QOS-AWARE SERVICE MANAGEMENT FOR INTERNET-SCALE DISTRIBUTED APPLICATIONS

BY
JINGWEN JIN
B.S., Federal University of Pernambuco, 1994
M.S., Federal University of Pernambuco, 1997

DISSERTATION
Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois
Abstract

Component-based software development has evolved from tightly coupled object-oriented style to loosely coupled service-oriented style in the past few years. The new service-oriented paradigm will eventually allow heterogeneous component-based systems to interoperate in large, open networks like the Internet, and allow applications to be dynamically aggregated in a plug-and-play manner.

The paradigm shift incurs more complex management issues, as services (both their functional and QoS aspects) should be managed not only as individuals, but also as aggregated entities based on their logical compositional structure required by the applications. The ultimate goal of service management is to ensure that an application, seen as a composition of multiple component services, is instantiated on top of a resource-sufficient network infrastructure satisfying specific QoS requirements, and runs adaptively according to runtime resource fluctuations and robustly despite runtime resource failures.

Service management must run across three stages: (1) state collection stage that deals with collection of functional and QoS states of individual services; (2) composition stage that sets up an application by properly selecting the best QoS-satisfied service instances and establishing the path; and (3) maintenance stage that takes care of resource adaptation and failure recovery at application runtime.

This dissertation focuses on providing scalable service management solutions, in which scalability is interpreted in two dimensions: network size and application size.

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 inves-
tigate two scalable approaches: (1) a hierarchical approach based on semi-global knowledge (by means of network topology clustering and QoS state clustering), with divide-and-conquer service management, and (2) a distributed approach based on local knowledge, with hop-by-hop decision at the service composition stage, local resource adaptation, and distributed failure detection, reporting, and recovery at the maintenance stage.

In terms of application size - when an application involves one-to-many group communication, we explore resource sharing, both in network bandwidth and computational resources, by proposing a special type of multicasting - service-added multicasting. Two service tree construction algorithms, Optimal-Service-Paths Tree (OSPT) and Longest-Prefix Tree (LPT), will be proposed and analyzed. Resource sharing is further maximized as we integrate traditional data multicasting into service multicasting. We call the integrated solution hybrid multicasting.

Results presented in this dissertation have been mostly implemented in *ns*-2, and extensive simulation tests have been conducted to verify their performances.
To my family
Acknowledgments

This thesis would not have been possible without the support of many people.

First and foremost, my deep gratitude goes to my thesis advisor, Professor Klara Nahrstedt, for her invaluable guidance and continuous support throughout my PhD study. She is not only a wonderful advisor in technical fields, but also a considerate person in many aspects of our personal lives. I thank her for her many insightful discussions and ideas, and her generous encouragement and support during my difficult days. I've always been amazed at Klara’s great stamina in work and unselfish dedication to the successes of her students. I feel extremely lucky for having had her as my advisor and will see her as a role model for my future career.

I would like to thank the honorable members of my thesis committee: Professors Roy Campbell, Gul Agha, and Dennis Mickunas, for their generous willingness in evaluating my work and providing invaluable feedbacks, which helped me to establish the final thesis scope.

The work presented in this dissertation was supported by numerous funding agencies: DARPA (under contract F30602-97-2-0121), NSF (under contracts CCR-9988199, EIA 99-72884 EQ, and ANI 03-23434), NASA (under contract NAG2-1406), and Capes/Brazil. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the authors and do not necessarily reflect the views of the awarding agencies.

Many university staff members have helped me during my PhD program in one way or another. I wish to thank them all. I am especially grateful to Anda Ohlsson, Andrea Whitesell, Barb Cicone, Angie Bingaman, Erna Amerman, Shirley Finke, Mary Beth Kelley, and Dana Kennedy, for their great administrative support, and to Nate Fyie, for his prompt
attendance to many of my computer maintenance problems.

For the five years that I have worked in the MONET (Multimedia Operating Systems and Networking) research group, I have received helps from many current and former group members, they are: Duangdao Wichadakul, Chui-Sian Ong, Albert Chu, King-Shan Lui, Jun Wang, Anupama Mahajan, Dongyan Xu, Li Xiao, Yuan Xue, Yi Cui, Samarth Shah, Wanghong Yuan, Kai Chen, Xiaohui Gu, Jin Liang, Wenbo He, William Conner, Won Jeon, Ying Huang, Hoang Nguyen, Zhenyu Yang, Bin Yu, Jigar Doshi, Vanish Talwar, and Raquel Hill. I thank them for their friendship and great helps.

I have received family-like friendships from numerous friends at UIUC: Xiaoai Jiang, Bin Zhou, Yuanzhi Zhou, Changjuan Chen, and Yong Fu. I thank them for their warm friendships, which especially helped me in the early years of my life in a new country. Most of them have left UIUC already, but I wish our friendship will transcend space and time.

My deepest love and appreciation go to my family - my unselfish parents, caring sister, loving husband, and lovely son who, together, made me an eternal fountain of energy and brought meanings to my existence. No thankful words are ever sufficient for me to express my deep gratitude to my parents, who unconditionally loved and educated me in all aspects of my life. My special thanks go to my always-together sister, who has been my true companion in the life and most caring sister I could ever wish. Meeting my husband during my PhD study was the most wonderful event which the random like fate arranged in my life. I thank my husband for his unique and dedicated love. Last, but definitely not least, I feel so felicitous for having had Roger, my son, the little creature who brought so much happiness to the entire family. I feel extremely fortunate for having such a wonderful family - my ultimate resort and source of everything. This thesis is dedicated to them.
# Table of Contents

List of Figures .......................................................... xi

Chapter 1  Introduction ................................................. 1
  1.1 The Service-Oriented-Architecture Model ...................... 2
  1.2 Motivating Application Scenarios ............................... 4
  1.3 Research Goals and Challenges .................................. 7
  1.4 Solution Overview ................................................ 9
  1.5 Dissertation Structure ........................................... 11

Chapter 2  Background, Related Work, and Problem Formulation .... 13
  2.1 Background ....................................................... 13
  2.1.1 Application Environment ..................................... 13
  2.1.2 Single Service Model ......................................... 14
  2.1.3 Compositional Service Model ............................... 14
  2.1.4 Background Components of the Management Framework Architecture ............................................ 16
  2.1.5 Notations and Terminology .................................. 17
  2.2 Related Work ..................................................... 18
  2.2.1 Service Composition ......................................... 18
  2.2.2 Multicast and Routing in Overlay Networks ............... 19
  2.3 Problem Formulation ............................................. 20
  2.4 Symbols ......................................................... 22

Chapter 3  Small-Scale Problems and Solutions ....................... 25
  3.1 Assumptions ..................................................... 26
  3.2 Related Work .................................................... 27
  3.3 Performance Metrics ............................................ 29
  3.3.1 Individual Performance Metrics ............................ 29
  3.3.2 Aggregated Performance Metric - $\mathcal{F}$ ................ 32
  3.4 State Information Maintenance ................................. 33
  3.5 QoS Service Path Computation .................................. 33
  3.5.1 Mapping ....................................................... 34
  3.5.2 Routing Computation .......................................... 36
  3.5.3 Complexity Analysis .......................................... 38
  3.6 Performance Evaluation ......................................... 39
  3.6.1 Performances of $\mathcal{F}$ .................................. 39
3.6.2 Integrated Approach vs Two-Phase Approach ................. 41
3.7 Conclusions .................................................................... 42

Chapter 4 Scalability in Network Size Aspect - Hierarchical Approach . 43
4.1 Challenges .................................................................... 43
4.2 Solution A ..................................................................... 45
  4.2.1 Hierarchical Topology Construction ......................... 47
  4.2.2 Service Routing Information Aggregation and Distribution ... 51
  4.2.3 Hierarchical Service Path Computation ..................... 52
  4.2.4 Performance Studies ............................................. 58
4.3 Solution B ..................................................................... 62
  4.3.1 Hierarchical Topology Construction ......................... 62
  4.3.2 Routing State Aggregation and Distribution ................ 68
  4.3.3 Hierarchical Service Path Computation ..................... 74
4.4 Conclusions .................................................................... 75

Chapter 5 Scalability in Network Size Aspect - Distributed Approach . 76
5.1 Foundations .................................................................... 76
  5.1.1 Service Discovery with Geometric Location Awareness .... 76
  5.1.2 Topology Setup and Routing State Obtainment .......... 77
  5.1.3 Routing Approach .................................................. 77
5.2 Hop-by-Hop Service Path Computation .............................. 78
  5.2.1 Local-Heuristics-Based (LHB) Approach ................... 78
  5.2.2 LHB Enhanced with Geometric Location Guidance .......... 79
  5.2.3 Routing Backtracking ............................................ 82
5.3 Performance Study ....................................................... 82
  5.3.1 Evaluation Methodology .......................................... 82
  5.3.2 Performances of Different Service Path Finding Approaches . 84
5.4 Conclusions .................................................................... 87

Chapter 6 Scalability in Application Size Aspect ......................... 88
6.1 Service Multicast Tree Construction Algorithms ................. 89
  6.1.1 Algorithms ......................................................... 89
  6.1.2 Performance Evaluation ......................................... 91
6.2 Distributed Construction of Service Multicast Trees ............. 93
  6.2.1 Design Overview .................................................. 93
  6.2.2 Incremental Service Tree Construction ..................... 95
  6.2.3 Hybrid Multicast .................................................. 97
  6.2.4 Performance Study ................................................ 100
6.3 Conclusions .................................................................... 103

Chapter 7 Maintenance ......................................................... 104
7.1 Tree Rearrangement ..................................................... 104
7.2 Failure Recovery ......................................................... 106
7.3 Performance Study ..................................................... 110
## List of Figures

1.1 The conceptual model of Service Oriented Architecture. ........................................... 3  
1.2 Web Service Architecture stack. .................................................................................. 3  
1.3 Three scenarios that make use of composite services to achieve dynamic content customization: (a) a one-to-one scenario; (b) a one-to-many Web-based multimedia application scenario; (c) a one-to-many military scenario. ........ 5  
2.1 A non-linear service graph (SG) ................................................................................... 15  
2.2 (a) a non-linear service request; (b) a linear service request. ................................. 15  
2.3 A service request tree consisting of the merge of multiple linear service requests. 15  
2.4 The service management substrate resides between the application layer and the service discovery/QoS compilation layer to make the component services infrastructure transparent to the application layer. ......................................................... 17  
3.1 (a) A mesh proxy/service topology; (b) a fully connected proxy/service topology. 27  
3.2 (a) Two service configurations - $c_1$ and $c_2$; (b) Part of a proxy network composed of $p_1$ and $p_2$. ........................................................ 28  
3.3 (a) A genuine widest path; (b) a normalized widest path. ........................................ 31  
3.4 Mapping process. ........................................................................................................ 35  
3.5 The back-tracking resource checking process in each relaxation. ............................. 38  
3.6 Performance values of $\mathcal{F}$ compared with those of the best and random cases.  40  
3.7 (a) Comparison of performance values of $\mathcal{F}$ between the integrated approach and the separate approach. (b) Success rates of integrated approach vs success rates of two-phase approach ......................................................... 42  
4.1 An example of $HFC_4$ topology: nodes within each cluster are fully connected, and clusters are fully connected by their border nodes (in shadow). ................. 46  
4.2 (a) A set of $N_p$ nodes in space $S$; (2) an MST connecting the nodes (inconsistent edges are marked with $X$); (3) clusters detected by removing the inconsistent edges in (b). ................................................................. 50  
4.3 Information learned by $C_{2.1}$ from $p$. .................................................................... 51  
4.4 Hierarchical routing at high level: (a) A single node - $p_d$ - first computes an inter-cluster service path connecting certain border nodes; (b) certain nodes within each cluster individually compute intra-cluster service paths and return their answers to $p_d$ to form the final service path. ......................................................... 53  
4.5 The service topology (network topology + services) of the example. ................. 54
4.6 Inter-cluster service routing (steps performed by the destination proxy $C_{21}$ of the service request) .................................................. 55

4.7 Computations of child service paths. Shortest paths are indicated with bold lines. ................................................................. 58

4.8 (a) Number of coordinates-related node-states kept at a single proxy; (b) number of service-related node-states maintained at a single proxy. .... 59

4.9 A comparison of service path lengths. ........................................... 62

4.10 Hypothetical and real $\eta$ for topologies of different sizes. ................. 64

4.11 Two feasible topologies (a) topology $a$ - HFC$_A$; (b) topology $b$ - HFC$_B$. 66

4.12 Two thresholds for preventing topology instability.......................... 68

4.13 Performance comparisons of 3 approaches: (1) passive approach; (2) active approach with global re-clustering; (3) active approach with local re-clustering operations. .................................................. 69

4.14 (a) Sample resource plane; (b) organized resource plane: shaded area represents QoS-weak area; service instances that fall beyond the shaded area are clustered. ...................................................... 72

4.15 Bloom filter technique. ............................................................. 73

4.16 An example of cluster-level service dag and information on how to label the link values. ......................................................... 74

4.17 Sample resource conditions of service types in different clusters. ........ 75

5.1 (a) Based on $LHB$, $p_s$ may choose a next hop that is not on the way to destination; (b) For making the routing aware of the hosts’ geometric locations, $p_s$ chooses a next hop that is on the way to destination. ................. 80

5.2 Finding a QoS-satisfied and potentially shortest service path hop-by-hop from $p_s$ to $p_d$ that satisfies the service graph $s_1 \rightarrow s_2 \rightarrow s_3$. .................................................. 81

5.3 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. 85

6.1 (a) $OSPT$: (a.1) optimal service path for $SR_1 = \langle p_0, s_1, s_2, s_5, p_5 \rangle$; (a.2) optimal service path for $SR_2 = \langle p_0, s_1, s_2, s_3, s_4, p_3 \rangle$; (a.3) merging the two service paths to form an $OSPT$; (b) $LPT$: (b.1) optimal service path for $SR_1 = \langle p_0, s_1, s_2, s_5, p_5 \rangle$; (b.2) optimal service path for $SR_{suffix} = \langle p_5, s_3, s_4, p_3 \rangle$; (b.3) attaching (b.2) to (b.1) at the graftable node of longest prefix $p_5$ to form an $LPT$; (b.4) optimal service path for $SR_{suffix} = \langle p_1, s_2, s_3, s_4, p_3 \rangle$; (b.5) since $p_5$ does not have enough bandwidth in its outgoing link, attach (b.4) to (b.1) at the graftable node of second longest prefix $p_1$ to form an $LPT'$. .... 90

6.2 (a) Total service path/tree costs of the algorithms in terms of delay; (b) Average end-to-end service path delay incurred by each algorithm. .............. 92

6.3 (a) Proxy machine resource consumption; (b) Bandwidth consumption on physical network links (for clarity purposes, those links whose load is zero are not shown). .................................................. 92

xii
6.4 (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 $p_4$. 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 $p_{d2}$.

6.5 (a) Pure service multicasting; (b) hybrid multicasting (service multicasting + data multicasting).

6.6 Exploring data multicast in a service multicast scenario: (a) a new service branch's first node, $p_8$, is initially directly attached to the graftable service node $p_4$ ($p_4$ as $p_8$' parent in the local data distribution tree); (b) $p_8$ gets parent-switched to $p_7$ in the data distribution tree.

6.7 Comparisons of: (a) host utilization; (b) physical link utilization; (c) end-to-end service path length; (d) global host utilization; (e) delay bandwidth product and; (f) global service path/tree cost among the different delivery modes: service unicast, pure service multicast, and hybrid multicast.

7.1 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.

7.2 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 $p_1$ triggers failures of multiple service nodes in the tree; right: The failures are individually detected by $p_2$, $p_3$, and member 2; (c) left: independent failure reporting; right: lazy failure reporting that back-off failure reporting if possible.

7.3 Total tree cost after adaptations vs local adaptation threshold with different service distribution probabilities.

7.4 Performance comparisons among IRIR, IRHR, and LRHR: metric 1 is the number of failure reports generated, metric 2 is the global service tree cost after the recovery completes, and metric 3 is the time needed for the recovery to complete. The numerical values have been normalized based on the results of IRIR (base case).
Chapter 1

Introduction

Over the last four decades, software development has gone through several programming models, each attempting to deal with increasing levels of software complexity. Higher and higher software modularity and reusability have been achieved as software development evolved from structured programming to procedural programming and to object-oriented programming.

To allow independently developed objects (expressed by interfaces) to cooperate, component-based models, such as CORBA [1], Java RMI [2], and DCOM [3], defining standard forms and interfaces between components, have been developed in the decade of 90’s. However, CORBA, Java RMI and DCOM components exhibit tight coupling features: objects are heavily dependent on each others’ interfaces. Moreover, invoking a remote procedure or method (through RPC or RMI) requires the caller to have some awareness of the called method (e.g., the exact signature of the method). These features make the component models suitable in a closed system, but hard for arbitrary systems to interoperate.

In the recent few years, the component-based object-oriented model has been further evolved into the Service-Oriented Architecture (SOA) [4], achieving looser coupling among software components and thus higher interoperability. The consequence of this paradigm shift is that more heterogeneous systems will be allowed to interoperate and it is expected that large deployment of component services will occur in the Internet scale. Future applications, which can be seen as compositions of many component services, may run on top of widely distributed Internet nodes.

SOA is the natural evolution in software development, enabling a higher level of abstrac-
tion, which has made software development more of a management task than a development task\(^1\). The goal of this dissertation is to study the service management issues incurred by this new component service model.

In the following, we will first present the basic SOA model, followed by some descriptions on our research motivation together with some interesting application scenarios that will benefit from the SOA model. We then outline our research goals and challenges as well as our solutions.

1.1 The Service-Oriented-Architecture Model

By definition from Wikipedia \([4]\), 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.

The main differences between a service in SOA and an object in CORBA, Java RMI or DCOM lie in two aspects. First, a service is ideally a self-contained, stateless software entity that can function independently, while an object may depend on the interfaces of many other interacting objects. Second, a service, which may comprise many objects running on a single machine, has a coarse-grained interface, while an object has a fine-grained 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 collect their meta-data (including functional specifications and QoS specifications) for future service requests issued by service consumers. A found service is bound between the service provider and the service consumer via a service contract and subsequently executed. This “find, bind,

\(^1\)Actually the concept of SOA has been developing for more than a decade, ever since CORBA extended the promise of integrating applications on disparate heterogeneous platforms. Object Management Group (OMG) has also developed CORBA bindings for WSDL to allow existing CORBA applications to continue to work in a web services world \([5]\).
and execute” paradigm not only provides transparency in service locations, but also enables late binding. Figure 1.1 depicts the conceptual model of the service-oriented architecture.

