Modeling and Exploiting QoS Prediction in Cloud and Service Computing

ZHANG, Yilei

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
September 2013

Thesis/Assessment Committee

Professor Pak Ching LEE (Chair)
Professor Michael R. LYU (Thesis Supervisor)
Professor Fung Yu YOUNG (Committee Member)
Professor Qing LI (External Examiner)

Abstract of thesis entitled:

Modeling and Exploiting QoS Prediction in Cloud and Service Computing

Submitted by ZHANG, Yilei

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in September 2013

Cloud computing is a new type of Internet-based computing, whereby shared resources, software, and information are provided as services to computers and other devices on demand. The architecture of the software systems involved in the delivery of cloud computing, typically involves multiple cloud components communicating with each other over application programming interfaces (API), usually implemented as Web services. Cloud computing has become a scalable service consumption and delivery platform. Web services are software systems designed to support interoperable machine-to-machine interaction over a network. The technical foundations of cloud computing include Service-Oriented Architecture (SOA), which is becoming a popular and major framework for building Web applications in the era of Web 2.0, whereby Web services offered by different providers are discovered and integrated over the Internet. Quality-of-Service (QoS) is usually employed to describe the non-functional properties of services in cloud and service computing. It becomes important to evaluate the QoS performance of services to differentiate the qualities of service candidates.

However, QoS evaluation is time and resource consuming. Conducting real-world evaluation is difficult in practice. Moreover, in some scenarios, QoS evaluation becomes impossible (e.g., the cloud

provider may charge for service invocations, too many services to be evaluated, etc.). Therefore, it is crucial to study how to build effective and efficient approaches to predict the QoS performance of services.

In this thesis, we first propose three QoS prediction methods which utilize the users' past usage experiences. The first prediction method employs the information of neighborhoods for making QoS value prediction and engages matrix factorization techniques to enhance the prediction accuracy. The second method provides time-aware personalized QoS value prediction service. The third method employs time information for efficient online performance prediction.

The predicted QoS values can be employed to a variety of applications in cloud and service computing. We propose two applications in this thesis. The first application employs QoS information to build a Web service search engine, which help users discover appropriate Web services to fulfill both functional and non-functional requirements. The second application employs dynamic QoS information to build a robust Byzantine fault-tolerant cloud systems.

論文題目： 云計算和服務計算的質量預測建模和應用

作者 ： 章一磊

學校 ： 香港中文大學

學系 ： 計算器科學及工程學系

修讀學位： 哲學博士

摘要 ：

雲計算是一種新型的基於互聯網的計算，即共享的資源、軟件和信息以服務的方式按照需求提供給計算機和其他設備。提供雲計算的軟件系統的體系結構通常涉及多個雲組件，用來在應用程序的接口（API）上互相通信，通常以 Web 服務的方式實現。 雲計算已經成為一個可伸縮的服務消費和交付平台。Web 服務是支持機器之間在交互網絡上互操作的一種軟件系統。雲計算的技術基礎包括面向服務的架構（SOA），不同服務商提供的 Web 服務在互聯網上被發現和集成，這已經成為 Web 2.0 的時代構建 Web 應用程序一種流行和主要的框架。服務質量（QoS）通常用來描述云計算和服務計算中服務的非功能特性。為了區分不同的備選服務的質量，衡量服務的 QoS 性能就變得尤為重要。

然而，衡量服務的 QoS 是非常耗費時間和資源的，在實踐中開展真實的測量時非常困難的。此外，在一些場景中，QoS 評價變得不可能（例如，雲提供商可能會收取組件調用費用，網絡上有太多的服務存在等）。因此，研究如何建立有效和高效的 QoS 性能預測方法，是十分關鍵的問題。

在這篇論文中，我們首先提出了三個 QoS 的預測方法，它們都是利用用戶過去的使用體驗來進行預測。第一種預測方法採用了相似用戶的信息來進行預測，並且使用了矩陣分解技術來提高預測的準確性。第二種方法提供了有時間感知的個性化 QoS 預測服務。第三種方法利用時間信息來提供高效的在線 QoS 性能預測服務。

預測出來的 QoS 信息可以被用在雲計算和服務計算的各種應用上。在這篇論文中，我們提出了兩個應用。第一個應用使用動態的 QoS 信息，以建立一個魯棒的拜占庭容錯雲系統。第二個應用採用 QoS 信息來構建一個 Web 服務搜索引擎，幫助用戶發現合適的 Web 服務，以滿足功能性和非功能性需求。

Acknowledgement

I would like to express my sincere gratitude and appreciation to my supervisors, Prof. Michael R. Lyu. I gain too much from their guidance not only on knowledge and attitude in doing research, but also on the presentation, teaching, and English writing skills. I will always be grateful for their supervision, encouragement and support at all levels.

I am grateful to my thesis committee members, Prof. Fung Yu Young and Prof. Pak Ching Lee for their helpful comments and suggestions about this thesis. My special thanks to Prof. Qing Li who kindly served as the external committee for this thesis. I would like to thank my mentor, Prof. Kishor S. Trivedi , for his guidance, support, insightful opinions and valuable suggestions when I was visiting Duke University as a visiting scholar.

I thank Zibin Zheng, Yu Kang and Jieming Zhu for their effort and constructive discussions in conducting the research work in this thesis. I also thank my colleagues, Xinyu Chen, Yangfan Zhou, Hao Ma, Haiqin Yang, Wujie Zheng, Xin Xin, Junjie Xiong, Tom Chao Zhou, Qirun Zhang, Baichuan Li, Guang Ling, Shouyuan Chen, Chen Cheng, Hongyi Zhang, Shenglin Zhao and many others.

Last but not least, I want to thank my parents. Without their deep love and constant support, this thesis would never have been completed.

To my beloved parents.

Contents

Abstract **i**

Acknowledgement **v**

1 Introduction **1**
1.1 Overview 1
1.2 Thesis Contributions 4
1.3 Thesis Organization 7

2 Backgound Review **12**
2.1 QoS in Cloud and Service Computing 12
2.2 QoS Prediction in Cloud and Service Computing . . 14
2.3 Web Service Searching 16
2.4 Fault-Tolerant Cloud Applications 17

3 Neighborhood-Based QoS Prediction **19**
3.1 Overview 19
3.2 Collaborative Framework in Cloud 22
3.3 Collaborative QoS Prediction 24
3.3.1 Problem Description 25
3.3.2 Latent Features Learning 28
3.3.3 Similarity Computation 30
3.3.4 Missing QoS Value Prediction 31
3.4 Experiments 33
3.4.1 Dataset Description 34

3.4.2 Metrics 35
3.4.3 Performance Comparison 36
3.4.4 Impact of Matrix Density 37
3.4.5 Impact of Top-K 40
3.4.6 Impact of Dimensionality 41
3.4.7 Impact of λ 43
3.5 Summary 44

4 Time-Aware Model-based QoS Prediction 46
4.1 Overview 46
4.2 Collaborative Framework for Web Services 49
4.3 Time-Aware QoS Prediction 51
4.3.1 Problem Description 52
4.3.2 Latent Features Learning 54
4.3.3 Missing Value Prediction 58
4.3.4 Complexity Analysis 59
4.4 Experiments 60
4.4.1 Experimental Setup and Dataset Collection . 60
4.4.2 Metrics 62
4.4.3 Performance Comparisons 63
4.4.4 Impact of Tensor Density 66
4.4.5 Impact of Dimensionality 67
4.5 Summary 70

5 Online QoS Prediction 71
5.1 Overview 71
5.2 Preliminaries 74
5.3 Online Service Level Performance Prediction 77
5.3.1 Problem Description 78
5.3.2 Time-Aware Latent Feature Model 79
5.3.3 Service Performance Prediction 82
5.3.4 Computation Complexity Analysis 86
5.4 System Level Performance Prediction 87

5.5 Experiments 90
5.5.1 Experimental Setup and Dataset Collection . 90
5.5.2 Metrics 92
5.5.3 Comparison 93
5.5.4 Impact of Data Density 97
5.5.5 Impact of Dimensionality 98
5.5.6 Impact of α and w 99
5.5.7 Computational Time Comparisons 101
5.5.8 System Level Performance Case Study . . . 101
5.6 Summary 104

6 QoS-Aware Web Service Searching 106
6.1 Overview 106
6.2 Motivation 110
6.3 System Architecture 112
6.4 QoS-Aware Web Service Searching 113
6.4.1 QoS Model 113
6.4.2 Similarity Computation 116
6.4.3 QoS-Aware Web Service Searching 118
6.4.4 Online Ranking 122
6.4.5 Application Scenarios 123
6.5 Experiments 125
6.5.1 QoS Recommendation Evaluation 125
6.5.2 Functional Matching Evaluation 129
6.5.3 Online Recommendation 130
6.5.4 Impact of λ 133
6.6 Summary 134

7 QoS-Aware Byzantine Fault Tolerance 136
7.1 Overview 136
7.2 System Architecture 139
7.3 System Design 141
7.3.1 System Overview 141

7.3.2 Primary Selection 143
7.3.3 Replica Selection 144
7.3.4 Request Execution 147
7.3.5 Primary updating 148
7.3.6 Replica Updating 149
7.4 Experiments 150
7.4.1 Experimental Setup 151
7.4.2 Performance Comparison 152
7.5 Summary 155

8 Conclusion and Future Work **157**
8.1 Conclusion 157
8.2 Future Work 158

Bibliography **160**

List of Figures

1.1 System Architecture 2
1.2 Thesis Structure 8

3.1 System Architecture 23
3.2 A Toy Example for QoS Prediction 25
3.3 Value Distributions 35
3.4 Impact of Matrix Density 39
3.5 Impact of Top-K 41
3.6 Impact of Dimensionality 42
3.7 Impact of λ 43

4.1 System Architecture 50
4.2 A Toy Example 51
4.3 User-Service-Time Tensor 52
4.4 Response-Time of Two Pairs of User-Service 56
4.5 QoS Value Distributions 62
4.6 Impact of Tensor Density 66
4.7 Impact of Dimensionality in Response-Time Dataset 68
4.8 Impact of Dimensionality in Throughput Dataset . . 69

5.1 Service-Oriented System Architecture 75
5.2 Online Performance Prediction Procedures 75
5.3 A Toy Example of Performance Prediction 78
5.4 Basic Compositional Structures 88
5.5 A Performance Composition Example 88
5.6 Response Time Value Distribution 92
5.7 Impact of Data Density 97

5.8 Impact of Dimensionality 98
5.9 Impact of α and w 99
5.10 An Online Shopping System 102
5.11 System Performance Improvement of Dynamically Service Composition 103

6.1 Service-Oriented System Architecture 107
6.2 Web Service Query Scenario 109
6.3 System Architecture 112
6.4 Value Distributions 126
6.5 NDCG of Top-K Web services 127
6.6 Recall and Precision Performance 130
6.7 QoS Value Distributions of Online Dataset 131
6.8 NDCG of Online Recommendation 132
6.9 Impact of λ 133

7.1 Architecture of Cloud Applications 139
7.2 Architecture of BFTCloud in Voluntary-Resource Cloud 140
7.3 Work Procedures of BFTCloud 141
7.4 Throughput Comparison for 0/0, 4/0, and 0/4 benchmarks as the number of cloud modules varies 153

List of Tables

3.1 Statistics of WS QoS Dataset 35
3.2 Performance Comparsions (A Smaller MAE or RMSE Value Means a Better Performance) 38

4.1 Statistics of WS QoS Dataset 62
4.2 Performance Comparsions (A Smaller MAE or RMSE Value Means a Better Performance) 64

5.1 Calculation of Aggregated Response Time 89
5.2 Statistics of Web Service Response Time Dataset . . 92
5.3 Performance Comparsions (A Smaller MAE or RMSE Value Means a Better Performance) 94
5.4 Performance Improvement of OPred 96
5.5 Average Computational Time Comparisons 100

6.1 User Query Examples 110
6.2 Web Service Examples 110
6.3 Statistics of WS QoS Dataset 126
6.4 NDCG values (A larger NDCG value means a better performance) 128
6.5 Statistics of Online QoS Dataset 131

7.1 Average Sending Times Per Request 154
7.2 Correct Rate of Committed Requests 155

Chapter 1

Introduction

1.1 Overview

Cloud computing [6, 25] is a new type of Internet-based computing, whereby shared resources, software, and information are provided to computers and other devices on demand [44]. With the exponential growth of cloud computing as a solution for providing flexible computing resources, more and more cloud applications emerge in recent years. The architecture of the Software-as-a-Service (SaaS) systems in the delivering of cloud computing, typically involves multiple cloud components communicating with each other over application programming interfaces, usually Web services. [110]. Cloud computing has become a scalable service consumption and delivery platform.

Web services are software systems designed to support interoperable machine-to-machine interaction over a network [50]. The technical foundations of cloud computing include Service-Oriented Architecture (SOA), which is becoming a popular and major framework for building Web applications in the era of Web 2.0 [78], whereby Web services offered by different providers are discovered and integrated over the Internet. Typically, a service-oriented system consists of multiple Web services interacting with each other over the Internet in an arbitrary way. In this thesis, service refers to Web service in service computing and cloud component which is delivered

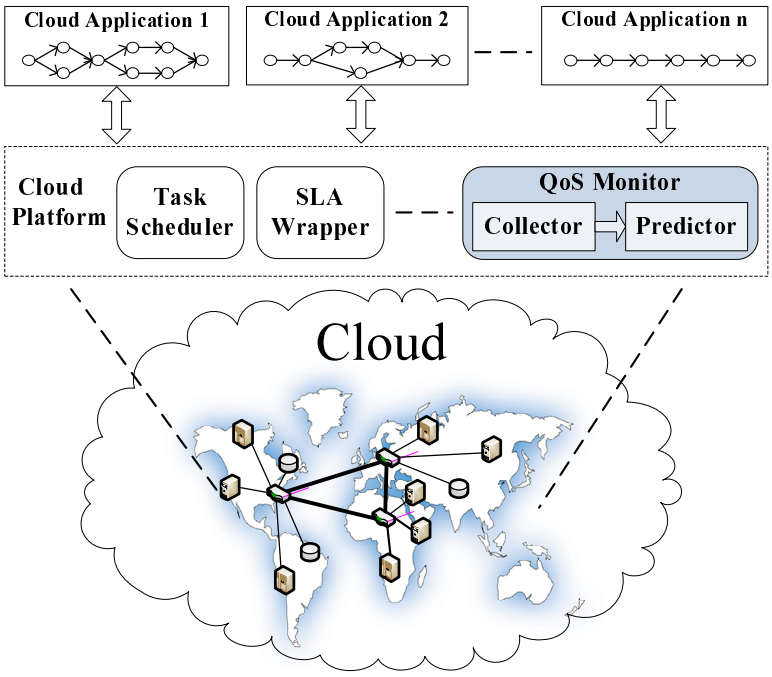


Figure 1.1: System Architecture

as a service in cloud computing.

Figure 1.1 shows the system architecture in cloud computing. In a cloud environment, the cloud provider holds a large number of distributed services (e.g. databases, servers, Web services, etc.), which can be provided to designers for developing various cloud applications. Designers of cloud applications can choose from a broad pool of distributed services when composing cloud applications. These services are usually invoked remotely through communication links and are dynamically integrated into the applications. The cloud application designers are located in different geographic and network environments. Since the users invoke services via different communication links, the quality of services they observed are diverse.

Quality-of-Service (QoS) is usually employed to describe the non-functional characteristics of services. It becomes a major concern for application designers when making service selection [43]. Moreover, for the existing cloud applications, by replacing low quality

services with better ones, the overall quality of cloud applications can be improved.

In recent year, a number of research tasks have been focused on optimal service selection [10, 115] and recommendation [128] in distributed systems or service computing. Typically, evaluations on the service candidates are required to obtain their QoS values. In cloud environment, due to their various locations and communication links, different users will have different QoS experiences when invoking even the same service. Personalized QoS evaluation is required for each user at the user-side. However, a service user in general only invoked a limited number of services in the past and only received QoS performance information of these invoked services. In practice, therefore, conducting real-world evaluations on services to obtain their QoS information from the users' perspective is quite difficult, because: (1) executing invocations for evaluation purposes becomes too expensive, since cloud providers who maintain and host services (e.g., Amazon EC2[1], Amazon S3[2], etc.) may charge for invocations; (2) with the growing number of available services over the Internet, it is time-consuming and impractical to conduct QoS evaluations on all accessible services; (3) component users need to focus on building cloud applications on top of various services. While conducting evaluation on a large number of service candidates would introduce extra cost and effort, and sharply slow down the application development progresses. Therefore, collecting historical usage records and conducting QoS prediction, which requires no additional invocation, is becoming an attractive approach. Based on the above analysis, in order to provide QoS information to application designers, we need to provide comprehensive investigation on QoS prediction approaches.

Employing the predicted QoS values, a QoS-aware Web service search engine can be enabled. Traditional Web service searching

[1]http://aws.amazon.com/ec2
[2]http://aws.amazon.com/s3

approaches only find the Web services to fulfill users' functionality requirements. However, Web services sharing similar functionalities may possess very different non-functionalities (e.g., response time, throughput, availability, usability, performance, integrity, etc.). Web services recommended by the traditional searching approach may not fulfill users' non-functional requirements. In order to find appropriate Web services which can fulfill both functional and non-functional requirements of users efficiently, QoS-aware searching approaches are needed.

Given the predicted QoS information, robust systems can be built based on redundant services by employing QoS-aware fault tolerance framework. Traditional fault tolerance framework [72] usually requires developing several different version of system services. However, due to the cost of development, the fault tolerance strategies are usually employed only for critical systems. In cloud computing, however, users can access multiple functional equivalent services via Internet at a very low cost. These services are usually developed and provided by different organizations, and can be dynamically composed to build a fault tolerance systems. Although some fault tolerance framework [65, 70, 124] have been proposed for traditional software systems, they cannot adopt to the highly dynamic cloud environment.

In order to provide accurate QoS prediction approaches, QoS-aware Web service searching mechanisms, and QoS-aware fault tolerant frameworks for cloud systems, we proposed five approaches to attack these challenges in this thesis.

1.2 Thesis Contributions

The main contributions of this thesis can be described as follows:

(1) **Neighborhood-Based QoS Prediction**
We formally identify the research problem of QoS value prediction in cloud computing. In order to accurately predict the

personalized service QoS values and to efficiently deliver the QoS information to cloud application designers, we propose a neighborhood-based approach by employing the historical QoS data to model the features on both users and services. Our approach learns the characteristics of users by non-negative matrix factorization (NMF) and explores QoS experiences from similar users to achieve high QoS value prediction accuracy. Our approach requires no additional invocation and can work well when the user-service matrix is very sparse. We further extend the approach to adopt the dynamic QoS information in real-time. Moreover, we conduct large-scale real-world experiments to study the QoS prediction result of our approach. 339 distributed users located in 30 countries and $5,825$ openly-accessible services from 73 countries are involved in the experiments. The experimental results show that compared with other approaches, our approach can achieve higher prediction accuracy.

(2) **Time-Aware Model-based QoS Prediction**
Traditional QoS prediction approaches usually try to prediction the average QoS performance of Web services. However, QoS performance of Web services is highly related to invocation time. The dynamic service status and the network environment make traditional QoS prediction approaches hard to be used in practice. To address this critical challenge, we propose a model-based time-aware personalized approach for Web service QoS prediction. The contributions of this chapter are threefold: (1) we formally define the critical problem of time-aware Web service QoS prediction. A user-side light-weight middleware is designed for automatically recording and sharing QoS experiences; (2) we employ the tensor factorization technique to systematically analyze the latent features of user, service and time, which are further utilized to addresses the difference over time in service computing literature; (3) we conduct large-scale real-world experiments to study the prediction accuracy and effi-

ciency of our approach compared with other state-of-the-art approaches. The experimental results show that our approach can predict QoS values more accurate than other approaches. Moreover, we publicly release our large-scale time-stamped Web service QoS dataset for future research.

(3) **Online QoS Prediction**
The service status and network environments are highly variable over time, which requires efficient QoS prediction of Web services at run-time. In this chapter, we propose an online QoS prediction approach for Web services. The contributions of this chapter include: (1) identifying the crucial problem of online QoS prediction for Web services and proposing a novel prediction framework to provide QoS value estimation at run-time. By employ time series analysis on feature trends, more accurate and efficient QoS prediction of Web service can be achieved. The complexity of our method is linear with the amount of newly observed performance information, which shows that our approach can make QoS prediction timely; (2) conducting large-scale extensive experiments for evaluating the effectiveness and efficiency of our approach. Moreover, we further develop an aggregation approach to predict a service-oriented system performance by utilizing the results of Web service QoS prediction. The experimental results and the system level case study show the efficiency and effectiveness of our approach.

(4) **QoS-Aware Web Service Searching**
This chapter aims at improving the current research of Web service searching by proposing a brand new Web service searching approach. Our approach systematically fusing the functional approach and non-functional approach to achieve better performance. The contributions of this chapter are three-fold: (1) different from previous approaches which only employ functional features or non-functional features, we propose the first Web

service searching approach which considers both functional and non-functional qualities of the service candidates; (2) we conduct a large-scale distributed experimental evaluation to verify the proposed approach. Functional and non-functional characters of $3,738$ Web services from 69 countries are studied. The experimental results demonstrate the effectiveness of our Web service searching approach; (3) we publicly release the real-world WSDL files of Web services and corresponding QoS records. More than 30 institutes have downloaded and utilized our datasets for extensive Web service research.

(5) **QoS-Aware Byzantine Fault Tolerance**
Due to the highly dynamic environment, traditional fault tolerance cannot tolerate malicious behaviors in cloud computing. To address this critical challenge, we propose a Byzantine fault tolerance framework for building robust cloud systems. The contributiona of this chapter are as following: (1) we identify the Byzantine fault tolerance problem in voluntary-resource cloud and propose a Byzantine fault tolerance framework to guaranteeing the robustness of cloud application. Our approach uses dynamical replication techniques to tolerate various types of faults; (2) we build a prototype system on a voluntary-resource cloud, which consists of 257 cloud resources from in 26 countries, to test the fault tolerance middleware we designed; (3) we conduct large-scale real-world experiments to study the reliability performance of our approahce. The experimental results indicate that our approach can tolerate various types of faults in cloud computing.

1.3 Thesis Organization

As shown in Figure 1.2, the rest of this thesis is organized as follows:

- Chapter 2

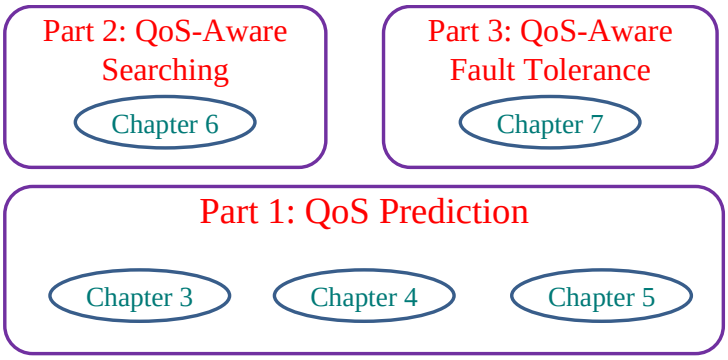


Figure 1.2: Thesis Structure

This chapter briefly reviews some background knowledge and work related to the main methodology that will be explored in this thesis.

- Chapter 3
  In this chapter, we propose a novel neighborhood-based approach (CloudPred), which is enhanced by character modeling, for providing collaborative and personalized QoS prediction of cloud components. We first present the QoS prediction scenario by a toy example. Then the QoS prediction problem in cloud computing is formally defined. After that, we present a latent feature learning algorithm to learn the user-specific and service-specific latent features. Based on the latent features, user and service similarity computation approaches are introduced. By identifying similar users and similar services to the active user-service pair, we formulate the CloudPred prediction Algorithm. We conduct extensive experiments to study the prediction accuracy of CloudPred and the impact of various parameters. The experimental results show that CloudPred achieves higher prediction accuracy than other competing methods.

- Chapter 4
  In this chapter, we present a model-based time-aware collab-

orative filtering approach for personalized QoS prediction of Web services. First, we endow a new understanding of user-perspective QoS experiences, which is based on the following observations: (1) during different time intervals, a user has different QoS experiences on the same Web service; (2) in general, the differences are limited within a range. Based on these observations, we formulate the time-aware personalized QoS prediction problem as the tensor factorization problem, and propose an optimization formulation with average QoS constraint. Second, we propose to predict the missing QoS values by evaluating how the user, service, and time latent features are applied to each other. Furthermore, we provide a comprehensive complexity analysis of our approach, which indicates that our approach is efficient and can be applied to large-scale systems. Extensive experiments are conducted to evaluate the prediction accuracy and parameter impacts. The experimental results show the effectiveness and efficiency of our time-aware QoS prediction approach.

- Chapter 5
  In this chapter, we present an online Web service QoS prediction approach by performing time series analysis on user-specific and service-specific latent features. Our online prediction approach includes four phases. In Phase 1, service users monitor the performance of Web service and keep the QoS records in local site. In Phase 2, distributed service users submit local QoS records to the performance center in order to obtain a better QoS prediction service from the performance center. The performance center collects QoS records from different users and generates a set of global QoS matrices. In Phase 3, a set of time-stamped user latent feature matrices and service latent feature matrices are learned from the global QoS matrices. After that, time series analysis are conducted on the latent matrices to build a QoS model in the performance center.

By evaluating how each factor applies to the active user and the corresponding service in the QoS model, personalized QoS prediction results can be returned to users on demand. In Phase 4, the system level QoS performance of service-oriented architecture is predicted by analyzing the service compositional structure and utilizing the service QoS prediction results. The complexity analysis indicates that our approach is efficiency and can be applied to large-scale online service-oriented systems. Finally, we conduct a number of experiments to study the performance of our approach and the impacts of algorithm parameters. We also study the effects of integrating service QoS information into the dynamic composition mechanism by a real-world service-oriented system case.

- Chapter 6
  In this chapter, we propose a QoS-aware Web service searching approach to explore the appropriate Web services to fulfill users' functional and non-functional requirements. We first describe the Web service searching scenarios and present the system architecture. Then, we present the QoS model to evaluate the non-functional utility of Web services. After that, functional similarity is introduced to evaluate the functional utility of Web services. Two QoS-aware Web service searching approaches are proposed: the score-based combination and the ranking-based combination. We further extend the ranking-based approach to online searching scenario. Moreover, three common application scenarios are introduced. Finally, a number of experiments are conducted to study the functional and non-functional performance of our approach. The comprehensive results of experiments show that our approach provides better Web service searching results.

- Chapter 7
  This chapter presents a fault tolerance framework for building

robust cloud applications at runtime. Our approach adopt dynamic QoS information to enable automatic system reconfiguration. We first introduce the architecture of our framework in voluntary-resource cloud. Then we present the work procedures of our approach in detail, including 5 phases: primary selection, replicas selection, request execution, primary updating, and replica updating. After that, we conduct real-world experiments by deploying the prototype of our approach as a middleware in a voluntary-resource cloud environment, which consists of 257 distributed computers located in 26 countries. The experimental results show that our approach guarantees high reliability enables good performance of cloud systems.

- Chapter 8
  The last chapter summarizes this thesis and provides some future directions that can be explored.

In order to make each of these chapters self-contained, some critical contents, e.g., model definitions or motivations having appeared in previous chapters, may be briefly reiterated in some chapters.

□ **End of chapter.**

Chapter 2

Backgound Review

2.1 QoS in Cloud and Service Computing

Cloud computing [6] has been in spotlight recently. Cloud computing has become a scalable services consumption and delivery platform [118]. The technical foundations of cloud computing include Service-Oriented Architecture (SOA) [33]. SOA is becoming a popular and major framework for building Web applications in the era of Web 2.0 [78]. A number of investigations have been carried out focusing on different kinds of research issues such as Web service selection [32, 112, 115, 117], Web service composition [3, 4, 113], SOA failure prediction [9], SOA reliability prediction [125], fault tolerance [65, 121], resiliency quantification [36], service ranking [128], resource consistency [95], resource allocation [27], workload balance [104], dynamically resource management [56], etc.

Quality-of-Service (QoS) has been widely employed as a quality measure for evaluating non-functional features of software systems and services [1, 115, 119]. A lot of research works have utilized QoS to describe the characteristics of services in cloud and service computing [51, 76, 79, 80, 87, 103]. Zeng et al. [116] use five QoS properties to compose Web service dynamically. Ardagna et al [5] employ five QoS properties to conduct flexible service composition processes. Alrifai et al [3] consider generic and domain-specific

QoS for efficient service composition.

QoS performance of services can be measured either from the provider's perspective or from the user's observation. QoS values measured at the service provider side (e.g. price, availability, etc.) are usually identical for different users, such as QoS used in the Service Level Agreement (SLA) [71] (e.g., IBM [58] and HP [88]). While QoS values observed by different users may vary significantly due to the unpredictable communication links and heterogeneous user environments. In this thesis, we mainly focus on observing QoS data from users' perspective and making use of the QoS data for QoS prediction, service selection, service searching, and fault tolerant framework building.

Based on the QoS performance of services, several approaches have been proposed for optimizing service selection [8, 10, 15, 31, 100, 115, 126] in improving the whole quality of web application, Web service composition [5, 3, 15, 16, 116], Web service recommendation [23, 103, 127], reliability prediction [17, 24, 38, 41, 86, 125], etc. Traditionally, reliability of a software system [73] is analyzed without considering the system performance, which is not accurate when applied to modern systems. Moreover, several QoS-aware approaches [28, 75, 87, 111, 115, 116] are proposed in cloud and service computing.

However, there is few real-world QoS data to verify these QoS-aware approaches. To collect the QoS data from the user-side, Zheng et al. [129] proposed a distributed evaluation framework and released the QoS datasets for further extensive research. Different from previous work [2, 106], they conduct large-scale real-world evaluations.

2.2 QoS Prediction in Cloud and Service Computing

The QoS-aware approaches usually assume that the QoS values are already known, while in reality a user cannot exhaustively invoke all the services. Although there existed some QoS evaluation approaches and publicly released QoS datasets, it is impossible to conduct personalized evaluation on all accessible services for all users. In this chapter, we focus on predicting missing QoS values by collaborative filtering approach to enable the QoS-aware approaches.

Collaborative filtering approaches are widely adopted in commercial recommender systems [12, 125]. Generally, traditional recommendation approaches can be categorized into two classes: memory-based and model-based. Memory-based approaches, also known as neighborhood-based approaches, are one of the most popular prediction methods in collaborative filtering systems. Memory-based methods employ similarity computation with past usage experiences to find similar users and services for making the performance prediction. The typical example of memory-based collaborative filtering include user-based approaches [11, 22, 45, 55, 97], item-based approaches [29, 52, 94, 67], and their fusion [40, 107, 125]. Typically, memory-based approaches employ the PCC algorithm [85] for similarity computation.

Model-based approaches employ machine learning techniques to fit a predefined model based on the training datasets. Model-based approaches include several types: the clustering models [114], the latent factor models [89], the aspect models [46, 47, 98, 99], etc. Lee et al. [63] presented an algorithm for non-negative matrix factorization that is able to learn the parts of facial images and semantic features of text. It is noted that there is only a small number of factors influencing the service performance in the user-service matrices, and that a user's factor vector is determined by how much each factor applies to that user. For a set of user-service matrices data,

three-dimensional tensor factorization techniques are employed for item recommendation [84].

The memory-based approaches employ the information from similar users and services for predicting missing values. When the number of users or services is too small, similarity computation for finding similar users or services is not accurate. When the number of users or services is too large, calculating similarity values for each pair of users or services is time-consuming. In contrast, model-based approaches are very efficient for missing value prediction, since they assume that only a small number of factors influence the service performance.

There is few work of collaborative filtering prediction for QoS values in cloud and service computing, since there lacks large-scale real-world QoS datasets for verifying the prediction accuracy. Some approaches [57, 101] employing a movie rating dataset, MovieLens [85] for simulation. Shao et al. [96] only conduct a small-scale experiments which involves 20 Web services for evaluating prediction accuracy.

The existing methods in the literature only consider two dimensions (i.e., user and Web service) while time factor is not included. The periodic features of service QoS values are ignored, which may improve the prediction accuracy significantly. Moreover, the high computational complexity makes it difficult to extend memory-based approaches to handle large amounts of time-aware performance data for timely prediction. There is a lack of fast algorithms to predict the QoS values at runtime to adapt the highly dynamic system environment in cloud and service computing.

In this thesis, we propose three approaches to address the QoS prediction problems in cloud and service computing, including memory-based prediction [122], time-aware prediction [120], and online prediction [123] approaches. We also conduct large-scale real-world experiments to verify the prediction accuracy and release the QoS datasets for further studies of other researchers.

2.3 Web Service Searching

Web service discovery [83] is a fundamental research area in service computing. Several papers in the literature conduct investigations on discovering Web services through syntactic or semantic tag matching in a centralized UDDI repository [81, 105]. However, since UDDI repository is no longer a popular style for publishing Web services, these approaches are not practical now.

Text-based matching approaches have been proposed for querying Web service [39, 108]. These works employ term frequency analysis to perform keywords searching. However, most text descriptions are highly compact, and contain a lot of unrelated information to the Web service functionality. The performances of this approaches are not fine in practice. Plebani et al. [82] extract the information from WSDL files for Web service matching. By comparing with other works [30, 42, 54], it shows better performance in both recall and precision. However, it also dose not consider non-functional qualities of Web services. Our searching approach, on the other hand, take both functional and non-functional features into consideration.

Alrifai et al. [3], Liu et al. [69] and Tao et al. [115] focus on efficiently QoS-driven Web service selection. Their works are all based on the assumption: the Web service candidates which can be select for composition have already been discovered and all meet requesters' functional requirements. Under this assumption these approaches cannot be directly applied into Web service search engine. In this thesis, we proposed WSExpress [119], a QoS-aware searching approach which employs both QoS and functionality information, to search appropriate Web services for users.

2.4 Fault-Tolerant Cloud Applications

Software fault tolerance techniques (e.g., N-Version Programing [7], distributed recovery block [59], etc.) are widely employed for building reliable systems [72]. Zhang et al. [119] propose a Web service search engine for recommending reliable Web service replicas. Salas et al. [90] propose an active strategy to tolerate faults in Web services. Zheng et al. [130] propose a ranking-based fault tolerance framework for building reliable applications in cloud. There are many fault tolerance strategies have been proposed for Web services [19, 20, 35, 91]. Typically, the fault tolerance strategies can be divide into two major types: passive strategies and active strategies. Passive strategies include FT-CORBA [66], FT-SOAP [65], etc. Active strategies include WS-Replication [90], SWS [64], FTWeb [93], etc.

However, these techniques cannot tolerate Byzantine faults like malicious behaviors. There are some works focus on Byzantine fault tolerance for Web services as well as distributed systems. BFT-WS [124] is a Byzantine fault tolerance framework for Web services. Based on Castro and Liskov's practical BFT algorithm [18], BFT-WS considers client-server application model running in an asynchronous distributed environment with Byzantine faults. $3f + 1$ replications are employed in the server-side to tolerate f Byzantine faults. Thema [77] is a Byzantine Fault Tolerant(BFT) middleware for Web services. Thema supports three-tiered application model, where the $3f + 1$ Web service replicas need to invoke an external Web service for accomplishing their executions. SWS [64] is a survivable Web Service framework that supports continuous operation in the presence of general failures and security attacks. SWS applies replication schemes and N-Modular Redundancy concept. Each web service is replicated into a service group to mask faults.

Different from above approaches, BFTCloud [121] proposed in this thesis aims to provide Byzantine fault tolerance for voluntary-

resource cloud, in which Byzantine faults are very common. BFT-Cloud select voluntary nodes based on both their reliability and performance characteristics to adapt to the highly dynamic voluntary-resource cloud environment.

□ **End of chapter.**

Chapter 3

Neighborhood-Based QoS Prediction

3.1 Overview

Cloud computing [6] is Internet-based computing, whereby shared resources, software, and information are provided to computers and other devices on demand. With the exponential growth of cloud computing as a solution for providing flexible computing resources, more and more cloud applications emerge in recent years. The systems architecture of the software systems involved in the delivery of cloud computing (named as cloud applications in this chapter), typically involves multiple cloud components communicating with each other over application programming interfaces, usually Web services [110]. How to build high-quality cloud applications becomes an urgent and crucial research problem.

In the cloud environment, designers of cloud applications, denoted as component users, can choose from a broad pool of cloud components when creating cloud applications. These cloud components are usually invoked remotely through communication links. Quality of the cloud applications is greatly influenced by the quality of communication links and the distributed cloud components. To build a high-quality cloud application, non-functional Quality-of-Service (QoS) performance of cloud components becomes an im-

portant factor for application designers when making component selection [43]. Moreover, for the existing cloud applications, by replacing low quality components with better ones, the overall quality of cloud applications can be improved.

In recent year, a number of research tasks have been focused on optimal component selection [10, 115] and recommendation [128] in distributed systems or service computing. Typically, evaluations on the component candidates are required to obtain their QoS values. In cloud environment, due to their various locations and communication links, different users will have different QoS experiences when invoking even the same cloud component. Personalized QoS evaluation is required for each user at the user-side. However, a cloud component user in general only invoked a limited number of cloud components in the past and only received QoS performance information of these invoked cloud components. In practice, therefore, conducting real-world evaluations on cloud components to obtain their QoS information from the users' perspective is quite difficult, because:

- Executing invocations for evaluation purposes becomes too expensive, since cloud providers who maintain and host cloud components (e.g., Amazon EC2[1], Amazon S3[2], etc.) may charge for invocations.
- With the growing number of available cloud components over the Internet, it is time-consuming and impractical to conduct QoS evaluations on all accessible cloud components.
- Component users need to focus on building cloud applications on top of various cloud components. While conducting evaluation on a large number of component candidates would introduce extra cost and effort, and sharply slow down the application development progresses.

