SPIDERNET: A QUALITY-AWARE SERVICE COMPOSITION MIDDLEWARE FRAMEWORK

BY

XIAOHUI GU

M.S., University of Illinois at Urbana-Champaign, 2001
B.S., Peking University, Beijing, 1999

THESIS

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, 2004

Urbana, Illinois
Abstract

Internet has become a service provision infrastructure instead of merely providing host connectivity. Examples of Internet services include various multimedia services such as media streaming and web services such as on-line shopping. Indeed, application services have become one of the driving force of today’s information technology advancement. The user desires to high quality, failure resilient and ubiquitous Internet services. However, due to the problems pertaining to scalability, reliability, manageability, and cost-effectiveness, the traditional monolithic approach to application service provisioning has become inadequate. To address the problems, service composition has recently been proposed to provide a component-based compositional approach to next-generation application service provisioning.

Service composition allows future application services to be automatically composed from atomic service components, in various distributed computing environments, based on the user’s dynamic service requirements. Although previous research has addressed different aspects in service composition, existing solutions lack user desired scalability, flexibility and multi-constrained quality-of-service (QoS) support. Thus, one major challenge, which is the focus of this dissertation, is to provide quality-aware service composition, which can automatically compose distributed service components satisfying the user’s quality requirements (e.g., service delay, data loss rate) for the composed service in different distributed computing environment.

To address the challenge, we present SpiderNet in this dissertation, a quality-aware service composition middleware framework. SpiderNet providing quality-aware, failure resilient
application services by modularizing the application service provisioning and replicating service components. SpiderNet achieves quality-aware service delivery by properly selecting among replicated service components and composing them according to the user’s QoS requirements. SpiderNet delivers ubiquitous services by dynamically adapting service delivery in ubiquitous computing environments. SpiderNet adopts a hybrid architecture consisting of core SpiderNet domains and access SpiderNet domains. The core SpiderNet is constructed as a service overlay network in the wide-area network, which provides major Internet service functions. The access SpiderNet is deployed in edge networks, which is responsible for adapting service delivery in ubiquitous computing environments.

The major contributions of SpiderNet are as follows. First, SpiderNet introduces service overlay network model that connects previously scattered distributed service components into an application-level overlay network. Thus, SpiderNet can provide QoS-aware and failure resilient on-demand application services based on the states information of service components and the overlay links between them. Second, a centralized QoS-aware service composition and failure recovery solution is proposed for overlay-based infrastructure SpiderNet assuming the service composer the global states information about the entire service overlay network. Third, a distributed QoS-aware service composition and failure recovery solution is proposed for decentralized peer-to-peer service overlay networks. Fourth, a two-tier service composition model is proposed for pervasive computing environments such as smart rooms where service components can be dynamically uploaded. Evaluation and validation to our system design is not only analytical but also experimental. We conducted extensive experiments using both large-scale simulations and prototype implementation. We also implement a set of proof-of-concept distributed multimedia applications on top of SpiderNet. Our experimental results demonstrate the implementation feasibility and performance efficiency of the SpiderNet system.
To my parents and Zhen.
Acknowledgments

First and foremost, my deepest gratefulness goes to my thesis advisor, Professor Klara Nahrstedt. I feel extremely lucky to have Klara as my advisor, who is always dedicated to the success of her students. To me, she is not only a wonderful advisor but also a perfect role model as a great female computer scientist. I would like to sincerely thank her for providing me with invaluable consistent guidance and support during my whole graduate study. She has provided me with a perfect balance between guidance and freedom, which allows me to pursue my own ideas along the right direction. I will always be grateful for her consistent encouragement and her belief in me.

I would like to thank the honorable members of my thesis committee, Professors Gul Agha, Jennifer Hou, and William Sanders, for their helpful discussions on the ideas in my work. Their insightful comments during my Ph.D. Preliminary Exam have greatly helped me to improve my work. I am also indebted to them for careful reading my dissertation and providing valuable feedbacks.

I would like to extend my special thanks to Professor Roy Campbell from whom I have learned a lot. I have learned how to conduct systems research from him through both his advanced operating systems class and the exciting Gaia project. I also had the opportunities to work with Dr. Dejan Milojicic (Hewlett Packard Research Labs) and Dr. Rong N. Chang, Dr. Zon-Yin Shae (IBM T.J. Watson Research Center). I was grateful to them for the inspiring discussions that help me generate part of the ideas in this dissertation.

I am grateful for the support and assistance from all current and previous members of the Multimedia Operating Systems and Networking (MONET) research group, especially
Baochun Li, Dongyan Xu, Duangdao Wichadakul, Bin Yu, Jingwen Jin, Chui Sian Ong, Wanghong Yuan, Kai Chen, Jin Liang, Jun Wang, Li Xiao, Yuan Xue, Yi Cui, and Won J. Jeon. I would like to thank them for all the insightful discussions. I am indebted to them for both technical and moral support during my graduate study.

The work presented in this dissertation was supported by the NASA grant under contract number NASA NAG 2-1406, National Science Foundation grant under contract number 98-70736, National Science Foundation Career Grant under contract number NSF CCR 96-23867, NSF CISE Infrastructure grant under contract number NSF EIA 99-72884. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, NASA or U.S. Government.

Last but not least, I would like to thank my family members for their greatest love and caring. Without their continuous support, I could not have today’s achievements. I am, and will be deeply indebted to my parents during my whole lifetime. I am also indebted to my husband Zhen who is the true companion for me in both work and life. I wish I could thank them and this dissertation is dedicated to them.
# Table of Contents

List of Tables ................................................................. ix
List of Figures ................................................................. xii

Chapter 1  Introduction ...................................................... 1
  1.1 Motivation ................................................................. 1
    1.1.1 Emerging Internet Services ......................................... 2
    1.1.2 Why Service Composition ........................................... 3
  1.2 Research Challenges .................................................. 5
  1.3 SpiderNet: A Quality-Aware Service Composition Middleware Framework .... 7
    1.3.1 High Level Overview ................................................ 7
    1.3.2 Major Contributions ............................................... 8
  1.4 Outline of the Dissertation .......................................... 10

Chapter 2  SpiderNet System Model ......................................... 12
  2.1 Application Service Model ............................................ 12
    2.1.1 Service Component Model .......................................... 12
    2.1.2 QoS Consistency Model ............................................ 14
    2.1.3 Service Path: A Simple Composite Service Model ................. 17
    2.1.4 Service Graph: A Complex Composite Service Model ............... 19
  2.2 Hybrid Horizontal Architecture ..................................... 20
    2.2.1 Core SpiderNet ..................................................... 21
    2.2.2 Access SpiderNet .................................................. 22
  2.3 Layered Vertical Architecture ....................................... 22
    2.3.1 Abstract Composite Service Layer ................................ 22
    2.3.2 Instantiated Distributed Service Layer .......................... 24
    2.3.3 Service Overlay Network Layer ................................... 25
  2.4 Service Composition Protocol ....................................... 27
  2.5 Assumptions .............................................................. 30

Chapter 3  Utility SpiderNet: A Contract Based Core SpiderNet ............... 32
  3.1 Introduction ............................................................. 32
  3.2 Managed Service Overlay Network .................................... 33
  3.3 Contract Based Service Composition ................................... 34
  3.4 Initial Service Composition .......................................... 36
List of Tables

6.1 Topologies used in experiments. ........................................ 95
7.1 Comparisons among different service distribution algorithms. ................................ 122
### List of Figures

2.1 Service component model and service composition model. .......................... 13  
2.2 Different relations between two condition hypercubes. ............................ 15  
2.3 Service path: linear composition of service components. ........................... 18  
2.4 Service graph: directed acyclic graph composition of service components. .... 19  
2.5 Hybrid SpiderNet system architecture in a horizontal view. ........................ 20  
2.6 Illustration of SpiderNet system architecture. .......................................... 23  
2.7 SpiderNet service composition protocol. ................................................. 27  
3.1 Managed service overlay network. ......................................................... 33  
3.2 SLA-based composed service delivery. .................................................... 34  
3.3 Illustration of the QSC-basic algorithm. .................................................. 39  
3.4 Illustration of the QSC-enhanced algorithm. ............................................ 40  
3.5 Pseudo code of the QSC-enhanced algorithm. .......................................... 42  
3.6 Global service composition failure recovery. ............................................ 44  
3.7 Local service composition failure recovery. ............................................. 45  
3.8 Average availability QoS violation rate under different system load. ............ 48  
3.9 Average response time QoS violation rate under different system load. .......... 48  
3.10 Average availability QoS violation degree under different system load. ......... 49  
3.11 Average response time QoS violation degree under different system load. ...... 49  
3.12 Average provisioning success rate under different system load (load balancing). 49  
3.13 Average per session outage occurrences under different system load (stability). 49  
4.1 A mixing service overlay network. ......................................................... 54  
4.2 Mixing service component and mixing service path. .................................. 56  
4.3 Illustration of mixing service path composition. ....................................... 57  
4.4 Session success rate comparison. ........................................................... 59  
4.5 Scaling property comparison. ............................................................... 59  
5.1 Bounded composition probing protocol. .................................................... 65  
5.2 Per-hop probe processing operations. ...................................................... 68  
5.3 Proactive failure recovery for failure resilient service composition. ............. 74  
5.4 P2P SpiderNet node software architecture. .............................................. 77  
5.5 Service session setup time in wide-area networks. .................................... 79  
5.6 Performance comparison among random, P2P SpiderNet, and optimal algorithms. 79
5.7 QoS-aware service composition performance comparison among different approaches .............................................. 81
5.8 QoS-aware service composition performance on three mesh overlays with 200, 500, and 1000 nodes ............................................. 81
5.9 Probing overhead on three mesh overlays with 200, 500, and 1000 nodes ................................................................. 83
5.10 Total system overhead on three mesh overlays with 200, 500, and 1000 nodes ........................................................... 83
5.11 Failure frequency comparison in a dynamic P2P network ...................................................... 84
5.12 Failure resilience of P2P SpiderNet under P2P network topology variation rate 100 peers/time unit ........................................ 84

6.1 Average provision success rate of the GC algorithm under different workload on the Tmesh0.1 topology .................................................................................................................. 98
6.2 Average QoS success rate of the GC algorithm under different workload on the Tmesh0.1 topology ......................................................... 98
6.3 Average provision success rate of the distributed algorithm under different workload on the Tmesh0.1 topology ........................................................................................................ 98
6.4 Average QoS success rate of the distributed algorithm under different workload on the Tmesh0.1 topology .................................................................................................................. 98
6.5 Cumulative provision success rate comparison among the optimal, GC, random, and LD algorithms on the Tmesh0.1 topology with 150 requests/minute ......................................................... 99
6.6 Cumulative QoS success rate comparison among the optimal, GC, random, and LD algorithms on the Tmesh0.1 topology with 150 requests/minute ......................................................... 99
6.7 Failure frequencies on a dynamic overlay network with variation rate 10 peers/minute with and without reactive failure recovery ................................................................. 101
6.8 Failure frequencies on a dynamic overlay network with variation rate 10 peers/minute with and without proactive failure recovery. Each session maintains on average 2.74 backup service paths ......................................................... 101
6.9 Failure frequencies on a dynamic overlay network with variation rate 100 peers/minute with and without reactive failure recovery ................................................................. 102
6.10 Failure frequencies on a dynamic overlay network with variation rate 100 peers/minute with and without proactive failure recovery. Each session maintains on average 2.74 backup service paths ......................................................... 102
6.11 Cumulative QoS success rate of the GC algorithm on different topology types with 150 requests/minute ......................................................... 102
6.12 Cumulative QoS success rate of the LD algorithm on different topology types with 150 requests/minute ......................................................... 102
6.13 Cumulative QoS success rate comparison of the GC algorithm on the Rmesh0.1, Rmesh0.3, and Rmesh0.5 with 150 requests/minute ......................................................... 103
6.14 Cumulative QoS success rate comparison of the LD algorithm on the Rmesh0.1, Rmesh0.3, and Rmesh0.5 with 150 requests/minute ......................................................... 103
6.15 Cumulative QoS success rate comparison of the GC algorithm on the Tmesh0.1, Tmesh0.3, and Tmesh0.5 with 150 requests/minute ......................................................... 104
6.16 Cumulative QoS success rate comparison of the LD algorithm on the Tmesh0.1, Tmesh0.3, and Tmesh0.5 with 150 requests/minute ......................................................... 104
7.1 The access SpiderNet system architecture in one access domain.

7.2 Major operations in the ordered coordination algorithm.

7.3 Illustration of service distribution problems.

7.4 The pseudo code of the optimal service distribution heuristic algorithm.

7.5 End-to-end QoS of different service compositions.

7.6 Overhead of each dynamic service composition actions illustrated in Figure 7.5.

7.7 Success rate comparisons among the fixed, random and our heuristic algorithms.
Chapter 1

Introduction

The algorithmic and architectural design and implementation of SpiderNet, a middleware framework for quality-aware service composition, is motivated by both the user’s need for high quality advanced distributed application services and emerging ubiquitous and peer-to-peer computing environments, which require new application service provisioning and management solutions. In this chapter, we introduce the motivation of our research in quality-aware service composition, review currently available solutions, and describe high-level overview and major contributions of SpiderNet service composition framework. We then outline the rest of the dissertation.

1.1 Motivation

Today’s Internet has evolved to become an indispensable service delivery infrastructure instead of merely providing best-effort data transmission between distributed hosts. Many Internet services, such as CNN news [2], Google searching service [5], and Yahoo map [7], are becoming integral parts of everyday life for almost everyone. In this dissertation, service refers to application service that can provide certain functionality to other applications or users. Examples of application services include various multimedia services such as media transcoding and audio mixing, web services such as on-line shopping, and emerging digital government/society services (e.g., intelligence data gathering, patient monitoring). Indeed,
the need for application services have become one of the driving force of today’s information technology advancement. However, because of problems pertaining to scalability, implementation and deployment difficulties, the current Internet infrastructure has become inadequate to meet user’s service requirements such as failure resilience, quality-of-service (QoS) and ubiquity.

### 1.1.1 Emerging Internet Services

Our research is motivated by the emerging Internet services such as ubiquitous computing [78], peer-to-peer computing [4], and utility computing [77, 41]. These emerging Internet services poses new challenges for the service provisioning and management system.

- **Decentralization.** Under the driving force of scalability, robustness, and cost-effectiveness, the Internet service delivery has evolved from the model of using centralized server host, to server-clusters, to server-networks (e.g., content delivery network [1]), and recently to peer-to-peer systems. The above Internet service provision evolution gains us the insight that future Internet service delivery will become decentralized, which means that both services and contents will be distributed on geographically dispersed hosts and provisioned by different service providers. Such a decentralization demands a service composition framework that can compose distributed service components into a complete service delivery.

- **Failure resilience.** Future Internet services are expected to be resilient, which means that they should be able to quickly recover from any performance failures or malicious attacks. Unfortunately, recent study [70] has shown that current Internet infrastructure is vulnerable to various failures and the failure recovery can take minutes or hours. Service composition promotes resilient service delivery in the following ways: (1) quick and easy detection of failures using service monitoring; and (2) quick recovery using alternative service components and application-level routing paths. The replicated
service components are produced either by the natural redundancy property of the large-scale service provision system or by explicitly enforced replication schemes [51].

- **Quality of Service (QoS).** Distributed application service delivery demands QoS support, especially for QoS-sensitive applications such as multimedia and e-commerce. In the recently proposed utility computing, QoS provision is enforced by the service level agreement contracts [41] and QoS violation is even related with financial penalty. Although QoS support has been extensively studied over the past decade in both network layer [15, 16, 31] and end-host systems [55, 18, 14, 29], the integration of QoS solutions for these two different areas has not been sufficiently addressed. Moreover, service composition poses new challenges to QoS provisioning since the composed service includes multiple service components connected by multi-hop application-level connections.

- **Ubiquitousness.** With the advent of ubiquitous computing era, the computer system has been extended to the whole physical space and receded into the background of our lives [78]. It consists of various stationary, embedded and mobile devices that are diverse in size, capability and resource capacity. Moreover, a single user possesses multiple heterogeneous devices ranging from desktop, laptop, handheld PC to cell phones. Thus, ubiquitous computing requires future service delivery is adaptive and configurable. The goals of ubiquitous service composition are two folds: (1) provide context-aware adaptation services to accommodate heterogeneous client devices; and (2) overcome resource constraints by aggregating the resources of multiple devices.

### 1.1.2 Why Service Composition

Traditionally, application services are provisioned in a centralized monolithic way. Each application service is hosted as a whole on a single computer or computer cluster. Users can access the application service via Internet. However, this centralized monolithic approach has
several serious limitations. First, it is very difficult to dynamically customize a monolithic service based on different user's requirements. Second, as users expect to get more and more functionalities and features from the application service, the complexity of the monolithic service has made it very difficult to manage. Finally, it is very expensive for the application service provider to provide the whole service solution from scratch by themselves.

To address the above problems, SpiderNet proposes a compositional approach to application service provisioning. In this dissertation, we will focus on distributed streaming applications such as multimedia streaming and XML data streaming. In the compositional approach, each service provisioning entity, called service node, only provides and manages a few number of service components. Each service component is self-contained application unit providing certain service functionality. We can connect distributed service components into a linear path called service path or a directed acyclic graph called service graph, which collectively deliver user required services. In other words, service composition system allows advanced application services to be dynamically assembled from distributed service components based on the users function and quality requirements. The networked service nodes could be deployed in either local pervasive computing environment such as smart rooms or across wide-area networks such as peer-to-peer systems.

The SpiderNet service composition framework can be beneficial to a range of important applications such as e-commerce, advanced multimedia services, distributed sensor monitoring, and scientific collaboration. We now describe some application examples to further motivate why service composition is desirable. First, we could use the multimedia service overlay to provide pervasive content distribution that often requires dynamic on-demand adaptations or transformations based on different computing context. For example, a Japanese person wants to listen to CNN sports news on his cell-phone. The system can automatically compose an html-to-speech service and English-to-Japanese language translation service to transform the original web content into the user desired content format. The second application to support distributed sensor monitoring where thousands of sensors are deployed
across wide-area networks. Service composition can provide various stream processing func-
tions such as aggregation, filtering, and tracking to efficiently deliver useful information to
the end-user. Another important application is to assist crisis action planning, which re-
quires on-demand customized media streaming with specialized annotations during a crisis
event such as September 11. The same media source needs to be augmented, annotated, and
filtered based on the needs of different user roles, such as fire-fighters, medical people, and
the governor.

1.2 Research Challenges

Before introducing SpiderNet, we first describe the specific research challenges our research
has been focused on. To the best of our knowledge, these challenges have not been solved
by existing related work that will be reviewed in Chapter 8.

- **Quality-of-Service management.** The first challenge is provide service compo-
sition with QoS assurances such as service time and data loss rate, which is called
QoS-aware service composition. Traditionally, QoS management has been studied in
different layers with different focuses (i.e., performance or resource). QoS-aware ser-
vice composition is challenging because it requires us to provide an integrated solution
considering performance, resource and more importantly function requirements simul-
taneously. Moreover, the composition topology could be directed acyclic graph instead
of a linear path in order to support parallel execution of service tasks. On the other
hand, QoS-aware service composition is performed at the distributed application layer,
which presents different challenges from the traditional network or end-system QoS
problems.

- **Efficient resource utilization.** In addition to satisfy individual QoS requirements,
we also want to achieve efficient resource utilization. This problem has different impli-
cations in different computing environment. For example, in the pervasive computing
environment, we need to consider how to overcome resource constraints of mobile devices such as limited memory. However, when we look at the peer-to-peer computing environment, the problem becomes how to efficiently distribute service provisioning load to different peers to achieve best load balancing.

- **Self-healing.** The third challenge is to provide automatic failure recovery to maintain the quality of composed services throughout the whole service session. The failure recovery is required to be both efficient and fast since we want to support soft real-time streaming applications.

- **Automatic consistency check.** The fourth challenge is to provide automatic consistency check between different service components. Because service components can be independently developed by different service providers, the input and output of two arbitrary service components may not be compatible with each other. So we have to check their consistencies in order to provide a correct composed service.

- **Flexibility.** The fifth challenge is to achieve flexible (i.e., expressive) service composition. Most previous solutions support linear composition structure with fixed composition order, which greatly limits the applicability and efficiency of service composition. For example, we should allow parallel execution of service functions instead of strict pipelined chaining of service functions. We should also explore exchangeable composition orders to enhance the QoS of composed services.
1.3 SpiderNet: A Quality-Aware Service Composition Middleware Framework

1.3.1 High Level Overview

SpiderNet is a distributed middleware framework that can provide quality-aware service composition in different distributed computing environments. By distributed middleware, we mean that SpiderNet is designed and implemented on top of existing Internet infrastructure and off-the-shelf operating systems. Thus, SpiderNet achieves easy deployability by avoiding any changes to the Internet infrastructure or operating systems. SpiderNet comprises of three conceptual layers: (1) abstract composite service layer, (2) instantiated composed service layer, and (3) service overlay network layer. The abstract composite service layer defines the interface between the SpiderNet middleware and the SpiderNet applications or users. This layer presents the QoS-aware advanced distributed application services to end-users or applications, which are relieved from the burden of QoS-aware composite service setup and management. The instantiated composed service layer defines dynamically created advanced application services composed from current available service components. The bottom layer is the service overlay network consisting of distributed service provisioning hosts that are connected via application-level virtual links.

To accommodate emerging heterogeneous computing environments (e.g., smart rooms, peer-to-peer systems, utility computing), SpiderNet adopts an integrated hybrid system architecture, which consists of two types of sub-systems: (1) core SpiderNet, and (2) access SpiderNet. The core SpiderNet is constructed as a service overlay network layered on top of wide-area networks. Each node in the service overlay network provides both individual application service functions and application-level data forwarding. We propose two different designs for the core SpiderNet: utility SpiderNet and peer-to-peer SpiderNet, which targets for the managed enterprise service environment and decentralized peer-to-peer systems,
respectively. The access SpiderNet is deployed in each local-area ubiquitous computing environment such as smart rooms. The major goal of access SpiderNet is to adapt the service delivery to overcome resource constraints on mobile devices, resource fluctuations, and user mobility in ubiquitous computing environments.

To provide quality-aware service composition during the entire service session, Spidernet partitions the service composition process into two phases: (1) service setup phase, and (2) service runtime phase. During the service setup phase, SpiderNet first acquire the service composition requirements from the end-user or application. Then, SpiderNet performs service discovery to find available service components that match the required service functions. Next, SpiderNet selects the best service components, checks inter-component consistencies, and composes them based on resource utilization optimization goal subject to the required QoS constraints and inter-component relations. If service components can be dynamically uploaded such as in a smart room, SpiderNet provides dynamic service distribution to achieve efficient resource utilization. After the above steps, the composed service session is established and the composed service enters runtime phase. During runtime, SpiderNet monitors the availability and performance of the composed service. If any failure (e.g., outage, QoS violations) is detected, composition failure recovery is triggered to maintain the quality of the composed service. Finally, the service session ends and resources are released.

1.3.2 Major Contributions

The major contributions of SpiderNet are summarized as follows:

- SpiderNet proposes a novel service overlay network to inter-connect previously isolated distributed service components via application-level overlay links. By keeping track of the states information of service components and overlay links, SpiderNet provides QoS-aware and failure resilient service composition. Thus, SpiderNet can deliver high-quality on-demand services in a cost-effective way by properly composing new services
SpiderNet provides a centralized solution for quality-aware service composition in utility computing environments [41]. In utility computing, the user is offered application services as utilities, which can be bought on-demand as the user would for electricity. One of the biggest challenges for utility computing is to provide QoS assurances for the service delivery, which is often enforced by the bilateral service level agreement (SLA) contract between the service provider and the user. The utility SpiderNet provides QoS-assured wide-area service composition based on the SLA contracts. The utility SpiderNet is managed by a single service provider called the portal service provider. The portal service provider deploys management nodes in strategic positions in the wide-area network, which form the control plane of the service overlay network.

Service composition is highly desirable in peer-to-peer (P2P) systems since services and data information are inherently dispersed. SpiderNet provides a fully decentralized service composition solution for P2P systems [38, 43, 40]. SpiderNet proposes a novel bounded composition probing (BCP) approach to scalable QoS-aware service composition, which can reduce composition overhead by several orders of magnitude and still remain the efficiency of QoS-aware service composition. To overcome dynamic peer arrivals/departures, the system provides integrated proactive and reactive failure recovery mechanisms to maintain both availability and performance of each active streaming session.

SpiderNet proposes a two-tier service composition model for pervasive computing environments such as smart rooms [39, 42, 37]. Ubiquitous SpiderNet is deployed in service access domains, such as smart rooms where service components can be dynamically uploaded. The major goal of ubiquitous SpiderNet is to perform context-aware service adaptation and service distribution in ubiquitous computing. The purpose of context-aware service adaptation is to fill the gap between the core SpiderNet and the access
SpiderNet. Service distribution is for overcoming the resource constraints of mobile devices using resource aggregation.

- We validate the feasibility and efficiency of SpiderNet using both prototype implementation and extensive simulations. To demonstrate the effectiveness of SpiderNet, we also implement a set of distributed multimedia applications on top of SpiderNet. Our prototype implementation shows that SpiderNet can provide QoS-aware service composition during setup phase within tens of million seconds in a smart room environment, and within a few seconds in wide-area network environment (i.e., PlanetLab [6]). To evaluate the efficiency of the QoS-aware service composition algorithms, we compare SpiderNet with both optimal solution and other common heuristic algorithms. Our experiments show that SpiderNet can achieve near-optimal performance with low overhead. We also conducted comparative study to evaluate the SpiderNet on top of different overlay topologies. Our results indicate that bounded degree mesh topology is most suitable for QoS-aware service composition.