An instantiation of SOA is the Web Service Architecture [6], which defines a series of standards for messaging, service description, and service discovery. The World Wide Web Consortium has defined the stack of web service architecture, which involves many layered and interrelated technologies, including XML, SOAP, WSDL, and UDDI, as shown in Figure 1.2 [6]. The vertical management component and the top layer in the stack are where the focus of this dissertation lies. However, we will not rely on any specific technologies, rather, our discussions will be revolved around the more generic service-oriented architecture model.
1.2 Motivating Application 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 requiring data of different formats, there is a strong need to perform protocol and content translations between communicating parties to bridge the gaps.

Value-added, transformational services have been proposed for such purposes [7, 8], and industrial companies such as Akamai [9] and IBM [7, 10] have spent efforts on creating transcoding services between sources and destinations to seamlessly bridge the heterogeneities. However, the diversity involved in the Internet makes it hard to provide a uniform and consistent usage experience to users. Furthermore, 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 based on end-to-end needs [11, 12, 13, 14]. We depict three interesting and useful Internet applications below.

- Figure 1.3(a) depicts a one-to-one scenario where a cell phone user who understands only English wants to learn the summary of a document written in Latin. The document can first be translated from Latin to English, then summarized, and finally converted to speech.

- Figure 1.3(b) depicts a one-to-many Web-based multimedia application where news videos from a CNN or Yahoo server get customized (e.g., transoded, filtered) ac-
Figure 1.3: Three scenarios that make use of composite services to achieve dynamic content customization: (a) a one-to-one scenario; (b) a one-to-many Web-based multimedia application scenario; (c) a one-to-many military scenario.
cording 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.

- Figure 1.3(c) depicts a military scenario whose audience may be war commanders (including their secretaries), soldiers, or journalists. The information source is content of videos captured from multiple cameras. An end user may ask to customize the source video according to its own needs or interests. For example, a general seeing a scenario may be more interested in learning the munition conditions or the overall aspects of the battlefields. Since there are too many battlefields to watch, he may be more interested in getting their statistical reports along with the video. On the other hand, a journalist interested in one of the scenes may choose to process that particular content in order to write a story. He may choose to have the source video go through image recognition to recognize the soldiers and then retrieve the soldiers’ personal information from the military database, and so forth. Simultaneously, multiple end users may ask to retrieve and process the same video content as needed.

From the scenarios depicted in Figure 1.3, 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. The SOA model can be adopted in other applications such as business applications (e.g., travel planning) as well. This dissertation targets to solve service management issues for those applications (e.g., multimedia applications) that have high demands on such resources as computational power and network bandwidth. However, although our discussions are revolved around multimedia streaming applications, the solutions developed in the dissertation apply to general component-service-based applications requiring QoS treatment.
1.3 Research Goals and Challenges

From the perspective of an application, it is fundamental that the component service infrastructure appear totally transparent. This requires a service management layer, between the component service infrastructure and the applications, that can take the charge of monitoring services (both their functional and QoS aspects) not only as individuals, but more importantly as aggregated entities based on the logical compositional structure required by the application. Overall, the purpose of the management layer is to ensure that the application as a whole runs efficiently and robustly, with the following features:

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

- **Scalability:** While management can be done centrally in small-scale networks and on an end-to-end basis, with the potential of SOA, it is desirable to design solutions that scale to large networks and to large group-based applications to cater to future demands.

- **Adaptivity:** During the lifetime of a service path, resource conditions may fluctuate. Properly reflecting resource fluctuations in the existing paths so that QoS is still satisfied or optimality is maintained, is vital.
• **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 recover from failures seamlessly.

All these features require the management entity to execute tasks in three different stages: a pre-stage (status collection), a path computation stage (composition), and a runtime stage (maintenance).

• **Status collection:** Before component services can be composed, the management entity needs to know their related states, including the functional specifications and QoS states (such as failure rate, availability, monetary cost, available resources on the machine running the service, and available incoming and outgoing bandwidths).

• **Composition:** At a high level, composition means establishment of a QoS-satisfied service path based on certain QoS requirements. For instance, an application, with a specific service structural need, may require that all related service instances run on top of machines with sufficient machine resources including memory space and CPU cycles, and that the end-to-end service path contain sufficient bandwidth and be delay-optimal (or shortest).

• **Maintenance:** Once a service path has been established, how to maintain it for the whole application life time, so that despite resource fluctuations and failures, the application runs robustly, is subject of the maintenance component.

With the potential of SOA being widely deployed in large networks, all of the above tasks must be done scalably, with scalability meaning two dimensions: network size and application size. The tasks can be broken down into many lower-level concrete challenges, including network topology formation, state distribution, service path computation, performance monitoring, failure detection and recovery, as we will see later in this dissertation.
1.4 Solution Overview

We start our investigation of the service management problems and solutions for one-to-one applications in small service networks, in which we focus on understanding the relevant QoS metrics, the challenges, and the solutions (including topology formation, state distribution, QoS service path computation). We then proceed to devise more scalable solutions.

As mentioned, scalability is perceived in two dimensions: network size and application size.

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 investigate two scalable approaches: hierarchical and distributed, which are two rules-of-thumb approaches adopted in networking and distributed systems when scalability becomes the concern.

• hierarchical approach: The idea is to let network nodes maintain semi-global knowledge of the system, by means of network topology clustering and QoS state clustering. Challenges pertinent to this approach include: how to identify clusters with application-level measurements so that the clustered network is congruent with the physical network; how to deal with continuous network node joins and leaves, so that global clustering optimality is maintained; how to properly aggregate and distribute service and QoS states so that nodes in the system maintain adequate amount of knowledge for service path computation; how do nodes compute a concrete service path with only semi-global knowledge. Our solution to these challenges can be summarized as follows. We use Internet distance information (measured as end-to-end delay) as our metric for clustering end-hosts. In order to efficiently obtain the distance map of a large network, we use the Global Network Positioning (GNP) mechanism [15] - so that both measurement and storage complexities can be significantly reduced. With our clustering objective well defined, we then adopt an appropriate clustering algorithm
to identify the clusters. Dynamic membership is efficiently supported in our system by allowing new nodes to join their nearest clusters and old nodes to leave as needed, followed by some low-overhead local reclustering operations when necessary. QoS state aggregation becomes a major problem when multiple QoS metrics are considered. We use two techniques - clustering and bloom filter - to effectively reduce the amount of data to be advertised among clusters. With only a semi-global view of the system, we devise a divide-and-conquer top-down approach for nodes to jointly compute service paths.

- **distributed approach:** In this approach, nodes only maintain local knowledge (each node may know the state of a very limited set of neighbors), and service paths are computed in a hop-by-hop manner. However, a hop-by-hop approach usually suffers from myopic effects, as local-based heuristics do not attempt to account for optimization of additive metrics such as end-to-end path length. Our solution is to incorporate the easily maintainable geometric location information by which the hop-by-hop approach can be guided to achieve good end-to-end delay performance.

**In terms of application size** - when the application involves one-to-many group communication, we explore resource sharing (in terms of network bandwidth and computational resources) by proposing two types of multicasting:

- **service-added multicasting:** Different from the traditional homogeneous data multicasting, in which all on-tree nodes simply act as relays and all tree links carry the same data, the service-added multicasting that we propose in this dissertation refers to a heterogeneous type of multicasting, in which nodes may have different functionalities - transforming the data from input to output - and links may carry heterogeneous data. We design and study two service multicast tree construction algorithms - Optimal-Service-Paths Tree (OSPT) and Longest-Prefix Tree (LPT).
• *hybrid multicasting*: Resource sharing can be further maximized with the integration of data multicasting into service-added multicasting. We call this a hybrid multicasting solution, and study the protocol issues needed for realizing hybrid multicasting on top of our service-added multicasting solution.

Based on our distributed solution for one-to-many group applications, we investigate challenges and solutions for maintaining component-service-based applications. Decentralization in service composition implies that our adaptation and failure recovery mechanisms must be decentralized accordingly. We let on-tree nodes monitor their local performances, and trigger local adaptation operations when needed. It is expected that the local adaptations together will improve the overall service tree quality. Failure recovery in service-added multicasting is made more difficult than it is in traditional multicast, because in service-added multicasting, a single network node failure may trigger multiple logical service failures possibly spanning multiple tree branches. We aim to devise distributed failure detection, reporting, and recovery mechanisms to provide robustness in service management.

1.5 Dissertation Structure

The dissertation will have the following structure.

• We review some general background and related work, as well as formulate our problems in Chapter 2. Tables listing symbols, notational conventions, and acronyms will be also presented in this Chapter.

• Chapter 3 describes the challenges and solutions for performing service composition in small networks.

• We divide issues of scalability in network size aspect into two chapters: Chapter 4 for our scalable solution based on the hierarchical approach, and Chapter 5 for our scalable solution based on the distributed approach. Issues of scalability in application size
aspect will be presented in Chapter 6, where we introduce our multicasting solutions. With the composition issues solved, we then proceed to study the maintenance issues in Chapter 7. Concentrating on one-to-many group-based applications in large networks, we devise local adaptation mechanism for service multicast trees constructed by the distributed, hop-by-hop approach. In addition, we derive distributed failure detection, reporting, and recovery mechanisms for achieving transparent robustness at the application level.

The concepts proposed in this dissertation are mostly implemented in ns-2 and extensive simulation tests are conducted. Performance results of each approach are properly placed in the related chapters.

- We conclude the dissertation with an outline of contributions and some future work in Chapter 8.
Chapter 2

Background, Related Work, and Problem Formulation

In this chapter, we introduce some general background knowledge (Section 2.1) and related work (Section 2.2). Background knowledge and related work that are specific to application environments will be presented in later chapters, when the subjects are appropriately discussed. Section 2.3 gives a formal formulation of the problems. Tables listing the symbols, notational conventions, and acronyms to be used in this dissertation will be presented in Section 2.4.

2.1 Background

2.1.1 Application Environment

Although SOA has been intensively investigated in the past few years, real deployment of SOA-based applications is still in its infancy. While future deployment of component services can take place in any type of networks (e.g., university networks and commercial ISP networks), we generalize and call such a network proxy network, and assume that it is under the same administrative domain, or if the proxy network comprises multiple smaller proxy networks, we assume they are fully cooperative. Such a proxy network is composed of end hosts, each installed with a set of component services, forming an overlay network.
2.1.2 Single Service Model

An individual 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 [16]. 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.

2.1.3 Compositional Service Model

When two services, $s_i$ and $s_j$, are to be composed, then the output quality of $s_i$ should equal the input quality of $s_j$. For instance, if two transcoders, MPEG2JPEG and JPEG2H261, are to be composed, then the output quality (e.g., frame rate, image solution, window size) of MPEG2JPEG must be equal to the input quality of JPEG2H261. The notation “$s_i \rightarrow s_j$” will be used to indicate that service $s_i$ is followed by service $s_j$.

Currently, QoS compilers such as [17] are already able to translate a user’s request (e.g., get secure Video-on-Demand service with viewing quality of 30fps and in format of JPEG) into a linear or non-linear service graph - SG, where a path that leads from any source service to any sink service is a viable configuration satisfying end-to-end service requirements. In such a service graph, all services have their input qualities and output qualities defined, according to user’s requirement. These application-level quality parameters will be further mapped by the QoS compiler into concrete resource requirements (e.g., CPU, memory, and network bandwidth). Hereafter we will only consider service graphs at resource level.

Figure 2.1 shows an example of non-linear SG (together with the services’ resource requirements) that has three viable configurations: $s_1 \rightarrow s_2 \rightarrow s_4$, $s_1 \rightarrow s_3 \rightarrow s_4$, and $s_1 \rightarrow s_3 \rightarrow s_5$. A service request $SR$ consists of a service graph (SG), and a pair of source proxy $p_s$ and destination proxy $p_d$. Figure 2.2 shows two examples of one-to-one service request, in which there is exactly one source and one destination. Figure 2.3 depicts a one-
Figure 2.1: A non-linear service graph (SG)

Figure 2.2: (a) a non-linear service request; (b) a linear service request.

to-many service request tree, in which there is one source, but multiple destinations. Such a service request tree may represent a merge of multiple linear service requests made out by a group of application users.

Figure 2.3: A service request tree consisting of the merge of multiple linear service requests.
2.1.4 Background Components of the Management Framework Architecture

Service management helps to achieve maximum transparency to users at the application layer; users should be unaware of the service component infrastructure as well as all or most negotiations related to solving the Internet heterogeneity problem. This can be achieved by delegating a user proxy to act on behalf of the user. Assuming the scenarios depicted in Figure 1.3, an end user may first contact a nearby proxy, which we call the dealer and assume it is installed with a QoS compiler that can 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 dealer 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 dealer has about the network, it can initiate service path computation. Below we depict the roles in more detail.

- **QoS compilation**: QoS compilation [18] 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, all resource requirements (e.g., in terms of CPU, memory, network bandwidth) are defined (Figure 2.1).

- **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. 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 [19, 20].
Figure 2.4: The *service management* substrate resides between the application layer and the service discovery/QoS compilation layer to make the component services infrastructure transparent to the application layer.

- *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 service graph, there needs to be a process, which we call *service management*, 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 2.4). Moreover, the management substrate must deal with resource fluctuations and failures. This management substrate will be the focus of our study.

### 2.1.5 Notations and Terminology

We denote the functional part of a service graph as $SG = (s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots)$ and a service request as $SR = (p, s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots, p_d)$. The request is for finding a service path between the source $p$, and the destination $p_d$ containing $s_1$, $s_2$, and $s_3 \ldots$, in sequence. A concrete *service path* $SP$ 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 $sn_x$ to be an instance of $sn_x$’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.

### 2.2 Related Work

#### 2.2.1 Service Composition

As presented, the component service model actually stemmed from the modularity concept in object-oriented languages, and has gained tremendous amount of research attention in the recent years for dynamic creation of complex Internet services from primitive components [11, 12, 13, 21, 22].

The Ninja project [11] is probably (one of) the first to propose using component services in the Internet to achieve scalability, robustness, and adaptivity of services. Among other elements of the Ninja architecture, the notion of logical and physical service paths was introduced, although no concrete work in how to compute them was done. Logical service paths correspond to service graphs in our terminology, specifying what services are to be composed and in what dependencies. A logical service path (service graph) is network-independent, while a physical service path, representing a mapping of a logical service path onto a service network, should specify a concrete network node per each service in the logical service path to mean that the service will get executed on the specified network node.

Other known projects that describe service composition architectures include: CANS
from NYU [23], SAHARA from Berkeley [22], and SWORD from Stanford [13]. However, none of them seemed to have paid proper research attention in the service composition problem.

Initial work in service path finding can be found in [24] at the overlay network layer, and [25] at the physical network layer. The former focused on resources (by assuming that all network nodes contain the same service set), and the latter only considered the path’s functional correctness issue. Both are suited for small-scale networks, as they use centralized approaches. Works in [26, 27] describe centralized approaches for computing performance constrained service paths, and are also targeted at small networks only. [28] describes a distributed approach for computing resource-sufficient service paths by having each service hop individually selecting its next service hop (which is, among all service neighbors, the one that contains the maximum amount of resources). All of the above works are for single-path findings. In [29], the authors classified service composition problems based on their numbers of sources and destinations, and studied three of them for small networks: Single-Source Single-Destination (SSSD), Multiple-Source Single-Destination (MSSD), and Multiple-Source Multiple-Destination (MSMD). Heuristics-based algorithms have been derived and evaluated.

### 2.2.2 Multicast and Routing in Overlay Networks

Multicast was initially proposed for the IP layer [30] with the purpose of reducing packet redundancies in the one-to-many delivery model. Since IP-layer multicast calls for changes at the network infrastructural level, its deployment has been greatly hindered. Around year 2000, the research community started to propose the idea of lifting multicast related operations from the IP layer to the application layer [31, 32, 33], and has since triggered tremendous amount of research interest in application-level routing and overlay networks in general [34, 35, 36]. Those works concentrated on data multicasting, in which all tree nodes function as relays and all tree links carry the same data.
Overlay network routing can be performed on top of either structured topologies [31, 37] or unstructured topologies [32, 33]. The former approach configures the overlay network into a partial mesh topology, so that the same routing protocols designed for the IP layer, such as OSPF (Open Shortest Path First) and M OSPF (Multicast Open Shortest Path First), can be directly employed at the overlay layer. In the latter approach, hosts are considered fully connected (since at the overlay layer, as long as the underlying physical network does not partition, all nodes are virtually fully connected), and for each application, a special topology (e.g., a multicast tree) is built and maintained.

2.3 Problem Formulation

As stated, service management essentially consists of computation and maintenance of optimal QoS-satisfied service paths, and can be seen as a special (service-added) QoS routing problem. Throughout the discussions in this dissertation, we will use the terminologies service management and service-added routing alternately. Formally, we define the problem as follows.

Given a weighted, connected network graph \( NG = ((V_{NG}, E_{NG}), (a_s, a_v, a_e), (c_v, c_e)) \), representing a service network, where:

1. \( V_{NG} \) is the set of vertices in \( NG \) representing network nodes (proxies); \( E_{NG} \subseteq V_{NG} \times V_{NG} \) is the set of edges in \( NG \) representing links among network nodes;

2. \( a_s \) is a service function which maps each node \( v \in V_{NG} \) to a set of component services \( a_s(v) \) representing service set available on node \( v \); \( a_v \) is a weight function which maps each node \( v \in V_{NG} \) to a non-negative number \( a_v(v) \) representing the amount of available resources on node \( v \); \( a_e \) is a weight function which maps each edge \( e \in E_{NG} \) to a non-negative number \( a_e(e) \) representing the amount of available resources (available bandwidth) on edge \( e \); \( \forall u \in V_{NG}, \forall v \in V_{NG}, if (u,v) \notin E_{NG}, \) then \( a_e(u,v) \) repre-
sents the bottleneck available resources on links along the shortest path from \( u \) to \( v \) (bottleneck bandwidth).

3. \( c_v \) is a weight function which maps each node \( v \in V_{NG} \) to a non-negative number \( c_v(v) \) representing the cost associated with using the node; \( c_e \) is a weight function which maps each edge \( e \in E_{NG} \) to a non-negative number \( c_e(e) \) representing the cost associated with using the edge; \( \forall u \in V_{NG}, \forall v \in V_{NG}, \text{if } (u, v) \notin E_{NG}, \text{then } c_e(u, v) \) is the sum of the costs of those links along the shortest path from \( u \) to \( v \).