[1] http://aws.amazon.com/ec2
[2] http://aws.amazon.com/s3

Based on the above analysis, it is crucial for the cloud platform to deliver a personalized QoS information service to the application designers for cloud component evaluation. In order to provide personalized QoS values on m cloud components for n users by evaluation, at least $n \times m$ invocations need to be executed, which is almost impossible when n and m is very large. However, without sufficient and accurate personalized QoS values of cloud components, it is difficult for the application designers to select optimal cloud component for building high-quality cloud applications. It is an urgent task for the cloud platform providers to develop an efficient and personalized prediction approach for delivering the QoS information service to cloud application designers.

To address this critical challenge, we propose a neighborhood-based approach, called CloudPred, for personalized QoS prediction of cloud components. CloudPred is enhanced by feature modeling on both users and components. The idea of CloudPred is that users sharing similar characteristics (e.g., location, bandwidth, etc.) would receive similar QoS usage experiences on the same component. The QoS value of cloud component c observed by user u can be predicted by exploring the QoS experiences from similar users of u. A user is similar to u if they share similar characteristics. The characteristics of different users can be extracted from their QoS experiences on different components by performing non-negative matrix factorization (NMF). By sharing local QoS experience among users, our approach CloudPred can effectively predict the QoS value of a cloud component c even if the current user u has never invoked the component c before. The experimental results show that compared with other well-known collaborative prediction approaches, CloudPred achieves higher QoS prediction accuracy of cloud components. Since CloudPred can precisely characterize users features (will be introduced in Section 3.3.2), even if some users have few local QoS information, CloudPred can still achieve high prediction accuracy.

In summary, this chapter makes the following contributions:

1. We formally identify the research problem of QoS value prediction in cloud computing and propose a novel neighborhood-based approach, named CloudPred, for personalized QoS value prediction of cloud components. CloudPred learns the characteristics of users by non-negative matrix factorization (NMF) and explores QoS experiences from similar users to achieve high QoS value prediction accuracy. We consider CloudPred as the first QoS value prediction approach in cloud computing literature.

2. We conduct large-scale experiments to study the prediction accuracy of our CloudPred compared with other approaches. The experimental results show the effectiveness of our approach. Moreover, we also publicly release our large-scale QoS dataset[3] for future research.

The remainder of this chapter is organized as follows: Section 3.2 describes the collaborative QoS framework in cloud environment. Section 3.3 presents our CloudPred approach in detail. Section 3.4 introduces the experimental results. Section 3.5 concludes the chapter.

3.2 Collaborative Framework in Cloud

Figure 3.1 shows the system architecture in cloud computing. In a cloud environment, the cloud provider holds a large number of distributed cloud components (e.g. databases, servers, Web services, etc.), which can be provided to designers for developing various cloud applications. The cloud application designers, called component users in this chapter, are located in different geographic and network environments. Since users invoke cloud components via differ-

[3]http://www.wsdream.net

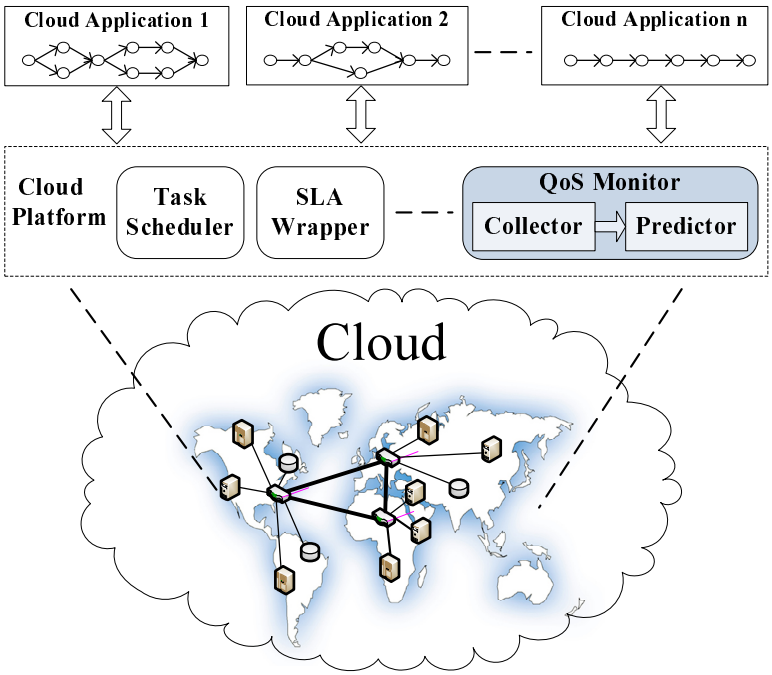


Figure 3.1: System Architecture

ent communication links, their usage experiences on cloud components are diverse in several QoS properties including response-time, throughput, etc. In order to provide personalized quality information of different components to application designers for optimal component selection, personalized QoS value prediction is an essential service of a cloud provider.

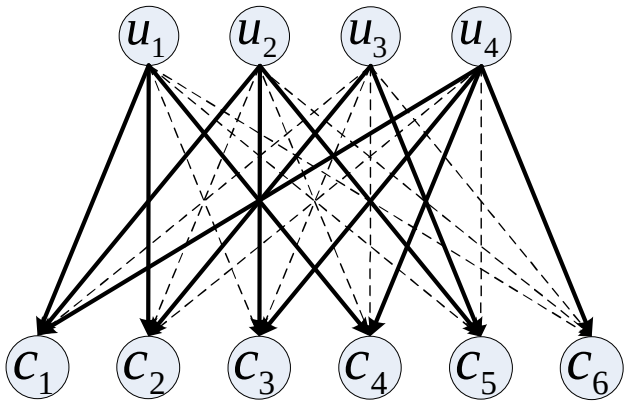
Within the cloud platform provided by a cloud provider, there are several modules implemented for managing the cloud components. Examples of management modules include *Task Scheduler*, which is responsible for task scheduling, *SLA Wrapper*, which is responsible for service level negotiation between cloud provider and users, etc. In this chapter, we focus on the design of *QoS Monitor*, which is responsible for monitoring the QoS performance of cloud components from the users' perspective. The *QoS Monitor* consists of two sub-units: *Collector*, which is used to collect QoS usage information from various component users, and *Predictor*, which is supposed to

provide personalized QoS value prediction for different component users.

The idea of our approach is to share local cloud component usage experience from different component users, to combine this local information to get a global QoS information of all components, and to make personalized QoS value prediction based on both global and local information. As shown in Figure 3.1, each component user keeps local records of QoS usage experiences on cloud components. Since cloud applications are running on an identical cloud platform, QoS information can be collected by an identical interface on the platform side. If a component user would like to get personalized QoS information service from the cloud provider, authorization should be given to *Collector* for accessing its local QoS records. *Collector* then collect those local QoS records from different component users. Based on the collected QoS information, *Predictor* can perform personalized QoS value prediction and forward the prediction results to component users for optimizing the design of cloud applications. The detailed collaborative prediction approach will be presented in Section 3.3.

3.3 Collaborative QoS Prediction

We first formally describe the QoS value prediction problem on cloud components in Section 3.3.1. Then we learn the user-specific and component-specific features by running latent features learning algorithm in Section 3.3.2. Based on the latent features, similarities between users and components are calculated in Section 3.3.3. Finally, the missing QoS values are predicted by applying the proposed algorithm CloudPred in Section 3.3.4.



(a) User-Component Invocation Graph

	c_1	c_2	c_3	c_4	c_5	c_6
u_1	0.98	0.23		0.22		
u_2	0.13		0.27		0.25	
u_3		0.37			0.36	
u_4	0.69		0.22	0.22		0.34

(b) User-Component Matrix

	c_1	c_2	c_3	c_4	c_5	c_6
u_1	0.98	0.23	**0.24**	0.22	**0.22**	**0.53**
u_2	0.13	**0.26**	0.27	**0.25**	0.25	**0.43**
u_3	**0.85**	0.37	**0.38**	**0.36**	0.36	**0.12**
u_4	0.69	**0.23**	0.22	0.22	**0.22**	0.34

(c) Predicted User-Component Matrix

Figure 3.2: A Toy Example for QoS Prediction

3.3.1 Problem Description

Let us first consider a typical toy example in Figure 3.2(a). In this bipartite graph $G = (U \cup C, E)$, its vertices are divided into two disjoint sets U and C such that each edge in E connects a vertex in U and one in C. Let $U = \{u_1, u_2, \cdots, u_4\}$ be the set of component users, $C = \{c_1, c_2, \cdots, c_6\}$ denote the set of cloud components, and E (solid lines) represent the set of invocations between U and C. This bipartite graph G is modeled as a weighted directed graph. Given a pair (i, j), $u_i \in U$ and $c_j \in C$, edge e_{ij} is included in E if user u_i has invoked component c_j before. The weight w_{ij} on edge e_{ij} corresponds to the QoS value (e.g., response-time in this example) of that invocation. Given the set E, our task is to effectively predict the weight of potential invocations (the broken lines).

The process of cloud component QoS value prediction is illus-

trated by a user-component matrix as shown in Figure 3.2(b), in which each entry denotes an observed weight in Figure 3.2(a). The problem we study in this chapter is then how to precisely predict the missing entries in the user-component matrix based on the existing entries. Once the missing entries are accurately predicted, we can provide users with personalized QoS information, which is valuable for automatic component ranking, component selection, task scheduling, etc.

We observe that although about half of the entries are already known in Figure 3.2(b), every pair of users still have very few commonly invoked components (e.g., u_1 and u_2 only invoke c_1 in common, u_3 and u_4 have no commonly invoked components even if together they invoke all the six components). Since the similarity between two users are calculated by comparing their obtained QoS values on common components, the problem of few common components observed above makes it extremely difficult to precisely calculate similarity between users. Motivated by latent factor model [89], we therefore first factorize the sparse user-component matrix and then use $V^T H$ to approximate the original matrix, where the low-dimensional matrix V denotes the user latent feature space, and the low-dimensional matrix H represents the low-dimensional item latent feature space. The rows in V and H represent different features. Each column in V represents an user and each column in H denotes a component. The value of a entry in the matrices indicates how the associated feature applies to the corresponding user or component. In this example we use four dimensions to perform the matrix factorization and obtain:

$$V = \begin{bmatrix} 0.32 & 0.15 & 0.31 & 0.33 \\ 0.23 & 0.15 & 0.26 & 0.28 \\ 0.30 & 0.20 & 0.24 & 0.34 \\ 0.47 & 0.23 & 0.59 & 0.21 \end{bmatrix},$$

$$H = \begin{bmatrix} 0.73 & 0.35 & 0.31 & 0.26 & 0.32 & 0.42 \\ 0.60 & 0.31 & 0.27 & 0.22 & 0.28 & 0.36 \\ 0.69 & 0.37 & 0.32 & 0.27 & 0.33 & 0.45 \\ 0.95 & 0.46 & 0.42 & 0.35 & 0.41 & 0.54 \end{bmatrix},$$

where columns in V and H denote the latent feature vectors of users and components respectively.

Note that V and H are dense matrices with all entries available. Then we calculate the similarity between users and components using 4-dimensional matrices V and H respectively. Therefore, all the missing values can be predicted by employing neighborhood-based collaborative method, as shown in Figure 3.2(c).

Now we formally define the problem of cloud component QoS value prediction as follows: Given a set of users and a set of components, predict the missing QoS value of components when invoked by users based on existing QoS values. More precisely:

> *Let U be the set of m users and C be the set of n components. A QoS element is a triplet (i, j, q_{ij}) representing the observed quality of component c_j by user u_i, where $i \in \{1, \cdots, m\}$, $j \in \{1, \cdots, n\}$ and $q_{ij} \in \mathbb{R}^k$ is a k dimension vector representing the QoS values of k^{th} criteria. Let Ω be the set of all pairs $\{i, j\}$ and Λ be the set of all known pairs (i, j) in Ω. Consider a matrix $W \in \mathbb{R}^{m \times n}$ with each entry w_{ij} representing the observed k^{th} criterion value of component c_j by user u_i. Then the missing entries $\{w_{ij}|(i, j) \in \Omega - \Lambda\}$ should be predicted based on the existing entries $\{w_{ij}|(i, j) \in \Lambda\}$.*

Typically the QoS values can be integers from a given range (e.g. $\{0, 1, 2, 3\}$) or real numbers of a close interval (e.g. $[-20, 20]$). Without loss of generality, we can map the QoS values to the interval $[0, 1]$ using the function $f(x) = (x - w_{min})/(w_{max} - w_{min})$, where w_{max} and w_{min} are the maximum and minimum QoS values respectively.

3.3.2 Latent Features Learning

In order to learn the features of the users and components, we employ matrix factorization to fit a factor model to the user-component matrix. This method focuses on filtering the user-component QoS value matrix using low-rank approximation. In other words, we factorize the QoS matrix into two low-rank matrices V and H. The idea behind the factor model is to derive a high-quality low-dimensional feature representation of users and components based on analyzing the user-component matrix. The premise behind a low-dimensional factor model is that there is only a small number of factors influencing QoS usage experiences, and that a user's QoS usage experience vector is determined by how each factor applies to that user and the items.

Consider the matrix $W \in \mathbb{R}^{m \times n}$ consisting of m users and n components. Let $V \in \mathbb{R}^{l \times m}$ and $H \in \mathbb{R}^{l \times n}$ be the latent user and component feature matrices. Each column in V represents the l-dimensional user-specific latent feature vector of a user and each column in H represents the l-dimensional component-specific latent feature vector of a component. We employ an approximating matrix $\tilde{W} = V^T H$ to fit the user-item matrix W:

$$w_{ij} \approx \tilde{w}_{ij} = \sum_{k=1}^{l} v_{ki} h_{kj}, \tag{3.1}$$

The rank l of the factorization is generally chosen so that $(m+n)l < mn$, since V and H are low-rank feature representations [63]. The product $V^T H$ can be regarded as a compressed form of the data in W.

Note that the low-dimensional matrices V and H are unknown and need to be learned from the obtained QoS values in user-component matrix W. In order to optimize the matrix factorization, we first construct a cost function to evaluate the quality of approximation. The distance between two non-negative matrices is usually employed to

define the cost function. One useful measure of the matrices' distance is the Euclidean distance:

$$F(W, \tilde{W}) = \|W - \tilde{W}\|_F^2 = \sum_{ij} (w_{ij} - \tilde{w}_{ij})^2, \tag{3.2}$$

where $\|\cdot\|_F^2$ denotes the Frobenius norm.

In this chapter, we conduct matrix factorization as solving an optimization problem by employing the optimized objective function in [63]:

$$\begin{aligned} \min_{V,H} \quad & f(V, H) = \sum_{(i,j)\in\Lambda} [\tilde{w}_{ij} - w_{ij} \log \tilde{w}_{ij}], \\ s.t. \quad & \tilde{w}_{i,j} = \sum_{k=1}^{l} v_{ki} h_{kj}, \\ & V \geq 0, \\ & H \geq 0. \end{aligned} \tag{3.3}$$

where $V, H \geq 0$ is the non-negativity constraints leading to allow only additive combination of features.

In order to minimize the objective function in Eq. (3.3), we apply incremental gradient descent method to find a local minimum of $f(V, H)$, where one gradient step intends to decrease the square of prediction error of only one rating, that is, $\tilde{w}_{ij} - w_{ij} \log \tilde{w}_{ij}$. We update the V and H in the direction opposite of the gradient in each iteration:

$$v_{ij} = v_{ij} \sum_k \frac{w_{ik}}{\tilde{w}_{ik}} h_{jk}, \tag{3.4}$$

$$h_{ij} = h_{ij} \sum_k \frac{w_{ik}}{\tilde{w}_{ik}} v_{jk}, \tag{3.5}$$

$$v_{ij} = \frac{v_{ij}}{\sum_k v_{kj}}, \tag{3.6}$$

$$h_{ij} = \frac{h_{ij}}{\sum_k h_{kj}}. \tag{3.7}$$

Algorithm 1 shows the iterative process for latent feature learning. We first initialize matrices V and H with small random non-negative values. Iteration of the above update rules converges to a local minimum of the objective function given in Eq. (3.3).

Algorithm 1: Latent Features Learning Algorithm

Input: W, l
Output: V, H
Initialize $V \in \mathbb{R}^{l \times m}$ and $H \in \mathbb{R}^{l \times n}$ with small random numbers;
repeat
  for *all* $(i, j) \in \Lambda$ **do**
    $\tilde{w}_{ij} = \sum_k v_{ki} h_{kj}$;
  end
  for *all* $(i, j) \in \Lambda$ **do**
    $v_{ij} \leftarrow v_{ij} \sum_k \frac{w_{ik}}{\tilde{w}_{ik}} h_{jk}$;
    $h_{ij} \leftarrow h_{ij} \sum_k \frac{w_{ik}}{\tilde{w}_{ik}} v_{jk}$;
    $v_{ij} = \frac{v_{ij}}{\sum_k v_{kj}}$;
    $h_{ij} = \frac{h_{ij}}{\sum_k h_{kj}}$;
  end
  for *all* $(i, j) \in \Lambda$ **do**
    $\tilde{w}_{ij} = \sum_k v_{ki} h_{kj}$;
  end
until *Converge* ;

3.3.3 Similarity Computation

Given the latent user and component feature matrices V and H, we can calculate the neighborhood similarities between different users and components by employing Pearson Correlation Coefficient (PCC) [85]. PCC is widely used in memory-based recommendation systems for similarity computation. Due to the high accuracy, we adopt PCC in this chapter for the neighborhood similarity computation on both sets of users and components. The similarity between two users u_i and u_j is defined by performing PCC computation on their l-dimensional latent feature vectors V_i and V_j with the follow-

ing equation:

$$S(u_i, u_j) = \frac{\sum_{k=1}^{l}(v_{ik} - \overline{v}_i)(v_{jk} - \overline{v}_j)}{\sqrt{\sum_{k=1}^{l}(v_{ik} - \overline{v}_i)^2}\sqrt{\sum_{k=1}^{l}(v_{jk} - \overline{v}_j)^2}}, \tag{3.8}$$

where $v_i = (v_{i1}, v_{i2}, \cdots, v_{il})$ is the latent feature vector of user u_i and v_{ik} is the weight on the k^{th} feature. $\overline{v}_i$ is the average weight on l-dimensional latent features for user u_i. The similarity between two users $S(i, j)$ falls into the interval $[-1, 1]$, where a larger value indicates higher similarity.

Similar to the user similarity computation, we also employ PCC to compute the similarity between component c_i and item c_j as following:

$$S(c_i, c_j) = \frac{\sum_{k=1}^{l}(h_{ik} - \overline{h}_i)(h_{jk} - \overline{h}_j)}{\sqrt{\sum_{k=1}^{l}(h_{ik} - \overline{h}_i)^2}\sqrt{\sum_{k=1}^{l}(h_{jk} - \overline{h}_j)^2}}, \tag{3.9}$$

where $h_i = (h_{i1}, h_{i2}, \cdots, h_{il})$ is the latent feature vector of component c_i and h_{ik} is the weights on the k^{th} feature. $\overline{h}_i$ is the average weight on l-dimensional latent features for component c_i.

3.3.4 Missing QoS Value Prediction

After computing the similarities between users, we can identify similar neighbors to the current user by ordering similarity values. Note that PCC value falls into the interval $[-1, 1]$, where a positive value means similar and a negative value denotes dissimilar. In practice, QoS usage experience of less similar or dissimilar users may greatly decrease the prediction accuracy. In this chapter, we exclude those users with negative PCC values from the similar neighbor set and only employ the QoS usage experiences of users with Top-K largest PCC values for predicting QoS value of the current user. We refer to the set of Top-K similar users for user u_i as Ψ_i, which is defined as:

$$\Psi_i = \{u_k | S(u_i, u_k) > 0, rank_i(k) \leq K, k \neq i\}, \tag{3.10}$$

where $rank_i(k)$ is the ranking position of user u_k in the similarity list of user u_i, and K denotes the size of set Ψ_i.

Similarly, a set of Top-K similar components for component c_j can be denote as Φ_j by:

$$\Phi_j = \{c_k | S(c_j, c_k) > 0, rank_p(k) \leq K, k \neq j\}, \tag{3.11}$$

where $rank_j(k)$ is the ranking position of component c_k in the similarity list of component c_j, and K denotes the size of set Φ_j.

To predict the missing entry w_{ij} in the user-component matrix, user-based approaches employ the values of entries from Top-K similar users as follows:

$$w_{ij} = \overline{w}_i + \sum_{k \in \Psi_i} \frac{S(u_i, u_k)}{\sum_{a \in \Psi_i} S(u_i, u_a)} (w_{kj} - \overline{w}_k), \tag{3.12}$$

where $\overline{w}_i$ and $\overline{w}_k$ are the average observed QoS values of different components by users u_i and u_k respectively.

For component-based approaches, entry values of Top-K similar components are employed for predicting the missing entry w_{ij} in the similar way:

$$w_{ij} = \overline{w}_j + \sum_{k \in \Phi_j} \frac{S(i_j, i_k)}{\sum_{a \in \Phi_j} S(i_j, i_a)} (w_{ik} - \overline{w}_k), \tag{3.13}$$

where $\overline{w}_j$ and $\overline{w}_k$ are the average available QoS values of component c_j and c_k by different users respectively.

In user-component-based approaches, the predicted values in Eq. (3.12) and Eq. (3.13) are both employed for more precise prediction in the following equation:

$$w^*_{ij} = \lambda \times w^u_{ij} + (1 - \lambda) \times w^c_{ij}, \tag{3.14}$$

where w^u_{ij} denotes the predicted value by user-based approach and w^c_{ij} denotes the predicted value by component-based approach. The parameter λ controls how much the hybrid prediction results rely

on user-based approach or component-based approach. The proper value of λ can be trained on a small sample dataset extracted from the original one. We summarize the proposed algorithm in Algorithm 2.

Algorithm 2: CloudPred Prediction Algorithm

Input: W, l, λ
Output: W^*

Learn V and H by applying Algorithm 1 on W;
for *all* $(u_i, u_j) \in U \times U$ **do**
  calculate the similarity $S(u_i, u_j)$ by Eq. (3.8);
end
for *all* $(c_i, c_j) \in C \times C$ **do**
  calculate the similarity $S(c_i, c_j)$ by Eq. (3.9);
end
for *all* $(i, j) \in \Lambda$ **do**
  construct similar user set Ψ_i by Eq. (3.10);
  construct similar component set Φ_j by Eq. (3.11);
end
for *all* $(i, j) \in \Omega - \Lambda$ **do**
  calculate w_{ij}^u by Eq. (3.12);
  calculate w_{ij}^i by Eq. (3.13);
  $w_{ij}^* = \lambda \times w_{ij}^u + (1 - \lambda) \times w_{ij}^c$;
end

3.4 Experiments

In this section, in order to show the prediction quality improvements of our proposed approach, we conduct several experiments to compare our approach with several state-of-the-art collaborative filtering prediction methods.

In the following, Section 3.4.1 gives the description of our experimental dataset, Section 3.4.2 defines the evaluation metrics, Section 3.4.3 compares the prediction quality of our approach with some other methods, and Section 3.4.4, Section 3.4.5, and Section 3.4.6

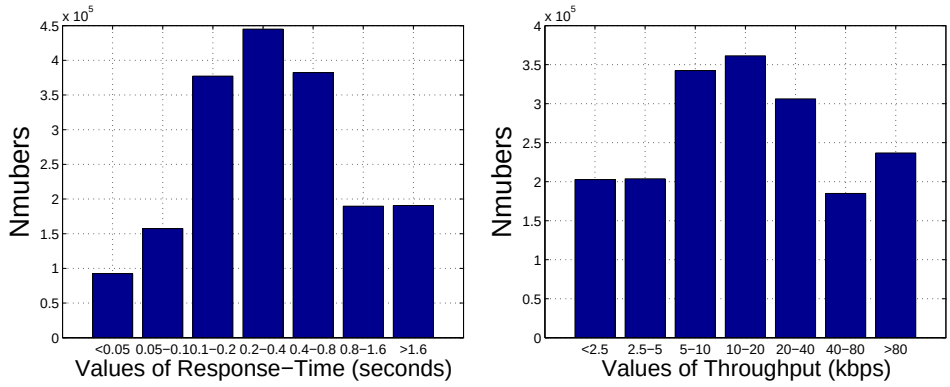
study the impact of training data density, Top-K, and dimensionality, respectively.

3.4.1 Dataset Description

In real world, invoking thousands of commercial cloud components for large-scale experiments is very expensive. In order to evaluate the prediction quality of our proposed approach, we conduct experiments on our Web service QoS dataset [129]. Web service, a kind of cloud component, can be integrated into cloud applications for accessing information or computing service from a remote system. The Web service QoS dataset includes QoS performance of 5,825 openly-accessible real-world Web services from 73 countries. The QoS values are observed by 339 distributed computers located in 30 countries from PlanetLab[4], which is a distributed test-bed consisting of hundreds of computers all over the world. In our experiment, each of the 339 computers keeps invocation records of all the 5,825 Web services by sending null operating requests to capture the characteristics of communication links. Totally 1,974,675 QoS performance results are collected. Each invocation record is a k dimension vector representing the QoS values of k criteria. We then extract a set of 339×5825 user-component matrices, each of which stands for a particular QoS property, from the QoS invocation records. For simplicity, we use two matrices, which represent response-time and throughput QoS criteria respectively, for experimental evaluation in this chapter. Without loss of generality, our approach can be easily extended to include more QoS criteria.

The statistics of Web service QoS dataset are summarized in Table 3.1. Response-time and throughput are within the range 0-20 seconds and 0-1000 kbps respectively. The means of response-time and throughput are 0.910 seconds and 47.386 kbps respectively. Figure 3.3 shows the distributions of response-time and throughput.

[4]http://www.planet-lab.org



(a) (b)

Figure 3.3: Value Distributions

Table 3.1: Statistics of WS QoS Dataset

Statistics	Response-Time	Throughput
Scale	0-20s	0-1000kbps
Mean	0.910s	47.386kbps
Num. of Users	339	339
Num. of Web Services	5,828	5,828
Num. of Records	1,974,675	1,974,675

Most of the response-time values are between 0.1-0.8 seconds and most of the throughput values are between 5-40 kbps.

3.4.2 Metrics

We assess the prediction quality of our proposed approach in comparison with other methods by computing Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). The metric MAE is defined as:

$$MAE = \frac{\sum_{i,j} |w_{ij} - w_{ij}^*|}{N}, \tag{3.15}$$

and RMSE is defined as:

$$RMSE = \sqrt{\frac{\sum_{i,j} (w_{ij} - w_{ij}^*)^2}{N}}, \tag{3.16}$$

where w_{ij} is the QoS value of Web service c_j observed by user u_i, w^*_{ij} denotes the QoS value of Web service c_j would be observed by user u_i as predicted by a method, and N is the number of predicted QoS values.

3.4.3 Performance Comparison

In this section, we compare the prediction accuracy of our proposed approach CloudPred with some state-of-the-art approaches:

1. UPCC (User-based collaborative filtering method using Pearson Correlation Coefficient): this method employs PCC to calculate similarities between users and predicts QoS value based on similar users [11, 96].

2. IPCC (Item-based collaborative filtering method using Pearson Correlation Coefficient): this method employs PCC to calculate similarities between Web services and predicts QoS value based on similar items (item refers to component in this chapter) [85].

3. UIPCC (User-item-based collaborative filtering method using Pearson Correlation Coefficient): this method is proposed by Ma et al. in [74]. It combines UPCC and IPCC approaches and predicts QoS value based on both similar users and similar Web services.

4. NMF (Non-negative Matrix Factorization): This method is proposed by Lee and Seung in [63]. It applies non-negative matrix factorization on user-item matrix for missing value prediction.

In this chapter, in order to evaluate the performance of different approaches in reality, we randomly remove some entries from the matrices and compare the values predicted by a method with the original ones. The matrices with missing values are in different sparsity. For example, 10% means that we randomly remove 90%

entries from the original matrix and use the remaining 10% entries to predict the removed entries. The prediction accuracy is evaluated using Eq.(3.15) and Eq.(3.16) by comparing the original value and the predicted value of each removed entry. Our proposed approach CloudPred performs matrix factorization in Section 3.3.2 and employs both similar users and similar Web services for predicting the removed entries. The parameter settings of our approach CloudPred are Top-K=10, dimensionality=20, $\lambda = 0.5$ in the experiments. Detailed impact of parameters will be studied in Section 3.4.4, Section 3.4.5 and Section 3.4.6.

The experimental results are shown in Table 3.2. For each row in the table, we highlight the best performer among all methods. From Table 3.2, we can observe that our approach CloudPred obtains better prediction accuracy (smaller MAE and RMSE values) than other methods for both response-time and throughput under different matrix densities. The MAE and RMSE values of dense matrices (e.g., matrix density is 80% or 90%) are smaller than those of sparse matrices (e.g., matrix density is 10% or 20%), since a denser matrix provides more information for predicting the missing values. In general, the MAE and RMSE values of throughput are larger than those of response-time because the scale of throughput is 0-1000 kbps, while the scale of response-time is 0-20 seconds. Compared with other methods, the improvements of our approach CloudPred are significant, which demonstrates that the idea of combining global and local information for QoS prediction is realistic and reasonable.

3.4.4 Impact of Matrix Density

In Figure 3.4, we compare the prediction accuracy of all the methods under different matrix densities. We change the matrix density from 10% to 90% with a step value of 10%. The parameter settings in this experiment are Top-K=10, dimensionality=20, and $\lambda = 0.5$.

Figure 3.4(a) and Figure 3.4(b) show the experimental results of

Table 3.2: Performance Comparsions (A Smaller MAE or RMSE Value Means a Better Performance)

Matrix Density	**Metrics**	**Response-Time (seconds)**				
		IPCC	**UPCC**	**UIPCC**	**NMF**	**CloudPred**
10%	MAE	0.7596	0.5655	0.5654	0.6754	**0.5306**
	RMSE	1.6133	1.3326	1.3309	1.5354	**1.2904**
20%	MAE	0.7624	0.5516	0.5053	0.6771	**0.4745**
	RMSE	1.6257	1.3114	1.2486	1.5241	**1.1973**
80%	MAE	0.6703	0.4442	0.3873	0.3740	**0.3704**
	RMSE	1.4102	1.1514	1.0785	1.1242	**1.0597**
90%	MAE	0.6687	0.4331	0.3793	0.3649	**0.3638**
	RMSE	1.4173	1.1264	1.0592	1.1121	**1.0359**

Matrix Density	**Metrics**	**Throughput (kbps)**				
		IPCC	**UPCC**	**UIPCC**	**NMF**	**CloudPred**
10%	MAE	31.6722	26.2015	22.6567	19.7700	**19.0009**
	RMSE	65.5220	61.9658	57.4653	57.3767	**51.8236**
20%	MAE	35.1780	21.9313	18.1230	15.7794	**15.4203**
	RMSE	66.6028	56.5441	50.0435	50.1402	**44.8975**
80%	MAE	29.9146	14.5497	12.4880	12.5107	**10.7881**
	RMSE	64.3079	44.3738	39.6017	39.2029	**36.8506**
90%	MAE	29.9404	13.8761	12.0662	11.6960	**10.4722**
	RMSE	63.7149	42.5534	38.0763	36.7555	**35.9225**

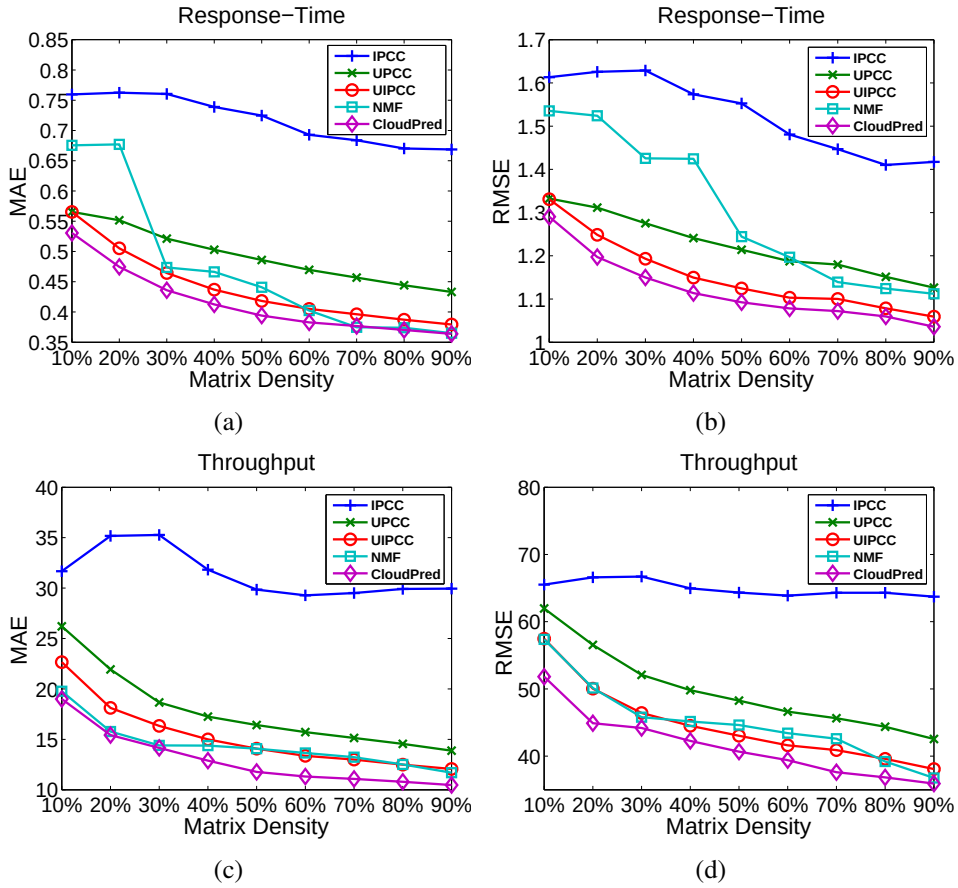


Figure 3.4: Impact of Matrix Density

response-time, while Figure 3.4(c) and Figure 3.4(d) show the experimental results of throughput. The experimental results show that our approach CloudPred achieves higher prediction accuracy than other competing methods under different matrix density. In general, when the matrix density is increased from 10% to 30%, the prediction accuracy of our approach CloudPred is significantly enhanced. When the matrix density is further increased from 30% to 90%, the enhancement of prediction accuracy is quite limited. This observation indicates that when the matrix is very sparse, collecting more QoS information will greatly enhance the prediction accuracy, which further demonstrates that sharing local QoS information among cloud component users could effectively provide personal-

ized QoS estimation.

In the experimental results, we observe that the performance of IPCC is much worse than that of other methods. The reason is that in our Web service dataset the number of users, which is 339, is much smaller than the number of components, which is 5258. When some entries are removed from the user-component matrices, the number of common users between two components, on average, are very small, which would greatly impact the accuracy of common user based similarity computation between components. Therefore, the prediction accuracy of similar items based method IPCC is greatly decreased by the inaccuracy similarity computation between components.

3.4.5 Impact of Top-K

The parameter Top-K determines the size of similar user and similar component sets. In Figure 3.5, we study the impact of parameter Top-K by varying the values of Top-K from 10 to 50 with a step value of 10. Other parameter settings are dimensionality=10 and $\lambda = 0.5$.

Figure 3.5(a) and Figure 3.5(b) show the MAE and RMSE results of response-time respectively, while Figure 3.5(c) and Figure 3.5(d) show the MAE and RMSE results of throughput respectively. The experimental results show that our approach CloudPred achieves best prediction accuracy(smallest MAE and RMSE values) when Top-K is set around 10. Under both sparse matrix, whose density is 10%, and dense matrix, whose density is 90%, all the prediction accuracies decreases when we decrease the Top-K value from 10 to 2 or increase from 10 to 18. This is because too small Top-K value will exclude useful information from some similar users and similar components, while too large Top-K value will introduce noise from dissimilar users and dissimilar components, which will impact the prediction accuracy.