1.4 Outline of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 introduces the SpiderNet system models including the application service model, SpiderNet system architecture, the service composition protocol steps, and the key assumptions made by SpiderNet. We introduce the SpiderNet architecture from both horizontal and vertical angles. In Chapter 3, we present the utility SpiderNet, an instance of the core SpiderNet. Utility SpiderNet is a contract-based federated service overlay network providing dynamic composed services for the user as utilities. In Chapter 4, we present a voice-over-IP conferencing system as an application example of the Utility SpiderNet. In Chapter 5, we present the P2P SpiderNet, the other instance of the core SpiderNet. The P2P SpiderNet is a fully decentralized and self-organizing composable service overlay network. In Chapter 6, we present a comparative
study to analytically and experimentally compare two different core SpiderNet designs and the effect of different overlay topology on the performance of QoS-aware service composition. In Chapter 7, we present the ubiquitous access SpiderNet that can adapt the composed services received from the core SpiderNet. We review related work in Chapter 8. Finally, Chapter 9 concludes this dissertation and suggests future work.
In this chapter, we introduce the SpiderNet system model. We first describe the application service model consisting of the service component model, inter-component QoS consistency model, and the composite service model. We then present the SpiderNet system architecture in terms of horizontal view and vertical view, respectively. Finally, we introduce the SpiderNet service composition protocol, followed by the key assumptions made by the SpiderNet system.

2.1 Application Service Model

2.1.1 Service Component Model

A service component, denoted by $s_i$, is a self-contained application unit providing a certain functionality (e.g., media transocoding, language translation, data encryption), which is illustrated by Figure 2.1 (a). Each service component has several input and output ports for receiving input application data units (ADUs) and sending output ADUs, respectively. Each input port is associated with a message queue to enable asynchronous communication between service components. Each service component consists of four items, (1) function name describing the service function provided by the service component, (2) service code representing the service implementation, (3) static meta-data, and (4) dynamic meta-data.
The static meta-data of a service component \( s_i \) consists of three parts: (1) the location of \( s_i \); (2) input quality requirements of the service component such as media format, frame rate, which is denoted by \( Q^{in} = [q^{in}_1, ..., q^{in}_d] \), and output quality properties of the service component, denoted by \( Q^{out} = [q^{out}_1, ..., q^{out}_d] \); and (3) adaptation policies \( \Gamma = \{\gamma_1, ..., \gamma_l\} \), where \( \gamma_i \) is expressed by an if-condition-then-action construct. The dynamic meta-data of a service component describe its fluctuating performance conditions, such as current service delay. We use statistical QoS vector \( Q^{s_i} = [q^{s_i}_1, ..., q^{s_i}_m] \) to characterize the dynamic QoS metrics of the service component. Each QoS metric \( q_k, 1 \leq k \leq m \) is represented by a random variable, whose histogram is constructed from a number of recent sample values.

Based on the histogram, we can estimate the probability density function (p.d.f.) of \( q_k \), denoted by \( \rho_{q_k} \). We use \( Pr(q_k \leq C) \) to define the satisfaction probability that the dynamic value of \( q_k \) is no larger than the required upper bound \( C \). We formally define the service component as follows,

**Definition 2.1.1.** A service component is defined as \( s_i = \langle F_i, Code, SMD, DMD \rangle \), where \( F_i \) represents the provided service function, \( Code \) defines the service implementation, \( SMD \) represents static meta-data, and \( DMD \) represents dynamic meta-data.

When we compose two service components, we need to address two key issues, illustrated
by Figure 2.1 (b). First, we need to check the QoS consistencies [39] between two different service components since they can be developed by different third-party service providers. The QoS consistency includes two aspects. First, we check whether $Q^{in}$ and $Q^{out}$ of the two composed service components are consistent. Second, we check whether the adaptation policies of the two service components conflict with each other. We will present the inter-component QoS consistency model in the next subsection. The second issue is to derive the dynamic QoS values of the composed service from those of its constituent service components and the network connection called service link. The dynamic QoS values of the composed service is defined as the accumulation of those of its constituent service components and service links.

### 2.1.2 QoS Consistency Model

The QoS consistency model includes two aspects: (1) the consistencies between output QoS parameters $Q^{out}$ of the current service component and input QoS parameters $Q^{out}$ of the next-hop service component; and (2) the compatibility between the adaptation policies of two connected service components. Unlike the IP-layer network where all routers provide a uniform data forwarding service, the node in the multimedia service overlay can provide different multimedia services, which makes it necessary to perform QoS consistency check between two connected service components. We first define the parametric consistency relation as follows,

**Definition 2.1.2.** Parametric consistency relation ($Q^{out}_{s_a} \preceq Q^{in}_{s_b}$). Given two service components $s_a$ and $s_b$, $Q^{out}_{s_a} \preceq Q^{in}_{s_b}$ if and only if $\forall i, 1 \leq i \leq d, \exists j, 1 \leq j \leq d$, (1) $q^{out}_{a_j} = q^{in}_{b_i}$, if $q^{in}_{b_i}$ is a single value, and (2) $q^{out}_{a_j} \subseteq q^{in}_{b_i}$, if $q^{in}_{b_i}$ is a range value.

The single value static QoS parameters include media format (JPEG, MPEG, etc.),

---

$^1$For simplicity, we assume that all QoS metrics are additive since a multiplicative metric (e.g., loss rate) can be transformed into additive parameters using logarithmic function. We also assume that all QoS metrics of different service components and network links are independent.
resolution (1024*768 pixels) and others. The range value static QoS parameters include frame rate ([10fps,30fps]), tracking precision ([0,100%]) and others. We can then check the parametric consistency between $Q^{\text{out}}$ of the current service component and $Q^{\text{in}}$ of the next-hop service component based on the above definition.

Besides to check the parametric consistencies, we also need to check whether the adaptation policies of the two service components are compatible with each other. Generally, we can express an adaptation policy using an if-condition-then-action construct. For example, a video tracking service can have the following adaptation policy, if CPU is below 40% and bandwidth is below 100kbps, then use RGB8 color. We say two adaptation policies are compatible if their actions will not cause any parametric in-consistency. For example, an adaptation policy of a service component specifies that the service component changes output media format from MPEGII to JPEG when the available CPU is lower than 40%. If the component’s successor specifies that the required input media format must be MPEGII, then the adaptation policy will potentially cause parametric in-consistency between the two service components.

We use hyper-cube $\pi$ to model adaptation conditions, where each condition attribute (e.g., CPU and bandwidth in the visual tracking example) represents one dimension of the hyper-cube. We check the compatibility of two adaptation policies based on the relations of their condition hypercubes. If the condition hypercubes of the two adaptation policies are equal, illustrated by Figure 2.2 (a), then the two adaptation policies will be triggered.
simultaneously. Thus, we need to check whether the two service components still satisfy the parametric consistency relation after applying the two adaptation policies, respectively. If two adaptation policies’ condition hypercubes do not have any intersection, illustrated by Figure 2.2 (b), then the two adaptation policies are never triggered at the same time. Thus, we need to check whether the new $SQ^{out}$ of $s_a$ after adaptation still satisfies the old $SQ^{in}$ of $s_b$ or the old $SQ^{out}$ of $s_a$ satisfies the new $SQ^{in}$ of $s_b$ after the adaptation. If the two hypercubes are partially overlapped, illustrated by Figure 2.2 (c), then we need to consider three possible scenarios, namely adaptation applied on $s_a$ only, or adaptation applied on $s_b$ only, or adaptations applied on both. Finally, we need to consider the cases where one hypercube is a subset of the other, illustrated by Figure 2.2 (d) and (e). Under these circumstances, we need to check upon two possible scenarios, namely adaptations happen on both service components or adaptation happens on the superset node only. We formally define the adaptation policy set compatibility relation as follows,

**Definition 2.1.3.** Adaptation Rule Set Compatibility Relation $(\Gamma_{s_a} \Join \Gamma_{s_b})$. We use $\gamma_i(SQ^{in})$ and $\gamma_i(SQ^{out})$ to represent the new $SQ^{in}$ and $SQ^{out}$ after the service component is changed by its adaptation policy $\gamma_i$. Given two adaptation policy sets $\Gamma_{s_a} = \{\gamma_{a_1}, \ldots, \gamma_{a_A}\}$ and $\Gamma_{s_b} = \{\gamma_{b_1}, \ldots, \gamma_{b_B}\}$, we define that two adaptation policy sets are compatible, denoted by $\Gamma_{s_a} \Join \Gamma_{s_b}$, if and only if $\forall \gamma_{a_i} \in \Gamma_{s_a}, \forall \gamma_{b_j} \in \Gamma_{s_b}$, (1) $\pi_a = \pi_b \Rightarrow \gamma_{a_i}(SQ^{out}_{s_a}) \preceq \gamma_{b_j}(SQ^{in}_{s_b})$; (2) $\pi_a \cap \pi_b = \emptyset \Rightarrow \gamma_{a_i}(SQ^{out}_{s_a}) \preceq SQ^{in}_{s_b} \land SQ^{out}_{s_b} \preceq \gamma_{b_j}(SQ^{in}_{s_b})$; (3) $\pi_a \cap \pi_b \neq \emptyset \land \pi_a \subseteq \pi_b \land \pi_b \not\subseteq \pi_a \Rightarrow \gamma_{a_i}(SQ^{out}_{s_a}) \preceq SQ^{in}_{s_b} \land SQ^{out}_{s_b} \preceq \gamma_{b_j}(SQ^{in}_{s_b}) \land \gamma_{a_i}(SQ^{out}_{s_a}) \preceq \gamma_{b_j}(SQ^{in}_{s_b})$, (4) $\pi_a \subset \pi_b \Rightarrow \gamma_{a_i}(SQ^{out}_{s_a}) \preceq \gamma_{b_j}(SQ^{in}_{s_b}) \land SQ^{out}_{s_a} \preceq \gamma_{b_j}(SQ^{in}_{s_b})$, (5) $\pi_b \subset \pi_a \Rightarrow \gamma_{a_i}(SQ^{out}_{s_a}) \preceq \gamma_{b_j}(SQ^{in}_{s_b}) \land SQ^{out}_{s_a} \preceq \gamma_{b_j}(SQ^{in}_{s_b})$.

Based on the above two definitions, we define the inter-component QoS consistency relation as follows,

**Definition 2.1.4.** Inter-component QoS consistency $(s_a \Leftrightarrow s_b)$. Given two service components $s_a$ and $s_b$, their static meta-data items $(SQ^{in}_{s_a}, SQ^{out}_{s_a}, \Gamma_{s_a})$ and $(SQ^{in}_{s_b}, SQ^{out}_{s_b}, \Gamma_{s_b})$. We
define that \( s_a \) is QoS consistent with \( s_b \) (\( s_a \Leftrightarrow s_b \)), if and only if (1) \( Q_{out}^{out} \leq Q_{in}^{in} \) and (2) \( \Gamma_a \triangleright \Gamma_b \).

In SpiderNet, static meta-data items are described using the XML-based markup language HQML [44]. SpiderNet check the QoS consistency between two service components using the HQML syntactic and semantic parsers [44] according to the above Inter-component QoS consistency definitions.

### 2.1.3 Service Path: A Simple Composite Service Model

Based on the above service component model, a simple composite service can be represented by a chain of service components, called *service path*, which is illustrated by Figure 2.3. We use \( SP \triangleq s_1 \rightarrow s_2 \ldots \rightarrow s_n \) to denote the service path that includes service components \( s_i, 1 \leq i \leq n \). The QoS values (e.g., delay, data loss rate) are defined as the “accumulation” among those of its constituent service components and service links. According to different interactions between SpiderNet and its users, we classify the service path into four different types:

- **Unicast service path.** A unicast service path, illustrated by Figure 2.3 (a), receives input ADU stream from the sender, processes the ADU stream using a chain of transformation services, and then delivers the ADU stream to the receiver. For example, Mary wants to stream her honeymoon trip video to her friend Jane who is having lunch in a restaurant and only has a smart cell-phone on hand. Thus, Mary can ask SpiderNet to perform a sequence of transformations, such as scaling for small screen and format transcoding, in order to fit the original video into Jane’s cell-phone.

- **Circular service path.** A circular service path, illustrated by Figure 2.3 (b), is a special case of unicast service path when the sender and receiver are mapped to the same host. For example, the user wants to make a digital photo album from the JPEG images taken by his digital camera. The digital photo album creation service is
composed of a list of autonomous services including image editing, adding background music, embedding captions, and html file generations. The user sends the stream of JPEG images to the first service component and then receives the stream of html pages of photo album from the last service component.

- **Pull service path.** A pull service path, illustrated by Figure 2.3 (c), provides not only a list of services, but also the original data content. For example, the user wants to be informed the latest sports news who only has a cell-phone on hand. Thus, SpiderNet first finds a news server providing the user’s interested news. Then a chosen intermediate service transforms the news web page into plain text string. Finally a text-to-audio service is selected to generate audio signals from the input text strings and then deliver the audio signal stream to the user’s cell-phone. A pull service path can be viewed as an extension of today’s web server service.

- **Push service path.** A push service path, illustrated by Figure 2.3 (c), provides a list of security and information processing services and then stores the client’s data stream into a selected storage server. A push service path can be regarded as an extended
network storage service but with better security support (authentication service) and enhanced data publication/suscription services. For example, the information classification service can automatically classify the input data stream based on their contents. Then the index generation service can generate an data index for the client’s data for later reference.

2.1.4 Service Graph: A Complex Composite Service Model

To allow parallel execution of service functions, we can also compose service components into a directed acyclic graph called service graph, illustrated by Figure 2.4. The nodes in the service graph represent the service components and the edges represent service link. We formally define the service graph as follows,

**Definition 2.1.5.** A service graph is defined as $SG = (S, L)$, $S = \{s_i | 1 \leq i \leq |S|\}$, $L = \{\ell_k | \ell_k = s_i \rightarrow s_j, 1 \leq k \leq |L|\}$, where $s_i$ and $\ell_k$ represent the service component and service link, respectively.

A service graph can be decomposed into multiple branch paths. For example, in Figure 2.4, the service graph consists of two branch paths $SP_1 \triangleq s_1 \rightarrow s_2 \rightarrow s_4$ and $SP_2 \triangleq s_6 \rightarrow s_1 \rightarrow s_3 \rightarrow s_5 \rightarrow s_6$. We define the QoS of a service graph as the worst value among those of its constituent branch paths. For example, if we assume that the QoS values are minimum optimal, then the QoS values of the service graph in Figure 2.4 is the largest QoS values of its two branch paths $SP_1$ or $SP_2$. The QoS values of the branch path is defined as the
accumulation of those of its constituent service components and service links. Similar to the service path, we can also have unicast service graph, circular service graph, push service graph, and pull service graph depending on the relationship between the SpiderNet and its end-user or application.

2.2 Hybrid Horizontal Architecture

We now describe the SpiderNet framework from a horizontal angle. To accommodate emerging heterogeneous computing environments (e.g., smart rooms, peer-to-peer systems, utility computing), SpiderNet adopts an integrated hybrid system architecture, which consists of two types of sub-systems: (1) core SpiderNet, and (2) access SpiderNet. The end-to-end service delivery in SpiderNet starts from the entrance access SpiderNet, transverses through the core SpiderNet, ends at the exit access SpiderNet. The core SpiderNet provides major service functions required the end-user or application. The access SpiderNet provides necessary service adaptation and resource aggregation mechanisms in the edge network domain to accommodate the requirements of ubiquitous computing. SpiderNet provides two different core instances, utility SpiderNet and peer-to-peer (P2P) SpiderNet, which targets for the utility computing and P2P computing, respectively. Next, we give the overview of the core SpiderNet and access SpiderNet.
2.2.1 Core SpiderNet

We have designed two different instances of the core SpiderNet: utility SpiderNet and P2P SpiderNet, which are introduced as follows,

- **Utility SpiderNet.** In utility computing, the user is offered quality computing services as utilities, which can be bought on-demand as the user would for electricity. One of the biggest challenge for utility computing is to provide QoS assurances for the service delivery, which is often enforced by the bilateral service level agreement (SLA) contract between the service provider and the user. The utility SpiderNet provides QoS-assured wide-area service path routing based on the SLA contracts. The utility SpiderNet is managed by a single service provider called portal service provider. The portal service provider deploys management nodes in strategic positions in the wide-area network. The management nodes form the control plane of the service overlay network, which are responsible for composing and maintaining composite services (i.e., service paths or service graphs). The utility SpiderNet targets to the managed enterprise service provisioning environment whose scale is expected to be a few hundred nodes.

- **P2P SpiderNet.** P2P systems are fully decentralized, self-organizing distributed systems, such as P2P file sharing systems [4]. In P2P SpiderNet, autonomous peers voluntarily join the service overlay network and inter-connect with each other via application-level virtual links. All peers are regarded as equal entities and make their own decisions on the service composition and maintenance. Thus, service discovery, service composition, and runtime failure recovery have to be provided in a completely decentralized and self-organizing way. Furthermore, P2P SpiderNet can have millions of nodes according to the scale of today’s P2P systems such as Gnutella [4]. Thus, the research focus of P2P SpiderNet is to achieve both scalability and efficiency in QoS-aware service composition.
2.2.2 Access SpiderNet

The access SpiderNet is deployed in the edge network domain such as smart rooms. The major goal of the access SpiderNet is to achieve ubiquitous service delivery through context-aware service adaptation and service distribution. The purpose of context-aware service adaptation is to fill the gap between the core SpiderNet and the access SpiderNet. The service adaptation often requires composing a set of adaptors (e.g., transcoding, scaling) to fulfill the service adaptation tasks. The purpose of service distribution is to overcome the resource constraints of mobile devices by transparently aggregating resources on multiple devices possessed by the user. Instead of instantiating the whole composed edge service on the resource-constrained mobile device, we properly partition it and offload part of it to nearby surrogates.

2.3 Layered Vertical Architecture

We now present the SpiderNet system architecture from a vertical angle. The vertical SpiderNet system architecture consists of three conceptual layers, illustrated by Figure 2.6. The top layer consists of abstract composite services that could be used by the end-user or upper-level application. It represents the interface between the SpiderNet system and the SpiderNet users or applications. The middle layer consists of instantiated distributed services that are composed from the available service components in the current environment. The bottom layer represents the service overlay network layered on top of the IP-layer network. In this section, we first present the detailed system models for the above three layers.

2.3.1 Abstract Composite Service Layer

A composite service request can be specified using a function graph and a QoS requirement vector, which is denoted by $\Upsilon = (\xi, Q_{target})$, where $\xi$ represents the function graph and $Q_{target}$ represents the user’s QoS requirements. The function graph specifies the required
service functions \( (F_i) \) and the inter-service dependency and commutation relations, which is illustrated by the top tier in Figure 2.6. The dependency relation from \( F_1 \) to \( F_2 \) means that the output of \( F_1 \) will be used as the input by \( F_2 \), which is denoted by \( F_1 \rightsquigarrow F_2 \). The commutation relation between \( F_1 \) to \( F_2 \) means that the composition order of \( F_1 \) and \( F_2 \) can be exchanged, which is denoted by \( F_1 \sim F_2 \). The commutation of two functions does not affect the the composed service function delivered to the end-user. However, the QoS provisioning or resource requirements of the composed service can be affected. We formally define the function graph as follows,

**Definition 2.3.1.** A function graph is defined as \( \xi = (F, DR, PR) \), \( F = \{F_i|1 \leq i \leq |F|\} \), \( DR = \{dr_i|dr_i \triangleq F_i \rightsquigarrow F_j|1 \leq i \leq |DR|\} \), \( PR = \{pr_i|pr_i \triangleq F_i \sim F_j|1 \leq i \leq |PR|\} \), where \(|F|\), \(|DR|\), and \(|PR|\) represent the cardinalities of the set \( F \), \( DR \), and \( PR \), respectively.

We use \( Q_{\text{target}} = \{(C^{q_1}, P^{q_1}), ..., (C^{q_m}, P^{q_m})\} \) to define the user’s statistical QoS requirements for the composed service, where \( (C^{q_i}, P^{q_i}) \) specifies the bound \( C^{q_i} \) and the satisfac-
tion probability\(^2\) \(P^{q_i}\) for the metric \(q_i\) that represents a QoS metric such as service delay and data loss rate \(^3\). For example, a composite service request for a mobile video streaming service can be specified as follows, \(Υ = \langle\{F_1 = VideoServer, F_2 = EmbedCaption, F_3 = ImageScaler, F_4 = ColorFilter, F_5 = VideoPlayer\}, \{F_1 \sim F_2, F_2 \sim F_3, F_4 \sim F_5, \{F_3 \sim F_4\}\}, [(10\text{ms}, 98\%)^{\text{Delay}}, (5\%, 99\%)^{\text{LossRate}}]\rangle). Users can either directly specify the composite service request using extensible markup language (XML) or use available visual specification environment such as QoSTalk [44, 80].

2.3.2 Instantiated Distributed Service Layer

This instantiated distributed service layer consists of service paths/graphs used by the current active service sessions, illustrated by the middle layer in Figure 2.6. Each service link is mapped to an overlay path by the overlay data routing layer\(^4\). If an overlay node contributes multimedia services on a service path/graph, it is called a service node. If an overlay node only performs application-level data relaying on the service path/graph, it is called a relay node. For example, in Figure 2.6, \(v_1\) is a service node and \(v_2\) is a relay node. The service paths/graphs are the results of QoS-aware service composition (QSC) in the service overlay network. We formulate the QSC problem as a two dimensional graph mapping problem. In one dimension, we can derive different composition patterns from the original function graph by considering the commutation links. In the other dimension, we can map each service function into different functionally duplicated service components because of the inherent redundancy property of the service overlay network. These duplicated service components provide the same functionality but can have different QoS properties (e.g., service time) and available resources on the local peer host (e.g., CPU, memory). Thus, we can derive different service paths/graphs from the function graph by considering the above two dimensions. The

\(^2\)The satisfaction probability is defined as the probability when the random variable \(q_i\) is less or equal to \(C^{q_i}\), assuming \(q_i\) is minimum-optimal.

\(^3\)Note that bandwidth requirement is regarded as a resource requirement metric.

\(^4\)In the current implementation of SpiderNet, the overlay data routing layer uses shortest path routing algorithm consider network delay metric only.
QSC problem is to find the best mapping from the function graph to the best qualified service path/graph that satisfies the user’s multi-constrained QoS requirements $Q^{req}$ and achieves best load balancing in the current service overlay network.

### 2.3.3 Service Overlay Network Layer

The service overlay network consists of a set of overlay nodes inter-connected by application-level connections called overlay links, illustrated by the bottom layer in Figure 2.6. Each SpiderNet node provides not only application-level data routing as in the conventional overlay networks[10, 24], but also a set of application service components. We formally define the service overlay network as follows,

**Definition 2.3.2.** A service overlay network is described by a weighted directed graph $G = (V, E)$, where $V$ represents the set of $|V|$ peers, denoted by $v_i, 1 \leq i \leq |V|$, and $E$ represents the set of $|E|$ overlay links, denoted by $e_j, 1 \leq j \leq |E|$.

Application-level data relaying [10] is required between two overlay nodes that are not directly connected. For example, in Figure 2.6, the data transmission between $v_1$ and $v_4$ has to be relayed via $v_2$ since $v_1$ and $v_4$ are not directly connected. The service overlay network topology can be constructed in different ways, which are introduced as follows:

- **Structured mesh topology.** Most previous overlay networks [24, 10, 72] are data-forwarding networks, which often have mesh overlay topologies. Unicast and multicast data routing are then performed on top of the mesh. The mesh topology can be either a complete graph or a bounded-degree graph. The major motivation for using a mesh topology is to achieve link failure resilience in overlay networks. When an overlay link fails during runtime, a detour path [10] can be calculated based on the mesh topology. A complete graph mesh offers the best routing performance since the data communication between any two nodes does not involve extra application-level data relays. However, it has serious scalability problem since each node has to constantly
monitor the overlay links from itself to all other nodes. The total monitoring overhead would be $O(N^2)$. To solve the problem, the bounded-degree mesh topology is proposed to reduce the monitoring overhead to $O(d \times N)$, where $d$ is the maximum node degree in the mesh. However, the bounded-degree mesh topology can lead to routing inefficiency because data communication between two un-adjacent overlay nodes have to be relayed by one or more intermediate overlay nodes.

The advantage of mesh overlay topology is that it allows overlay link failure recovery to be easily implemented by application-level routing on top of the mesh. However, it cannot solve the overlay node failures. The disadvantage of mesh overlay topology is that it can adversely affect the data delivery efficiency (e.g., prolonged delay), which, however, is extremely important to multimedia streaming applications. Thus, a mesh overlay topology needs extra optimization and maintenance in order to minimize routing inefficiency (e.g., delay stretch) caused by the application-level relays. Even the simplest optimization of mesh topology can be NP-hard [66]. Moreover, in variable topology networks like P2P networks, mesh maintenance can be more difficult.

- **Unstructured transient topology.** SpiderNet can have a transient overlay topology based on the on-demand service composition requirement [38]. Each SpiderNet node does not have a fixed set of neighbors as in the mesh topology. Instead, the neighbor set of each SpiderNet node is dynamically set by the routing algorithm. For example, if a composed streaming path currently traverses from node $v_i$ to node $v_j$, then node $v_j$ is regarded as the neighbor of node $v_i$. Thus, no extra application-level relays are required between two communicating SpiderNet nodes. In order to control the monitoring overhead, the neighbor set size is also bounded. If a new node needs to be added to $v_i$’s neighbor set while the neighbor set has reached its upper-bound, an old neighbor is removed from $v_i$’s neighbor set according to some aging algorithm.

The advantage of transient overlay topology is that it can achieve both high routing
performance and low monitoring overhead. Furthermore, it is more resilient to the variable topology system such as P2P networks. Its disadvantage is that it cannot use application-level routing to recover link failures between two communicating service components. Instead, such failures can be recovered by switching to a replicated service or content components. The assumption here is that service overlay network are highly redundant, which means that each service or content component is replicated on different nodes.

