. We denote the disturbance caused by replacement of $s_{nx}$ as $\theta$, and the utility associated with the replacement as $\mu$. We therefore define the real benefit $\beta$ of replacing $s_{nx}$ by $s_{ny}$ as a function of performance gain, disturbance, and utility: $\beta = \gamma * \frac{\theta}{\mu}$, and replacement only takes place if $\beta$ is larger than the defined threshold.

Once a performance monitor has detected that a replacement would yield a performance improvement $\beta$ that is higher than the threshold value, the current service node $s_{nx}$ will hand over its roll to $s_{ny}$. It is important that parent and child service nodes do not perform handovers simultaneously, since they use each other as a reference point in the detection phase. To guarantee this property, before handing over, the current service node needs to synchronize with its parent and child nodes (basically blocking them from doing concurrent handovers). We better illustrate the idea with a simple example in Figure 9. The circled service node represents a performance monitor that monitors the regional performance by periodically checking if there are other candidates (found by invoking a service discovery system) with better performances that can replace itself. In the example, $p_3$ is a node also serving $s_2$. Node $p_4$ then asks $p_3$ to probe the available machine resource as well as the delay and available bandwidths between $p_3$’s potential parent and children if $p_3$ were to replace $p_4$. Suppose $p_3$ satisfy all resource requirements and also has the benefit $\beta$ improved by some percentage larger than the threshold, then $p_4$ will hand over its roll to $p_3$. While negotiations between $p_4$ and $p_3$ are going on, $p_4$’s upstream nodes ($p_1$) and downstream nodes ($p_7$ and $p_8$) should be blocked from doing the similar.

5.3.2 Failure Recovery. Traditional failure recovery mechanisms used in routing fall into two approaches: protection-based approach and restoration-based approach [Aaron Striegel and G. Manimaran 2002]. In the protection-based approach, dedicated protection mechanisms, such as backup paths, are employed to cope with failures on the primary path. In the restoration-based approach, on detection of a failure, an attempt is made to reroute the path around the faulty nodes and links. The protection-based approach has been adopted in [B. Raman and R. H. Katz 2003; Xiaohui Gu, Klara Nahrstedt 2002] for recovering single service paths. However, this approach is not suitable in multicast scenarios because of two reasons: (1) it is prohibitively expensive, in terms of resource allocations, to maintain one or more backup trees for the primary tree; (2) the dynamic membership feature causes the primary tree to change over time, thus there would be too much of overhead to keep the backup trees up-to-date. For these reasons, the restoration-based approach is more suitable for multicasting.

In this paper, we consider only hardware failures, and assume the fail-stop failure model in the sense that failures are detectable (e.g., by use of timeout). Different from traditional routing, in which failure of a node or link means only a single

---

The term $c(s_{nx})$ is defined as the sum of $s_{nx}$’s neighboring link costs ($c(s_{nx})$), which are measured as delays.

ACM Transactions on Internet Technology, Vol. ?, No. ?, to appear
Fig. 9. Tree rearrangement: (a) All on-tree nodes monitor their local performances. For instance, $p_4$ monitors its local performance and tries to compare itself with an alternative candidate service instance at $p_3$. (b) $p_4$ hands over its roll to $p_3$ because the replacement would yield better local performance.

failure on the path or tree, in service-added routing, failure of a single physical node or link may trigger failures of several spots in the service path or tree. This is so because a single network node may be contributing several services in the path or tree.

Before discussing failure recovery, we first need to devise a failure detection mechanism. While the use of heartbeat messages is a common mechanism for failure detection, there are additional challenges caused by the fact that one physical node can serve multiple (consecutive or non-consecutive) component services. Due to the dependency complexities, we need to further derive the physical node dependency graph for detection (this will be clearer as we show an example later). Each on-tree network node then periodically sends heartbeat messages to its physical parent and, if the parent does not respond within a specified time, then the current node will infer that the parent has failed. Upon the detection, the current node $n$ tries to find out its closest live ancestor, and asks it to initiate a hop-by-hop routing process towards $n$ to locate suitable instances for the failed services in between.

An example is shown in Figure 10. Figure 10(a) depicts a functional service tree together with the group members. As stated, each physical node monitors the liveness of its physical parent. The physical node dependency graph that shows the monitoring relations is shown on the right side of Figure 10(a).