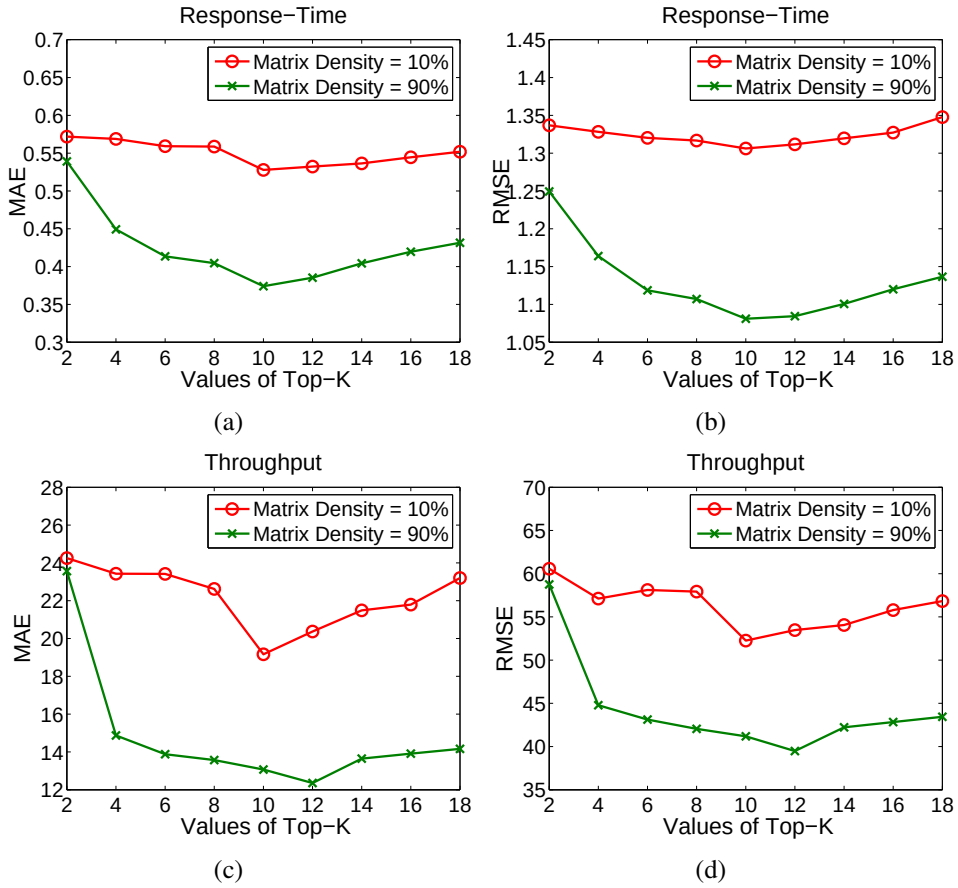


Figure 3.5: Impact of Top-K

3.4.6 Impact of Dimensionality

The parameter dimensionality determines the number of latent features used to characterize user and cloud component. In Figure 3.6, we study the impact of parameter dimensionality by varying the values of dimensionality from 10 to 50 with a step value of 10. Other parameter settings are Top-K=10, and $\lambda = 0.5$.

Figure 3.6(e) and Figure 3.6(f) show the MAE and RMSE values of response-time, while Figure 3.6(g) and Figure 3.6(h) show the MAE and RMSE values of throughput. When the matrix density is 90%, we observe that our approach CloudPred achieves the best performance when the value of dimensionality is 30, while smaller values like 10 or larger values like 50 can potentially hurt the prediction

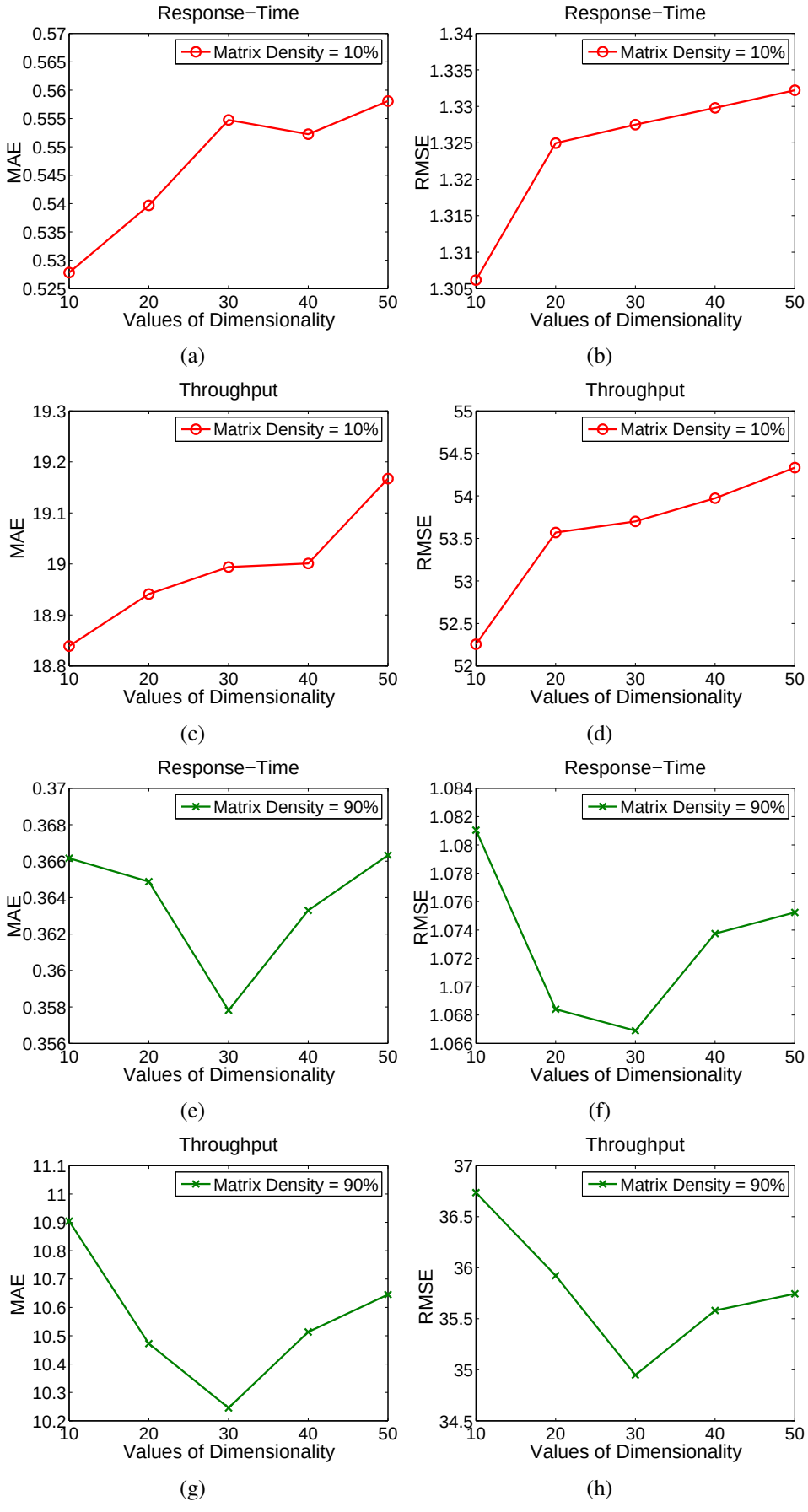
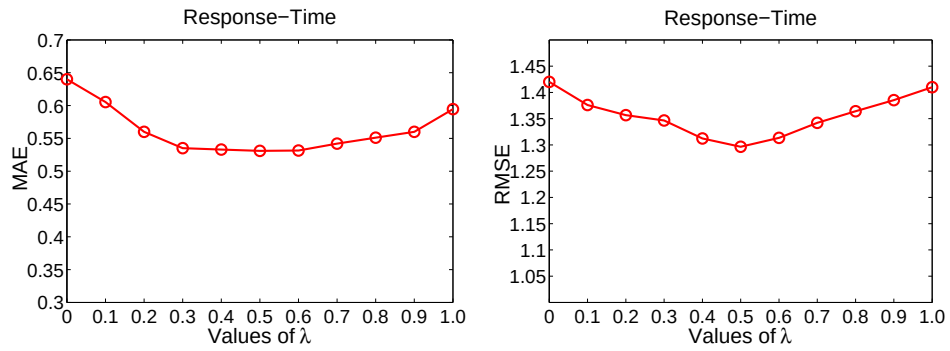


Figure 3.6: Impact of Dimensionality



(a) (b)

Figure 3.7: Impact of λ

accuracy. This observation indicates that when the user-component matrices are dense, 10 latent factors are not enough to characterize the features of user and component which are mined from the rich QoS information, while 50 latent factors are too many since it will cause overfitting problem. When the matrix density is 10%, we observed that the prediction accuracy of our approach CloudPred decreases (MAE and RMSE increase) when the value of dimensionality is increased from 10 to 50. This observation indicates that when the user-component matrices are sparse, 10 latent factors are already enough to characterize the features of user and component which are mined from the limited user-component QoS information, while other larger values of dimensionality will cause the overfitting problem.

3.4.7 Impact of λ

The parameter λ determines how much the final prediction results rely on user-based approach or component-based approach. A larger value of λ means user-based approach contributes more to the hybrid prediction. A smaller value of λ means component-based approach contributes more to the hybrid prediction. In Figure 3.7, we study the impact of parameter λ by varying the values of λ from 0 to 1 with a step value of 0.1. Other parameter settings are dimensionality=10

and Top-K=10.

Figure 3.7(a) and Figure 3.7(b) show the MAE and RMSE results of response-time respectively. The experimental results show that the value of λ impacts the recommendation results significantly, which demonstrates that hybrid the user-based approach and component-based approach improves the recommendation accuracy. The prediction accuracies increases when we increase the value of λ at first. But when λ surpasses a certain threshold, the prediction accuracy decrease with further increase of the value of λ. This phenomenon coincides with the intuition that purely using the user-based approach or purely using the component-based approach cannot generate better results than hybrid these two approaches. From figure 3.7, we observed that when $\lambda \in [0.4, 0.7]$, CloudPred achieves the best performance, while a smaller value or a larger value can potentially degrade the prediction performance. Moreover, the insensitivity of the optimal value of λ shows that the parameter of CloudPred is easy the train.

3.5 Summary

Based on the intuition that a user's cloud component QoS usage experiences can be predicted by exploring the past usage experience from both the user and its similar users, we propose a novel neighborhood-based approach, which is enhanced by feature modeling on both user and component, called CloudPred, for collaborative and personalized QoS value prediction on cloud components. Requiring no additional invocation of cloud components, CloudPred makes the QoS value prediction by taking advantage of both local usage information from similar users and similar components and global invocation information shared by all the users. The extensive experimental results show that our approach CloudPred achieves higher prediction accuracy than other competing methods.

Since the Internet environment is highly dynamic, the QoS per-

formances of a cloud component may be variable against time (e.g., due to the network traffic, server workload, etc.). In our current approach, the QoS values are observed over a long period, which represent the average QoS performance of cloud components. Since the average QoS performance of cloud components is relatively stable, the predicted QoS values provide valuable information of unused cloud components for the users. In our future work, we will explore an online prediction algorithm to handle the dynamically changing QoS values by fusing with the time information.

Currently, we are collecting QoS information of Web service, which is a kind of cloud component. In the future, we will conduct more experiments to evaluate our approach in commercial clouds which contain multiple kinds of cloud components. For future work, we will investigate more techniques for improving the similarity computation (e.g., clustering models, latent factor models, data smoothing, etc.). We will also conduct more investigations on the correlations and combinations of different QoS properties.

□ **End of chapter.**

Chapter 4

Time-Aware Model-based QoS Prediction

4.1 Overview

Web services are software systems designed to support interoperable machine-to-machine interaction over a network [50]. With the exponential growth of Web service as a method of communications between heterogeneous systems, Service-Oriented Architecture (SOA) is becoming a popular and major framework for building Web applications in the era of Web 2.0 [78], whereby Web services offered by different providers are discovered and integrated over the Internet. Typically, a service-oriented application consists of multiple Web services interacting with each other in several tiers. How to build high quality service-oriented applications becomes an urgent and crucial research problem.

With the growing number of Web services over the Internet, designers of service-oriented applications can choose from a broad pool of functionally identical or similar Web services when creating applications. Web services are usually deployed in remote servers and accessed by users through Internet connections. The quality of a service-oriented application, therefore, is greatly influenced by the quality of the invoked Web services. To build high-quality service-oriented applications, non-functional Quality-of-Service (QoS) per-

formance of Web services becomes a major concern for application designers when making service selections [32]. However, the QoS performance of Web services observed from the users' perspective is usually quite different from that declared by the service providers in Service Level Agreement (SLA), due to:

- QoS performance of Web services is highly related to invocation time, since the service status (e.g., workload, number of clients, etc.) and the network environment (e.g., congestion, etc.) change over time.
- Service users are typically distributed in different geographical locations. The user-observed QoS performance of Web services is greatly influenced by the Internet connections between users and Web services. Different users may observe quite different QoS performance when invoking the same Web service.

Based on the above analysis, providing time-aware personalized QoS information of Web services is becoming more and more essential for service-oriented application designers to make service selection [32, 115], service composition [3, 4], and automatically late-binding at runtime [13].

In reality, a service user usually only invokes a limited number of Web services in the past and thus only observes QoS values of these invoked Web services. Without sufficient time-aware personalized QoS information, it is difficult for application designers to select optimal Web services at design time and replace low quality Web services with better ones at runtime. In practice, invoking Web services from users' perspectives for evaluation purpose is quite difficult, and includes the following critical drawbacks:

- Executing service invocations to obtain QoS information is too expensive for service users, since service providers may charge for invocations. At the same time, invocations for evaluation purpose consume resources of service users and service providers.

- With the growing number of Web services over the Internet, it is time-consuming to evaluate all the Web services. Moreover, some potentially appropriate Web services may not be discovered by the current user.
- To monitor the QoS performance of Web services continuously, service users need to conduct service invocations periodically, which will introduce a heavy workload to service users.
- Since service users are not experts in service evaluation, it will take a solid effort from service users to evaluate the Web services in-depth. The time-to-market constraints will also limit the amount of resources for service evaluation.

It becomes an urgent task to explore a time-aware personalized prediction approach for efficiently estimating missing QoS information of Web services for different service users. To address this critical challenge, we propose a model-based approach, called WSPred, for time-aware and personalized QoS prediction of Web services. WSPred collects time-aware QoS information from geographically distributed service users, and combines the local information to get a global user-service-time tensor. By performing tensor factorization, user-specific, service-specific and time-specific latent features are extracted from the past QoS experiences of different service users. The unknown QoS values are therefore estimated by analyzing how the user features are applied to the corresponding service features and time features. We collect a large-scale real-world Web service QoS dataset and conduct extensive experiments to compare the QoS prediction accuracy with several other state-of-the-art approaches. The experimental results show the effectiveness and efficiency of our proposed approach WSPred.

In summary, this chapter makes the following contributions:

- We formally identify the critical problem of time-aware Web service QoS prediction and propose a novel collaborative framework to achieve QoS information sharing among service users.

A user-side light-weight middleware is designed for automatically recording and sharing QoS experiences.

- We propose a novel time-aware personalized QoS prediction approach WSPred, which analyzes latent features of user, service and time by performing tensor factorization. We consider WSPred as the first QoS prediction approach which addresses the difference over time in service computing literature.
- We conduct large-scale real-world experiments to study the prediction accuracy and efficiency of our WSPred compared with other state-of-the-art approaches. Moreover, we publicly release our large-scale Web service QoS dataset[1] for future research. To the best of our knowledge, it is the first multi-user QoS dataset with time series information in the Web service literature.

The remainder of this chapter is organized as follows: Section 4.2 describes the collaborative framework for sharing QoS information between service users. Section 4.3 presents our WSPred approach in detail. Section 4.4 introduces the experimental results. Section 4.5 concludes the chapter.

4.2 Collaborative Framework for Web Services

In this section, we present the collaborative framework for QoS prediction of Web services. Figure 4.1 shows the system architecture. Within a service-oriented Web application, several Web services are employed to implement complicated functions. These Web services are connected with each other in multiple tiers. For each tier, an optimal Web service will be selected from a set of functional equivalent service candidates. Typically the Web service candidates are provided by different organizations and are distributed in different

[1] http://www.wsdream.net/

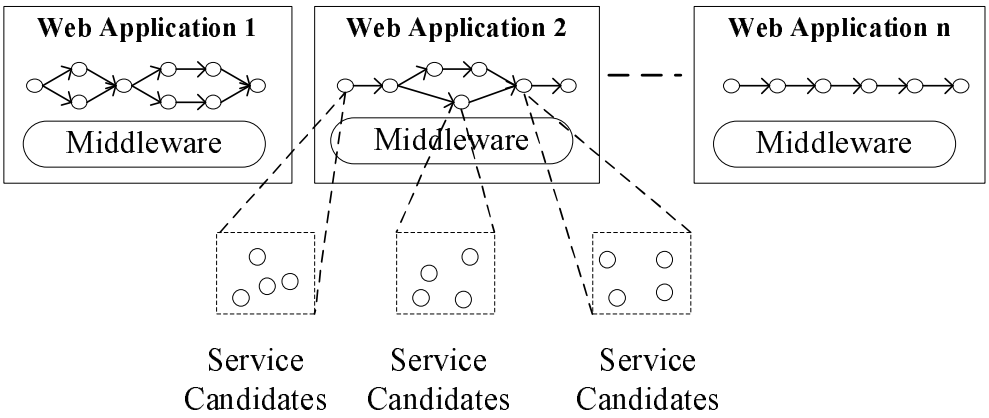


Figure 4.1: System Architecture

geographic locations and time zones. When invoked through communication links, the user-side usage experiences are influenced by the network environments and the server-side status at invocation time.

The mechanism proposed in this chapter is to (1) share local Web service usage experiences from different service users, (2) combine these pieces of local information together to get global QoS information for all service candidates, (3) extract time-specific user features and service features, and (4) make personalized time-aware QoS value prediction based on these features. As shown in Figure 4.2, each service user keeps local records of QoS usage experience on Web services and is encouraged to contribute its local records to obtain records from other users. By contributing more individually observed Web service QoS information, a service user can obtain more global QoS information from other users, thus obtaining more accurate Web service QoS prediction values. Given accurate QoS prediction results, service users could select the potentially optimal services for composing service-oriented Web applications. The detailed collaborative prediction approach will be presented in Section 4.3.

Since most of the service users are not experts in service testing, to reduce the efforts of service users spent on testing the service QoS

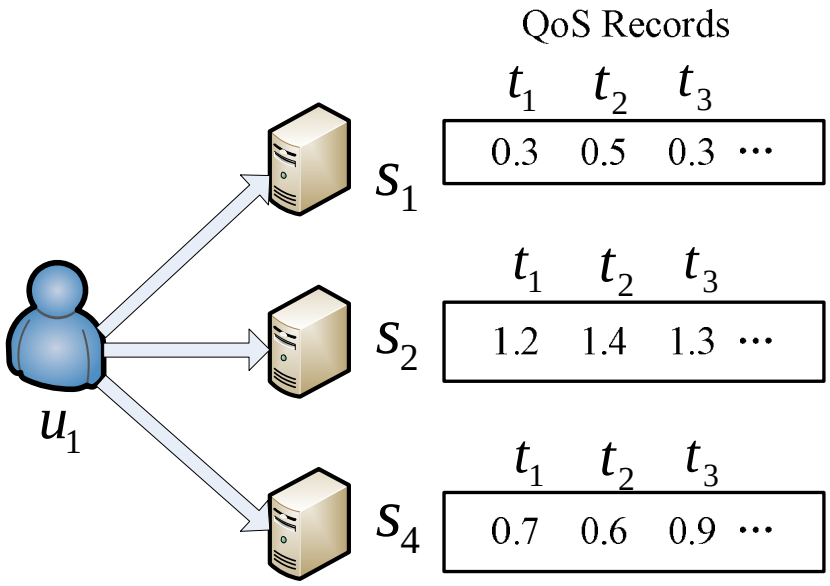


Figure 4.2: A Toy Example

performance, we design a user-side light-weight middleware for service users to automatically record QoS values of invocations and to contribute the local records to the server for obtaining more invocation results from other service users. Within the middleware, there are three management components: *Monitor*, *Collector* and *Predictor*. *Monitor* is responsible for monitoring the QoS performance of Web services when users sends invocations. *Collector* is responsible for contributing local QoS information to other users and for collecting shared QoS information from other users. *Predictor* is responsible for providing time-aware personalized QoS value prediction based on local and other users' QoS information collected by *Collector*.

4.3 Time-Aware QoS Prediction

Previous Web service related techniques such as selection, composition, and orchestration only employ average QoS performance of service candidates at design-time. In recent Web service literatures,

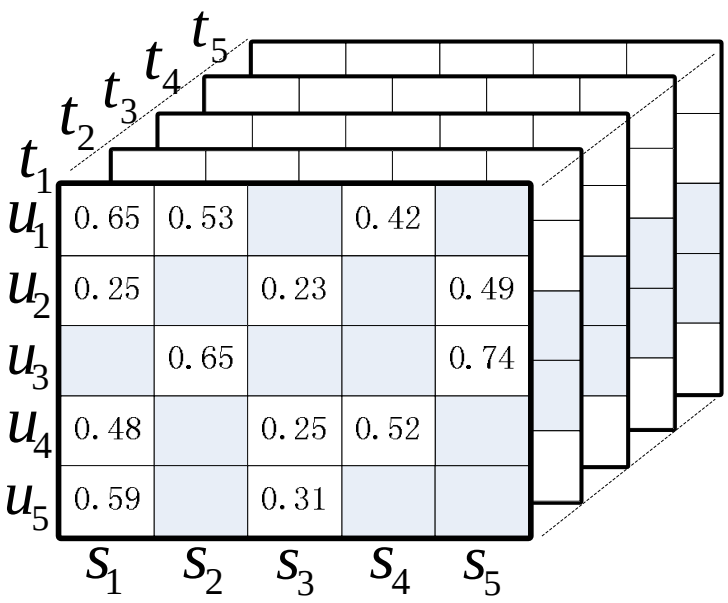


Figure 4.3: User-Service-Time Tensor

most of the state-of-the-art techniques can automatically update corresponding Web services with better ones at run-time, which requires time-specific QoS performance of Web services.

In this section, we first formally describe the QoS value prediction problem on Web services in Section 4.3.1. Then we propose a latent feature learning algorithm to learn the user-specific, service-specific, and time-specific features in Section 4.3.2. The missing QoS values are predicted by applying the proposed algorithm WSPred in Section 4.3.3. Finally, the complexity analysis is conducted in Section 4.3.4.

4.3.1 Problem Description

Figure 4.2 illustrates a toy example of the QoS prediction problem we study in this chapter. In this figure, user u_1 has used three Web services s_1, s_2 and s_4 in the past. u_1 recorded the observed QoS performance of Web services s_1, s_2 and s_4 with specific invocation time in local site. By integrating all the QoS information from other users,

we form a three-dimensional user-service-time tensor as shown in Figure 4.3. In this example, totally there are 5 users (from u_1 to u_5), 5 services (from s_1 to s_5) and 5 time intervals (from t_1 to t_5). The tensor is split into several slices with each one representing a time interval. Within a slice, each entry denotes an observed QoS value of a Web service from a user during the specific time interval. The problem we study in this chapter is how to efficiently and precisely predict the missing entries in the user-service-time tensor based on the existing entries.

Now we formally define the problem of QoS prediction for Web services as follows: Given a set of users and a set of Web services, based on the existing QoS values from different users, predict the missing QoS values of Web services when invoked by users at different time intervals. More precisely:

> *Let U be the set of m users, S be the set of n Web services, and T be the set of c time intervals. A QoS element is a quartet (i, j, k, q_{ijk}) representing the observed quality of Web service s_j by user u_i at time interval t_k, where $i \in \{1, \cdots, m\}$, $j \in \{1, \cdots, n\}$, $k \in \{1, \cdots, c\}$ and $q_{ijk} \in \mathbb{R}^p$ is a p-dimensional vector representing the QoS values of p criteria. Let Ω be the set of all triads $\{i, j, k\}$ and Λ be the set of all known triads (i, j, k) in Ω. Consider a tensor $Y \in \mathbb{R}^{m \times n \times c}$ with each entry Y_{ijk} representing the observed p^{th} criterion value of service s_j by user u_i at time interval t_k. Then the missing entries $\{Y_{ijk} | (i, j, k) \in \Omega - \Lambda\}$ should be predicted based on the existing entries $\{Y_{ijk} | (i, j, k) \in \Lambda\}$.*

Typically, the QoS values can be integers from a given range (e.g., $\{0, 1, 2, 3\}$) or real numbers. Without loss of generality, we can map the QoS values to the interval $[0, 1]$ using the following

function:

$$f(x) = \begin{cases} 0, & \text{if } x < Y_{min} \\ 1, & \text{if } x > Y_{max} \\ \frac{x - Y_{min}}{Y_{max} - Y_{min}}, & \text{otherwise} \end{cases}$$

where Y_{max} and Y_{min} are the specified upper bound and lower bound of QoS values respectively.

4.3.2 Latent Features Learning

In order to learn the latent features of users, services, and time, we employ tensor factorization technique to fit a factor model to the user-service-time tensor. The factorized user-specific, service-specific and time-specific matrices are utilized to make further missing entries prediction. The idea behind the factor model is to derive a high-quality low-dimensional feature representation of users, services and time by analyzing the user-service-time tensor. The premise behind a low-dimensional factor model is that there is only a small number of factors influencing QoS usage experiences, and that a user's QoS usage experience vector is determined by how each factor applies to that user, the corresponding service and the specific time interval. Examples of physical feature are network distance between the user and the server, the workload of the server, etc. Latent features are orthogonal representing the decomposed results of physical factors.

In the chapter, we consider an $m \times n \times c$ QoS tensor consisting of m users, n services and c time intervals. A low-rank tensor factorization approach seeks to approximate the QoS tensor Y by a multiplication of l-rank factors [84],

$$Y \approx C \times_u U \times_s S \times_t T, \tag{4.1}$$

where $C \in \mathbb{R}^{l \times l \times l}$, $U \in \mathbb{R}^{m \times l}$, $S \in \mathbb{R}^{n \times l}$ and $T \in \mathbb{R}^{c \times l}$ are latent feature matrices. l is the number of latent features. Each column in U, S and T representing a user, a Web service and a time interval,

respectively. $\times_u$, $\times_s$ and $\times_t$ are tensor-matrix multiplication operators with the subscript showing in which direction on the tensor to multiply the matrix (i.e., $C \times_u U = \sum_{i=1}^{l} C_{ijk} U_{ij}$). C is set to the diagonal tensor:

$$C = \begin{cases} 1, & \text{if } i = j = k \\ 0, & \text{otherwise} \end{cases}$$

Typically, $l \ll mnc$ since in the real world, each user has invoked only a small portion of Web services, and the tensor Y is usually very sparse. From the above definition, we can see that the low-dimensional matrices U, S and T are unkonwn and need to be estimated.

To estimate the quality of tensor approximation, we need to construct a loss function for evaluating the error between the estimated tensor and the original tensor. The distance between two tensors is usually employed to define the loss function:

$$\frac{1}{2}||Y - \hat{Y}||_F^2, \tag{4.2}$$

where $||\cdot||_F^2$ denotes the Frobenius norm. However, due to the reason that there are a large number of missing values, we only factorize the observed entries in tensor Y. Hence, we employ the following loss function instead:

$$\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{c} I_{ijk} (Y_{ijk} - \hat{Y}_{ijk})^2, \tag{4.3}$$

where I_{ijk} is the indicator function that is equal to 1 if user u_i invoked service s_j during the time interval t_k and equal to 0 otherwise. To avoid the overfitting problem, we add three regularization terms to Eq. (4.3) to constrain the norms of U, S and T. Hence we conduct the tensor factorization as to solve the following optimization

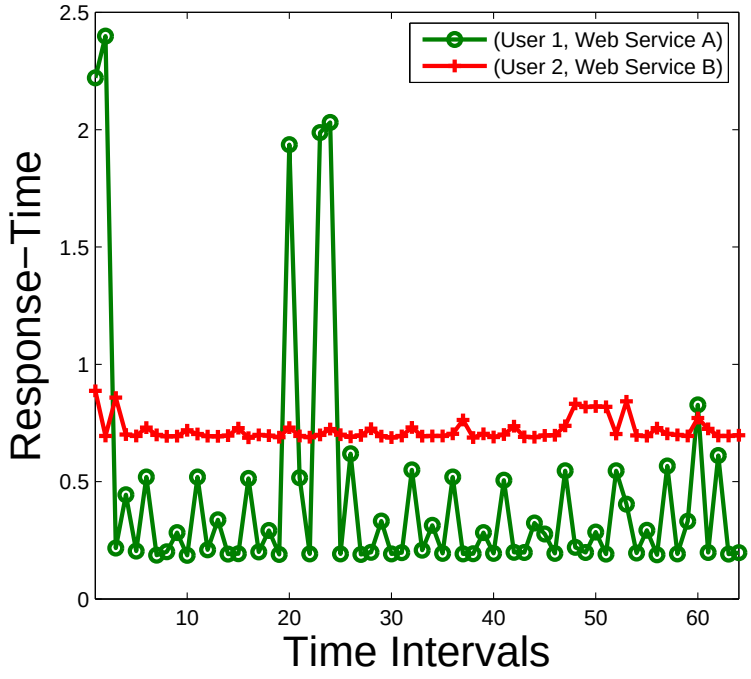


Figure 4.4: Response-Time of Two Pairs of User-Service

problem:

$$\begin{aligned}\min_{U,S,T}\mathcal{L}(Y,U,S,T) &= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} I_{ijk}(Y_{ijk}-\hat{Y}_{ijk})^2 \\ &+ \frac{\lambda_1}{2}||U||_F^2+\frac{\lambda_2}{2}||S||_F^2+\frac{\lambda_3}{2}||T||_F^2,\end{aligned} \tag{4.4}$$

where $\lambda_1, \lambda_2, \lambda_3 > 0$. λ_1, λ_2 and λ_3 defines the importance of regularization terms. In other words, the optimal solution is highly rely on the error we evaluated in the first term. λ_1, λ_2 and λ_3 defines the degree of accuracy in the first term to avoid overfitting problem. The optimization problem in Eq. (4.4) minimizes the sum-of-squared-errors objective function with quadratic regularization terms.

Figure 4.4 gives a comprehensive illustration of the Web service response-time observed by different service users. We randomly se-

lect two service users (User 1 and User 2) and two real-world Web services (Web Service A and Web Service B) from the experiment described in Section 4.4. As shown in Figure 4.4, during different time intervals, a user has different QoS experiences on the same Web service. In general, the differences are limited within a range (e.g., most of the response-time values of (User 1, Web Service A) are within the range of 0.2-0.6s and most of the response-time values of (User 2, Web Service B) are within the range of 0.7-0.9s). This observation indicates that although the QoS values of a particular user-service are different during different time intervals, they fluctuate around the average QoS value of the user-service pair during all time intervals. Based on this observation, we further add a regularization term to Eq. (4.4) to prevent the predicted QoS values from varying a lot against the average QoS value. We define the prediction with average QoS value constraint as the following optimization problem:

$$\begin{aligned}\min_{U,S,T} \mathcal{L}_{\mathcal{A}}(Y,U,S,T) &= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} I_{ijk}(Y_{ijk}-\hat{Y}_{ijk})^2 \\ &+ \frac{\lambda_1}{2}||U||_F^2 + \frac{\lambda_2}{2}||S||_F^2 + \frac{\lambda_3}{2}||T||_F^2 \\ &+ \frac{\eta}{2}\sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{c} I_{ijk}(\hat{Y}_{ijk}-\bar{Y}_{ij})^2,\end{aligned} \tag{4.5}$$

where $\eta > 0$, and $\bar{Y}_{ij}$ denotes the average QoS value of Web service s_j observed by user u_i during all the time intervals. η controls how much the prediction method should engage the information of average QoS performance. In the extreme case, if we use a very small value of η, we only perform tensor factorization without considering the global QoS information. On the other side, if we use a very large value of η, the average QoS performance will dominate the learning processes.

A local minimum of the objective function given by Eq. (4.5) can be found by performing incremental gradient descent in feature vectors U_i, S_j and T_k:

$$\begin{aligned}
\frac{\partial \mathcal{L}_{\mathcal{A}}}{\partial U_{if}} &= \sum_{j=1}^{n}\sum_{k=1}^{c} I_{ijk}(\hat{Y}_{ijk} - Y_{ijk}) S_j^T T_k + \lambda_1 U_{if} \\
&+ \eta \sum_{j=1}^{n}\sum_{k=1}^{c} I_{ijk}(\hat{Y}_{ijk} - \bar{Y}_{ij}) S_j^T T_k, \\
\frac{\partial \mathcal{L}_{\mathcal{A}}}{\partial S_{jf}} &= \sum_{i=1}^{m}\sum_{k=1}^{c} I_{ijk}(\hat{Y}_{ijk} - Y_{ijk}) U_i^T T_k + \lambda_2 S_{jf} \\
&+ \eta \sum_{i=1}^{m}\sum_{k=1}^{c} I_{ijk}(\hat{Y}_{ijk} - \bar{Y}_{ij}) U_i^T T_k, \\
\frac{\partial \mathcal{L}_{\mathcal{A}}}{\partial T_{kf}} &= \sum_{i=1}^{m}\sum_{j=1}^{n} I_{ijk}(\hat{Y}_{ijk} - Y_{ijk}) U_i^T S_j + \lambda_3 T_{kf} \\
&+ \eta \sum_{i=1}^{m}\sum_{j=1}^{n} I_{ijk}(\hat{Y}_{ijk} - \bar{Y}_{ij}) U_i^T S_j. \qquad (4.6)
\end{aligned}$$

Algorithm 3 shows the iterative process for latent feature learning. We first initialize matrices U, S and T with small random non-negative values. Iteration of the update rules derived from Eq. (4.6) converges to a local minimum of the objective function given in Eq. (4.5).

4.3.3 Missing Value Prediction

After the user-specific, service-specific and time-specific latent feature spaces U, S and T are learned, we can predict the QoS performance of a given service observed by a user during the specific time interval. For the missing entry Y_{ijk} in the QoS tensor, the value

Algorithm 3: Latent Features Learning Algorithm

Input: Y, l, λ, η
Output: U, S, T

Initialize $U \in \mathbb{R}^{l \times m}$, $S \in \mathbb{R}^{l \times n}$ and $T \in \mathbb{R}^{l \times c}$ with small random numbers;
repeat
  for *all* $(i, j, k) \in \Lambda$ **do**
    $\hat{Y}_{ijk} = \sum_{f=1}^{l} U_{if} S_{jf} T_{kf}$;
  end
  for *all* (i, j) **do**
    $\bar{Y}_{ij} = \frac{\sum_{k=1}^{c} I_{ijk} Y_{ijk}}{\sum_{k=1}^{c} I_{ijk}}$;
  end
  for *all* $(i, j, k) \in \Lambda$ **do**
    for $(f = 1; f \leq l; f++)$ **do**
      $U_{if} \leftarrow U_{if} - [(\hat{Y}_{ijk} - Y_{ijk}) S_j^T T_k + \lambda U_{if} + \eta(\hat{Y}_{ijk} - \bar{Y}_{ij}) S_j^T T_k]$;
      $S_{jf} \leftarrow S_{jf} - [(\hat{Y}_{ijk} - Y_{ijk}) U_i^T T_k + \lambda S_{jf} + \eta(\hat{Y}_{ijk} - \bar{Y}_{ij}) U_i^T T_k]$;
      $T_{kf} \leftarrow T_{kf} - [(\hat{Y}_{ijk} - Y_{ijk}) U_i^T S_j + \lambda T_{kf} + \eta(\hat{Y}_{ijk} - \bar{Y}_{ij}) U_i^T S_j]$;
    end
  end
until *Converge* ;

predicted by our method is defined as

$$\hat{Y}_{ijk} \;=\; I_{ijk} \sum_{f=1}^{l} U_{if} S_{jf} T_{kf}. \tag{4.7}$$

4.3.4 Complexity Analysis

The main computation of gradient methods is evaluating the objective function $\mathcal{L}_{\mathcal{A}}$ and their gradients against variables. The computational complexity of evaluating the objective function $\mathcal{L}_{\mathcal{A}}$ is $O(\rho_Y l + \rho_Y c)$, where ρ_Y is the number of nonzero entries in the tensor Y, l is the dimensions of the latent features, and c is the number of time intervals. The computational complexities for the gradients $\frac{\partial \mathcal{L}_{\mathcal{A}}}{\partial U}$, $\frac{\partial \mathcal{L}_{\mathcal{A}}}{\partial S}$ and $\frac{\partial \mathcal{L}_{\mathcal{A}}}{\partial T}$ in Eq. (4.6) are $O(\rho_Y l + \rho_Y c)$. Therefore, the total computational complexity in one iteration is $O(\rho_Y l + \rho_Y c)$, which indicates that theoretically, the computational time of our method

is linear with recept to the number of observed QoS values in the user-service-time tensor Y. Note that because of the sparsity of Y, $\rho_Y << mnc$, which indicates that the computation time grows slowly with respect to the size of Tensor Y. This complexity analysis shows that our proposed approach is very efficient and can be applied to large-scale systems.

4.4 Experiments

In this section, we conduct several experiments to compare our approach with several state-of-the-art collaborative filtering prediction methods. In the following, Section 4.4.1 introduces the experimental setup and gives the description of our experimental dataset, Section 4.4.2 defines the evaluation metrics, Section 4.4.3 compares the prediction quality of our approach with other competing methods, and Section 4.4.4 and Section 4.4.5 study the impact of tensor density and dimensionality, respectively.

4.4.1 Experimental Setup and Dataset Collection

To evaluate our proposed QoS prediction approach in the real-world, we implement a tool WSMonitor for monitoring the QoS performance of Web service, and collect a large-scale Web service QoS dataset for conducting various experiments.

WSMonitor is implemented and deployed with JDK 6.0, Eclipse 3.3, Axis 2, and Apache 2.2.17. WSMonitor first crawls a set of WSDL files from the Internet and generates a list of openly-accessible Web services. For each Web service in the list, WSMonitor automatically generates a java class for service invocation by employing the WSDL2Java tool from the Axis package [48]. Totally, 5,871 classes are generated for 5,871 Web services. By calling the functions within a class, null operation requests are sent to the corresponding Web service for capturing the QoS performance.