We have applied both approaches for constructing the overlay network topology of SpiderNet, which will be presented in later chapters.

### 2.4 Service Composition Protocol

The service composition protocol includes two phases: (1) service setup phase, and (2) service runtime phase, which is illustrated by Figure 2.7. The service setup phase mainly includes

![SpiderNet service composition protocol diagram](image-url)
the following major protocol steps:\footnote{Each protocol step is executed by either a distributed algorithm or a centralized algorithm depending on the target computing environment (i.e., federated service infrastructure or peer-to-peer systems).}

- **Service request specification.** The end-user or application first specifies the composite service requirements including both function graph and QoS requirements such as service time and data loss rate. The function graph can be directly specified using the visual programming environment QoSTalk\cite{44, 80, 45} or high-level XML-based specification language HQML\cite{44}. Alternatively, the user can specify his or her high-level function requirements such as *secure mobile video-on-demand*. Then, the system can automatically generate the function graph using the automatic planning/inference tools such as SWORD\cite{60}.

- **Service discovery.** SpiderNet then performs service discovery to find available service components that match the service functions specified in the function graph. The service components can be located on different overlay nodes distributed in local area networks or wide-area networks. SpiderNet has implemented both centralized and decentralized keyword-based service discovery sub-systems, which are used for small-scale centralized computing environment and large-scale peer-to-peer computing environment, respectively.

- **Initial service composition.** Based on the service discovery results, SpiderNet selects proper service components, check their consistencies, and compose them into the user desired services. The goal of SpiderNet service composition is to provide on-demand services with user desired QoS assurances, failure resilience and ubiquitousness properties. The complete service composition algorithm includes two parts: (1) service composition in the core SpiderNet, which composes user desired service functions; and (2) service composition in the access SpiderNet to provide service adaptation for ubiquitous service delivery. One major contribution of this dissertation work is to provide
initial service composition solutions for different distributed computing environments, which will be introduced in the later chapters.

- **Service distribution.** If service components can be dynamically uploaded and migrated among distributed hosts, such as in a smart room environment, SpiderNet provides service distribution to achieve efficient resource utilization. The goal of resource-efficient service distribution is to proportionally distribute the service workload onto different distributed hosts according to their current available resources. However, the assumption of the above approach is that service components can be executed by an arbitrary host. Moreover, our experiments indicate that dynamic service uploading and migration is relatively expensive, even in a local distributed computing environment (i.e., smart rooms). Thus, only ubiquitous SpiderNet executes the service distribution step. In the core SpiderNet, we assume that service components are uploaded in advance. SpiderNet only discovers and composes available service components but does not change the placement of the service components.

After the above steps, the composed service enters the runtime phase, which mainly includes the following major protocol steps:

- **Service monitoring.** During runtime, SpiderNet monitors the liveness and performance of all active service sessions. If any failure is detected, the failure diagnosis is triggered to pinpoint the failure points on the composed service path or service graph. The failure encompasses both outage failures and quality-of-service violations.

- **Failure diagnosis.** The goal of failure diagnosis is to discover which service components or service links become failed on the service path or service graph. Thus, SpiderNet can recover the failure of the service session by replacing the failed service components and service links.
• **Service composition failure recovery.** In order to achieve failure resilience, SpiderNet needs to provide fast and efficient failure recovery to maintain the quality of the composed service during the whole service session. Failure recovery is especially important for long-lived streaming applications that are prone to failures. SpiderNet explores and compares different failure recovery approaches for different distributed computing environments, which will be presented in detail by later chapters.

• **Service session tear down.** Finally, the SpiderNet user or application notifies the SpiderNet system of the ending of the service session. SpiderNet then releases the resources of the service session and deletes maintained service session states.

### 2.5 Assumptions

We now state a number of key assumptions made by the SpiderNet system.

• **Services are replicated.** SpiderNet considers a redundant computing environment where each service function can be fulfilled by multiple service components. Such a redundancy property is represented by two aspects. First, the same service function (e.g., video player) can be provided by multiple service components (e.g., real player, windows media player). These service components can be different in terms of implementations, and meta-data properties, such as QoS specifications. Second, the same service component can be replicated on different overlay nodes. The above redundancy property comes either from the natural redundancy property of large-scale distributed systems, such as peer-to-peer systems or system’s enforced automatic replications [68, 25]. However, SpiderNet does not assume any specific replication strategies. The task of SpiderNet is to properly choose the proper service components and compose them into an user desired service delivery.
• **Function graph is given.** SpiderNet assumes that the user’s functional requirements are given in the form of function graph. In other words, SpiderNet does not provide the automatic translation from the user’s high-level service function requirements into the function graph. Such a translation problem has been addressed by previous research work such as Q-Compiler [80] and SWORD [60].

• **SpiderNet nodes are cooperative and trusted by each other.** SpiderNet is based on the third-party service provisioning model. The application’s data can be processed by one or more SpiderNet nodes that are third-party service providers. We currently assume that SpiderNet nodes are cooperative with each other. Such trustiness can be enforced by pre-determined legal contracts or system-enforced security infrastructure [19].
Chapter 3

Utility SpiderNet: A Contract Based Core SpiderNet

3.1 Introduction

In this chapter, we present the utility SpiderNet, an instance of the core SpiderNet, which offers quality computing services to its users as utilities. With the utility SpiderNet, the user does not need to maintain its own complex information technology (IT) infrastructure for just occasionally using a few application services. Instead, the user can “buy” IT services from computing utility providers as they would buy electricity from power companies. Based on this model, SpiderNet is considered as a big computing service utility system managed by a service provider who provides on-demand services for its users. We call such a service provider the *portal service provider* (PSP). The QoS assurances in utility computing are enforced through bilateral *service level agreement* (SLA) contracts between the PSP and the SpidrNet users. To achieve cost-effectiveness, the PSP out-sources the provisioning of some service components to some third-party service provider, called *application service provider* (ASP). The ASP provides autonomous service components with QoS assurances that are enforced by bilateral SLA contracts between the PSP and ASP.

The organization of this chapter is as follows. We first introduce the architecture of a managed service overlay network. Second, we describe the SLA extension of the basic
composed service delivery model presented in Chapter 2. Then we describe the initial service composition and runtime failure recovery algorithms used by the utility SpiderNet, followed by the performance evaluation. Finally we summarize our work on the utility SpiderNet.

### 3.2 Managed Service Overlay Network

The managed service overlay network includes an additional management layer which is responsible for managing the service overlay network layer. Figure 3.1 illustrates the managed service overlay network. In order to control and manage the utility SpiderNet, the PSP deploys a set of management entities at strategic positions to form the SpiderNet management layer. The management entities include portals deployed in edge networks, and monitoring agents co-located with all SpiderNet nodes. The major components in the portal include: (1) overlay topology manager that controls the peering of overlay nodes; (2) service registry manager that keeps a centralized service meta-data directory; (3) monitoring agent that monitors the service levels of the composed service and also the states of overlay links.
and nodes; and (4) service composer that constructs and maintains service paths for the clients within its responsible domain. Each monitoring agent monitors the service levels of its co-located overlay node, and the link states of overlay links adjacent to the local overlay node (e.g., delay, bandwidth). Each monitor agent periodically reports the local monitoring results to all portals. Thus, each portal can construct a global view of the service overlay network by aggregating the states information received from the monitoring agents. The utility SpiderNet targets to the enterprise service provisioning environment, which is expected to consist of several hundreds of SpiderNet nodes.

### 3.3 Contract Based Service Composition

The utility SpiderNet focuses on addressing the problem of contract-based service path composition problem, illustrated by Figure 3.2. A typical service path starts from the sender to an entrance portal, then traverses through the chosen service components, finally passes an exit portal and ends at the receiver. We assume that each service component $s_i$ and overlay link $e_i$ is associated with an SLA contract specifying its QoS assurances (e.g., service time or network delay, data loss rate) offered by the ASP (or ISP) to the PSP. On the other hand, a
composite service $X$ is associated with an SLA contract specifying the QoS assurances that the PSP promises to the user. In the utility SpiderNet, the PSP provides portal-to-portal QoS assurances in service composition to its users. In other words, QoS assurances are defined from the entrance portal to the exit portal for the composed service.

The utility SpiderNet considers two important QoS metrics used by most real-world SLA contracts [77]: Response Time ($RT_{s_i}$) for a service component $s_i$, or delay ($D_{\ell_i}$) for an overlay link $\ell_i$, and availability ($A_{s_i}$ or $A_{\ell_i}$). Based on real-world SLA contracts, the availability metric is defined as $\frac{\text{scheduled service time} - \text{total period of unavailability}}{\text{scheduled service time}}$. The connection between two service components is called service link $\ell_i$, which is mapped to an overlay network path $\ell_i = \Gamma : e_1 \rightarrow ... \rightarrow e_\gamma$, which is calculated by the overlay data routing algorithm executed in the service overlay network. The availability $A_{\ell_j}$ of the service link ($\ell_j$) is calculated as $\prod_{e_i \in \Gamma} A_{e_i}$ assuming the availability probability of all overlay links are independent; and the delay $D_{\ell_j}$ of the service link ($\ell_j$) is derived using $\sum_{e_i \in \Gamma} D_{e_i}$.

Although we only mentioned unicast service path in the above model, we now show the model can be easily applied to other service path types. In the circular service path where the sender and receiver are mapped to the same host, the entrance and exit portals, in Figure 3.2, are accordingly mapped to the same portal. In the pull service path where the application sender does not exit, we can consider the entrance portal and the service link from the entrance portal to the first service component are all symbolic, which means they do not contribute to the calculation of end-to-end QoS metrics for the composed service delivery. Similarly, in the push service path where the application receiver does not exist, the exit portal and the service link from the last service component to the exit portal do not contribute to the QoS metrics of the composed service. One limitation of the utility SpiderNet is that it only supports service path composition, not service graph composition. In contrast, the P2P SpiderNet supports both service path and service graph composition, which will be presented in Chapter 5.
3.4 Initial Service Composition

The user contacts the utility SpiderNet through a well-known address and specifies its desired services in terms of function and QoS requirements. The request is then redirected to a nearby portal based on the system’s pre-defined proximity metric. The portal then constructs a service path in the utility SpiderNet to satisfy the user’s requirements. In this section, we first formally describe the problem of initial service composition that is then proved to be NP-hard. Second, we present two heuristic algorithms to approximate the optimal solution.

3.4.1 Problem Definition

Suppose each service component \(s_i\) is associated with an SLA contract specifying its: availability \((A_{s_i})\) and response time \((RT_{s_i})\). Each service link \((\ell_j)\) is also associated with: availability \((A_{\ell_j})\) and delay \((D_{\ell_j})\). The availability \((A_x)\) and response time \((RT_x)\) of the composed application service \(X\): \(s_0 \rightarrow s_1 \rightarrow s_2 \ldots \rightarrow s_n \rightarrow s_{n+1}\) can be derived as follows \(^1\), where \(s_0\) represents the entrance portal and \(s_{n+1}\) represents the exit portal,

\[
A_x = \prod_{i=0}^{n+1} A_{s_i} \cdot \prod_{j=0}^{n} A_{\ell_j} \tag{3.1}
\]

\[
RT_x = \sum_{i=0}^{n+1} RT_{s_i} + \sum_{j=0}^{n} D_{\ell_j} \tag{3.2}
\]

In order to make availability becomes additive and minimum-optimal, we change the equation (1) into the follows,

\[
ln \frac{1}{A_x} = \sum_{i=0}^{n+1} ln \frac{1}{A_{s_i}} + \sum_{j=0}^{n} ln \frac{1}{A_{\ell_j}} \tag{3.3}
\]

To provide QoS, the resource requirements of a service path must be satisfied. Currently, we only consider CPU and bandwidth resources. We define the term \(cpu \text{ ratio}\) for a service

\(^1\)We assume that the \(RT_{s_i}\) and \(D_{\ell_j}\) are both measured for one ADU.
component $s_i$ as $CR_{s_i} = \text{cpu}_{s_i}^{\text{required}} / \text{cpu}_{s_i}^{\text{available}}$. The $\text{cpu}_{s_i}^{\text{required}}$ represents the required CPU resource for running a new $s_i$ process. The $\text{cpu}_{s_i}^{\text{available}}$ represents the available CPU resource in the physical hosting environment of $s_i$. If $CR_{s_i} \leq 1$, then the service request can be admitted. Otherwise, the service request will be denied. Moreover, the smaller the $CR_{s_i}$, the more advantageous we choose the service component in terms of CPU load balancing because we start the new service process on a light-loaded host. Similarly, we define the term bandwidth ratio for the service link $l_j$ as $BR_{l_j} = \text{bandwidth}_{l_j}^{\text{required}} / \text{bandwidth}_{l_j}^{\text{available}}$.

With the above notations, we can formulate the QoS-aware service composition problem as follows,

**Definition 3.4.1. QoS-aware Service Composition (QSC) Problem.** Suppose we are given a directed graph representing the overlay network topology, $G = (V, E)$, where $V$ and $E$ are the sets of $N$ nodes and $M$ links, respectively. Suppose also each service component $s_i$ is characterized by nonnegative values of 2 additive QoS attributes ($\text{ln1}_{A_i}^{s_i}$, $\text{RT}_{s_i}$), $i = 1...N$. Each overlay link $l_i$ is also characterized by nonnegative values of 2 additive QoS attributes ($\text{ln1}_{A_i}^{l_i}$, $\text{D}_{l_i}$), $i = 1...M$. Given the user’s QoS requirements for the composed service $X$ ($\text{ln1}_{A_x}^{\text{target}}$, $\text{RT}_{x}^{\text{target}}$), the problem of QoS-assured service path routing is to compose a service path $SP \triangleq s_1 \rightarrow s_2... \rightarrow s_n$ from $s_0$ (entrance portal) to $s_{n+1}$ (exit portal), such that $\text{ln1}_{A_x}^{\text{target}} \leq \text{ln1}_{A_x}^{\text{target}}$ (equ.(3)) and $\text{RT}_{x}^{\text{target}} \leq \text{RT}_{x}$ (equ. (2)), and also $CR_{s_i} \leq 1$ for every $s_i$, $i = 0... n+1$, and $BR_{l_j} \leq 1$ for all $l_j$, $j = 0...n$.

We now prove that the QSC problem is NP-hard.

**Theorem 3.4.1.** QSC problem is NP-hard.

**Proof.** We prove this by showing that the *Multiple Constrained Path selection (MCP) problem*, which is known to be NP-hard [34] maps directly to a special case of the QSC problem. The MCP problem is defined as follows: Given a network that is represented by a directed graph $G = (V, E)$, where $V$ is the set of $N$ nodes and $E$ is the set of $M$ links. Each link $\ell \in E$ is associated with two nonnegative additive values: $w_1(\ell)$ and $w_2(\ell)$. Given two constraints
and $c_2$, the problem is to select a path $p$ from source node $s$ and destination node $t$ such that $w_1(p) \leq c_1$ and $w_2(p) \leq c_2$, where $w_1(p) = \sum_{\ell \in p} w_1(\ell)$ and $w_2(p) = \sum_{\ell \in p} w_2(\ell)$. The above problem is identical to the following special case of the QSC problem. Suppose all SpiderNet nodes and links have infinite resources. Thus, the resource requirements $CR_{si} \leq 1$ or $BR_{ij} \leq 1$ are always satisfied. Suppose also $A_{si} = 1$ and $RT_{si} = 0$, $i = 1...N$. Let $w_1(\ell) = ln \frac{1}{A_\ell}$ and $w_2(\ell) = D_\ell$. Let $c_1 = ln \frac{1}{A_{target}}$ and $c_2 = RT_{target}$. Thus, an identity transformation makes the MCP problem a special case of our QSC problem. Thus, the QSC problem is also NP-hard.

The above description of the QSC problem also neglects several important practical issues. First, in the real world SLA contract, the service levels (QoS metrics) are often specified using average values measured over a long time period such as a month or a year [77]. However, a composed service can last only several minutes or hours. Hence, to compose each specific service path, we need not only consider the QoS values specified in SLA contracts but also the current performance of the SpiderNet nodes and links. Second, the PSP may concurrently serve thousands of user requests by composing available service components. To achieve best QoS assurances for all SpiderNet users, we need to address load balancing to achieve best resource utilization of the overall system. Third, the user’s QoS requirements may be impossible to satisfy due to, for example, the unavailability of a specific type of service components. Hence, the goal of our QSC algorithms is to compose a service path that can best avoid QoS violations or minimize the QoS violation degree if a violation occurs. Commercial SLA contracts usually make the PSP lose more revenue when the degree of QoS violation increases [77].

### 3.4.2 Basic QSC Heuristic Algorithm

We now provide a polynomial heuristic algorithm, called QSC-basic, for the QSC problem. The key idea is to comprehensively consider different factors (e.g., SLA contracts, recent
performance, system load) and use Dijkstra algorithm. It primarily involves the following steps:

- Given a function graph, we first generate the *candidate graph* whose *ith* column in the candidate graph includes all the candidate service components providing the *ith* service in the functional graph, illustrated by Figure 3.3(a);

- Check the inter-component QoS consistency (Chapter 2) between two communicating service components on the service path. If $P_{out}$ of a service component $s_i$ satisfies the $P_{in}$ of its successor $s_j$, we add a directed edge from $s_i$ to $s_j$, illustrated by Figure 3.3(b).

- Assign cost value to each edge of the candidate graph. To comprehensively consider different factors, the “cost value” on the edge $l_k = (s_i, s_j) \in E, s_i, s_j \in V$ is defined using the following integrated metric:

$$
c(l_k) = \frac{ln \frac{1}{A_{l_k}}}{ln \frac{1}{A_{l_k}}^{max}} + \frac{ln \frac{1}{A_{s_j}}}{ln \frac{1}{A_{s_j}}^{max}} + \frac{D_{l_k}}{D_{max}} + \frac{RT_{s_j}}{RT_{max}}
$$

$$
+ \frac{ln \frac{1}{A_{l_k}}^{recent}}{ln \frac{1}{A_{l_k}}^{max}} + \frac{ln \frac{1}{A_{s_j}}^{recent}}{ln \frac{1}{A_{s_j}}^{max}} + \frac{D_{l_k}^{recent}}{D_{max}} + \frac{RT_{s_j}^{recent}}{RT_{max}}
$$

$$
+ \frac{CR_{s_j}}{CR_{max}^{max}} + \frac{BR_{l_k}}{BR_{max}^{max}}
$$

(3.4)
The ratio $\frac{\alpha}{\alpha_{\text{max}}}$ ($\alpha$ represents any of the above parameters) represents the normalized cost in terms of one specific factor (e.g., QoS assurances or load balancing).

- Run the Dijkstra algorithm to find the shortest path, illustrated by Figure 3.3(c). The shortest path is returned as the initial service composition result.

### 3.4.3 Enhanced QSC Heuristic Algorithm

Figure 3.4: Illustration of the QSC-enhanced algorithm.

The QSC-basic algorithm does not consider each individual QoS constraint for finding a
qualified service path. We now present an enhanced algorithm, called QSC-enhanced, using a modified Dijkstra algorithm. After generating the candidate graph, we associate each edge $l_k = (s_i, s_j) \in E, s_i, s_j \in V$ with a “cost value”, which is a bit different from equation 3.4 and defined as follows,

$$
c(l_k) = w_0 \cdot \frac{\ln \frac{1}{A_{ik}}}{\ln \frac{1}{A_{i}}} + w_1 \cdot \frac{\ln \frac{1}{A_{j}}}{\ln \frac{1}{A_{j}}} + w_2 \cdot \frac{D_{ik}}{D_{max}} + w_3 \cdot \frac{RT_{s_j}}{RT_{max}} + w_4 \cdot \frac{\ln \frac{1}{A_{ik}}}{\ln \frac{1}{A_{i}}} + w_5 \cdot \frac{\ln \frac{1}{A_{j}}}{\ln \frac{1}{A_{j}}} + w_6 \cdot \frac{D_{re}nt_{ik}}{D_{max}} + w_7 \cdot \frac{RT_{re}cent_{s_j}}{RT_{max}} + w_8 \cdot \frac{CR_{s_j}}{CR_{max}} + w_9 \cdot \frac{BR_{l_k}}{BR_{max}}$$  \hspace{1cm} (3.5)

The weight $w_i$ represents the significance of the $i$’th factor for selecting a qualified service path. The higher the $w_i$ value, the more important the factor is. Different from the QSC-basic algorithm where all factors are considered equal, the QSC-enhanced algorithm dynamically changes the importance of different factors by adjusting $w_i$ accordingly. The adjustment of $w_i$ is based on the “pressure” of different QoS constraints. Figure 3.4 illustrates four intermediate steps to find a service path using the QSC-enhanced algorithm. If the current accumulated value of a QoS attribute (e.g., response time) approaches its target, we increase its importance weight in the hopes that its accumulation will catch up in the later stage of the service path finding. Suppose $s_i$ is the current chosen node by the Extract Min procedure in the Dijkstra algorithm, whose final shortest path from the source $s_0$ is just determined. We define the response time pressure $“PRT”$, availability pressure $“Pavail”$, and the weight
Input:
\[ G = (V, E) \]: service overlay network;
\[ G^c = (V^c, E^c) \]: candidate graph.

\[ A_{target} \]: availability constraint; \[ RT_{target} \]: response time constraint.

\[ s_0 \]: entrance portal; \[ s_n \]: exit portal.

\[ S \]: the set of nodes whose shortest paths are determined.
\[ Q \]: priority queue including all the vertices in \( V^c - S \).

Output:
service path \( SP \) from \( s_0 \) to \( s_n \).

Pre-conditions:
\[ S \leftarrow \emptyset; Q \leftarrow V^c; w_i = 0.1, i = 0..9. \]

1. generate candidate graph \( G^c = (V^c, E^c) \) from \( G \).
2. for each edge \( e = (u, v) \in E^c, u, v \in V^c \) do
   3. \( c(e) \leftarrow \sum w_i \cdot \frac{\alpha}{\alpha_{max}} \), equation (5);
4. while \( Q \neq \emptyset \) do
   5. \( u \leftarrow \text{Extract Min}(Q) \);
   6. \( S \leftarrow S \cup \{u\} \);
   7. adjust weight \( w_i, 0 \leq i \leq 9 \), equation (6)(7)(8);
   8. for each vertex \( v \in \text{neighbor}(u) \) do
      9. \( \text{relax}(u,v) \);
10. return the shortest path \( SP \) from \( s_0 \) to \( s_n \).

Figure 3.5: Pseudo code of the \textit{QSC-enhanced} algorithm.

\[ P^RT = \frac{RT_{s_0 \rightarrow s_i}}{RT_{target}}, \quad P^{avail} = \frac{\ln \frac{1}{A_{s_0 \rightarrow s_i}}}{\ln \frac{1}{A_{target}}} \] (3.6)

weights for availability: \( w_0 = w_1 = w_4 = w_5 = \frac{1}{4} \cdot \frac{P^{avail}}{P^{avail} + P^RT} \cdot (1 - w_8 - w_9) \) (3.7)

weights for response time: \( w_2 = w_3 = w_6 = w_7 = \frac{1}{4} \cdot \frac{P^RT}{P^{avail} + P^RT} \cdot (1 - w_8 - w_9) \) (3.8)

Figure 3.5 shows the pseudo code for the algorithm. Both \textit{QSC-basic} and \textit{QSC-enhanced} algorithms have the same computational complexity \( O(K^2) \), where \( K \) is the number of nodes in the candidate graph. This bound can be improved if we use other implementation for the priority queue so that the running time of \text{Extract Min} can be reduced.
3.5 Service Composition Failure Recovery

One major goal of SpiderNet service composition is to achieve failure resilient service delivery. Recent study has shown that the wide-area IP routing (i.e., Border Gateway Routing (BGP) routing) is prone to performance failures or outages. The BGP routing failure recovery can take minutes [10]. Moreover, different SpiderNet nodes, that are autonomous end-hosts or clusters, can dynamically fail or experience outages due to software/hardware upgrade, for example. Hence, in order to achieve fault-resilience, it is necessary to provide runtime service path maintenance, which can quickly recover service path from any failures and maintain the assured quality levels for the user. When the co-located monitor agent detects failures, all related portals are notified. The portal then check its local session table that contains the session information of all the current service paths it maintains. If any current service path is affected by the failed service components or links, the portal recovers it by using a dynamic QoS-aware service composition algorithm that finds a new qualified service path for the session. We have designed two different dynamic QoS-aware service composition algorithms: (1) \textit{DQSC-complete}, which completely re-composes a new service path without considering the original service path; and (2) \textit{DQSC-partial}, which partially re-composes a new service path based on the original service path.

