A Framework for Migration of Conventional Client-Server Software Systems to Cloud

by

Jianbo Zheng

Bachelor of Management, Shandong University of Science and Technology, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Weichang Du, Ph.D., Computer Science
Examining Board:	Huajie Zhang, Ph.D., Computer Science
	Wei Song, Ph.D., Computer Science
	Donglei Du, Ph.D., Business Administration

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May, 2014

Abstract

As an emerging model for the delivery of software services, Software as a Service (SaaS) becomes a trend in software industry due to its low investment, flexibility and accessibility. However, migration of conventional client-server software systems and applications to SaaS may involve complicated processes. This thesis proposes a framework named A2SF for helping software developers to migrate conventional client-server applications to high quality SaaS based applications in cloud environments, with multi-tenancy support, without re-developing or modifying the original applications. The migration framework consists of four components: service proxy, data proxy, tenant management, and cloud resources management. The four framework components, together with an original client-server application, can be seamlessly deployed on the cloud as an SaaS software. A prototype of A2SF has been implemented on the Amazon AWS cloud platform. Based on A2SF, the thesis also describes a general cloud migration process for client-server applications and presents a case study of migrating a real-world client-server application to Amazon AWS cloud.

Acknowledgements

I would like to express my great thanks to my supervisor, Dr. Weichang Du, who gave me his patient, encouragement, guidance, and comments on almost every detail and aspect of my thesis.

I would also like to thank to the University of New Brunswick, the faculty of computer science, for offering me all the facilities and resources that necessary to complete my degree.

Finally, special thanks to my family far away back in China, for their sincere support, encouragement and love. And also special thanks to all my friends, for spending time with me and making me happiness.

Table of Contents

Abstract ii

Acknowledgments iii

Table of Contents vii

List of Tables viii

List of Figures xi

Abbreviations xii

1 Introduction 1

1.1 Motivation 1

1.2 Thesis Overview 5

1.3 Thesis Structure 7

2 Background 8

2.1 Conventional Client-Server Model 8

2.2 Cloud Computing 11

2.2.1 Infrastructure as a service . . . 13
2.2.2 Platform as a service . . . 14
2.2.3 Software as a service . . . 15
2.2.4 Multitenancy . . . 16
2.2.5 SaaS maturity model . . . 18
2.3 Challenges of Migration . . . 20
2.4 Related Works . . . 23

3 Framework Design **25**
3.1 Overview . . . 25
3.1.1 Requirements . . . 26
3.2 Architecture Design . . . 30
3.2.1 Service proxy layer . . . 32
3.2.1.1 General service request process . . . 32
3.2.2 Tenant manager . . . 34
3.2.2.1 Tenant identification process . . . 34
3.2.2.2 Tenant status management process . . . 36
3.2.3 Service instance manager . . . 37
3.2.3.1 Service instance generation and recycle process 37
3.2.4 Data access layer . . . 40
3.2.4.1 Database data proxy process . . . 41

4 Implementation **43**
4.1 Overview . . . 43

4.2 Component Implementation 45
4.2.1 Implementation of TCP proxy server 46
4.2.2 Implementation of service proxy server 48
4.2.2.1 Implementation of HttpHandler 49
4.2.2.2 Implementation of tenant manager 50
4.2.2.3 Implementation of service instance manager . 52
4.2.3 Implementation of data proxy server 54
4.2.3.1 Implementation of MysqlHandler 55
4.2.4 Implementation of application management centre . . . 57
4.2.4.1 Implementation of application management . 58
4.2.4.2 Implementation of tenant management 60

5 Case Study: SugarCRM Cloud Migration 63

5.1 SugarCRM Client-Server Application 64
5.2 Software Migration and Deployment 66
5.2.1 Analyze original application 66
5.2.1.1 Check compatibility 66
5.2.1.2 Separate user privacy related components . . 67
5.2.1.3 Determine cloud services 68
5.2.2 Create migration package 69
5.2.3 Deploy the migration package on Amazon cloud 71
5.3 Test the Migrated System 73

6 Experiments and Evaluation 80

6.1 Experiments on Time Latency 81
6.1.1 Objective of the experiments 81
6.1.2 Experiments and results 84
6.1.2.1 Experiment on the original client-server system 85
6.1.2.2 Experiment on the migrated system 86
6.1.3 Result analysis 88
6.2 Other Experiments 89
6.3 A2SF Evaluation 90
6.3.1 Usability 90
6.3.2 Performance 91
6.3.3 Scalability 91
6.3.4 Security 92
6.3.5 Flexibility 93

7 Conclusions **94**
7.1 Summary 94
7.2 Contributions 95
7.3 Future Work 96

Bibliography **101**

Vita

List of Tables

2.1 Three service models of cloud computing 12

5.1 The classification of SugarCRM components 67
5.2 Characteristics of Amazon EC2 and RDS used in the case study 69
5.3 Characteristics of Amazon S3 used in the case study 69
5.4 Contents of SugarCRM migration package 70
5.5 List of the adding tenants' information 74

6.1 Tenant key (IP) list 85
6.2 The result of the experiment with a single tenant 86
6.3 The list of tenants settings in the experiment 87
6.4 The results of experiment with 10 tenants 87

List of Figures

2.1 Architecture of a typical three-tier client-server model 10

2.2 Three service models of cloud computing 12

2.3 Infrastructure as a Service 13

2.4 Platform as a Service 14

2.5 Software as a Service 15

2.6 Three levels of isolation implementation 16

2.7 Four-level SaaS maturity model 19

2.8 Architecture of a typical multi-tenant SaaS model 20

3.1 Simple solution for the service management 29

3.2 Improved solution for the service management 30

3.3 The A2SF runtime architecture 31

3.4 The general service request process in A2SF 33

3.5 End user authentication process of A2SF 35

3.6 The process of tenant status management 36

3.7 The process of service instance generation and recycle 38

3.8 The process of database data proxy 41

4.1 Main components in the implementation of A2SF 44
4.2 Class diagram of TCP proxy server 47
4.3 Class diagram of the HttpHandler module 49
4.4 Class diagram of the tenant manager module 50
4.5 Class diagram of the service instance manager module 52
4.6 Class diagram of MysqlHandler 56
4.7 Screenshot of application management centre homepage 58
4.8 Screenshot of application information edit page 59
4.9 Screenshot of tenant list page 60
4.10 Screenshot of tenant information edit page 61
4.11 Screenshot of tenant usage report page 62

5.1 Screenshot of the customized demo homepage before being migrated to cloud 65
5.2 The deployment diagram of the migrated SaaS system 71
5.3 Screenshot of tenant not found page 74
5.4 Screenshot of tenant “UNB” administrator’s profile page . . . 75
5.5 Screenshot of tenant “STU” administrator’s profile page . . . 76
5.6 Screenshot of proxy server logs 76
5.7 Screenshot of running instances in EC2 management console . 77
5.8 Screenshot of running instances after deleting tenant “STU” . 78
5.9 Screenshot of running instances after tenant “UNB” offline . . 79

6.1 The Composition of a response time before migrated to the cloud 81
6.2 The Composition of a response time after migrated to the cloud by the A2SF 82
6.3 The testing program 84

List of Abbreviations

Ajax	Asynchronous JavaScript and XML
AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
CRM	Customer Relationship Management
EC2	Amazon Elastic Computing Cloud
IaaS	Infrastructure as a Service
JDK	Java Development Kit
PaaS	Platform as a Service
RDS	Amazon Relational Database Service
S3	Amazon Simple Storage Service
SaaS	Software as a Service
SDK	Software Development Kit
SSH	Secure Shell
URL	Uniform Resource Location
VPN	Virtual Private Network

Chapter 1

Introduction

1.1 Motivation

As an emerging model for the delivery of computing resources as utility in a dynamic and scalable way, the cloud computing model is attracting more and more attention in the IT community. Based on the level of services, cloud computing has three types of service models, Software as a Service(SaaS), Platform as a Service(PaaS), and Infrastructure as a Service(IaaS). With the improvement of Internet infrastructure and the evolution of cloud computing, SaaS is rapidly becoming the standard software platform for many organizations that are seeking to reduce their IT costs and take advantage of cloud, such as flexibility, quick deployment, and scalability.