Failures of some nodes (e.g., $p_2$ and $p_4$) are simpler to deal with, while failures of certain other nodes (e.g., $p_1$) yield more complex situations. For example, if $p_2$ fails
Fig. 10. Detection of failure: (a) left: a service tree plus the membership information; right: the graph that represents the monitoring relations among the nodes. (b) left: the failure of p₁ triggers failures of multiple service nodes in the tree; right: the failures are individually detected by p₂, p₃, and member 2; (c) left: independent failure reporting; right: lazy failure reporting that back-off failure reporting if possible.

(detectable by member 1), then member 1 will try to locate a live ancestor closest to itself (in this case p₁) and afterwards p₁ initiates a hop-by-hop routing process towards member 1 to recuperate service s₃. Failure of p₁ is more complex to deal with, as the node participates in multiple positions and branches of the tree. The failure itself may be detected by three nodes: p₂, p₃, and member 2, which report to their closest live ancestors - the root and the node p₃ in this case.

We discuss only recoveries initiated by the root, as this is a complex case involving parallel failures of multiple branches. While the root may recover one branch at a time simply based on the arrival time of the requests, certain overheads incurred by recovery synchronization need to be considered. If: (1) p₂’s recovery request precedes p₃’s - once p₂’s request has been satisfied, p₃’s request can be ignored (because p₃’s request is part of p₂’s request); (2) p₃’s request precedes p₂’s - p₂’s recovery request can only be initiated after p₃’s request has been satisfied, because the recovered service node (say s₁ is mapped to p₁) s₁/p₁ will serve as a reference point for part of p₃’s request. Furthermore, p₃’s recovery request will be initiated by p₁ instead of the root. From this example, we see that different recovery orders will affect the overall recovery time due to delays associated with communication and synchronization.

Optimization of the recovery ordering is hard to achieve because node R (the closest ancestor that is responsible for initiating recovery operations) is unable to...
predict the total recovery time. To overcome this problem, we employ a heuristic of minimum recovery dependencies (MRD) to try to minimize the overall recovery time. Assuming nodes report failures independently: upon receiving failure report from one node, node \( R \) is able to deduce, from the functional tree information, if other branches would be affected by the failure. In the example of Figure 10, if \( p_3 \)'s failure report arrived at the root first, the root is able to deduce that the failure also would affect \( p_2 \). Using the MRD heuristic, the root can initiate recovery action for \( p_2 \) even without receiving \( p_2 \)'s failure report, because \( p_3 \)'s request will get naturally satisfied after \( p_2 \)'s request gets satisfied (thus reducing recovery dependencies).

An alternative approach for dealing with failure reporting and recovery works as follows: since \( p_3 \) (upon detecting failures of \( p_1 \)) is able to deduce that another node - \( p_2 \) - will also eventually detect the same failure, \( p_3 \) may just adopt a lazy failure reporting mechanism by backing off its report indefinitely. By doing so, the number of total error reports will be reduced. However, this may increase the total recovery time as \( p_2 \) may only be able to detect the same node failure later. In order to follow the heuristic of minimizing recovery dependencies, only node \( p_3 \) should back off failure reporting; node \( p_2 \) should always report immediately in this case.

6. VALIDATION

We implemented our service management framework (including the service composition and maintenance solutions) as well as a minimal set of aiding components (enhanced service discovery system etc.) in the well-known network simulator ns-2. This section is devoted to performance studies of the proposed approaches.

6.1 Evaluation Methodology

Our physical Internet topologies are generated by the transit-stub model [E. Ze-gura, K. Calvert, S. Bhattacharjee 1996], by using the GT-ITM Topology Generator software. A number of physical nodes are randomly chosen as proxy nodes, whose service capability and machine capacity are assigned by certain functions. The end-to-end available bandwidth from an overlay proxy node \( p_a \) to another overlay proxy node \( p_b \) is the bottleneck bandwidth of the shortest physical path from \( p_a \) to \( p_b \). Among the physical network nodes, a small set of them (10 nodes) are chosen to be the landmark nodes - \( L \), based on which the proxies can derive their coordinates in the geometric space defined by \( L \{T. S. Eugene Ng, Hui Zhang 2002 \}. We use geometric space of 5 dimensions to represent the distance map of a network topology; calculation of geometric coordinates is done by using the software available at http://www-2.cs.cmu.edu/~eugeneng/research/gnp/. Construction of the Delaunay triangulation overlay mesh for control message purposes is aided by the Qhull software developed by the Geometry Center at University of Minnesota (http://www.geom.umn.edu/software/qhull).