Let \( SG = (V_{SG}, E_{SG}) \) denote a service graph in which \( V_{SG} = \{s_1, s_2, \ldots, s_k\} \) and \( E_{SG} = \{(s_1 \rightarrow s_2), \ldots, (s_j \rightarrow s_k)\} \). Then a one-to-one service request can be denoted as \( SR_{one2one} = (SG, (p_s, p_d)) \), in which \( p_s \) and \( p_d \) are the source and the destination bounding the service graph. Given a weighted tree graph \( RG = ((V_{RG}, E_{RG}), (r_s, r_l), (g_s, g_l)) \) representing a service request graph (similar to that shown in Figure 2.3), where:

1. \( V_{RG} = V^E_{RG} + V^I_{RG} \), \( V^E_{RG} \subseteq V_{NG} \) is the set of external nodes (root and leaves) in \( V_{RG} \) specifying the source and the destinations of service requests, and \( V^I_{RG} \) is the set of internal nodes specifying required services between the root (source) and the leaves (destinations); \( E_{RG} \subseteq V_{RG} \times V_{RG} \) is the set of edges;

2. \( r_s \) is a weight function which maps each node \( s \in V^I_{RG} \) to a non-negative number \( r_s(s) \) representing the amount of resource required to execute service \( s \); \( r_l \) is a weight function which maps each link \( l \in E_{RG} \) to a non-negative number \( r_l(l) \) representing the amount of resource required on link \( l \);

3. \( g_s \) is a weight function, unknown at the time being, which maps each service \( s \in V^I_{RG} \) to a non-negative number \( g_s(s) \) representing the cost required to execute service \( s \); \( g_l \) is a weight function, also unknown at the time being, which maps each link \( l \in E_{RG} \) to a non-negative number \( g_l(l) \) representing the cost incurred by using link \( l \).
The problem is to find a mapping function \( F_m : V_{RG} \rightarrow V_{NG} \), mapping each node in \( V_{RG} \) to a network node in \( V_{NG} \) as follows: (1) \( \forall w \in V^E_{RG}, m(w) = w \); (2) \( \forall s \in V^I_{RG}, \) if \( m(s) = v \) (where \( v \in V_{NG} \)), then \( s \in a_s(v) \) and \( r_s(s) \leq a_v(v) \). In addition, \( \forall (u, v) \in E_{RG} \), if \( m(u) = u' \) and \( m(v) = v' \), then \( r_{l}(u, v) \leq a_{e}(u', v') \).

After mapping, the cost of each link in \( RG \) becomes labeled as the cost of the corresponding edge in \( NG \); i.e., \( \forall (u, v) \in E_{RG} \), if \( m(u) = u' \), and \( m(v) = v' \), then \( g_{l}(u, v) = c_{e}(u', v') \). Also, the cost of each node \( s \) in \( V^I_{RG} \) becomes labeled as that cost of using the network node on which the service is mapped; i.e., \( \forall s \in V^I_{RG}, \) if \( m(s) = v \), then \( g_{s}(s) = c_{e}(v) \). The optimization problem is: find \( F_m \) such that the resulting sum of the costs \( \sum_{l \in E_{RG}} g_{l}(l) + \sum_{s \in V^I_{RG}} g_{s}(s) \) is minimized. The overall problem is a resource-constrained optimization problem.

When the request tree graph only has one leaf, then the problem is a one-to-one (or end-to-end) service composition problem. In [29], this case was called SSSD (Single Source, Single Destination). By reducing the problem of bin packing (which is known to be NP complete) to the SSSD service composition problem, it can be proven that SSSD (or one-to-one) service composition problem is NP complete. Since the one-to-one service composition problem is a special case of the one-to-many service composition problem, it further implies that the one-to-many service composition problem is NP complete.

### 2.4 Symbols

Table 2.1 lists most of the symbols and notational conventions used and to be used throughout this dissertation, and Table 2.2 lists the acronyms.
Table 2.1: Symbols and notational conventions used in this dissertation.

<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>a service component $i$</td>
</tr>
<tr>
<td>$p_\alpha$</td>
<td>a proxy node $\alpha$</td>
</tr>
<tr>
<td>$sn_x = s_i/p_\theta$</td>
<td>a service node $x$ (representing mapping of service $s_i$ onto proxy $p_\alpha$)</td>
</tr>
<tr>
<td>$\text{service}(sn_x) = s_i$</td>
<td>function $\text{service}$ (represents the service part of a service node)</td>
</tr>
<tr>
<td>$\text{proxy}(sn_x) = p_\theta$</td>
<td>function $\text{proxy}$ (represents the proxy part of a service node)</td>
</tr>
<tr>
<td>$Q_{in}, Q_{out}$</td>
<td>input and output QoS</td>
</tr>
<tr>
<td>$R_s : Q_{in} \times Q_{out} \rightarrow R$</td>
<td>resource usage function of service $s$</td>
</tr>
<tr>
<td>$s_i \rightarrow s_j$</td>
<td>dependency relation between $s_i$ and $s_j$: $s_i$ is followed by $s_j$</td>
</tr>
<tr>
<td>$SG$</td>
<td>Service Graph</td>
</tr>
<tr>
<td>$SR$</td>
<td>Service Request</td>
</tr>
<tr>
<td>$SP$</td>
<td>Service Path</td>
</tr>
<tr>
<td>$NG$</td>
<td>Network Graph</td>
</tr>
<tr>
<td>$RG$</td>
<td>Request Graph</td>
</tr>
<tr>
<td>$F_n$</td>
<td>mapping function</td>
</tr>
<tr>
<td>$d(p_i, p_j)$</td>
<td>distance (delay) between two proxies $p_i$ and $p_j$</td>
</tr>
<tr>
<td>$h(p_i, p_j)$</td>
<td>number of hops between two proxies $p_i$ and $p_j$</td>
</tr>
<tr>
<td>$bw_{\text{norm}}(p_i, p_j)$</td>
<td>normalized bandwidth between two proxies $p_i$ and $p_j$</td>
</tr>
<tr>
<td>$vl(p_i)$</td>
<td>volatility of proxy $p_i$</td>
</tr>
<tr>
<td>$pc_{\text{norm}}(p_i)$</td>
<td>normalized proxy capacity of $p_i$</td>
</tr>
<tr>
<td>$\mathcal{F}$</td>
<td>aggregated service routing performance metric</td>
</tr>
<tr>
<td>$L$</td>
<td>set of landmark nodes</td>
</tr>
<tr>
<td>$C_i$</td>
<td>a cluster of network nodes (proxies)</td>
</tr>
<tr>
<td>$C_{i,j}$</td>
<td>a network node (proxy) within a clustered system (within cluster $C_i$)</td>
</tr>
<tr>
<td>$\eta$</td>
<td>number of clusters yielding least communication overhead in a given network</td>
</tr>
<tr>
<td>$mr_i$</td>
<td>machine resource</td>
</tr>
<tr>
<td>$bw_i$</td>
<td>bandwidth</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>number of QoS-satisfied service instances within a data cluster</td>
</tr>
<tr>
<td>$BF$</td>
<td>Bloom filter</td>
</tr>
<tr>
<td>$th_m, th_s$</td>
<td>thresholds for preventing cluster merge and split operations</td>
</tr>
<tr>
<td>$P_i$</td>
<td>a point in a resource plane</td>
</tr>
<tr>
<td>$T_f$</td>
<td>functional service tree</td>
</tr>
<tr>
<td>$T_d$</td>
<td>local data distribution tree</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>percentage of performance improvement</td>
</tr>
<tr>
<td>$\theta$</td>
<td>disturbance caused by replacement of a service node</td>
</tr>
<tr>
<td>$\mu$</td>
<td>utility associated with replacement of a service node</td>
</tr>
<tr>
<td>$\beta = \gamma \cdot \frac{\theta}{p_i}$</td>
<td>benefit of replacing a service node</td>
</tr>
</tbody>
</table>
### Table 2.2: Acronyms used in this dissertation.

<table>
<thead>
<tr>
<th>acronym</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>service DAG</td>
<td>service directed acyclic graph</td>
</tr>
<tr>
<td>GT-ITM</td>
<td>Georgia Tech Internetwork Topology Model</td>
</tr>
<tr>
<td>HFC</td>
<td>Hierarchically-Fully-Connected topology</td>
</tr>
<tr>
<td>MST</td>
<td>Minimum Spanning Tree</td>
</tr>
<tr>
<td>GNP</td>
<td>Global Network Positioning</td>
</tr>
<tr>
<td>SCT_L</td>
<td>service capability table of a local cluster</td>
</tr>
<tr>
<td>SCT_G</td>
<td>service capability table of the global clustered system</td>
</tr>
<tr>
<td>SCI</td>
<td>service capability information</td>
</tr>
<tr>
<td>CSP</td>
<td>Cluster-level Service Path</td>
</tr>
<tr>
<td>SPR</td>
<td>Single-Path Routing</td>
</tr>
<tr>
<td>MPR</td>
<td>Multiple-Path Routing</td>
</tr>
<tr>
<td>LHB</td>
<td>Local-Heuristics-Based approach</td>
</tr>
<tr>
<td>GLG</td>
<td>Geometric-Location-Guided approach</td>
</tr>
<tr>
<td>PIM</td>
<td>Protocol Independent Multicast</td>
</tr>
<tr>
<td>IGMP</td>
<td>Internet Group Management Protocol</td>
</tr>
<tr>
<td>OSP_T</td>
<td>Optimal Service-Paths Tree</td>
</tr>
<tr>
<td>LPT</td>
<td>Longest-Prefix Tree</td>
</tr>
<tr>
<td>MRD</td>
<td>Minimum-Recovery-Dependencies heuristic</td>
</tr>
<tr>
<td>IRIR</td>
<td>Independent Reporting and Independent Recovery</td>
</tr>
<tr>
<td>IRHR</td>
<td>Independent Reporting and Heuristic-based Recovery</td>
</tr>
<tr>
<td>LRHR</td>
<td>Lazy Reporting and Heuristic-based Recovery</td>
</tr>
</tbody>
</table>
Chapter 3

Small-Scale Problems and Solutions

Assuming component services are distributed in a media proxy network that is small enough so that maintaining global state about the network is feasible, service management (or service-added routing) can then rely on a centralized approach.

QoS service-added routing needs special attention because it differs from the conventional QoS (data) routing in several aspects: (1) *service functionality and service dependency* - While data routing is solely based on network connectivities, service-added routing depends on, in addition to network connectivities, service functionality of the network nodes and dependency relations among services. Later on, we will refer to these two issues as service requirements. (2) *resource heterogeneity* - While in QoS data routing, resource requirement throughout a single data path is homogeneous [38], in QoS service-added routing, resource requirement throughout a service path is heterogeneous, because different services may have different requirements on machine capacity and I/O bandwidths\(^1\). (3) *loop formation* - While data paths should be always loop-free, there is no similar restriction in service paths because it is perfectly sound for a single proxy to be visited recurrently for different services. As a consequence of these issues, solutions for QoS data routing become inapplicable to QoS service-added routing.

In this chapter, after some brief descriptions on assumptions and related work, we will derive a generic solution for performing service composition in a small network environment using a centralized solution. We start by looking at different categories of performance

\(^1\)Different services are likely to require different amounts of machine resources (CPU, memory etc), and the fact that services are transformational often makes the output data rate be different from the input data rate.
metrics that are relevant for service composition in overlay networks in Section 3.3, followed by a brief description on state information obtainment and maintenance in Section 3.4, and a detailed presentation on how to compute a QoS-optimum service path in Section 3.5. Performance results of the solution will be presented in Section 3.6.

The purpose of this chapter is to allow us to understand the major challenges and problem solutions for service composition in small networks, which will serve us as a basis for the understanding of our future chapters focusing on such issues as scalability, adaptivity, and robustness.

3.1 Assumptions

As long as the underlying network does not partition, a proxy overlay network can be considered fully connected. However, due to the routing information measurement/maintenance cost issue, application-level networks are usually configured into partial graphs [31, 37, 39]. These types of topologies are mostly suitable for data routing. In service-added routing, due to the fact that any pair of services has probability of becoming service neighbors, and that the service neighboring issue is not resolved until a service request is actually issued by a client at run time, partially connected topologies configured beforehand usually does not cater well to the run-time service dependency needs. Figure 3.1(a) shows a mesh proxy topology. If we are to seek a functional service path for \( s_1 \rightarrow s_2 \) from \( p_1 \) to \( p_2 \), then either \( p_3 \) or \( p_4 \) needs to be included into the path to act as a relay. It is clear that the more relays, the longer the service paths, and the maximum service path efficiency is only achieved if the proxy overlay topology is fully connected. If the topology is fully connected as shown in Figure 3.1(b), then all kinds of service dependencies can be satisfied without involving any relay nodes. We leave the topology issue open, as the solution is applicable to both full and partial topologies.
3.2 Related Work

Given a proxy/service topology, a single service request, and a certain routing metric \(-m\), the QoS service-added unicast routing problem is to find a functional path between two end points such that it satisfies the service and resource requirements and that its performance in terms of \(m\) is optimized. In other words, the problem is how to map a service request onto a proxy overlay network so that the resulting service path is optimum in \(m\).

Work on end-to-end service composition can be found in [25, 40, 28, 26]. Paper [25] assumes services are distributed in network routers, and composition seeks to optimize end-to-end delay only (without other QoS metrics). In [40], the authors apply an extended Dijkstra’s algorithm on top of the proxy topology to compute the best service-to-proxy mapping in terms of a resource safety function, under the assumptions that all services are available on all proxies (thus service functionality is not a concern) and that a service request has linear dependencies. However, the solution becomes unsuitable for different routing metrics or different assumptions (e.g., if only subsets of services are available on proxies, or if service requests have non-linear dependencies). In [28], a two-step solution was presented: first, a resource-shortest service configuration (the one with minimum aggregate resource requirement) is selected from the service graph, then for the linear service configuration, a service path is sought in a distributed fashion. However, decoupling service configuration selection from service path finding may yield sub-optimal solutions. As an example, assuming
Figure 3.2: (a) Two service configurations - c₁ and c₂; (b) Part of a proxy network composed of p₁ and p₂.

the two service configurations - c₁ and c₂ - in Figure 3.2(a): using the “minimum aggregate resource requirement” selection criterion, c₂ is the preferred configuration. However, the real conditions in proxy network (Figure 3.2(b)) indicate that choosing c₁ would be actually more advantageous, because CPU-memory bandwidths are usually much larger than those of wide area network links (ratios in magnitude of 100 or even higher can be assumed). Therefore, if both services - s₁ and s₂ - can be instantiated on a single proxy, the bandwidth requirement between them becomes negligible compared to CPU bandwidth. In conclusion, whether or not a bandwidth (or proxy capacity) requirement is high cannot be concluded off-line from its absolute value; rather, it should be compared against the available one. Moreover, in [28], after a resource-shortest configuration has been selected, the distributed service path finding process adopts a greedy approach (each hop individually chooses the best-performing neighbor as its next hop) which, cannot guarantee the overall service path to be optimal. This is a limitation of distributed routing: we either get sub-optimal results by using certain heuristics to restrict the probing area, or get optimal results by flooding probes to the whole network.

Unlike [40, 28], our approach intends to solve the QoS service-added routing problem in a generic and integrated way, so that it can adapt easily to new routing metrics and new environments, and the paths computed by a single proxy approximate optimal result. Overall, the solution has the following features: (1) generic - The solution is generic enough
to be applicable to any environment assumptions and any routing metrics. (2) integrated - Service configuration selection is integrated with service path finding, so that the computed paths are optimal. (3) aggregated - We identify several routing metrics most relevant to application-level service-added routing and show how they can be aggregated to optimize the computed service paths; i.e., we present a performance function that aggregates and optimizes several routing metrics at the same time. (4) source-based - Like [40], our solution also adopts a source-based approach, by maintaining full service routing information in each proxy. Compared to distributed approaches, source-based ones are preferred for their fastness in establishing service paths and for not flooding the network when seeking paths.

### 3.3 Performance Metrics

Depending on application specifics, performance metrics can vary greatly. Example metrics include: service path delay, proxy hop count, proxy network bandwidth, proxy capacity, and proxy volatility. Ideally, the goal is to find a service path that satisfies the service requirements and that is optimal in all relevant performance aspects. However, optimizing multiple metrics at the same time is sometimes impossible, as they may conflict with each other. In this section, we present a single routing function - $\mathcal{F}$ - that aggregates the above metrics, so that optimizing the value of $\mathcal{F}$ more or less tends to optimize the individual metrics. Before presenting $\mathcal{F}$, we first discuss the individual routing performance metrics.

#### 3.3.1 Individual Performance Metrics

Performance metrics can be classified into three categories: additive, concave, and multiplicative [41]. Among our interested performance metrics, delay and hop count are additive; bandwidth and proxy capacity are concave; and proxy volatility is multiplicative. Recall that a concrete service path may have the form: $SP = (-/p_0, s_1/p_1, \ldots, s_n/p_n, -/p_{n+1})$, where $p_0$ and $p_{n+1}$ are source and destination proxies, respectively; $s_i/p_j$ means that service $s_i$ is
mapped onto proxy $p_j$, and $- / p_i$ means no service is mapped onto $p_i$, i.e., $p_i$ acts as a relay.

**Delay and hop count (additive):** Delay $d$ of the service path $SP - d(SP)$ is the time required for the data to get through service path $SP$, which includes transmission delay and service execution delay; i.e., $d(SP) = \sum_{i=0}^{n} d(p_i, p_{i+1}) = \sum_{i=0}^{n} trans(p_i, p_{i+1}) + \sum_{i=1}^{n} exec(s_i / p_i)$. We consider hop count at the proxy granularity, and the number of proxy hops is the number of times the service path needs to switch proxies (if a proxy is visited twice for two different services, then it is counted twice). The goal of service-added routing is to find a path such that the aggregated delay and/or the total number of hops are minimized.

**Bandwidth and proxy capacity (concave):** While in data routing, bandwidth requirement in a single data path is homogeneous, and bandwidth optimization is achieved by seeking the *widest* path [38], in service-added routing, due to the heterogeneous resource requirement issue, selecting the *widest*-path in absolute value is no longer appropriate. For instance, in Figure 3.3(a), using the widest-path criterion, service $s$ would be routed through proxy $p_j$, leaving the residual bandwidth from $p_j$ to $p_d$ zero. While the main objective of the widest-path selection in data routing is to balance traffic on the Internet links, we see that traffic balance is not achieved with the genuine widest-path selection in service-added routing. To achieve traffic balancing, the residual bandwidth needs to be normalized based on the bandwidth requirement. We define the normalized bandwidth as the ratio of residual bandwidth to required bandwidth: $bw_{norm} = \frac{bw_{res}}{bw_{req}}$. After the normalization process, the widest-path selection criterion can again help to achieve better traffic balance. Using the normalized widest-path criterion, $s$ should be routed through $p_i$ (shown in Figure 3.3(b))

Note that the bandwidth we talk about here is *service bandwidth* (bandwidth between two services) in that, when two services are located in two separate proxies, then the service bandwidth is the network bandwidth between the two proxies, and when two services are located in one single proxy, then the bandwidth is that proxy’s CPU-memory bandwidth.