Nowadays most enterprise applications are based on the conventional client-server model. In this computing model, software vendors develop and sell

copies of installation packages and licenses to customers and then assist them to deploy the software in their own local IT infrastructures. For example, SAP's Business One is a typical software using this model.

While in the SaaS model, software is deployed on cloud and provides its services via networks. Customers no longer have to purchase or own the software package. Instead, they consume and pay for the services on demand. In this model, software vendors become service providers, and customers become tenants. The case of Salesforce's CRM [24] is a good example. One signal service instance can serve many tenants at the same time. Moreover, SaaS software rapidly supports onboarding new tenants, which is essential feature to a growing company.

Compared with the conventional client-server software model, the SaaS model has significant advantages. It gives both software vendors (service providers) and customers (tenants) more flexibility in investments of IT infrastructures and makes them achieve a higher profits margin by leveraging the scale of their business. In conventional computing models, IT infrastructures, including hardware and software, were purchased and maintained by customers. Moreover, it could take days (or weeks) to process a request for extra resources such as extra storages or additional servers. While in the SaaS model, the resources were provided in a service form and consumed on an "on demand" or "pay-as-you-go" basis. Tenants can pay software service providers ongoing fees, which usually are relatively low at first and smoothly increase or decrease as the business needs more or less capacity to scale up and down.

The SaaS model makes the economies of scale possible. By using virtualization and multi-tenancy technologies, an application instance can provide services to multiple end users of multiple tenants at the same time, which improves the efficiency in the usage of infrastructure and reduces the cost of unit services. For example, when a new tenant comes, the service provider can provide the appropriate services quickly just by modifying the configuration of applications instead of purchasing the new infrastructures for the new tenant. In addition, being different from other commodities, software products need more maintenance after selling, like patch updates and software upgrades. In the conventional way, software vendors need to maintain the software one by one or let the customers maintain them by themselves. In SaaS, by deploying the application on the cloud center, the service providers can maintain the software in one place for all the tenants. Besides, the SaaS model is also a good solution for piracy issues, which is also a long term problem for software vendors.

Recently more enterprises have been attracted to build their services or applications on cloud platforms and many of them successfully achieve their business objectives, such as Microsoft, Google, Apple, Salesforce, NetSuite [9], etc. From Gartner's survey in March 2012, worldwide SaaS revenue would reach $14.5 billion in 2012, which is a 17.9 percent increase over 2011's $12.5 billion. And this healthy growth rate is set to continue, according to the research company [10], which predicts a $22.1 billion SaaS market in 2015. Today, more enterprises embrace cloud computing by making their plans or

are in process of migrating their applications or services from their local IT environments to cloud computing environments. However, the application migration process from local infrastructures to the cloud environment turns out to be quite complicated. How to correctly and effectively migrate enterprise applications to cloud platforms poses a critical challenge for the research community.

The challenges of migration to cloud are not only from software management, such as redeployment process or cloud platform selection, but also from software developing techniques. This thesis focuses on the technical side of the migration challenges.

There are many challenges in the migration of client-server applications to multi-tenant SaaS based applications. Firstly, how to transit a single customer supported application to a multi-tenant supported SaaS system? Secondly, after the transition, how to make sure tenants and tenant data are secure? Thirdly, as the on-demand service supply is one of the biggest advantages of cloud computing, how to implement the self-adjust scalability of the migrated system to gain the economy of scale? Last but not least, how to control the cost and risk in the migration process, and how to maximally reuse the existing code of legacy applications and meanwhile gain the maximum of the advantages of cloud computing?

To meet the above challenges, the current approaches are usually either redeveloping the legacy applications on the PaaS platform or redeploying the existing mature applications on the IaaS platform without considering multi-

tenancy and scalability. Both of these approaches are insufficient. In the former approach, it is a waste of the mature legacy software asset. Moreover, the new techniques of cloud computing and various platform APIs make the cost and risk of redevelopment fairly high. While in the latter approach, the migrated applications cannot gain the basic advantages, like flexibility, quick deployment, and scalability from cloud computing. What even worse is that the efficiency of the migrated application might dramatically reduce due to the unpredictable network conditions of cloud.

1.2 Thesis Overview

In this thesis, we propose a general cloud migration framework for conventional client-server applications named Migration Framework of Application to SaaS (A2SF). The A2SF targets to software developers or vendors and helps them transform their conventional client-server applications to multi-tenant SaaS applications without changing the original code. By using A2SF, software developers can easily integrate their existing client-server applications with cloud infrastructures as an integrated multi-tenant SaaS system. The A2SF is composed by four major components: service proxy component, data proxy component, tenant management component, and service instance management component. The four framework components together with the server software of an original client-server application are deployed on the cloud as an SaaS application.

The service proxy and data proxy components of A2SF are multi-tenant awareness components [13] and provide the security for resource access to the migrated system. At runtime, instances of the server side application, as service instances, are dynamically allocated to different tenants by the service instance management. The service proxy component, as the front end of the deployed SaaS, identifies tenants from client connections and forwards client service requests to the corresponding service instances. Based on the source of a data access request, the data proxy component identifies its corresponding tenant and forwards the access request to the virtual database on the cloud which has been allocated to the tenant. All the data accesses from service instances are be intercepted by the data proxy.

The tenant management component is responsible for managing the tenant information and providing various services, like providing the tenant identification service to other components. The service instance management component can dynamically generate and recycle service instances for the tenants based on the application configuration script and the tenants' customization information.

In this thesis research, we implemented a prototype of the A2SF. We chose the Amazon AWS cloud platform as the prototype's base cloud environment. All the components of the prototype as well as the target application were running on the Amazon cloud.

Based on the A2SF, we also developed a general cloud migration process for client-server applications, as well as a case study of migrating a real-

world client-server application to the Amazon AWS using A2SF. Also, several experiments were conducted to verify and evaluate the A2SF framework.

1.3 Thesis Structure

The rest of the thesis is organized as follows: Chapter 2 introduces the background and literature review. Chapter 3 describes the architecture of the framework and the major components of the framework. Chapter 4 describes the implementation of the main components of the framework and discusses the approach and strategies about how to implement them on the Amazon AWS cloud. Chapter 5 presents a case study to show how to migrate a typical web-based client-server application to the cloud based on the A2SF. Chapter 6 describes several experiments to evaluate the A2SF framework. The thesis ends with the conclusions and future work in Chapter 7.

Chapter 2

Background

This chapter contains four sections. Section 2.1 introduces the conventional client-server model and its typical architecture. Section 2.2 introduces cloud computing and related concepts, such as the multi-tenant and the SaaS maturity model. In Section 2.3, from the architectural view, we analyze the challenges faced by the cloud migration of conventional client-server applications. Section 2.4 summarizes the related works on the cloud migration of conventional software.

2.1 Conventional Client-Server Model

The client-server model is one of the most popular and widely used architectures in enterprise software. The client-server model replaced the mainframe-terminal model in the 1980s. The functions of a client-server system can be

divided into the client side and server side, which usually run on different computers and communicate with each other via network connections. In this model, the client side application is implemented as a service requester which interacts with user and displays results. The server side is defined as a service provider that processes the requests and returns the results back to the clients [25]. Conventionally, both the client side and server side applications are deployed and used in the enterprise's local intranet. Nowadays, more applications are based on the world-wide Internet. For example, the WWW (World Wide Web) is a typical client-server system. Moreover, a web application is also called a browser/server application and is widely adopted in the IT industry. This web based client-server model consists of thin, portable, universal clients (browsers) and super fat web application servers. The communication between browsers and servers are based on hypertext transfer protocol (HTTP).