6.2 Comparison of Service Unicast Approaches

In this section, we measure performances of the service unicast approaches (GLG, LHB, GLG-LHB, and LHB-GLG) described in Section 5.1. We further run a hop-by-hop approach that is based on random-walk (RANDOM), to serve as a base case to all of the approaches in study. The following performance metrics are used
Fig. 11. Comparisons of the service unicast approaches in terms of: (a) host utilization; (b) physical link utilization; (c) delay-bandwidth product; (d) service path length; and (e) gradual path finding success rate in the backtracking-off mode.

for the evaluations: host utilization, link utilization, service path length, delay \times \text{bandwidth product}, and path finding success rate.

The environment settings for the test are as follows. The physical network contains 600 nodes, and among them, 10 are selected as landmarks and 500 as proxies. We randomly generated 5000 requests (of varying lengths) between randomly selected pairs of proxies. We compare the performances under two different resource
settings: one with sufficient resources to admit all service requests, and the other with insufficient resources, in which case late join requests may get rejected because of resource scarcity.

**Sufficient-resource settings:** In sufficient-resource settings, since all service requests get successfully admitted, the performance metrics of interest are host utilization, link utilization, service path length, and delay × bandwidth product. The comparative results of several service unicast routing approaches are shown in Figure 11. As has been predicted, since GLG genuinely seeks shortest QoS-satisfied service paths, load balancing on hosts and links is poor. This is indicated by the fact that the GLG curves are steep. LHB does in fact help to keep a more balanced network and machine load, as the next service hop is the one that maximizes an aggregate function of available bandwidth and machine capacity. On the other hand, LHB performs poorly in terms of delay bandwidth product (Figure 11 (c)) and service path length (Figure 11 (d)), because service paths computed by LHB are long, and therefore demand more network resources. However, in these respects GLG performs best, because service paths computed by this approach tend to be short, and as such, require less network resources. GLG-LHB's performances are quite close to those of LHB, and LHB-GLG has good performances overall.

**Insufficient-resource settings:** After certain resources get exhausted, a join request may be denied. The performance metric of interest in such an insufficient-resources scenario is path finding success rate which, in some way, indicates how well load balancing is achieved. Figure 11(e) shows the path finding success rates of the different service unicast approaches with back-tracking turned off. As has been expected, since GLG does not take load balancing into consideration, certain resources may become exhausted more quickly than other approaches that consider load balancing, and as a consequence, path finding success rate was lowest in GLG. In the backtracking-off mode, LHB and LHB-GLG have achieved similar aggregate success rates. However, when we turn on backtracking, the aggregate success rate of LHB-GLG surpasses that of LHB by 4.9%. This is because LHB-GLG has incurred less network resource consumption.

From the above performance analyses, we see that none of the approaches performs best in all aspects. GLG's performances in terms of service path lengths and delay-bandwidth product are significantly superior to others', but is worst in path finding success rates. LHB is one of the best in finding service paths successfully, but incurs longer service paths than others and as a consequence, tends to require more network resources. From all approaches, LHB-GLG seems to have best balanced these contradictory factors, as it incurs relatively short service paths while maintaining a high path finding success rate.

### 6.3 Comparison of Service Unicast, Pure Service Multicast, and Hybrid Multicast

In this section, we study the performance benefits of employing pure service multicast and hybrid multicast. Since LHB-GLG is a service unicast approach that strikes best balance among the performance metrics, we employ LHB-GLG as the
building block for incrementally constructing a multicast tree. Simulations are run for multicast group sizes of 256, where service requests are randomly selected from a complete binary functional service tree of depth 5.

For these comparisons, we set up sufficient-resource environments. As we can see from Figure 12 (a), there is not too much difference, in terms of host utilization, between pure service multicast and hybrid multicast. This was expected because
local data multicast would not further diminish the number of service executions. Figure 12 (b) shows that hybrid multicast yields much lower link utilization than pure service multicast. Not surprisingly, the two multicast cases yield tremendous delay bandwidth product savings compared to unicast (Figure 12 (e)). Compared to service unicast, service multicast incurs longer end-to-end service paths in all cases, and hybrid multicast incurs longer paths than service multicast in most of cases (Figure 12 (c)). However, the longer end-to-end service paths in hybrid multicast are justified by lower global tree costs (Figure 12 (f)) due to service path sharing. Since we are dealing with a video streaming example, the longer end-to-end service path would affect the initial observed latency. However, such delay is only one-time as future data will arrive at a fixed rate.