---

2Note that the bandwidth we talk about here is *service bandwidth* (bandwidth between two services) in that, when two services are located in two separate proxies, then the service bandwidth is the network bandwidth between the two proxies, and when two services are located in one single proxy, then the bandwidth is that proxy’s CPU-memory bandwidth.
machine capacity as a single numerical value. Let \( bw_{\text{norm}}(p_i, p_j) \) denote the normalized residual bandwidth from \( p_i \) to \( p_j \), and let \( pc_{\text{norm}}(p_i) \) denote the normalized residual proxy capacity of \( p_i \), then: \( bw_{\text{norm}}(SP) = \min \{ bw_{\text{norm}}(p_0, p_1), bw_{\text{norm}}(p_1, p_2), \ldots, bw_{\text{norm}}(p_n, p_n + 1) \} \), and \( pc_{\text{norm}}(SP) = \min \{ pc_{\text{norm}}(p_0), pc_{\text{norm}}(p_2), \ldots, pc_{\text{norm}}(p_n + 1) \} \). With traffic and proxy load balancing in mind, the goal of service-added routing is to find a path so that the bottleneck normalized bandwidth and the bottleneck normalized proxy capacity are widest; i.e., \( bw_{\text{norm}}(SP) \) and \( pc_{\text{norm}}(SP) \) are maximized.

**Proxy volatility (multiplicative):** Another metric relevant to QoS service-added routing is proxy volatility - probability of a proxy being down. Given that proxies may vary greatly in this respect, the objective is to find a service path whose aggregate volatility is the lowest, so that the transmission will most likely be successful. Let \( vl(p_i) \) denote the volatility of proxy \( p_i \), then \( vl(SP) = 1 - \prod_{i=1}^{n}(1 - vl(p_i)) \). Note that each proxy \( p_i \) in \( SP \) is counted once for \( SP \)'s volatility, even if \( p_i \) is visited more than once for different services. Minimizing \( vl(SP) \) amounts to maximizing \( \prod_{i=1}^{n}(1 - vl(p_i)) \) (the probability of successful transmission), which follows the multiplicative composition rule.
3.3.2 Aggregated Performance Metric - \( \mathcal{F} \)

Many multimedia applications require that multiple performance metrics, instead of just one, be optimized at the same time. A common approach to achieving multiple-metric optimization is to define a function and generate a single metric from multiple parameters.

Let: (1) \( p_{i-1} \) and \( p_i \) denote two proxies onto which two consecutive services are mapped; (2) \( d(p_{i-1}, p_i) \), \( h(p_{i-1}, p_i) \), \( bw_{\text{norm}}(p_{i-1}, p_i) \) denote, respectively, the delay, hop count, and normalized bandwidth between nodes \( p_{i-1} \) and \( p_i \); (3) and \( vl(p_i) \) and \( pc_{\text{norm}}(p_i) \) denote, respectively, the volatility and normalized proxy capacity of \( p_i \). We define our aggregated performance function as follows:

\[
\mathcal{F}(p_{i-1}, p_i) = \frac{d(p_{i-1}, p_i) \cdot h(p_{i-1}, p_i) \cdot vl(p_i)}{\alpha \cdot bw_{\text{norm}}(p_{i-1}, p_i) + (1 - \alpha) \cdot pc_{\text{norm}}(p_i)}
\]

Note that if two services are located in one single proxy, i.e., \( p_{i-1} = p_i \), then \( h \) is zero, otherwise, it’s one if the proxy topology is fully connected, and it’s the number of hops that separate \( p_{i-1} \) and \( p_i \) if the topology is partial. The reasoning behind the function \( \mathcal{F} \) is: (1) delay, hop count, and volatility are all metrics that we want to minimize, thus they are put at the upper part of the fraction, and since they are incomparable to each other, the most meaningful operation among them is multiplication; (2) normalized bandwidth and normalized proxy capacity are something that we want to maximize, thus they are put at the lower part of the fraction. They are comparable, because both are normalized values, thus summation can be a meaningful operation between them. The terms \( \alpha (0 \leq \alpha \leq 1) \) and \( (1 - \alpha) \) are used to adjust the weight of each metric in the aggregated function. The soundness of this function will be confirmed through simulations in Section 3.6. \( \mathcal{F} \) is additive, thus \( \mathcal{F}(SP) = \sum_{i=0}^{n-1} \mathcal{F}(p_i, p_{i+1}) \). Our goal is to find a QoS-satisfied service path that minimizes \( \mathcal{F}(SP) \).
3.4 State Information Maintenance

We let each proxy periodically monitor the performance values such as its own machine capacity/volatility, the communication delay/bandwidth from itself to other neighbors and distributes this information, along with the proxy’s service capability information, to others either directly or by using the link-state-like protocol.

The delay information between two network nodes can be measured by using a simple ping protocol, and methods for measuring end-to-end available bandwidths can be found in [42, 43]. Thus, every proxy maintains an up-to-date global (service and resource) state about the nodes and links in the system.

3.5 QoS Service Path Computation

In data routing, given a network topology, then a classic graph algorithm, such as the Dijkstra’s algorithm and its variants, can be applied to find an optimal network path between two nodes. However, given a proxy/service topology (a graph) and a service request (another graph), none of the existing graph algorithms can be applied directly to compute an optimal service path between two nodes, due to the service requirements. The goal of computing an optimal service path can be accomplished by first completing a mapping process, which takes the proxy/service topology and the service request, and maps them into a Directed Acyclic Graph (service DAG). In such a service DAG, any path that goes from the source to the sink node satisfies the service functionality and dependency requirements, thus reducing the complexity of the service-added routing problem greatly. Once obtained the service DAG with the correspondent node and link resource values, algorithms similar to those for QoS data routing can be applied, albeit with a more complex resource checking process (as shown in Section 3.5.2), to compute an optimal QoS service path in terms of a given performance metric. We now proceed to describe these steps in detail.
3.5.1 Mapping

The mapping process takes two pieces of information, proxy/service topology and service request, and maps them into a service DAG as shown in Figure 3.4. Detailed procedures of mapping are as follows (refer also to Figure 3.4):

1. *instance finding* - Find, for each requested service, instances of it in the proxy overlay. For example, $s_1$ in the service request has instances in three different proxies, $p_1$, $p_2$, and $p_3$. These instances are labeled as $s_1/p_1$, $s_1/p_2$, and $s_1/p_3$. Recall that the notation “$s_i/p_j$” means “service $s_i$ at proxy $p_j$”.

2. *connecting* - Create link from one service node $sn_i$ to another node $sn_j$ in the service DAG if, in the service request, $service(sn_i)$ has a directed link to $service(sn_j)$. This is so because we assume the proxy topology is not partitioned, thus any proxy should be able to reach any other proxy either directly or indirectly.

3. *labeling and resource screening* - Label all related nodes/links with measured performance values (such as delay, hop count, residual bandwidth, residual proxy capacity, and proxy volatility), and screen out those nodes and links whose residual proxy capacity and residual bandwidth are less than those required by the services. When a node is screened out, then all of its in/out links should be also screened out. For clarity purposes, we only label the residual bandwidths and residual proxy capacities in Figure 3.4. As mentioned in Section 3.3, bandwidth could be CPU-memory bandwidth or network bandwidth depending on whether or not two consecutive services are mapped onto a single proxy. In Figure 3.4, we label CPU bandwidths as 1000. If the proxy topology is fully connected, then the bandwidth between two service nodes $s_i/p_\alpha$ and $s_j/p_\beta$ is that of the logical link connecting $p_\alpha$ to $p_\beta$, which is measured directly [42, 43]. If the topology is not fully connected, the bandwidth between the two service nodes should be the bottleneck bandwidth of the shortest overlay path between $p_\alpha$ and $p_\beta$. 

34
Figure 3.4: Mapping process.
In parallel of labeling, nodes and links whose residual resources are lower than the required values should be immediately screened out. Such nodes and links are shown in dashed circles and dashed lines in Figure 3.4.

In [25], the authors also devised a service routing method based on mapped topologies. In their approach, for each service in the request, a new layer of the network topology is created, and linked with the layer above it. A shortest-path algorithm is then performed from the source node at layer 0 to the destination node at layer \( l \) (assuming \( l \) is the number of requested services to be in the path). This mapping may generate huge graphs, because each layer is a copy of the complete network graph. In networks of fair sizes, where it's reasonable to assume the number of instances per service is far less than the number of proxy nodes (i.e., per service distribution is generally sparse), our mapping method would generate much smaller graphs. Also, in [25], no QoS issues were considered.

### 3.5.2 Routing Computation

Once the service DAG has been obtained, if it isn’t because of the loop problem, then conventional graph algorithms, such as the Dijkstra’s algorithm or DAG-Shortest-Paths algorithm, can be directly adopted to compute paths that optimize certain routing metric. Due to the loop problem, the resource screening step done in the mapping process does not guarantee that a path computed by using a conventional graph algorithm such as the Dijkstra’s algorithm, would contain sufficient resources. Taking Figure 3.5 as an example: if the final computed service path is \( \langle \ldots, s_1/p_1, s_2/p_1, \ldots \rangle \), then just knowing that \( p_1 \) had enough resources at the mapping stage is not sufficient, because resource checking was only done individually; at \( s_1/p_1 \), it was checked only whether or not the resources were sufficient to serve \( s_1 \), and at \( s_2/p_1 \), it was checked only whether or not the resources were sufficient to serve \( s_2 \). However, in practice, either or both services can be mapped onto \( p_1 \), and the residual resources may fail to meet the requirements in the latter case: when both services
are mapped onto \( p_1 \). Note that we are interested in streaming applications, where services mapped onto a single proxy will get executed in parallel as the streaming data passes through.

This indicates that the resource screening at the mapping stage is not sufficient. We address the problem by adding a backtracking resource checking process in a traditional shortest-paths algorithm such as the DAG-Shortest-Paths algorithm. Taking \( \mathcal{F} \) as the routing metric, and \( sn_u \) and \( sn_v \) as two connecting service nodes in the service DAG, we only perform \( \text{relax}(sn_u, sn_v, \mathcal{F}) \) after verifying that \( sn_v \)’s updated residual proxy capacity and \( (sn_u, sn_v) \)’s updated residual bandwidth are both non-negative. The node \( sn_v \)’s updated proxy capacity is calculated by backtracking to \( sn_u \) and all \( sn_u \)’s predecessors, and subtracting from \( sn_v \)’s current proxy capacity, the amount of proxy resources that has been consumed by \( sn_u \) and \( sn_u \)’s predecessors. Similar procedure is done for bandwidth checking. The extended DAG-Shortest Paths algorithm, which adds one line (line 4) into the original algorithm [44], is presented below (note that \( \text{pred}(sn_u) \) denotes the predecessor node of \( sn_u \):

---

**DAG-SHORTEST-PATHS**(G, \( \mathcal{F} \), s)

1. \( \text{INITIALIZE-SINGLE-SOURCE}(G, s) \)
2. for each vertex \( sn_u \) taken in topologically sorted order
3. do for each vertex \( sn_v \in \text{Adj}[sn_u] \)
4. \( \text{if PERSISTENT-RESOURCE-CHECKING} \) \((\ldots, \text{pred}((\text{pred}(sn_u)), \text{pred}(sn_u), sn_u, sn_v)) \)
5. \( \text{do RELAX}(sn_u, sn_v, \text{delay}) \)

**bool PERSISTENT-RESOURCE-CHECKING**\((\langle sn_1, sn_2, \ldots, sn_{k-1}, sn_k \rangle) \)

\[
\text{if } p_{\text{req}}(sn_k) - \sum_{i=1}^{k} p_{\text{req}}(sn_i) |\text{proxy}(sn_i) = \text{proxy}(sn_k) | \geq 0 \\
\text{and if } b_{\text{req}}(sn_{k-1}, sn_k) - \sum_{i=1}^{k-1} b_{\text{req}}(sn_i, sn_{i+1}) |\text{proxy}(sn_i) = \text{proxy}(sn_{k-1}), \text{proxy}(sn_{i+1}) = \text{proxy}(sn_k) | \geq 0
\]

then return \( \text{true} \)

else return \( \text{false} \)
3.5.3 Complexity Analysis

**Mapping:** Let $N_p$ denote the number of proxies in the proxy overlay, let $N_n$ and $N_l$ denote, respectively, the numbers of nodes and links in the service graph (SG), and let $K$ denote the maximum number of instances per service in the overlay network (for sparse per service distribution, $K \ll N_p$). Thus, the number of nodes in the service DAG - $V$ - can be written as $V = O(N_n \ast K)$, and the number of links in the service DAG - $E$ - can be written as $E = O(N_l \ast K^2)$. The complexity of labeling the resource values of the nodes and links depends on how the proxy topology is structured. If the topology is considered a fully connected graph, then performance values of, for instance, bandwidth and delay between two neighboring services, are directly measured and obtained. In this case, labeling a single link or node’s performance value takes constant time. However, in case the proxy topology is not fully connected (e.g., a mesh), then the delay between two neighboring services should be the aggregated delay of all links that make up the shortest path between the two nodes, and the bandwidth between two neighboring services should be the bottleneck bandwidth of all links on the shortest path. Both values can be derived using algorithms such as Dijkstra, Bellman-Ford, or Floyd-Warshall, whose performance is no larger than $O(N_p^3)$. Note that aggregated delays or bottleneck bandwidths may be computed once and cached for future service path computations. Updates are only necessary if the delay or bandwidth of certain links have been changed since the last report of state.

**Routing Computation Using DAG-Shortest-Paths* Algorithm:** The complexity of applying the DAG-Shortest-Paths* algorithm on top of the service DAG, whose number
of nodes is $V$ and whose number of links is $E$, is dominated by the for loops. Let $l_{sc}$ denote the size of longest service configuration in SG, then the complexity of the back-tracking resource checking process inside the inner for loop is $O(l_{sc})$, (the resource checking can back track up to $l_{sc}$ nodes/links). The total complexity of DAG-Shortest-Paths* is $O((V + E) \times l_{sc})$.

### 3.6 Performance Evaluation

We implemented our solutions in ns2 and measure their performances in several aspects. The physical topologies used in the tests all follow the Transit-Stub model described in [45].

#### 3.6.1 Performances of $F$

In study 1 and study 2, we assume a fully connected proxy topology, where each proxy monitors its own node and link conditions actively, and reports the results to other proxies in the system periodically. We consider an overlay topology of 20 proxies, where each proxy is assigned a random amount of capacity, and each link is assigned a random amount of bandwidth. Each proxy has a set of locally available services, and has certain volatility associated with it. We roughly assume the ratio of CPU bandwidth to network bandwidth to be 100.

In this study, we concentrate on evaluating the performance of our aggregated metric $F$ in aspects of its individual metrics: delay, hop count, normalized bandwidth, normalized proxy capacity, and proxy volatility, by comparing it with performances of two cases: best and random. In the best case, the path computation only seeks to optimize one of the single metrics (such as delay or hop count). In the random case, a service path is chosen randomly. Note that in all cases, the found paths always satisfy the service and resource requirements, the difference lies in their optimization metrics (in the random case, optimization metric is none). We run 20 test cases; with each test case consisting of 200 randomly generated client requests. All 20 test cases have the same environment settings, e.g., in terms of initial proxy
Figure 3.6: Performance values of $F$ compared with those of the best and random cases. Capacities and bandwidths. Service requests arrive randomly with a maximum inter-arrival time of 60s, and each established service path may remain active for up to 30 minutes.

Figure 3.7 shows the performances of $F$ in terms of individual metrics; the values for each test case are averaged over the results obtained for 200 client requests. We see that $F$’s individual performances range between best and random. In most cases, they are closer to best than random, which indicate that $F$ is a sound aggregated function. Only in the figure of “normalized capacity” is $F$’s performance comparable to random. This is so because, by nature, $F$ tends to map consecutive services onto a single proxy in order to reduce hop count and network bandwidth usage. Since not all metrics can be optimized at the same time, the merit comes at a price of compromising proxy capacity balancing in these cases. We say that proxy capacity balancing is compromised in these cases because when two consecutive services have chances of mapping onto a single proxy ($h = 0$), then the value of $F$ becomes zero, independent of how large or how small the value of $pc_{norm}$ is at the lower fraction of $F$. 

40
3.6.2 Integrated Approach vs Two-Phase Approach

We learned that decoupling resource configuration selection from service path finding may yield sub-optimal paths. In this study, we concentrate on showing the performance differences between the integrated approach and the two-phase approach. In our simulation, each service graph may contain somewhere between three and eight service configurations. In the two-phase approach, we first select a “resource-shortest” path out of the service graph according to [28], and then compute a service path that optimizes $\mathcal{F}$. Note that in [28], after a resource-shortest path is selected, service paths will be computed distributively by using a greedy approach (each hop selects the best next hop greedily), but in our test, after a resource-shortest path has been selected, we compute an overall optimal path, because we are only interested in studying the effect of separating configuration selection from service path finding. Figure 3.7(a) shows the performance results of the two approaches (the $x$-axis shows the number of the test cases, and the $y$-axis shows the $\mathcal{F}$ value averaged over 200 client requests). The integrated approach yields better overall performance than the separate approach. Note that in order to make the comparisons fair, in all of the simulation tests conducted so far, we made the available resources at proxies and networks sufficiently large, so that all path findings are successful.

We also compare the path finding success rates of two approaches in faces of resource scarcity. We set lower amounts of resources in the proxies and networks, and let service paths remain active for longer periods of time, so that resources may become used up at some times. We also ran 20 test cases, each consisting of 1000 client requests. Figure 3.7(b) shows that integrated approach clearly yields better path finding success rates. These simulation results indicate that service configuration selection should not be decoupled from service path finding whenever possible. We say whenever possible because in certain situations, the merit may not worth the associated cost. Tradeoffs need to be carefully evaluated. For instance, in the case of distributed routing, there’s tradeoff between the breadth of flooding
Figure 3.7: (a) Comparison of performance values of $\mathcal{F}$ between the integrated approach and the separate approach. (b) Success rates of integrated approach vs success rates of two-phase approach

and path efficiency. Since our approach adopts source routing, integrating configuration selection with service path finding is doable with no additional cost.

3.7 Conclusions

In this chapter we identified the key problems and challenges encountered in QoS service-added routing, and presented a generic approach to solving the QoS service routing problem (through an additional mapping process) in network environments in which network nodes can afford to maintain global system state. With the understanding of problems and solutions in a simple case, in the next chapters, we will concentrate on such aspects as scalability, adaptivity, and robustness of the solutions.
Chapter 4

Scalability in Network Size Aspect - Hierarchical Approach

Centralized solutions as described in Chapter 3 do not scale because routing information maintenance overhead grows quickly with the size of the network. Two rules-of-thumb approaches adopted in large networked or distributed systems for achieving scalability in network size are the hierarchical approach and distributed approach. We devise and describe the two approaches in two separate chapters: this chapter for the hierarchical approach, and Chapter 5 for the distributed approach.

The purpose of the hierarchical approach is to organize a large network into smaller clusters/groups, so that topology abstraction and state information aggregation become possible to significantly reduce state maintenance overhead [46, 47, 48].

While hierarchical routing is not a new concept (as hierarchical QoS data routing in ATM networks has been investigated in [48, 47, 49]), QoS service-added routing in overlay networks presents a number of unique challenges that demand special efforts. We first identify the unique challenges in Section 4.1, followed by two solutions, solution A in Section 4.2 and solution B in Section 4.3, with different designs and at a gradually increasing level of complexity. Describing the design evolution allows us to obtain an insightful understanding of the related subjects.