A typical design of a client-server application uses a three-tier pattern [7]: presentation tier, business logic tire, and data/resources tier. Figure 2.1 shows an architecture diagram of a typical three-tier client-server model. As shown in Figure 2.1, from left to right, there are clearly three tiers in this architecture: clients for presentation tier, application servers for business logic tier, and resource manager and resources for data/resource tier. In the presentation tier, clients interact with customers and send their requests to the application servers in the business logic tier. The application servers contain the business logic components which process client requests using the

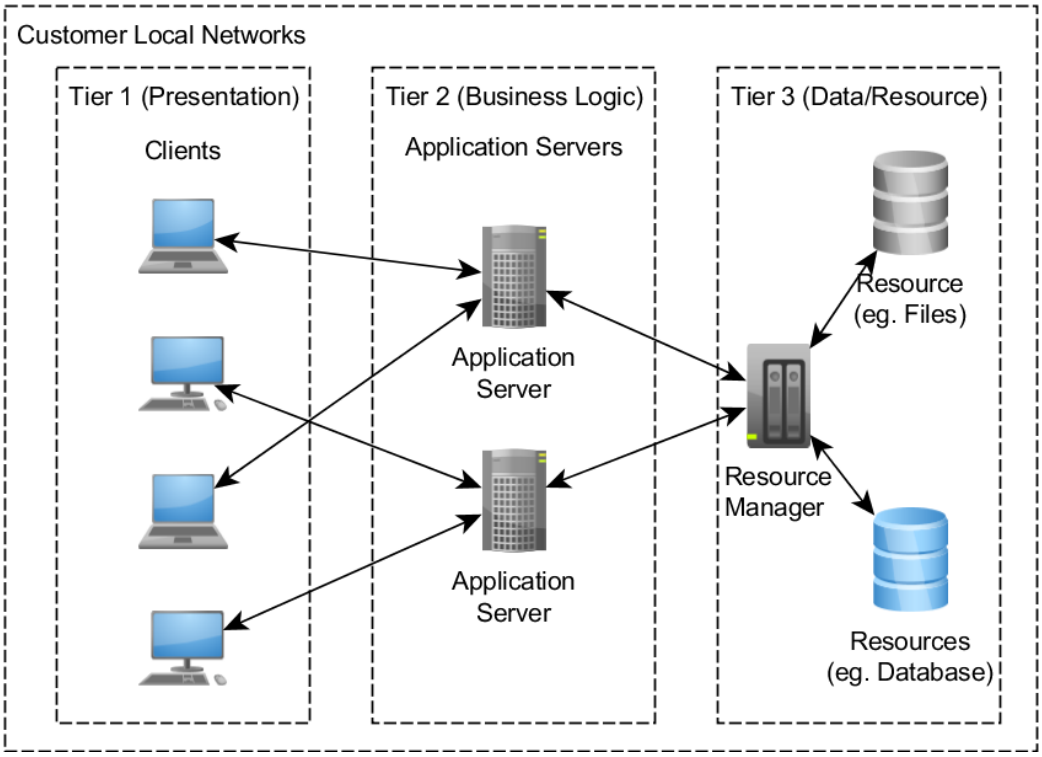

Figure 2.1: Architecture of a typical three-tier client-server model

data/resources in the data/resource tier and return the computing result back to the client. In the data/resource tier, resource managers such as MySQL database servers store and retrieve the customer's business data to support the application servers. The three tiers communicate with each other via network connections and work as together a whole system. Conventionally all these components are deployed on the customer's local networks and are used exclusively for individual customers, but not sharing among customers.

2.2 Cloud Computing

Cloud computing has received significant attention in recent years. According to a definition by NIST [18] (National Institute of Standards and Technology), cloud computing is:

"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

This definition also states that the cloud model has five essential characteristics: on-demand self-service, broad network access, resource pooling, rapid, and measured services. The core concept of cloud computing is sharing computing resources on a data center and delivering them as a utility to users on-demand. In fact, cloud computing originates from the traditional computing and network technologies like distributed computing, grid computing,

parallel computing, etc. Generally cloud computing services can be divided

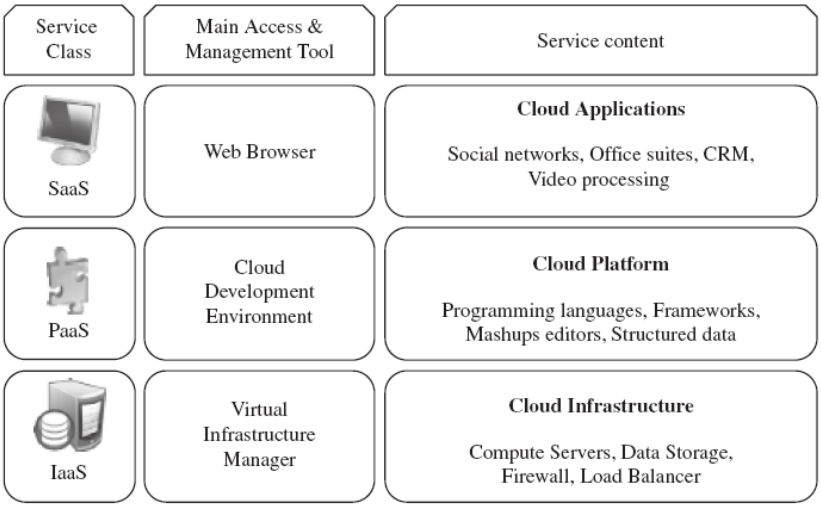

Figure 2.2: Three service models of cloud computing

into three service models. From lower level to higher, they are Infrastructure as a service, Platform as a service, and Software as a service, as shown in Figure 2.2. Each model is based and implemented on its lower model. Table 2.1 gives brief descriptions of the three cloud service models.

Table 2.1: Three service models of cloud computing

Model	**Description**
IaaS (Infrastructure as a Service)	Hardware is delivered as a service, including processing power, storage, bandwidth, etc.
PaaS (Platform as a Service)	Programming platform and tools like Java, MySQL, Python and APIs are delivered as a service.
SaaS (Software as a Service)	Applications or software are delivered as a service.

2.2.1 Infrastructure as a service

By supplying basic computing resources, Infrastructure as a Service is sometimes called Hardware as a Service (for instance, the capacity of processing, network equipment, data center spaces or servers). In addition, the scale of the resources supply can be up and down according to users' requirements. This can happen "as a result of the rapid advances in hardware virtualization" [32] as shown in Figure 2.3 [2]. This model especially meets the needs

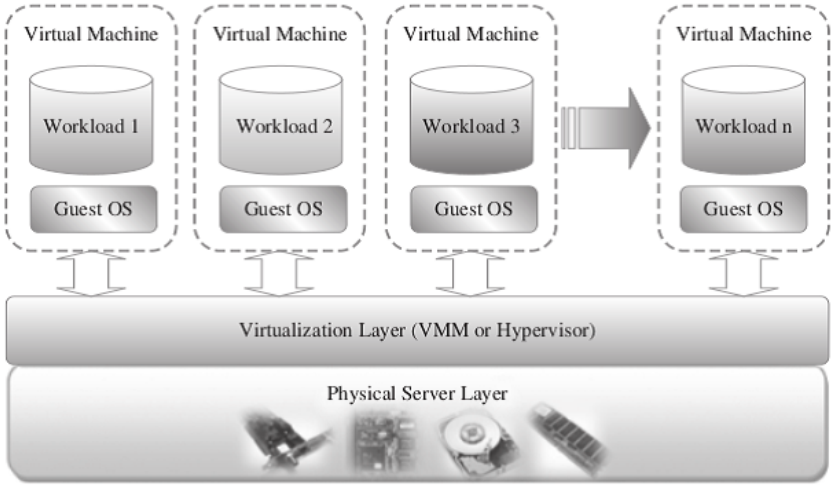

Figure 2.3: Infrastructure as a Service

of enterprise users as it relieves them from needing to spend budget on building and maintaining data centers [31]. Examples of IaaS are Amazon Elastic Cloud (EC2) [8] and Amazon Simple Storage Service (S3) [23].

2.2.2 Platform as a service

Platform as a Service is a bridge between hardware and application [11]. It provides software developers a development platform from the Internet which contains all resources and environments required to build applications. Developers can build their own applications on the PaaS platform and deliver them as the SaaS Software as shown in Figure 2.4 [26].