We deploy the WSMonitor on 142 distributed computers located in 22 countries from PlanetLab[2], which is a distributed test-bed consisting of hundreds of computers all over the world. Totally, 4,532 publicly available real-world Web services from 57 countries are monitored by each computer continuously. 1,339 of the initially selected Web services are excluded in this experiment due to: 1) authentication required and 2) permanent invocation failure (e.g., the Web service is shutdown). In our experiment, each of the 142 computers sends null operation requests to all the 4,532 Web services during every time interval. The experiment lasts for 16 hours with a time interval lasting for 15 minutes.

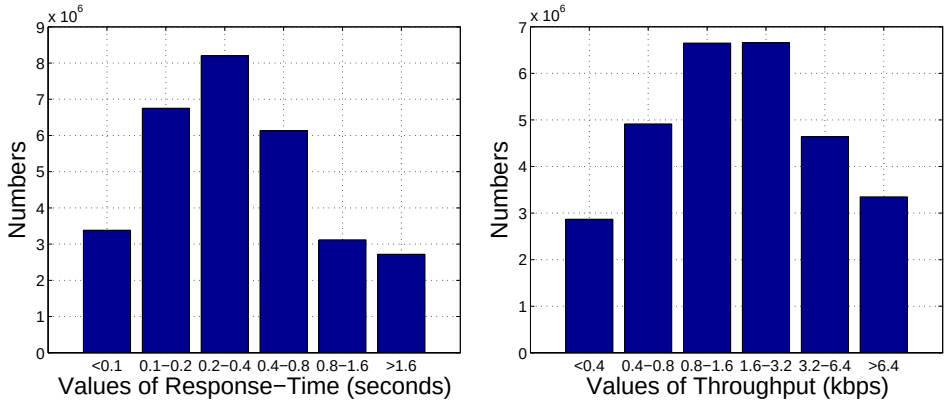
By collecting invocation records from all the computers, finally we include 30,287,611 QoS performance results into the Web service QoS dataset. Each invocation record is a k dimension vector representing the QoS values of k criteria. We then extract a set of $142 \times 4532 \times 64$ user-service-time tensors, each of which stands for a particular QoS property, from the QoS invocation records. For simplicity, we employ two tensors, which represent response-time and throughput QoS criteria respectively, for experimental evaluation in this chapter. Without loss of generality, our approach can be easily extended to include more QoS criteria.

The statistics of Web service QoS dataset are summarized in Table 4.1. Response-time and throughput are within the range of 0-20 seconds and 0-1000 kbps respectively. The means of response-time and throughput are 3.165 seconds and 9.609 kbps respectively. The distributions of the response-time and throughput values of the user-service-time tensors are shown in Figure 4.5(a) and Figure 4.5(b) respectively. Most of the response-time values are between 0.1-0.8 seconds and most of the throughput values are between 0.8-3.2 kbps.

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

Table 4.1: Statistics of WS QoS Dataset

Statistics	Response-Time	Throughput
Scale	0-20s	0-1000kbps
Mean	3.165s	9.609kbps
Num. of Users	142	142
Num. of Web Services	4,532	4,532
Num. of Time Intervals	64	64
Num. of Records	30,287,611	30,287,611



(a) (b)

Figure 4.5: QoS Value Distributions

4.4.2 Metrics

We assess the prediction quality of our proposed approach in comparison with other methods by computing Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). The metric MAE is defined as:

$$MAE = \frac{\sum_{ijk} |\hat{Y}_{ijk} - Y_{ijk}|}{N}, \tag{4.8}$$

and RMSE is defined as:

$$RMSE = \sqrt{\frac{\sum_{ijk} (\hat{Y}_{ijk} - Y_{ijk})^2}{N}}, \tag{4.9}$$

where Y_{ijk} is the QoS value of Web service s_j observed by user u_i at time interval t, $\hat{Y}_{ijk}$ denotes the QoS value of Web service s_j would be observed by user u_i at time interval t_k as predicted by a method, and N is the number of predicted QoS values.

4.4.3 Performance Comparisons

In this section, in order to show the effectiveness of our proposed Web service QoS prediction approach, we compare the prediction accuracy of the following methods:

1. **MF1**-This method considers the user-service-time tensor as a set of user-service matrix slices in terms of time. For each slice, the prediction method proposed by Lee and Seuing in [63] is employed. It applies non-negative matrix factorization on user-item matrix for missing value prediction.

2. **MF2**-This method first compresses the user-service-time tensor into a user-service matrix. For each entry in the matrix, the value is the average of the specific user-service pair during all the time intervals. After obtaining the compressed user-service matrix, it applies the non-negative matrix factorization technique proposed by Lee and Seuing [63] on user-item matrix for missing value prediction.

3. **TF**-This is a tensor factorization-based prediction method. It applies tensor factorization on user-service-time tensor to extract user-specific, service-specific and time-specific characterizes. The missing value is then predicted based on how these characterized apply to each other.

4. **WSPred**-This method is proposed in this chapter. It is a tensor factorization-based recommendation with average QoS value constraints.

Table 4.2: Performance Comparsions (A Smaller MAE or RMSE Value Means a Better Performance)

Tensor Density	**Metrics**	**Response-Time (seconds)**			
		MF1	**MF2**	**TF**	**WSPred**
5%	MAE	3.4137	2.9187	2.9184	**2.5580**
	RMSE	5.3423	5.1024	4.7508	**4.3626**
10%	MAE	2.8518	2.8421	2.7888	**2.4990**
	RMSE	5.0667	4.5563	4.5696	**4.2892**
45%	MAE	2.4241	2.2679	2.2511	**2.1462**
	RMSE	4.3240	4.2541	4.2071	**3.9200**
50%	MAE	2.3959	2.2596	2.2127	**2.1266**
	RMSE	4.2996	4.1490	4.0169	**3.8943**

Tensor Density	**Metrics**	**Throughput (kbps)**			
		MF1	**MF2**	**TF**	**WSPred**
5%	MAE	10.5460	8.8317	8.7997	**8.2761**
	RMSE	46.6735	43.4769	39.5133	**39.0962**
10%	MAE	9.9839	8.7522	8.5080	**8.0131**
	RMSE	46.6656	39.7740	39.2792	**38.6251**
45%	MAE	8.6773	7.9590	7.9471	**6.9398**
	RMSE	45.0077	39.9388	38.6964	**36.5724**
50%	MAE	8.6224	7.8306	7.8045	**6.8558**
	RMSE	44.9407	38.9388	38.6964	**36.5724**

Since memory-based approaches require much more computation time than model-based approaches, we only compare the above four model-based approaches. Since the matrix factorization technique cannot be directly applied to time-aware prediction problem, we extend the prediction approach [63] in two different ways, which derive method MF1 and MF2 respectively.

In order to evaluate the performance of different approaches in reality, we randomly remove some entries from the tensors and compare the values predicted by a method with the original ones. The

tensors with missing values are in different densities. For example, 10% means that we randomly remove 90% entries from the original tensor and use the remaining 10% entries to predict the removed entries. The prediction accuracy is evaluated using Eq. (4.8) and Eq. (4.9) by comparing the original value and the predicted value of each removed entry. The values of λ and η are tuned by performing cross-validation [49] on the observed QoS data. Without lost of generality, the parameter settings of all the approaches are $l = 20$ and $\lambda 1 = \lambda 2 = \lambda 3 = \eta = 0.001$ in the experiments conducted in this chapter. Detailed impact of tensor density and dimensionality is studied in Section 4.4.4 and Section 4.4.5.

The QoS value prediction accuracies evaluated by MAE and RMSE are shown in Table 4.2. For each row in the table, we highlight the best performer among all methods. From Table 4.2, we can observe that the tensor factorization-based prediction methods (i.e., TF and WSPred) outperform the matrix factorization-based prediction methods (i.e., MF1 and MF2), since the tensor factorization-based methods use the time-specific features as additional information. We also observe that our approach WSPred constantly performs better (smaller MAE and RMSE values) than the other approaches, including TF, for both response-time and throughput under both dense tensors and sparse tensors. This demonstrates the advantage of time-aware prediction algorithm with the constraints of average QoS performance. In Table 4.2, the MAE and RMSE values of dense tensors (e.g., tensor density is 45% or 50%) are smaller than those of sparse tensors (e.g., tensor density is 5% or 10%), since a denser tensor provides more information for predicting the missing values. In general, the MAE and RMSE values of throughput are larger than those of response-time because the scale of throughput is 0-1000 kbps, while the scale of response-time is 0-20 seconds. Compared with other methods, the improvements of our approach WSPred are significant, which demonstrates that the idea of considering time information for QoS prediction is realistic and reasonable.

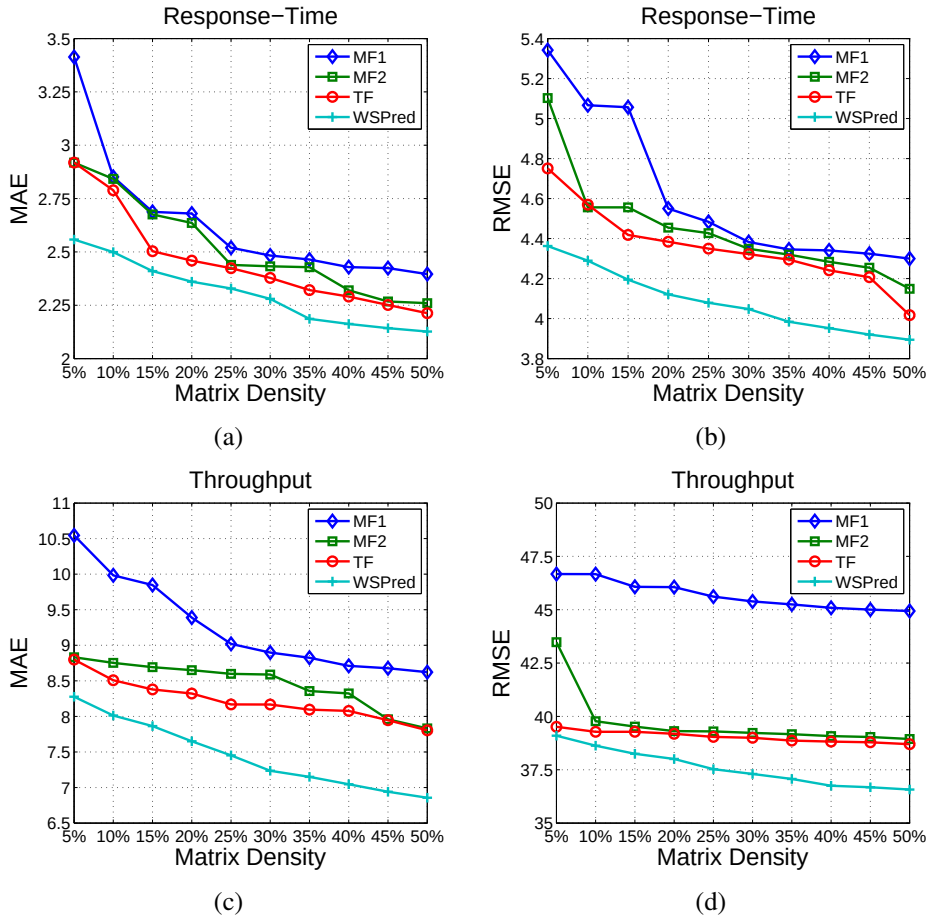


Figure 4.6: Impact of Tensor Density

4.4.4 Impact of Tensor Density

In Figure 4.6, we compare the prediction accuracy of all the methods under different tensor densities. We change the tensor density from 5% to 50% with a step value of 5%. The parameter settings in this experiment are $l = 20$ and $\lambda 1 = \lambda 2 = \lambda 3 = \eta = 0.001$.

Figure 4.6(a) and Figure 4.6(b) show the experimental results of response-time, while Figure 4.6(c) and Figure 4.6(d) show the experimental results of throughput. The experimental results show that our approach WSPred achieves higher prediction accuracy (lower MAE and RMSE values) than other competing methods under different tensor density. In general, when the tensor density is increased

from 5% to 20%, the prediction accuracy of our approach WSPred is significantly enhanced. When the tensor density is further increased from 20% to 50%, the enhancement of prediction accuracy is quite limited. This observation indicates that when the tensor is very sparse, collecting more QoS information will greatly enhance the prediction accuracy, which further demonstrates that considering both the difference between time intervals and the average QoS performance could effectively provide personalized QoS estimation.

In the experimental results, we observe that the performance of MF1 is worse than that of other methods. The reason is that MF1 only extracts the user-specific and service-specific features without considering the relationship between QoS performance in time intervals. In general, MF2 performs better than MF1, since MF2 computes the average QoS performance before performing matrix factorization. Applying the features extracted from the original tensor, MF2 predicts the average QoS performance for a particular user-service pair. This observation further demonstrates that the average QoS performance of a particular user-service pair can provide valuable information when predicting the missing QoS value of the user-service pair in a particular time interval.

4.4.5 Impact of Dimensionality

The parameter dimensionality l determines the number of latent features applied to characterize user, service and time. In Figure 4.7 and Figure 4.8, we study the impact of parameter dimensionality by varying the values of l from 5 to 50 with a step value of 5. Other parameter settings are $\lambda 1 = \lambda 2 = \lambda 3 = \eta = 0.001$.

Figure 4.7 and Figure 4.8 show the MAE and RMSE values of response-time and throughput respectively. We observe that in both figures, as l increases, the MAE and RMSE decrease (prediction accuracy increases), but when l surpasses a certain threshold like 20, the MAE and RMSE increase (prediction accuracy decreases) with

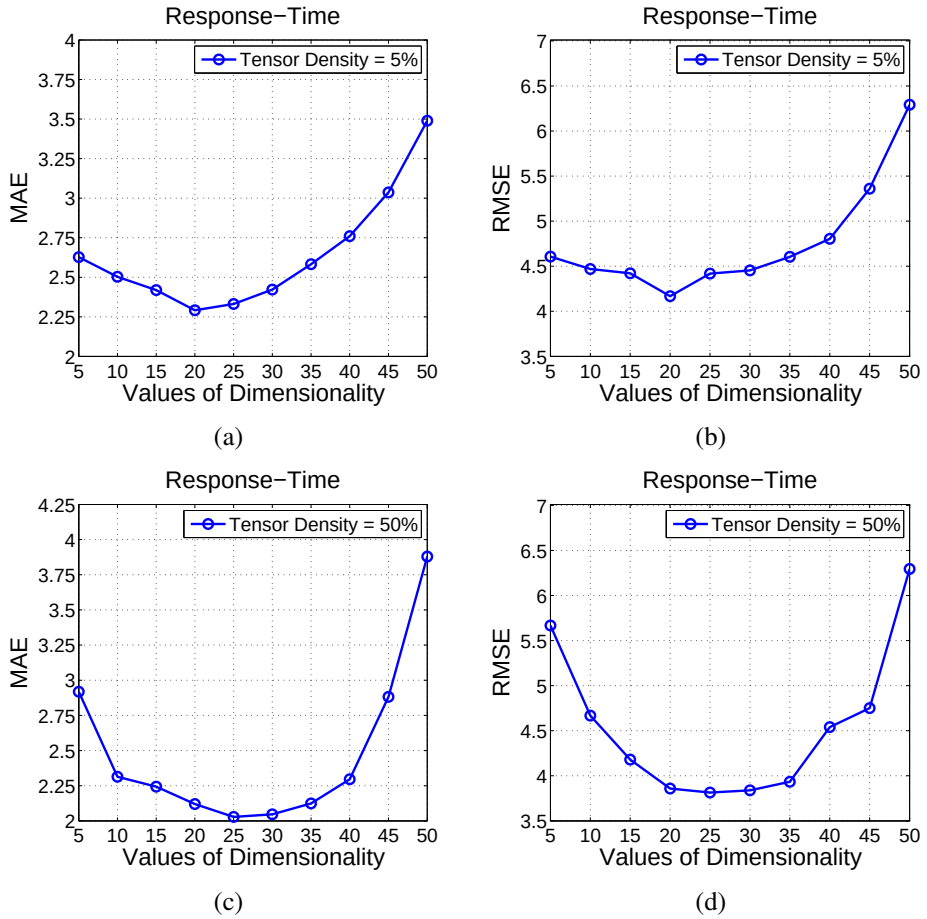


Figure 4.7: Impact of Dimensionality in Response-Time Dataset

further increase of the value of l. This observation indicates that too few latent factors are not enough to characterize the features of user, service and time, while too many latent factors will cause an overfitting problem. There exists an optimal value of l for characterizing the latent features. In both Figure 4.7 and Figure 4.8, when the tensor density is 50%, we observe that our approach WSPred achieves the best performance when the value of dimensionality is 25, while smaller values like 5 or larger values like 50 can potentially reduce the prediction accuracy. When the tensor density is 5%, we observe that the prediction accuracy of our approach WSPred achieves the best performance when the value of dimensionality is

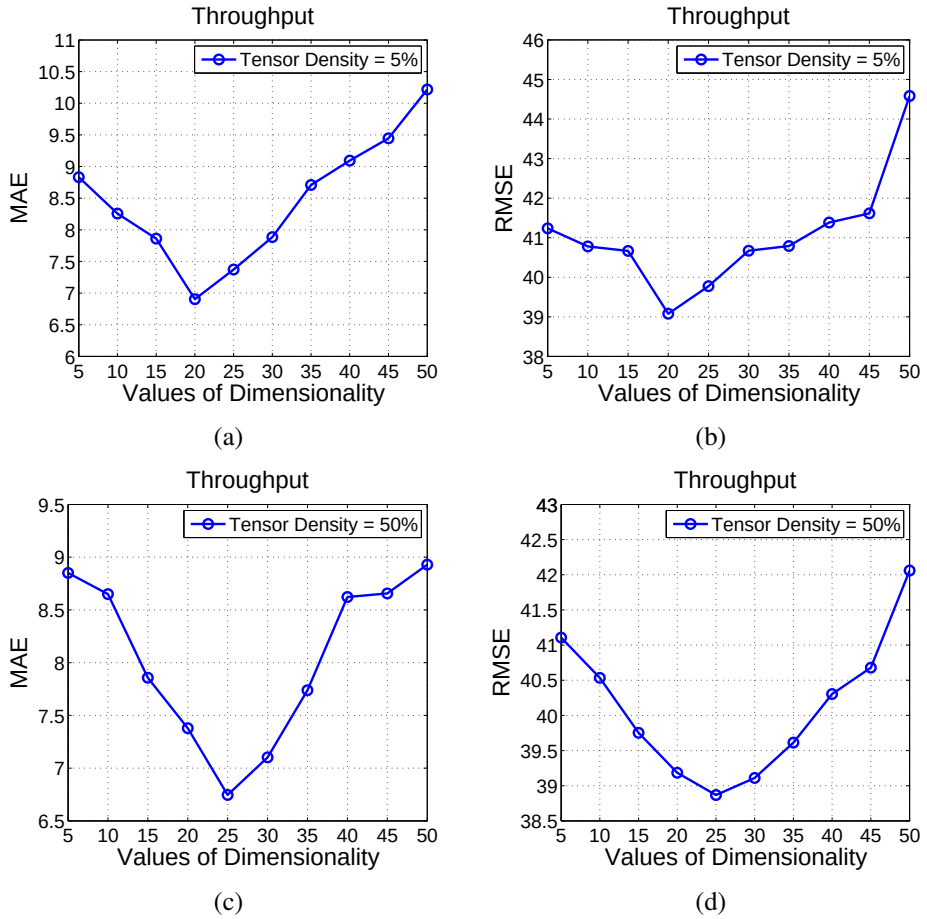


Figure 4.8: Impact of Dimensionality in Throughput Dataset

20, while smaller values like 5 or larger values like 50 can potentially reduce the prediction accuracy. This observation indicates that when the user-service-time tensor is sparse, 20 latent factors are already enough to characterize the features of user, service and time which are mined from the limited user-service-time QoS information. On the other hand, when the tensor is dense, more latent factors, like 25, are needed to characterize the latent features since more QoS information can be obtained from the original tensor.

4.5 Summary

Based on the intuition that a user's Web service QoS usage experience can be predicted by using the past usage experience from different users, we propose a novel model-based approach, called WSPred, for time-aware personalized QoS value prediction for Web services. By employing a collaborative framework, WSPred performs feature modeling on user, Web service and time based on the QoS usage experience collected from both local and global users. Requiring no additional invocation of Web services, WSPred makes the QoS prediction by evaluating how the user-specific, service-specific and time-specific latent features apply to each other. The extensive experimental results show that our proposed WSPred outperforms the state-of-the-art QoS prediction approaches for Web services.

For future work, we will investigate more techniques for improving the prediction accuracy (e.g., data smoothing, utilizing content information, etc.). Currently, we predict the values of different QoS properties independently. In the future, we will also conduct more investigations on the correlations and combinations on the different QoS properties. WSPred predicts missing QoS values based on the past QoS experience and the available QoS information in the current time interval. If no QoS information is available in the current time interval, WSPred purely depends on the past experience. In the future, we will explore an online prediction algorithm to perform time series analysis for prediction and extend WSPred to handle updated QoS information at run-time.

□ **End of chapter.**

Chapter 5

Online QoS Prediction

5.1 Overview

Web services are software systems designed to support interoperable machine-to-machine interaction over a network. With the exponential growth of Web service as a method of communications between heterogeneous systems, Service-Oriented Architecture (SOA) is becoming a major framework for building Web systems in the era of Web 2.0 [78]. In service computing, Web services offered by different providers are discovered and integrated to implement complicated functions. Typically, a service-oriented system consists of multiple Web services interacting with each other over the Internet in an arbitrary way. How to build high-quality service-oriented systems becomes an urgent and crucial research problem.

Low response time is one of the most important requirements of the service-oriented systems, which are widely employed in e-business and e-government. Typically, the response time performance of service-oriented systems involves two parts: local execution time at the system side and the response time of invoking remote Web services. While the local execution time is relatively short, the response time of invoking Web services is usually much longer, which greatly influences the system performance. The reason is that Web services are usually deployed in different geographical locations and invoked via Internet connections. Moreover, the

remote Web services may be running on cheap and poor performing servers, leading to a decrease of service performance. In order to build service-oriented systems with good performance, it is important to identify Web services with low response time for composition. Moreover, by identifying the Web services with long response time at runtime, system designers can replace them with better ones to enhance the overall system performance.

Typically, Web services are considered as black boxes to service users. The user-side observed performance is employed to evaluate the qualities of Web services. Since the service status (e.g., workload, CPU allocations, etc.) and the network environment (e.g., congestions, bandwidth, etc.) may change over time, response time of Web services varies a lot during different time intervals. In order to identify low response time Web services timely, real-time performance of Web services needs to be continuously monitored.

Based on the above analysis, providing real-time performance information of Web services is becoming more and more essential for service-oriented system designers to build high-quality systems and to maintain the performance of the systems at runtime. However, evaluating the performance of service-orientated systems at runtime is not an easy task, due to the following reasons:

- Since users (SOA systems) and services are typically distributed in different geographical locations, the user-observed performance of Web services is greatly influenced by the Internet connections between users and Web services. Different users may observe quite different performance when invoking the same Web service.
- Real-time performance evaluation may introduce extra transaction workload, which may impact the user experience of using the systems.
- The purpose of performance evaluation is to monitor the current system performance status and allow designers to make

adjustments in order to guarantee the performance in the future. This requires frequent performance evaluation, since infrequent evaluation cannot provide useful information to designers for choosing appropriate services in the following time.

It becomes an urgent task to explore an online personalized prediction approach for efficiently estimating the performance of Web services for different service users. Based on the performance information of Web services, the overall performance of a service-oriented system can be estimated by aggregating the performance of services invoked by the system. In this chapter, we propose a service performance estimation framework for providing personalized performance information to the users. The performance of services is predicted by collaborative work of users. We collect time-aware performance information from geographically distributed service users. Due to the fact that a service user usually only invokes a small number of Web services in the past and thus only observes performance of these invoked Web services, the collected performance information is usually sparse. In order to precisely predict the performance of Web service when invoked by users, we employ a set of latent features to characterize the status of Web services and users. Examples of physical feature are network distance between the user and the service server, the workload of the server, etc. Latent features are orthogonal representation of the decomposed results of physical factors. We extract the latent features of users and services in the past time slice from the collected service performance information. By analyzing the trend of the feature changes, we estimate the features of users and services in the current time. Then the personalized performance of Web service is predicted by evaluating how the features of users apply to features of services.

In summary, this chapter makes the following contributions:

- We propose an online performance prediction framework for estimating the user observed performance of service-oriented

systems. Our approach employs the past usage experiences of different users to efficiently predict the performance of service-oriented systems online.

- We collect a large-scale real-world Web service performance dataset and conduct extensive experiments for evaluating the performance of our proposed approach OPred. Totally, 4,532 Web services are monitored by 142 service users and 30,287,611 invocation results are collected. Moreover, we publicly release our large-scale real-world Web service performance dataset for future research.

The rest of this chapter is organized as follows: Section 5.2 describes the service-oriented system architecture and introduces the online performance prediction procedures. Section 5.3 and Section 5.4 present our online service performance prediction approach OPred in detail. Section 5.5 presents the experimental results. Section 5.6 concludes the chapter.

5.2 Preliminaries

Figure 5.1 shows the architecture of a typical service-oriented system. Within a service-oriented system, several abstract tasks are combined to implement complicated functions. For each abstract task, an optimal Web service is selected from a set of functionally equivalent service candidates. By composing the selected services, a service-oriented system instance is implemented for task execution. The problem of finding functionally equivalent Web service candidates has been discussed by a lot of previous work [91, 116], which is outside the scope of this work. Typically the Web service candidates are provided by different organizations and distributed in different geographical locations and time zones. When invoked through communication links, the user-side usage experiences are influenced by the network environments and the server-side status

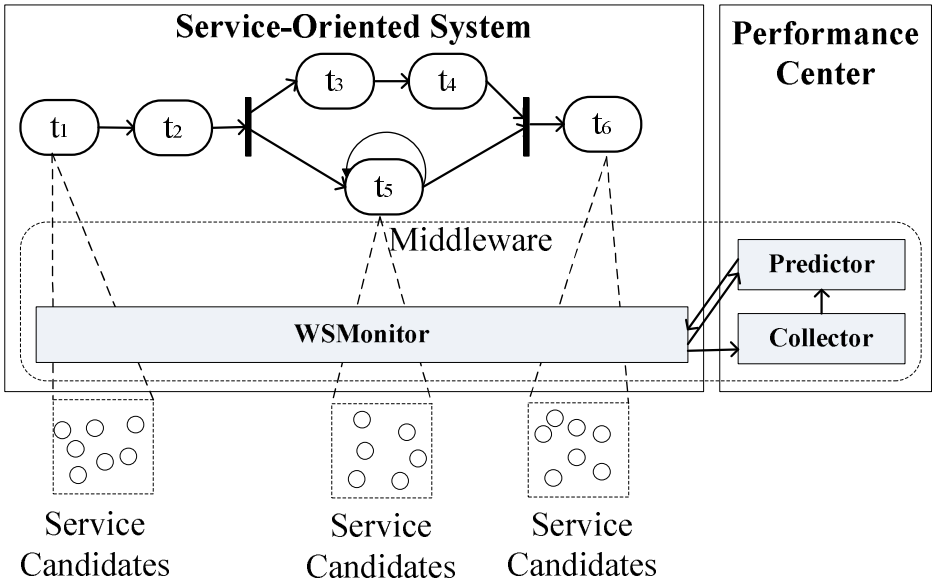


Figure 5.1: Service-Oriented System Architecture

at invocation time. Since service-oriented systems are increasingly running on large numbers of dynamic services, users often encounter highly dynamic and uncertain performance of service-oriented systems.

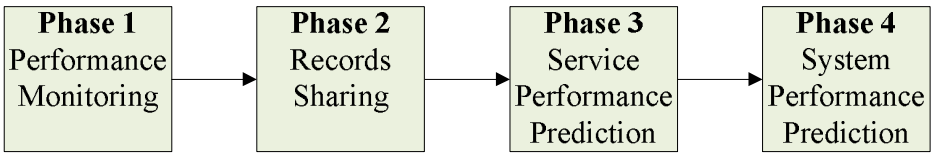


Figure 5.2: Online Performance Prediction Procedures

As shown in Figure 5.2, the online performance prediction mechanism proposed in this chapter contains four phases. In phase 1, each service user keeps local performance records of the Web services. In phase 2, local Web service usage experiences are uploaded to the performance center. Each user is encouraged to contribute its local records to obtain performance prediction service from the performance center. By contributing more individually observed Web service performance records, a service user can obtain more accurate performance prediction results from the performance center. By

combining performance records of several users, the performance center can obtain global performance information for all services. In phase 3, by performing time series analysis on the extracted time-specific user features and service features, a performance model is built in the performance center for personalized service performance prediction. The premise behind the performance model is that there is a small number of latent factors influencing the user observed service performance, and that a user's observed service performance is determined by how each factor applies to that user and the corresponding service at the current time slice. In phase 4, given the service level performance information, the overall performance of a service-oriented system is predicted based on the analysis of service compositional structures. When the most recent service performance information is available, an online prediction algorithm is applied for quickly updating the performance model, which requires no effort of recalculation for catching the performance trend. The detailed online service performance prediction approach is presented in Section 5.3.

In Figure 5.1, we can observe that the overall execution time of a service-oriented system mainly contains two parts: local computation time at the system side and response time of invoking remote services. The highly dynamic performance of service-oriented systems is mainly due to the highly dynamic response time of the composed services, while the local execution time is relatively stable. To improve the performance of systems at runtime, optimal Web service of each abstract task should be identified timely to replace the bad ones for composition. The overall performance of systems with different compositional options can be compared by estimating the total response time required for invoking all the composed services. The detailed system level performance prediction approach will be presented in Section 5.4.

Since most of the service users are not experts in service testing, to reduce the efforts of service users spent on testing the service per-

formance, we design a light-weight middleware for service users to automatically record invocation results, contribute the local records to the performance center, and receive performance prediction results from the performance center. Within the middleware, there are three management components: *WSMonitor*, *Collector* and *Predictor*. *WSMonitor* is deployed on the user side. *Collector* and *Predictor* are deployed on the performance center. *WSMonitor* is responsible for monitoring the performance of Web services and sending local records to the performance center. *Collector* is responsible for collecting shared performance records from users. *Predictor* is responsible for providing time-aware personalized performance prediction based on users' performance information collected by *Collector*.

5.3 Online Service Level Performance Prediction

In this section, we propose a collaborative method to predict the performance of services. Previous Web service related techniques such as selection [32, 112, 115, 117], composition [3, 4, 113], and orchestration [34] typically only employ average performance of service candidates at design-time. In the recent Web service literature, most of the state-of-the-art techniques can automatically update corresponding Web services with better ones at runtime. Therefore, making personalized time-specific performance prediction of Web services for different users becomes a critical task.

In this section, we first formally describe the online performance prediction problem of Web services in Section 5.3.1. Then we propose a latent feature learning algorithm to learn the time-aware user-specific and service-specific features in Section 5.3.2. The performance of services is predicted by applying the proposed online algorithm in Section 5.3.3. Finally, the complexity analysis is conducted in Section 5.3.4.

5.3.1 Problem Description

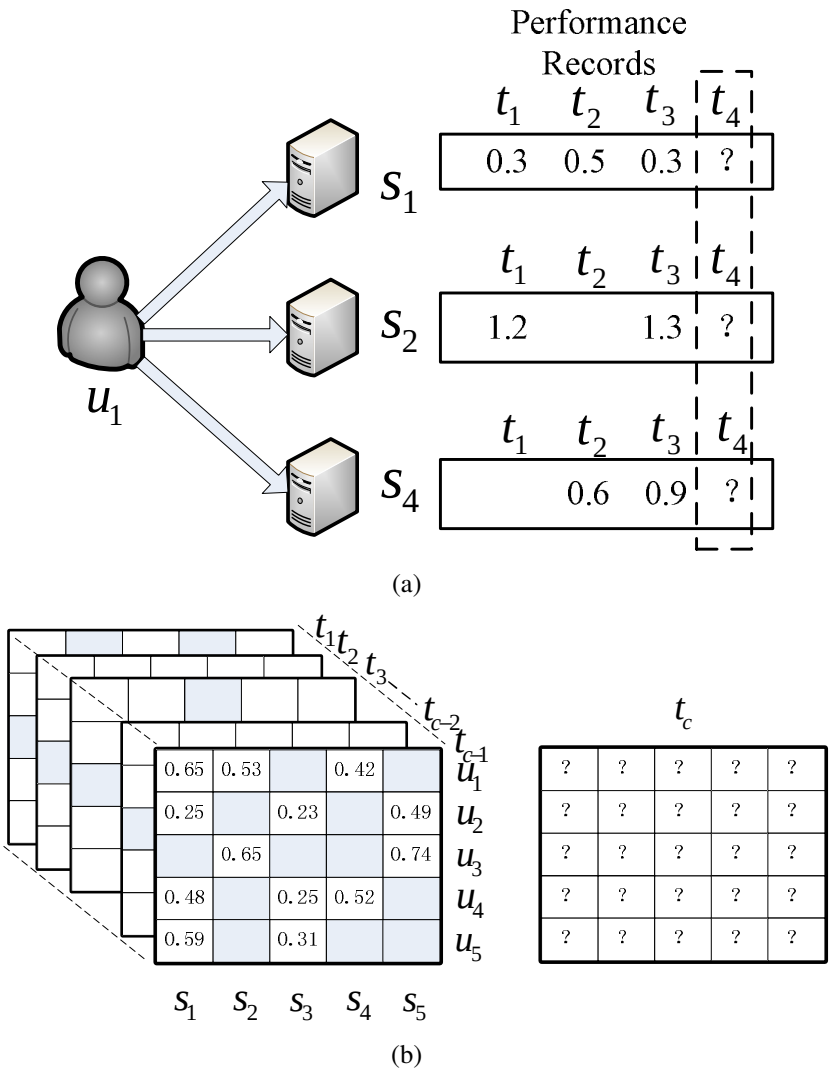


Figure 5.3: A Toy Example of Performance Prediction

Figure 5.3(a) illustrates a toy example of the performance prediction problem we study in this chapter. In this figure, service user u_1 has used three Web services s_1, s_2 and s_4 in the past. u_1 recorded the observed performance of Web services s_1, s_2 and s_4 with time stamp in the local site. By integrating all the performance information from different users, we can form a set of matrices as shown in Figure 5.3(b) with each matrix representing a time slice. In this

example, there are totally 5 users (from u_1 to u_5) and 5 services (from s_1 to s_5). Within a matrix, each entry denotes the observed performance (e.g., response time) of a Web service by a user during a specific time slice. A missing entry denotes that the corresponding user did not invoke the service in the time slice. The problem we study in this chapter is how to efficiently and precisely predict performance of services observed by a user in the next time slice based on the previously collected performance information.

Let U be the set of m users and S be the set of n Web services. In each time slice t, the observed response time from all users is represented as a matrix $R(t) \in \mathbb{R}^{m \times n}$ with each existing entry $r_{ui}(t)$ representing the response time of service i observed by user u in time slice t. Given the set of matrices $\Psi = \{R(k)|k < t_c\}$, matrix $R(t_c)$ should be predicted representing the expected response time of services in time slice t_c.

Without loss of generality, we can map the response time values to the interval $[0, 1]$ using the following function:

$$f(x) = \begin{cases} 0, & \text{if } x < r_{min} \\ 1, & \text{if } x > r_{max} \\ \frac{x - r_{min}}{r_{max} - r_{min}}, & \text{otherwise} \end{cases}$$

where r_{max} and r_{min} are the upper bound and lower bound of the response time values, respectively, which can be defined by users.

5.3.2 Time-Aware Latent Feature Model

In order to learn the latent features of users and services, we employ a matrix factorization technique to fit a feature model to user-service matrix in each time slice. The factorized user-specific and service-specific features are utilized to make further performance prediction. The idea behind the feature model is to derive a high-quality low-dimensional feature representation of users and services by analyzing the user-service matrices. It is noted that there is only a small

number of features influencing performance experiences, and that a user's performance experience vector is determined by how each feature is applied to that user and the corresponding service. Examples of physical features are network distance between the user and the server, the workload of the server, etc. Latent features are orthogonal representation of the decomposed results of physical features. Consider the matrix $R(t) \in \mathbb{R}^{m \times n}$ consisting of m users and n services. Let $p(t) \in \mathbb{R}^{l \times m}$ and $q(t) \in \mathbb{R}^{l \times n}$ be the latent user and service feature matrices in time slice t. Each column in $p(t)$ represents the l-dimensional user-specific latent feature vector of a user and each column in $q(t)$ represents the l-dimensional service-specific latent feature vector of a service. We employ an approximating matrix to fit the user-service matrix $R(t)$, in which each entry is approximated as:

$$\hat{r}_{ui}(t) = p_u^T(t) q_i(t) \tag{5.1}$$

where l is the rank of the factorization which is generally chosen so that $(m + n)l < mn$, since $p(t)$ and $q(t)$ are low-rank feature representations [63]. This matrx factorization procedure (i.e., decompose the user-service matrix $R(t)$ into two matrices $p(t)$ and $q(t)$) has clear physical meanings: Each column of $q(t)$ is a factor vector including the values of the l factors for a Web service, while each column of $p(t)$ is the user-specific coefficients for a user. In Eq. (5.1), the user-observed performance on service i at time t (i.e., $\hat{r}_{ui}(t)$) corresponds to the linear combination of the user-specific coefficients and the service factor vector.