6.4 Cost Analysis of Tree Rearrangement

The local tree rearrangement operations together contribute to a global tree quality improvement. As described in Section 5.3.1, we have set thresholds to suppress those tree rearrangement activities that only yield small performance gains. We therefore study the relations between threshold and global performance. For simplicity, it is assumed that the effects of $\theta$ and $\mu$ in local benefit $\beta = \gamma * \frac{\theta}{\mu}$ cancel off.

Figure 13 depicts the total tree costs (in logical units) after adaptations versus local adaptation thresholds with different service distribution probabilities. The experiment settings were as follows: similar to the settings described in Section 6.3, we used group sizes of 256, whose service requests are drawn from a pool of complete binary functional tree of depth 5. We first run the service multicast tree construction program to incrementally build a service multicast tree. After that, the local performance monitors are turned on, and local rearrangements are triggered if the potential performance improvement is larger than the local threshold values. The total tree cost is measured after all local rearrangement operations have been stabilized. At low threshold values, tree rearrangements are triggered more often, thus leading to better global performances. At higher threshold values, only those rearrangement operations that yield larger performance gains are triggered. Therefore, leading to lower global performance gain (higher final global tree cost). We can also see differences of global tree costs for different service distribution probabilities: sparser service distribution yields higher tree costs and denser service distribution yields lower tree costs overall, which are in match with our intuitions.

6.5 Comparison of Failure Recovery Schemes

We compare performances of three failure reporting and recovery approaches as described in Section 5.3.2: independent reporting and independent recovery (IRIR), independent reporting and heuristic-based recovery (IRHR), and lazy reporting and heuristic-based recovery (LRHR). The approaches are evaluated under three metrics: number of failure reports, global tree costs after recoveries, and time needed for recovery. The experiment was conducted as follows: still using multicast group size of 256 and binary functional tree as before, after the service multicast tree has

---

7Service distribution probability $x$ means that each component service is randomly distributed at $x\%$ network nodes.
been built, we randomly failed on-tree network nodes one by one. The results shown in Figure 14 are based on 3 runs, each run with 20 node failures. The values have been normalized based on the results of IRIR (base case). We can see that the lazy reporting mechanism helps to reduce the number of failure reports significantly, and the heuristic-based recovery mechanism helps to maintain better-cost service multicast trees after recoveries. Between IRHR and LRHR, there is tradeoff: while IRHR incurs better recovery time, it yields larger number of failure reports than LRHR. This is so because by having the network nodes independently reporting failures to a live ancestor, the ancestor is likely to be aware of the failure sooner, and thus can start recovery operations sooner. On the other hand, by adopting the lazy reporting mechanism (LRHR), failures are likely to be noticed later, because certain failure reports are suppressed because the detecting node assumes that other nodes will eventually detect and report the same.

ACM Transactions on Internet Technology, Vol. ?, No. ?, to appear
7. CONCLUSIONS

The component service technology has been widely advocated in the past few years, and has spawned quite a few new research problems, including service discovery, service composition/orchestration, security, and service deployment. In this paper, we have designed an integral QoS-aware service management framework that scales to large networks by means of decentralization and to one-to-many applications by means of multicasting. Such a management framework is necessary for delivering large-scale integrated and robust applications seamlessly and efficiently despite the fact that the underlying component services are widely distributed. To our knowledge, this is the most complete work in scalable service management dealing with both service composition and service maintenance. Furthermore, this management framework includes novel service multicasting and hybrid multicasting mechanisms to achieve efficient resource sharing. Our performance results show service multicast and hybrid multicast mechanisms to be effective ways to minimize resource utilizations.

8. ACKNOWLEDGMENTS

This material is based upon work supported by NSF (under Awards No. CCR-9988199 and EIA 99-72884 EQ) and NASA (under Award No. NAG2-1406). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the awarding agencies.

The authors would like to thank the anonymous reviewers for their insightful comments.

REFERENCES


Received November 2005; Accepted December 2006