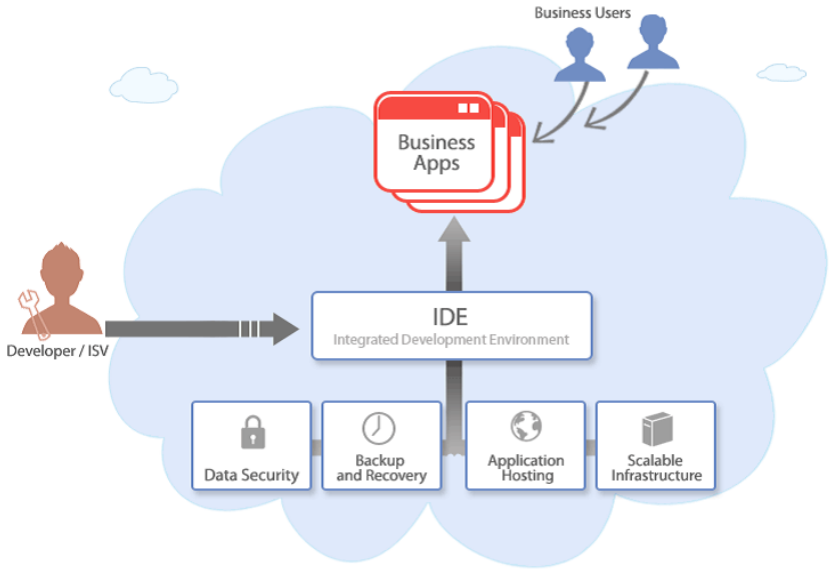

Figure 2.4: Platform as a Service

The responsibility of PaaS is to provide cloud APIs to developers and execute their applications on the cloud. It is also in charge of the responses to external requests, as well as running scheduled jobs included in the application [26]. Since the platform is transparent to end users, PaaS can share application servers among end users and dynamically scale the resource allocation when

the loads increase or decrease. Recently, more PaaS products have come out, such as Google App Engine [12], and Microsoft Azure Services Platform [20].

2.2.3 Software as a service

Software as a Service also is referred as Application as a Service. In this model, software is hosted in the cloud, and software services are provided to different customers via the internet with the "pay-as-you-go" charging model, as shown in Figure 2.5. In this model, end users can use software

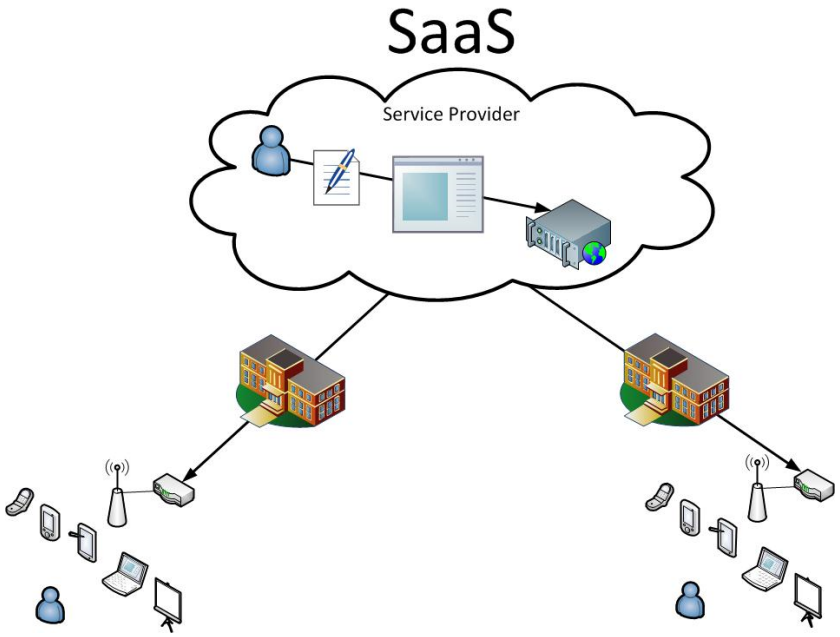

Figure 2.5: Software as a Service

services anywhere and anytime from various types of devices, such as PCs, phones, and tablets, without being concerned about installation and main-

tenance of the software [31]. Some examples of SaaS software are Salesforce Customer Relationship Management (CRM) [24], NetSuite [9], and Google Office Productivity applications [14].

2.2.4 Multitenancy

Multitenancy is one of the fundamental technologies of cloud computing for sharing infrastructure and resources cost-efficiently among multiple customers (tenants). With multitenancy, an application is designed to virtually partition its data and configuration, so that clients of a particular customer tenant can work with a customized virtual application instance. Although all tenants share the same software, each tenant feels like he/she is the sole customer of the software.

There are three levels to implement the multitenancy isolations: physical-level isolation, virtualization-level isolation, and application-level isolation [13]. Figure 2.6 illustrates the implementation of multitenancy isolation in the cloud.

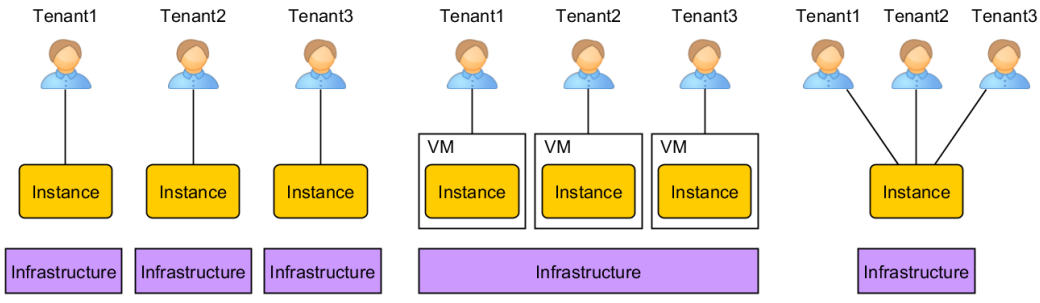

Figure 2.6: Three levels of isolation implementation

In the physical-level isolation, software is duplicated and deployed on different physical computers. There is no resource sharing among the tenants. This kind of isolation has higher security than the other two types of isolation. However, due to the lower resource sharing, the hardware cost for SaaS providers is also relatively higher. The physical-level isolation is suitable for large corporation tenants with higher security requirements.

In the virtualization-level isolation, by using the virtualization technology, the infrastructure is shared by virtual machines which are logical computers but can be used in the same way as physical machines. The SaaS software is deployed on virtual machines. In this model, there are almost no change needed to migrate conventional software to virtual machines. This model has limited resource sharing and scalability.

The application-level isolation is implemented based on programming control. As shown in Figure 2.6, one application instance serves all the tenants simultaneously. So the key point of implementing an multitenancy SaaS application is to have a well designed tenant isolation architecture. This model is the most efficient multitenancy model. However, here multitenancy supported design also means that it maybe very hard to migrate a legacy application to a native multitenancy application, unless it is being re-developed.

In summary, the multitenancy technology is essential to the SaaS applications. The choice of isolation levels depends on the balance between the benefits and cost (including risk). The physical-level isolation has the highest security but the lowest scalability and efficiency of resource sharing and

reuse. The application-level isolation has the highest scalability and efficiency of resources sharing and reuse, while applications in this model are more complex in development and maintenance, also making the cost and risk higher. The virtualization-level isolation is between these above two models. In this thesis, the A2SF framework uses a hybrid of the virtualization-level isolation and application-level isolation.

2.2.5 SaaS maturity model

A well-designed SaaS application should be scalable, multi-tenant efficient, and configurable. Configurability, multi-tenant efficiency, and scalability are three key attributes of SaaS applications. We can classify SaaS applications from poorly designed to well-designed. Fred [6] presents a four-level SaaS maturity model that explains and puts into perspective of the above key attributes of SaaS. Generally, each level is distinguished from the previous one by the addition of one of the three attributes listed above. Figure 2.7 shows the SaaS maturity model. As shown in Figure 2.7, in the two basic levels (Level 1 and 2), lower layer IT infrastructure resources are shared. In Level 3 and 4, multitenancy is the core feature with a single instance of application serving multiple customers (tenants) to achieve the minimum operational cost. In this thesis, A2SF uses the combination of Level 1 and Level 4.