In order to optimize the matrix factorization in each time slice, we first construct a cost function to evaluate the quality of approximation. The distance between two non-negative matrices is usually employed to define the cost function. In this chapter, due to the reason that there are a large number of missing values in practice, we only factorize the observed entries in matrix $R(t)$. Hence we conduct the matrix factorization as to solve the following optimization

problem:

$$\begin{aligned} & \min \mathcal{L}(p_u(t), q_i(t)) \\ = & \frac{1}{2}\sum_{u=1}^{m}\sum_{i=1}^{n} I_{ui}(r_{ui}(t) - g(\hat{r}_{ui}(t)))^2 \\ + & \frac{\lambda_1}{2}||p(t)||^2 + \frac{\lambda_2}{2}||q(t)||^2, \end{aligned} \tag{5.2}$$

where $\lambda_1, \lambda_2 > 0$, I_{ui} is the indicator function that is equal to 1 if user u invoked service i during the time slice t and equal to 0 otherwise. $(r_{ui}(t) - g(\hat{r}_{ui}(t)))^2$ evaluates the error between predicted value and groundtruth value collect from real-world. To avoid the overfitting problem, we add two regularization terms to Eq. (5.2) to constrain the norms of $p(t)$ and $q(t)$ where $||\cdot||^2$ denotes the Frobenius norm. λ_1 and λ_2 defines the importance of regularization terms. In other words, the optimal solution is highly rely on the error we evaluated in the first term. λ_1 and λ_2 defines the degree of accuracy in the first term to avoid overfitting problem. It The optimization problem in Eq. (5.2) minimizes the sum-of-squared-errors objective function with quadratic regularization terms. $g(x) = 1/(1 + exp(-x))$, which maps $\hat{r}_{ui}(t)$ to the interval $[0, 1]$. By solving the optimization problem, we can find the most appropriate latent feature matrices $p(t)$ and $q(t)$ to characterize the users and services, respectively.

A local minimum of the objective function given by Eq. (5.2) can be found by performing incremental gradient descent in feature vectors $p(t)$ and $q(t)$:

$$\begin{aligned} \frac{\partial L}{p_u(t)} = & I_{ui}(g(\hat{r}_{ui}(t)) - r_{ui}(t))g'(\hat{r}_{ui}(t))q_i(t) \\ & +\lambda_1 p_u(t), \end{aligned} \tag{5.3}$$

$$\begin{aligned} \frac{\partial L}{q_i(t)} = & I_{ui}(g(\hat{r}_{ui}(t)) - r_{ui}(t))g'(\hat{r}_{ui}(t))p_u(t) \\ & +\lambda_2 q_i(t). \end{aligned} \tag{5.4}$$

Algorithm 4 shows the iterative process for time-aware latent feature learning. We first initialize matrices $p(t)$ and $q(t)$ with small random non-negative values. Iterations of the update rules derived from Eq. (5.3) and Eq. (5.4) allow the objective function given in Eq. (5.2) converge to a local minimum.

Algorithm 4: Time-Aware Latent Features Learning

Input: $R(t)$, l, λ_1, λ_2
Output: $p(t)$, $q(t)$

Initialize $p(t) \in \mathbb{R}^{l \times m}$ and $q(t) \in \mathbb{R}^{l \times n}$ with small random numbers;
Load the performance records from matrix $R(t)$;
Calculate the objective function value $\mathcal{L}(p_u(t), q_i(t))$ by Eq. (5.1) and Eq. (5.2);
repeat
    Calculate the gradient of feature vectors $\frac{\partial L}{p_u(t)}$ and $\frac{\partial L}{q_i(t)}$ according Eq. (5.3) and Eq. (5.4), respectively;
    Update the latent user and service feature matrices $p(t)$ and $q(t)$;
    $p_u(t) \leftarrow p_u(t) - \frac{\partial L}{p_u(t)}$;
    $q_i(t) \leftarrow q_i(t) - \frac{\partial L}{q_i(t)}$;
    Update the objective function value $\mathcal{L}(p_u(t), p_i(t))$ by Eq. (5.1) and Eq. (5.2);
until *Converge* ;

5.3.3 Service Performance Prediction

After the user-specific and service-specific latent feature spaces $p(t)$ and $q(t)$ are learned in each time slice t, we can predict the performance of a given service observed by a user during the next time slice. The service performance prediction is conducted in two phases: offline phase and online phase. In the offline phase, the performance information collected from all the service users is used for statically modeling the trends of user features and service features. By employing a time series analysis, the features of users and services in the next time slice are calculated based on the evolutionary algorithm. The predicted features are further applied for calculating

the predicted performance of services in the next time slice. In the online phase, the newly observed service performance information by users at runtime is integrated into the feature model builded in the offline phase. By employing the incremental calculation algorithm, the feature model is updated efficiently to catch the latest trend for ensuring the prediction accuracy.

Phase 1: Offline Evolutionary Algorithm

Given the latent feature vectors of users and services in time slices before t_c, the latent feature vectors in time slice t_c can be predicted by precisely modeling the trends of features. Intuitively, older features are less correlated with a service's current status or a user's current characteristics. To characterize the latent features at time slice t_c, the prediction calculation should rely more on the information collected in the latest time slices than that collected in older time slices. In order to integrate the information from different time slices, we therefore employ the following temporal relevance function [68]:

$$f(k) = e^{-\alpha k}, \tag{5.5}$$

where k is the amount of time that has passed since the corresponding information was collected. $f(k)$ measures the relevance of information collect from different time slices for making prediction on latent features at time t_c. Note that $f(k)$ decreases with k. By employing the temporal relevance function $f(k)$, we can assign a weight for each latent feature vector depending on the collecting time when making prediction. In the temporal relevance function, α controls the decaying rate. By setting α to 0, the evolutionary nature of the information is ignored. A constant temporal relevance value of 1 is assigned to latent feature vectors in all the time slices, which means latent feature vectors in time slice t_c are predicted simply by averaging the vectors before time slice t_c. Since $e^{-\alpha}$ is a constant value, the value of temporal relevance function can be recursively

computed: $f(k+1) = e^{-\alpha} f(k)$, in which $e^{-\alpha}$ denotes the constant decay rate.

By analyzing the collected performance data, we obtain two important observations: (1) Within a relatively long time period such as one day or one week, the service performance observed by a user may vary significantly due to the highly dynamic service side status (e.g., workloads of weather forecasting service may increase sharply when weekends are coming.) and user side environment (e.g., network latency would increase during the office hours). (2) Within a relatively short time period such as one minute or one hour, a service performance observed by a user is relatively stable. The above two observations indicate that the feature information of latent feature vectors in time slice t_c can be predicted by utilizing the feature information collected before t_c. Moreover, the performance curve in terms of time should be smooth, which means more recent information is placed with more emphasis for predicting the performance in time slice t_c. Therefore, we estimate the feature vectors in time slice t_c by computing the weighted average of feature vectors in the past time slice:

$$\hat{p}_u(t_c) = \frac{\sum_{k=1}^{w} p_u(t_{c-k}) f(k)}{\sum_{k=1}^{w} f(k)}, \tag{5.6}$$

$$\hat{q}_i(t_c) = \frac{\sum_{k=1}^{w} q_i(t_{c-k}) f(k)}{\sum_{k=1}^{w} f(k)}, \tag{5.7}$$

where $\hat{p}_u(t_c)$ and $\hat{q}_i(t_c)$ are the predicted user feature vector and service feature vector in time slice t_c, respectively. w controls the information of how many past time slices are used for making prediction. In Eq. (5.6) and Eq. (5.7), large weight values are assigned to the feature vectors in recent slices while small weight values are assigned to the feature vectors in old slices.

Given the predicted latent feature vectors $\hat{p}_u(t_c)$ and $\hat{q}_i(t_c)$, we can predict the service performance value observed by a user in time slice t_c. For the user u and the service i, the predicted performance

value $\hat{r}_{ui}(t_c)$ is defined as

$$\hat{r}_{ui}(t_c) = \hat{p}_u^T(t_c)\hat{q}_i(t_c) \tag{5.8}$$

Phase 2: Online Incremental Algorithm

In this phase, we propose an incremental algorithm for efficiently updating the feature model built in phase 1 at runtime as new performance data are collected in each time slice. In time slice t_{c-1}, $\hat{p}_u(t_{c-1})$ and $\hat{q}_i(t_{c-1})$ are predicted based on the data collected during the time slice t_{c-2-w} and t_{c-2}. During the time slice t_{c-1}, there would be some services invoked by several different users. Therefore, newly observed service performance values are available and collected from users. The new performance data are stored in a user-service matrix $R(t_{c-1})$ representing information in time slice t_{c-1}. By performing matrix factorization on $R(t_{c-1})$, latent feature vectors $p_u(t_{c-1})$ and $q_i(t_{c-1})$ in time slice t_{c-1} are learned from the real performance data. According to Eq. (5.6) and Eq. (5.7), the feature vector prediction needs to be recomputed repeatedly at each time slice using all the vectors in previous w time slices, which is highly computationally expensive. In order to predict the feature vectors in time slice t_c more efficiently, we rewrite the Eq. (5.6) and Eq. (5.7) as follows:

$$\begin{aligned}\hat{p}_u(t_c) = \; & e^{-\alpha}\left(\frac{p_u(t_{c-1})}{\sum_{k=1}^{w} f(k)} + \hat{p}_u(t_{c-1})\right. \\ & \left. - \frac{p_u(t_{c-1-w})f(w)}{\sum_{k=1}^{w} f(k)}\right),\end{aligned} \tag{5.9}$$

$$\begin{aligned}\hat{q}_i(t_c) = \; & e^{-\alpha}\left(\frac{q_i(t_{c-1})}{\sum_{k=1}^{w} f(k)} + \hat{q}_i(t_{c-1})\right. \\ & \left. - \frac{q_i(t_{c-1-w})f(w)}{\sum_{k=1}^{w} f(k)}\right),\end{aligned} \tag{5.10}$$

where $e^{-\alpha}$, $f(w)$ and $\sum_{k=1}^{w} f(k)$ are constant values. $p_u(t_{c-1-w})$ and $q_i(t_{c-1-w})$ are feature vectors calculated in time slice t_{c-1-w} and can

be stored with only constant memory space. $p_u(t_{c-1})$ and $q_i(t_{c-1})$ can be quickly calculated in time slice t_{c-1} since the computation complexity of matrix factorization is very low. Note that in Eq. (5.9) and Eq. (5.10), we obtain a recursive relation between $[p_u(t_{c-1}), q_i(t_{c-1})]$ and $[p_u(t_c), q_i(t_c)]$, which means the feature model in time slice t_{c-1} can be efficiently updated for predicting the feature vectors in new time slice t_c.

In the online phase, it could be possible that a new user or service is found. Since there is no prior information about the user or the service in the previous time slices, it is difficult to precisely predict the corresponding features by employing the online Incremental Algorithm. To address the cold start problem, we employ average performance for prediction. More precisely, the prediction for a new user or a new service is set as follows:

$$\hat{r}_{ui}(t) = \begin{cases} \bar{r}_i(t), & \text{if new user and old service} \\ \bar{r}_u(t), & \text{if old user and new service} \\ \bar{r}(t), & \text{if new user and new service} \end{cases}$$

where $\bar{r}_i(t)$ is the average predicted performance of service i observed by all users in time slice t, $\bar{r}_u(t)$ is the average predicted performance of all services observed by user u in time slice t, $\bar{r}(t)$ is the average predicted performance of all user-service pairs in time slice t.

5.3.4 Computation Complexity Analysis

The offline phase includes learning latent features in w time slices and running an evolutionary algorithm. The main computation is evaluating the objective function $\mathcal{L}$ and its gradients against the variables. Since the matrix $R(t)$ is typically sparse, the computational complexity for evaluating the objective function $\mathcal{L}$ in each time slice is $O(\rho_r l)$, where ρ_r is the number of nonzero entries in the matrix $R(t)$, l is the dimension of the latent features. The computa-

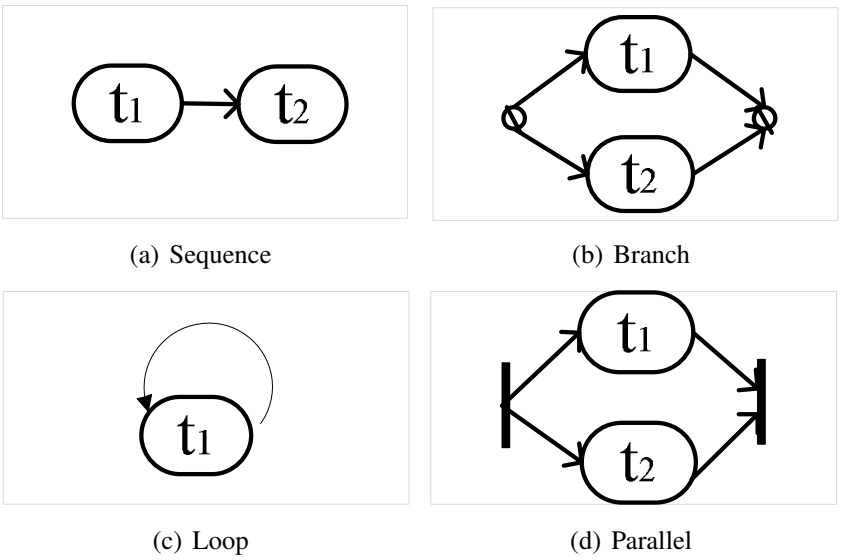
tional complexities for the gradients $\frac{\partial \mathcal{L}}{\partial p_u(t)}$ and $\frac{\partial \mathcal{L}}{\partial q_i(t)}$ in Eq. (5.3) and Eq. (5.4) are $O(\rho_r l)$. Therefore, the total computational complexity in one iteration is $O(\rho_r l w)$, where w is the number of time slices. In the online phase, the main computation is factorizing the new performance matrix in time slice t. The computational complexity of online incremental algorithm is $O(\rho_r l)$.

The analysis indicates that theoretically, the computational time of offline algorithm is linear with respect to the number of observed performance entries in one time slice and the total number of time slices whose information is used for prediction. Note that because of the sparsity of $R(t)$, $\rho_r << mn$, which indicates that the computation time grows slowly with respect to the size of matrix $R(t)$. The computational time of the online algorithm is linear with the amount of newly observed performance information, which indicates that our proposed approach can efficiently integrate the performance model with new information and make online prediction timely. This complexity analysis shows that our proposed approach is very efficient and can be applied to large-scale systems.

5.4 System Level Performance Prediction

In this section, we first present the aggregated response time calculation methods for basic compositional structures. Then, by analyzing the service flow, the system level response time can be predicted in a hierarchical way. The overall performance of a system consists of service response time and local execution time. Local execution time refers to the computation time between service invocations in local system. Since the variance of system performance at runtime is mainly due to the highly varying service response time, local execution time, which is relatively constant at runtime, is not included in the defined system level performance.

Typically, there are four types of basic compositional structures, i.e., sequence, branch, loop, and parallel. The response time of each



(a) Sequence (b) Branch

(c) Loop (d) Parallel

Figure 5.4: Basic Compositional Structures

structure can be calculated by aggregating the response time of its sub-tasks as shown in Table 5.1.

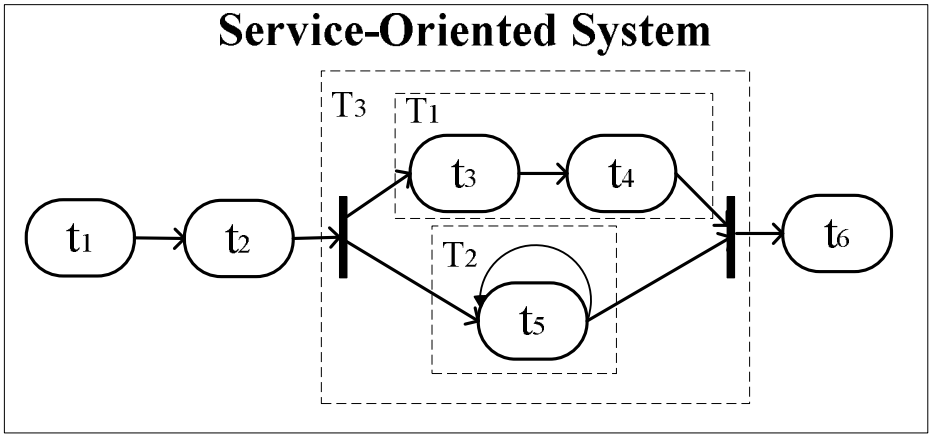


Figure 5.5: A Performance Composition Example

For predicting the overall execution time of a service flow, we first decompose the system structure to a set of basic compositional structures in a hierarchical way. Then the end-to-end system execution time is calculated in a bottom up way. Take Figure 5.5 as an example, first the execution time of basic compositional structures

Table 5.1: Calculation of Aggregated Response Time

Structure	Calculation Method	Meaning of Notation
Sequence	$r = \sum_{i=1}^{n} r_i$	n: number of sequential sub-tasks r_i: response time of the i^{th} sub-task
Branch	$r = \sum_{i=1}^{n} p_i r_i$	n: number of branches r_i: response time of the i^{th} branch p_i: probability of the i^{th} branch to be executed
Loop	$r = \sum_{i=1}^{n} p_i r_i i$	n: maximum looping times r_i: response time of the i^{th} sub-task p_i: probability of executing the sub-task for i times
Parallel	$r = \max_{i=1}^{n} r_i$	n: number of branches r_i: response time of the i^{th} branch

T_1 and T_2 is calculated by employing the aggregation methods of sequence and loop, respectively. Then the execution time of T_3 is calculated by employing aggregation method for branch compositional structure. Finally, the overall system execution time is calculated by employing aggregation method for sequence on t_1, t_2, T_3 and t_6.

With the aggregation approach, designers of service-oriented systems can estimate the performance of systems at design-time. At runtime, the user observed system level performance can be efficiently predicted automatically. Once the system performance is decreased at runtime, by analyzing the system structure in a top down way, bad performance services can be quickly identified. With the predicted service performance information, dynamical service composition techniques can be employed to improve the system performance by replacing the long response time services with better ones.

5.5 Experiments

In this section, we conduct two experiments to evaluate our online performance prediction approach. In the first experiment, by comparing with several state-of-the-art service performance prediction methods, we present the effectiveness and efficiency of our approach. In the second experiment, we study the service flow of a real-world service-oriented system. We also study the performance improvement by integrating the predicted performance information of our approach into the dynamic composition mechanism.

In the following, Section 5.5.1 introduces the experimental setup and gives the description of the experimental dataset. Section 5.5.2 defines the evaluation metrics. Section 5.5.3 compares the prediction quality of our approach with other competing approaches. Section 5.5.4, Section 5.5.5 and Section 5.5.6 study the impact of data density, dimensionality, and parameter α and w, respectively. Section 5.5.7 compare the computational time of different approaches. Section 5.10 studies the system level performance prediction.

5.5.1 Experimental Setup and Dataset Collection

To evaluate the service level performance prediction quality of our proposed approach in the real world, we implement a tool WSMonitor for collecting the performance information of Web services. WSMonitor is deployed as a middleware on the user-side, which can continuously monitor the user experienced performance of invoked services. By sharing the user side observed performance to performance center, it can obtain performance prediction service from performance center at runtime.

WSMonitor is implemented and deployed with JDK 6.0, Eclipse 3.3, Axis 2, and Apache 2.2.17. Within WSMonitor there are several modules including *WSDL Crawler*, *Code Generator*, and *Performance Monitor*. *WSDL Crawler* first crawls a set of WSDL files from the Internet and generates a list of openly-accessible Web ser-

vices. For each Web service in the list, *Code Generator* automatically generates a java class for service invocation by employing the WSDL2Java tool from the Axis package [48]. Totally, 5,871 classes are generated for 5,871 Web services. By calling the functions generated by Code Generator, *Performance Monitor* is able to send operation requests to Web services and record the corresponding response time with time stamps.

We deploy the WSMonitor on 142 distributed computers located in 22 countries from PlanetLab[1], which is a distributed test-bed consisting of hundreds of computers all over the world. Each computer acts as a service user by invoking the listed Web services from time to time. Totally, 4,532 publicly available real-world Web services from 57 countries are monitored by each computer continuously. 1,339 of the initially selected Web services are excluded in this experiment due to: 1) authentication required and 2) permanent invocation failure (e.g., the Web service is shutdown). In our experiment, each of the 142 computers sends operation requests to all the 4,532 Web services in every time slice. The experiment lasts for 16 hours with one time slice lasting for 15 minutes.

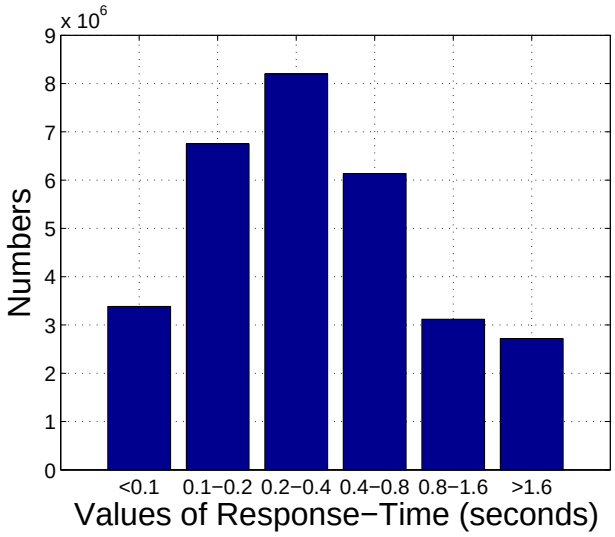
By collecting performance records from all the computers, finally 30,287,611 performance results are included into the Web service response time dataset. The response time of all the 4,532 Web services observed by all the 142 service users during 64 time slices can be presented as a set of 142×4532 user-service matrices, each of which stands for a particular time slice.

The statistics of Web service response time dataset are summarized in Table 5.2. Response-time is within the range of 0-20 seconds, whose mean is 3.165 seconds. The distribution of the response-time values of all the matrices is shown in Figure 5.6(a). From Figure 5.6(a) we can observe that most of the response-time values are between 0.1-0.8 seconds.

[1] http://www.planet-lab.org

Table 5.2: Statistics of Web Service Response Time Dataset

Statistics	Response Time
Scale	0-20s
Mean	3.165s
Num. of Users	142
Num. of Web Services	4,532
Num. of Time Slices	64
Num. of Records	30,287,611



(a)

Figure 5.6: Response Time Value Distribution

5.5.2 Metrics

We assess the prediction quality of our proposed approach in comparison with other methods by computing Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). The metric MAE is defined as:

$$MAE = \frac{\sum_{uit} |\hat{r}_{ui}(t) - r_{ui}(t)|}{N}, \tag{5.11}$$

and RMSE is defined as:

$$RMSE = \sqrt{\frac{\sum_{uit}(\hat{r}_{ui}(t) - r_{ui}(t))^2}{N}}, \tag{5.12}$$

where $r_{ui}(t)$ is the response time value of Web service i observed by user u in time slice t, $\hat{r}_{ui}(t)$ denotes the predicted response time value of Web service i would be observed by user u in time slice t, and N is the number of predicted response time values in the experiments.

5.5.3 Comparison

In this section, in order to show the effectiveness and efficiency of our proposed online Web service performance prediction approach, we compare the service level prediction accuracy of the following methods:

- **UPCC**-This is a neighborhood-based method which employs Pearson Correlation Coefficient to calculate similarities between users. It predicts response time of services based on the observed performance from similar users [11, 96]. Since UPCC cannot perform online prediction for the next time slice, we extend the traditional UPCC by using the average performance from similar users for prediction.
- **IPCC**-This is a neighborhood-based method which employs Pearson Correlation Coefficient to calculate similarities between services. It predicts response time of services based on the performance of similar services [85]. Similar to UPCC, we make an extension to IPCC in order to compare the online prediction quality with other methods.
- **MF**-This method first compresses the set of user-service matrices into an average user-service matrix. For each entry in the matrix, the value is the average of the specific user-service pair

Table 5.3: Performance Comparsions (A Smaller MAE or RMSE Value Means a Better Performance)

Data Density	**RMSE**	**Response Time (seconds)**					
		UPCC	**IPCC**	**MF**	**TF**	**WSPred**	**OPred**
5%	Mean	5.312	5.289	5.329	4.751	4.362	**4.330**
	Best	5.263	5.276	5.321	4.747	4.358	**4.327**
10%	Mean	5.043	4.972	5.079	4.567	4.287	**4.151**
	Best	4.962	4.946	5.063	4.563	4.283	**4.148**
45%	Mean	4.425	4.371	4.337	4.208	3.923	**3.855**
	Best	4.388	4.342	4.318	4.202	3.918	**3.851**
50%	Mean	4.352	4.354	4.298	4.016	3.899	**3.809**
	Best	4.331	4.336	4.274	4.012	3.894	**3.808**

Data Density	**MAE**	**Response Time (seconds)**					
		UPCC	**IPCC**	**MF**	**TF**	**WSPred**	**OPred**
5%	Mean	3.720	3.213	3.387	2.915	2.559	**2.417**
	Best	3.687	3.207	3.381	2.911	2.555	**2.413**
10%	Mean	3.264	2.841	2.873	2.786	2.495	**2.376**
	Best	3.243	2.812	2.851	2.782	2.488	**2.374**
45%	Mean	2.627	2.455	2.436	2.253	2.141	**2.029**
	Best	2.613	2.431	2.423	2.247	2.137	**2.026**
50%	Mean	2.619	2.417	2.391	2.211	2.130	**2.011**
	Best	2.609	2.404	2.384	2.207	2.125	**2.008**

during all the time slices. After obtaining the compressed user-service matrix, it applies the non-negative matrix factorization technique proposed by Lee and Seuing [63] on user-service matrix for missing value prediction. The predicted values are used as the response time of the corresponding user-service pair in the next time slice.

- **TF**-This is a tensor factorization based prediction method. It combines the set of user-service matrices as a tensor with a third dimension representing the time. Then it applies tensor factorization on the user-service-time tensor to extract user-specific, service-specific and time-specific characteristics. The missing value is then predicted based on how these characteristics apply to each other.

- **WSPred**-This is a tensor factorization-based prediction method [120]. Different from method **TF**, it adds average performance value constraints when extracting the latent characteristics.

- **OPred**-This method is proposed in this chapter. Firstly the user features and service features are extracted in each time slice by employing matrix factorization. Then the user features and service features in the new time slice are predicted by performing time analysis on the feature trends. Finally, the response time of user-service pairs is predicted by evaluating how the predicted features of users and services are applied to each other.

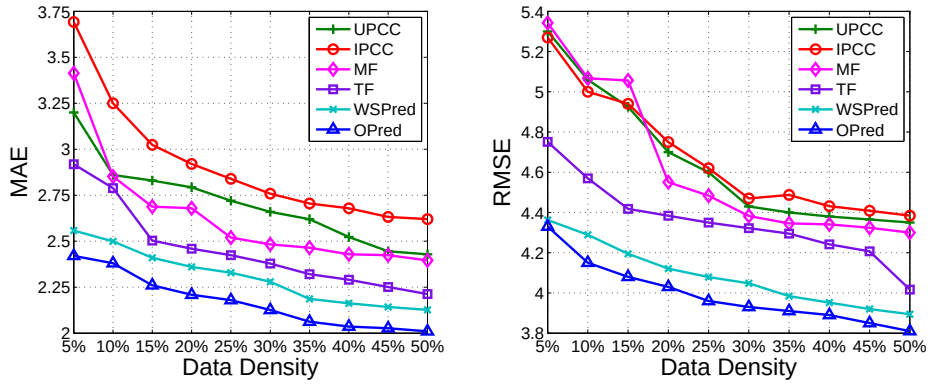
In order to evaluate the performance of different approaches in reality, we randomly remove some entries from the performance matrices to obtain observation matrices and compare the values predicted by a method with the original ones. The observation matrices with missing values are in different densities. For example, 10% means that we randomly remove 90% entries from the original matrices and use the remaining 10% entries for prediction. Note that under a certain density, we employ different approaches to predict

Table 5.4: Performance Improvement of OPred

Competing Approach	Performance Improvement of OPred
UPCC	22-36%
IPCC	16-25%
MF	15-28%
TF	9-17%
WSPred	1-6%

the values by using the same observation matrix. The prediction accuracy is evaluated using Eq. (5.11) and Eq. (5.12) by comparing the original values and the predicted values in the corresponding matrices. The values of λ_1 and λ_2 are tuned by performing cross-validation [49] on the observed performance data. Without lost of generality, the parameter settings of all the approaches are $l = 20$, $w = 8$, $\alpha = 0.2$ and $\lambda 1 = \lambda 2 = 0.001$ in the experiments conducted in this chapter. Detailed impacts of parameters are studied in Section 5.5.4, Section 5.5.5 and Section 5.5.6, respectively.

The service performance prediction accuracies evaluated by MAE and RMSE are shown in Table 5.3. A smaller MAE or RMSE value means a better performance. From Table 5.3, we can observe that the time-aware prediction methods (i.e., TF and OPred) outperform the non time-aware prediction methods (i.e., UPCC, IPCC and MF), since the time-aware methods employ the time-specific features as additional information for performance prediction. We also observe that our approach OPred constantly performs better than TF under both dense data and sparse data. This is because OPred assigns different weights on the performance information collected in different time slices. The prediction results rely more on recent user and service features than older ones. By setting $f(x)$ in Eq. (5.5) to a constant value (e.g., $f(x) = 1$), OPred is reduced to TF. WSPred further improves TF by employing a regularization term to prevent the predicted values from varying a lot against the average performance value. WSPred catches the periodic features of service per-



(a) (b)

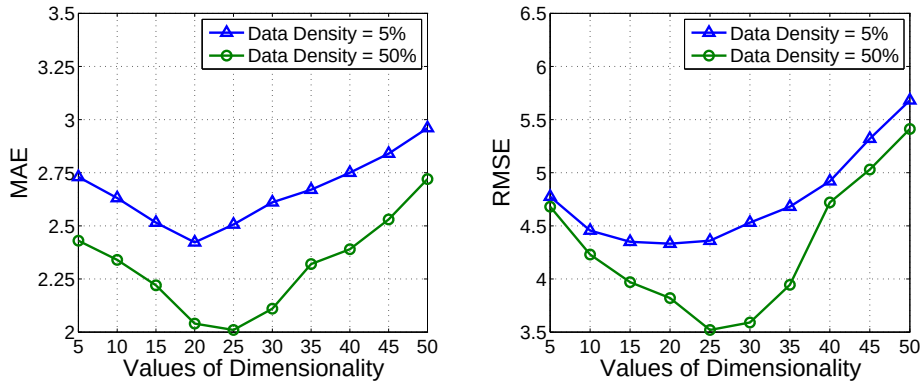
Figure 5.7: Impact of Data Density

formance. OPred proposed in this chapter captures not only the periodic features but also the non-periodic features of service performance. Therefore, OPred can predict the performance trend more precisely than WSPred. Moreover, WSPred is not an online approach and requires more computational time than OPred. The computational time is compared in Section 5.5.7. In Table 5.3, the MAE and RMSE values of dense data (e.g., data density is 45% or 50%) are smaller than those of sparse data (e.g., data density is 5% or 10%), since denser data provide more information for prediction. Performance improvement of OPred is shown in Table 5.4. Our online approach OPred improves the prediction accuracy by 22-36%, 16-25%, 15-28%, 9-17% and 1-6% relative to UPCC, IPCC, MF, TF and WSPred, respectively. The improvements are significant, which indicates the prediction effectiveness of OPred.

5.5.4 Impact of Data Density

In Figure 5.7, we compare the prediction accuracy of all the methods under different data densities. We change the data density from 5% to 50% with a step value of 5%. The parameter settings in this experiment are $l = 20$, $w = 8$, $\alpha = 0.2$ and $\lambda 1 = \lambda 2 = 0.001$.

In Figure 5.7(a) and Figure 5.7(b), the experimental results show



(a) (b)

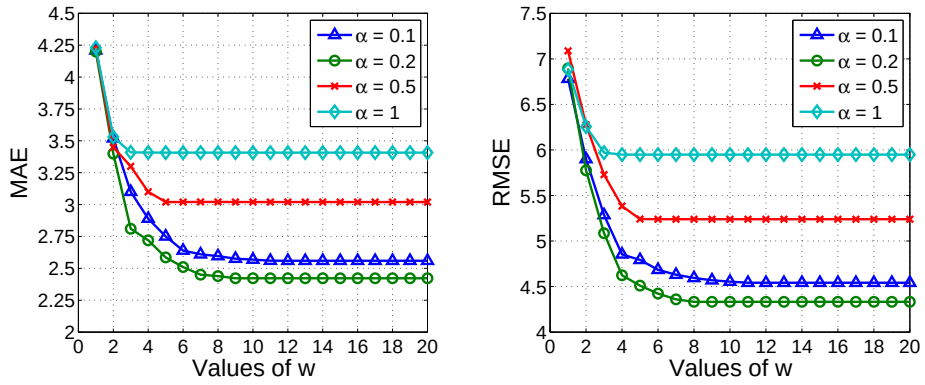
Figure 5.8: Impact of Dimensionality

that our approach OPred achieves higher prediction accuracy (smaller MAE and RMSE values) than other competing methods under different data density. In general, when the data density is increased from 5% to 20%, the prediction accuracy of our approach OPred is significantly enhanced. When the data density is further increased from 20% to 50%, the enhancement of prediction accuracy will decrease. This observation indicates that when the data are very sparse, collecting more performance information will greatly enhance the prediction accuracy.

5.5.5 Impact of Dimensionality

The parameter dimensionality l determines the number of latent features applied to characterize users and services. In Figure 5.8, we study the impact of parameter dimensionality by varying the values of l from 5 to 50 with a step value of 5. Other parameter settings are $w = 8$, $\alpha = 0.2$ and $\lambda 1 = \lambda 2 = 0.001$.

In Figure 5.8, we observe that as l increases, the MAE and RMSE decrease (prediction accuracy increases), but when l surpasses a certain threshold like 20, the MAE and RMSE increase (prediction accuracy decreases) with further increase of the value of l. This observation indicates that too few latent factors are not enough to charac-



(a) (b)

Figure 5.9: Impact of α and w

terize the features of user and service, while too many latent factors will cause an overfitting problem. There exists an optimal value of l for characterizing the latent features. When the data density is 50%, we observe that our approach OPred achieves the best performance when the value of dimensionality is 25, while smaller values like 5 or larger values like 50 can potentially reduce the prediction accuracy. When the data density is 5%, we observe that the prediction accuracy of our approach OPred achieves the best performance when the value of dimensionality is 20, while smaller values like 5 or larger values like 50 can potentially reduce the prediction accuracy. This observation indicates that when the service performance data are sparse, 20 latent factors are already good enough to characterize the features of user and service, which are mined from the limited performance information. On the other hand, when the data are dense, more latent factors, like 25, are needed to characterize the latent features since more performance data are available.

5.5.6 Impact of α and w

The parameter α controls the decaying rates of weights assigned to different time slices. A larger value of α gives more weights to the recent time slices. w controls the information of how many past

Table 5.5: Average Computational Time Comparisons

Approach	Computational Time	Percentage of A Time Slice
UPCC	10.095m	67.3%
IPCC	9.735m	64.9%
MF	1.575m	10.5%
TF	1.860m	12.4%
WSPred	2.055m	13.7%
OPred	0.240m	1.6%

time slices are used for making prediction. In Figure 5.9, we vary the values of w from 1 to 20 with a step value of 1. Other parameter settings are $\lambda 1 = \lambda 2 = 0.001$.

Figure 5.9 shows the impacts of α and w on MAE and RMSE. We observe that as w increases, the values of MAE and RMSE decrease (prediction accuracy increase) at first, but when w pass a certain threshold, the MAE and RMSE converge. This phenomenon coincides with the intuition that employing past performance information from more time slices can increase prediction accuracy. When w surpasses a certain threshold, the MAE and RMSE decrease little with further increase of the value of w. The reason is that when w is large enough, small weight values are assigned to the information of older time slices, which contribute little to the prediction accuracy. This observation indicates that too large w is unnecessary. The thresholds are different under different values of α. Since a larger value of α gives more weights to the recent time slices, the threshold is smaller than those under smaller values of α. In Figure 5.9, OPred achieves the best performance when $\alpha = 0.2$. The observation confirms with the intuition that with a large value of α useful information from older time slice will be lost, and with a small value of α noisy data will cause the decrease of prediction accuracy.

5.5.7 Computational Time Comparisons

In Section 5.3.4, we theoretically analyze the computation time of OPred. In this section, we compare the computation efficiencies of different approaches. In our experiments, one time slice lasts for 15 minutes. We compare the average computational time of a prediction approach with the length of a time slice. The data used for performance prediction are the same for all approaches. From Table 5.5, we observe that the computational time of OPred takes less than 2% of a time slice. This observation is consistent with the time complexity analysis in Section 5.3.4 and shows that our proposed approach OPred is efficient and can be applied to large-scale systems in real-world. TF and WSPred use more than 10% of a slice time to conduct prediction, since they are not online approaches and need to rebuild the model whenever new data are available. TF performs better than WSPred because WSPred contains an extra term in the objective function representing the average performance constraints. MF performs better than TF and WSPred because time factor is not considered when predicting the performance values. UPCC and IPCC perform worst since they are neighborhood-based approaches and take a lot of time to find the relationship between users and services.