3.5.1 Global Failure Recovery

The global failure recovery recomposes a completely new service path upon failures without considering the original service path. Figure 3.6 illustrates the complete service path recomposition algorithm \textit{DQSC-complete}. Figure 3.6 (a) shows the candidate graph and the original service path, in which the service component \(s_{12}\) is poorly-performing and the link between \(s_{21}\) and \(s_{31}\) is broken. The \textit{DQSC-complete} algorithm first modifies the candidate graph by removing those failed or poorly-performing service components, and also replacing the broken links with alternate overlay routing paths when it is possible, illustrated by Figure
3.6 (b). In this example, $s_{12}$ is removed from the second column of the candidate graph. The failure service link between $s_{21}$ and $s_{31}$ is replaced with an alternate overlay path decided by the underlying overlay routing algorithm. The recovered link is illustrated as a dotted line in Figure 3.6 (b). Then we use the QSC-enhanced algorithm, described in Section 3.4.3, to compose a new service path that satisfies the quality levels specified in the SLA contract between the PSP and the user. Thus, the service session can be quickly recovered from outage or quality degradations by properly switching from the failed service path to the new service path.\(^2\)

### 3.5.2 Local Failure Recovery

Contrasting with the DQSC-complete algorithm, DQSC-partial algorithm only partially recomposes the failed service path based on the original service path. Figure 3.7 (a) shows the same example of composed application as Figure 3.6 (a). In the original service path, service component $s_{12}$ is poorly-performing and the service link between $s_{21}$ and $s_{31}$ is broken. In Figure 3.7 (b), however, we modify the candidate graph by not only removing the poorly-performing service component $s_{12}$ and replacing the service link between $s_{21}$ and $s_{31}$, but also removing, in the column where the old service component is good, the other candidate service components. In this example, the service components $s_{21}$ and $s_{31}$ still perform well. We remove the service components $s_{22}$ and $s_{23}$ in the third column of the candidate graph, and

\(^2\)We assume that the states of all the service components can be recovered by software.
service components $s_{32}$ and $s_{33}$ in the fourth column. Then, we still use the QSC-enhanced algorithm to compose a new service path in the modified candidate graph, illustrated in Figure 3.7 (c). The purpose of such an approach is to be able to keep those original well-performing service components in the new service path. Hence, we can reduce the migration overhead for switching from the old service path to the new one as well as maintain the quality levels. The computational complexity of DQSC-complete and DQSC-partial is still $O(K^2)$, where $K$ is the number of nodes in the candidate graph.

### 3.5.3 Analytical Comparison

Both DQSC-complete and DQSC-partial algorithms can quickly re-compose a new service path to recover from the outage/quality degradations. However, each of them has both advantages and disadvantages. The advantage of the DQSC-complete algorithm is that it can re-compose a better service path in terms of QoS assurances (e.g., shorter response time, higher availability), than the DQSC-partial algorithm, since it has more choices of service components. The disadvantage of the DQSC-complete algorithm is that the service re-composition takes longer time since it re-composes the whole service path. On the contrary, the advantage of the DQSC-partial algorithm is that it is quicker and easier to implement since it only changes part of the service path. However, its disadvantage is that the new composed service path may not be optimal. We will further discuss and compare those two different dynamic service composition approaches in the next section.
3.6 Performance Evaluation

In this section, we evaluate the performance of the initial and dynamic QoS-assured service path routing algorithms. We first describe our evaluation methodology. Then we present and analyze the simulation results.

3.6.1 Evaluation Methodology

We evaluate the performance of the service path routing algorithms using extensive simulations. We first use the degree-based Internet topology generator Inet 3.0 [81] to generate a power-law random graph topology with 3200 nodes to represent the Internet topology. We then randomly select 500 nodes as the overlay nodes and 40 other nodes as the portals. We assume an equal-degree mesh topology for the utility SpiderNet. Each node is randomly assigned 5 other nodes as its neighbors.

The initial resource availability of each IP link and service component is uniformly distributed in a certain range. The SLA values of each IP link or service component are also uniformly distributed within certain range. Different values reflect the heterogeneity and diversified quality guarantees in overlay. Moreover, to simulate the performance variation in the real world, the QoS attributes of each IP link and service component are set by uniform distribution functions, with SLA values as the mean values. We assume the Dijkstra shortest path algorithm for both the IP layer and overlay layer routing, using the instantaneous value of delay as the routing metric. The bandwidth of an overlay link is the bottleneck bandwidth along the IP network path. The delay of an overlay link is the addition of the delays along the IP network path. The accumulated delay is then normalized according to the results of Internet Round-Trip-Time study.

During each minute, certain number of user requests are generated. The user request is represented by any of 40 composite service templates that comprise 2 to 6 services. Each user session lasts from 15 to 60 minutes. The metrics we use for evaluating the QoS assurances
include QoS violation rate and QoS violation degree. The QoS violation rate is measured by the ratio of the sessions during which QoS violation happens over the total sessions. For each session, the QoS violation is said to happen if the measured average QoS attribute values (i.e., availability, response time) is worse than that specified in the SLA contract. The QoS violation degree measures that if a QoS violation occurs, how severe the QoS violation is. It is measured by the ratio of difference between the measured QoS attribute value and its target value, over the target value. Those two metrics are often associated with the financial refund/penalty policies specified in the real world SLA contracts. The minimization of those two metrics means to reduce the financial loss of the service provider.

The metric we use for evaluating the load balancing is the provisioning success rate. A composed service provisioning is said to be successful if and only if during its entire session, all the service components and links’ resource requirements on the service path are always satisfied. The composed service provisioning success rate is defined as the number of successful requests over the total number of all requests. Higher provisioning success rate represents improved load balancing in the utility SpiderNet.

For comparison, we also implement two common heuristic algorithms for composing service path: fixed and random algorithms. The fixed algorithm performs static service composition, which always select the same service component for a specific service function. The random algorithm randomly chooses service components to compose the service path.

### 3.6.2 Results and Analysis

Figure 3.8 and Figure 3.9 show the simulation results about the violation rates of two QoS attributes: availability and response time, respectively. In Figure 3.9, the X axis represents different session request rate, calculated by the number of composed service session requests per minute. The range of session request rate is selected to reflect different system workload put on the utility SpiderNet. The Y axis shows the average QoS violation rate for the availability attribute, achieved by the fixed, random and our four QSC (QoS-assured service
path routing) algorithms. $QSC$-$Basic$ and $QSC$-$Enhanced$ represent the two initial service path routing algorithms. Both of them do not include any dynamic service path re-routing mechanisms for failure recovery. Both $DQSC$-$Complete$ and $DQSC$-$Partial$ use the $QSC$-$Enhanced$ for the initial service path routing and also dynamically recovers from the service outage/quality degradations by completely or partially re-composing the service path. Each average availability QoS violation rate ($\Psi_1$) value is calculated and averaged over a period of 200 minutes for all successfully composed sessions. The results show that all the four QSC algorithms achieve much lower $\Psi_1$ than the fixed and random algorithms. The $QSC$-$Enhanced$ has as much as 20% improvements than the $QSC$-$Basic$. Both $DQSC$-$Complete$ and $DQSC$-$Partial$ further reduce ($\Psi_1$) to almost 0% lower. The reason is that the application-level service outage recovery can quickly finish in a few seconds while the IP-layer Internet path recovery may take several minutes or even hours [10]. However, the performance difference between $DQSC$-$Complete$ and $DQSC$-$Partial$ is very small.

Similarly, Figure 3.9 shows the results of the QoS violation rate for the response time ($\Psi_2$). Again, the QSC algorithms achieve much lower $\Psi_2$ than fixed and random algorithms. The performance order of different QSC algorithms is $QSC$-$Basic \preceq (worse than) QSC$-$Enhanced \preceq DQSC$-$Partial \preceq DQSC$-$Complete$. The reason why the DQSC algorithms are better than the $QSC$-$Enhanced$ is that service path re-routing always uses recent performance and load.
Figure 3.10: Average availability QoS violation degree under different system load.

Figure 3.11: Average response time QoS violation degree under different system load.

Figure 3.12: Average provisioning success rate under different system load (load balancing).

Figure 3.13: Average per session outage occurrences under different system load (stability).

information, which allows it to make better service component choices in terms of response time.

Figure 3.10 and Figure 3.11 show the results about the QoS violation degree. Figure 3.10 shows the availability QoS violation degree ($\Psi_3$). Once again, the QSC algorithms consistently achieve better performance than fixed and random. Figure 3.11 shows similar results for the response time QoS violation degree. Hence, the simulation results further validate our algorithms by showing that QSC algorithms can not only greatly reduce the violation rate but also achieve lower violation degree when QoS violations occur.
Figure 3.12 shows the results about the composed service provisioning success rate. Similar to the above experiments, each provisioning success rate value is calculated and averaged over a period of 200 minutes. The results show that all four QSC algorithms can similarly achieve much higher provisioning success rate, namely better load balancing, than the fixed and random. Finally, Figure 3.13 shows the results about the stability of our dynamic service composition algorithm. We observe that both dynamic service re-composition algorithms, DQSC-complete and DQSC-partial, are stable. They do not have large fluctuation when the system load increases.

In all of the above experiments, we observe that the performance gains of the DQSC-Complete are very small compared to the DQSC-Partial. The reason is that the initial service composition algorithm QSC-Enhanced is already very good. Hence, the selected service components in the old service path are still, by large probability, the best ones when we re-compose the service path. Hence, it is a near optimal solution that we keep the old good service components in the new service path, which is exactly the DQSC-Partial algorithm.

3.7 Summary

In this chapter, we have presented a core SpiderNet design for utility computing, called utility SpiderNet, which can provide QoS-aware service composition and failure recovery based on SLA contracts. The major contributions of the utility SpiderNet are as follows. First, we formally define the QoS-aware service composition problem and prove that it is NP-hard. Second, we provide an efficient approximation algorithm to compose service paths, which can both satisfy multiple QoS constraints and achieve good load balancing in the service overlay network. Third, we explore and compare both partial and complete dynamic service re-composition algorithms, which can quickly recover a service path from failures or QoS violations. We have implemented a large-scale simulation testbed for the utility SpiderNet. Our extensive simulation results show that the utility SpiderNet can provide QoS-aware and
load balanced service composition based on the SLA contracts. The simulation results also indicate that the partial dynamic service re-composition algorithm can achieve similar QoS assurances as the complete service re-composition algorithm but with much lower service path recovery overhead.
Chapter 4

Voice-Over-IP Conferencing: A Case of Utility SpiderNet

In this chapter, we present an overlay-based voice-over-IP (VoIP) conferencing system as an application example of the utility SpiderNet. The VoIP conferencing system is built on top of SpiderNet, which can provide scalable QoS-aware VoIP conferencing services for multiple user sessions.

4.1 Introduction

Internet has evolved into an indispensable service delivery infrastructure instead of merely providing host connectivity. IP telephony is a promising Internet service, particularly because of the significant revenue it can generate. A simple VoIP system includes two participants, where the original voice signal is periodically sampled, encoded into a bit stream, and sent over the Internet to the receiving end. In this paper, we consider an advanced VoIP service: a multi-party (or multipoint) conference service, which could include three or more participates.

The common design of a multi-party VoIP conferencing system relies on the use of a centralized multipoint control unit (MCU), which is responsible for aggregating the voices of all conference participants. However, the centralized approach suffers from the problems of: (1) scalability, where a single MCU can be short of resources (e.g., audio channels, network
bandwidth) for a large conference including hundreds of participants or many concurrent conferencing sessions, (2) poor reliability due to the single point of failure, and (3) degraded quality-of-service (QoS) when most conference members are far away from the centralized MCU. Another alternative VoIP conferencing system design is to employ either IP-layer or application-layer multicast [22]. Although the multicast approach can theoretically save network bandwidth, the construction and maintenance of multiple conference trees are often too complicated for practical use, especially when we consider multi-constrained QoS requirements.\footnote{Previous assessment study [54] has indicated that both delay and packet loss rate greatly affect the user perceived quality of the VoIP service.} 

Hence, we propose a novel overlay based VoIP conferencing system called Venus. Distributed Venus nodes are interconnected into an application-level service overlay network for resource aggregation and failure resilience. Each Venus node provides both voice mixing service and application-level data routing. Given a conferencing request, the system dynamically composes a mixing service path consisting of a number of selected Venus nodes, based on the number and locations of conference participants, and the users’ multi-constrained QoS requirements. Large-scale simulation results illustrate that Venus outperforms the centralized approach under different workload conditions. Venus also demonstrates better scaling property while we increase the number of networked voice mixers.

The organization of this chapter is as follows. We first introduce the Venus system model. Second, we present the QoS-aware VoIP conferencing session setup and maintenance algorithms. Third, we present the performance evaluation. Finally, we summarize this chapter.

### 4.2 VoIP Conferencing System Model

We now introduce the Venus system model that consists of (1) mixing service overlay network, (2) mixing service component, (3) mixing service path, and (4) conferencing service model.
Mixing service overlay network. The mixing service overlay network (MSON) forms the Venus system’s communication substrate, illustrated by Figure 4.1. Each overlay node called mixer can provide voice mixing as well as application-level data routing. For example, in Figure 4.1, mixers $v_2$ and $v_5$ provide voice mixing while $v_4$ provides application-level relaying between $v_2$ and $v_5$. A mixer can be a third-party node, which provides a service level agreement (SLA) specifying the mixing services’ resource capacity (e.g., audio channels) and QoS properties (e.g., service time). For QoS management, we introduce portals deployed at edge networks, and service monitors co-located with the mixers. The portals, such as $P_1$ and $P_2$ in Figure 4.1, are the service access points for the conferencing clients, which collectively define the QoS assurance boundary of the Venus system. Service monitors measure the local resource and QoS states and report the state information to all portals. Thus, each portal can construct a global view of the MSON in terms of resource and QoS states.

Mixing service component. Each mixing service component takes $k$ input voice signals and generates $k$ different aggregated signals, which is illustrated by Figure 4.2 (a). For example, if the mixing service component has three inputs $c_1$, $c_2$, and $c_3$, then it generates three outputs $c_2 + c_3$, $c_1 + c_3$, and $c_1 + c_2$, which are sent back to $c_1$, $c_2$, and $c_3$, respectively. Each mixer can instantiate multiple mixing service components under the constraint of its resource capacity. For example, if the mixer has 20 channels and each mixing service
component needs 5 channels, then it can instantiate at most 4 mixing service components.

Mixing service path. We can compose a set of mixing service components into a mixing service path\(^2\), which is illustrated by Figure 4.2 (b). Although the mixing service components on the mixing service path provide the same functionality, they are different in terms of the voice content. For example, in Figure 4.2 (b), \(M_1\), \(M_2\), and \(M_3\) have the mixed voice of \(c_1+c_2\), \(c_3+c_4\), and \(c_5+c_6\), respectively. To allow each participant to hear the speeches of all other members, we must further compose the three mixing service components. We can prove that each client connected to the mixing service path receives a mixed voices of all other conference members by using the induction proof on the length of the path.

Conferencing service model. Each participant in a conferencing session is notified in advance with a unique conference identifier. For simplicity, we assume that all conference members of a specific conference session contact the Venus system at the same time via the portals using the conference identifier. We define the source portal as the portal to which the conference initiator is connected, which is responsible for composing the best mixing service path used by the conference session. The QoS metric of the conference session is defined based upon the quality measures of the mixing service path between two end portals (e.g., \(P_1\) and \(P_3\) in Figure 4.2 (b)). The rationale of the definition is that the portal-to-portal QoS is within the controllable range of the Venus system, which essentially decides the QoS perceived by conference clients. The mixing service path will be torn down at the end of the conferencing session.

### 4.3 VoIP Conferencing System Design

In this section, we present the design details of the Venus system. We first describe the QoS-aware mixing service path composition followed by the runtime mixing service path.

\(^2\)We are aware that mixing service components can be composed into other topologies such as trees, which however requires more complicated construction and maintenance algorithms as well as extra resources for multi-level mixing. Thus, we only consider the case of composing mixing service path in this paper.
4.3.1 Mixing Service Path Composition

Each Venus client participates in a VoIP conference session through a Venus portal. For example, in Figure 4.3, clients $c_1$ and $c_2$ connect to the portal $P_1$; clients $c_3$ and $c_4$ connect to the portal $P_2$; and clients $c_5$ and $c_6$ connect to the portal $P_3$. Then, each portal $P_i$ selects a number of candidate mixers for mixing voices of each client group. The candidate mixers can be selected based on a combined distance metric considering available audio channels of the mixers, network delay and loss rate from the portal to the candidate mixers. The number of candidate mixers is a configurable system parameter. For example, in Figure 4.3, we select three candidate mixers for each client group. Our simulation study indicates that a small number of candidate mixers (i.e., 10 mixers) can suffice the QoS provisioning goal.

Next, we need to select among candidate mixers and compose them into a mixing service path that can mix the voices of all conference participants. We formulate the problem of mixing service path composition as a multi-constrained optimal path finding problem, namely finding the best mixing service path that achieves optimal load balancing subject to the multi-constrained QoS requirements (e.g., delay and loss rate). In [41], we have proven that the above problem is NP-complete and provided a modified Dijkstra algorithm for the problem, which can achieve near-optimal performance. The key idea is to introduce an adaptive aggregated cost metric that considers multiple factors including multiple QoS
metrics and load balancing objective. To satisfy multiple QoS constraints, we modify the Dijkstra algorithm by adaptively adjusting the importance weights of different factors in the aggregated cost metric based on their constraint pressures. More details of the algorithm can be found in [41].

An interesting property of composing a mixing service path is that the mixing service path can be any permutation of the selected mixing service components. For example, in Figure 4.3, the final mixing service path can be either $M_{12} \leftrightarrow M_{22} \leftrightarrow M_{31}$ or $M_{12} \leftrightarrow M_{31} \leftrightarrow M_{22}$. Thus, we consider all permutations of the mixing service path to further improve the QoS provisioning of the composed conferencing service. We formalize the above problem into a travelling salesman problem (TSP), which is to find a cheapest way of starting from the source portal, visiting all the selected mixers and returning to the source portal\textsuperscript{3}. Since the TSP is also NP-complete, Venus uses a heuristic algorithm for finding the best mixing service path from all permutations.

After deciding the mixing service path, Venus instantiates a voice mixing service component for each client group on the selected mixer. Then, Venus sets up the conferencing session and notifies all conference members that the conferencing service is ready for use.

\textsuperscript{3}The cost of the edge back to the source portal is set as 0 since it is not included in the mixing service path.
4.3.2 Mixing Service Path Maintenance

During runtime, the conferencing service can experience significant QoS violations or service outages due to the failures of IP-layer network links or mixers. To achieve robust VoIP conferencing service, Venus provides runtime failure detection and recovery mechanisms to maintain the availability and QoS of all active conferencing session. The source portal of a VoIP conferencing session is responsible for monitoring and maintaining the liveness and QoS of the mixing service path for each conferencing session. The failure recovery is performed at multiple layers. First, Venus relies on the overlay data routing to recover the service outage or performance failures of IP-layer network links [48]. For example, in Figure 4.3, if the overlay path from $M_{12}$ to $M_{22}$ is broken, the overlay data routing layer will dynamically find an alternative overlay path from $M_{12}$ to $M_{22}$. However, overlay data re-routing cannot recover the failures of the mixers on the mixing service path (e.g., $M_{12}$ becomes unavailable).

Under that circumstance, the source portal dynamically re-composes a new mixing service path to recover the failures. To achieve fast failure recovery, we consider a localized path repair algorithm [41], which finds a new qualified mixing service path that does not include the broken mixers but has the largest overlap with the old mixing service path (i.e. the largest number of common mixers with the old mixing service path). For example, if $M_{12}$ in Figure 4.3 fails, then we can recover the conferencing session by using an alternative suboptimal path $M_{11} \leftrightarrow M_{21} \leftrightarrow M_{32}$.

4.4 Performance Evaluation

We evaluate performance of Venus using large-scale simulations. We first use a degree-based Internet topology generator Inet 3.0 [81] to generate a power-law random graph topology with 3200 nodes to represent the IP-layer network. We then randomly select a number of nodes as Venus nodes and portals. The initial resource capacity and average QoS values of each network link and mixer are uniformly distributed. Each conferencing session lasts
5 to 30 time units. Each simulation runs 2000 time units. During each time unit, certain number of VoIP conferencing requests are randomly generated. Each VoIP conferencing request includes 10 to 200 participants whose locations are uniformly distributed.

We define the metric session success rate for performance evaluation. A QoS-aware VoIP conferencing session is provisioned successfully if and only if (1) the system has enough resources for the conferencing session, and (2) the average QoS values (i.e., delay, loss rate) measured over the whole conferencing session satisfy the required QoS values. For comparison, we also implement the centralized-random algorithm that randomly selects a mixer for each conference session, and centralized-optimal algorithm that selects the best single mixer for each conference session.

Figure 4.4 illustrates the average session success rate achieved by the algorithm of the Venus system presented in Section 4.3.1, centralized-optimal algorithm, and centralized-random algorithm, under increasing workloads on an MSON with 100 mixers. Each average success rate is measured over all conferencing requests generated during the 2000 time units simulation. We observe that Venus can achieve much higher success rates than the other two algorithms by efficiently aggregating resources of distributed mixers and finding best mixing service path under delay and loss rate constraints. Figure 4.5 shows the service
success rate comparison under the same request rate (10 requests per time unit) on different MSONs with sizes 50, 100 and 150 nodes respectively. The results demonstrate that the Venus system presents much better scaling property than the other two approaches. The system performance increases almost linearly as we increase the number of mixers.

4.5 Summary

In this chapter, we have presented a novel QoS-aware VoIP conferencing system called Venus using a composable application-level service overlay network. Venus achieves better QoS provisioning and resource utilization than a common VoIP conferencing system that uses a centralized MCU for a multi-party conference session. Venus can be easily deployed, which does not require any IP-layer or application-layer multicast support. Large-scale simulation results demonstrate the efficiency of the Venus system.
Chapter 5

P2P SpiderNet: A Decentralized Self-Organizing Core SpiderNet

In this chapter, we present P2P SpiderNet, a self-organizing, fully decentralized design for the core SpiderNet. In contrast to the utility SpiderNet that assumes the service composer has the global states information, P2P SpiderNet performs QoS-aware service composition and failure recovery in a fully decentralized fashion. Moreover, P2P SpiderNet supports both service path and service graph composition topologies. P2P SpiderNet can also explores exchangeable composition orders to enhance the QoS of composed services.

5.1 Introduction

Peer-to-peer (P2P) systems are special distributed systems where computer hosts called peers can communicate directly among themselves via application-level connections. Different from the conventional distributed systems, P2P systems are often fully decentralized and self-organizing. Recently, P2P systems have drawn much research attention with the popularity of various P2P file sharing systems such as Gnutella [4]. In this paper, we propose a service-oriented P2P system called P2P service overlay where peers can provide not only media files but also a number of application service components such as media transcoding and data filtering as well as application-level data routing. The goal of such a P2P service overlay is to enable efficient service sharing that is beyond the file sharing offered by current P2P
systems [4].

P2P service overlays are attractive since they promote Internet-scale service sharing without any administration cost or centralized infrastructure support. New services can be flexibly composed from available service components based on the user’s function and quality-of-service (QoS) requirements. Thus, P2P service overlays can achieve better scalability, manageability, and configurability in application service provisioning than the conventional client-server system model. Service composition across wide-area networks also becomes necessary in the P2P service overlay since service components are naturally distributed on different peers. Application examples of such P2P service overlays include: (1) pervasive content distribution, where the user can request adaptive content distribution to heterogeneous receivers with on-demand transformations and value-added customization; and (2) collaborative scientific computation [28], where geographically distributed research labs can leverage each other’s solutions such as data analysis tools to conduct complex scientific experiments with lower development and computation cost.

Although previous research projects (e.g., [53, 64, 21, 63, 28]) have addressed the problems of service/resource selection and composition, they present the following major limitations when applied to P2P systems. First, most systems adopt a centralized approach that assumes the global system states information. However, the above assumption becomes impractical for a large-scale P2P system that often consists of thousands of peers dispersed across the wide-area network. Moreover, extended wide-area network delay and dynamic peer arrivals/departures can exacerbate the problem since it requires frequent information updates and thus large system overhead in order to alleviate states information imprecision. Second, most previous work does not consider the dynamic node changes that are common in P2P systems. Third, most existing solutions only support linear service composition with fixed composition order, which greatly limits the applicability and efficiency of service composition. We have presented the preliminary design of the P2P service composition framework in [38], which partially addressed the first two problems.
In this chapter, we present a P2P SpiderNet to address all of the above problems within a unified framework. For scalability, P2P SpiderNet executes a novel bounded composition probing (BCP) protocol to provide fully decentralized QoS-aware service composition. In contrast to centralized schemes (e.g., [63, 41]), BCP performs on-demand selective states collection using a limited number of composition probes. The key observation behind our approach is that because of the service function constraints, QoS-aware service composition only needs the QoS and resource states information about those peers that can provide required service functions. These function-qualified peers often form a small sub-graph of the whole P2P network. Thus, it is more efficient to perform on-demand selective states collection than blindly maintain global states at each peer using expensive periodical states update. Compared to the conventional network probing, BCP has controllable overhead and considers service-specific requirements such as service function constraints and inter-service dependency/commutation relations.

Furthermore, P2P SpiderNet provides efficient failure recovery to maintain the quality of composed services throughout the whole service session. P2P SpiderNet adopts proactive failure recovery scheme to achieve fast failure recovery for soft realtime streaming applications. The proactive failure recovery approach maintains a small number of backup compositions for each active service session. The backup compositions are adaptively selected based on the conditions of the current composition and the user’s QoS requirements. Thus, we can avoid the delay and overhead of triggering BCP to find a new composition if one of the maintained backup compositions can recover the failure.

We demonstrate the feasibility and efficiency of the P2P SpiderNet service composition framework using prototype implementation. We conduct extensive experiments by evaluating the prototype on both large-scale simulation testbed and wide-area network testbed PlanetLab [6]. The experimental results show that P2P SpiderNet can achieve near-optimal QoS-aware service composition performance with low overhead. Compared to the centralized approach that requires global states maintenance, P2P SpiderNet can reduce the system
overhead by more than one order of magnitude. Moreover, P2P SpiderNet can achieve failure resilient service composition in a dynamic P2P network by maintaining a few number (e.g., less than 3) of backup compositions for each session.

The rest of this chapter is organized as follows. First, we briefly describe a decentralized service discovery scheme used by the P2P SpiderNet. Second, we present the initial decentralized QoS-aware service composition used by the service session setup phase. Third, we describe the proactive failure recovery for maintaining the quality of composed services during service sessions. Fourth, we present the experimental results. Finally, we summarize this chapter.