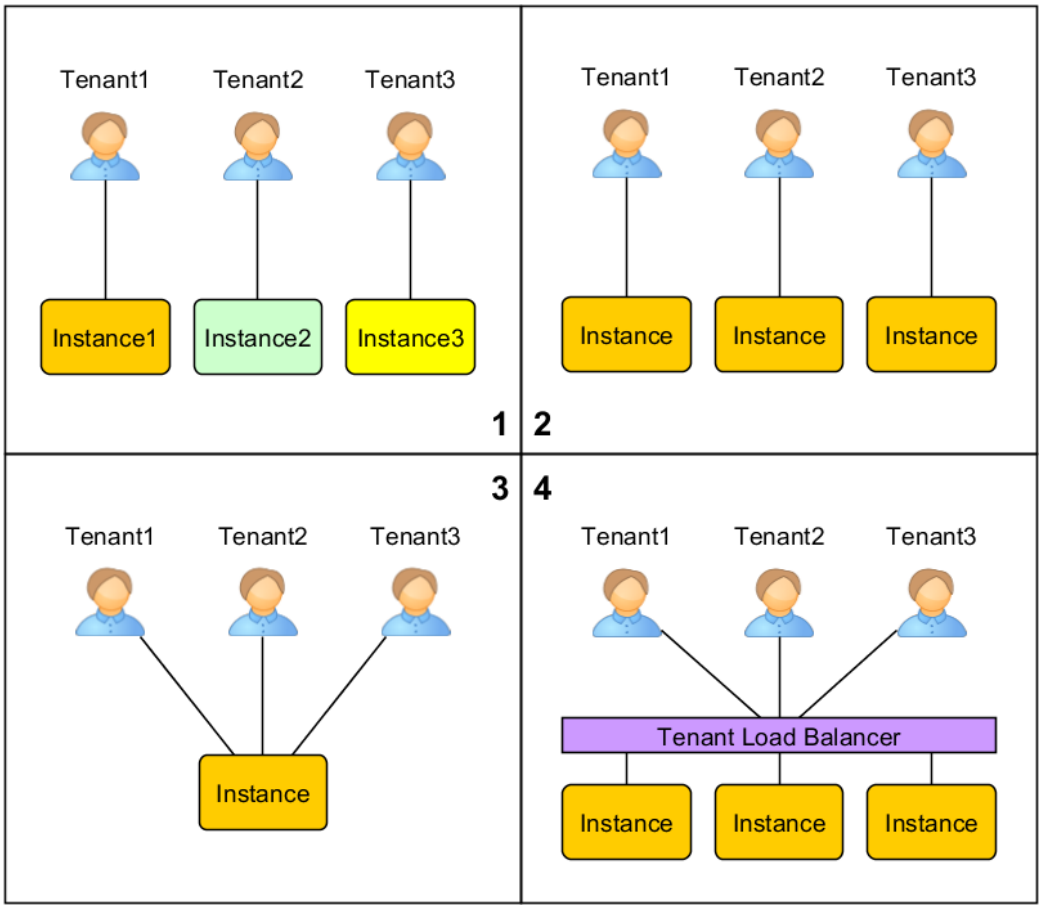

Figure 2.7: Four-level SaaS maturity model

2.3 Challenges of Migration

The challenges of migrating conventional client-server software to the cloud are not only from software operation and maintenance, such as cloud platform selection and software redeployment on the cloud, but also from the huge differences between these two architectures.

In Section 2.1, we present the architecture of the conventional client-server model. As a conventional computing model, most of client-server model applications are designed to serve only one customer. Meanwhile, SaaS software emphasizes the ability of resources sharing, flexible scales, and multi-tenant serving. Figure 2.8 is an architecture diagram of the multi-tenant SaaS model.

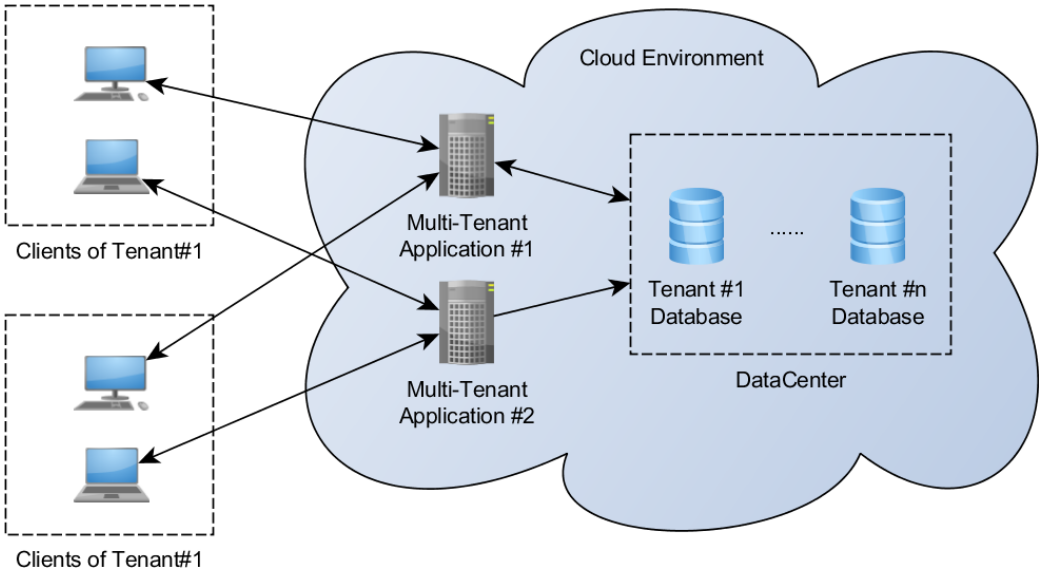

Figure 2.8: Architecture of a typical multi-tenant SaaS model

As shown in Figure 2.8, in the multi-tenant SaaS model, one service (application) always serves multiple tenants (customers). Application 1 serves many clients from different tenants (tenant1 and tenant2) at the same time. The business data of different tenants are also stored in the same data center and even in the same database.

From the architectural point of view, there are three major challenges in the cloud migration of client-server applications to multi-tenant SaaS applications.

1. **Challenge on tenant management and identification**

   One of the essential features of SaaS software is multi-tenancy, that is, the SaaS model software should be able to serve clients from multiple tenants at the same time. So the migrated system is required to identify tenants and manage their information and customizations. Meanwhile, the client-server applications are designed to be used by only one customer, without consideration of being shared by multiple tenants.

2. **Challenge on tenant isolation and data security**

   As an SaaS application can serve many tenant at the same time, the privacy protection and data isolation among different tenants should be supported. Although all tenants essentially share the same application code and infrastructure, they should not affect with each other during the running time. The first issue is data security which also means how

to protect tenants' private data. On one hand, tenants should be only allowed to access their own data. On the other hand, the execution of handling a request inside the application should also be isolated to avoid the runtime data corruption. The second issue is about availability and performance. For example, if one of the tenants crashes its service by some fatal operation, other tenants should not be affected at the same time.

3. **Challenge on dynamic scalability**

   As an SaaS application runs on the cloud environment, the migrated system should have the ability to self-adjust on scales based on the demand of online live tenants. For instance, when the number of online live tenants increases, the application should also allocate more cloud resources and increase the service capability. Whereas, when the number goes down, the application should release the spared cloud resources as well.

2.4 Related Works

Recently, the migration of conventional applications to multi-tenancy software has received more attention by academia and industry. In the past few years, much research related to cloud migration was conducted.

In 2011, Gartner [17] analyzed five ways of migrating existing applications to the cloud: re-host on IaaS, refactor for PaaS, revise for IaaS or PaaS, rebuild on PaaS, and replace with SaaS, and gives advices on how to choose among the methods. Amazon [1] [30] also gives its migration instruction about migrating an existing system to Amazon cloud.

Guo et al. [13] proposed a multi-tenancy enabling programming model and framework which contains a set of approaches and common services that support and speed up multi-tenant SaaS application development.

Cai et al. [3] [4] proposed an end-to-end methodology and toolkit for transforming existing web applications into multi-tenant SaaS applications. However, while using the toolkit, the developers have to modify the original source code as well as the server configuration.

Song et al. [27] defined a SaaSify Flow Language and proposed an SFL tool which would help convert Java web applications to SaaS applications. However, this tool also has its limitations. First, the SFL tool seems only to support Java web applications with JDBC. Second, since the SFL tool uses memory in thread to keep and share tenant information among layers, it is reasonable to believe that the tool could not support an application whose

components are deployed on different computers.

Enabling applications to support multi-tenancy either during application development or by adapting existing web applications to support multi-tenancy has also been investigated by several research project such as [5] [15] [16] [19] [22]. Generally speaking, these projects have made good progress in multi-tenant SaaS migration to cloud.

In this thesis, we propose a migration framework for easy migration of applications to SaaS, to help software developers or vendors transform their conventional client-server applications to multi-tenant SaaS applications without revising the original code. Migration easiness is the key of our proposal.