5.5.8 System Level Performance Case Study

In this section, we evaluate our approach OPred by using a sample service-oriented system. Figure 5.10 shows a typical online shopping system. It allows customers to browse and order products from the shopping website. In this shopping system, the designer integrates three Web services for providing users access to various product suppliers, banks and shippers. This example is taken from the online services provided by a gift website [37].

The service flow is illustrated in Figure 5.10. By sending product queries to suppliers, the shopping system can obtain plenty of prod-

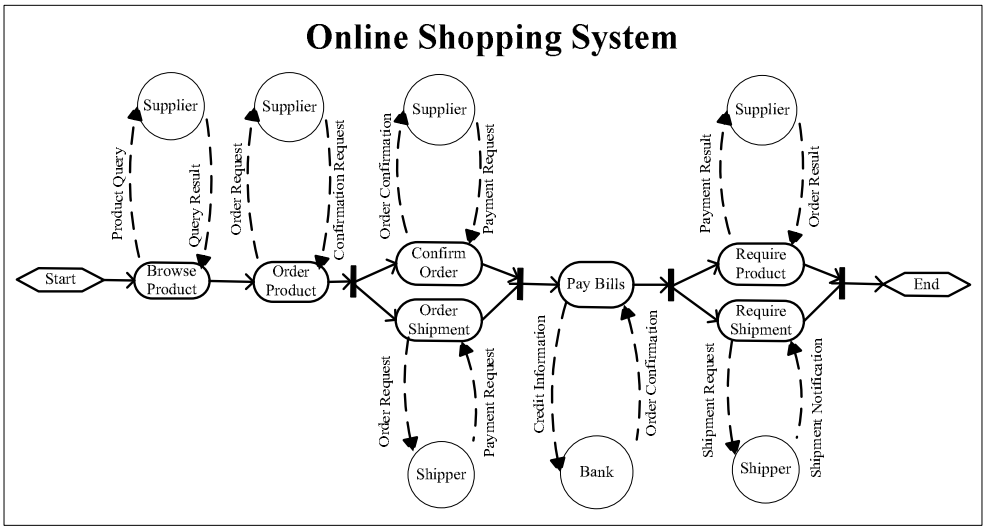


Figure 5.10: An Online Shopping System

uct information, which allows customers to browse various products on the website. Once a customer decides to buy a product, the shopping system sends an order request with product information to the corresponding supplier. The supplier then reserves a product for the customer and replies the shopping system with an order confirmation request. At this point, the shopping system needs to send an order confirmation to the supplier and an order request to a shipper service. Once the shopping system receives payment requests from both the product supplier and a shipper service, it proceeds to launch a payment transaction via a credit card payment service (e.g., paypal). In the task of paying bills, customer's credit card information is transferred to the bank, and an invoice is sent back by the bank. Finally, the product supplier is notified of an bank invoice to complete the purchase. At the same time, a request is sent to the shipper to arrange the shipment of the product. Once the product is aboard, the shipper notifies the shopping system with estimated arrival date of the shipment.

After we find a set of functional identical Web services from the performance dataset for each abstract task in the shopping system.

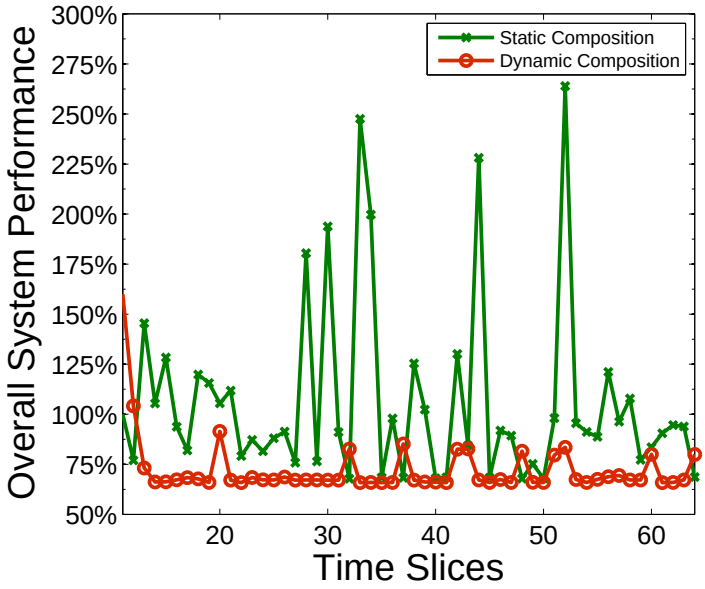


Figure 5.11: System Performance Improvement of Dynamically Service Composition

The predicted service performance results are used to predicting the end-to-end performance of shopping system by employing the compositional methods in Section 5.4. As discussed before, by calculating system performance, poor services can be identified in a hierarchical way. Then the identified services can be replaced with better ones to maintain the overall system performance at runtime. In Figure 5.11, we compare the system performance of static composition and dynamical composition. In static composition, for each abstract task we randomly choose a service from the set of functional identical candidates. The set of selected services is fixed in all time slices. In dynamical composition, the predicted service performance of OPred is employed to select the optimal services for task executions in each time slice. In this thesis, we focus on dynamic selecting atomic services. The comparison begins from time slice 11 since the performance information of the first 10 time slices is used as training data for OPred. The system performance of static composition method in time slice 11 is chosen as baseline. Other performance

is compared with baseline in percentage (a smaller number means better performance). From Figure 5.11 we can observe that the system performance of static composition is unstable at runtime. This is because the performance of some selected services is unstable, which impacts the system overall performance. For dynamic composition, since OPred can precisely predict service performance, the service-oriented system can be updated by integrating potentially optimal services at runtime. The system performance of dynamical composition maintains stable in a good level, which indicates the effectiveness of OPred.

5.6 Summary

Based on the intuition that a user's current Web service performance usage experience can be predicted by using the past usage experience from different users, we propose a novel online service performance prediction approach, called OPred, for personalized performance prediction at runtime. Using the past Web service usage experience from different users, OPred builds feature models and employs time series analysis techniques on feature trends to make personalized performance prediction for different service users. The predicted service performance is critical for identifying poor services and maintaining the system performance timely. The extensive experimental results show that our proposed OPred outperforms the state-of-the-art performance prediction approaches in terms of prediction accuracy. The case study on a typical shopping system shows the effectiveness of OPred.

For future work, we will investigate more techniques for improving the prediction accuracy (e.g., data smoothing, utilizing content-aware information, etc.). We will conduct experiments on more real-world service-oriented systems to evaluate the effectiveness and efficiency of OPred when applied to different domains.

□ **End of chapter.**

Chapter 6

QoS-Aware Web Service Searching

6.1 Overview

With a set of standard protocols, i.e., SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery and integration), Web services provided by different organizations can be discovered and integrated to develop applications [26]. With the growing number of Web services in the Internet, many alternative Web services can provide similar functionalities to fulfill users' requests. Syntactic or semantic matching approaches based on services' tags in UDDI repository are usually employed to discover suitable Web services [105]. However, discovering web services from UDDI repositories suffers several limitations. First, since UDDI repository is no longer a popular style for publishing Web services, most of the UDDI repositories are seldom updated. This means that a significant part of information in these repositories is out of date. Second, arbitrary tagging methods used in different UDDI repositories add to the complexity of searching Web services of interest.

To address these problems, an automated mechanism is required to explore existing Web services. Considering that WSDL files are used for describing Web services and can be obtained in several ways other than UDDI repositories, several WSDL based Web ser-

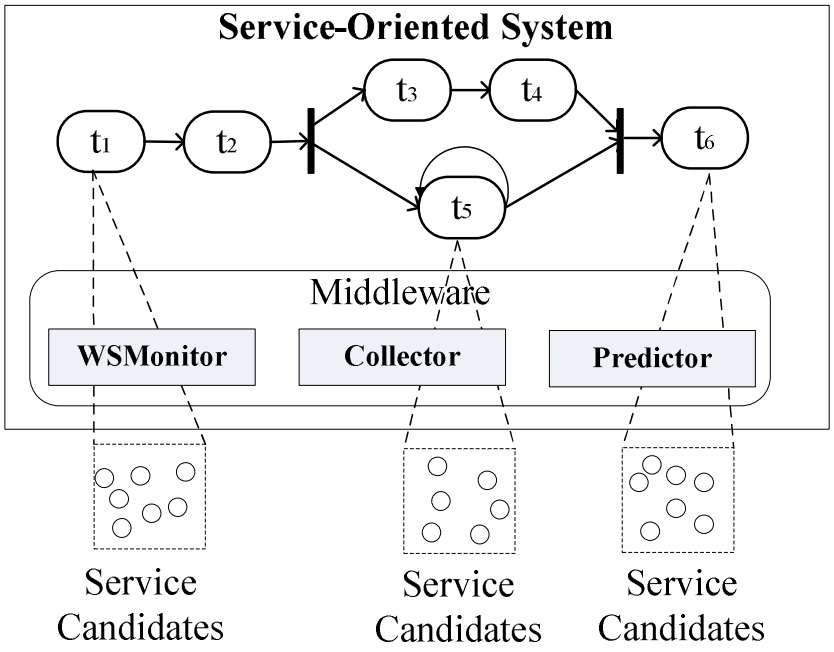


Figure 6.1: Service-Oriented System Architecture

vice searching approaches are proposed. Such as *Binding Point*[1], *Grand Central*[2], *Salcentral*[3], and *Web Service List*[4]. However, these engines only simply exploit keyword-based search techniques which are obviously insufficient for catching the Web services' functionalities. First, keywords cannot represent Web services' underlying semantics. Second, since a Web service is supposed to be used as part of the user's application, keywords cannot precisely specify the information user needs and the interface acceptable to the user. In this chapter, we employ not only keywords but also operation parameters to comprehensively capture Web service's functionality.

In addition, Web services sharing similar functionalities may possess very different non-functionalities (e.g., response time, throughput, availability, usability, performance, integrity, etc.). In order to effectively provide personalized Web service ranking, it is requi-

[1]http://www.bindingpoint.com/
[2]http://www.grandcentral.com/directory/
[3]http://www.salcentral.com/
[4]http://www.webservicelist.com/

site to consider both functional and non-functional characteristics of Web services. Unfortunately, the Web service search engines mentioned above cannot distinguish the non-functional differences between Web services.

QoS-driven Web service selection is a popular research problem [3, 69, 115]. A basic assumption in the field of selection is that all the Web services in the candidate set share identical functionality. Under this assumption, most of the selection approaches can only differentiate among Web services's non-functional QoS characteristics, regardless of their functionalities. While these QoS-driven selection approaches are directly employed to Web service search engines, several problems will arise. One is that Web services whose functionalities are not exactly equivalent to the user searching query are completely excluded from the result list. Another problem is that Web services in the result list are ordered only according to their QoS metrics, while combining both functional and non-functional attributes is a more reasonable method.

To address the above issues, we propose a new Web service discovering approach by paying respect to functional attributes as well as non-functional features of Web services. A search engine prototype, WSExpress, is built as an implementation of our approach. Experimental results show that our search engine can successfully discover user-interested Web services within top results. In particular, the contributions of this chapter are three-fold:

- Different from all previous work, we propose a brand new Web service searching approach considering both functional and non-functional qualities of the service candidates.
- We conduct a large-scale distributed experimental evaluation on real-world Web services. 3,738 Web services (15,811 operations) located in 69 countries are evaluated both on their functional and non-functional aspects. The evaluation results show that we can recommend high quality Web services to the user.

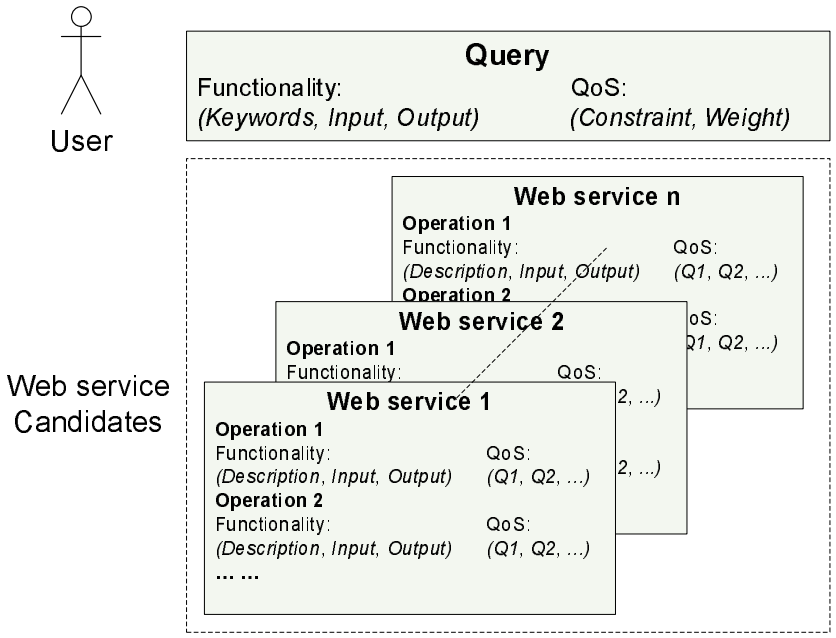


Figure 6.2: Web Service Query Scenario

The precision and recall performance of our functional search is substantially better than the approach in previous work [82].

- We publicly release our large-scale real-world Web service WSDL files and associated QoS datasets[5] for future research. To the best of our knowledge, our dataset is the first publicly-available real-world dataset for functional and non-functional Web service searching research.

The rest of this chapter is organized as follows: Section 6.2 introduces Web service searching backgrounds. Section 6.3 presents the system architecture. Section 6.4 presents our QoS-aware searching approach. Section 6.5 describes our experimental results. Section 6.6 concludes the chapter.

Table 6.1: User Query Examples

User Query	Functionality			QoS	
	Keywords	Input	Output	Constraint (C1, C2, C3)	Weight (W1, W2, W3)
Query 1	car	name, type	price	(0.5, 0.5, 0.2)	(0.4, 0.4, 0.2)
Query 2	weather	city, country	weather	(0.6, 0.3, 0.3)	(0.3, 0.4, 0.3)

Table 6.2: Web Service Examples

Service ID	Operation Name	Input	Output	QoS (Q1, Q2, Q3)
WS 1	CarPrice	name, type	price	(0.8, 0.6, 0.6)
WS 2	AutomobileInformation	name, model	price, color, company	(0.2, 0.4, 0.6)
WS 3	VehicleRecommend	name, model, usage	rent, primecost, provider	(0.6, 0.8, 0.5)

6.2 Motivation

Figure 6.2 shows a common Web service query scenario. A user wants to find an appropriate Web service which contains operations that can be integrated as part of the user's application. The user needs to specify the functionality of a suitable operation by filling the fields of keywords, input and output. Also the user may have some special requirements on service quality, such as the maximum price. These personal requirements can be represented by setting the QoS constraint field. The criticality of different quality criteria for a user can be defined by setting the QoS weight field.

A lot of Web services can be accessed over the Internet. Each service candidate provides one or more operations. Generally, these operations can be described in the structure shown in Figure 6.2. Each operation includes a name, the parameters of input and output elements, and the descriptions about the functionality of this operation as well as the Web services it belongs to in its associated WSDL document. The service quality associated with this operation is represented by several criteria values, e.g., Q1, Q2 in Figure 6.2.

Table 6.1 shows Web service query examples. In query 1, a user wants to find a Web service that can provide appropriate operations for displaying prices of different types and brands of cars. The input

[5]http://wiki.cse.cuhk.edu.hk/user/ylzhang/doku.php?id=icwsdata

information provided by the user for that particular operation is the types and names of cars. This query is structured into three parts: *keywords*, *input* and *output*. The *keywords* part defines in which domain is the query about. In this example, the user concerns about the domain "car". The *input* part contains "name" and "type" since they can be provided by the user. The *output* part is set as "price" to specify the information the user wants to obtain from an appropriate operation.

In Table 6.2 we enumerate three possible results for the user's search query. Web service 1 provides one operation *CarPrice* and this operation's functionality is almost the same as what the user specifies in the query. In addition, the service quality meets the user's requirements. Web service 2 provides operation *Automobile-Information*. Operation *AutomobileInformation* can provide many information details including the price of the automobiles after invoked with "name" and "model" as input. However, some service quality criteria, such as the service price (Q1) and the response time (Q2), are beyond the user's tolerance. Operation *VehicleRecommend* provided by Web service 3 recommends suitable vehicles for the user to rent. Although its target is to suggest the most suitable vehicle and vehicle rental companies to the user, it can also be invoked for obtaining the prices of cars due to the prime cost information provided. Besides, operation *VehicleRecommend*'s service quality fits the user's constraints and preferences quite well. Among these three Web services, the most suitable one is Web service 1, and another acceptable one is Web service 3, but Web service 2 is not highly suggested due to its service quality. Thus, a reasonable order of the recommendation list for the user's query is Web service 1, Web service 3, and Web service 2.

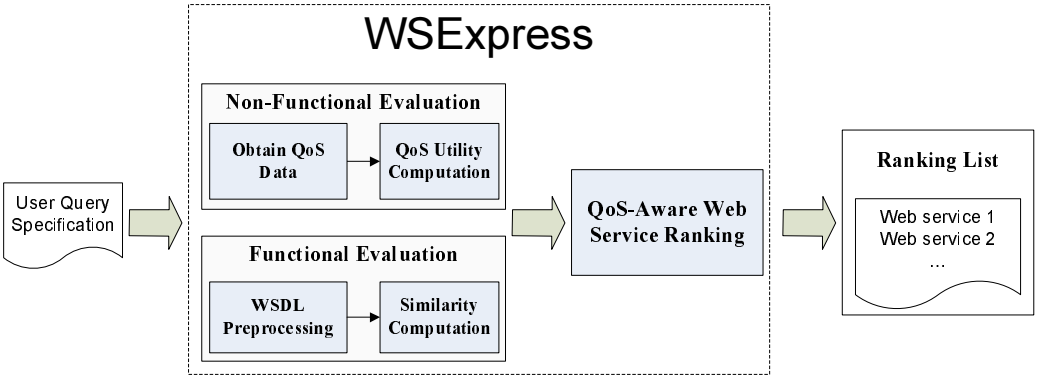


Figure 6.3: System Architecture

6.3 System Architecture

Now we describe the system architecture of our Qos-aware Web service search engine. As shown in Figure 6.3, after accepting a user's query specification, our search engine should be able to provide a practical Web service recommendation list. The search engine consists of three components: non-functional evaluation,ki functional evaluation, and QoS-aware Web service ranking.

There are two phases in the non-functional evaluation component. In phase 1, the search engine obtains QoS criteria values of all the available Web services. In phase 2, the search engine computes the QoS utilities of different Web services according to the constraints and preferences specified in the QoS part of the user's query.

The functional evaluation component contains two phases. In phase 1, the search engine carries out a preprocessing work on the WSDL files associated to the Web services. This work aims at removing noise and improving accuracy of functional evaluation. In phase 2, the search engine evaluates the Web service candidates' functional features. These features are described by similarities between the functionality specified in the query and the functionality of operations provided by those Web services.

Finally, the search engine combines both functional and non-

functional features of Web services in the QoS-aware Web service ranking component. A practical and reasonable Web service recommendation list is then provided as a result to the user's search query.

6.4 QoS-Aware Web Service Searching

6.4.1 QoS Model

In our QoS model we describe the quantitative non-functional properties of Web services as quality criteria. These criteria include generic criteria and business specific criteria. Generic criteria are applicable to all Web services like response time, throughput, availability and price, while business criteria such as penalty-rate are specified to certain kinds of Web services.

By assuming m criteria are employed for representing a Web service quality, we can describe the service quality using a QoS vector $(q_{i,1}, q_{i,2}, \ldots, q_{i,m})$, where $q_{i,j}$ represents the j^{th} criterion value of Web service i.

Some QoS criteria values of Web services, such as penalty rate and price, can be obtained from the service providers directly. However, other QoS attributes' values like response time, availability and reliability need to be generated from all the users' invocation records due to the differences between network environments. In this chapter, we use the approach proposed in [122] to collect QoS performance on real-world Web services.

We put all the Web services' QoS vectors together and form a QoS matrix Q. Each row in Q represents a Web service, while each column represents a QoS criterion value.

$$Q = \begin{pmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,t} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,t} \\ \vdots & \vdots & \vdots & \vdots \\ q_{s,1} & q_{s,2} & \cdots & q_{s,t} \end{pmatrix} \tag{6.1}$$

A utility function is used to evaluate the multi-dimensional quality of a Web service. The utility function maps a QoS vector into a real value for evaluating the Web service candidates. To represent user priorities and preferences, three steps are involved into the utility computation: (1) The QoS criteria values are normalized to enable a uniform measurement of the multi-dimensional quality of service independent of their units and ranges. (3) The weighted evaluation on criteria are carried out for representing user's constraints, preference and special requirements.

Normalization

In this step each criterion value is transformed to a real value between 0 and 1 by comparing it with the maximum and minimum values of that particular criterion. For some criterion the possible absolute value could be very large or infinite. A pair of maximum and minimum values are specified for every criterion respectively. Let $q_{i,u}$ be the upper bound value and $q_{i,l}$ be the lower bound value for the i^{th} criterion, respectively. Every QoS value is transformed according to the following equations:

$$f(x) = \begin{cases} r_{min}, & \text{if } x < r_{min} \\ r_{max}, & \text{if } x > r_{max} \\ x, & \text{otherwise} \end{cases}$$

The normalized value of $q_{i,j}$ can be represented by $q'_{i,j}$ as follows:

$$q'_{i,j} = \frac{q_{i,j} - q_{i,0}}{q_{i,n} - q_{i,0}} \tag{6.2}$$

Thus, the QoS matrix Q is transformed into a normalized matrix Q' as follows:

$$Q' = \begin{pmatrix} q'_{1,1} & q'_{1,2} & \cdots & q'_{1,t} \\ q'_{2,1} & q'_{2,2} & \cdots & q'_{2,t} \\ \vdots & \vdots & \vdots & \vdots \\ q'_{s,1} & q'_{s,2} & \cdots & q'_{s,t} \end{pmatrix} \tag{6.3}$$

Utility Computation

Some Web services need to be excluded from the candidate set due to their inconsistency with the user's QoS constraints. The QoS constraints set the worst quality user can accept.These constraints are usually set according to the application developers' experience or computed by some QoS driven composition algorithm. Web service with any QoS criterion grade unsatisfying user constraint may cause problem while integrated into user's application. For example, if a service fails to return the result within a given period of time, another service may exit with a error code time out while waiting for the result. Assume a user's constraint vector is $C = (c_1, c_2, \ldots, c_m)$, in which c_i sets the minimum normalized i^{th} criterion grade. We will only consider those Web services whose criteria grades are all larger than the constraints. In other words, we delete the rows which fail to satisfy the constraints from Q' and produce a new matrix Q^*:

$$Q^* = \begin{pmatrix} q^*_{1,1} & q^*_{1,2} & \cdots & q^*_{1,t} \\ q^*_{2,1} & q^*_{2,2} & \cdots & q^*_{2,t} \\ \vdots & \vdots & \vdots & \vdots \\ q^*_{s,1} & q^*_{s,2} & \cdots & q^*_{s,t} \end{pmatrix} \tag{6.4}$$

For the sake of simplicity, we only consider positive criteria whose values need to be maximized (negative criteria can be easily transformed into positive attributes by multiplying -1 to their values).

Finally, a weight vector $W = (w_1, w_2, \ldots, w_m)$ is used to represent user's priorities on preferences given to different criteria with $w_k \in \mathbb{R}_0^+$ and $\sum_{k=1}^{m} w_k = 1$. The final QoS utilities vector $U = (u_1, u_2, \ldots)$ of Web service candidates are therefore can be computed as follows:

$$U = Q^* * W^T \tag{6.5}$$

in which u_i is the i^{th} Web service QoS utility value within range $[0, 1]$.

6.4.2 Similarity Computation

Web services provide reusable functionalities. The functionalities are described by the input and output parameters defined in WSDL file.

Now we describe a similarity model for computing similarities between a user query and Web service operations. In this model, a vector $(Keywords, Input, Output)$ is used to represent the functionality part of a user query as well as the functionality part of Web service operations. Particularly, the keywords of a Web service operation are abstracted from the descriptions in its associated WSDL file. Three phases are involved in the similarity search: WSDL preprocessing, clustering parameters and similarity computation.

WSDL Preprocessing

In order to improve the accuracy of similarity computation for operations and user query in our approach, we first need to preprocess the WSDL files. There are two steps as follows:

1. Identify useful terms in WSDL files. Since the descriptions, operation names and input/output parameters' names are made manually by the service provider, there are a lot of misspelled and abbreviated words in real-world WSDL files. This step replace such kind of words with normalized forms.

2. Perform word stemming and remove stopwords. A stem is the basic part of the word that never changes even when morphologically infected. This process can eliminate the difference between inflectional morphemes. Stopwords are those with little substantive meaning.

Similarity Computation

Now we describe how to measure the similarities of Web service operations to a user's query. The functionality part a user's query

R_f consists of three elements $R_f = (r^k, r^{in}, r^{out})$. The *keywords* element is a vector $r^k = (r_1^k, r_2^k, \ldots, r_l^k)$, where r_i^k is the i^{th} keyword. Moreover, the *input* element $r^{in} = (r_1^{in}, r_2^{in}, \ldots, r_m^{in})$ and the *output* element $r^{out} = (r_1^{out}, r_2^{out}, \ldots, r_n^{out})$, where r_i^{in} and r_i^{out} are the i^{th} terms of input element and output element respectively. A Web service operation also consists of three elements $OP_f = (K, In, Out)$. The *keywords* element of operation i is a vector of words $K^i = (k_1^i, k_2^i, \ldots, k_{l'}^i)$. The *input* and the *output* elements are vectors $In^i = (in_1^i, in_2^i, \ldots, in_{m'}^i)$ and $Out^i = (out_1^i, out_2^i, \ldots, out_{n'}^i)$ respectively. Thus, users' queries and Web service operations are described as sets of terms. By applying the TF/IDF (Term Frequency/Inverse Document Frequency) measure [92] into these sets, we can measure the cosine similarity s_i between Web service operation i and a user's query.

Vector Similarity (VS) measures the cosine of the angle between two corresponding vectors and set it as the similarity of the two vectors. In similarity search for Web service, the two vectors measured are Web service operation and user query:

$$s_i = \frac{\sum_{i=1}^{t} r_i \cdot t_i}{\sqrt{\sum_{i=1}^{t} r_i^2} \cdot \sqrt{\sum_{i=1}^{t} t_i^2}} \tag{6.6}$$

Pearson Correlation Coefficient (PCC), another popular similarity measurement apporach, was introduced in a number of recommender systems for similarity computation, since it can be easily implemented and can achieve high accuracy. The similarity between an operation and a user's query can be calculated by employing PCC as follows:

$$s_i = \frac{\sum_{i=1}^{t} (r_i - \bar{r}) \cdot (t_i - \bar{t})}{\sqrt{\sum_{i=1}^{t} (r_i - \bar{r})^2} \cdot \sqrt{\sum_{i=1}^{t} (t_i - \bar{t})^2}} \tag{6.7}$$

where $\bar{r}$ is average TF/IDF value of all terms in a operation vector and $\bar{t}$ is average TF/IDF value of all terms in a user's query vector.

The PCC similarity value s_i is in the interval of -1 and 1 and a larger value means indicates a higher similarity.

6.4.3 QoS-Aware Web Service Searching

With an increasing number of Web services being made available in the Internet, users are able to choose functionally appropriate Web services with high non-functional qualities in a much larger set of candidates than ever before. It is highly necessary to recommend to the user a list of service candidates which fulfill both the user's functional and non-functional requirements.

Utility Computation

A final rating score r_i is defined to evaluate the conformity of each Web service i to achieve the search goal.

$$r_i = \lambda \cdot \frac{1}{\log(p_{s_i} + 1)} + (1 - \lambda) \cdot \frac{1}{\log(p_{u_i} + 1)}, \tag{6.8}$$

where p_{s_i} is the functional rank position and p_{u_i} is the non-functional rank position of Web service i among all the service candidates. Since the absolute values of similarity and service quality indicate different features of Web service and include different units and range, rank positions rather than absolute values is a better choice to indicate the appropriateness of all candidates. $\frac{1}{\log(p+1)}$ calculates the appropriateness value of a candidate in position p for a query. $\lambda \in [0, 1]$ defines how much the functionality factor is more important than the non-functionality factor in the final recommendation.

λ can be a constant to allocate a fixed percentage of the two parts' contributions to the final rating score r_i. However, it is more realistic if λ is expressed as a function of p_{s_i}:

$$\lambda = f(p_{s_i}) \tag{6.9}$$

λ is smaller if the position in similarity rank is lower. This means a Web service is inappropriate if it cannot provide the required functionality to the users no matter how well it serves. The relationship between searching accuracy and the formula of λ will be identified to extend the search engine prototype in our future work.

Rank Aggregation

After receiving the users' query, the functional component of WSExpress computes the similarity s_i in Section 6.4.2 between search query R_f and operations of Web service i, while the non-functional component of WSExpress employs R_q to compute the QoS utility u_i in Section 6.4.1 of each Web service i.

Now our goal is to consider user's preferences on both functional and non-functional features and provide a rank list by combing evaluation results of the two aspects of service candidates. Given the user's preference on functional and non-functional aspect, we can provide a personalized rank list by assigning each service candidate a certain score based on its positions in similarity ranking and QoS utility ranking. In other words, we aggregate the rankings of similarity and QoS utility according user defined preference.

We formally describe the optimal rank aggregation problem in the following. Given a set $S = \{s_1, s_2, \cdots\}$ of service candidates, an ranking list $l =< l(1), l(2), \cdots >$ is an permutation of all service candidates, where $l(i)$ denotes the service at position i of l. Given two ranking lists l_p, l_q of similarity and QoS utility, respectively, the optimal rank list l_o, which is an aggregation of l_p and l_q should be recommended to users.

Given the similarity values or QoS utility scores of candidates, we assume that there is an uncertainty of ranking list l_p or l_q. In other words, any service $s_j \in S$ is assumed to be possible for ranked in the top position of l. But different services may have different likelihood values. Under this assumption, we define the top one probability of

Web service s_j as follows:

$$P(s_j) = \frac{f(r_j)}{\sum_{k=1}^{m} f(r_k)}, \tag{6.10}$$

where $f(x)$ can be any monotonically increasing and strictly positive function, $P(s_j) > 0$ and $\sum P(s_j) = 1$. For simplicity, we take the exponential function for $f(x)$ [14]. Note that the top one probabilities $P(s_j)$ form a probability distribution over the set of services S. The top one probability indicates the probability of a service being ranked in the top position of a user's ranking list. By Eq. (6.10), a Web service with high similarity value or QoS utility value is assigned to a high probability value.

In order to estimate the quality of recommended Web service list, we need to define the distance between two ranking lists []. Ranking list distance evaluate the similarity of two lists. A distance value is smaller if more items are ordered in the similar positions. Given two ranking list l_1 and l_2 over the Web service set S, the distance between l_1 and l_2 is defined by:

$$d(l_1, l_2) = -\sum_{j=1}^{m} P(s_{1j})P(s_{2j}), \tag{6.11}$$

where $s_1 j$ is the service in the $j^t h$ position of l_1 and $s_2 j$ is the service in the $j^t h$ position of l_2.

We therefore define the Web service recommendation as the following optimization problem:

$$\min_{l_o} \mathcal{L}(l_p, l_q) = \lambda d(l_o, l_p) + (1 - \lambda) d(l_o, l_q), \tag{6.12}$$

where $d(l_o, l_p)$ is the distance between the optimal ranking list and the functionality ranking list, $d(l_o, l_q)$ is the distance between the optimal ranking list and the non-functionality ranking list, λ controls the trade off between functionality and non-functionality.

Intuitively, Web services recommended by the final ranking list are functional comply with the users' requirements and with high

QoS level. Our goal is to find a rank of all candidate in U that minimize the objective value function Eq. 6.12. One possible approach to solve the problem is check all the possible ranking lists in the solution space and select the optimal ranking which minimize the objective value function Eq. 6.12. The size of solution space is $O(n!)$ for n candidates. In fact, this is a NP-complete problem, which can be proved by transforming into a NP-complete problem of finding minimum cost perfect matching in the bipartite graph. Therefore we propose a greedy algorithm to find an suboptimal solution as follows:

Algorithm 5: Greedy Rank Aggregation

Input: a candidate set *S*, two ranking lists l_p and l_q
Output: a optimal rank aggregation l_o^*

for *each service* s_j *in* S **do**
  $P_1(s_j) = \frac{f(u_j)}{\sum_{k=1}^{m} f(u_k)}$;
  $P_2(s_j) = \frac{f(sim_j)}{\sum_{k=1}^{m} f(sim_k)}$;
  $AP(s_j) = \lambda P_1 s_j + (1-\lambda) P_2(s_j)$;
end
Generate a ranking list l_o^* of all the service candidates according to their AP values;
$d_{l_o^*, l_p} = -\sum_{j=1}^{m} P(s_{pj}) P(s_{oj})$;
$d_{l_o^*, l_q} = -\sum_{j=1}^{m} P(s_{qj}) P(s_{oj})$;
$\mathcal{L}^*(l_p, l_q) = \lambda d(l_o^*, l_p) + (1-\lambda) d(l_o^*, l_q)$;
for *each candidate* s *in* S **do**
  change the position of s higher or lower;
  $d_{l_o, l_p} = -\sum_{j=1}^{m} P(s_{pj}) P(s_{oj})$;
  $d_{l_o, l_q} = -\sum_{j=1}^{m} P(s_{qj}) P(s_{oj})$;
  $\mathcal{L}(l_p, l_q) = \lambda d(l_o, l_p) + (1-\lambda) d(l_o, l_q)$;
  if $\mathcal{L}^*(l_p, l_q) < \mathcal{L}(l_p, l_q)$ **then**
    $l_o^* = l_o$;
  end
end
return l_o^*;

6.4.4 Online Ranking

In this section we propose an online service recommendation algorithm. Since the QoS performance of Web services are dynamic at runtime, the ranking list should adopt the updated QoS information. Therefore, the Optimal Rank Algorithm is extended to integrate QoS information dynamically. In our ranking aggregation approach, a nice property is that before aggregation the functional utility and nonfunctional utility are calculated independently. For functional similarity search, the ranking list remains the same in different time intervals. The QoS ranking list is changing from time to time. Therefore, the optimal recommendation list should be adopted to the new QoS value accordingly. The Online Service Recommendation Algorithm is described as follows:

Algorithm 6: Online Service Recommendation

Input: a candidate set *S*, an optimal ranking list l_o, functional ranking lists l_p, a new QoS matrix Q

Output: a new optimal rank aggregation l_o^*

Conduct normalization on the new QoS matrix Q according to 6.2;
Compute the QoS utility vector U according to 6.5;
for *each service* s_j *in* S **do**
    $P_2(s_j) = \frac{f(sim_j)}{\sum_{k=1}^{m} f(sim_k)}$;
end
$l_o^* = l_o$;
for *each candidate* s *in* S **do**
    change the position of s higher or lower;
    $d_{l_o,l_p} = -\sum_{j=1}^{m} P(s_{pj})P(s_{oj})$;
    $d_{l_o,l_q} = -\sum_{j=1}^{m} P(s_{qj})P(s_{oj})$;
    $\mathcal{L}(l_p, l_q) = \lambda d(l_o, l_p) + (1-\lambda)d(l_o, l_q)$;
    if $\mathcal{L}^*(l_p, l_q) < \mathcal{L}(l_p, l_q)$ **then**
        $l_o^* = l_o$;
    end
end
return l_o^*;

6.4.5 Application Scenarios

Searching Styles

To attack the above problem, we propose a novel search engine which can provide the user with brand new searching styles. We define a user search query in the form of a vector $R = (R_f, R_q)$, which contains functionality part R_f and non-functionality part R_q for representing the user's ideal Web service candidate. $R_q = (C, W)$ defines the user's nonfunctional requirements, where C and W set the user's constraints and preferences on QoS criteria separately as mentioned in Section 6.4.1. Our new searching procedure consists of three styles in the following discussion.

Keywords Specified In this searching style, the user only needs to simply enter the keywords vector r^k and QoS requirements R_q. The keywords should capture the main functionality the user requires in the search goal. In Table 6.1 as an example, since the user needs price information of cars, it is reasonable to specify "car" or "car, price" as the keywords vector.

Interface Specified In order to improve the searching efficiency, we design the "interface specified" searching style. In this style, the user specifies the expected functionality by setting the input vector r^{in} and/or output vector r^{out} as well as QoS requirements R_q. The input vector r^{in} represents the largest amount of information the user can provide to the expected Web service operation, while the output vector represents the least amount of information that should be returned after invoking the Web service operation.

Similar Operations For a more accurate and advanced Web service searching, we design the "similar operation" searching style by combining above two styles. This style is especially suitable in the following two situations. In the first situation, the user has already received a Web service recommendation list by performing one of the above searching styles. The user decides the Web service to explore in detail, checks the inputs and outputs of its operations,

and even tries some of the operations. After carefully inspecting a Web service the user may find that this Web service is not suitable for the applications. However, the user does not want to repeat the time-consuming inspecting process for other service candidates. This style enables the user to find similar Web service operations by only modifying a small part of the previous query to exclude these inappropriate features. In the second situation, the user already integrates a Web service into the application for a particular functionality. However, due to some reason this web service becomes unaccessible. Without requesting an extra query process, the search engine can automatically find other substitutions.