5.2 Decentralized Service Discovery

This section briefly describes our decentralized service discovery substrate that allows each peer to locate services in the P2P service overlay without assuming a centralized service directory. We implement the decentralized service discovery based on the Pastry distributed hash table (DHT) system [67]. The basic function of the DHT system is to map a data key to a responsible peer. We realize the keyword-based service discovery by adding one meta-data layer on top of the DHT system. Due to the space limitation, we only briefly describe the decentralized service discovery as follows:

Service registration. When a peer wants to share a service component, it registers the service component by storing the component’s static meta-data (e.g., location, input QoS, output QoS) into the DHT system. The peer first generates a key by applying a secure hash function on the function name of the service component. It then stores the meta-data into the DHT using the key. Because all functionally duplicated service component share the same function name and thus the key, the DHT system will store the meta-data list of the duplicated service components on the same DHT assigned peer.
5.3 Initial Service Composition

In this section, we present the probing-based decentralized service composition solution used at the service session setup phase. We first introduce the bounded composition probing protocol. Then we describe the per-hop probe processing algorithm, followed by the optimal composition selection algorithm.

5.3.1 Bounded Composition Probing Protocol

Given a service composition request, the application sender\(^1\) invokes the BCP protocol, which is illustrated by Figure 5.1. The BCP protocol includes four major steps:

**Step 1. Initialize the probe.** The source first generates a composition probing message, called probe, which is illustrated by Figure 5.1 (a). The probe carries the information of function graph and the user’s QoS/resource requirements. To control the probing overhead, the probe carries a *probing budget* (\(\beta\)) that defines how many probes we could use.

\(^1\)For simplicity, we use a unicast streaming application as an example.

---

**Service discovery.** When a peer wants to discover the list of service components matching certain function name, it can generate a key by applying the same secure hash function on the function name. Then, it can generate a query message using the key which will be routed to the assigned peer by the DHT system. The peer then returns the meta-data list of the duplicated service components to the requesting peer.

---

**Figure 5.1: Bounded composition probing protocol.**

---
for a composition request. The probing budget represents the trade-off between the probing overhead and composition optimality. Larger probing budget allows us to examine more candidate service graphs, which allows us to find a better qualified service graph. Thus, our solution can provide an adaptive composition solution with tunable performance by properly adjusting the probing budget. For example, we can use larger probing budget for the request with (1) higher priority, (2) stricter QoS constraints, or (3) more complex function. We can also adaptively adjust the probing budget based on the user feedbacks and historical information.

To achieve efficient composition probing, we associate a probing quota \( \alpha_i \) with each function \( F_i \) in the function graph, which defines the number of duplicated service components to probe for \( F_i \). The probing quota allows us to achieve differentiated allocation of the probes among different functions. For example, we can assign higher probing quota for the function with more duplicated service components.

**Step 2. Distributed probe processing.** Each peer processes a probe independently using only local information until the probe arrives at the destination, illustrated by Figure 5.1 (b). The goal of hop-by-hop distributed probe processing is to collect needed information and perform intelligent parallel searching of multiple candidate service graphs. We will describe this step in detail in Section 5.3.3.

**Step 3. Optimal composition selection.** The destination collects the probes for a request with certain timeout period, illustrated by Figure 5.1 (c). It then selects the best qualified service graph based on the resource and QoS states collected by the probes. We will discuss this step in detail in Section 5.3.4.

**Step 4. Setup service session.** Finally, the destination sends an acknowledge message along the reversed selected service graph to confirm resource allocations and initialize service components at each intermediate peer, illustrated by Figure 5.1 (d). Then the application sender starts to stream application data units along the selected service graph. If no qualified service graph is found, the destination returns a failure message to the source directly.
5.3.2 Efficient Composition Probing

To achieve efficient composition probing, we first introduce the concept of probing budget that allows us to precisely control the number of probes we could use for composition request. The probing budget represents the trade-off between the probing overhead and composition optimality. Larger probing budget allows us to examine more candidate service graphs, which allows us to find a better qualified service graph. Thus, our solution can provide an adaptive composition solution with tunable performance by properly adjusting the probing budget.

For example, we can use larger probing budget for the request with (1) higher priority, (2) stricter QoS constraints, or (3) more complex function. We can also adaptively adjust the probing budget based on the user feedbacks and historical information.

Although the probing budget could control the total probing overhead, it cannot guarantee the fair sharing of the probing budget among different service functions. If there are many candidate service components for each service function, dividing \( \beta_0 \) among all candidate service components can quickly use up the probing budget at early stage of the composition probing. To address the problem, we associate a probing quota \( \alpha_i \) with each service function \( F_i \) to limit the number of service components that can be probed for \( F_i \). In the basic distributed service composition algorithm, we assume that all service functions are equally important. We will describe the differentiated probing quota allocation in Section 5.3.5. Let us assume that we are given a function graph \( \xi \) includes \( k \) branch paths \( \tau_1, ..., \tau_k \), each of which includes \( L_i \) service functions with \( Z_i \) \((Z_i \geq 1, 1 \leq i \leq k)\) different permutations (i.e., composition patterns). If each service function is associated with the same probing quota \( \alpha \), then the total probes generated on the branch path with \( L_i \) service functions and \( Z_i \) permutations is \( Z_i \cdot \alpha^{L_i} \). Thus, we can derive \( \alpha \) based on the following inequalities:

\[
Z_i \cdot \alpha^{L_i} \leq \beta_0/k, \quad 1 \leq i \leq k.
\] (5.1)
5.3.3 Per-hop Probe Processing

We now describe the per-hop probe processing algorithm at a peer $v_i$ which is illustrated by Figure 5.2. The per-hop probe processing mainly includes four steps:

**Step 2.1 Resource/QoS check and soft resource allocation.** When a peer receives a probe, it first check whether the QoS and resource values of the probed service graph already violate the user’s requirements. If the accumulated QoS and resource values already violate the user’s requirements, the probe is dropped immediately. Otherwise, the peer will temporarily allocate required resources to the expected application session. However, the resource allocation is *soft* since it will be cancelled after certain timeout period if the peer does not receive a confirmation message. The purpose of this soft resource allocation is to avoid conflicted resource admission caused by concurrent probe processing. Thus, we can guarantee that the probed resources are still available at the end of the probing process.

**Step 2.2 Derive next-hop functions.** The peer derives next-hop functions according to the dependency and commutation relations in the function graph. All the functions dependent on the current function are considered as next-hop functions. For example, in Figure 5.2, the current function $F_1$ has two dependent next-hop functions $F_2$ and $F_3$. For each next-hop function $F_k$ derived above, if there is a exchange link between $F_k$ and $F_l$, then $F_l$ is also considered as a possible next-hop function. For example, in Figure 5.2, since $F_3$ can be commutated with $F_4$, $F_4$ is also considered as the next-hop of $F_1$. The probing budget...
is proportionally distributed among next-hop functions according to their probing quotas.

**Step 2.3 Select next-hop service components.** For each next-hop function $F_k$, $v_i$ first retrieves the meta-data of duplicated service components using the decentralized service discovery substrate described in Section 5.2. Let $\beta_k$ denote the current probing budget for $F_k$. Let $\alpha_k$ define the probing quota allocated for $F_k$. Then the number of probes that can be used by $F_k$ is denoted by $I_k = \min(\beta_k, \alpha_k)$. Let $Z_k$ denote the number of duplicated service components for $F_k$. If $I_k \geq Z_k$, then $v_i$ has enough probing budget to probe all the duplicated service components. Each probe has a new probing budget $\lfloor \frac{\beta_k}{Z_k} \rfloor$. However, if $I_k < Z_k$, we cannot probe all duplicated service components. Then $v_i$ needs to select $I_k$ most promising ones based on the local information. Currently, $v_i$ uses a composite metric for next-hop service component selection, which comprehensively considers various local information such as network delay and available bandwidth to candidate next-hop service components, failure probability of candidate next-hop service components, and others. Finally, $v_i$ spawns $I_k$ new probes to examine the selected next-hop service components. Each new probe has a probing budget $\lfloor \frac{\beta_k}{I_k} \rfloor$.

**Step 2.4 Set probe content.** First, each new probe inherits the QoS and resource states from its parent probe. Then, $v_i$ adds the local QoS states (e.g., $Q^p$ of current-hop service component) and resource states (e.g., available CPU, memory on $v_i$) into the new probe, illustrated by Figure 5.2.

### 5.3.4 Optimal Composition Selection

The destination selects the best qualified service graph based on the information collected by the received probes. If the function graph has a linear path structure, each probe records a complete service composition. However, if the function graph has a DAG structure, each probe only collects the information for one composition branch. For example, in Figure 5.1, each probe traverses either branch $F_1 \rightarrow F_2 \rightarrow F_4$ or $F_1 \rightarrow F_3 \rightarrow F_4$. Thus, we need to first merge the branches into complete service graphs.
Next, the destination selects the qualified service graphs by comparing the QoS states of candidate service graphs with the user’s QoS requirements. Then, the destination selects the best service graph from all the qualified ones based on the load balancing goal. For this purpose, we define a cost aggregation function $\psi^\lambda$ as follows,

$$
\psi^\lambda = \sum_{s_j/v_j \in \lambda} \sum_{i=1}^{n} w_i \cdot \frac{r_{s_j}^i}{r_{\ell_j}^i} + w_{n+1} \cdot \sum_{\ell_j/v_j \in \lambda} \frac{b_{\ell_j}^i}{b_{\ell_j}^{\lambda^i}}
$$

(5.2)

where $r_{s_j}^i$ defines the requirement of $s_j$ for the $i$'th end-system resource type (e.g., CPU, memory), $r_{\ell_j}^i$ defines the current resource availability on the peer $v_j$ for the $i$'th resource type, $b_{\ell_j}^i$ defines the bandwidth requirement on the service link $\ell_j$, $b_{\ell_j}^{\lambda^i}$ defines the bandwidth availability on the underlying overlay network path $\ell_j$, and $w_i$ represents the importance of different resource types. We can customize $\psi^\lambda$ by assigning higher weights to more critical resource types. The rationale of using $\psi^\lambda$ to evaluate the load balancing property of $\lambda$ is that smaller $\psi^\lambda$ means that the available resources along the service graph exceed the required resources by a larger margin. Thus, the service graph with the minimum $\psi^\lambda$ can achieve the best load balancing in the current P2P computing environment.

Finally, the source receives an acknowledge message carrying the information of the best service graph and a set of other qualified service graphs that can be used as backup service graphs for failure recovery, which will be introduced in the next section.

### 5.3.5 Enhanced Composition Probing

We can improve the above basic distributed service composition solution in three aspects: (1) caching, (2) pruning, and (3) differentiated probing quota allocation, which are described as follows,
Caching composition probing results. Each overlay node can cache the qualified service graphs found by recent composition probing operations. When a node \( v_i \) receives a composite service request with the same abstract function, it can avoid invoking the composition probing to find a new service graph if the cached service graph can satisfy the user’s QoS constraints. Each cached service graph is only kept for a short period of time to assure the validity and optimality of the cached service graph. Moreover, before we use the cached service graph, we can send a single composition probe\(^2\) along the cached service graph to validate whether it is still qualified. Thus, we can greatly reduce the composition probing overhead by eliminating unnecessary composition probing operations.

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

Differentiated probing quota allocation. In section 5.3.2, we have described the uniform probing quota allocation scheme, which assumes that all service functions and branch paths are equally important. We now describe a differentiated probing quota allocation scheme, which considers the differences among service functions and branch paths in the

---

\(^2\)If the service graph has \( k \) branch paths, then we need \( k \) composition probes to examine the service graph.

\(^3\)Because the composition probing follows the function constraints specified by the function graph and QoS constraints are minimum optimal, the satisfaction probabilities will not be increased by further accumulations.
function graph. First, we decide the probing quota ratios among different service functions. Suppose the function graph $\xi$ includes $L$ service functions, $\{F_1, \ldots, F_L\}$. We use $\nu_i \cdot \alpha$, $1 \leq i \leq L$, to represent the probing quota allocated to the service function $F_i$, where $\nu_i$ is the probing quota weight associated with the service function $F_i$. We can decide the value of $\nu_i$ based on different policies. For example, we can assign a higher weight to the service function that has more candidate service instances since it needs more probes to search different alternatives.

Suppose a service function $F_i$ can be mapped to $\sigma_i$ different service instances. We can calculate $\nu_i$ as $\nu_i = \sigma_i / \sum_{j=1}^L \sigma_j$. If all service functions have the same number of duplicated service instances, we can assign larger $\nu_i$ to more critical service functions. We can achieve more efficient consumption of the probing budget $\beta_0$ by partitioning $\beta_0$ differentially among various service functions.

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

To implement the above differentiated probing quota allocation in the distributed service composition, we replace the uniform probing budget partition scheme, presented in Section 5.3.2, with a proportional probing budget partition scheme, which is described as follows. When a service node $v_i$ receives a probe whose probing budget is $\beta$ and there are $T$ next-hop service functions $F_1, \ldots, F_T$, $v_i$ proportionally divides $\beta$ among $T$ next-hop service functions as follows. Suppose there are $k_i$ branch paths that are rooted at the service function $F_i$, which are denoted by $\tau_{i_1}, \ldots, \tau_{i_{k_i}}$. Then, the probing budget allocated to $F_i$ is decided by
\[ \beta_i = \left\lfloor \frac{\sum_{j=1}^{k_i} \omega_{ij}}{\sum_{i=1}^{T} \sum_{j=1}^{k_i} \omega_{ij}} \cdot \beta \right\rfloor, \] which means that the proportion of the probing budget allocated to \( F_i \) is decided by the ratio between weight sum of all branch paths rooted at \( F_i \) and the weight sum of all branch paths rooted at all next-hop service functions \( F_1, \ldots, F_T \).

We now prove that the above proportional probing budget partition scheme can guarantee that each branch path \( \tau_i \) receives its share of probing budget \( \omega_i \cdot \beta_0 \).

**Theorem 5.3.1.** Suppose the function graph \( \xi \) includes \( B \) branch paths \( \tau_1, \ldots, \tau_B, B > 1 \). The proportional probing budget partition scheme can guarantee that each branch path \( \tau_i \) is allocated with \( \omega_i \cdot \beta_0 \) probing budget, \( 1 \leq i \leq B \).

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

When \( L = k+1 \), let us consider the first fan-out service function \( F_g \). Suppose \( F_g \) has \( T \) next-hop service functions \( F_1^{\text{next}}, \ldots, F_T^{\text{next}} \). We use \( \xi_i \) to denote the sub-graph of \( \xi \), which is rooted at \( F_i^{\text{next}} \). Suppose \( \xi_i \) includes \( k_i \) branch paths, which are denoted by \( \tau_{i1}, \ldots, \tau_{ik_i} \). Since \( F_g \) is first fan-out service function, the total probing budget allocated to \( F_g \) is \( \beta_0 \). \( F_g \) is the root of all the \( B \) branch paths included in \( \xi \). Thus, we have \( \sum_{i=1}^{T} \sum_{j=1}^{k_i} \omega_{ij} = \sum_{i=1}^{B} \omega_i = 1 \). According to the proportional probing budget partition scheme, the probing budget allocated to \( F_i^{\text{next}} \) is \( \left( \sum_{j=1}^{k_i} \omega_{ij} / \sum_{i=1}^{T} \sum_{j=1}^{k_i} \omega_{ij} \right) \cdot \beta_0 = \sum_{j=1}^{k_i} \omega_{ij} \cdot \beta_0 \). Since the length of \( \xi_i \) is at most \( k \), the induction hypothesis applies to \( \xi_i \). Each branch path \( \tau_{ij}, 1 \leq j \leq k_i \) included in \( \xi' \) is allocated with probing budget \( w'_i \cdot \beta'_0 \), where \( w'_i = \omega_{ij} / \sum_{j=1}^{k_i} \omega_{ij} \) and \( \beta'_0 = \sum_{j=1}^{k_i} \omega_{ij} \cdot \beta_0 \). Thus, the probing budget allocated to \( \tau_{ij} \) is \( \left( \omega_{ij} / \sum_{j=1}^{k_i} \omega_{ij} \right) \cdot \left( \sum_{j=1}^{k_i} \omega_{ij} \cdot \beta_0 \right) = \omega_{ij} \cdot \beta_0 \). Thus, the theorem holds. \( \square \)
5.4 Proactive Failure Recovery

P2P SpiderNet adopts a proactive approach to maintaining the quality of composed services during service runtime. The source maintains a small number of backup service graphs for each active service session. Thus, the source can quickly recover failures by switching from the broken service graph to one of the backup service graphs\(^4\). For example, in Figure 5.3, the source maintains one backup service graph for the service session. When the current service graph (shown in solid line) fails, the source can quickly switch to the backup service graph, shown in dash line, to recover the failure\(^5\). Different from the conventional multi-path routing, P2P SpiderNet does not send real application data along multiple service graphs to achieve fault tolerance. Instead, the source only periodically sends low-rate measurement probes along these backup service graphs to monitor their liveness and QoS/resource conditions. The low-rate probing data is defined as the service graph maintenance overhead. The reactive failure recovery is triggered only when all backup service graphs become unqualified as well, which will invoke the BCP protocol to find a new qualified service graph.

The advantages of the proactive failure recovery are two fold. First, we can achieve fast failure recovery by avoiding the delay of finding a new service graph if the backup service

\(^4\)Due to the space limitation, we omit the discussion about the failure detection design details.

\(^5\)We assume that the service component is either stateless or has only soft states that can be recovered quickly by software.
A graph can be used, which is especially important for soft real time applications such as multimedia streaming. Second, we can reduce failure recovery overhead by avoiding invoking the relatively expensive BCP protocol to find a new service graph. Because P2P systems are highly dynamic, service graphs are prone to failures, especially for long-lived applications such as multimedia streaming. Thus, it is worthwhile to pay a small maintenance overhead for both fast failure recovery and greatly reduced composition probing overhead.

To achieve efficient proactive failure recovery, we need to answer two key questions: (1) how many backup service graphs should be maintained for a specific service session; and (2) which qualified service graphs should be selected as backup service graphs, which are described in the following sections.

5.4.1 Backup Service Graph Number

If we maintain too many backup service graphs, the maintenance overhead will be large. If we maintain too few backup service graphs, we may fail in providing the required quality assurances and failure resilience. Thus, the number of backup service graphs represents the trade-off between the maintenance overhead and the quality of service composition. For efficiency, P2P SpiderNet adopts an adaptive approach to deciding the number of backup service graphs based on the relationship between the QoS $Q^\lambda$ and failure probability $F^\lambda$ of the current service graph $\lambda$ and the user required QoS $Q^{req}$ and failure probability $F^{req}$.

Intuitively, if the QoS and failure probability of the current service graph are much better than the user’s requirements, we could just maintain a few backup service graphs. Otherwise, we need to maintain more backup service graphs to achieve required QoS assurances and failure resilience. Formally, we can calculate the number of backup service graphs, denoted
by $\gamma$, as follows,

$$\gamma = \min\left(\lfloor U \cdot \left( \sum_{i=1}^{m} \frac{q^\lambda_i}{q^\lambda_{req}} + \frac{F^\lambda}{F^\lambda_{req}} \right) \rfloor, (C - 1) \right) \quad (5.3)$$

where $U$ is a configurable system parameter defining the upper bound of backup service graph number, and $C$ represents the total number of qualified service graphs found by initial service composition using BCP.

### 5.4.2 Backup Service Graph Selection

Given the number of backup service graphs, we need to decide which qualified service graphs should be selected as backup service graphs. On one hand, we want to achieve failure resilience, which implies that backup service graphs should be disjoint for failure independence; On the other hand, we want to achieve real time failure recovery, which implies that the backup service graph should be overlapped with the current service graph for fast failure recovery. Hence, we should carefully consider the tradeoff between the above two conflicting requirements. P2P SpiderNet selects backup service graphs as follows:

- For each service component $s_i$ on the current service graph $\lambda$, we select a qualified service graph that does not include $s_i$ but has the largest overlap (i.e., largest number of common service components) with $\lambda$. Hence, if $s_i$ fails, we can quickly recover $\lambda$ by switching to the above backup service graph with lowest overhead.

- In order to handle multiple concurrent service component failures, we continue to select backup service graphs, which do not include every two service components, every three service components, and so forth.

- Under the constraint of backup service graph number, we may not be able to include all of the above desired backup service graphs. Thus, we start from selecting backup
Figure 5.4: P2P SpiderNet node software architecture.

service graphs for recovering bottleneck service components, which have the largest failure probabilities.

5.5 Performance Evaluation

In this section, we evaluate the performance of P2P SpiderNet using both large-scale simulations and prototype implementation evaluated in the wide-area network testbed, called PlanetLab [6].

5.5.1 Prototype Implementation and Evaluation

We have implemented a prototype of the P2P SpiderNet system. Each P2P SpiderNet node software is a multi-threaded running system written in about 13K lines of java code. Figure 5.4 illustrates the software architecture for one P2P SpiderNet node. There are 6 major
modules: (1) service lookup agent is responsible for discovering the list of service instances, which is implemented on top of the Pastry software [67]; (2) service graph composition module performs initial service graph finding and runtime service graph maintenance; (3) session manager maintains session information for current active sessions; (4) data transmission module is responsible for sending, receiving, and forwarding application data; (5) overlay topology manager maintains the neighbor set; (6) monitoring module is responsible for monitoring the network/service states of neighbors.

As proof-of-concept, we also implemented a set of multimedia service components to populate our P2P service overlay. Each service component provides one of the following six functions: (1) embedding weather forecast ticker; (2) embedding stock ticker; (3) upscaling video frames; (4) downscaling video frames; (5) extracting sub-image; and (6) re-quantification of video frames. We deploy one service component on each P2P SpiderNet node, which is randomly selected from the above six multimedia service components. Our experiments use 102 Planetlab hosts that are distributed across U.S. and Europe. Thus, the average replication degree of each multimedia service is $102/6 = 17$. We then implement a customizable video streaming application on top of the P2P SpiderNet service composition system. The customizable video streaming application allows the user to perform wide-area P2P video streaming with desired transformations and enriched content. We have deployed and evaluated the P2P SpiderNet system with the video streaming application on the wide-area network testbed PlanetLab [6]. The end-application on each node periodically submits random service composition requests to the P2P SpiderNet system.

First, we measure the service session setup time in the wide-area network, which includes (1) decentralized service discovery time, (2) initial service graph finding time using the bounded composition probing protocol, and (3) service session initialization time. Figure 5.5 illustrates the average service session setup time using more than 500 requests generated from 102 different PlanetLab hosts. The current prototype of the P2P SpiderNet system can setup a service session within several seconds, which is acceptable for long-lived streaming
applications that usually lasts tens of minutes or several hours. The above service setup time can be reduced with implementation implements and tunings.

Second, we compare the QoS provisioning performance of P2P SpiderNet with the random and optimal algorithms. We consider service composition requiring three different functions. We ask different approaches to find the best qualified service composition that has minimum end-to-end service delay. Because each service function has on average 17 instances, the average number of probes required by the optimal algorithm is $17^3 = 4913$. As shown by Figure 5.6, the average service delay of the service graphs discovered by the P2P SpiderNet reduces with a growing probing budget. When the probing budget is very low, P2P SpiderNet degenerates into the random algorithm, so the overhead is low, but the service quality is not satisfactory. When larger probing budget is allowed, the service graph quality improves, and when the probing budget reaches a certain threshold, it asymptotically approaches the optimal performance. However, P2P SpiderNet can achieve near optimal performance with much lower overhead (i.e., $200/4913 = 4\%$) than the unbounded flooding scheme performing exhaustive searching.

Figure 5.5: Service session setup time in wide-area networks.

Figure 5.6: Performance comparison among random, P2P SpiderNet, and optimal algorithms.
5.5.2 Simulation Results

We have implemented an event-driven multimedia service overlay simulator using C++. The simulator first uses the degree-based Internet topology generator Inet-3.0 [81] to generate a power-law graph with 3200 nodes to represent the Internet topology.\(^7\) To construct the multimedia service overlay, we randomly select certain number nodes as overlay nodes and connect them into certain overlay network. We have used two different overlay topologies in our experiments: (1) mesh topology, where each node has an equal node degree \(\theta\), and (2) power-law graph topology\(^8\), where node degrees follow a power-law distribution\(^9\). We use \(\theta = |V| \cdot 10\%\) in our experiments. Once the node degrees are chosen, the nodes are connected into a topologically-aware overlay network using the Short-Long algorithm presented in [66].

To simulate the dynamic QoS values, we generate the dynamic QoS values using either uniform distribution function or normal distribution function. The histogram for each random variable includes 30 sample values and 10 bins. We choose the mean and deviation values based on real-world Internet service level agreement (SLA) contract and the profiling results of fully implemented multimedia services. We simulate the IP-layer and overlay-layer data routing using the shortest path routing algorithm. The P2P SpiderNet distributed service composition is then performed on top of the overlay data routing. Each overlay node provides two service components. Each service component performs a service function that is randomly selected from \(\lfloor |V| / 5 \rfloor\) service functions. Thus, the average service duplication ratio is \(2 |V| / |V|^5 = 10\), which conforms to our assumption that a service function can be mapped to a limited number of service instances. The function graph \(\xi\) of the request is randomly selected from 200 pre-defined templates, which include two to five service functions with one or two branch paths. The statistical resource and QoS requirements are uniformly distributed. Dur-

\(^7\)We choose Inet-3.0 instead of other Internet topology generator because recent research [75] has shown that the degree-based Internet topology generator can most accurately capture the current Internet structure.

\(^8\)Recent study [69] has shown that real-world peer-to-peer overlay networks, such as Gnutella [4], have the power-law graph topology.

\(^9\)If one ranks all nodes from the most connected to the least connected, then the \(i^{th}\) most connected node has \(\omega / i^{\alpha}\) neighbors, where \(\alpha\) and \(\omega\) are constants. We set \(\omega = 3 \cdot \alpha\) and \(\omega = 1.5\) to make its average node degree equal to \(\theta\) for the purpose of comparison.
Figure 5.7: QoS-aware service composition performance comparison among different approaches.