Chapter 3

Framework Design

This chapter describes the architecture design of the A2SF framework. We first give an overview of the framework and analyze its requirements in Section 3.1. Then in Section 3.2, we propose the runtime architecture of the framework and describe how its major components work together.

3.1 Overview

The aim of the A2SF is to design and implement a general cloud based multi-tenancy framework which can be used for migrating conventional client-server applications to the cloud in an easy way and without modifying the original code.

3.1.1 Requirements

As discussed in the previous chapter, due to the differences between the conventional client-server model and the SaaS model, there are three major challenges for the migration: tenant management and identification, tenant isolation and data security, and system scalability.

The A2SF should be able to help software developers overcome these three challenges easily. In other words, the A2SF should fulfill the following requirements: (i) The framework should provide a tenant management subsystem that works well with legacy client-server applications. (ii) The framework should also provide solutions for tenant isolation and data protection. (iii) The framework should be able to dynamically adjust the scale of the migrated system on demand. In addition, as a non-functional requirement, the framework should make the migration task as easy as possible. The followings are the required tasks for A2SF:

1. **Tenant management and identification**

   Generally speaking, a conventional client-server application is designed for being used exclusively by users of the customer's organization. It usually has an authentication module and customization module, but does not have a tenant management module, even the concept of tenant. So for migration of a conventional application, the first step should be adding the tenant identification and management module to the migrated SaaS application. For more details, when a service request

arrives at the application servers, the migrated SaaS application should be able to identify which tenant the service request belongs to. Then, after identification, the migrated SaaS application provides the customized service to this client request by handling the request based on its tenant data which the client belongs to. Furthermore, the migrated SaaS application should provide an interface to allow tenants to customize their applications based on their own needs.

In order to give the migrated SaaS system multi-tenancy awareness, the framework should provide a general tenant management subsystem to manage tenant information and service configuration. For tenant identification, the framework should give each tenant a token which is carried by each service request from clients of this tenant. By the carried token, the tenant identification module can tell which tenant the service request is from. This module should work with the authentication module of the original application and provide identification and authentication functions for the migrated SaaS system.

2. **Multiple isolations for data security**

   To support multiple tenants simultaneously, SaaS tenant privacy protection is an essential requirement for the migrated SaaS system. In the A2SF, multiple approaches of isolation should be applied and implemented.

   One isolation method should be virtualization-based isolation. By this

isolation method, in the migrated SaaS system, the runtime data and configuration of different tenants should be stored in different virtual machines.

The other method should be application-based isolation. As discussed in the previous chapter, logically, there are three tiers in a client-server application, and these tiers are independent from others and only communicate with each other via the network. The A2SF should provide access control layers between the three tiers. The access control layers should be able to tell which tenant the resource access request is from, and allow or deny the resource access request based on tenant's identity.

The framework should provide two types of multi-tenant awareness layers for tackling the access control challenge: service proxy layer and data access layer. The service proxy layer should work between the presentation tier and the business logic tier and be responsible for load balancing and service access control. Meanwhile, the data access layer should work between the business logic tier and the data/resource tier for data/resources access control. In this way, the migrated SaaS system can support application level isolation among different tenants.

3. **Automatic service resource management for scalability**

   A2SF should run on the cloud and for each tenant generate a customized service instance on a different virtual machine. However, if

A2SF simply assigns each tenant a virtual machine and deploys the tenant-customized application on it, as shown in Figure 3.1, the migrated SaaS system only matches the first level in the SaaS maturity model. Such migrated SaaS system would not be able to achieve high scalability and reduce operation cost.

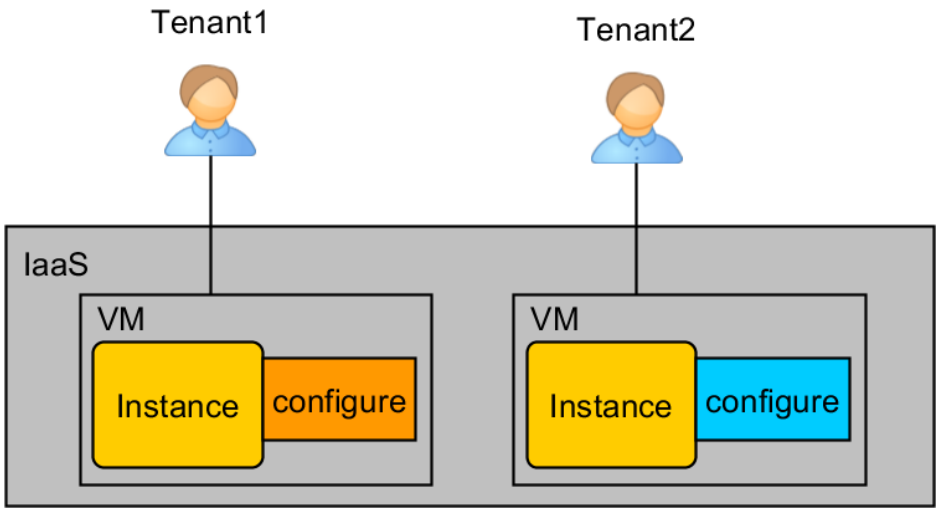

Figure 3.1: Simple solution for the service management

In the cloud environment, the resources possessed by the migrated SaaS system are allocated and released in the unit of service instance. So the framework needs a service instance management module which is responsible for generating and recycling service instances for different tenants. As shown in Figure 3.2, working as a load balancer between client and service instances, the A2SF will help the migrated SaaS system allocate and recycle resources automatically. In this way, the

migrated SaaS system can partially achieve level 4 in the SaaS maturity model without revising the original code.

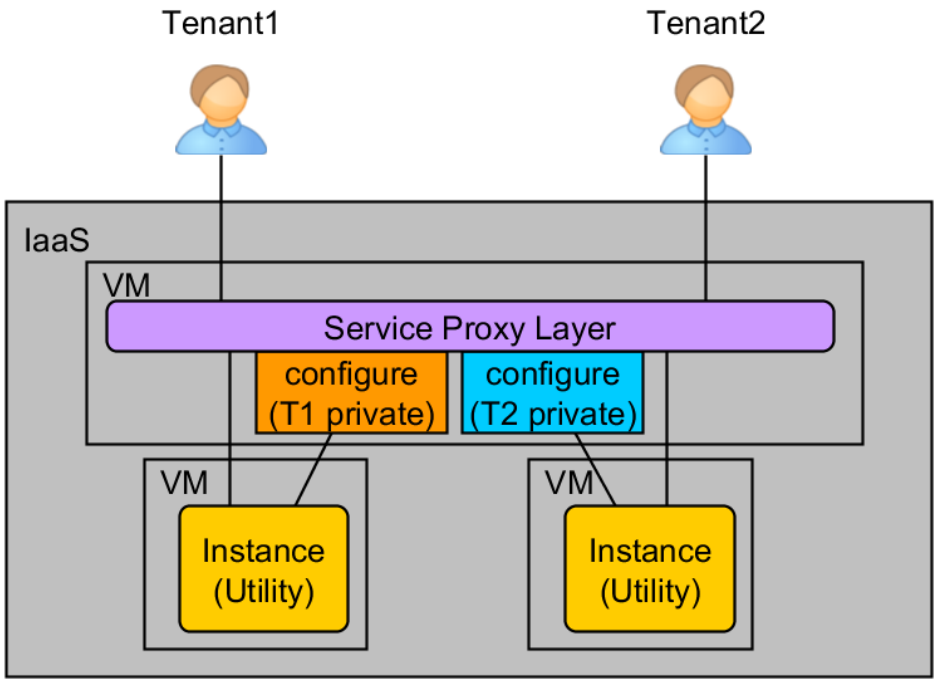

Figure 3.2: Improved solution for the service management

3.2 Architecture Design

The A2SF aims to be a multi-tenancy supported framework for cloud migration of conventional client-server applications. Figure 3.3 depicts the A2SF runtime architecture, as well as the overall A2SF and key components. There are four major components in the framework, including the Service Proxy Layer, Tenant Manager, Service Instance Manager, and Data Access Layer.