Now we discuss in detail how the functional evaluation component operates in different scenarios.

- If only the keywords vector in the functionality part of the user query is defined, the similarity is computed in Section 6.4.2 using the keywords vector r^k of the query and the keywords vector K extracted from the descriptions, operation names, and parameter names.

- If the input and output vectors in the functionality part of the user query are defined, the input similarity and output similarity are computed in Section 6.4.2 using the input/output vector r^{in}/r^{out} of the query and the input/output vector In/Out of an operation. The functional similarity is a combination of input and output similarities.

- If the whole functionality part of a query is available. The functional similarity of an operation is a combination of the above two kinds of similarities, which is computed using R_f and OP_f.

6.5 Experiments

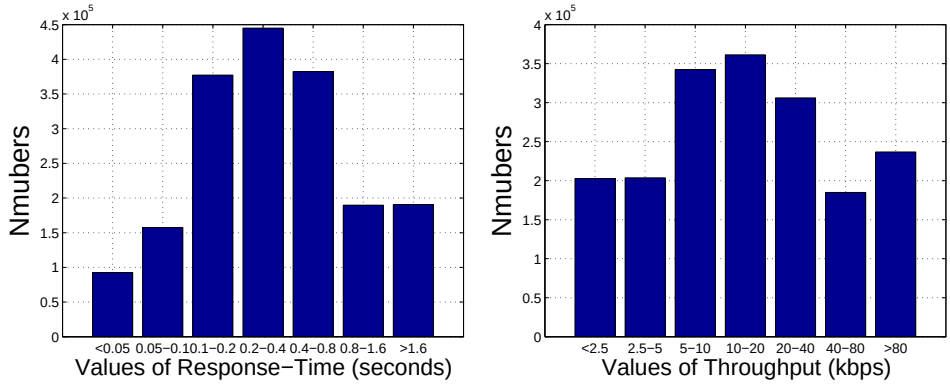
The aim of the experiments is to study the performance of our approach compared with other approaches (e.g., the one proposed by [82]). We conduct two experiments in Section 6.5.1 and Section 6.5.2, respectively. Firstly, we show that the top-k Web services returned by our approach have much more QoS gain than other approaches. Secondly, we demonstrate that our approach can achieve highly relevant results as good as other similarity based service searching approaches even there is no available QoS values.

6.5.1 QoS Recommendation Evaluation

In this section, we conduct a large-scale real-world experiment to study the QoS performance of the top-k Web services returned by our searching approach.

To obtain real-world WSDL files, we developed a Web crawling engine to crawl WSDL files from different Web resources (e.g., UDDI, Web service portal, and Web service search engine). We obtain totally 3,738 WSDL files from 69 countries. Totally 15,811 operations are contained in these Web services. To measure the non-functional performance of these Web services, 339 distributed computers in 30 countries from Planet-lab [6] are employed to monitor these Web services. The detailed non-functional performance of Web service invocations are recorded by these service users (distributed computer nodes). Totally 1,267,182 QoS performance results are collected. Each invocation record is a k dimension vector representing the QoS values of k criteria. For simplicity, we use two matrices, which represent response-time and throughput QoS criteria respectively, for experimental evaluation in this chapter. Without loss of generality, our approach can be easily extended to include more QoS criteria.

[6] http://www.planet-lab.org



(a) (b)

Figure 6.4: Value Distributions

Table 6.3: Statistics of WS QoS Dataset

Statistics	Response-Time	Throughput
Scale	0-20s	0-1000kbps
Mean	0.910s	47.386kbps
Num. of Users	339	339
Num. of Web Services	3,738	3,738
Num. of Records	1,267,182	1,267,182

The statistics of Web service QoS dataset are summarized in Table 6.3. Response-time and throughput are within the range 0-20 seconds and 0-1000 kbps respectively. The means of response-time and throughput are 0.910 seconds and 47.386 kbps respectively. Figure 6.4 shows the distributions of response-time and throughput. Most of the response-time values are between 0.1-0.8 seconds and most of the throughput values are between 5-40 kbps.

In most of the searching scenarios, users tend to look at only the top items of the returned result list. The items in the higher position, especially the first position, is more important than the items in lower positions in the returned result list. To evaluate the qualities of top-k returned results in a ranked list, we employ the Normalized Discounted Cumulative Gain (NDCG), a standard IR measure [53]

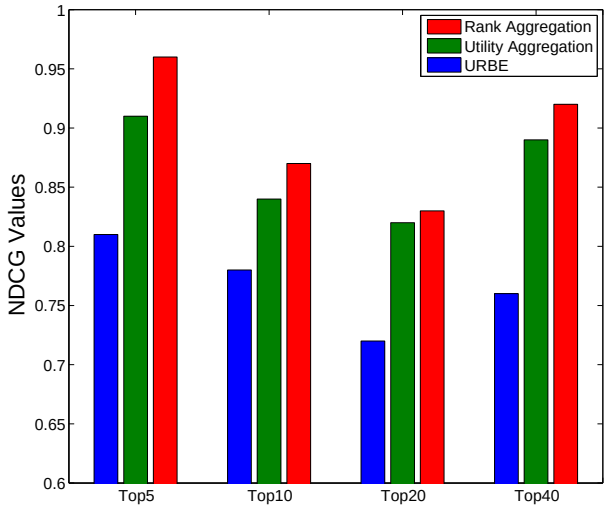


Figure 6.5: NDCG of Top-K Web services

approach as performance evaluation metric. Let $s_1, s_2, \ldots, s_p$ be a ranked list of Web services produced by a searching approach. Let u_i be the associated QoS utility value of Web service s_i, which ranked in position p_i. Discounted Cumulative Gain (DCG) and NDCG of at rank k are defined respectively as:

$$DCG_k = u_i + \sum_{i=2}^{k} \frac{u_i}{\log_2 p_i}, \tag{6.13}$$

$$NDCG_k = \frac{DCG_k}{IDCG_k} \tag{6.14}$$

where IDCG is the maximum possible gain value that is obtained with the optimal re-order of k Web services in the list $s_1, s_2, \ldots, s_p$. For example, consider the following QoS utility values which are ordered according to the position of associated Web services in a ranked Web service list:

$u = [0.3, 0.2, 0.3, 0, 0, 0.1, 0.2, 0.2, 0.3, 0]$

The perfect ranking would have QoS utility values of each rank of

$u = [0.3, 0.3, 0.3, 0.2, 0.2, 0.2, 0.1, 0, 0, 0]$

which would give ideal DCG utility values.

Table 6.4: NDCG values (A larger NDCG value means a better performance)

Domain	Query ID	Top5		Top10		Top20		Top40	
		URBE	WSExpress	URBE	WSExpress	URBE	WSExpress	URBE	WSExpress
Business	1	0.437	0.661	0.444	0.599	0.439	0.633	0.527	0.659
	2	0.653	0.653	0.668	0.721	0.657	0.666	0.634	0.645
	3	0.402	0.502	0.456	0.512	0.502	0.544	0.574	0.603
	4	0.200	0.767	0.303	0.697	0.399	0.667	0.496	0.699
Education	5	0.603	0.742	0.604	0.753	0.598	0.664	0.631	0.717
	6	0.621	0.732	0.571	0.715	0.574	0.675	0.598	0.696
	7	0.645	0.688	0.579	0.671	0.560	0.643	0.632	0.662
	8	0.509	0.642	0.562	0.642	0.575	0.633	0.600	0.672
Science	9	0.423	0.538	0.478	0.549	0.495	0.572	0.502	0.578
	10	0.573	0.731	0.525	0.717	0.546	0.693	0.602	0.702
	11	0.632	0.819	0.613	0.823	0.583	0.757	0.628	0.774
	12	0.622	0.754	0.593	0.728	0.582	0.681	0.597	0.734
Weather	13	0.214	0.574	0.245	0.551	0.243	0.559	0.259	0.581
	14	0.713	0.825	0.701	0.814	0.687	0.802	0.725	0.824
	15	0.431	0.581	0.346	0.566	0.465	0.566	0.530	0.606
	16	0.475	0.611	0.485	0.519	0.501	0.529	0.525	0.543
Media	17	0.409	0.516	0.419	0.485	0.403	0.496	0.589	0.530
	18	0.393	0.519	0.373	0.488	0.450	0.527	0.532	0.567
	19	0.544	0.740	0.554	0.683	0.512	0.642	0.551	0.683
	20	0.504	0.678	0.473	0.613	0.451	0.559	0.497	0.602

To study the performance of our approach, we compared our WSExpress Web service searching engine with the URBE [82], a keywords matching approach, employing our real-world dataset described above. Totally 5 query domains are studied in this experiment. Each domain contains 4 user queries. Figure 6.5 shows the NDCG values of top-k recommended Web services. The top-k NDCG values of our WSExpress engine are considerably higher than URBE (i.e., 0.767 of WSExpress compared with 0.200 of URBE for Top5 and 0.697 of WSExpress compared with 0.303 of URBE for Top10). This means that, given a query, our search engine can recommend high quality Web services in the first positions.

Table 6.4 shows the NDCG values of top-k recommended Web services in the five domains. In most of the queries, NDCG values of WSExpress are much higher than URBE. In some search scenarios such as query 2, the NDCG values of WSExpress and URBE for Top5 are identical, since in this particular case the most functional appropriate Web services have the most appropriate non-functional

properties. In other words, these Top5 Web services have highest QoS utilities and similarity values. However, while more top Web services are considered, such as Top10, the NDCG values of WSExpress are becoming much higher than URBE.

6.5.2 Functional Matching Evaluation

In this experiment, we study the relevance of the recommended Web services to the user's query without considering non-functional performance of the Web services. By comparing our approach with URBE, we observe that the top-k Web services in our recommendation list are highly relevant to the user's query even without any available QoS values.

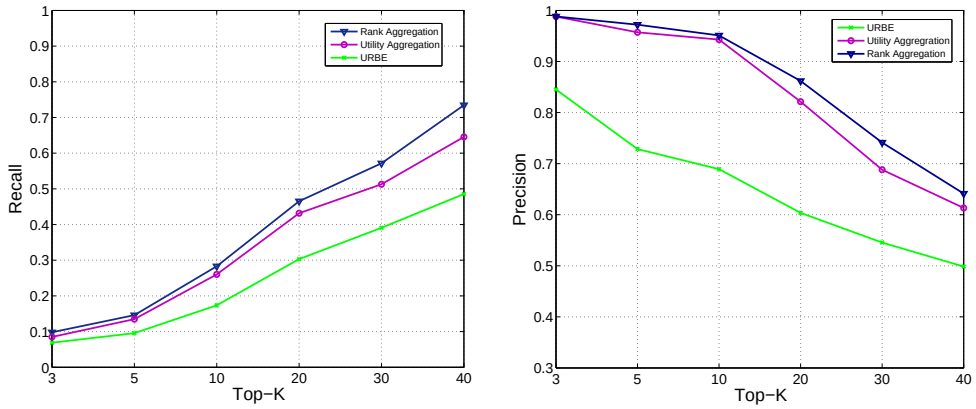
The benchmark adopted for evaluating the performance of our approach is the OWL-S service retrieval test collection OWLS-TC v2 [60]. This collection consists of more than 570 Web services and 1,000 operations covering seven application domains (i.e., education, medical care, food, travel, communication, economy, and weaponry). The benchmark includes WSDL files of the Web services, 32 test queries, and a set of relevant Web services associated to each of the queries. Since the QoS feature is not considered in this experiment, we set the QoS utility value of each Web service as 1.

Top-k recall ($Recall_k$) and *top-k precision* ($Precision_k$) are adopted as metrics to evaluate the performance of different Web search approaches. $Recall_k$ and $Precision_k$ can be calculated by:

$$Recall_k = \frac{|Rel \cap Ret_k|}{|Rel|}, \tag{6.15}$$

$$Precision_k = \frac{|Rel \cap Ret_k|}{|Ret_k|}, \tag{6.16}$$

where Rel is the relevant set of Web services for a query, and Ret_k is a set of top-k Web services search results.



(a) (b)

Figure 6.6: Recall and Precision Performance

Since user tends to check only top few Web services in common search scenario, an approach with high top-k precision values is very practical in reality. Figure 6.6 shows the experimental results of our WSExpress approach and the URBE approach. In Figure 6.6(a), the top-k recall values of WSExpress are higher than URBE. In Figure 6.6(b), the top-k precision values of WSExpress are considerably higher than URBE, indicating that more relevant Web services are recommended in high positions by our approach.

6.5.3 Online Recommendation

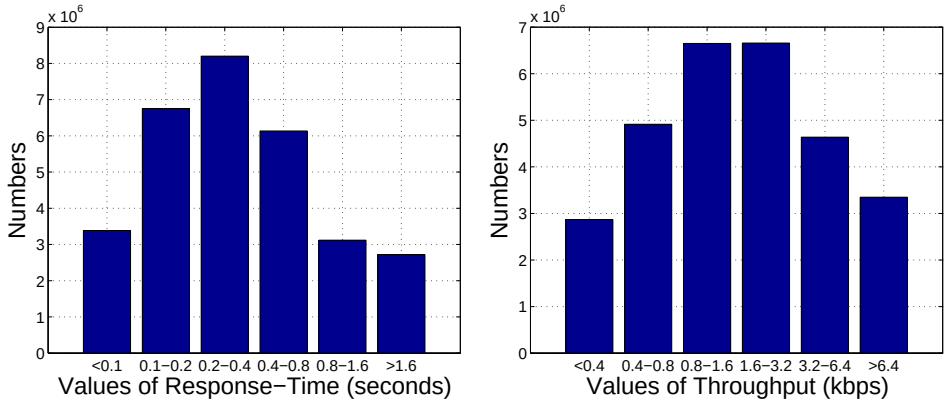
In this chapter, we propose an online Web service recommendation approach. Different from the previous ranking approach, it adopts the real time QoS information to recommend Web services. In this section, we evaluate the performance of online recommendation approaches.

In this experiment we deploy 142 distributed computers located in 22 countries from PlanetLab. Totally, 4,532 publicly available real-world Web services from 57 countries are monitored by each computer continuously. In our experiment, each of the 142 computers sends null operation requests to all the 4,532 Web services during

Table 6.5: Statistics of Online QoS Dataset

Statistics	Response-Time	Throughput
Scale	0-20s	0-1000kbps
Mean	3.165s	9.609kbps
Num. of Users	142	142
Num. of Web Services	4,532	4,532
Num. of Time Intervals	64	64
Num. of Records	30,287,611	30,287,611

every time interval. The experiment lasts for 16 hours with a time interval lasting for 15 minutes. By collecting invocation records from all the computers, finally we include 30,287,611 QoS performance results into the Web service QoS dataset. Each invocation record is a k dimension vector representing the QoS values of k criteria. We then extract a set of $142 \times 4532 \times 64$ user-service-time tensors, each of which stands for a particular QoS property, from the QoS invocation records. For simplicity, we employ two tensors, which represent response-time and throughput QoS criteria respectively, for experimental evaluation in this chapter. Without loss of generality, our approach can be easily extended to include more QoS criteria.



(a) (b)

Figure 6.7: QoS Value Distributions of Online Dataset

The statistics of Web service QoS dataset are summarized in Ta-

ble 6.3. Response-time and throughput are within the range of 0-20 seconds and 0-1000 kbps respectively. The means of response-time and throughput are 3.165 seconds and 9.609 kbps respectively. The distributions of the response-time and throughput values of the user-service-time tensors are shown in Figure 6.5.1 and Figure 6.5.1 respectively. Most of the response-time values are between 0.1-0.8 seconds and most of the throughput values are between 0.8-3.2 kbps.

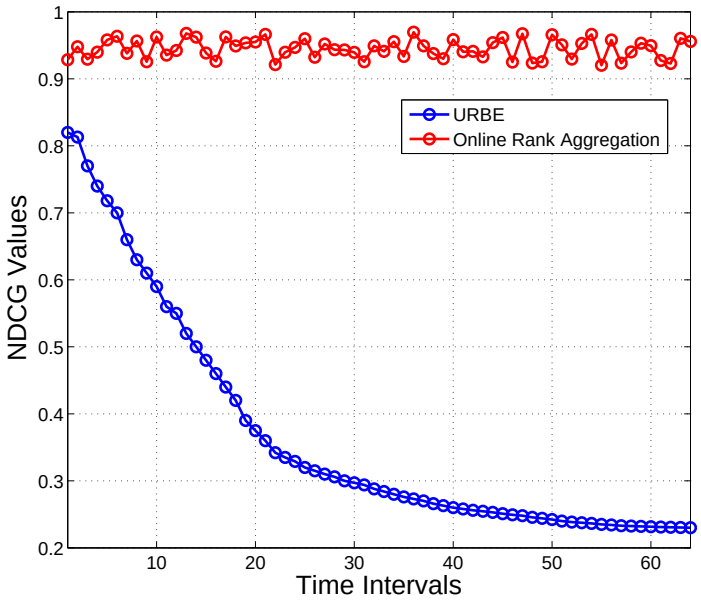
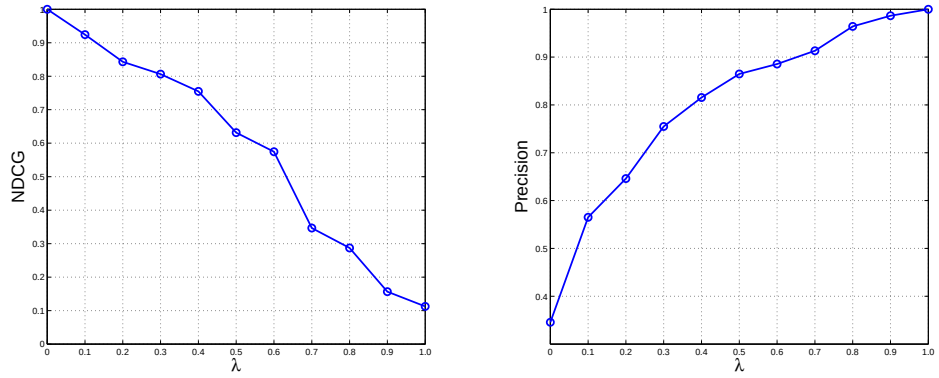


Figure 6.8: NDCG of Online Recommendation

The experimental results are shown in Figure 6.8. Each time interval lasts for 15 minutes. The parameter setting is Top-K=5. From Figure 6.8, we observe that in each time interval, online recommendation approach has a higher NDCG value than URBE, which means Web services with better QoS performance are recommended compared with URBE. Since URBE cannot adopt the dynamic QoS information for recommendation in time, the NDCG values of approach URBE decreases significantly as the time passed. After about 30 time intervals, the NDCG value is below 0.3 which means QoS performance of the recommended Web services has a high probability that cannot fulfill the users' non-functional requirements. In our



(a) (b)

Figure 6.9: Impact of λ

online rank aggregation approach, we employ the latest QoS information of Web services for recommendation. Therefore, the NDCG values are maintained in a high level, which indicates that we can always recommend appropriate Web services with high QoS performance to the users.

6.5.4 Impact of λ

In our method, the parameter λ controls the user's preference on functionality and non-functionality. A larger value of λ means functionality is preferred. In Figure 6.9, we study the impact of λ by varying the the values of $lambda$ from 0 to 1 with a step value of 0.1. Other parameter settings are Top-K=10.

Figure 6.9(a) shows the NDCG values and Figure 6.9(b) shows the Precision values. From Figure 6.9(a), we observe that λ impacts the NDCG performance significantly, which demonstrates that incorporating the QoS information greatly improves the non-functional quality of recommended Web services. In general, when the value of λ is increased from 0 to 1, the NDCG value is decreased. This observation indicates that if functionality is preferred, the QoS performance of recommended Web services is decreased. If $\lambda = 0$,

we only employ the QoS information for Web service recommendation, therefore the NDCG value is 1. If $\lambda = 1$, we only employ the functional similarity information for Web service recommendation, therefore the NDCG value is very small. From Figure 6.9(b), we observe that λ also impacts the precision significantly, which demonstrates that incorporating the functional similarity information greatly improves the recommendation accuracy. In general, when the value of λ is increased from 0 to 1, the precision value is increased. This observation indicates that if functionality is preferred, the functional requirements can be fulfilled well. If $\lambda = 0$, we only employ the QoS information for Web service recommendation, therefore the precision value is vary small. If $\lambda = 1$, we only employ the functional similarity information for Web service recommendation, therefore the precision value is 1. In other cases, we fuse the information of QoS and functionality for Web service recommendation.

A proper value of λ is highly related to the preference of the user. The user defines the importance of functionality and non-functionality. A proper value of λ can be defined by analyzing the impact of λ on a small sample dataset.

6.6 Summary

In this chapter we present a novel Web service search engine WSExpress to find the desired Web service. Both functional and non-functional characteristics of Web services are captured in our approach. We provide user three searching styles in the WSExpress to adapt different searching scenarios. A large-scale real-world experiment in distributed environment and a experiment on benchmark OWLS-TC v2 are conducted to study the performance of our search engine prototype. The results show that our approach outperforms related works.

In future work, we will conduct data mining in our dataset to

identify for which formulas of λ our search approach can achieve optimized performance. Clustering algorithms for similarity computation will be designed for improving functional accuracy of searching result. Finally, the non-functional evaluation component will be extended to dynamically collect quality information of Web services.

□ **End of chapter.**

Chapter 7

QoS-Aware Byzantine Fault Tolerance

7.1 Overview

Cloud computing [6, 25] is Internet-based computing, whereby shared resources, software, and information are provided to computers and other devices on demand [44]. Currently, most of the clouds are deployed on two kinds of infrastructures. One is well-provisioned and well-managed infrastructure [102] managed by a large cloud provider (e.g., Amazon, Google, Microsoft, IBM, etc.). The other one is voluntary-resource infrastructure which consists of numerous user-contributed computing resources [21]. With the exponential growth of cloud computing as a solution for providing flexible computing resource, more and more cloud applications emerge in recent years. How to build high-reliable cloud applications, which are usually large-scale and very complex, becomes an urgent and crucial research problem.

Typically, cloud applications consist of a number of cloud modules. The reliability of cloud applications is greatly influenced by the reliability of cloud modules. Therefore, building high-reliable cloud modules becomes the premise of developing high-reliable cloud applications. Traditionally, testing schemes [73] are conducted on the software systems of cloud modules to make sure that the reliability

threshold has been achieved before releasing the software. However, reliability of a cloud module not only relies on the system itself, but also heavily depends on the node it has deployed and the unpredictable Internet. Traditional testing has limited improvement on the reliability of a cloud module under voluntary-resource cloud infrastructure due to:

- Computing resources, denoted as nodes in the cloud, are frangible. Different from the powerful and performance-guaranteed nodes managed by large cloud providers, user-contributed nodes are usually highly dynamic, much cheaper, less powerful, and less reliable. The reliability of a cloud module deployed on these nodes is mainly determined by the robustness of nodes rather than the software implementation.

- Communication links between modules are not reliable. Unlike nodes in well-provisioned cloud infrastructure, which are connected by high speed cables, nodes in voluntary-resource cloud infrastructure are usually connected by unpredictable communication links. Communication faults, such as time out, will greatly influence the reliability of cloud applications.

Based on the above analysis, in order to build reliable cloud applications on the voluntary-resource cloud infrastructure, it is extremely urgent to design a fault tolerance mechanism for handling different faults. Typically, the reliability of cloud applications is effected by several types of faults, including: node faults like crashing, network faults like disconnection, Byzantine faults [62] like malicious behaviors (i.e., sending inconsistent response to a request [109]), etc. The user-contributed nodes, which are usually cheap and small, makes malicious behaviors increasingly common in voluntary-resource cloud infrastructure. However, traditional fault tolerance strategies cannot tolerate malicious behaviors of nodes.

To address this critical challenge, we propose a novel approach, called Byzantine Fault Tolerant Cloud (BFTCloud), for tolerating

different types of failures in voluntary-resource clouds. BFTCloud uses replication techniques for overcoming failures since a broad pool of nodes are available in the cloud. Moreover, due to the different geographical locations, operating systems, network environments and software implementation among nodes, most of the failures happened in voluntary-resource cloud are independent of each other, which is the premise of Byzantine fault tolerance mechanism. BFTCloud can tolerate different types of failures including the malicious behaviors of nodes. By making up a BFT group of one primary and $3f$ replicas, BFTCloud can guarantee the robustness of systems when up to f nodes are faulty at run-time. The experimental results show that compared with other fault tolerance approaches, BFTCloud guarantees high reliability of systems built on the top of voluntary-resource cloud infrastructure and ensures good performance of these systems.

In summary, this chapter makes the following contributions:

1. We identify the Byzantine fault tolerance problem in voluntary-resource cloud and propose a Byzantine fault tolerance framework, named BFTCloud, for guaranteeing the robustness of cloud application. BFTCloud uses dynamical replication techniques to tolerate various types of faults including Byzantine faults. We consider BFTCloud as the first Byzantine Fault Tolerant Framework in cloud computing literature.

2. We have implemented the BFTCloud system and test it on a voluntary-resource cloud, Planet-lab [1], which consists of 257 user-contributed computing resources distributed in 26 countries. The prototype implementation indicates that BFTCloud can be easily integrated into cloud nodes as a middleware.

3. We conduct large-scale real-world experiments to study the performance of BFTCloud on reliability improvement compared with other approaches. The experimental results show

[1] http://www.planet-lab.org

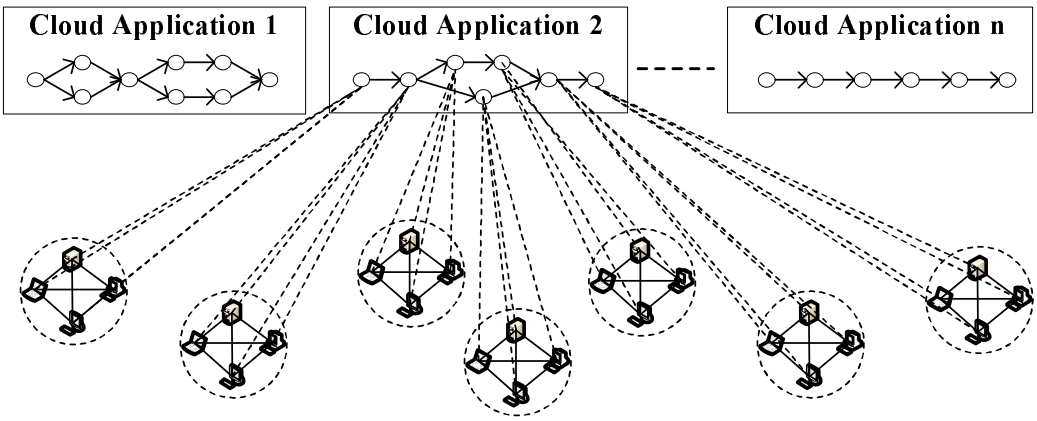


Figure 7.1: Architecture of Cloud Applications

the effectiveness of BFTCloud on tolerating various types of faults in cloud.

The rest of this chapter is organized as follows: Section 7.2 describes the system architecture of BFTCloud. Section 7.3 presents our BFTCloud fault tolerate mechanism in detail. Section 7.4 introduces the experimental results. Section 7.5 concludes the chapter.

7.2 System Architecture

We begin by using a motivating example to show the research problem in this chapter. As shown in Figure 7.1, cloud applications usually consist of a number of modules. These modules are deployed on distributed cloud nodes and connected with each other through communication links. Each module is supposed to finish a certain task (e.g., product selection, bill payment, shipping addresses confirming, etc.) for a cloud application (e.g., shopping agency, etc.). A cloud module will form a sequence of requests (e.g., browsing products, choosing products, etc.) for the task (e.g, product selection, etc.) and send the requests to a group of nodes in the voluntary-resource cloud for execution.

Figure 7.2 shows the system architecture of BFTCloud in voluntary-

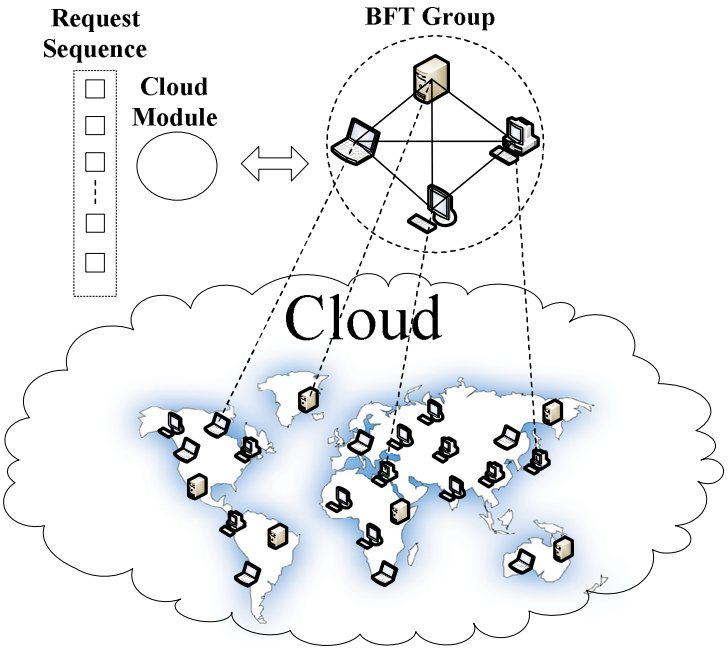


Figure 7.2: Architecture of BFTCloud in Voluntary-Resource Cloud

resource cloud environment. Under the voluntary-resource cloud infrastructure, end-users contribute a larger number of computing resources which can be provided to cloud modules for request execution. Typically, computing resources in the voluntary-resource cloud are heterogeneous and less reliable, and malicious behaviors of resource providers cannot be prevented. Byzantine faults could be very common in a user-contributed cloud environment. In order to guarantee the robustness of the module, the replication technique is employed for request execution upon the user-contributed nodes. After a cloud module generated a sequence of requests, it first needs to choose a BFT group from the pool of cloud nodes for request execution. Since cloud nodes are located in different geographic locations with heterogeneous network environments, and the failure probabilities of nodes are diverse, a monitor is implemented on the cloud module side as a middleware for monitoring the QoS performance and failure probability of nodes. By considering the QoS performance and failure probability, the cloud module first chooses

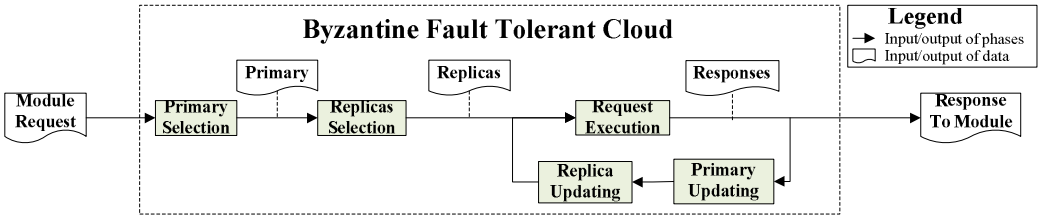


Figure 7.3: Work Procedures of BFTCloud

a node as primary and send the current request to the primary. After that, a set of replicas are selected according to their failure probability and QoS performance to both the cloud module and the primary. The primary and replicas form a BFT group for executing requests from the cloud module. After the BFT group returns responses to the current request, the cloud module will judge whether the responses can be committed. Then the cloud module will send the next request or resend the current request to the BFT group. If some nodes of the BFT group are identified as faulty, the cloud module will update the BFT group to guarantee the system reliability. The detailed approach will be presented in Section 7.3.

7.3 System Design

In this section, we present BFTCloud, a practical framework for building robust systems with Byzantine fault tolerance under voluntary-resource cloud infrastructure. We first give an overview on the work procedures of BFTCloud in Section 7.3.1. Then we describe the five phases of BFTCloud in Section 7.3.2 to Section 7.3.6 respectively.

7.3.1 System Overview

Figure 7.3 shows the work procedures of BFTCloud. The input of BFTCloud is a sequence of requests with specified QoS requirements (e.g., preferences on price, capability, bandwidth, workload, response latency, failure probability, etc.) sent by the cloud mod-

ule. The output of BFTCloud is a sequence of committed responses corresponding to the requests. BFTCloud consists of five phases described as follows:

1. **Primary Selection:** After accepting a request from the cloud module, a node is selected from the Cloud as the primary. The primary is selected by applying the primary selection algorithm with respect to the QoS requirements of the request.

2. **Replica Selection:** In this phase, a set of nodes are selected as replicas by applying a replica selection algorithm with respect to the QoS requirements of the request. The primary then forwards the request to all replicas for execution. The selected replicas together with the primary make up a BFT group.

3. **Request Execution:** In this phase, all members in the BFT group execute the request locally and send back their responses to the cloud module. After collecting responses from the BFT group within a certain period of time, the cloud module will judge the consistency of responses. If the BFT group respond consistently, the current request will be committed and the cloud module will send the next request. If the BFT group responds inconsistently, the cloud module will trigger a fault tolerance procedure to tolerate up to f faulty nodes and trigger the primary updating procedure and/or replica updating procedure to update the group members. If more than f nodes are identified as faulty, the cloud module will resend the request to the refresh BFT group and enter into the request execution phase again.

4. **Primary Updating:** In this phase, faulty primary in the BFT group will be identified and replaced by a newly selected primary.

5. **Replica Updating:** In this phase, faulty replicas in the BFT group will be identified and updated according to the informa-

tion obtained from the request execution phase. The replica updating algorithm will be applied to replace the faulty replicas with other suitable nodes in the cloud.

7.3.2 Primary Selection

Under the voluntary-resource cloud infrastructure, a cloud module will send the request directly to a node which it believes to be the primary. Therefore, the primary plays a important role in a BFT group. The responsibilities of primary include: accepting requests from the cloud module, selecting appropriate replicas to form a BFT group, forwarding the request to all replicas, and replacing faulty replicas with newly selected nodes. Since failures happened on primary will greatly decrease the overall performance of a BFT group, the requirements on primary attributes (e.g., capability, bandwidth, workload, etc.) are more strict than those on replicas. In order to select an optimal primary, we propose a primary selection algorithm.

We model the primary selection problem under voluntary-resource cloud infrastructure as follows:

> *Let N be the set of nodes available in the cloud and Q be the set of m dimension vectors. For each node n_i in N, there is a $q_i = (q_{i1}, q_{i2}, \cdots, q_{im})$ in Q representing the QoS values of m criteria. Given a priority vector $W = (w_1, w_2, \cdots, w_m)$ on the m QoS criteria, the optimal primary should be select from the set N.*

Note that $w_k \in \mathbb{R}^+$ and $\sum_{k=1}^{m} w_k = 1$. Typically the QoS values of can be integers from a given range (e.g. $0, 1, 2, 3$ or real numbers of a close interval (e.g. $[-20, 20]$). Without loss of generality, we can map a QoS value to the interval $[0, 1]$ using the function $f(x) = (x - q_{min})/(q_{max} - q_{min})$, where q_{max} and q_{min} are the maximum and minimum QoS values of the corresponding criterion respectively.

The proposed primary selection algorithm is shown in Algorithm 7. After accepting the priority vector from the cloud module, a rating

Algorithm 7: Primary Selection Algorithm

Input: N, Q, W
Output: n^*

```
n* = null;
r_max = 0;
for all n_i ∈ N do
    r_i = Σ_{k=1}^{m} q_ik × w_k;
    if r_i > r_max then
        n* = n_i;
        r_max = r_i;
    end
end
return n*;
```

value r_i is computed for each node $n_i \in N$ as follows:

$$r_i = \sum_{k=1}^{m} q_{ik} \times w_k, \tag{7.1}$$

where r_i fall into the interval $[0, 1]$. The cloud module will choose the node n^*, which has the highest rating value, as the primary:

$$n^* = \arg\max_{n_i \in N} r_i. \tag{7.2}$$

7.3.3 Replica Selection

After the primary is selected in Section 7.3.2, a set of replicas should be chosen to form a BFT group. Since replicas in a BFT group need to communicate with both the primary and the cloud module, the QoS performance of a node should be considered from both the cloud module perspective and the primary perspective. Let q_i be the QoS vector of node n_i observed by the cloud module and q_i' be the QoS vector of node n_i observed by the primary. Then the combined QoS vector q_i'' is calculated by a set of transformation rules as follows:

- minimum: $q''_{ik} = min(q_{ik}, q'_{ik})$, for QoS criterion like bandwidth.
- average: $q''_{ik} = avg(q_{ik}, q'_{ik})$, for QoS criterion like response time.
- equality: $q''_{ik} = q_{ik} = q'_{ik}$, for QoS criterion like price.

Without loss of generality, the rule set can be easily extended to include more rules for calculating complex QoS criterion values.

Given the combined QoS vector q''_i, we can evaluate how appropriate the node n_i is as a replica of the BFT group. A score s_i is assigned to each node $n_i \in N$ as follows:

$$s_i = \sum_{k=1}^{m} q''_{ik} \times w_k, \tag{7.3}$$

where s_i falls into the interval $[0, 1]$. After ordering the scores, we can select the nodes ranked in high positions as replicas of the BFT group.

In order to decide the replication degree, we first consider the failure probability of a BFT group in its entirety. Since the BFTCloud guarantees the execution correctness when up to f nodes are faulty, a BFT group is faulty if and only if more than f nodes are faulty. We define the failure probability of a BFT group σ as follows:

$$P_\sigma = P(|F| > f), \tag{7.4}$$

where F is the set of failure nodes in σ.