Figure 5.8: QoS-aware service composition performance on three mesh overlays with 200, 500, and 1000 nodes.

ing each time unit, certain number of composite service requests are generated. Each service session lasts 5 to 15 time units. A QoS-aware service composition is said to be successful, if and only if the composed service graph (1) satisfies the function graph requirements, (2) satisfies the user’s resource requirements (e.g., CPU, network bandwidth), and (3) satisfies the user’s QoS requirements (e.g., delay, data loss rate). The composition success rate is calculated by \( \frac{\text{SuccessNumber}}{\text{RequestNumber}} \).

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

First, we compare the performance of different algorithms. Figure 5.7 illustrates the composition success rate achieved by different algorithms under different workload conditions. We used two variations of our scheme, “probing-0.2” and “probing-0.1”, which uses 20% and 10% of the probes required by the optimal algorithm, respectively. Each round of simulation
lasts 2000 time units. Each success rate is averaged over all the requests generated during 2000 time units simulation duration. We observe that P2P SpiderNet can achieve near-optimal performance with much lower overhead, which is also much better than the random and static algorithms. We also conduct the same experiments on a 500 node multimedia service overlay with power-law graph topology. The results also show the similar trend.

Second, we evaluate how the performance and overhead of P2P SpiderNet scale with the sizes of the multimedia service overlay. We show the results on the mesh overlay topology only. The results on the power-law graph topology demonstrate the similar trend. We use three different multimedia service overlays with 200, 500, and 1000 nodes, respectively. Figure 5.8 shows the composition success rate achieved by different algorithms on the three different multimedia service overlays. The results illustrate that P2P SpiderNet can consistently achieve near-optimal performance (i.e., > 95% of the optimal performance) on the three different service overlays. Compared to the random and static algorithms, P2P SpiderNet can achieve as much as 300% better performance than the random algorithm and 400% better performance than the static algorithm. Moreover, P2P SpiderNet presents much better scaling property than the random and static algorithms. When we increase the service overlay size from 200 nodes to 500 nodes, P2P SpiderNet can achieve as much as 130% performance improvements by efficiently utilizing added resources while the random and static algorithm can achieve at most 50% improvements. The improvements from 500 nodes to 1000 nodes are not too much since the system resources of 500 nodes already meet the resource requirements of the workload.

Third, we evaluate the overhead of the P2P SpiderNet system. Figure 5.9 illustrates the probing overhead comparison between the optimal algorithm and the P2P SpiderNet algorithm on the three different multimedia service overlays. The probing overhead is measured by the total number of probing messages generated in the whole service overlay during each time unit. The results show that P2P SpiderNet has much lower probing overhead than the optimal algorithm. The overhead increasing rate of the P2P SpiderNet algorithm is also
slower than that of the optimal algorithm as we increase the size of the service overlay. Figure 5.10 illustrates the total system overhead comparison between P2P SpiderNet and the conventional centralized approach. The total system overhead of the centralized approach is calculated by $|V|^2$, $|V| = 200, 500, 1000$, which represents the system overhead lower-bound required by the global states update assuming the states update can be finished in one time unit. The total system overhead of P2P SpiderNet includes two parts: local states update and probing overhead. The local states update overhead is calculated as $|V|^2 \cdot 10\%$ since each node has $|V| \cdot 10\%$ neighbors. The results demonstrate that P2P SpiderNet has much lower overhead than the centralized approach. The overhead reduction becomes larger as we increase the overlay size. Thus, P2P SpiderNet has much better scaling property than the centralized approach.

Second, we evaluate the efficiency of our proactive failure recovery scheme. We use the metric “failure frequency” to define the number of failures occurred during each time unit. Figure 5.11 illustrates the failure frequency in a dynamic P2P network where 1% of peers randomly fail during each time unit. We observe that by maintaining on average 2.74 backup service graphs per session, the proactive failure recovery can recovery almost all the failures. We then conduct the same experiments on a highly dynamic P2P network, where 10% of peers fail randomly during each time unit. Figure 5.12 illustrates the failure frequency
Figure 5.11: Failure frequency comparison in a dynamic P2P network.

Figure 5.12: Failure resilience of P2P SpiderNet under P2P network topology variation rate 100 peers/time unit.

results under the highly dynamic P2P network. We observe that P2P SpiderNet can recover on average 50% failures by maintaining on average 2.74 backup service graphs.

5.6 Summary

In this chapter, we have presented P2P SpiderNet, a self-organizing, fully decentralized design for the core SpiderNet. The major contributions of the P2P SpiderNet are summarized as follows. First, P2P SpiderNet provides fully decentralized QoS-aware and resource-efficient service composition using bounded composition probing. Second, P2P SpiderNet provides proactive failure recovery to achieve failure resilient service composition. Third, P2P SpiderNet achieves flexible service composition by supporting directed acyclic graph composition topologies and considering exchangeable composition orders to enhance the composed service’s quality. Finally, we demonstrate the feasibility and efficiency of the P2P SpiderNet system using both large-scale simulations and prototype implementation.
Chapter 6

A Comparative Study for the Core SpiderNet

In this chapter, we compare different design alternatives for the core SpiderNet. We first compare different initial service composition and failure recovery algorithms used by the utility SpiderNet and P2P SpiderNet. We then compare different overlay topology alternatives for constructing the service overlay network. Our comparative study is conducted in the context of QoS-aware service path composition in a dynamic service overlay network.

6.1 Introduction

In the previous chapters, we have presented two different core SpiderNet designs, utility SpiderNet and P2P SpiderNet. We now present a systematic comparative study for the problem of providing QoS-aware service path composition in dynamic service overlay networks. The comparison is conducted in three dimensions:

- First, we compare two different initial service composition approaches: (1) global-view-based centralized approach, denoted by $GC$, which is used by the utility SpiderNet; and (2) local-view-based decentralized approach, denoted by $LD$, which is used by the P2P SpiderNet. The GC scheme assumes that the service composition decision-maker, called service composer, has the global states information about the entire service
overlay network. Based on the global information, GC then uses a centralized algorithm (i.e., QSC-enhanced algorithm presented in Chapter 3) to calculate the best qualified service path. In contrast, the LD approach only needs each peer to maintain service and network states about its neighbors. The QoS-aware service path finding is performed in a fully distributed fashion, which includes local computation on local states only. Our theoretical analysis and simulation experiments illustrate the performance and scalability properties of these two different designs.

- Second, we compare the reactive failure recovery approach used by the utility SpiderNet with the proactive failure recovery approach used by the P2P SpiderNet. Long lived composed services such as multimedia streaming are prone to failures due to dynamic changes in service overlay networks (e.g., node/link failures). Thus, runtime failure recovery are necessary for providing failure-resilient services. Thus, we compare two different failure recovery mechanisms, reactive failure recovery that re-calculates a new service path after failure is detected, and proactive failure recovery that replaces a failed service path with a maintained backup service path.

- Third, we study the effect of service overlay network topology on the performance of QoS-aware service composition. The overlay topology of a dynamic service overlay network directly affects the efficiency of overlay data routing (e.g., inter-node delay, inter-node loss rate, inter-node bandwidth). Then the efficiency of overlay data routing can significantly affect the performance of QoS-aware service composition. Thus, we conduct extensive simulations with a wide range of topology parameters to illustrate how the performance of QoS-aware service composition depends on the service overlay network topology.

The rest of this chapter is organized as follows. First, we analytically compare two different design alternatives for the initial QoS-aware service composition. Second, we analytically compare two different service composition failure recovery approaches. Third, we investigate
how different overlay topologies affect the performance of QoS-aware service composition. Fourth, we present the experimental comparisons among different design alternatives, which further validate and quantify our analytical comparisons. Finally, we conclude this chapter.

6.2 Initial Service Composition Alternatives

We have presented two different approaches to initial service composition Chapter 5: (1) global-view-based centralized approach (or GC algorithm) presented in Chapter 3, which assumes that the service composer has the global states information about the service overlay network and uses a centralized heuristic algorithm to find the best qualified service path; and (2) local-view-based distributed approach (or LD algorithm) presented in Chapter 5, which assumes that each peer node only has local information and uses a distributed probing-assisted heuristic algorithm to find the best qualified service path. In this section, we qualitatively compare the above two different approaches in terms of response time, performance and overhead.

**Responsiveness.** Under the assumption of global view at each peer, the centralized heuristic algorithm can quickly respond to the user’s composed service request with a qualified service path or a failure message. The distributed algorithm, however, has a relatively longer response time due to the composition probing delay in the wide-area network. We have measured the response time of the distributed service composition algorithm with our initial prototype in the wide-area network testbed PlanetLab [6]. It takes about several seconds for the composition probing to find a service path across wide-area networks (see Chapter 5). It is acceptable for the setup of a service session that usually lasts for tens of minutes or several hours.

**Performance.** Because the problem of finding optimal QoS-aware service path is NP-complete, even GC algorithm can only provide approximate solutions. In contrast, the distributed approach provides us with a full spectrum of tunable solutions. If we set the
probing budget as 1, the composition probing algorithm becomes random service composition if we use random next-hop selection, or greedy service composition if we use greedy next-hop selection. If we set the probing budget as infinite, the composition probing algorithm becomes exhaustive searching (i.e., flooding) that can provide the optimal solution. In-between, we can have all kinds of tradeoff between the probing overhead and system performance.

**Overhead.** The major overhead of the GC algorithm comes from the global view maintenance. To keep a consistent global view at each peer, we need to perform either periodical or triggered flooding of each peer’s states. Different from the conventional network routing, QoS-aware service composition needs not only link states but also service states. Moreover, unlike the IP-layer network where routers are relatively stable, dynamic service overlay networks are highly dynamic with arbitrary peer arrivals and departures. Thus, to maintain the up-to-date view of the dynamic service overlay network, the frequency of states updating must be high. Otherwise, stale states can cause the algorithm to find sub-optimal service path [46]. The major overhead of distributed algorithm comes from the on-demand composition probing. By leveraging the scalable P2P lookup service, the composition probing only sends probes to those functionally fulfilling peers. The composition probing is triggered by user requests. Thus, the probing overhead is decided by both user request rate and the probing budget, which decides the maximum number of generated probes during one round of BCP. The probing budget depends on the number of candidate service paths, which is decided by the number of redundant service instances and the number of service tasks on the service path. We now quantify the above overhead analysis as follows.

**Definition 6.2.1.** The GC algorithm overhead is defined as the total number of states update messages generated per minutes \(^1\) in the whole dynamic service overlay network.

**Theorem 6.2.1.** In a dynamic service overlay network \(G = (V, E)\), if at each peer, the state update times per minute is at least \(U\), then the lower-bound of the GC algorithm overhead is \(U|V|(|V| + |E|)\).

\(^1\)For simplicity, we use minute as time unit in the whole chapter.
Proof. The global states at each peer must include states of all peers whose number is $|V|$ and all overlay links whose number is $|E|$. Thus, each peer must receive at least $|V| + |E|$ messages to update the states of all peers and overlay links. Because each state is updated by at least $U$ times per minutes, the number of messages received by each peer per minutes is at least $U(|V| + |E|)$. The total number of messages received by all peers must be no less than $U|V|(|V| + |E|)$. Because the number of messages generated should be no less than the messages received, the global states maintenance overhead is at least $U|V|(|V| + |E|)$.

One special case is a dynamic service overlay network that adopts periodical global states update. If the update period is $M$ minutes (e.g., 0.5 minutes), then the update times per minutes is $\lfloor \frac{1}{M} \rfloor$ (e.g., 2 times). If the average node degree in the dynamic service overlay network is $\bar{\sigma}$, then we have $|E| = \bar{\sigma} \cdot |V|$. Based on the Theorem 6.2.1, we have the following corollary immediately.

**Corollary 6.2.1.** In a dynamic service overlay network $G = (V, E)$ that uses periodical global states update with a uniform update period $M$ minutes, the lower-bound of the GC algorithm overhead is $\lfloor \frac{|V|^2 \cdot \bar{\sigma}}{M} \rfloor < \lfloor \frac{|V|^2 \cdot (1 + \bar{\sigma})}{M} \rfloor$.

**Proof.** With the uniform update period $M$ minute, the state update times per minute at each peer is $U = \lfloor \frac{1}{M} \rfloor$ times per minute. Thus, according to Theorem 6.2.1, the lower-bound of the GC algorithm is $\lfloor \frac{|V|(|V| + |E|)}{M} \rfloor$. Because $|E| = \bar{\sigma} \cdot |V|$, we have $\lfloor \frac{|V|(|V| + |E|)}{M} \rfloor = \lfloor \frac{|V|^2 \cdot (1 + \bar{\sigma})}{M} \rfloor$, which is larger than $\lfloor \frac{|V|^2 \cdot \bar{\sigma}}{M} \rfloor$.

We now give the overhead upper-bound of the LD algorithm.

**Definition 6.2.2.** The LD algorithm overhead is defined as the total number of probing messages generated per minute and the total number of local states update messages generated per minute in the whole dynamic service overlay network.

**Theorem 6.2.2.** Given a dynamic service overlay network $G = (V, E)$, if the peak user request rate is $Z$ requests per minute, the maximum initial probing budget is $\beta_{\text{max}}$, the average
node degree is \( \bar{\sigma} \), the minimum local states update period is \( T \) minutes, then the overhead upper-bound of the LD algorithm is
\[
Z \cdot \beta_{\text{max}} + \left\lfloor \frac{|V| \cdot \bar{\sigma}}{M} \right\rfloor.
\]

**Proof.** Because the composition probing is triggered by the user request, the number of probing operations invoked per minute is no larger than \( Z \). Because the composition probing algorithm guarantees that the number of probe messages generated during one round of probing is no larger than its probing budget whose upper-bound is \( \beta_{\text{max}} \), the upper-bound of the total probing messages generated per minute is \( Z \cdot \beta_{\text{max}} \). Because each peer has on average \( \bar{\sigma} \) neighbors and the minimum update period is \( M \), the total number of local states update messages generated in the whole dynamic service overlay network is \( \left\lfloor \frac{|V| \cdot \bar{\sigma}}{M} \right\rfloor \). Thus, the distributed algorithm overhead is
\[
Z \cdot \beta_{\text{max}} + \left\lfloor \frac{|V| \cdot \bar{\sigma}}{M} \right\rfloor.
\]

Based on Corollary 6.2.1 and Theorem 6.2.2, we can compare the overhead between the GC algorithm and distributed algorithm. The GC algorithm has larger overhead than the distributed algorithm if
\[
\left\lfloor \frac{|V|^2 \cdot \bar{\sigma}}{M} \right\rfloor > Z \cdot \beta_{\text{max}} + \left\lfloor \frac{|V| \cdot \bar{\sigma}}{M} \right\rfloor \tag{6.1}
\]

Given a large-scale dynamic service overlay network where \( |V|^2 \gg |V| \), we can simplify Inequality 6.1 as follows,
\[
|V|^2 > Z \cdot \beta_{\text{max}} \cdot \frac{M}{\bar{\sigma}} \tag{6.2}
\]

A large-scale dynamic service overlay network can include thousands of nodes, which makes \( |V|^2 \) a huge value. On the other hand, dynamic service overlay networks are highly dynamic, which implies that \( M \) should be small in order to keep track of frequent peer arrivals/departures. \( \beta_{\text{max}} \) is related to the number of candidate service paths for each abstract service graph, which depends on how many service instances each service task can be mapped to, and how many service tasks can be included on a service path. As we will shown
in Section 6.5, the distributed algorithm can achieve near-optimal performance with only a small probing budget. \( \bar{\sigma} \) depends on the overlay topology. Thus, the distributed algorithm has less overhead than the GC algorithm in a large-scale dynamic dynamic service overlay network if \( Z < \frac{|V|^2 \cdot \bar{\sigma}}{\beta^{max} \cdot M} \). For example, if \( |V| = 1000, \bar{\sigma} = 100, \beta^{max} = 100, \) and \( M = 0.5 \) minutes, then the distributed algorithm has more overhead than the GC algorithm only if \( Z > 2 \cdot 10^6 \). In fact, the upper bound of the distributed algorithm overhead can be lower when we considering the caching of recently found service paths, which can reduce the composition probing overhead if a cached service path can satisfy the user’s requirements. Thus, generally speaking, the distributed QoS-aware service composition algorithm has much less overhead than the GC algorithm in a large-scale dynamic dynamic service overlay network.

### 6.3 Failure Recovery Alternatives

The original service path can become failed or experience serious quality degradations during runtime due to peer departures, service instance deletions, service outages or significant quality degradations. Thus, we need to provide failure recovery mechanisms in order to provide fault-resilient service composition. There are two major recovery approaches: (1) **reactive failure recovery**, where we calculate a new service path on-demand to repair to the broken service path, (2) **proactive failure recovery**, where we maintain a small number of good backup service paths throughout the service session, which can used immediately for recovering the broken service path. Reactive failure recovery is more suitable for centralized approach where service path recalculation is fast. In contrast, proactive failure recovery is more suitable for decentralized approach because service path recalculation is usually too slow for recovering failures in real time streaming applications. In this section, we present the above two different failure recovery designs and analytically compare them. We first qualitatively compare two different reactive failure recovery approaches, complete re-calculation and partial re-calculation. The advantage of complete re-calculation algorithm
is that it may find another qualified service path with higher probability than the partial re-calculation algorithm because it has more choices of service instances. The disadvantage of the complete re-calculation algorithm is that it may find a very different service path from the original service path, which can lead to higher failure recovery complexity because of more service instance replacement. Thus, the selection between the complete re-calculation and partial re-calculation is the trade-off between the failure recovery success rate and the failure recovery complexity. In our previous study [41], we found that reactive failure recovery using the partial re-calculation algorithm can achieve better tradeoff between the failure recovery success rate and the failure recovery complexity than that using the complete re-calculation algorithm.

In contrast to the reactive failure recovery approach, proactive failure recovery does not re-calculate a new service path on-the-fly, but rather replaces the broken service path with a pre-determined backup service path. The more backup service paths we maintain, the higher failure recovery success rate we can achieve. However, more backup service paths also means higher failure recovery overhead because we need to constantly send measurement probes along more backup service paths. Similar to the probing budget in the decentralized QoS-aware service composition algorithm, the number of maintained backup service paths decides a spectrum of tradeoff between the failure recovery success rate and failure recovery overhead in the proactive failure recovery algorithm. We now quantify the proactive failure recovery overhead as follows.

**Definition 6.3.1.** The proactive failure recovery overhead is defined as the number of measurement probe messages generated per minute in the whole dynamic service overlay network.

**Theorem 6.3.1.** Given a dynamic service overlay network \( G = (V,E) \), if peak request rate is \( A \) requests per minute, the maximum session duration is \( D \) minutes, the maximum number of maintained backup service paths per session is \( \Gamma \), the maximum probing budget is \( \beta_{\text{max}} \gg \Gamma \), and the minimum maintenance period is \( T \) minutes, then the upper-bound of the proactive
failure recovery overhead is $\lfloor \frac{A \cdot D \cdot \Gamma}{T} \rfloor$.

Proof. The number of backup service paths maintained per session is $\min(\Gamma, \Theta - 1)$, where $\Theta$ represents the number of qualified service paths found by the composition probing. Because $\Theta$ does not exceed the maximum probing budget $\beta^{\text{max}}$, $\min(\Gamma, \Theta - 1)$ should be less than $\min(\Gamma, \beta^{\text{max}})$. Because $\beta^{\text{max}} \gg \Gamma$, the number of backup service paths maintained per session is no larger than $\Gamma$. Each session generates at most $\lfloor \frac{\Gamma}{T} \rfloor$ measurement probes per minute to maintain its backup service paths. Given the peak user request $A$ and maximum session duration $D$, the maximum number of active sessions during a minute is $A \cdot D$. Thus, the maximum number of backup maintenance messages generated per minute is $\lfloor \frac{A \cdot D \cdot \Gamma}{T} \rfloor$. \qed

6.4 Overlay Topology Alternatives

In this section, we introduce different design alternatives for the P2P overlay topology and analyze their effects on the QoS-aware composed service delivery. For this purpose, we define three metrics to characterize the efficiency of an overlay topology: (1) overlay delay, which is defined as the average inter-node delay on the overlay network. The inter-node overlay delay is decided by the inter-node overlay path, which consists of a set of overlay links. The overlay delay can be larger than the average inter-node delay on the IP-layer network due to the additional application-level forwarding; (2) overlay loss rate, which is defined as the average inter-node loss rate on the overlay network; and (3) overlay bandwidth, which is defined as the average inter-node available bandwidth on the overlay network.

The efficiency of an overlay topology can affect the QoS provisioning of composed service because the above overlay efficiency metrics directly affect the end-to-end QoS of composed service paths. We now discuss different overlay topology construction approaches in three aspects:

Topology type. An extreme case is that we do not control the overlay topology at all. Indeed, most current service overlay networks such as Gnutella P2P file swapping system
[4] falls into this category. Recent measurement study [69] has shown that Gnutella dynamic service overlay network has power-law topology. If one ranks all nodes from the most connected to the least connected, then the $i^{th}$ most connected node has $\omega/i^\alpha$ neighbors, where $\omega$ and $\alpha$ are constants. However, the power-law topology is not suitable for service composition because the high rank nodes can easily become performance bottleneck. Our experimental results validate this hypothesis. The other common overlay topology type is mesh topology [24, 72, 10], where node degree is equal among all peers. Our experimental results illustrate that mesh topology is most suitable for streaming application because it presents best balancing between routing resilience and bandwidth sharing.

**Node degree.** After deciding the topology type, we need to decide the node degree of the dynamic service overlay network. An extreme case is a complete graph mesh [10], where each peer is directly connected with all other peers. The complete graph mesh provides the best routing performance since the data communication between any two peers does not involve extra application-level data relays unless their direct connection is broken or inferior to other overlay paths. However, maintaining a complete graph mesh has serious scalability problem because each node has to maintain up-to-date states of all other peers. In a large-scale dynamic service overlay network with thousands of peers, the complete graph mesh is impractical. Thus, a large-scale overlay network should have bounded node degree. However, in the bounded-degree overlay mesh, additional application-level forwarding becomes mandatory when two peers are not directly connected in the overlay mesh. Hence, the decision about the overlay mesh degree is the tradeoff between the monitoring overhead and overlay routing efficiency. We have conducted the experiments to study the effects of the mesh degree on the performance of QoS-aware service composition, which will be presented in Section 6.5.

**Neighbor selection.** Given the node degree, each peer needs to decide which peers should be selected as its neighbors. A simple scheme is to perform random selection. However, the random selection can cause routing inefficiencies because two directly connected
peers can be actually far away. For example, a peer in New York randomly select two neighbors, which happen to be in London. Thus, the data sent by the New York peer to a Boston peer has to be relayed by the distant London peer. In order to achieve efficient overlay routing, the topology of the overlay network should be congruent with the underlying IP-level network topology. Thus, people have proposed heuristic algorithms for the topologically-aware overlay construction. In this chapter, we use the “short-long” algorithm presented in [66] to construct the topologically-aware overlay network. We have conducted experiments to evaluate how the performance of QoS-aware composed service depends on different neighbor selection schemes, which will be presented in Section 6.5.

### 6.5 Experimental Comparison

In this section, we present the experimental comparison among the above described different design alternatives. The experimental results quantify and validate our analytical comparisons. We first describe our experimental setup, followed by the definitions of comparison metrics used in the experiments. Then we present and analyze the experimental results.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Degree</th>
<th>Neighbor</th>
<th>Delay</th>
<th>Loss rate</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rmesh0.1</td>
<td>mesh</td>
<td>50</td>
<td>random</td>
<td>151 ms</td>
<td>0.0175</td>
<td>1461 Kbps</td>
</tr>
<tr>
<td>Rmesh0.3</td>
<td>mesh</td>
<td>150</td>
<td>random</td>
<td>119 ms</td>
<td>0.0139</td>
<td>1576 kbps</td>
</tr>
<tr>
<td>Rmesh0.5</td>
<td>mesh</td>
<td>250</td>
<td>random</td>
<td>108 ms</td>
<td>0.0124</td>
<td>1641 kbps</td>
</tr>
<tr>
<td>Tmesh0.1</td>
<td>mesh</td>
<td>50</td>
<td>short-long</td>
<td>124 ms</td>
<td>0.0148</td>
<td>1558 kbps</td>
</tr>
<tr>
<td>Tmesh0.3</td>
<td>mesh</td>
<td>150</td>
<td>short-long</td>
<td>108 ms</td>
<td>0.0125</td>
<td>1647 kbps</td>
</tr>
<tr>
<td>Tmesh0.5</td>
<td>mesh</td>
<td>250</td>
<td>short-long</td>
<td>102 ms</td>
<td>0.0117</td>
<td>1687 kbps</td>
</tr>
<tr>
<td>Rpowerlaw</td>
<td>power-law</td>
<td>50 (avg)</td>
<td>random</td>
<td>266 ms</td>
<td>0.0303</td>
<td>1291 kbps</td>
</tr>
<tr>
<td>Tpowerlaw</td>
<td>power-law</td>
<td>50 (avg)</td>
<td>short-long</td>
<td>212 ms</td>
<td>0.0246</td>
<td>1386 kbps</td>
</tr>
</tbody>
</table>

Table 6.1: Topologies used in experiments.
6.5.1 Experimental Setup

The experimental comparisons are performed by large-scale extensive simulations under a wide range of parameters. The simulation setup is described as follows. First, we use the degree-based Internet topology generator Inet-3.0 [81] to generate a 5000 node IP-layer network\(^2\). Each physical network link is assigned an initial bandwidth capacity whose value is uniformly distributed. Then, we randomly select 1000 nodes as the dynamic service overlay network nodes and connect them into different overlay topologies. Table 6.1 describes the properties of all topologies used in our experiments. The overlay delay, overlay loss rate, and overlay bandwidth values are the measured average inter-node delay, loss rate and bandwidth on the overlay network with the corresponding topology type, node degree, neighbor selection scheme, listed in the same row.