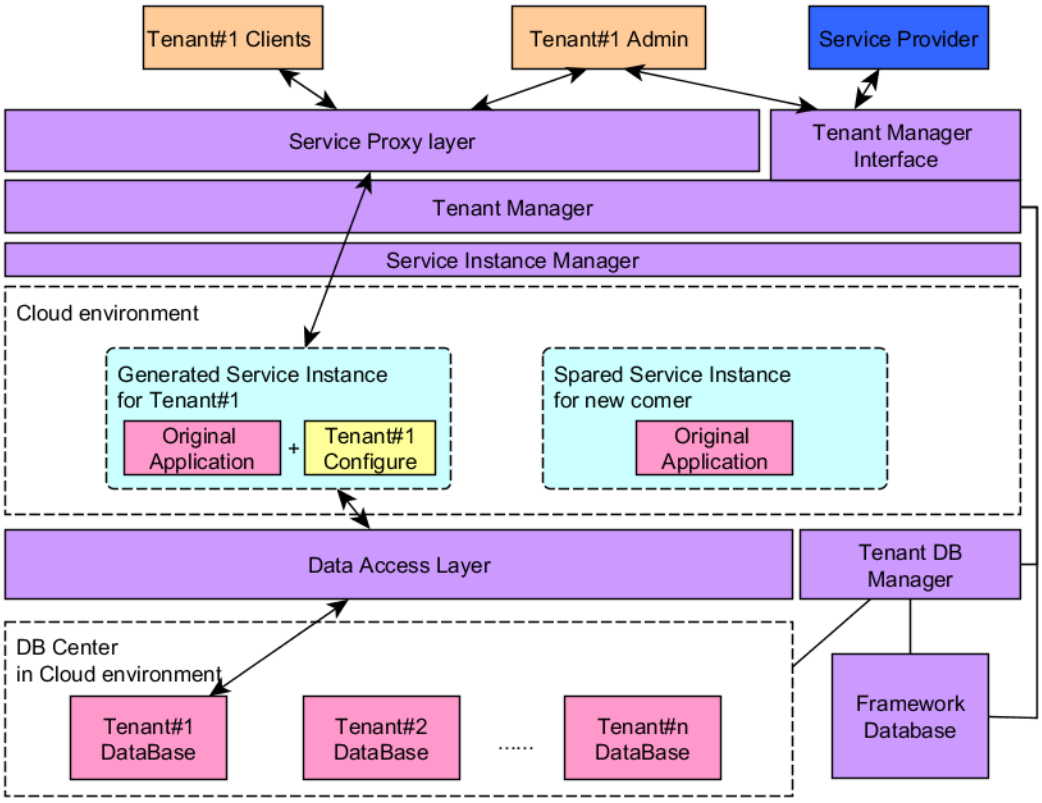

Figure 3.3: The A2SF runtime architecture

3.2.1 Service proxy layer

The service proxy layer is responsible for service access control. In the A2SF, tenants are only allowed to access their own service instances. As shown in Figure 3.3, the service proxy layer is the entrance of the migrated SaaS system, and all the end users of tenants send service requests to this proxy. When the service proxy layer starts up, it loads all the tenants' proxy rules. After receiving a end user request, the service proxy layer firstly identifies from which tenant this request is by invoking the tenant manager's identification service. Only a requests form a registered tenant can be identified. Once the tenant of the request identified, the service proxy layer handles the request based on the tenant's proxy rules and forwards the request to a proper service instance. If the request cannot be identified, the service proxy layer denies the service request and returns an error message as a result.

3.2.1.1 General service request process

As the targeting application will be migrated to the cloud as a whole package without revising the code, the original communication protocols as well as the business logic processes between clients and server do not need to be changed in the migrated SaaS system. However, the new processes in the migrated SaaS system are also different. In the new processes, each service request will have its identification check first, and then, the request will be processed and dispatched to the proper service instance. Figure 3.4 shows the general process of handling a service request in the migrated SaaS system.

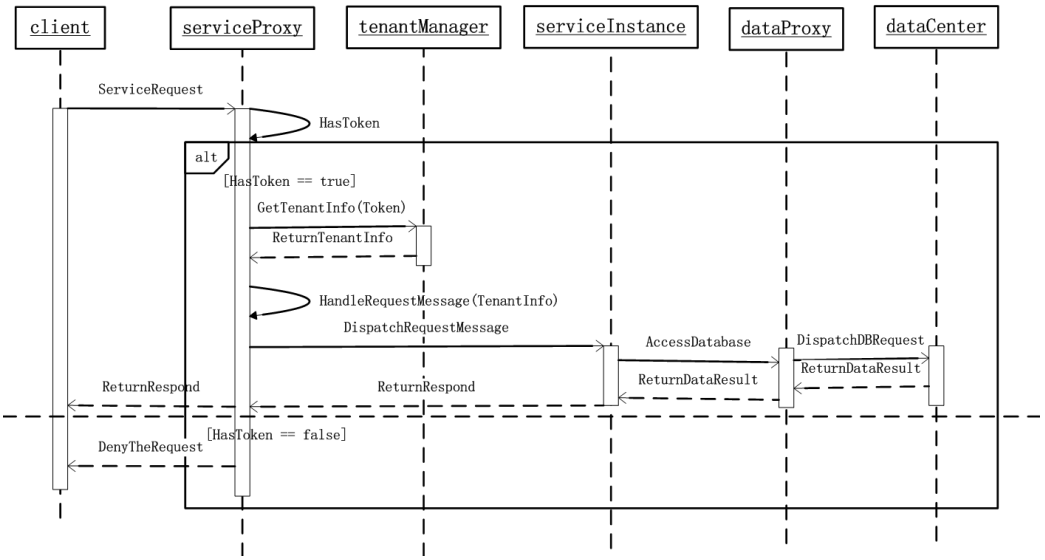

Figure 3.4: The general service request process in A2SF

As shown in Figure 3.4, a service request with its tenant token is sent from a client and will be caught by the service proxy layer. This layer will firstly determine whether the request has a valid tenant token or not. If the request does not have a valid tenant token, the service proxy layer then refuses further services. If the request has a valid tenant token, the service proxy layer will send a tenant identification request to the tenant manager component to get the tenant's information, including the proxy rules and the service instance information. Based on the tenants' information, the service proxy layer will modify the service request message and deliver it to the corresponding service instance. After receiving the service response from the service instance, the service proxy layer then returns the response to the requesting client.

3.2.2 Tenant manager

The tenant manager provides the following tenant management services: tenant identification service and tenant status service.

1. The tenant management services will be used by the software service provider which provides and manages the migrated SaaS system and the tenant administrator which manages all tenants. The software service provider can carry out CRUD operations on the tenants and set the customized rules for them. The tenant administrator can customize its service instance and view the service usage report for each tenant.

2. The tenant identification service is responsible for verifying the tenant token and returning the tenant's information.

3. The tenant status service is used to get the tenants' current statuses. Based on the tenants' current statuses, the A2SF is able to manage cloud resources by adjusting the number of service instances allocate to the tenants.

3.2.2.1 Tenant identification process

As discussed in the previous sections, tenant identification is required on every client service request. After the tenant identification is confirmed, the service request is delivered to the corresponding customized service instance, typically for user authentication. When the end user logins in to the migrated

SaaS system, the login request will be sent to the service proxy layer first, then the service proxy layer will identify which tenant this end user belongs to. After identified, the username and the password carried by the login request will be checked by the original system's authentication module. And the result will be sent back to the end user via the service proxy layer. Figure 3.5 shows the process of end user authentication in the migrated SaaS system.