In order to reduce the cost of request execution, the replication degree f should be as small as possible, and the failure probability of a BFT group σ should be guaranteed under a certain threshold at the same time. Let P_0 be the threshold of P_σ defined by the cloud module. The replication degree decision problem can be formulated as an optimization problem:

Algorithm 8: Replica Selection Algorithm

Input: N, Q, Q', W, P_0
Output: σ

$\sigma = null$;
$f = 0$;
$P_\sigma = P^*$;
for *all* $n_i \in N$ **do**
  $q_i'' \leftarrow (q_i, q_i')$ by applying the set of transformation rules;
  $s_i = \sum_{k=1}^{m} q_{ik}'' \times w_k$;
end
Generate a permutation $< n_1', n_2', \cdots >$ of the set N such that $s_1' \geq s_2' \geq \cdots$;
while $P_\sigma > P_0$ **do**
  $f = f + 1$;
  $\sigma = \{n_1', n_2', \cdots, n_{3f}'\}$;
  $P_\sigma = 0$;
  for *all* $F \in \Omega$ **do**
    $P_\sigma = P_\sigma + \prod_{n_i \in F} P_i \prod_{n_j \in \sigma \setminus F} (1 - P_j)$;
  end
end
return σ;

$$
\begin{aligned}
\min_{f} \quad & f = \frac{|\sigma| - 1}{3}, \\
& P_\sigma = \sum_{F \in \Omega} \prod_{n_i \in F} P_i \prod_{n_j \in \sigma \setminus F} (1 - P_j), \\
& P_\sigma < P_0, \\
& \Omega = \{F | f < |F|\}.
\end{aligned} \tag{7.5}
$$

where P_i is the failure probability of node n_i, and Ω is the set of events that more than f nodes of the BFT group σ are fault. Note that a solution to this problem decides the replication degree and the replicas of BFT group σ at the same time. We summarize the replica selection algorithm in Algorithm 8.

7.3.4 Request Execution

After the BFT group members are determined, requests can be sent to the BFT group for execution. The cloud module first forms a request sequence and sends the sequence of requests to the primary. The primary will order the requests and forward the ordered requests to all the BFT group members. Each member of the BFT group will execute the sequence of requests and send the corresponding responses back to the cloud module. The cloud module collects all the received responses from the BFT group members and make a judgement on the consistence of responses. A action strategy will be chose according to the consistence of responses as follows:

- **Case 1:** The cloud module receives $3f + 1$ consist responses from the BFT group. In this case, the cloud module will commit the current request since there is no fault happens in the current BFT group.

- **Case 2:** The cloud module receives between $2f + 1$ to $3f$ consist responses. In this case, the cloud module can still commit the current request since there are less than $f + 1$ faults happened. To commit the current request and identify the faulty nodes, the cloud module assembles a commit certificate and sends the commit certificate to all the BFT group members. Each member will acknowledge the cloud module with a local-commit message once it receives the commit certificate from the cloud module. If more than $2f + 1$ local-commit messages are received the cloud module will commit the current request and invokes a replica updating procedure to replace all the faulty BFT group members with new members. If less than $2f + 1$ local-commit messages are received, the cloud module will resend the commit certificate until it receives local-commit messages from more than $2f + 1$ members.

- **Case 3:** The cloud module receives less than $2f + 1$ response

messages. In this case, either the primary is faulty or more than $f+1$ replicas are faulty. The cloud module will then resend the current request again but to all members this time. Each replica forwards the request to the node it believes to be the primary. If the replica receives a request from the primary within a given time and the proposed sequence number is consistency with that sent by the cloud module, the replica will execute the request and send response to the cloud module. If the replica does't receive an ordered request from the primary within a given time, or the request sequence number isn't consistency with the request sent by the cloud module, the primary must be faulty. The replica will send a primary election proposal to all replicas to trigger a primary updating procedure.

- **Case 4:** The cloud module receives more than $2f+1$ responses, but fewer than $f+1$ responses are consistency. This indicates inconsistent ordering of requests by the primary. The cloud module will send a proof of misbehavior of primary to all the replicas, and trigger a primary updating procedure.

7.3.5 Primary updating

When the primary is faulty, a primary updating procedures will be triggered in the Request Execution phase. The procedures of primary updating phase are as follows:

1. A replica which suspects the primary to be faulty sends an primary election proposal to all the other replicas. However, it still participates in the current BFT group as it may be only a network problem between the replica and the primary. Other replicas, once receiving a primary election proposal, just store it since the primary election proposal could arrive from a faulty replica as well.

2. If a replica receives $f+1$ primary election proposals , it in-

dicates that the primary is really faulty. It will send a primary selection request to the cloud module. The cloud module then will start the primary selection phase and return a new primary which is one of the current replicas. The replica then sends a primary updating message to all the other replicas, which includes the new primary name and $f + 1$ primary election proposals. Other replicas which receive such primary updating message again confirm that at least $f + 1$ replicas sent a primary election proposal, and then resend the primary updating message together with the proof to the new primary.

3. If the newly selected primary receives $2f + 1$ primary updating messages, it sends a new BFT group setup message to all the replicas, which again includes all the primary updating messages as proof.

4. A replica which received and confirmed the new BFT group setup message, will send out a BFT group confirm message to all replicas.

5. If a replica receives $2f + 1$ BFT group confirm messages, it starts performing as a memeber in the new BFT group.

7.3.6 Replica Updating

In the voluntary-resource cloud environment, nodes are highly dynamic and fragile. Different types of faults (e.g., response time out, unavailable, arbitrary behavior, etc.) may happen to the nodes after a period of time. Under voluntary-resource cloud infrastructure, the failure probability of a BFT group increases sharply as the fraction of faulty nodes increases. The failure probability of a BFT group under the condition that a set of replicas are already faulty is:

$$\begin{aligned} P_\sigma &= P(|F| > f|F^*) \\ &= P(|F \setminus F^*| > f - f^*), \end{aligned} \tag{7.6}$$

where F^* is the set of replicas which are faulty already.

To ensure the failure probability of a BFT group below a certain threshold, we need to replace the members once they are identified to be faulty. Moreover, due to the highly dynamic voluntary-resource cloud environment, the QoS performance of nodes are changed rapidly. Updating replicas timely could keep the performance of a BFT group stable.

Let T be the set of new nodes which will be added to the current BFT group. F^* is the set of nodes which will be removed from the current BFT group. Let σ' be the new BFT group with updated replicas. We have $\sigma' = \sigma \setminus F^* \cup T$, where T consists of nodes which are in the top $|T|$ positions ordered by score in Eq. (7.3).

The new BFT group σ', which can tolerate up to f' nodes failure, should satisfy $P_{\sigma'} > P_0$. Therefore, the replica updating problem is reduced to a replication degree decision problem, which can be further formalized as an optimization problem as follows:

$$\begin{aligned} \min_{f'} \quad & f' = \frac{|\sigma'| - 1}{3}, \\ & P_{\sigma'} = \sum_{F' \in \Lambda} \prod_{n_i \in F'} P_i \prod_{n_j \in \sigma' \setminus F'} (1 - P_j), \\ & P_{\sigma'} < P_0, \\ & \Lambda = \{F' | f < |F'|\}. \end{aligned} \tag{7.7}$$

where Λ is the set of events that more than f' nodes of the BFT group σ' are fault. We summarize the replica updating algorithm in Algorithm 9.

7.4 Experiments

In this section, in order to study the performance improvements of our proposed approach, we conduct several experiments to compare our BFTCloud with several other fault tolerance approaches.

Algorithm 9: Replica Updating Algorithm

Input: N, Q, Q', W, P_0, σ, F^*
Output: σ'

$\sigma' = \sigma \setminus F^*$;
$T = null$;
$f' = \lceil \frac{3f-|F^*|}{3} \rceil$;
$P'_\sigma = P^*$;
for *all* $n_i \in N \setminus \sigma$ **do**
  $q''_i \leftarrow (q_i, q'_i)$ by applying the set of transformation rules;
  $s_i = \sum_{k=1}^{m} q''_{ik} \times w_k$;
end
Generate a permutation $< n'_1, n'_2, \cdots >$ of the set $N \setminus \sigma$ such that $s'_1 \geq s'_2 \geq \cdots$;
$T = \{n'_1, n'_2, \cdots, n'_{3f'-|\sigma'|}\}$;
$\sigma' = \sigma' \cup T$;
while $P'_\sigma > P_0$ **do**
  $f' = f' + 1$;
  $T = \{n'_1, n'_2, \cdots, n'_{3f'-|\sigma'|}\}$;
  $\sigma' = \sigma' \cup T$;
  $P'_\sigma = 0$;
  for *all* $F \in \Lambda$ **do**
    $P'_\sigma = P'_\sigma + \prod_{n_i \in F'} P_i \prod_{n_j \in \sigma' \setminus F'} (1 - P_j)$;
  end
end
return σ';

In the following, Section 7.4.1 describes the system implementation of BFTCloud and the experimental settings, and Section 7.4.2 compares the performances of BFTCloud with some other fault tolerance methods.

7.4.1 Experimental Setup

We have implemented our BFTCloud approach by Java language and deployed it as a middleware in a voluntary-resource cloud environment. The cloud infrastructure is constructed by 257 distributed computers located in 26 countries from Planet-lab, which is a dis-

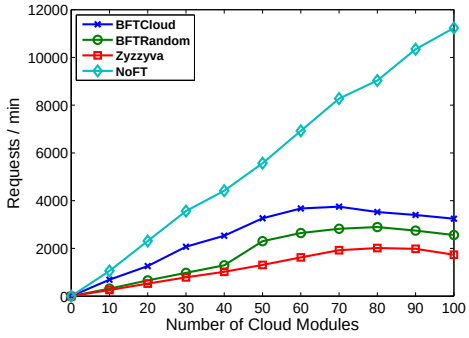
tributed test-bed consisting of hundreds of computers all over the world. Each computer, named as node in the cloud infrastructure, can participate several BFT groups as a primary or replica simultaneously.

Based on the voluntary-resource cloud infrastructure, we conduct large-scale experiments study the performance improvements of BFTCloud compared with other approaches. In our experiments, each node in the cloud is configured with a random malicious failure probability, which denotes the probability of arbitrary behavior happens in the node. Note that the failure probability of a node observed by other nodes is not necessarily equal to the malicious failure probability since other types of faults (e.g., node crashing, disconnection, etc.) may also occur. Each node keeps the QoS information of all the other nodes and updates the information periodically. For simplicity, we use response-time for QoS evaluation in this chapter. Without loss of generality, our approach can be easily extended to include more QoS criteria. We also employed 100 computers from Planet-lab to perform as cloud modules.

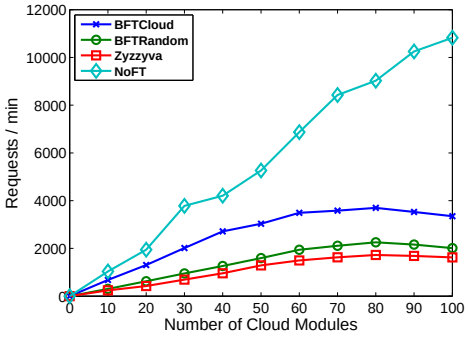
7.4.2 Performance Comparison

In this section, we compare the performance of our approach BFTCloud with other fault tolerance approaches in the voluntary-resource cloud environment. We have implemented four approaches:

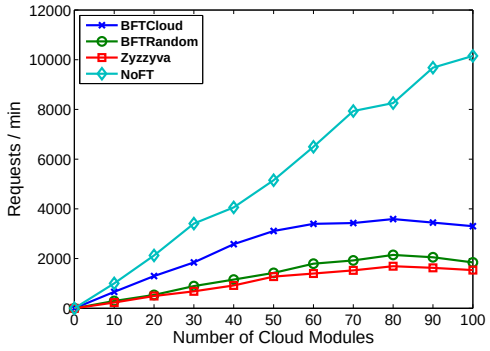
- NoFT: No fault tolerance strategy is employed for task execution in the voluntary-resource cloud.
- Zyzzyva: A state-of-the-art Byzantine Fault tolerance approach proposed in [61]. The cloud module sends requests to a fixed primary and a group of replicas. There is no mechanism designed for adopting the dynamic voluntary-resource cloud environment.
- BFTCloud: The Byzantine Fault tolerance framework proposed



(a) Request/Response Size: 0/0 KB



(b) Request/Response Size: 4/0 KB



(c) Request/Response Size: 0/4 KB

Figure 7.4: Throughput Comparison for 0/0, 4/0, and 0/4 benchmarks as the number of cloud modules varies

in this chapter. The cloud module employs Algorithm 1-3 to mask faults and adopt the highly dynamic voluntary-resource environment.

- BFTRandom: The framework is the same with BFTCloud. However, this approach just randomly selects nodes in primary selection, replica selection, primary updating, and replica updating phases.

In Figure 7.4, we compare the throughput of all approaches in terms of different number of cloud modules by executing null operations. We change the number of cloud module from 0 to 100 with a step value of 10. The requests are sent by a variable number of cloud modules in each experiment (0-100). We conducts experiments on

Table 7.1: Average Sending Times Per Request

Benchmark	BFTCloud	BFTRandom	Zyzzyva	NoFT
0/0 KB	1.3428	1.7096	2.9167	1.0725
4/0 KB	1.3035	1.7248	3.1002	1.1042
0/4 KB	1.3820	1.7340	3.2058	1.3055

three benchmarks [61] with different request and response size. The sizes of request/response messages are 0/0KB, 4/0KB, and 0/4KB in Figure 7.4(a), Figure 7.4(b), and Figure 7.4(c) respectively. The parameter settings in this experiment are $P_0 = 0.5$ and $timeout = 500ms$, where $timeout$ defines the maximum waiting time for a message. From Figure 7.4, we can observe that our approach BFTCloud can commit more requests per minute than Zyzzyva and BFTRandom under different sizes of request/response messages. The reason is that BFTCloud always choose nodes with low failure probabilities as BFT group members. Therefore, the high reliability of BFT group guarantees that in most cases a request can be committed without be resent. Note that NoFT achieves the highest throughput among all approaches since NoFT employs no fault tolerance mechanism. Every request will be committed once the cloud module received a reply. However, NoFT cannot guarantee the correctness of committed requests, which will be discussed in Table 7.2. Table 7.1 shows the average sending times of a request by the cloud module before it is committed. A request can be committed with much fewer sending times in BFTCloud than request in Zyzzyva, since BFT group members in BFTCloud are carefully selected and the probability of successfully executing a request is higher than that in Zyzzyva. Moreover, BFTCloud always choose nodes with good QoS performance as BFT group members which makes requests and responses are transmitted more quickly than other approaches. In general, BFTCloud achieves high throughput of committed requests which demonstrates that the idea of considering failure probability and QoS performance when selecting nodes is realistic and reason-

Table 7.2: Correct Rate of Committed Requests

size	BFTCloud	BFTRandom	Zyzzyva	NoFT
0/0 KB	0.9855	0.9468	0.8726	0.2589
4/0 KB	0.9840	0.9259	0.8925	0.2107
0/4 KB	0.9794	0.9278	0.8621	0.2216

able.

In Table 7.2, we evaluate the correctness of committed requests of different approaches. The experimental result shows that among all the committed requests, the percentage of correctly committed requests is highest in BFTCloud. This is because BFTCloud can guarantee a low probability P_0 that more than f BFT group members are faulty. While Zyzzyva cannot guarantee the failure probability of BFT group since the primary and replicas in Zyzzyva are fixed. Most of the requests are not correctly committed in NoFT despite high throughput of NoFT, since no fault tolerance mechanism is employed.

7.5 Summary

In this chapter, we propose BFTCloud, a Byzantine fault tolerance framework for building reliable systems in voluntary-resource cloud infrastructure. In BFTCloud, replication techniques are employed for improving the system reliability of cloud applications. To adapt to the highly dynamic voluntary-resource cloud environment, BFTCloud select voluntary nodes based on their QoS characteristics and reliability performance. Faulty voluntary resources will be replaced with other suitable resources once they are identified. The extensive experimental results show the effectiveness of our approach BFTCloud on guaranteeing the system reliability in cloud environment.

In the future, we will conduct more experimental analysis on open-source cloud applications and conduct more investigations on

different QoS properties of voluntary resources. We will conduct more experiments to study the impact of parameters and investigate the optimal values of parameters in different experimental settings.

□ **End of chapter.**

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis is aiming at advancing quality engineering in cloud and service computing. This thesis consists of three parts: the first part deals with service QoS prediction, the second part focuses on QoS-aware Web service searching, and the third part concentrates on QoS-aware fault-tolerant systems in cloud computing.

In the first part, we present three QoS prediction approaches for services. We first propose a neighborhood-based collaborative QoS prediction approach, which is enhanced by character modeling, for services. The second method is a model-based time-aware collaborative filtering approach which utilize time information to capture the periodicity features of service QoS values. Finally, we propose an online QoS prediction approach which employing time series analysis to adapt to the highly dynamic service computing environment. The online prediction approach consists of an offline evolutionary algorithm and an online incremental algorithm for precisely predicting the QoS values of services at runtime. The experimental results and the system level case study show the efficiency and effectiveness of our approach.

In the second part, we propose a QoS-aware Web service search engine. In order to provide better searching results to users for fulfilling their Web service requirements, we systematically fusing the

functional approach and non-functional approach to achieve better performance. Moreover, we conduct experiments on the real-world Web services. The collected WSDL files and QoS datasets are released for the Web service research community.

In the third part, we conduct a fault tolerance study on cloud applications. By taking the advantage of multiple functional equivalent services over the Internet, we design a Byzantine fault tolerance framework to build robust systems in voluntary-resource cloud environments. Our fault tolerance framework employs dynamic QoS information of services to select the most suitable services for system integration. The experimental results show the effectiveness of this framework.

In general, the goal of our work is to predict and utilize the QoS information in cloud and service computing as accurate and effective as possible. Our released QoS datasets enable the extensive research of other researchers.

8.2 Future Work

There are several research directions we can conduct further investigations in the future.

For the service QoS prediction, we plan to conduct more research on the correlation of multiple QoS characteristics since the different QoS characteristics are considered independently in current stage. The relationship between different QoS properties may provide some useful information for improving the prediction accuracy. Another direction worthy of investigation is how to explore the relationship between user information and service information to enhance the prediction accuracy.

For the QoS-aware Web service searching, we plan to design a clustering algorithm improve the accuracy of functional similarity computation. Currently, we only use the average QoS performance of Web services. However, due to the dynamic network environment

and service status, we plan to extend the non-functional evaluation module to adopt dynamic QoS information of Web services.

For the QoS-aware fault-tolerance framework in cloud computing, we can conduct more studies on the correlation of different types of failure, since failures may not be independent of each other. Moreover, failures of different services in the cloud application may have correlation with each other.

□ **End of chapter.**

Bibliography

[1] N. Ahmed, M. Linderman, and J. Bryant. Towards Mobile Data Streaming in Service Oriented Architecture. In *Proc. of SRDS'10*, pages 323–327.

[2] E. Al-Masri and Q. H. Mahmoud. Investigating web services on the world wide web. In *Proceedings of the 17th international conference on World Wide Web*, pages 795–804. ACM, 2008.

[3] M. Alrifai and T. Risse. Combining Global Optimization with Local Selection for Efficient QoS-Aware Service Composition. In *Proc. of International Conference on World Wide Web (WWW'09)*, pages 881–890, 2009.

[4] M. Alrifai, D. Skoutas, and T. Risse. Selecting skyline services for qos-based web service composition. In *Proc. of WWW'10*, pages 11–20, 2010.

[5] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *Software Engineering, IEEE Transactions on*, 33(6):369–384, 2007.

[6] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[7] A. Avizienis. The methodology of n-version programming. *Software fault tolerance*, pages 23–46, 1995.

[8] W. Balke and M. Wagner. Towards personalized selection of web services. *Proc. of WWW'03*, 2003.

[9] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Proc. of ISSRE'09*, pages 109–119, 2009.

[10] P. Bonatti and P. Festa. On optimal service selection. In *Proc. of WWW'05*, pages 530–538, 2005.

[11] J. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of UAI'98*, pages 43–52, 1998.

[12] X. Cai, M. Bain, A. Krzywicki, W. Wobcke, Y. Kim, P. Compton, and A. Mahidadia. Learning collaborative filtering and its application to people to people recommendation in social networks. In *Proc. of ICDM'10*, pages 743–748, 2010.

[13] G. Canfora, M. Di Penta, R. Esposito, and M. Villani. QoS-aware replanning of composite Web services. In *Proc. of ICWS'05*, pages 121–129, 2005.

[14] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *Proc. 24th international conference on Machine learning*, pages 129–136, 2007.

[15] V. Cardellini, E. Casalicchio, V. Grassi, and F. Lo Presti. Flow-based service selection forweb service composition supporting multiple qos classes. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 743–750. IEEE, 2007.

[16] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *Proceedings of the the 7th joint meet-*

ing of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 131–140. ACM, 2009.

[17] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281–308, 2004.

[18] M. Castro and B. Liskov. Practical Byzantine fault tolerance. *Operating Systems Review*, 33:173–186, 1998.

[19] P. Chan, M. R. Lyu, and M. Malek. Reliableweb services: Methodology, experiment and modeling. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 679–686. IEEE, 2007.

[20] P. P. W. Chan, M. R. Lyu, and M. Malek. Making services fault tolerant. In *Service Availability*, pages 43–61. Springer, 2006.

[21] A. Chandra and J. Weissman. Nebulas: using distributed voluntary resources to build clouds. In *Proc. of HOTCLOUD'09*, 2009.

[22] W. Chen, J. Chu, J. Luan, H. Bai, Y. Wang, and E. Chang. Collaborative filtering for orkut communities: discovery of user latent behavior. In *Proc. of WWW'09*, pages 681–690, 2009.

[23] X. Chen, X. Liu, Z. Huang, and H. Sun. Regionknn: A scalable hybrid collaborative filtering algorithm for personalized web service recommendation. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 9–16. IEEE, 2010.

[24] R. C. Cheung. A user-oriented software reliability model. *Software Engineering, IEEE Transactions on*, (2):118–125, 1980.

[25] M. Creeger. Cloud Computing: An Overview. *ACM Queue*, 7(5):1–5, 2009.

[26] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 6(2):86–93, 2002.

[27] A. Danak and S. Mannor. Resource Allocation with Supply Adjustment in Distributed Computing Systems. In *Proc. of ICDCS'10*, pages 498–506, 2010.

[28] V. Deora, J. Shao, W. Gray, and N. Fiddian. A quality of service management framework based on user expectations. In *Proc. of ICSOC'03*, pages 104–114, 2003.

[29] M. Deshpande and G. Karypis. Item-based top-n recommendation algorithms. *ACM Transactions on Information Systems (TOIS)*, 22(1):143–177, 2004.

[30] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *Proc. 30th Intl. Conf. on Very Large Data Bases (VLDB'04)*, pages 372–383, 2004.

[31] J. El Haddad, M. Manouvrier, G. Ramirez, and M. Rukoz. Qos-driven selection of web services for transactional composition. In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 653–660. IEEE, 2008.

[32] J. El Haddad, M. Manouvrier, and M. Rukoz. Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE Transactions on Services Computing*, pages 73–85, 2010.

[33] T. Erl. *Service-oriented architecture*, volume 8. Prentice Hall New York, 2005.

[34] D. Fensel, F. Facca, E. Simperl, and I. Toma. Web service modeling ontology. *Semantic Web Services*, pages 107–129, 2011.

[35] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 152–161. IEEE, 2003.

[36] R. Ghosh, F. Longo, V. Naik, and K. Trivedi. Quantifying Resiliency of IaaS Cloud. In *Proc. of SRDS'10*, pages 343–347, 2010.

[37] Gift Website. http://bjqad.com/yawen/mall/index.asp.

[38] S. S. Gokhale and K. S. Trivedi. Reliability prediction and sensitivity analysis based on software architecture. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 64–75. IEEE, 2002.

[39] K. Gomadam, A. Ranabahu, M. Nagarajan, A. P. Sheth, and K. Verma. A faceted classification based approach to search and rank web apis. In *Proc. 6th Intl. Conf. on Web Services (ICWS'08)*, pages 177–184, 2008.

[40] S. Gong. A collaborative filtering recommendation algorithm based on user clustering and item clustering. *Journal of Software*, 5(7):745–752, 2010.

[41] V. Grassi and S. Patella. Reliability prediction for service-oriented computing environments. *Internet Computing, IEEE*, 10(3):43–49, 2006.

[42] Y. Hao, Y. Zhang, and J. Cao. WSXplorer: Searching for desired web services. In *Proc. 19th Intl. Conf. on Advanced Information System Engineering (CaiSE'07)*, pages 173–187, 2007.

[43] B. Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, 2008.

[44] T. Henzinger, A. Singh, V. Singh, T. Wies, and D. Zufferey. FlexPRICE: Flexible Provisioning of Resources in a Cloud Environment. In *Proc. of CLOUD'10*, pages 83–90, 2010.

[45] J. Herlocker, J. Konstan, A. Borchers, and J. Riedl. An algorithmic framework for performing collaborative filtering. In *Proc. of SIGIR'99*, pages 230–237, 1999.

[46] T. Hofmann. Collaborative filtering via gaussian probabilistic latent semantic analysis. In *Proc. of SIGIR'03*, pages 259–266, 2003.

[47] T. Hofmann. Latent semantic models for collaborative filtering. *ACM Transactions on Information Systems (TOIS)*, 22(1):89–115, 2004.

[48] http://axis.apache.org/axis2/java/core.

[49] http://en.wikipedia.org/wiki/Cross validation.

[50] http://en.wikipedia.org/wiki/Web_service.

[51] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl. Qos aggregation for web service composition using workflow patterns. In *Enterprise distributed object computing conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, pages 149–159. IEEE, 2004.

[52] M. Jamali and M. Ester. Trustwalker: a random walk model for combining trust-based and item-based recommendation. In *Proc. of KDD'09*, pages 397–406, 2009.

[53] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information System*, 20(4):422–446, 2002.

[54] Y. Jianjun, G. Shengmin, S. Hao, Z. Hui, and X. Ke. A kernel based structure matching for web services search. In *Proc. 16th Intl. Conf. on World Wide Web (WWW'07)*, pages 1249–1250, 2007.

[55] R. Jin, J. Chai, and L. Si. An automatic weighting scheme for collaborative filtering. In *Proc. of SIGIR'04*, pages 337–344, 2004.

[56] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu. Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures. In *Proc. of ICDCS'10*, pages 62–73, 2010.

[57] K. Karta. An investigation on personalized collaborative filtering for web service selection. *Honours Programme thesis, University of Western Australia, Brisbane*, 2005.

[58] A. Keller and H. Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[59] K. Kim and H. Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, 38(5):626–636, 2002.

[60] M. Klusch, B. Fries, and K. Sycara. Automated semantic web service discovery with owls-mx. In *Proc. 5th Intl. Conf. on Autonomous agents and multiagent systems (AAMAS '06)*, pages 915–922, 2006.

[61] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proc. of SOSP'07*, pages 45–58, 2007.

[62] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[63] D. Lee and H. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.

[64] W. Li, J. He, Q. Ma, I. Yen, F. Bastani, and R. Paul. A framework to support survivable web services. In *Proc. of IPDPS'05*, page 93b, 2005.

[65] D. Liang, C.-L. Fang, C. Chen, and F. Lin. Fault tolerant web service. In *Software Engineering Conference, 2003. Tenth Asia-Pacific*, pages 310–319. IEEE, 2003.

[66] D. Liang, C.-L. Fang, S.-M. Yuan, C. Chen, and G. E. Jan. A fault–tolerant object service on corba. *Journal of Systems and Software*, 48(3):197–211, 1999.

[67] G. Linden, B. Smith, and J. York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

[68] N. N. Liu, M. Zhao, E. Xiang, and Q. Yang. Online evolutionary collaborative filtering. In *Proc. of the fourth conference on Recommender systems (RecSys'10)*, pages 95–102, 2010.

[69] Y. Liu, A. H. Ngu, and L. Z. Zeng. Qos computation and policing in dynamic web service selection. In *Proc. 13th Intl. Conf. on World Wide Web (WWW'04)*, pages 66–73, 2004.

[70] A. Luckow and B. Schnor. Service replication in grids: ensuring consistency in a dynamic, failure-prone environment.

In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7. IEEE, 2008.

[71] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck. A service level agreement language for dynamic electronic services. *Electronic Commerce Research*, 3(1-2):43–59, 2003.

[72] M. Lyu. *Software fault tolerance*. John Wiley & Sons, 1995.

[73] M. Lyu et al. Handbook of software reliability engineering. 1996.

[74] H. Ma, I. King, and M. Lyu. Effective missing data prediction for collaborative filtering. In *Proc. of SIGIR'07*, pages 39–46, 2007.

[75] E. M. Maximilien and M. P. Singh. Conceptual model of web service reputation. *Acm Sigmod Record*, 31(4):36–41, 2002.

[76] D. A. Menascé. Qos issues in web services. *Internet Computing, IEEE*, 6(6):72–75, 2002.

[77] M. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *Proc. of SRDS'05*, pages 131–140, 2005.

[78] T. O reilly. What is Web 2.0: Design patterns and business models for the next generation of software. *Communications and Strategies*, 65:17, 2007.

[79] J. O'Sullivan, D. Edmond, and A. Ter Hofstede. What's in a service? *Distributed and Parallel Databases*, 12(2-3):117–133, 2002.

[80] M. Ouzzani and A. Bouguettaya. Efficient access to web services. *Internet Computing, IEEE*, 8(2):34–44, 2004.

[81] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In *Proc. 1st Intl. Semantic Web Conf. (ISWC'02)*, pages 333–347, 2002.

[82] P. Plebani and B. Pernici. Urbe: Web service retrieval based on similarity evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 21(11):1629–1642, 2009.

[83] S. Ran. A model for web services discovery with qos. *ACM Sigecom exchanges*, 4(1):1–10, 2003.

[84] S. Rendle and L. Schmidt-Thieme. Pairwise interaction tensor factorization for personalized tag recommendation. In *Proc. of WSDM'10*, pages 81–90, 2010.

[85] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186. ACM, 1994.

[86] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3):241–252, 2003.

[87] S. Rosario, A. Benveniste, S. Haar, and C. Jard. Probabilistic qos and soft contracts for transaction-based web services orchestrations. *Services Computing, IEEE Transactions on*, 1(4):187–200, 2008.

[88] A. Sahai, A. Durante, and V. Machiraju. Towards automated sla management for web services. *Hewlett-Packard Research Report HPL-2001-310 (R. 1)*, 2002.

[89] R. Salakhutdinov and A. Mnih. Probabilistic matrix factorization. *Advances in neural information processing systems (NIPS)*, 20:1257–1264, 2008.

[90] J. Salas, F. Perez-Sorrosal, M. Patiño-Martínez, and R. Jiménez-Peris. Ws-replication: a framework for highly available web services. In *Proceedings of the 15th international conference on World Wide Web*, pages 357–366. ACM, 2006.

[91] N. Salatge and J. Fabre. Fault Tolerance Connectors for Unreliable Web Services. In *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 51–60, 2007.

[92] G. Salton. *The SMART Retrieval System—Experiments in Automatic Document Processing*. Prentice-Hall, Inc., 1971.

[93] G. T. Santos, L. C. Lung, and C. Montez. Ftweb: A fault tolerant infrastructure for web services. In *EDOC Enterprise Computing Conference, 2005 Ninth IEEE International*, pages 95–105. IEEE, 2005.

[94] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[95] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme. An autonomic approach for replication of internet-based services. In *Proc. of SRDS'08*, pages 127–136, 2008.

[96] L. Shao, J. Zhang, Y. Wei, J. Zhao, B. Xie, and H. Mei. Personalized qos prediction forweb services via collaborative filtering. In *Proc. of ICWS'07*, pages 439–446, 2007.

[97] Y. Shi, M. Larson, and A. Hanjalic. Exploiting user similarity based on rated-item pools for improved user-based collaborative filtering. In *Proc. of Recsys'09*, pages 125–132, 2009.

[98] L. Si and R. Jin. Flexible mixture model for collaborative filtering. In *ICML*, volume 3, pages 704–711, 2003.

[99] P. Singla and M. Richardson. Yes, there is a correlation:-from social networks to personal behavior on the web. In *Proc. of WWW'08*, pages 655–664, 2008.

[100] A. Soydan Bilgin and M. P. Singh. A daml-based repository for qos-aware semantic web service selection. In *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 368–375. IEEE, 2004.

[101] R. M. Sreenath and M. P. Singh. Agent-based service selection. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):261–279, 2004.

[102] L. Tang, J. Dong, Y. Zhao, and L. Zhang. Enterprise Cloud Service Architecture. In *Proc. of CLOUD'10*, pages 27–34, 2010.

[103] N. Thio and S. Karunasekera. Automatic measurement of a qos metric for web service recommendation. In *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, pages 202–211. IEEE, 2005.

[104] K. Tsakalozos, M. Roussopoulos, V. Floros, and A. Delis. Nefeli: Hint-Based Execution of Workloads in Clouds. In *Proc. of ICDCS'10*, pages 74–85, 2010.

[105] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller. Meteor-s wsdi: A scalable p2p infrastructure of registries for semantic publication and discovery of web services. *Inf. Technol. and Management*, 6(1):17–39, 2005.

[106] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Depend-*

able Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on, pages 566–571. IEEE, 2009.

[107] J. Wang, A. De Vries, and M. Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proc. of SIGIR'06*, pages 501–508, 2006.

[108] Y. Wang and E. Stroulia. Semantic structure matching for assessing web service similarity. In *Proc. 1st Intl. Conf. on Service Oriented Computing (ICSOC'03)*, pages 194–207, 2003.

[109] Wikipedia. http://en.wikipedia.org/wiki/byzantine_fault_tolerance.

[110] Wikipedia. http://en.wikipedia.org/wiki/cloud_computing.

[111] G. Wu, J. Wei, X. Qiao, and L. Li. A bayesian network based qos assessment model for web services. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 498–505. IEEE, 2007.

[112] P. Xiong, Y. Fan, and M. Zhou. Qos-aware web service configuration. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and humans*, 38(4):888–895, 2008.

[113] P. Xiong, Y. Fan, and M. Zhou. A petri net approach to analysis and composition of web services. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 40(2):376–387, 2010.

[114] G. Xue, C. Lin, Q. Yang, W. Xi, H. Zeng, Y. Yu, and Z. Chen. Scalable collaborative filtering using cluster-based smoothing. In *Proc. of SIGIR'05*, pages 114–121, 2005.

[115] T. Yu, Y. Zhang, and K. Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web (TWEB)*, 1(1):6, 2007.

[116] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering (TSE)*, 30(5):311–327, 2004.

[117] L. Zhang, S. Cheng, C. Chang, and Q. Zhou. A pattern-recognition-based algorithm and case study for clustering and selecting business services. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 42(1):102–114, 2012.

[118] L.-J. Zhang, J. Zhang, and H. Cai. *Services computing*. Springer, 2007.

[119] Y. Zhang, Z. Zheng, and M. Lyu. WSExpress: A QoS-aware Search Engine for Web Services. In *Proc. of ICWS'10*, pages 91–98, 2010.

[120] Y. Zhang, Z. Zheng, and M. Lyu. Wspred: A time-aware personalized qos prediction framework for web services. In *Proc. of IEEE Symposium on Software Reliability Engineering (ISSRE'11)*, pages 210–219, 2011.

[121] Y. Zhang, Z. Zheng, and M. R. Lyu. Bftcloud: A byzantine fault tolerance framework for voluntary-resource cloud computing. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 444–451. IEEE, 2011.

[122] Y. Zhang, Z. Zheng, and M. R. Lyu. Exploring latent features for memory-based qos prediction in cloud computing. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 1–10. IEEE, 2011.

[123] Y. Zhang, Z. Zheng, and M. R. Lyu. Real-time performance prediction for cloud components. In *Object/Component/Service-Oriented Real-Time Distributed*

Computing Workshops (ISORCW), 2012 15th IEEE International Symposium on, pages 106–111. IEEE, 2012.

[124] W. Zhao. BFT-WS: A Byzantine fault tolerance framework for web services. In *Proc. of EDOC'07*, pages 89–96, 2008.

[125] Z. Zheng and M. Lyu. Collaborative reliability prediction of service-oriented systems. In *Proc. of ICSE'10*, pages 35–44, 2010.

[126] Z. Zheng and M. R. Lyu. A distributed replication strategy evaluation and selection framework for fault tolerant web services. In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 145–152. IEEE, 2008.

[127] Z. Zheng, H. Ma, M. Lyu, and I. King. Wsrec: A collaborative filtering based web service recommender system. In *Proc. of ICWS'09*, pages 437–444, 2009.

[128] Z. Zheng, Y. Zhang, and M. Lyu. CloudRank: A QoS-Driven Component Ranking Framework for Cloud Computing. In *Proc. of SRDS'10*, pages 184–193, 2010.

[129] Z. Zheng, Y. Zhang, and M. Lyu. Distributed QoS Evaluation for Real-World Web Services. In *Proc. of ICWS'10*, pages 83–90, 2010.

[130] Z. Zheng, T. Zhou, M. Lyu, and I. King. Ftcloud: A component ranking framework for fault-tolerant cloud applications. In *Proc. of ISSRE'10*, pages 398–407.