4.1 Challenges

We identify the following as the most salient challenges for performing hierarchical QoS service-added routing in overlay networks.
1. identification of clusters at the application layer and management of dynamic membership of network nodes - In a physical network, clusters are mostly defined manually by humans based on properties of the networks such as location, administrative domain, or connectivity. However, an overlay network has a virtual, fully-connected topology as long as the underlying physical network does not partition. To manually define clusters and configure topology of a large overlay network is infeasible. Therefore, we need a solution that automatically (1) detects clusters that conform to the underlying physical topology, (2) defines proper links among the nodes as well as roles for nodes in a hierarchical topology, and (3) copes with dynamic membership of network nodes, including allowing nodes to join/leave the system and performing reclustering operations to maintain clustering optimality.

2. proper aggregation and distribution of routing information - Since the main purpose of adopting a hierarchical-based solution is to reduce state maintenance overhead by means of topology aggregation and state information aggregation, we need to aggregate and distribute routing information properly, so that the aggregated information is meaningful and concise (yet sufficient) for hierarchical path computation. In addition to service capability information of the network nodes/clusters, we want to consider communication delay, available machine resource, and available bandwidth as QoS parameters.

3. computation of efficient service paths in a hierarchically structured network topology - While our solution described in Chapter 3 is suitable for service routing over flat topologies, how can a single-level flat topology solution be extended for hierarchically structured network? In terms of QoS, we seek to find delay-optimum service paths that satisfy resource conditions.

We target to develop solution frameworks to deal with the challenges above. In the following, we describe two solutions with different designs and at an increasing level of com-
plexity to allow us to understand the design insights. Solution A in Section 4.2 considers communication delay as the only QoS parameter, and does not deal with dynamic membership issues in the overlay network. The purpose of describing this simpler solution is to allow us to understand the fundamentals of hierarchical service routing; the performance results show some tradeoffs between our hierarchical approach and a mesh-based flat-topology solution. In Section 4.3 we proceed to describe a more complete solution, that deals with all of the above-described challenges with an alternative hierarchical topology. We analyze why the design change would be desirable.

4.2 Solution A

For the first problem, since delay (proximity) will be used as a performance metric when seeking service paths, we will use proximity as a metric for clustering network nodes. To achieve this goal, a combination of mechanisms (for distance map obtainment and clustering) will be adopted, so that the clustered overlay proxy network is congruent with the underlying physical network. Once the clusters (of proxies) have been detected, we will create the topology in such a way that nodes within a single cluster are fully connected, and the clusters themselves are also fully connected by their representatives. We name such a topology an HFC (Hierarchically Fully-Connected) topology.

In solution A, we choose to have border nodes be cluster representatives. Thus, in an $HFC_A$ topology, clusters are fully connected by their border nodes, so that any pair of intra-cluster nodes can communicate to each other directly, and any pair of inter-cluster nodes can communicate to each other via their border nodes. Figure 4.1 depicts an example of a bi-level $HFC_A$ topology.

This topology design choice stems from the following observation. Service routing exhibits some different features (e.g., service functionality and service dependency) than data routing that make the use of partial mesh topologies, which is suitable for data routing, not as
suitable. This is so because, in data routing, nodes simply participate as relays, and pass data as is to their neighboring nodes. However, in service routing, as the data traverses, nodes may be required to process it differently using different services. Since how services are to be composed (i.e., the service dependency issue) is mostly resolved at the runtime, two runtime-defined neighboring services may appear to be several nodes apart in a statically configured partial mesh. In other words, meshes configured statically do not reflect well service dependency needs that are resolved at execution time. In fact, highest efficiency in service paths can be only achieved if the overlay network is considered fully connected (assuming the underlying physical network does not partition). We thereby try to make the hierarchical topology as fully connected as possible. While large networks cannot afford full topologies, after having done proximity-based clustering, small groups of nearby nodes will afford to be fully connected. In a bi-level $HFC_A$ hierarchy, two nodes (thus also the services installed on them) are at most two nodes away. By limiting the number of hops between any pair of services will potentially increase service path efficiencies.

For the second problem, since the topology clustering was based on node proximity, it’s viable to think of intra-cluster distances to be negligible compared with inter-cluster distances. Besides the distance information, we only need to aggregate and maintain the
service capability information (SCI) of the clusters. Assuming that each service can be uniquely named, and a single proxy’s SCI is represented as a set of service names, we can aggregate SCI of a group of proxies as the union of their individual SCI sets. State information will be further discussed in Section 4.2.2.

For the third problem, we will adopt the solution developed in Chapter 3 for intra-cluster service routing, and use a modified version for inter-cluster service routing. The resulting solution will perform hierarchical service routing in a top-down, divide-and-conquer fashion, in a sense that a single node with partial global state of the system first resolves the inter-cluster service routing problem, and then lets certain proxies inside those clusters along the cluster-level path find intra-cluster service routes.

4.2.1 Hierarchical Topology Construction

Our $HFC_A$ topology has the following basic properties:

1. **distance-based clustering:** Nodes are clustered by their proximity in the Internet.

2. **connectivity:** All intra-cluster nodes are fully connected among themselves, and all clusters are fully connected by their border nodes. Later on, we will call links crossing two clusters *external links*, and all other links within single clusters *internal links*.

3. **border node selections:** The border nodes between two clusters are selected to be the two closest nodes belonging to the two clusters.

4. **visibility:** Each cluster is visible by its border nodes from outside.

An $HFC_A$ topology makes service routing in a large-scale overlay network feasible and efficient because of the following reasons. First, clustering allows topology abstraction so as to reduce the state information maintenance overhead. Second, clustering based on the proxies’ proximity makes small groups of closely located proxies afford to be fully connected to best cater to runtime-defined service dependency needs. Third, the border node selection
rule maximizes routing efficiency between clusters due to geometric properties. Also due to geometric properties, for clusters of reasonable sizes, it’s very unlikely that a single node will be selected to be border nodes to all other clusters in the system, which improves load balancing on border nodes. Lastly, when dealing with hierarchical topologies, the most common way of topology aggregation is to represent a group of nodes as a single logical node \([46]\). Such a representation is simplest, but also introduces more imprecision \([50]\). In our framework \(A\), we will make all border nodes of a cluster (several nodes instead of a single one) represent a group. Such a visibility feature will help find efficient service paths with better precisions. This issue will become clearer in Section 4.2.3.

The construction of an \(HFC_A\) topology follows three major steps: distance map obtainment, clustering, and selection of border nodes, which are described below.

**Distance Map Obtainment**

As end-to-end latency is an important application-level measurable metric that gives a good reflection of the underlying physical network \([51, 15]\), we will use this metric to represent Internet distances. Suppose \(N_p\) is the number of network nodes in an overlay network, if we are to get a complete distance map for the clustering purpose, then \(O(N_p^2)\) end-to-end measurements are needed, and \(O(N_p^2)\) entries will be in the map, which is an expensive operation. However, in \([15]\), the authors made two interesting observations, which will allow us to reduce the complexity of the work significantly. Through real-world Internet distance measurements, it was observed that: (1) Internet hosts can be mapped into an \(N_k\) dimensional coordinate space such that the geometric distance between every pair of nodes more or less reflects the network distance (round-trip propagation and transmission delay) between the nodes; (2) without directly measuring the distance between a pair of network nodes \(p_x\) and \(p_y\), this distance information can be calculated (predicted) if we know the distances between \(p_x\) and a set of common landmark nodes \(L\), and the distances between \(p_y\) and \(L\). Therefore, to obtain the complete distance map of \(N_p\) overlay proxies, we do the
following:

1. Set up a small group of $N_m$ landmarks - $L$, and let each of them measure the distances between itself and all other landmarks. To minimize the effect of Internet noises, we take the minimum value of several measurements.

2. Map the obtained distance map of $L$ into a $N_k$-dimensional geometric space $S$ with minimum error. This is a function minimization problem solvable by mathematical methods such as [52].

3. Each proxy obtains the coordinates of $L$, and measures the distances between itself and $L$ in order to be able to derive its own coordinates relative to $L$’s positions, again through some function minimization method [52]. To minimize Internet noises, the minimum of several measurements should be taken.

Using such an approach, a complete distance map of $n$ proxies can be obtained by using $O(N_m^2 + N_p N_m)$ measurements and will only have $O(N_k N_p)$ of entries ($N_p \gg N_m$ and $N_p \gg N_k$), compared to $O(N_p^2)$ measurements and $O(N_p^2)$ entries in a genuine approach. Note that the goal of setting up the landmarks is to provide to regular proxies some reference points in the geometric space - $S$; the landmarks will not participate in any other activities.

**Clustering by Graph Theory**

We assume a particular proxy - $p$ - is elected to perform the clustering operations once all of the proxies have reported their own coordinates to it. Clustering is an active research area spanning several fields. Many clustering methods exist in the literature for different objectives, and most of them are guided by certain laws or principles [53]. Among them, we cite the famous Gestalt principle of grouping by *proximity*. Based on this principle, Zahn [54] demonstrated how the minimum spanning tree (MST) can be used to detect clusters. We adopt this method to detect proxy clusters as follows.
Figure 4.2: (a) A set of $N_p$ nodes in space $S$; (2) an MST connecting the nodes (inconsistent edges are marked with $X$; (3) clusters detected by removing the inconsistent edges in (b).

1. Construct the MST for the set of $N_p$ nodes in $S$.

2. Identify inconsistent edges in the MST.

3. Remove the inconsistent edges to form connected components and call them clusters.

The crucial step in the algorithm is the definition of inconsistency. We consider an edge to be inconsistent when its length is significantly larger than the average of nearby edge lengths [54]. Let $a$ denote the length of the link being examined - $l$, and let $b$ denote the average length of links of $l$'s left and right sub-trees. We say that $l$ is inconsistent if $a/b > k$, where $k$ is a selected number, e.g., 2, 3, ... .

Selection of Border Proxies

We also let $p$ carry out the border proxies selection operation. As stated, in the $HFC_A$ topology, clusters are fully connected by their border proxies. Thus, between every pair of clusters, a pair of border proxies is selected. For maximum routing efficiency, the two border proxies between a pair of clusters are selected to be the pair of nearest proxies belonging to the two clusters. Let $C_x = \{C_{x,1}, C_{x,2}, \ldots, C_{x,m}\}$ and $C_y = \{C_{y,1}, C_{y,2}, \ldots, C_{y,n}\}$ denote two clusters, and let $[C_{x,b}, C_{y,b}]$ denote their border proxies, where $C_{x,b} \in C_x$ and $C_{y,b} \in C_y$, then
for all other pairs of proxies \([C_{x,i}, C_{y,j}]\) (such that \(C_{x,i} \neq C_{x,b}, C_{y,j} \neq C_{y,b}, C_{x,i} \in C_x\), and \(C_{y,j} \in C_y\)), \(d(C_{x,i}, C_{y,j}) \geq d(C_{x,b}, C_{y,b})\).

Once the proxy \(p\) is done with clustering and border node selection, it will distribute the relevant topology information to each proxy in the system. In particular, each proxy in the system will learn the following from \(p\): (1) its own cluster’s ID and membership information, i.e., who are other members belonging to this cluster; (2) the cluster IDs in the system and their border proxies; (3) coordinates of all members within the cluster and coordinates of all border proxies in the system. Figure 4.3 depicts the information learned by \(C_{2,1}\) from \(p\).

### 4.2.2 Service Routing Information Aggregation and Distribution

As stated, service routing in this case A requires two pieces of information: distance and service functionality/capability. Since each proxy has already the relevant coordinates information that allows itself to derive distances between any intra-cluster nodes and between any pair of clusters (represented by the corresponding border nodes), only the service capability information needs to be further distributed. Each proxy will maintain two Service Capability Tables, one for all proxies in its own cluster - \(SCT_i\), and the other for all clusters in the system - \(SCT_C\). The following protocol will be adopted for distribution and maintenance of the nodes’ service capability information.
1. **Local State:** Every proxy \( p_i \) in the system periodically distributes its local service capability information through a *local state* message to all of the proxies within its own cluster. In a *local state* message, \( p_i \) lists the names of services installed on \( p_i \). A proxy \( p_j \) that receives a *local state* message will update its \( SCT_L \).

2. **Aggregate State:** Each border proxy \( p_b \) periodically aggregates the service capability information of its own cluster and distributes it through an *aggregate state* message to the neighboring border nodes in other clusters. In an *aggregate state* message, \( p_b \) lists the names of all services available in its entire cluster. Assume \( S_1, S_2, \ldots, S_m \) are the sets of services available at proxies \( p_1, p_2, \ldots, p_m \) in a certain cluster \( C_i \). Then the aggregate service set \( S \) of this cluster is the union of all \( S_i \)'s; i.e., \( S = S_1 \cup S_2 \cup \ldots \cup S_m \). A border node \( p_b' \) that receives such a packet updates its own \( SCT_G \), and is responsible for forwarding it to other proxies of its own cluster. Any proxy that receives a forwarded *aggregate state* packet simply updates its own \( SCT_G \).

### 4.2.3 Hierarchical Service Path Computation

Since in the \( HFC_A \) framework, the topology and state information has been abstracted at certain point, we will not be able to find a concrete service path in one single step. Instead, we will have to perform service path finding hierarchically, first at the cluster level and then at the proxy level. With only an abstract state of other clusters in the system, no single proxy is able to find a concrete service path solely on its own, unless all requested services can be satisfied in the local cluster. The general idea of hierarchical service routing is to first let some proxy (e.g., the destination proxy specified in the request) look for a cluster-level service path, so that to which cluster each service is mapped to is defined. Later, the proxy can let certain nodes of those particular clusters decide which specific in-cluster proxies will be the concrete providers of those services and combine their decisions to obtain the final service path. Figure 4.4 depicts a picture of hierarchical service routing at high
Figure 4.4: Hierarchical routing at high level: (a) A single node - $p_d$ - first computes an inter-cluster service path connecting certain border nodes; (b) certain nodes within each cluster individually compute intra-cluster service paths and return their answers to $p_d$ to form the final service path.

level. We will call such a service path finding mechanism divide-and-conquer. Different than the flat-topology solution in Chapter 3, which mostly finds optimal solutions in terms of a given performance metric, now we can no longer guarantee that a service path - jointly computed by nodes at different levels of the topology - is optimal. That is, although we sought to optimize distance independently at different levels of service routing, the overall service path may not be optimal due to the topology and state information abstractions.

In parallel to describing the general procedures of hierarchical service routing below, we provide a small example for better illustration. The topology, as well as the detailed service capability information, are shown in Figure 4.5. There are four clusters, $C_0$, $C_1$, $C_2$, and $C_3$, whose elements (proxies) are labeled as $C_{0.0}$, $C_{1.2}$, ..., according to how they are clustered, to ease our reading. Services available at each proxy are listed at the right side of the figure.

**Inter-Cluster Service Path Finding**

Without loss of generality, we assume a service request, which comprises a source proxy, a service graph, and a destination proxy, and issued by a client, is sent to the destination proxy, and the output of this destination proxy will feed into the client’s input.
Figure 4.5: The service topology (network topology + services) of the example.

1. **map:** Based on the aggregate state information maintained at $p_d$, find instances of the requested services in all clusters in the system by looking up $p_d$’s $SCT_G$, and construct a service DAG as described in Chapter 3.

   *Example:* Based on the aggregate global state perceived by $C_{2.1}$ (Figure 4.6(a)), map the service request (Figure 4.6(b)) into a service DAG (Figure 4.6(c)) by finding instances of each service in $SCT_G$. The two end nodes of such a service DAG are, respectively, the IDs of the clusters in which the source and destination proxies fall; $p_d$ knows its own cluster’s ID, and can query the source proxy for the source proxy’s cluster ID. Note that the distance labeled on each link between a pair of clusters - $C_i$ and $C_j$, is the distance between the border proxies of two clusters. The distance is zero if $C_i = C_j$. Although for simplicity the example only considers linear service graph, the solution can be easily extended to also consider non-linear service graphs, as shown in Chapter 3.

2. **apply shortest-paths algorithm:** On top of the service DAG, a shortest-path algorithm such as Dijkstra’s algorithm, or DAG-shortest-paths algorithm can be applied to compute a shortest path. We will call the resulting path a CSP (Cluster-Level Service Path). Such a CSP is a service path comprised by possibly several external clusters,
Figure 4.6: Inter-cluster service routing (steps performed by the destination proxy $C_{2.1}$ of the service request).
whose fine-resolution states are not known at \( p_d \).

Example: Although simply applying a classical shortest-paths algorithm (e.g., DAG-shortest-paths algorithm) on top of the service DAG would result in a cluster-level shortest service path, whose total distance is the sum of the lengths of external border links making up the path, we modify the DAG-shortest-paths algorithm in such a way that selection of a shortest path also takes into account internal distances as much as possible. If, in the service topology of Figure 4.6, there are two cluster-level service paths (path 1: \( C_0 \rightarrow C_1 \rightarrow C_2 \) and path 2: \( C_0 \rightarrow C_3 \rightarrow C_2 \)) that both satisfy the given request, just judging from the external links, the proxy \( C_{2.1} \) would see no difference between the two, because both paths have their total external link lengths of 45. However, if \( C_{2.1} \) considers the internal distances between the border nodes as well, the latter might be a preferred path because: (1) in path 1, the service path will leave \( C_0 \) from \( C_{0.1} \), enter \( C_1 \) from \( C_{1.0} \), leave \( C_1 \) from \( C_{1.2} \), enter \( C_2 \) from \( C_{2.0} \), and finally reach the destination - \( C_{2.1} \), the total distance will be no less than \( 20 + 5 + 25 + 2 = 52 \); (2) in path 2, the service path will leave \( C_0 \) from \( C_{0.0} \), enter \( C_3 \) from \( C_{3.0} \), leave \( C_3 \) from \( C_{3.2} \), enter \( C_2 \) from \( C_{2.2} \), and finally reach the destination - \( C_{2.1} \), the total distance will be no less than \( 30 + 15 + 1 = 46 \). At this point, we have no way to know how long the intra-cluster service paths will be, but since the lower-bound distance of path 1 is higher than that of path 2, there’s no reason for us not to prefer the latter to the former. In order to take the internal distances into account, we need to add a back-tracking procedure before performing the regular \textit{relax} procedures in a classical shortest-paths algorithm in a way similar to what has been described in Section 3.5.2. The shortest service path is shown in bold lines in Figure 4.6(c).