We consider two important QoS metrics, delay and loss rate, to represent the user’s multi-constrained QoS requirements for the composed service. The QoS metric values of each physical network link are uniformly distributed. We simulate the distributed data routing at both IP-layer and overlay-layer using the shortest path routing algorithm with delay metric only. Each peer provides \([1,3]\) service instances. Each service instance can fulfill a service task that is randomly selected from 200 pre-defined service tasks. Thus, each service task can be mapped to on average \(\frac{1000 \times 2}{200} = 10\) different service instances\(^3\). We set the QoS metric values of service instances based on the profiling results of different fully implemented multimedia services (e.g., video scaling, audio mixing, video mixing, caption embedding, transcoding). The initial end-system resource capacities (e.g., CPU, memory) are uniformly distributed.

During each minute, a certain number of composed service requests are generated, which come from randomly selected peers. The abstract service graph of each service request is

\(^2\)We choose Inet-3.0 instead of other Internet topology generator because recent research [75] has shown that the degree-based Internet topology generator can most accurately capture the current Internet structure.

\(^3\)The exact number of duplicated service instance numbers does not change the trend of the experimental results. But the assumption here is that the duplicated service instance number is much smaller than the total peer number.
randomly selected from 50 different templates, which include [2,6] service tasks. Each service session lasts [5,30] minutes. The user’s QoS requirements are uniformly distributed. For comparison, we also implement three other algorithms, optimal, random and static algorithms. The optimal algorithm is realized using exhaustive searching of all candidate service paths. The random algorithm randomly selects among candidate service instances to compose a service path. The static algorithm always chooses a dedicated service instance for a specific service task.

Provision success rate. A composed service is said to be provisioned successful if we can instantiate a service path with enough resources in the current dynamic service overlay network. The provision success rate is calculated by $\theta/\chi$, where $\theta$ denotes the number of successfully provisioned service requests and $\chi$ denotes the total request number.

QoS success rate. A composed service is said to be QoS successful if we can successfully instantiate a service path with enough resources, and the QoS values of the service path satisfy the user’s QoS requirements $Q^{req}$. The QoS success rate metric is calculated by $\eta/\chi$, where $\eta$ denotes the number of service requests with successful QoS provisioning and $\chi$ defines the total request number.

Overhead. We define the overhead of QoS-aware composed service provisioning as the number of control messages generated per minute. To evaluate the scalability of different approaches and their suitability for large-scale dynamic service overlay networks, we compare the overhead of different approaches while they achieve similar performance.

### 6.5.2 Initial Service Composition Comparison Results

We first evaluate the performance of the GC QoS-aware service composition algorithm and distributed QoS-aware service composition algorithms using the provision success rate and QoS success rate metrics. In all experiments presented in this subsection and next subsection, we use the $Tmesh0.1$, which is a 500 node topologically-aware mesh overlay. We will present the comparison results among different overlay topologies in Section 6.5.4. In Figure
Figure 6.1: Average provision success rate of the GC algorithm under different workload on the Tmesh0.1 topology.

Figure 6.2: Average QoS success rate of the GC algorithm under different workload on the Tmesh0.1 topology.

Figure 6.3: Average provision success rate of the distributed algorithm under different workload on the Tmesh0.1 topology.

Figure 6.4: Average QoS success rate of the distributed algorithm under different workload on the Tmesh0.1 topology.
Figure 6.5: Cumulative provision success rate comparison among the optimal, GC, random, and LD algorithms on the Tmesh0.1 topology with 150 requests/minute.

6.1 - Figure 6.4, the X axis shows different request rate representing the number of composed service requests per minute. The range of request rate is selected to reflect different workload conditions of the dynamic service overlay network. In Figure 6.1 and Figure 6.2, the Y axis shows average provision success rate and average QoS success rate, achieved by the GC QoS-aware service composition algorithm, random algorithm, and static algorithm, respectively. Each average success rate value is calculated and averaged over the entire two hours simulation. Figure 6.3 and Figure 6.4 show the similar comparison results among the distributed QoS-aware service composition algorithm, random algorithm, and static algorithm. The probing budget of the distributed algorithm is calculated by $\beta_0 = \min(5\% \cdot \beta_{\text{max}}, 200)$, where $\beta_{\text{max}}$ denotes the number of probes required by the optimal algorithm$^4$. We observe that both GC heuristic algorithm and distributed heuristic algorithm consistently achieve much better performance than the random algorithm and static algorithm. The random algorithm performs a bit better than the static algorithm due to its inherent load balancing property.

We conduct the second set of experiments to compare the performance of GC algorithm, distributed algorithm, and the optimal algorithm. Figure 6.5 and Figure 6.6 illustrate the

$^4$The probing budget calculation may not be the best. But it suffices to demonstrate that SpiderNet can achieve better performance than other common approaches even with a small probing budget.
cumulative provision success rate and cumulative QoS success rate achieved by different algorithms under the same request rate, respectively. Different from the average success rate value that is averaged over the entire simulation duration, the cumulative success rate value is sampled at each minute \( t \), which is averaged over the duration from the beginning to the time \( t \). Because the problem is NP-complete, the optimal solution has exponential computation complexity. Thus, we limit ourselves to a special case of finding service paths with three service tasks. The results show that the GC heuristic algorithm can achieve near-optimal performance. The performance of the distributed QoS-aware service composition heuristic algorithm is tunable. With a larger probing budget, the performance of the distributed heuristic algorithm improves. However, the distributed algorithm can approach the optimal performance with much lower overhead. In this case, the optimal algorithm requires on average \( 10^3 \) probes for each request because each request includes three service tasks and each service task can be mapped to 10 service instances on average. The distributed heuristic algorithm can achieve 95% of the optimal provision success rate and 90% of the optimal QoS success rate with the probing budget 27.

Although both GC heuristic algorithm and distributed heuristic algorithm can achieve near-optimal performance, the overhead of the distributed heuristic algorithm is much less than that of the GC heuristic algorithm, as illustrated by our analytical comparison in Section 6.2. In our experiments, the measured overhead of the distributed algorithm (i.e., control messages per minute) includes at most 1600 probe messages and 50 \( \cdot \) 500 neighbor monitor messages. In contrast, the GC algorithm requires at least 500\(^2\) messages for maintaining the global view with a one minute update period.

### 6.5.3 Failure Recovery Comparison Results

We now compare the performance of reactive and proactive failure recovery algorithms. The reactive failure recovery is combined with the GC QoS-aware service composition and uses the partial service path re-calculation algorithm. The proactive failure recovery is combined with
the distributed service composition algorithm. We simulate a dynamically changing dynamic service overlay network, where a number of peers randomly fail or leave the dynamic service overlay network during each minute. The peer failure/departure can cause composed service failure if its corresponding service path includes the failed peers. We compare the failure frequency with and without the failure recovery on the dynamically changing dynamic service overlay network. Figure 6.7 and Figure 6.8 illustrate the failure frequency on a moderately changing dynamic service overlay network, where 2% of peers fail randomly during each minute. We observe that both reactive and proactive failure recovery algorithms can recover almost all failures by finding a new service path or using a maintained backup service path. The proactive failure recovery algorithm maintains on average 2.74 backup service paths for each service session. We then conduct the same experiments on a highly dynamic dynamic service overlay network, where 20% of peers fail randomly during each minute. Figure 6.9 and Figure 6.10 illustrate the failure frequency results under the highly dynamic dynamic service overlay network for the reactive failure recovery algorithm and proactive failure algorithm, respectively. We observe that the reactive failure recovery algorithm can recover on average 70% failures. The proactive failure recovery algorithm can recover on average 50% failures by maintaining on average 2.74 backup service paths. The results show that both failure
recovery algorithms are quite efficient. However, the proactive failure recovery has much less overhead than the reactive failure recovery because it does not require the global view information and only needs a small number of backup service paths.

### 6.5.4 Overlay Topology Comparison Results

We now evaluate how the QoS provisioning of the composed service depends on the overlay topology. Figure 6.11 and Figure 6.12 illustrates the cumulative QoS success rate
achieved by the GC algorithm and the distributed algorithm on four different overlay topologies: Rmesh0.1, Tmesh0.1, Rpowerlaw, and Tpowerlaw, whose properties are described by Table 6.1. For comparison, the average node degree of the power-law topology is kept the same as the mesh topology. We observe that the overlay topology can significantly affect the QoS provisioning of the composed service, especially for the distributed algorithm. Both algorithms achieve best performance on the topologically-aware mesh topology (i.e., Tmesh0.1). The results indicate that we should explicitly manage the overlay topology in order to achieve best QoS provisioning for the composed service.

Finally, we investigate how QoS provisioning of the composed service depends on the node degree of the mesh topology. Figure 6.13 and Figure 6.14 illustrate the cumulative QoS success rate achieved by the GC QoS-aware service composition algorithm and distributed QoS-aware service composition algorithm on different random mesh overlay with increasing node degrees. Figure 6.15 and Figure 6.16 illustrate the results of similar experiments on different topologically-aware mesh overlay with increasing node degrees. Surprisingly, we observe that a more connected overlay mesh with a larger node degree does not improve the QoS provisioning a lot than a less connected overlay mesh. On the other hand, a mesh topology with a large node degree demands high states maintenance overhead because each
Figure 6.15: Cumulative QoS success rate comparison of the GC algorithm on the Tmesh0.1, Tmesh0.3, and Tmesh0.5 with 150 requests/minute.

Figure 6.16: Cumulative QoS success rate comparison of the LD algorithm on the Tmesh0.1, Tmesh0.3, and Tmesh0.5 with 150 requests/minute.

peer needs to maintain update-to-date states of more neighbors. Our results suggest that a topologically-aware overlay mesh with a small node degree is most suitable for QoS-aware service composition.

6.6 Summary

In this chapter, we present a comprehensive comparative study for providing QoS-aware service path composition in dynamic service overlay networks. The comparative study explores principle design alternatives and provides insights about the trade-offs between performance and overhead of different approaches. In summary, our comparative study reveals four important findings. First, despite the fact that the problem of QoS-aware service path composition is NP-hard, both GC heuristic algorithm and LD heuristic algorithm can achieve near-optimal performance with polynomial computational complexity. Second, the overhead comparison between the GC algorithm and LD algorithm depends on a range of factors. However, both theoretical analysis and simulation experiments indicate that the LD algorithm has much lower overhead than the GC algorithm in large-scale dynamic service overlay networks. Third, runtime failure recovery is important for providing failure resilient services
in dynamic service overlay networks. Both reactive and proactive failure recovery can achieve
good failure recovery performance. However, proactive failure recovery can provide faster
failure recovery than the reactive approach in large-scale dynamic service overlay networks.
Fourth, our simulation experiments illustrate that the overlay topology type can significantly
impact the performance of QoS-aware service composition. In contrast, the service compo-
sition performance is less affected by the node degree, especially for the topologically-aware
mesh topology.
Chapter 7

Access SpiderNet: Ubiquitous Two-Tier Service Composition

In this chapter, we present the access SpiderNet to provide flexible dynamic service composition in local ubiquitous computing environments such as smart rooms. The major goal of the access SpiderNet is to adaptively fill gap between the services received from the core SpiderNet (i.e., utility SpiderNet or P2P SpiderNet) and the ubiquitous service delivery requirements in the service access domain. Different from the core SpiderNet, the access SpiderNet performs two-tier service composition, which assumes that service components can be dynamically uploaded or migrated among distributed hosts.

7.1 Introduction

Ubiquitous computing [78] has extended the computer system to the whole physical space and leaded to a more dynamic distributed system than ever before, with many devices and services coming and going frequently. Moreover, nowadays a single user often possesses multiple heterogeneous devices ranging from desktop, laptop, to PDA. The user may use any of those devices as the portal, with the help of other personal devices and/or proxy hosts, to perform tasks. Many quality sensitive distributed multimedia applications, such as video-on-demand and visual tracking, are being deployed in such a ubiquitous computing environment. Thus, a big challenging problem is to provide a dynamic service composition
model to enable seamless delivery of distributed multimedia applications, in the ubiquitous computing environment, with best possible Quality-of-Service (QoS) guarantees.

The problem of service composition has been addressed in different research work [36, 50, 83]. However, most of the proposed approaches do not meet the expectations of the user community and fall short of the potential for ubiquitous computing. We identify two key problems as follows:

- The first problem is brought by the inflexible way service components are composed to form a distributed application delivery. Dynamic insertion of mediating services [50] provides certain adaptability, but also unnecessary overhead when the server or client service itself can be dynamically changed. Specifying a polymorphic distributed application [83] using multiple service paths provides a more flexible solution. However, those service paths are often predefined and fixed. They lack the adaptability to accommodate changes that are unknown at design time. Moreover, all of the above approaches can only handle linear service compositions which have serious limitations to support complex distributed multimedia applications.

- Second, although ubiquitous computing environments provide more abundant resources than ever before, most of them are under-utilized. Putting all mediating services in a single host makes them easy to maintain, but also vulnerable to malicious attacks. It is highly desirable that those services can dynamically bind available resources in the runtime environment. On the other hand, a single user often possesses multiple heterogeneous devices and many proxy hosts are also available to users everywhere (e.g. office, conference room, hotel). Aggregating those resources efficiently can definitely help to overcome resource limitations of mobile devices and provide better QoS for users. However, dividing a distributed application delivery appropriately and binding services to suitable available devices is a challenging problem and has not been systematically addressed yet [13].
The access SpiderNet addresses above challenges by proposing an integrated two-tier service composition model. The service composition model includes two tiers: (1) service adaptation tier and (2) service distribution tier. The former is responsible for choosing a set of suitable services, discovered in the current environment, to compose a customized application delivery adapting to an arbitrary client device. The latter is responsible for dividing a distributed application into several partitions and dispatching them to different devices according to the current distributed resource availability. We have implemented a prototype of our service composition model as part of the Gaia OS [3], an enabling infrastructure for the smart spaces. Due to the scalability requirement, we structure the smart spaces hierarchically by grouping devices into different domains. Each domain contains one domain server, which provides the key infrastructure services for the entire domain space, in the same way as today’s operating systems do for a single desktop. The service composition model is implemented as part of the domain server. It cooperates with other domain services, such as the event service, to dynamically configure distributed applications for the user.

The rest of this chapter is organized as follows. We first introduce the system architecture of the access SpiderNet. Then we present the context-aware service adaptation provided by the access SpiderNet. Third, we describe the service distribution schemes in the access SpiderNet. Finally, we present the experimental results followed by the summary about the access SpiderNet.

### 7.2 Access SpiderNet System Architecture

The access SpiderNet is modelled as part of a centralized domain server that is deployed in each service access domain at the edge of the Internet, illustrated by Figure 7.1. A ubiquitous computing domain can be either a room (e.g., smart room) or a building (e.g., Sieble building), depending on the maximum number of devices allowed in one domain. As illustrated in Figure 7.1, each ubiquitous computing domain includes many heterogenous
input devices (e.g., cameras, camcorders, remote controllers), output devices (e.g., plasma displays, printers, audio systems) and processing machines (PC clusters).

Each domain has a domain server which provides the key infrastructure services for various stationary, embedded and mobile devices, in the same way as today’s operating system does for a single desktop. The access SpiderNet is regarded as part of the domain server that provides infrastructure support for the ubiquitous delivery of composed service on the client-side such as display, printout or storage of the data received from the ServFlow within the wide-area network. Unlike the utility SpiderNet or the P2P SpiderNet where services are typically long-running and immutable, the access SpiderNet supports a much more flexible service model where services can be dynamically uploaded and instantiated; or stopped and deleted.

The two key domain services provided by the access SpiderNet are: (1) context-aware service adaptation, which can automatically compose a set of adaptors to adapt the output from the ServFlow to the current computing context; (2) service distribution, which splits the client-side services with the adaptors into several partitions in order to overcome the resource constraints of mobile devices. The access SpiderNet cooperates with other domain services [3] such as space repository, space monitor, and event manager to get such information as
resource availability on each device in the domain, addition/removal of devices, available services in the domain.

7.3 Service Adaptation

In this section, we describe the context-aware service adaptation performed by the access SpiderNet. The context here refers to the current available client devices (e.g., cell-phone, PDA, HDTV), available client services (e.g., display service, playback service), and available resources in the current ubiquitous computing environment. We first describe the goals of the service adaptation. Then, we present the major operations in the context-aware service adaptation. Finally, we introduce the ordered coordination algorithm to automatically generate the service adaptation plan.

7.3.1 Service Adaptation Goal

One of the key features of ubiquitous computing is to allow the user to access services anywhere. For example, if the user wants to display a video, he or she should be able to perform the task using whatever display devices available in the current environment. Hence, services on the client-side (client services in short) should be as much flexible as possible. To achieve the goal, we argue that the client services should be abstractly named using high level descriptions. For example, we should use “print” to name the printing service, and “display” to name the video display service. The service details, such as the screen size and video format for the display service, should be resolved by the access SpiderNet system during the service instantiation time.

Because of the above abstract naming strategy, the access SpiderNet needs adaptations to coordinate the interactions among dynamically discovered services. The adaptations are realized by various transformation service components, called adaptors. The adaptors are also useful to fill the gap between the client service (e.g. display, printout) requirements
of the wide-area ServFlow and the current available client services. Furthermore, the ubiquitous computing environment such as a smart room is featured by its abundant output devices such as audio systems, plasma displays, and screens. Thus, the user can use these output devices simultaneously to enjoy multiple synchronized output streams (e.g., audio, video and documents) received from different wide-area ServFlows. As a result, the client services can also include: (1) aggregators that synchronizes multiple streams; and (2) splitters that separates a composite stream into multiple atomic streams and dispatches them to different output devices. The goal of service adaptations in the access SpiderNet is to automatically weave data sources from the wide-area ServFlow, client service components, adaptors, aggregators, and splitters in a consistent way to realize ubiquitous service delivery.

### 7.3.2 Service Adaptation

We now present the major operations of the service adaption performed by the access SpiderNet, which are described as follows,

- **Acquire the abstract service graph for the client service.** As we mentioned above, the client service should be abstractly named using high-level service descriptions. We call this high-level service description the *abstract service graph*. It includes abstract specifications about each service component including aggregators and splitters, the client service needs, and also the interactions/dependencies among these components. We assume that client service developer provide the abstract service graph that is stored in the domain server as part of the mete-data information of the client service.

- **Discover service instances in the current environment.** Once the abstract service graph is acquired, the domain server find suitable service instances in the current environment, according to the high-level descriptions in the abstract service graph. If multiple service instances are discovered to match the same higher level descriptions, the user is notified and asked to choose one. For example, the domain server can find mul-
tiple printing services (color printing, black-white printing) or multiple video display services (HDTV-format on Plasma display, MPEG-4 format on laptop). The user can choose one of them according to his or her preference.

- **Check inter-service consistencies and coordinate ad-hoc interactions.** The service instances, returned by the second step, are concrete service components discovered in the current environment. They include more detailed and specific information than their abstract descriptions. The access SpiderNet needs to check the inter-component QoS consistencies (Chapter 2) between discovered service instances. If any inconsistent interactions are discovered, the available adaptors are composed into the final instantiated service graph to coordinate the interactions.

- **Generate the final instantiated service graph.** After the third step, a consistent instantiated service graph is generated. The access SpiderNet then decides how to properly partition the service graph to best utilize the resource pools in the current environment.

Among the above four steps, the third one forms the key part of the service adaptation. It tackles two major problems: (1) fast and efficient QoS consistency check among discovered service instances; and (3) automatic correction of inconsistent interactions. We will explain in detail how these problems are addressed in the next section.

### 7.3.3 Ordered Coordination

We propose the *Ordered Coordination (OC)* algorithm to perform inter-service consistency check and automatic correction in the instantiated service graph. It includes the following major steps, illustrated in Figure 7.2: (1) topologically sort the instantiated service graph; (2) check the inter-service consistency, in the reverse order of topologically sorting, between each node and its predecessors, according to the inter-component QoS consistency relation (Chapter 2); (3) If any inconsistency is found, possible automatic corrections are performed. The access SpiderNet can insert a set of adaptors, such as transcoding and scaling services, to
resolve input/output inconsistencies between two interacting service components. The computational complexity of \( OC \) algorithm is \( O(V+E) \), where \( V \) and \( E \) are the numbers of service components and edges in the final service graph, respectively.

### 7.4 Service Distribution

This section presents the service distribution scheme in the access SpiderNet, illustrated by Figure 7.3. After the access SpiderNet generates a consistent service graph, it properly splits the service graph into several partitions to overcome resource constraints of mobile devices and best utilize the resource pools in the ubiquitous computing environment. We first describe the service distribution goal of the access SpiderNet. Then we formally define...
the service distribution problem and prove that it is NP-hard. We then provide a polynomial heuristic algorithm to approximate the optimum algorithm.

7.4.1 Service Distribution Goal

The goal of service distribution in the access SpiderNet includes two aspects: (1) *overcome resource constraints of mobile devices*. Many mobile devices such as pocket PCs and cell-phones have very limited resources and computing power. If we instantiate the whole service graph on the mobile device, it can exceed the resource capacity of the mobile device. In order to overcome the resource limitations for the mobile device, the access SpiderNet splits the service graph into several partitions. Only one of the partition is instantiated on the mobile device in order to reduce the resource requirements on the mobile device; (2) *improve overall utilization of resource pools*. As illustrated by Figure 7.1, ubiquitous computing environments provide us with more abundant resources than ever before. It is highly desirable that service components can dynamically bind available resources in the ubiquitous computing environment to achieve best overall resource utilization.

7.4.2 Problem Definition

Suppose each service component in the service graph is associated with an end-system *resource requirement* vector \( R = [r_1, r_2, ..., r_m] \), with \( r_i \ (1 \leq i \leq m) \) as the required amount of the i'th resource type (e.g., CPU, memory). Each edge \( e = (u, v) \) in the graph is assigned an integer weight “\( c(u,v) \)”, representing the required bandwidth from node \( u \) to \( v \). We use vector \( RA \) to represent *resource availability* on each device. We assume that \( R \) and \( RA \) represent the same set of resources and obey the same order. We use \( b_{i,j} \) to represent the end-to-end available network bandwidth from the \( ith \) to the \( jth \) device. To formally define the service distribution problem, we introduce the following definitions.
DEFINITION 3.1 The addition of two resource vectors for service component A and B, \( R^A = (r^A_1, r^A_2, ..., r^A_m) \) and \( R^B = (r^B_1, r^B_2, ..., r^B_m) \), is defined as the following:

\[
R^A + R^B = [r^A_1 + r^B_1, r^A_2 + r^B_2, ..., r^A_m + r^B_m]
\]  

(7.1)

DEFINITION 3.2 Given resource requirement vector \( R = [r_1, r_2, ...r_m] \) and resource availability vector \( RA = [ra_1, ra_2, ...ra_m] \), \( R \leq RA \) if and only if

\[
\forall i, 1 \leq i \leq m, r_i \leq ra_i,
\]  

(7.2)

DEFINITION 3.3 Let \( G = (V,E) \) be a directed graph. A \( k \)-cut in \( G \) is a partitioning of \( V \) into nonempty subsets \( V_1, ..., V_k \). An edge \( e \) belongs to the \( k \)-cut if its endpoints belong to different subsets of the partition \( V_1, ..., V_k \).

For instance, Figure 7.3 shows a three-way-cut of the service graph. The edges belonging to the cut are \{ \( e_{1,2}, e_{1,8}, e_{5,2}, e_{5,8}, e_{9,8}, e_{2,7}, e_{8,7}, e_{8,6} \) \}.

DEFINITION 3.4 Given the service graph \( G = (V,E) \) and \( K (2 \leq K \leq V) \) available devices, we define that \( G \) can "fit into" those \( k \) devices, if and only if there exits a \( k \)-cut \( (V_1,...,V_k) \) of the graph, such that

- \( \forall j, 1 \leq j \leq k, \sum_{v_i \in V_j} R^i \leq RA^j \), where \( R^i \) represents the resource requirement vector of component \( v_i \), \( RA^j \) represents the resource availability on the \( j \)th device;

- \( \forall i, j, 1 \leq i, j \leq (k - 1), \forall e = (u, v) \in E, u \in V_i, v \in V_j, \sum_{e \in E, u \in V_i, v \in V_j} c(u, v) \leq b_{i,j} \), where \( c(u, v) \) represents the communication throughput on the edge \( e \) that belongs to the \( k \)-cut, \( b_{i,j} \) represents the available bandwidth between the \( i \)th and \( j \)th devices.

Usually, there exist multiple \( k \)-cut schemes that can fit the service graph into \( k \) devices. Thus, the service distributor needs to find the one with minimum cost aggregation. The cost aggregation is defined as follows:

DEFINITION 3.5 Given a \( k \)-cut \( \Phi = (V_1, ..., V_k) \) for the service graph \( G = (V,E) \), its
Cost Aggregation (CA) can be calculated in the following way:

\[
CA(\Phi) = \sum_{j=1}^{k} \sum_{i=1}^{m} w_i \cdot \frac{r_i}{ra_j} + \sum_{1 \leq i,j \leq k} w_{m+1} \cdot \frac{T_{i,j}}{b_{i,j}}
\]

where \(T_{i,j} = \sum_{u \in V_i, v \in V_j} c(u, v)\) and \(w_i (1 \leq i \leq (m+1))\) are nonnegative values so that \(\sum_{i=1}^{m+1} w_i = 1\).