3. distribute child service requests (divide): Dissect the original service request into smaller portions, and distribute these child requests to the corresponding clusters.
Example: Once getting a shortest cluster-level service path, the destination proxy $C_{2,1}$ is responsible for dissecting the original request into pieces. Starting from the source node in the CSP, a new child request is formed for a consecutive set of nodes if these consecutive nodes are all mapped into the same cluster. For each child request, the selection of source proxy and destination proxy is done as follows. Let $(C_0, C_1, \ldots, C_n)$ denote the sequence of clusters making up the CSP, and let the notation $b_{i,j}$ denote the border node inside cluster $C_i$ connecting to cluster $C_j$, then in general, the source proxy of a child request $i$ is $b_{i,i-1}$, and the destination node is $b_{i,i+1}$. If it is the first child request ($i = 0$), the source proxy is that indicated in the original service request; and if it is the last child request ($i = n$), the destination node is that indicated in the original service request. Figure 4.6(d) shows the dissected child service requests. The first two child requests are to be sent to $C_{0,1}$ and $C_{1,2}$, respectively, and the third one will be taken care of by $C_{2,1}$ itself.

4. compose child service paths (conquer): Wait for results of those child service requests to arrive, and compose all child service paths into a single one. This is the final service path.

Example: The destination proxy $C_{2,1}$ waits for all child service paths\footnote{Computations of child service paths are shown in Section 4.2.3.} to arrive, and combine them as shown in Figure 4.6(e).

Intra-Cluster Service Path Finding

Since at the cluster level, proxy $p_d$ only knows in which cluster each requested service should be located, it will rely on some in-cluster proxies to further decide which proxy, specifically, will be the concrete provider of each service. A proxy $p_x$, upon receiving a child service request that consists only of services satisfiable by $p_x$’s cluster, will compute an optimal intra-cluster service path by using the solution described in [55]. The basic procedure consists of
two steps: mapping and applying shortest-paths algorithm on top of the obtained service DAG. Different from the mapping step done for inter-cluster service routing above, where each service is mapped into a cluster by looking up $SCT_C$, now each service will be mapped onto concrete proxies by looking up $p_x$’s $SCT_L$. After completing the computation of a shortest child service path, $p_x$ sends the result (child service path) back to $p_d$ for it to be composed with others.

*Example:* Computations of intra-cluster service paths are shown in Figure 4.7.

### 4.2.4 Performance Studies

In this section, we conduct simulation tests to study the performances of our hierarchical framework. The simulations are done by using the well-known simulator *ns2*, and our Internet topologies are generated following the *transit-stub* model [45]. We will measure performances of the $HFC_A$ framework in two aspects: state information maintenance overhead and service path efficiency.
Figure 4.8: (a) Number of coordinates-related node-states kept at a single proxy; (b) number of service-related node-states maintained at a single proxy.

State Information Maintenance Overhead

The biggest advantage of hierarchical routing is that state information maintenance overhead is reduced through topology abstraction. We will study the performance of the HFC<sub>A</sub> topology by comparing it to that of single-level topologies. Since state in service routing includes two pieces of information: distance and service capability, we will quantify their overheads separately. Overhead is quantified in number of node-states; if a single node <i>p</i> keeps <i>N</i><sub>p</sub> node-states for particular state (either distance or service capability), it means that <i>p</i> maintains that many entries in its corresponding state table where each entry can be either for a single node or for a cluster, depending on situations.

Coordinates-Related Overhead: In this work, since we used coordinates-based distance map in the HFC<sub>A</sub> framework, we will also assume this for single-level topology service routing. In a single-level topology, each proxy is required to keep coordinates of all proxies in the system. Therefore, assuming <i>N</i><sub>p</sub> is the size of the overlay network, the per proxy coordinates-related overhead is <i>N</i><sub>p</sub> node-states. However, in an HFC<sub>A</sub> topology, each proxy is only required to keep the coordinates of its local cluster nodes and the coordinates of all border nodes in the system. We set up overlay topologies of different sizes: 250, 500, 750, and 1000 proxies, on top of physical topologies generated by using the TS model [45]. Each overlay topology is tested on top of 10 different physical topologies. Figure 4.8(a) shows the
results averaged over 10 tests.

**Service-Capability-Related Overhead:** In a single-level topology, each proxy again, needs to maintain service capability information for all proxies ($N_p$ nodes) in the system. However, in the $HFC_A$ framework, the number of such nodes perceived at a single proxy is the sum of the number of nodes in each proxy’s own cluster plus the number of clusters in the system. Figure 4.8(b) shows results of the simulation tests with the same setups as above.

While in flat topologies, both overheads would increase linearly with constant one, the increases are much slower in the hierarchical case (with dramatically smaller constants), meaning the $HFC_A$ framework scales much better than flat topologies. Although theoretically in the worst case, hierarchical organizations may not produce advantages, for example, when most of the nodes fall into one cluster, such undesirable phenomena did not happen in our simulation tests, and we think that extremely unbalanced network node distribution would not happen in practice either.

**Service Path Efficiency**

The goal of service routing is to find *efficient* service paths. Since in this solution, we only consider distance as the routing metric, we say that for a single service request, a shorter path is more efficient that a longer one. We will compare path efficiencies achieved by our hierarchical service routing framework ($HFC_A$ with topology abstraction) against those of regular mesh-based solutions. At the same time, we will also quantify the performance losses solely caused by topology aggregation. Hence we will compare path efficiencies achieved by our $HFC_A$ framework against those achieved by $HFC_A$ without topology abstraction. To be fair, performances will be compared in the same simulation environments. We wrote programs to generate random simulated environments for overlay topologies with 250, 500, 750, and 1000 proxies. Table 4.1 shows the settings. We conduct two sets of tests for different purposes.
We first compare path efficiencies of hierarchical service routing against those of regular meshes. A regular mesh is constructed with the following rules: each proxy creates links to its 1-4 nearest neighbors, and 1-2 randomly chosen, farther located neighbors (to make the topology connected). In a single-level (mesh) topology solution, each node maintains global state of the system. Thus, a single node is able to find an optimal service path by applying the methods described in Chapter 3. The first two bars in each 3-bar set of Figure 4.9 show the average service path lengths obtained for the two tests. Each point in the figure corresponds to the average result of up to 5 runs (on top of 5 different physical topologies), with 1000 client requests per each run. We see that despite the distance information imprecision introduced in the $HFC_A$ framework, the performance of the $HFC_A$ framework is still comparable to (actually slightly better than) single-level mesh solutions. This is the case because in the $HFC_A$ topology, the number of hops between two overlay nodes (or neighboring services) is constrained to no more than two. As predicted, this feature potentially increases service path efficiencies.

The biggest disadvantage of hierarchical routing is that path efficiencies may get compromised due to routing information imprecision. Thus, it is interesting to examine the performance degradation caused solely by topology aggregation in hierarchical service routing (i.e., performance compromise of doing hierarchical service routing). For this purpose, we compare performances achieved by $HFC_A$ topologies in two cases. In case one, we perform the topology aggregation as described before. In the second case, we do not perform any topology abstraction or state information aggregation on top of the $HFC_A$ topology.
Therefore, each proxy will have full state of the whole system. The last two bars in each 3-bar set of Figure 4.9 give us a comparison between the two; the differences between the two show the performance deterioration caused by the distance imprecision in hierarchical service routing.

4.3 Solution B

The framework described in the previous section has ignored QoS parameters (such as bandwidth and machine resources) and the network dynamic membership feature. We now design a solution framework to tackle all the challenges listed in Section 4.1. Again, our descriptions will be divided into three portions: hierarchical topology construction (Section 4.3.1), service routing information aggregation and distribution (Section 4.3.2), and hierarchical service path computation (Section 4.3.3).

4.3.1 Hierarchical Topology Construction

The construction of a hierarchical topology still requires a distance map, whose method of obtainment remains the same as that described in solution A in Section 4.2. We will
concentrate on describing some clustering problems as well as the dynamic membership issue which we overlooked in the previous solution.

**Detecting Clusters**

There are several difficulties associated with clustering: (1) properly interpreting clusters, (2) choosing or devising proper clustering mechanisms/methods, and (3) choosing proper number of clusters.

We start by defining two objectives that are pertinent to network clustering.

- **goal a**: Cluster network nodes in such a way that high density network nodes are connected together, separated from other such clusters by a region containing a relatively low density of nodes.

- **goal b**: Cluster network nodes in such a way that any pair of intra-cluster nodes are more closely-located than any pair of inter-cluster nodes; i.e., within cluster variation should be minimized and between-cluster variation should be maximized.

Apparently the two objectives look similar, but they actually yield different clustering methods, and thus also different clusterings. As we can see, the MST-based clustering method described in solution A can best achieve the first objective. The method works well for detecting disjoint clusters. However, we observe that it is subject to the “chain” phenomenon, which may cause nodes that are far away from each other to be partitioned into the same cluster as long as between these nodes there is a non-discontiguous distribution of nodes. In Internet communications, such “chain” phenomenon incurs communication inefficiency within a cluster.

The goal of minimizing within-cluster variation or maximizing between-cluster variation actually better characterizes Internet host clustering and potentially achieves better communication efficiencies. According to [56], minimizing within-cluster variation (or maximizing
between-cluster variation) amounts to minimizing square-error, and the clustering method that best achieves this goal is the \textit{square-error} clustering method.

The general objective is to obtain that partition which, for a fixed number of clusters, minimizes the square-error\footnote{Suppose a given set of \( n \) nodes has been partitioned into \( k \) clusters \( \{ c_1, c_2, \ldots, c_k \} \). The square-error for cluster \( c_i \) is the sum of the squared Euclidean distances between each node in \( c_i \) and its cluster center (centroid), and the square-error for the entire clustering is the sum of all of the within-cluster variations.}. The resulting partition is also referred to as the minimum variance partition. Square-error clustering can be done iteratively, by following the following algorithm: (1) select an initial partition with \( k \) clusters; (2) repeat the following steps until the cluster membership stabilizes; (2.a) generate a new partition by assigning each node to its closest cluster center; (2.b) compute new cluster centers as the centroids of the clusters.

The Square-Error clustering algorithm requires the number of clusters as a parameter. However, determining the right number of clusters can be difficult \cite{57}. In our case, the number of clusters \( \eta \) should be chosen such that the resulting clustering yields least communication overhead\footnote{The communication overhead of a clustered network includes inter-cluster and intra-cluster overheads, which is defined as the sum of squared distances among inter-cluster nodes and among cluster representatives (in this case centroids).}. Our intuition is that \( \eta = \sqrt{n} \) in general. However, in practice the distribution pattern of the nodes may affect \( \eta \).

We therefore conducted a study to verify this. Based on the transit-stub model, we
generate network topologies, of which subsets of \( n \) nodes are randomly selected as overlay nodes. We run Square-Error clustering algorithms over the \( n \) nodes, and finds the number of clusters that would yield least communication overhead through exhaustive search. In the tests, we used network topologies of 600 nodes, of which 50 to 550 nodes are selected to be overlay nodes. Figure 4.10 depicts the hypothetical \( \eta \) and the real \( \eta \) (mean and standard deviation calculated from 10 runs), showing that the real \( \eta \) approximates the hypothesized values. We will use these approximated values in our future clustering and re-clustering operations.

**Creating a Hierarchical Topology**

Having a set of representative nodes representing a cluster as shown in our solution \( A \) certainly enhances the precision of representation. However, it also makes it hard to generalize the topology beyond two layers. Furthermore, in solution \( A \), because the border nodes have to participate in service paths, they may become overloaded, and data delivery may become sub-optimal.

We now consider a simpler topology - topology \( b \) (Figure 4.11(b)) - in which a cluster is represented only by one node. For best representativeness, we choose the centroid to be the representative of a cluster. In literature, the mathematical centroid of a set of multi-dimensional data points is defined to be the data point that is the mean of the values in each dimension. However, in a practical network of nodes, we refer to that physical node closest to the mathematical centroid to be centroid. In the new hierarchical topology, which we call \( HFC_B \), nodes within a single cluster are fully connected, and clusters are fully connected by their unique representatives. The representatives are only responsible for exchanging routing state information, but do not have to participate in all service paths. To the outside world, a cluster is represented only by one node - its centroid, and nodes between different clusters can communicate directly for communication efficiency.

In topology \( b \), by having one single node representing a cluster, it becomes easier to
Figure 4.11: Two feasible topologies (a) topology a - $HFC_A$; (b) topology b - $HFC_B$.

generalize and extend the topology to hierarchies of more than two layers. Besides, since no unnecessary relay nodes need to be involved in a service path, data delivery can be achieved more efficiently. Considering these facts, we adopt topology b in our new framework.

Coping with Dynamic Membership

The above-mentioned clustering method (Square-Error) can be conducted by a single node - p - if all other nodes report their geometric coordinates to it. However, such an operation only defines a static clustering, but does not take dynamic node join/leave into account.

Dynamic membership can be dealt with passively or actively. In a passive approach, after an initial clustering, the clustered system will absorb membership dynamics passively by letting a new node join an existing cluster (ideally the closest cluster for best performance) and an existing node leave its local cluster. No reclustering efforts will be made.

After a node has joined or left, the overall clustering quality may have become suboptimal. Without any active reclustering operations, clustering quality will deteriorate over time. There are two options for doing active reclustering: global reclustering and local reclustering. In the global reclustering approach, after each or a number of join/leave operation(s), the whole network is re-clustered, possibly affecting all network nodes. Frequent
re-organizations of topology in a large scale will cause high routing state instability and disruption of services. For practical reasons, we want re-clustering to take place smoothly as to affect only a small area in the network.

The difficulty with the local re-clustering approach is in finding reasonable local re-clustering operations which would, in a long run, lead to a good global clustering quality, while incurring low overhead. We design two types of local operations that can be performed distributively (i.e., without relying on a central node): namely local cluster split and local cluster merge. Assuming each cluster representative also advertises the size of the cluster (number of nodes inside the cluster) to others, and based on the fact that $\eta = \sqrt{n}$ on average, we use the total number of nodes in the system to trigger local split and local merge operations.

Let $n$ be the size of an existing clustered system. If the next dynamic membership operation is a join, the system size after the join operation would be $n + 1$. If $\text{floor}(\sqrt{n+1}) - \text{floor}(\sqrt{n}) = 1$, then after the join operation, we trigger a local cluster split operation (more details described below). Similarly, if the next dynamic membership operation is a leave, the system size after the leave operation would be $n - 1$. If $\text{floor}(\sqrt{n}) - \text{floor}(\sqrt{n-1}) = 1$, then after the leave operation, we trigger a local cluster merge operation (more details described below).

It can be noted that if parallel joins and leaves keep the size of the network $n$ around the square of some number $k$, i.e., $n = k^2$, then frequent split and merge operations may be triggered, making the clustered topology very instable at times. We remedy this problem by defining and using two thresholds $th_m$ and $th_s$ around $n$ ($th_m \leq n \leq th_s$) to prevent instable triggerings; $th_m$ is set so that no merge operation would be triggered when the size of the network stays between $n - th_m$ and $n$, and $th_s$ is set so that no split operation would be triggered when the size of the network stays between $n$ and $n + th_s$.

Our local cluster split and local cluster merge operations work as follows:

- **local cluster split**: When a trigger of split operation is detected, we let $a\%$ largest
clusters in the system tentatively split their local clusters into 2, and report to each other the distance between the two tentative clusters’ centroids. The one that has largest centroid distance will win and thus commit its local cluster split.

- **local cluster merge:** When a trigger of merge operation is detected, we let \( b\% \) smallest clusters in the system find their closest clusters, and report to each other their distances to those closest clusters. The one with smallest distance will win, and thus commit the merge operation.

We call this approach *active approach with local reclustering*, and perform tests to verify its performances by comparing it with two other cases: (1) *passive approach*, and (2) *active approach with global reclustering*.

We perform two tests. In test 1, we start with a network of size 200. After the initial clustering, we let 300 new nodes continuously join the system. In test 2, we start with a network of size 500. After the initial clustering, we let 300 nodes continuously leave the system. In both cases, reclustering is triggered based on the policies shown above\(^4\). Results are shown in Figure 4.13. We see that supporting dynamic membership with our local reclustering operations helps to maintain good clustering quality (measured by the total communication overhead) and low overhead (measured as the number of affected nodes).

### 4.3.2 Routing State Aggregation and Distribution

In this solution B, the considered QoS parameters are *communication delay*, *available machine resources*, and *available bandwidth*. Since distribution of full state among nodes of a

\(^4\)Since the size of the network either grows or shrinks in each test, we did not need to set the two thresholds.
Figure 4.13: Performance comparisons of 3 approaches: (1) passive approach; (2) active approach with global reclustering; (3) active approach with local reclustering operations.
single cluster is straightforward, we focus on studying aggregation and distribution of QoS parameters among clusters.

**Distance Information**

The distance between two clusters will be measured as the distance between their cluster representatives (centroids). To minimize the representation error, a cluster $C$ also advertises to the outside world the average internal distance - $d_{internal}(C)$ - average of distances taken among all intra-cluster nodes.

**Resource Conditions**

Let $N_s$ be the number of distinctive services within a cluster, and $N_q$ be the average number of instances per service. If we are to fully represent the QoS parameters of those service instances as well as of those inter-relations (e.g., delay, bandwidth), then a complexity of $O(N_s^2N_q^2)$ is associated per cluster. We can think of two ways to reduce such representational complexity: we can try to represent the full data set at a coarse granularity (reduced granularity), or alternatively, we can choose to represent only the “strong” data set (reduced data points).

- **Reduced Granularity:** We try to reduce this complexity in several ways. First, since the network nodes are clustered in such a way that intra-cluster variations are minimized and inter-cluster variations are maximized, in coarse granularity, it is feasible to consider nodes within a single cluster having distances (communication delays) that are negligible compared to inter-cluster distances. In terms of bandwidth, instead of representing the available end-to-end bandwidth between each pair of services, we denote only the available access bandwidth of each service\(^5\). Having done this, each service

\(^5\)Assume two services $s_1$ and $s_2$ are advertised with access bandwidths of $bw_1$ and $bw_2$, then we can estimate that the end-to-end bandwidth is upper-bounded by $\min(bw_1, bw_2)$. At the application layer, such estimated values are sufficient, as the obtaining of precise information is very costly and sometimes unnecessary.
can then be advertised \textit{individually} with the associated machine resources and access bandwidth conditions, which has a complexity of $(O(N_q))$. At this point, the space complexity required to represent the state of a cluster would be $O(N_sN_q)$.

We represent the resource conditions associated with each service type in a bi-dimensional $mr$-$bw$ plane as shown in Figure 4.14. In order to further reduce the space complexity associated with a service (which is $O(N_q)$), we decide to aggregate the data into a small number of clusters. Before aggregation, since it does not make sense to advertise those service instances whose resource conditions are very low, we define a QoS-weak region (delimited by a bandwidth threshold and a machine resource threshold) in which covered instances are not advertised. Aggregation and advertisement only apply to the points that lie outside of the QoS-weak region.

Once the clusters have been detected, we represent each data cluster as a triple $(c, r, z)$, with $c$ representing the centroid of the data cluster, $r$ representing a radius, and $z$ representing the number of instances covered in the area.

- \textit{Reduced Data Points/Data Filtering:} Alternatively, we can choose to represent a data set by choosing to represent only a subset of data that would be most meaningful, instead of trying to reduce the granularity of the data set. Consider two points in a resource plane: $P_1 = [mr_1, bw_1]$ and $P_2 = [mr_2, bw_2]$. We say that $P_1 \preceq P_2$ if $mr_1 \leq mr_2$ and $bw_1 \leq bw_2$, $P_1 \succeq_{mr} P_2$ if $mr_1 \leq mr_2$ and $bw_1 \geq bw_2$, and $P_1 \preceq_{bw} P_2$ if $mr_1 \geq mr_2$ and $bw_1 \leq bw_2$. We say that a point $P$ is hidden if there is another point $P'$ in the resource plane such that $P \preceq P'$. The cluster may choose to advertise $P'$ and omit $P$. Given a set of data points, it requires certain algorithm or heuristics to choose a meaningful subset of data points as representatives.
Figure 4.14: (a) Sample resource plane; (b) organized resource plane: shaded area represents QoS–weak area; service instances that fall beyond the shaded area are clustered.

**Service Collocationness**

As mentioned in Chapter 3, in service routing, the available bandwidth between two communicating services depends greatly on whether the services locate on different nodes or on a single node; in the former case, the available bandwidth between services is the available bandwidth between the network nodes, while in the latter case, the available bandwidth can be considered as the cpu-memory bandwidth in the machine, which is far greater than network bandwidths. However, by advertising services as individuals, we have lost such important information.

Depending on the method of representation, fully representing such information can be expensive. In order to reduce the required space complexity, we use a Bloom filter, which is a simple space-efficient randomized data structure for representing a set in order to support membership queries at the cost of introducing false positives.

The Bloom filter, conceived by Burton H. Bloom in 1970 [58], is a space-efficient probabilistic data structure that is used to test whether or not an element is a member of a set. False positives are possible, but false negatives are not. Elements can be added to the set, but not removed (though this can be addressed with a counting filter). The more elements
that are added to the set, the larger the probability of false positives.

An empty Bloom filter is a bit array of \( m \) bits, all set to 0. There must also be \( k \) different hash functions defined, each of which maps a key value to one of the \( m \) array positions. To add an element, feed it to each of the \( k \) hash functions to get \( k \) array positions. Set the bits at all these positions to 1. To query for an element (test whether it is in the set), feed it to each of the \( k \) hash functions to get \( k \) array positions. If any of the bits at these positions are 0, the element is not in the set - if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements. Assume that a hash function selects each array position with equal probability. The probability of false positives is \( (1 - e^{-kn/m})^k \) [59]. Bloom filter has had many applications in the networking research in the past few years [60].

In our case, by modeling the data information as a set in which elements are pairs of co-residing services, we will also be able to use a Bloom filter to represent the data at a very low cost. We say that an element \((s_i, s_j)\) is in the set if \( s_i \) and \( s_j \) reside on any of the machines in the local cluster. An empty Bloom filter is a bit array of \( m \) bits, all set to 0. We define \( k \) different hash functions, each of which maps a key value \((s_i, s_j)\) to one of the \( m \) array positions. Adding an element will cause \( k \) array positions (computed by the \( k \) hash functions) to be set to 1. To query for an element (test whether it is in the set), we test if the \( k \) array positions are all set to 1. If any of the bits at these positions are 0, the element is not in the set. If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements (false positive).
Figure 4.16: An example of cluster-level service dag and information on how to label the link values.

### 4.3.3 Hierarchical Service Path Computation

Having nodes maintaining a semi-global system state, they can then jointly compute service paths in a top-down divide-and-conquer approach in a way similar to that shown in solution A. Since service path computation is more complicated at cluster level due to state aggregation, we will only concentrate on the discussion of inter-cluster service path computation. Intra-cluster service path computation remains the same as that described in Chapter 3.

#### Inter-Cluster Service Path Computation

Based on the service capability information advertised by the clusters, we first construct a service DAG as described in solution A. We determine whether or not a node (representing a network cluster in this case) is potentially resource-satisfied by predicting the number of QoS-satisfied service instances ($\varepsilon$) inside, which is calculated as the multiplication of its cardinality and the percentage of the cluster area that falls into the QoS-supported region$^6$. It is reasonable that the confidence of a cluster containing a concrete QoS-satisfied instance is directly proportional to the value of $\varepsilon$.

Assuming the service dag shown in Figure 4.16. We concentrate on determining the

---

$^6$We call the region right and above the minimum resource requirement lines QoS-supported region, and all other regions QoS-unsupported regions.
QoS-supportedness of the nodes at the cluster level. The figure on the right depicts a subportion of the figure on the left. Let Figure 4.17 depict the QoS conditions of the three cluster-level service nodes. We say that a service node is QoS-satisfied if \( \varepsilon \) is higher than a defined threshold. The higher \( \varepsilon \), the higher the confidence about the service node. Nodes with \( \varepsilon < \text{threshold} \) should be screened out from the service dag. Our goal is to compute a shortest service path over the QoS-satisfied service dag.

The links will be labeled with distance information in the following way. If the two end nodes belong to two different clusters, then the labeled distance equals the distance between the two clusters. On the other hand, if the two end nodes belong to a single cluster, then the distance is labeled based on whether or not the adjacent services pair can be found in the Bloom filter as follows: (1) 0 if \((s_i, s_j) \in BF\), i.e., if \(s_i\) and \(s_j\) potentially reside on a single node, and (2) \(d_{\text{internal}}(C)\) if \((s_i, s_j) \notin BF\), i.e., if \(s_i\) and \(s_j\) do not potentially co-locate, then their distance should be the average distance among intra-cluster nodes.

### 4.4 Conclusions

This chapter has dealt with most of the challenges encountered in a hierarchical service management solution, which span from topology formation, to QoS state aggregation/distribution, and to divide-and-conquer service composition and maintenance.
Chapter 5

Scalability in Network Size Aspect - Distributed Approach

An alternative routing solution that is scalable in network size is the distributed approach, by having nodes maintain states of limited neighborhoods (as opposed of maintaining semi-global states in the hierarchical approach), and compute routes hop-by-hop, from source to destination.

In this chapter, we investigate the main issues associated with adopting such an approach and provide a geometric-location guided hop-by-hop service routing computation which, in addition to considering resource conditions of candidate service nodes within one hop, will also take into account the global service path length.

5.1 Foundations

Below we present foundations that support our solution. More detailed descriptions will be followed in later sections.

5.1.1 Service Discovery with Geometric Location Awareness

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 will be retrieved by enhanced service discovery engines) and using it to estimate Internet
distances (communication delays) between hosts. The relative geometric coordinates of nodes in a large network can be efficiently obtained by the Global Network Positioning (GNP) approach [15] as described in Chapter 4. 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.

5.1.2 Topology Setup and Routing State Obtainment

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, and a service path is built for each application scenario. Routing state is measured on-demand. Upon receiving a request for a service path, a network node initiates certain probing activities to learn about the resource conditions of its associated neighbors as well as the link conditions in between¹.

5.1.3 Routing Approach

We adopt a hop-by-hop approach to computing service paths 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 first discovers 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.

¹Methods for measuring end-to-end available bandwidths can be found in [42, 43].

77
5.2 Hop-by-Hop Service Path Computation

In traditional routing, hop-by-hop QoS 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 [61]. Usually MPR is done by multiplying 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 multiplied at outgoing links. MPR may find better paths than SPR, but at the cost of higher message overhead. To minimize the overhead spent on probing, we adopt an SPR approach. 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 [62]. In data routing there is always the shortest network path (maintained by, e.g., the distance vector or link state protocol) that serves as guidance 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. Therefore, the simplest way of computing a service path hop-by-hop is to only consider local resource conditions and disregard the concern of overall path length, as described below.

5.2.1 Local-Heuristics-Based (LHB) Approach

Existing SPR-based hop-by-hop unicast QoS service routing approach [28] works as follows: starting from the source, the current node selects, among all probed service neighbors, the one whose aggregated 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 5.1(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 $s$ ($s/p_i$ and $s/p_j$) are equally good in terms of network bandwidth and machine capacity, for being unaware of the relative location of the two service instances, $p_s$ may choose $s/p_i$, which is off the way to destination.

### 5.2.2 LHB Enhanced with Geometric Location Guidance

The weakness of *LHB* can be remedied if we enhance the service discovery system and let it return also the geometric location information of the queried service instances. By doing so, we can let the current node select the 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 5.1(b): $p_s$ chooses $s/p_j$ because $p_j$ 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 5.2, 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
Figure 5.1: (a) Based on LHB, $p_i$ may choose a next hop that is not on the way to destination; (b) For making the routing aware of the hosts’ geometric locations, $p_i$ chooses a next hop that is on the way to destination.

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 5.2(a), knowing $p_1$ and $p_2$ are hosts in which $s_1$ resides, $p_i$ probes available end-to-end bandwidth and delay to $p_1$ and to $p_2$, as well as available machine capacities of $p_1$ and $p_2$. Based on the probing results, $p_i$ 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 $p_i$, suppose both instances have sufficient resources, $p_i$ then applies a shortest paths algorithm [55] 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 on the way to future service instances and the destination (Figure 5.2(a’)). In Figure 5.2(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 5.2(b’) shows how $p_1$ chooses the most delay-efficient and QoS-satisfied next service hop. Note that in this case, the probed bandwidth between $p_1$ and $p_3$ does not meet the requirement, thus the correspondent link is 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 ap-
Figure 5.2: Finding a QoS-satisfied and potentially shortest service path hop-by-hop from $p_s$ to $p_d$ that satisfies the service graph $s_1 \rightarrow s_2 \rightarrow s_3$. 
proaches: (a) **LHB-GLG**: applies a shortest paths algorithm [55] 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. Approach LHB-GLG actually resembles the *widest-shortest* approach, and approach GLG-LHB resembles the *shortest-widest* approach in traditional routing. 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.

### 5.2.3 Routing Backtracking

SPR-based hop-by-hop routing may end up in an unsuccessful state even if there exists a qualified path. 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). We consider back-tracking to immediately-previous node if the current node/link yields unsatisfactory performance quality.

### 5.3 Performance Study

We implemented the service routing framework in the well-known network simulator *ns-2*. This section is devoted to performance studies of the proposed approaches.

#### 5.3.1 Evaluation Methodology

Our physical Internet topologies are generated by the *transit-stub* model [45], 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 $a$ to another overlay proxy node $b$ is the bottleneck bandwidth of the shortest physical path from $a$ to $b$. 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 $L$[15]. We use geometric space of 5 dimensions in our simulations; calculation of geometric coordinates is done by using the software available at http://www-2.cs.cmu.edu/~eugeneng/research/gnp/.

We use the following performance metrics for the evaluations of routing approaches (performance metrics of other issues such as adaptation and failure recovery are described later in the result sections):

- **Host Utilization**: is the ratio of amount of machine resources in use to the machine’s total amount of resources. In simulations, we represent a machine’s computing capacity as a single numerical value, although in reality, it should be a resource vector of multiple parameters (e.g., memory, cpu).

- **Link Utilization**: is the ratio of used bandwidth to the total bandwidth of the physical network links that measures how much the physical links are loaded.

- **Service Path Length**: is the sum of individual virtual link lengths that make up the service path, where the virtual link lengths are represented as end-to-end delays.

- **Delay × Bandwidth Product**: The purpose of this metric is to measure the volume that the streaming data occupies in the network. For example, if the streaming data requires 2MB of bandwidth on a physical link whose single trip delay is 10ms, then the volume of data is said to be 20MB*ms.

- **Path Finding Success Rate**: is the rate of finding service paths successfully. Service path finding failures may occur when resources are scarce, or when there is no instance of the required service(s). However, in our following tests, there will be always at least

83
one instance of each service in the system, thus failures can only be caused by resource scarcity.

5.3.2 Performances of Different Service Path Finding Approaches

In this section, we measure performances of the service path finding approaches (GLG, LHB, GLG-LHB, and LHB-GLG) described in Section 5.2. 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 for the evaluations: host utilization, link utilization, service path length, delay × bandwidth product, and path finding success rate.

The simulation 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 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 5.3. 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

\footnote{For visibility, link utilization and proxy utilization are plotted as a transformed inverse cumulative distribution function, also known as inverse survival function. Service path length is plotted as an inverse cumulative distribution function.}
Figure 5.3: 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.
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 5.3 (c)) and service path length (Figure 5.3 (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 5.3(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 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.

86
5.4 Conclusions

In this chapter, we have improved an existing local-heuristics-based hop-by-hop approach by incorporating the geometric location guidance in the hop-by-hop path computation. The result is that, as has been verified by our simulation tests, service paths will be significantly shorter and consume less network resources. Our geometric-location-enhanced hop-by-hop solution will serve us as a basis for computing service multicast trees incrementally in the next chapter, when we deal with the scalability issue in application size aspect.
Chapter 6

Scalability in Application Size Aspect

As can be observed from the scenarios in Figure 1.3, if an application involves sending data from a single source to multiple destinations, then establishing individual service paths through which the data is transformed and distributed would be a waste of resources, even if the destinations’ service requests do not match exactly. We propose a novel type of multicast called *service-added multicast* that tries to explore partial or total matching of service paths. The idea is that if two service requests have a nonempty prefix in common, then part of their service paths can be merged so that both network bandwidth and machine computational resources can be saved.

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: the service multicast tree is built incrementally based on the unicast service routing solution described in Chapter 5.

A key concept in multicast-related problems is the *graftable on-tree node* concept. For example, the PIM protocol first tries to find an on-tree node for the newly joining member, $p_m$, by forwarding $p_m$’s joining request along the shortest path towards the source to see if the request hits some on-tree node before reaching the source. Once an on-tree node has been identified, a branch starting from this on-tree node and ending at $p_m$ is constructed. The new member $p_m$ is said to be *grafted* on to the on-tree node. Unlike the conventional data multicast, where every on-tree node functionally qualifies as a graftable node for all
other group members, in service multicast, due to the service constraints, an on-tree node \( p_g \) would only qualify as a graftable node for a member 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 \( SR \).

6.1 Service Multicast Tree Construction Algorithms

In this section, we propose two algorithms, \( \text{Optimal-Service-Paths Tree (OSPT)} \) and \( \text{Longest-Prefix Tree (LPT)} \), for building service trees incrementally, and study their performances.

6.1.1 Algorithms

1. \( \text{OSPT (Optimal-Service-Paths Tree)} \): In OSPT, each individual end-to-end path is an optimal service path computed by the solution presented in Chapter 3, and the merge of these individually optimal service paths forms an \( \text{Optimal-Service-Paths Tree (OSPT)} \). This approach resembles the shortest-paths tree approach used in data multicast. A disadvantage of such trees is that they may not have best overall performances, because path sharing is not the primary goal and only happens as a “side effect”. An advantage of \( \text{OSPT} \) is that at the time path sharing is being explored, no individual would be penalized in favor of overall performance. Figure 6.1(a) depicts an example of building an \( \text{OSPT} \). For clarity purposes, the QoS issues are not shown in the figure. However, all individual service paths are constructed under the up-to-date resource conditions, and the final paths satisfy also the QoS requirements.

2. \( \text{LPT (Longest-Prefix Tree)} \): Different from \( \text{OSPT} \), where path sharing only occurs when individually sought service paths happen to merge at certain point, in the \( \text{LPT} \) approach, we enforce path sharing as follows. Whenever a new member \( p_m \) joins, an explicit search for a graftable on-tree node is firstly performed; if any graftable node \( p_g \) is identified, then a service branch satisfying the suffix of the current service request (to be clearer later) is added between \( p_g \) and \( p_m \). As has been stated, an on-tree node
Figure 6.1: (a) OSPT: (a.1) optimal service path for $SR_1 = \langle p_0, s_1, s_2, s_3, p_5 \rangle$; (a.2) optimal service path for $SR_2 = \langle p_0, s_1, s_2, s_3, s_4, p_3 \rangle$; (a.3) merging the two service paths to form an OSPT; (b) LPT: (b.1) optimal service path for $SR_1 = \langle p_0, s_1, s_2, s_3, p_5 \rangle$; (b.2) optimal service path for $SR_{suffix} = \langle p_5, s_3, s_4, p_3 \rangle$; (b.3) attaching (b.2) to (b.1) at the graftable node of longest prefix $p_5$ to form an LPT; (b.4) optimal service path for $SR_{suffix} = \langle p_1, s_2, s_3, s_4, p_3 \rangle$; (b.5) since $p_5$ does not have enough bandwidth in its outgoing link, attach (b.4) to (b.1) at the graftable node of second longest prefix $p_1$ to form an LPT'.

$p_g$ only qualifies as a graftable node for a member 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 $SR$. In order to maximize service path sharing, we search for an on-tree node whose up-tree service path is the longest prefix (among all candidates on the service tree) of $SR$. Since a prefix of $SR$ has been satisfied by $p_g$, we only need to find a service branch for the corresponding suffix of $SR$ from $p_g$ to the joining node. A service tree constructed using this approach will be named a Longest-Prefix Tree (LPT). Figure 6.1(b) depicts an example of LPT; the second service branch is attached to the graftable node of longest prefix. Due to network bandwidth limitations, it is possible that the found graftable node does not have any bandwidth available in its outgoing links. When this happens, the search for a graftable node should go up one level in the service tree and consider the second longest prefix. Figures 6.1(b.4) and 6.1(b.5) depict such a case. Bandwidth limit thus naturally imposes degree constraints in constructed service trees.
OSPT and LPT in service multicast resemble the Shortest-Paths tree and Steiner tree, respectively, in data multicast. OSPT and Shortest-paths tree are both a combination of individually optimal paths, and sharing only happens if the paths happen to merge; while LPT and Steiner tree explicitly seek path sharing by attaching the newly joining member to a suitable on-tree node. The major difference between OSPT and LPT is that in the former, individual service paths do not get penalized in favor of global tree optimization, while the latter may penalize an individual to achieve global optimization.

6.1.2 Performance Evaluation

We perform simulation tests in ns2 to measure performances in several aspects. The physical topologies used in the tests all follow the Transit-Stub model described in [45]. We concentrate on studying performances related to tree establishment.

Our main objective in this study is to learn potential advantages of service multicast over service unicast, and see performance differences between the two service multicast tree building approaches. We also compare the performances of our incrementally built service trees with those of the Steiner service tree sought from doing exhaustive searches. Path computations seek to optimize delay, but guarantee satisfactions of bandwidth and machine resource needs. In the following tests, the service overlay network consists of 40 proxies, with each one assigned 2 to 5 services (in a pool of 20) as its locally available services. The physical topology consists of 100 nodes. A number of client nodes randomly generate and submit 100 service requests to a single proxy node (the root proxy). The same overlay configuration is run over two different physical networks.