For any end-system resource type \(r_i\) (e.g., memory, cpu), \(w_i \cdot \frac{r_i}{ra_j}\) is a normalized value between 0 and 1, where \(w_i\) represents the significance of this resource type. Generally, we assign higher weights for more critical resources. For the network resource type, \(w_{m+1} \cdot \frac{\sum_{u \in V_i, v \in V_j} c(u,v)}{b_{i,j}}\) is a normalized value between 0 and 1, where \(w_{m+1}\) represents the significance of network resource. In both cases, the normalized value represents the cost the user pays for using a specific type of resource to perform his or her tasks. Intuitively, the more important (higher weight) and more scarce (smaller resource availability) the resource is, the larger cost (larger normalized value) it takes the user to use it. Minimizing the cost aggregation can help improve the total resource utilization and reduce the contention on critical resources. As a result, the user’s QoS requirements can be better preserved and more applications can be supported simultaneously given the union of all resources. Thus, the problem of optimal service distribution is to find a \(k\)-cut for the given service graph, which can make the graph fit into the current \(k\) available devices and also minimizes the cost aggregation for the user.

We now show that the general problem of finding the optimal service distribution (\(k\)-cut) that makes the service graph fit into \(k\) devices and also minimizes the cost aggregation is NP-hard.

**Theorem 1** Finding the optimal service distribution (OSD) that makes the service graph fit into \(k\) devices and also minimizes the cost aggregation is NP-hard.

**Proof:** We prove this by showing that the minimum directed multi-way cut problem which is known to be NP-hard [35] maps directly to a special case of our service distribution
problem. The minimum directed multi-way cut problem is as follows: Let $G = (V, E)$ be a
directed graph and let $c(u,v)$ be a non-negative capacity function associated with the edge
e $e = (u,v)$.

$$Minimize \sum_{1 \leq i,j \leq k, u \in V_i, v \in V_j} c(u,v)$$ (7.4)

where $V_1, V_2, \ldots, V_k$ are $k$ non-empty subsets of $V$ and form a $k$-cut of graph $G$. The above
problem is identical to the following special case of our problem. Suppose each of the $k$
available devices has infinite end-system resource availability. Thus, any $k$-cut of service
graph $G$ can satisfy the "fit into" constraints. We also assume that every service component
can be assigned to any of the $k$ available devices. In addition, we let (1) $w_i$ ($1 \leq i \leq m$) be 0
and $w_{m+1}$ be 1; (2) every available bandwidth $b_{i,j}$ be 1 (Gbps). An identity transformation
makes the minimum directed multi-way cut problem a special case of our OSD problem,
shown by equation (4). Thus, the OSD problem is also NP-hard. □

7.4.3 A Service Distribution Heuristic Algorithm

In this section, we present a polynomial approximate optimum algorithm for the OSD prob-
lem. We first discuss several important practical issues neglected in the previous problem
definition. Then we present the heuristic algorithms that takes the practical issues into
account.

First, we assume that every service component can be instantiated on any device. But
the assumption does not hold in reality and some services must run on certain device. For
example, the display service in the video-on-demand application must run on the client
device. Second, the model described so far is not heterogeneity-aware. In other words, the $k$
available devices are assumed to be the same. To solve the first problem, we can first "pin"
those special components on proper devices by inserting them into the corresponding subsets
($V_i$, Definition 3.3) of partitions. For the second problem, we need to normalize both the
**OptimalServiceDistribution**

**Input:** Service Graph $G = (V,E)$, $K$ devices $D = \{ RA_1, RA_2, \ldots, RA_k \}$; 

**Output:** $k$-cut $(V_1, V_2, \ldots, V_k)$ of $G$; 

/*insert those service components that cannot be instantiated anywhere into their proper devices */

Insert $(V,D)$;

/*sort the $K$ devices in decreasing order of RA*/

DeviceList = Sort$(D)$;

while $|V| > K$ do 

$D_i = \text{top}(\text{DeviceList})$;

if $V_i$ has node $v_i$ then $v_j = \text{LargestNeighbor}(v_i)$; else $v_j = \text{LargestNode}(V)$;

if $v_j$ can be satisfied by RA$i$ then if $v_j$ is largestNeighbor then merge $v_j$ with $v_i$, delete $v_j$ from $V$; insert $v_j$ into $V_i$, update RA$i$; else if ($v_j$ can be satisfied by the other RA$m$) then if ($V_m$ has node $v_m$) then merge $v_j$ with $v_m$, delete $v_j$ from $V$; insert $v_j$ into $V_m$, update $RA_m$; else return failure; /*ask for lower QoS level*/

DeviceList = Sort$(D)$;

return $(V_1, V_2, \ldots, V_k)$;

**Help Functions**

/*sort the $K$ devices in decreasing order of RA*/

Sort$(D)$ for $l = 1$ to $K-1$ do for $n = l+1$ to $K$ do $\sum_{i=1}^{n} v_i - \frac{r^{(x)}}{v_i} - \frac{r^{(y)}}{v_i} + \frac{r_{in1}}{v_i} + \frac{r_{in2}}{v_i} < 0$ then swap $(D_l, D_n)$;

Largest Neighbor $(v_i)$ /*$v_i$ keeps the current largest neighbor of $v_i$*/ for all uninserted neighbors $v_i$ do if $\sum_{j=1}^{n} \frac{r_{inj}}{v_i} + \frac{r_{inj}}{v_i} < 0$ then $v_i = v_j$;

LargestNode$(V)$ /*$v_i$ keeps the current largest uninserted node in $V$*/ for all uninserted nodes $v_i$ in $V$ do if $\sum_{i=1}^{n} \frac{c(l,m)}{v_{i,n}^{(x)}} + \frac{c(j,n)}{v_{j,n}^{(x)}} < 0$ then $v_i = v_j$;

Figure 7.4: The pseudo code of the optimal service distribution heuristic algorithm.

**resource requirement** and **resource availability** values on heterogeneous machines to those on a benchmark machine. For example, if we use a laptop as the benchmark machine and assume the resource availabilities of a PDA and a PC are $RA^{PDA} = \{32\text{MB (memory)}, 100\% (CPU)\}$ and $RA^{PC} = \{256\text{MB},100\%\}$, then the two normalized resource availability vector values on the benchmark machine (laptop) may become $N(RA^{PDA}) = \{32\text{MB}, 40\%\}$, $N(RA^{PC}) = \{256\text{MB}, 500\%\}$. We assume that the memory availability values are not affected by device heterogeneity. However, the normalized CPU availability should be changed according to the speed difference between the heterogeneous device and the benchmark machine. Similarly, the **resource requirement** values also need to be normalized to those on a benchmark machines.

In the general case, the above normalization functions can be derived through experimental measurements.

We now provide a polynomial heuristic algorithm for the OSD problem. Figure 7.4 shows
the pseudo code for the algorithm. It primarily involves the following steps: (1) insert those service components, that cannot be instantiated arbitrarily, into their proper devices; (2) repeat sorting the k available devices in decreasing order of their resource availabilities and insert the next chosen service components to the current head of the sorted device list, namely the device that currently has the largest resource availability. If the head device contains a service component A, then the next chosen component is A’s neighbor, which has the largest resource requirements \(^1\). We then insert the chosen component into the head device and merge it with A. If the head device is empty, then the next chosen service component is the one which has the largest resource requirements among all remaining service components. Repeat the above procedure until every service component has been inserted into a proper device.

7.5 Performance Evaluation

We have implemented a prototype of the service composition model as part of the Gaia OS [3], an enabling infrastructure for smart spaces, and performed several experiments based on both the prototype and simulations.

7.5.1 Prototype Experimental Results

Our first set of experiments is performed based on the prototype, using two distributed multimedia applications, mobile audio-on-demand and video conferencing implemented in our lab. The servers/gateways in our testbed are either Sun Ultra-60 workstations or Pentium III 900 PCs. The client devices are either PCs, laptops (IBM Thinkpad), or PDAs (HP Jornadas). we demonstrate that the service composition model is able to provide soft QoS guarantees to the user in the ubiquitous computing environment. Figure 7.5 shows

\(^1\)Both resource availability and resource requirement are measured using the weighted sum of different resources. Due to the page limit, the detailed equations are not shown.
Figure 7.5: End-to-end QoS of different service compositions.

Experimental results for both applications. The results from the mobile audio-on-demand application show that our dynamic service composition can flexibly accommodate common runtime changes, such as user mobility and handoff between heterogenous devices. The experiment with the video conferencing application demonstrates the ability of our framework to handle on-demand service composition for non-linear service graph. Figure 7.6 shows the dynamic service composition overhead, during the above experiments. For the mobile audio-on-demand application, we assume that the required service components are already installed on the target devices in advance. Thus there is no dynamic downloading overhead involved. However, in the video conferencing application, we assume that all required service components need to be downloaded on demand from the component repository. The state handoff time includes the handoff protocol overhead and also the buffering time for the first frame at the interruption point. Since the PDA is connected with the wireless network while the PC is connected with the ethernet, the state handoff time from PC to PDA is longer than that from PDA to PC. Overall, the results show that the overhead of the dynamic service composition is relatively small compared to the entire execution time of the application.
Moreover, the dynamic downloading overhead, which occupies the largest proportion of the total overhead, can often be avoided if the required components are already on the target devices.

7.5.2 Simulation Results

To demonstrate the efficiency of the proposed heuristic service distribution algorithm, we conducted two sets of experiments based on simulations. First, we compare the relative performances of different heuristic algorithms (random and ours) with the optimal algorithm. The optimal algorithm uses exhaustive search for the optimal service distribution solution. Since the problem is NP-hard, we limit ourselves to the special case of two-way cut. We assume two heterogeneous devices (PC, PDA) are used, with initial normalized resource availability vectors $RA^1 = [256MB, 300\%]$, $RA^2 = [32MB, 100\%]$, respectively. We consider service graphs with 10 to 20 service components. Each component has, on average, 3 to 6 outbound edges. Other parameters including resource requirement vectors, communication throughput on each edge and weight values are uniformly distributed. Table 7.1 summarizes
<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Average</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>25%</td>
<td>0%</td>
</tr>
<tr>
<td>Our Heuristic</td>
<td>91%</td>
<td>60%</td>
</tr>
<tr>
<td>Optimal</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 7.1: Comparisons among different service distribution algorithms.

Figure 7.7: Success rate comparisons among the fixed, random and our heuristic algorithms.

The comparison results for 150 randomly generated service graphs. The first column in Table 7.1 is the average performance of each heuristic, measured by the ratio of cost aggregation between the optimal solution and the solution found by the heuristic, averaged over all 150 graphs. The second column is the percentage of 150 graphs for which our heuristic or the random algorithm was able to find the exact optimal solution.

Second, we compare the overall success rate achieved by our heuristic algorithm with random and fixed algorithms. A service composition request is said to be successful if the service graph can fit into the current available devices. The success rate is calculated by the ratio of the number of successful service composition requests to the number of total configuration attempts. We assume three heterogeneous devices (desktop, laptop, and PDA) are used, with initial normalized resource availability vectors $RA_1 = [256MB, 300\%], RA_2 = [128MB, 100\%], and RA_3 = [32MB, 50\%]$ respectively. The available bandwidths $b_{1,2}, b_{1,3},$ and $b_{2,3}$ are initialized to be 50Mbps, 5Mbps, and 5Mbps respectively. We randomly
create 5000 application requests over 1000 hours period. Each request randomly selects a service graph from 5 predefined ones. Each graph has 50 to 100 nodes with on average 5 to 10 outbound edges. The length of each application is exponentially distributed from 5 minutes to 1 hours. Other parameters, including resource requirement vectors, communication throughput on each edge and weight values, are uniformly distributed. When a new application starts or an old application stops, both our heuristic and random algorithms make the re-distribution decisions, but the fixed algorithm does not. The success rate is calculated every 50 hours. Figure 7.7 shows the comparison results among the fixed, random and our heuristic algorithms. Fixed algorithm has the lowest success rate because it lacks dynamic service distribution considerations. Random algorithm provides better success rate since it benefits from the flexibility of dynamic service distribution. Finally, our heuristic algorithm consistently maintains the highest success rate because it considers both resource availability and resource requirements during dynamic service distributions.

7.6 Summary

In this chapter, we have presented the design and evaluation of the access SpiderNet. The major contributions of the access SpiderNet include: (1) identify two key problems, service adaptation and service distribution for the ubiquitous service delivery; (2) provide the design and polynomial service adaptation algorithm, which provide automatic inter-service consistency check and adaptor insertion; and (3) define the optimal service distribution problem which is shown to be NP-hard and then providing a polynomial approximation algorithm for the problem. We have conducted experiments using both prototype and simulations. The experimental result show both the feasibility and efficiency of the access SpiderNet.
Chapter 8

Related Work

In this chapter, we review related work. SpiderNet is related to a range of different research efforts. We classify the related work into several categories, each of which is briefly introduced and compared with SpiderNet.

8.1 Service Composition

Recently, service composition has received a lot of research attention. For example, the SA-HARA project [61, 62, 63] addressed the fault-resilience and load balancing problem for the wide-area service composition. The SWORD project [60] proposed a developer toolkit for the web service composition. It can automatically generate a functional composition plan given the functional requirements for the composed application. SWORD only addressed the off-line and static service composition problem. The functional composition plan generated by SWORD can be part of the constraint specification, which SpiderNet takes for finding and instantiating a qualified service path/graph. Nakao et al. [56] addressed the problem of composing end-to-end media object paths using a centralized approach with two stages: global path construction and local path construction. The SPY-Net framework [82] addressed the problem of resource contention for finding multimedia service paths in small-scale media proxy networks. The CANS project [33] provided an on-line function composition framework, which can dynamically compose a list of service functions during runtime to adapt to
heterogeneous end-system and network conditions. SpiderNet differs from the above work by focusing on the QoS and resource management issues in service composition. Moreover, SpiderNet provides customized solutions for different distributed computing environments.

8.2 Service Discovery

To provide dynamic service composition, SpiderNet first needs to discover service components in the distributed computing environments. Previous research work (e.g., SDS [26], INS [8, 12], Active Names [76]) has addressed the problem of expressive service discovery. The Active Names project also experimentally demonstrates the benefit of composing multiple service extensions. SpiderNet extends the above work by providing QoS-aware and load sensitive mapping of function graph (instead of a single service name) into an instantiated composed service in heterogenous distributed computing environments. In doing so, SpiderNet considers the states of not only individual (replicated) service components, but also the application-level connections between service components.

8.3 In-Network Processing

Active networks [79] have advocated the idea of pushing computation into network by allowing application to inject programs into network. Active Service framework [9] addresses a simplified problem that discovers and configures existing services to satisfy the application’s requirements. In [71], the authors proposed a distributed virtual machine to move system services out of clients and distribute them onto network servers. SpiderNet is inspired by the above work and provides an alternative application-level overlay based approach to providing composed services. Thus, SpiderNet has the advantages of flexibility and deployability.
8.4 Overlay Networks and Grid

Recently, various overlay networks have been proposed, such as peer-to-peer data lookup overlays [67, 74, 65, 11], locality-aware overlays [47], application-level multicast overlays [24, 20], resilient routing overlay [10], and Internet Indirection Infrastructure overlay [73]. The major difference between SpiderNet and the above work is that SpiderNet focuses on composed service delivery rather than merely data transport or lookup. In doing so, SpiderNet tackles the challenge of scalable decentralized resource and QoS management while instantiating the composed service. The Opus project [17] proposed an overlay utility service that performs wide-area resource mapping based on the application’s performance and availability requirements. Different from Opus that uses hierarchy, aggregation, and approximation for tracking global states, SpiderNet proposes on-demand states collection that is more suitable for flat overlay structure and QoS-aware composed service delivery. Wide-area resource management has also been studied in the context of Grid computing (e.g., [32, 52]). SpiderNet complements previous work by integrating decentralized resource management into dynamic service composition.

8.5 Peer-to-Peer Computing

Recently, with the popularity peer-to-peer (P2P) file sharing systems, such as Gnutella [4], peer-to-peer computing has drew much research attention. Much research work has been devoted to provide scalable P2P lookup services, such as Chord [74] and CAN[65], pastry [67], and Tapestry [11]. The P2P SpiderNet uses these P2P data lookup solution for keyword-based service lookup. However, the service components provided by P2P SpiderNet node are not just music files, but also various intermediate processing services such as media transcoding and scaling, video embedding, language translation, context-aware encryption and decryption. Moreover, these service components can be composed on-the-fly according to the user requirements.
8.6 QoS Routing

Service composition is also related to the problems of quality-of-service (QoS) routing [23] and IP-anycast [49], which all select network paths based on certain constraints or metrics. The QoS routing algorithms solve the problem of finding a network path that satisfies single or multiple end-to-end QoS constraints such as delay and bandwidth. However, service path/graph routing addresses not only end-to-end QoS constraints but also many other new constraints such as the service constraint. Furthermore, the resource requirements of a service path/graph are no longer uniform along the entire data transmission path. For example, the bandwidth requirements along a service path/graph can be different since intermediate service components can change the forwarded data. Thus, the QoS routing algorithms are not sufficient for service composition. IP-Anycast [49] also addresses the related problem, where a server has multiple replicated instances that share the same anycast address. A single server instance is properly chosen for the client according to the routing system’s definition of distance. In SpiderNet, however, each service component can have multiple replicated instances, deployed on different overlay nodes. service composition needs to properly select and also compose multiple service components based on various inter-service dependency and commutation relations.

8.7 Workflow System

Workflow systems [59] have been developed in the industrial world to automate their e-business processes. Recently, service composition has been actively addressed in those e-business systems in order to achieve flexibility and cost-effectiveness. For example, the eFlow project at HP labs [30] provided an adaptive and dynamic service composition mechanism for the commercial e-business process management. One major difference between SpiderNet and those Workflow system is that SpiderNet targets to streaming applications while Workflow system targets to e-business processes. We also neglect several business-oriented
procedures such as negotiation. Instead, SpiderNet focuses on addressing scalability, fault-resilience and resource utilization while composing service paths/graphs.

8.8 Ubiquitous Computing

The problem of service adaptation for ubiquitous computing has been addressed by different research work. The Odyssey project [57, 58] introduced an application-aware adaptation service (adaptors) within the end host to accommodate resource changes, such as wireless network bandwidth fluctuations. However, Odyssey did not support composition of a set of adaptors. In the Path project [50], the authors propose the concept of “Path” to compose commercial-off-the-shelf (COTS) services on demand in ubiquitous computing environment. Although the concept of “Path” can solve the type mismatches for ubiquitous multimedia streaming applications, it lacks QoS and resource management. In the 2K$q$ project [83], the authors use a set of different compositions to represent the same application with different QoS levels so that the general QoS support is explicitly included into the service composition solutions. However, the service composition list for a particular application is predefined and thus cannot accommodate unexpected runtime changes which are a common case in the ubiquitous computing environment.
Chapter 9
Concluding Remarks

This dissertation addresses the problem of QoS-aware service composition in heterogeneous distributed computing environments. Service composition enables future distributed application services such as advanced multimedia streaming, to be automatically composed from atomic service components. Compared to the conventional monolithic service provisioning approach, service composition achieves better customizability, manageability, and cost-efficiency. We have presented SpiderNet, an integrated middleware framework to support QoS-aware service composition in emerging heterogenous distributed computing environments. SpiderNet can provide high quality, failure resilient, and ubiquitous application services to the end-users or applications. The organization of this chapter is as follows. First, we summarize our major contributions. Then, we briefly discuss possible future research directions.

9.1 Contributions

In an attempt to design and implement an integrated QoS-aware service composition framework to support high quality, failure resilient and ubiquitous application services in next-generation Internet, this dissertation makes the following specific contributions:

- Service overlay network model. SpiderNet proposes a novel service overlay network model to connect previously isolated distributed service components via application-
level overlay links. Thus, SpiderNet can provide QoS-aware and failure resilient services by dynamically selecting and composing proper service components based on the states information of the service overlay network and the user’s service requirements.

- **Hybrid layered architecture.** To accommodate heterogeneous computing environments, SpiderNet adopts an integrated hybrid system architecture, which consists of two types of sub-systems: (1) *polymorphic core SpiderNet* (i.e., utility SpiderNet and P2P SpiderNet), and (2) *ubiquitous access SpiderNet*. To achieve an easy-to-use framework, SpiderNet provides a layered vertical architecture consisting of (1) *abstract composite service layer*, (2) *instantiated distributed service layer*, and (3) *service overlay network layer*.

- **Utility SpiderNet.** The utility SpiderNet provides a centralized contract-based service composition solution for the core SpiderNet, which targets the utility computing environment. Application services are offered as utilities, which can be bought on-demand as the user would for electricity. The utility SpiderNet provides both (1) centralized contract-based QoS-aware service composition at service setup phase, and (2) reactive runtime failure recovery to maintain the QoS of composed services during service sessions. As a case study, we present a voice-over-IP (VoIP) conferencing system built on top of the utility SpiderNet to demonstrate the benefit of service composition.

- **P2P SpiderNet.** The P2P SpiderNet provides a decentralized QoS-aware service composition solution for the core SpiderNet, which targets large-scale dynamic P2P systems. The P2P SpiderNet proposes a novel bounded composition probing (BCP) approach to scalable QoS-aware service composition, which can reduce composition overhead by several orders of magnitude and still remain the efficiency of QoS-aware service composition. To overcome dynamic peer arrivals/departures, the system provides proactive failure recovery mechanisms to maintain the QoS of composed services during service sessions.
• **Access SpiderNet.** The access SpiderNet provides a two-tier service composition model for pervasive computing environments such as smart rooms. The major goal of the access SpiderNet is to perform service adaptation and service distribution in ubiquitous computing. Service adaptation can bridge the gap between the core SpiderNet and the access SpiderNet. Service distribution can overcome the resource constraints of mobile devices by placing service components onto multiple devices.

We validate the feasibility and efficiency of SpiderNet using both prototype implementation and extensive simulations. To demonstrate the effectiveness of SpiderNet, we also implement a set of distributed multimedia applications on top of SpiderNet. Our prototype implementation shows that SpiderNet can provide QoS-aware service composition during setup phase within tens of million seconds in a smart room environment, and within a few seconds in wide-area network environment (i.e., PlanetLab [6]). To evaluate the efficiency of the QoS-aware service composition algorithms, we compare SpiderNet with both optimal solution and other common heuristic algorithms. Our experiments show that SpiderNet can achieve near-optimal performance with low overhead. We also conducted comparative study to evaluate the SpiderNet on top of different overlay topologies. Our results indicate that bounded degree mesh topology is most suitable for QoS-aware service composition.

### 9.2 Open Questions

Open questions still exist in realizing a fully automatic service composition system and, more generally, an autonomic service management framework. We discuss several of the most interesting ones as follows.

• **Secure service composition.** SpiderNet currently lacks the security support in dynamic service composition. Secure service composition is important for open shared computing systems such as P2P networks and becomes necessary for crisis response
systems where composed services must meet certain security requirements. Thus, one possible extension to SpiderNet is to integrate a trust management system into the SpiderNet framework to support secure service composition.

• **Energy efficient service composition.** Energy efficiency is important for composing services in wireless sensor networks. The bounded composition probing scheme used by the P2P SpiderNet has indicated that on-demand selective states collection and proactive failure recovery can greatly reduce the number of control messages, which can potentially lead to energy savings. Thus, one interesting problem is to revise the P2P SpiderNet design with the energy constraint to provide an energy-efficient service composition solution.

• **Mutable component management.** SpiderNet currently assumes that service components in the service overlay network are immutable and always on. Given a user request, SpiderNet discovers qualified service components and composes them into user required service delivery. However, some components, such as data objects or mobile agents, can be migrated among distributed computer hosts. Thus, an interesting open problem is how to properly place, migrate, and replicate mutable components in the service overlay network to provide efficient service delivery.

• **Smart associative computing.** Ubiquitous computing promotes the proliferation of mobile devices which achieve mobility at the expense of constrained resources (e.g., CPU, memory, battery life) and limited I/O capabilities (e.g., small screen). SpiderNet has shown that dynamic device aggregation can effectively alleviate the CPU speed and memory constraints of mobile devices. Under the above premise, we envision a new computing paradigm called smart associative computing where mobile devices can be seamlessly associated with various nearby devices to overcome all kinds of constraints such as CPU, memory, battery power, and input/output limitations. Thus, the user can use a personal mobile device such as a cell-phone or a personal digital assistant as
a portal to seamlessly access any services at anytime anywhere.

- **Overlay network management.** The overlay network, particularly its topologies and routing policies, should be dynamically managed reacting to (1) underlying IP network changes, (2) overlay node departures/arrivals, and (3) malicious attacks. Our experience with SpiderNet has indicated that overlay topology can greatly affect the performance of service composition. SpiderNet currently lacks an efficient and unobtrusive overlay network management subsystem, which can adapt to both underlying IP network changes and upper-level application execution patterns.
References


Vita

Xiaohui Gu was born in October 1976. She received her Bachelor of Science degree in Computer Science from Peking University, Beijing, China in July 1999. In August 1999, She was admitted to the Department of Computer Science in the University of Illinois at Urbana-Champaign. She received her Master of Science degree in Computer Science in January 2001. Her Master’s thesis is entitled “Visual Quality-of-Service Programming Environment for Distributed Heterogeneous Systems”, which is awarded David J. Kuck Best Master Thesis Award in the Department of Computer Science at UIUC. In summer 2001, she worked on adaptive offloading service project as a summer intern at Hewlett Packard Research Labs, Palo Alto, CA. In summer 2002 and 2003, she worked on the quality-aware service management as a summer intern at IBM T.J. Watson Research Center, NY. Her general research interests include distributed systems, computer networks, and operating systems with a current focus on overlay networks, peer-to-peer systems, grid computing, and ubiquitous computing. She was awarded ILLIAC Fellowship and Saburo Muroga Fellowship by the Department of Computer Science at UIUC. After graduation, she will join IBM T.J. Watson Research Center, Hawthorne, NY as a research staff member.