We study and compare the performances of the algorithms in aspects of: total service paths/tree cost, average end-to-end path delay, and resource consumptions. Figure 6.2(a) shows the performance results of the algorithms in terms of total cost of the built service paths/tree. Both multicast algorithms outperform UNICAST significantly. Between the multicast algorithms, OSPT's total tree cost is slightly higher because, as mentioned pre-
Figure 6.2: (a) Total service path/tree costs of the algorithms in terms of delay; (b) Average end-to-end service path delay incurred by each algorithm.

Figure 6.3: (a) Proxy machine resource consumption; (b) Bandwidth consumption on physical network links (for clarity purposes, those links whose load is zero are not shown).
viously, in OSPT, path sharing only occurs as side effects. Performance of our incremental tree construction algorithms, OSPT and LPT, without any adaptation, incur about 30% and 22-24%, respectively, higher global cost than the optimal case. However, If we allow adaptation in LPT, as described further in Chapter 7, at 0 threshold, its performance is comparable to that of OPTIMALM.

Figure 6.2(b) shows the average end-to-end (root-to-leaf) path delays of the algorithms. As both UNICAST and OSPT construct shortest end-to-end service paths, these two perform best. LPT incurs longer end-to-end service paths because a single member’s performance may be compromised in favor of global tree performance. Compared to LPT, OPTIMALM incurred slightly less end-to-end service paths.

Figure 6.3(a) shows machine resource consumptions on each proxy, and Figure 6.3(b) shows bandwidth consumptions on each physical network link (for one of the test cases). Both multicast algorithms outperform the unicast algorithm, and between the two multicast algorithms, LPT is better, which is consistent with our previous analysis. Again, compared to LPT, OPTIMALM incurred slightly less end-to-end service paths.

6.2 Distributed Construction of Service Multicast Trees

In this section, we will investigate a distributed way of constructing a service multicast tree using the LPT algorithm derived in Section 6.1.

6.2.1 Design Overview

We describe the most salient features of our solution framework below:

- **Hop-by-hop routing based on resource conditions and geometric location information:** Without maintaining full state information, network nodes will jointly
compute service paths in a hop-by-hop manner. Routing takes resource conditions as well as geometric location information of the Internet hosts into consideration. More descriptions can be found in Chapter 5.

- **Foreground topology and background topology:** To minimize delays, a service path or tree should be built without involving relay nodes; that is, network nodes that do not contribute special functionality to the application should be by-passed. Therefore, service paths/trees are built on top of an unstructured topology. However, we maintain a separate structured topology for background communications (i.e., for control messages). 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 [37], because Delaunay triangulation is a spanner graph that possesses some nice properties\(^1\) and method of incremental construction of Delaunay triangulation network has been derived [37].

By using such a geometric topology, control messages can be routed by using an online, local routing method, such as the greedy approach or compass routing approach [63]. A local routing method has the advantage of not requiring global information of the topology in order to route from source to destination. For example, in compass routing, all information available at any point of routing is: the coordinates of the destination, the current position, and the directions of the outgoing links from the current node. Compass routing takes the following approach: starting at source, the current node chooses the outgoing link with the closest slope to that of the line segment connecting the current node to the destination. It has been proven that Delaunay triangulation D supports compass routing, which means that for every pair of nodes in D, compass routing always produces a path from source to destination [63]. Note

\(^1\)A path found within a Delaunay triangulation has length bound by a constant times the straight-line distance between the endpoints of the path.
that the adoption of Delaunay triangulation for overlay network topology and compass routing for control messages is only a choice in our design, for simplicity and efficiency purposes. Alternative network topologies and routing methods may be used as well.

- **Incremental multicast tree construction:** We construct a service multicast tree incrementally based on the unicast service routing solution. Moreover, we seek to maximize resource sharing by integrating data multicast into service multicast, thus providing a hybrid multicasting solution.

### 6.2.2 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 later) to identify the best functionally graftable service node \( s_i/p_\theta \). The current node \( p_i \) then forwards the request to \( p_\theta \) if \( p_\theta \neq p_i \). With a prefix of \( SR \) satisfied at point \( p_\theta \), \( p_\theta \) calculates the suffix of \( SR \), and starts a hop-by-hop routing process (by using a unicast service routing solution described in Chapter 5 towards destination 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 grafted 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 6.4 depicts an example of how a service multicast tree is built and managed. In Figure 6.4(a), assume \( p_{d1} \) is the first group member. After \( p_{d1} \) has joined, the on-tree proxy nodes \( p_1, p_2, p_4, p_7 \), and \( p_{d1} \) will obtain a copy of the functional service tree - \( T_f \) - depicted on the right side of Figure 6.4(a). When \( p_{d2} \) joins, a service request \( SR_2 = (p_1, s_1 \rightarrow s_2 \rightarrow s_4, p_{d2}) \) is sent from \( p_{d2} \) towards the source by using compass routing. The request hits an on-tree node \( p_1 \) before it reaches \( p_1 \). Since \( p_1 \) has a copy of \( T_f \), it finds that \( p_4 \) is the best functionally grafted node for the current request, thus forwarding the request to \( p_4 \). In Figure 6.4(b), a service branch is established hop-by-hop from the grafted node \( p_4 \) to \( p_{d2} \). Since the grafted 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 6.4(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 \( SR_3 = (p_1, s_1 \rightarrow s_2 \rightarrow s_3, p_{d3}) \), then \( p_{d3} \) can get attached to \( p_7 \) 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
Figure 6.4: (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 $P_{d2}$.

expect the length of a service branch satisfying only the suffix of the request to be shorter than an individually built service path that needs to satisfy the whole request.

### 6.2.3 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 6.5(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,
Figure 6.5: (a) Pure service multicasting; (b) hybrid multicasting (service multicasting + data multicasting).

taking Figure 6.5(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 6.5(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 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
Figure 6.6: Exploring data multicast in a service multicast scenario: (a) a new service branch’s first node, $p_8$, is initially directly attached to the graftable service node $p_4$ ($p_4$ as $p_8$’ parent in the local data distribution tree); (b) $p_8$ gets parent-switched to $p_7$ in the data distribution tree.

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 \( p_y \), so that \( p_y \) 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 6.6 depicts what the global functional service tree and the local data distribution tree would look like in the scenarios. In Figure 6.6(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 6.6(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 6.6(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.

**6.2.4 Performance Study**

We implemented the service routing framework (including service unicast, pure service multicast, and hybrid multicast delivery modes) in the well-known network simulator *ns-2*. This section is devoted to performance studies of the proposed approaches.
Evaluation Methodology

Our physical Internet topologies are generated by the \textit{transit-stub} model \cite{45}, 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 $p_i$ to another overlay proxy node $p_j$ is the bottleneck bandwidth of the shortest physical path from $p_i$ to $p_j$. 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 $L[15]$. We use geometric space of 5 dimensions in our simulations; 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).

The parameters that we use for the performance evaluation are the same as those described in Chapter 5 (in Section 5.3).

Service Unicast \textit{vs} Pure Service Multicast \textit{vs} Hybrid Multicast

In this section, we study the performance benefits of employing pure service multicast and hybrid multicast. Since \textit{LHB-GLG} is a service unicast approach that strikes best balance among the performance metrics, we employ \textit{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 6.7 (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 6.7 (b) shows that hybrid
Figure 6.7: Comparisons of: (a) host utilization; (b) physical link utilization; (c) end-to-end service path length; (d) global host utilization; (e) delay bandwidth product and; (f) global service path/tree cost among the different delivery modes: service unicast, pure service multicast, and hybrid multicast.
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 6.7 (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 6.7 (c)). However, the longer end-to-end service paths in hybrid multicast are justified by lower global tree costs (Figure 6.7 (f)) due to service path sharing.

6.3 Conclusions

Exploring scalability by means of multicasting allows network resources to be better conserved through resource sharing. This is a fact evidenced in IP multicast since late 80’s [30]. At overlay networks, the emerging distributed component service model requires totally new algorithms and protocols for constructing trees, as we have seen in this chapter. The performance results show the comparative advantages of service multicasting (hybrid multicasting) over service unicasting under circumstances defined in this chapter.
Chapter 7

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. 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 in the service management framework. The following discussions rely on the same environment assumed in Chapters 5 and 6, in which service trees are constructed incrementally using a distributed hop-by-hop approach.

7.1 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 [64]. 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\(^1\) 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 \(sn_x\), we define the potential performance improvement as \(\gamma = \frac{c(sn_x) - c(sn_y)}{c(sn_x)}^2\). 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 \(sn_x\) as \(\theta\), and the utility associated with the replacement as \(\mu\). We therefore define the real benefit \(\beta\) of replacing \(sn_x\) by \(sn_y\) as a function of performance gain, disturbance, and utility: \(\beta = \gamma \times \frac{\theta}{\mu}\), and replacement only takes place if \(\beta\) is larger than 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 \(sn_x\) will hand over its roll to \(sn_y\). It is important that parent and child service nodes do not perform

---

\(^1\)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.

\(^2\)The term \(c(sn_x)\) is defined as the sum of \(sn_x\)'s neighboring link costs \((c(sn_x))\), which are measured as delays.
Figure 7.1: 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.

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 7.1. 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 then the threshold, then \( p_4 \) will hand over its roll to \( p_3 \).

7.2 Failure Recovery

Traditional failure recovery mechanisms used in routing fall into two approaches: protection-based approach and restoration-based approach [65]. 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
Figure 7.2: 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 $p_1$ triggers failures of multiple service nodes in the tree; right: The failures are individually detected by $p_2$, $p_3$, and member 2; (c) left: independent failure reporting; right: lazy failure reporting that back-off failure reporting if possible.
made to reroute the path around the faulty nodes and links. The protection-based approach has been adopted in [66, 28] 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 dissertation, 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 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 7.2. Figure 7.2(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 7.2(a).
Failures of some nodes (e.g., $p_2$ and $p_3$) 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 (detectable by member 1), then member 1 will try to locate a live ancestor closest to itself (in this case $p_1$) and afterwards $p_1$ initiates a hop-by-hop routing process towards member 1 to recuperate service $s_3$. Failure of $p_1$ 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: $p_2$, $p_3$, and member 2, which report to their closest live ancestors - the root and the node $p_3$ 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) $p_2$'s request precedes $p_3$'s - once $p_2$'s request has been satisfied, $p_3$'s request can be ignored (because $p_3$'s request is part of $p_2$'s request); (2) $p_3$'s request precedes $p_2$'s - $p_2$'s recovery request can only be initiated after $p_3$'s request has been satisfied, because the recovered service node (say $s_1$ is mapped to $p'_1$) $s_1/p'_1$ will serve as a reference point for part of $p_3$'s request. Furthermore, $p_3$'s recovery request will be initiated by $p'_1$ 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 7.2, 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.

### 7.3 Performance Study

The solutions proposed in this chapter are, again, implemented in the network simulator *ns-2*.

#### 7.3.1 Evaluation Methodology

Evaluation of our tree rearrangement and failure recovery algorithms are based on service trees constructed hop-by-hop and incrementally in the previous Chapter. Therefore, the evaluation methodology mostly remains the same, except that in the following simulation environment, some node failures will be artificially introduced.

#### 7.3.2 Tree Rearrangement

The local tree rearrangement operations together contribute to a global tree quality improvement. As described in Section 7.1, 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 $\beta = \gamma \cdot \frac{\theta}{\mu}$ cancel off.

Figure 7.3 depicts the total tree costs (in logical units) after adaptations versus local
adaptation thresholds with different service distribution probabilities$^3$. The experiment set-
tings were as follows: similar to the settings described in Section 6.2.4, 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.

7.3.3 Failure Recovery

We compare performances of three failure reporting and recovery approaches as described
in Section 7.2: independent reporting and independent recovery (IRIR), independent report-
ing 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 con-
ducted as follows: still using multicast group size of 256 and binary functional tree as before,

$^3$Service distribution probability $x$ means that each component service is randomly distributed at $x\%$
network nodes.
Figure 7.3: Total tree cost after adaptations vs local adaptation threshold with different service distribution probabilities.

after the service multicast tree has been built, we randomly failed on-tree network nodes one by one. The results shown in Figure 7.4 are based on 3 runs, each run with 20 node failures. The numerical 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.

7.4 Conclusions

In this chapter, we have derived local performance monitoring and adaptation, as well as distributed failure detection, reporting, and recovery mechanisms. The solutions are a part of the framework described in Chapters 5 and 6, which make the service composition resilient
Figure 7.4: Performance comparisons among IRIR, IRHR, and LRHR: metric 1 is the number of failure reports generated, metric 2 is the global service tree cost after the recovery completes, and metric 3 is the time needed for the recovery to complete. The numerical values have been normalized based on the results of IRIR (base case) in faces of resource fluctuations and failures.
Chapter 8

Conclusions

As we can see, to make the component services infrastructure transparent to the application layer, service management should deal with a series of issues, including (dynamic) topology setup, state obtainment, aggregation and distribution, and service path/tree computation and maintenance. While most of the previous work has concentrated on studying the service management problem for one-to-one applications in small network environments, we have taken a step further to investigate solution scalability, as well as adaptivity and robustness in this dissertation.

8.1 Contributions

In summary, we systematically studied the service management issues from simplest case (one-to-one application in small networks) to more complex cases (larger networks and one-to-many group-based application scenarios). Describing the evolution of problem challenges and solutions allowed us to better understand our design purposes and design alternatives for the same problem in same or different application environments. Overall, the contributions of this dissertation can be listed as follows:

- For a small network and one-to-one application environment, We have derived a centralized service management solution to support the distributed component services model seamlessly and efficiently. We identified some important QoS metrics and key issues that distinguish service-added QoS routing from the traditional QoS routing. By adding resource normalization, graph mapping, and backtracking resource verification
processes, we were able to make the two problems similar. We derived an aggregated QoS metric $\mathcal{F}$ such that optimizing $\mathcal{F}$ amounts to optimizing the individual QoS metrics. Our performance results verified the soundness of this aggregated metric.

- As stated, the focus of this dissertation has been on providing solution scalability, adaptivity, and robustness. We have tackled the problems as follows:

  - **Scalability:** Scalability has been investigated in two dimensions: network size and application size. In terms of network size, we wanted to be able to manage services (both as individuals and as composites) when the network size is so large that maintaining global system status is not practical. In terms of application size, we wanted to allocate system resources wisely to group of users, so that through sharing, resource usage can be reduced significantly.

    * **Scalability in network size aspect:** when the network becomes too large such that centralized service management relying on global knowledge of the network becomes impractical, we investigated two scalable approaches: hierarchical and distributed, which are two rules-of-thumb approaches adopted in networking and distributed systems when scalability becomes the concern.

    * **Hierarchical approach:** The purpose of the hierarchical approach is to organize a large network into smaller clusters/groups, so that topology abstraction and state information aggregation become possible to significantly reduce state maintenance overhead. Many challenges are associated with this approach, including: (1) identification of clusters at the application layer and management of dynamic membership of network nodes, (2) proper aggregation and distribution of routing information, and (3) computation of efficient service paths in a hierarchically structured network topology. We identified clusters of a network by first obtaining its distance map in a geometric space, and then running MST-
based or Square-Error clustering algorithms to detect initial clustering. We further devised low-overhead local reclustering operations to absorb network dynamics while keeping a good clustering quality. QoS aggregation has been performed with two mechanisms, data clustering/filtering and Bloom filter, which would help reduce the amount of data to be advertised among clusters. In such an environment where network nodes maintain a semi-global status of the system, we devised a top-down divide-and-conquer approach for computing efficient service paths.

- distributed approach: An alternative routing solution that is scalable in network size is the distributed approach, by having nodes maintain states of limited neighborhoods (as opposed of maintaining semi-global states in the hierarchical approach), and compute routes hop-by-hop, from source to destination. We investigated the main issues associated with adopting such an approach and provided a geometric-location guided hop-by-hop service routing computation which, in addition to considering resource conditions of candidate service nodes within one hop, also took global service path length into consideration. The benefit of considering easily maintainable geometric location information of the network nodes was that, as has been verified by our simulation tests, service paths were made significantly shorter and consumed less network resources.

* Scalability in application size aspect: When an application involves sending data from a single source to multiple destinations, then establishing individual service paths through which the data is transformed and distributed would be a waste of resources, even if the destinations’ service requests do not match exactly. We proposed a novel type of multicast called service-added multicast that tried to explore partial or total matching of service paths. Two algorithms for constructing service multicast trees: Optimal-Service-Paths Tree
(OSPT) and Longest-Prefix Tree (LPT), were proposed and studied against the UNICAST and OPTIMAL\textsubscript{m} cases. Further performance improvement was possible with the integration of data multicast into the service multicast solution.

- **Adaptivity**: Due to network dynamics, e.g., resource fluctuation and application membership dynamics, the structure of a service multicast tree may become “distorted” over time, thus degrading its performance. For the continuous operation of the application at a good QoS level, we devised local adaptation mechanisms for a service multicast tree to improve itself gradually.

- **Robustness**: During the application runtime, resources may fail. We used a restoration-based approach which, on detection of a failure, would attempt to reroute the path around the faulty nodes and links. Different from traditional routing, in which failure of a node or link means only a single 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. We provided distributed failure detection, reporting, and recovery mechanisms to support solution robustness.

## 8.2 Future Work

There are many directions for future work in this topic, we list a few below:

- **Dynamic service deployment**: In this dissertation, we ignored the service deployment issue, and our service management assumed that services were pre-deployed. In practice, service deployment should cater to practical needs; with denser distributions in neighborhoods in which a service is likely to be instantiated more frequently, and sparser distribution in neighborhoods in which the service is likely to be instantiated less frequently. A service may start off being randomly replicated in a number of network nodes. After that, its usage pattern may be monitored and predictions on its
future usage can be made. Based on its current and predicted usage patterns, dynamic service distribution actions can take place.

- **killer applications and real software components:** How important the issues of this dissertation will be in our real life will largely depend on whether or not there will be killer resource-demanding applications that would benefit from the SOA model. We envisioned that SOA will be used in large, open networks (like the Internet) to solve the heterogeneity problem. We expect real transformational software components to be widely developed and deployed in popular networks (e.g., Akamai’s proxy network).
References


122


[57] Stan Salvador and Philip Chan. Determining the Number of Clusters/Segments in Hierarchical Clustering/Segmentation Algorithms. In 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI’04), Boca Raton, Florida.


Vita

Jingwen Jin was born and raised in Shangyu, Zhejiang, China. She went to Brazil with her parents and sister during high school, and received her education in Brazil until masters. She obtained her Bachelor of Science degree in Physics in December 1994 and Master of Science degree in Computer Science in December 1997, both from the Federal University of Pernambuco, Brazil. She came to the USA in August 1999, and since then has been a PhD student in the Department of Computer Science at University of Illinois at Urbana-Champaign. She joined Professor Klara Nahrstedt’s MONET (Multimedia Operating Systems and Networking) research group in August 2000. During her PhD study, she interned at Inktomi Inc. in the summer of 2001, and at IBM T. J. Watson in the summer of 2004. After graduation, she will join Intel’s Communications Technologies Lab as a research scientist.