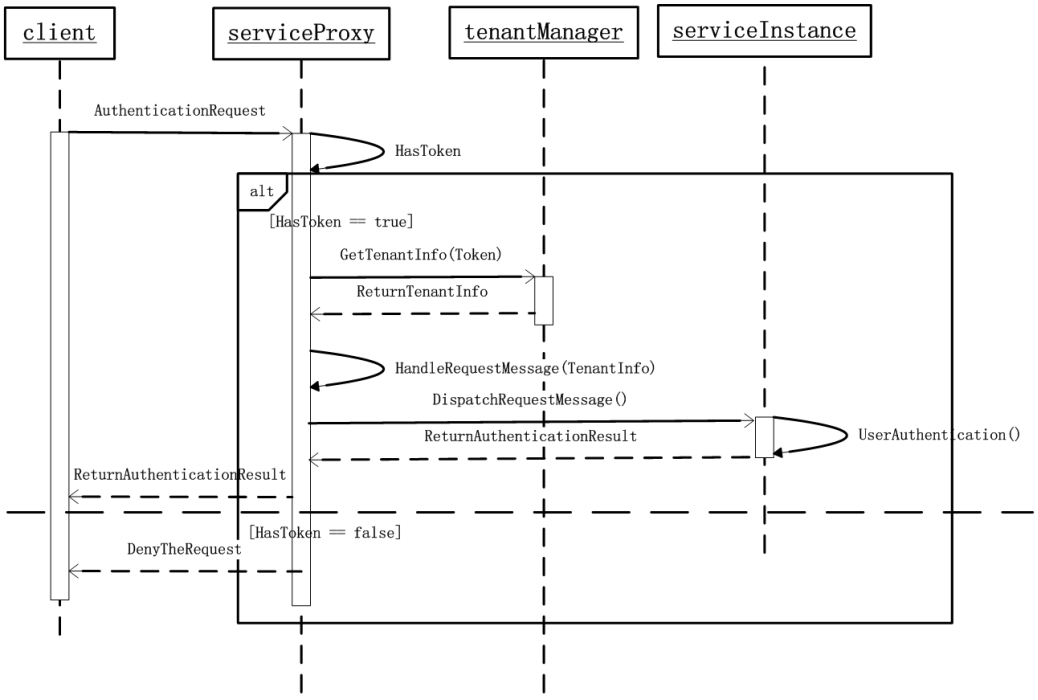

Figure 3.5: End user authentication process of A2SF

3.2.2.2 Tenant status management process

Tenant status management updates the tenants' statuses based on the tenants' service requests and the connections. In the A2SF, tenant status information includes tenant service instance status, tenant connection number, tenant alive time, and so on. Based on tenants' statuses, the framework will adjust the current number of service instances. Figure 3.6 shows the process of tenant status management.

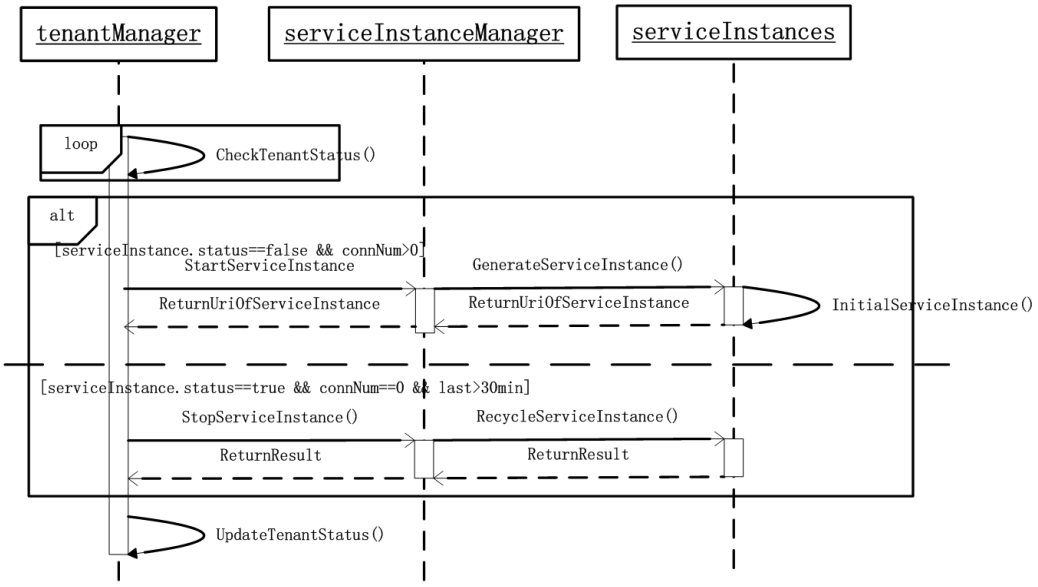

Figure 3.6: The process of tenant status management

As shown in Figure 3.6, the tenant manager loops the tenant list by same time interval to keep the tenant status updated. For a tenant, if there is no service instance running, but has some client connection requests waiting, the tenant manager will invoke the "start service instance" method of the

service instance manager to generate a customized service instance for this tenant. If the number of tenant connections is zero, and the idling time of its service instance is more than 30 minutes, the tenant manager will recycle the service instance of this tenant.

3.2.3 Service instance manager

The service instance manager is responsible for managing the running service instances, including service instance generation and recycling. This module wraps the operations of the IaaS services, as well as the remote controlling operations of the virtual host provided by the underlying cloud providers. In this way, the framework can be adapted with multiple cloud services, without a lock-in of the particular cloud service provider. Moreover, as the virtual machine launching time maybe too long for a real time responding system, in order to shorten the service instance generating time, the framework also implements a virtual machine pool in the service instance manager and always keeps a number of spare vms waiting for generating service instances for more tenants.

3.2.3.1 Service instance generation and recycle process

The service instance allocates for a tenant, as the server side for that tenant's application, composed of a running virtual machine and a customized server software is deployed on the virtual machine.

In A2SF, each online tenant which has clients or end-users accessing the mi-

grated SaaS application is assigned a customized service instance. Whether or not to generate a service instance is controlled by the tenant manager. As discussed in the previous section, the tenant manager keeps the tenants statuses and updates them regularly. When a tenant status matches its rule of generating or recycling the service instance, the tenant manager will invoke the services provided by the service instance manager to generate or terminate the service instance for the tenant.

In this framework design, we intentionally separate the cloud manager (in our implemented prototype, it is named EC2Manager) from the service instance manager, so that our framework can be easily supported by different cloud service providers. Figure 3.7 shows the process of service instance generation and recycling.

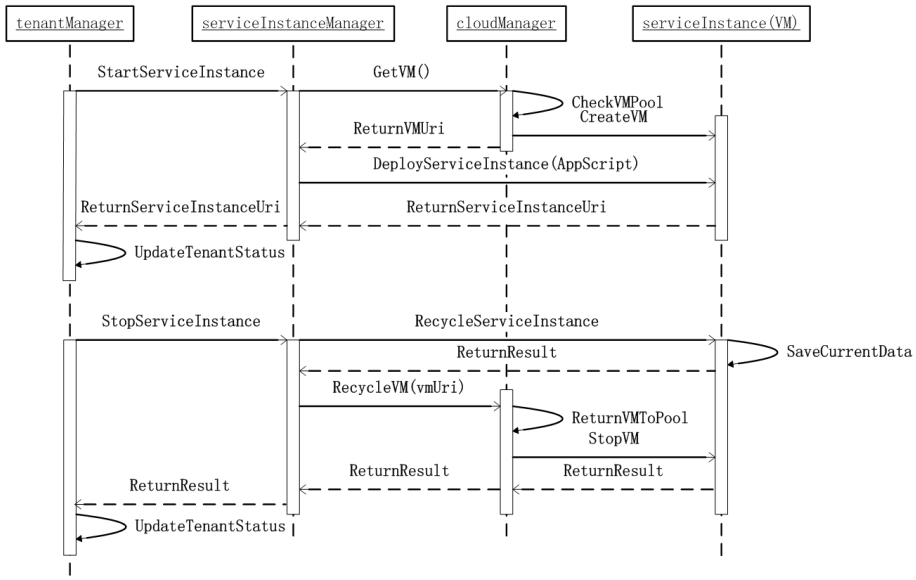

Figure 3.7: The process of service instance generation and recycle

As shown in Figure 3.7, the generation and recycling processes start with the tenant manager. And the service instance manager completes the details of the operations.

1. **Service instance generation steps**

   When the service instance manager receives a service instance generation request for a returning tenant, firstly, the service instance manager gets an available virtual machine from the cloud. Secondly, the service instance manager deploys the original application on the virtual machine. Thirdly, the service instance manager customizes the original application by loading the tenants private data to the virtual machine from the A2SF's data centre and initiates the deployed application. Finally, the service instance manager sends the result and service instance information to the tenant manager to update the tenants statuses.

2. **Service instance recycling steps**

   When the service instance manager receives a service instance recycling request for an off-line tenant, firstly, the service instance manager stops the application. Secondly, the service instance manager collects the tenants local private data on the virtual machine instance and saves it to the A2SF's data centre. Finally, the service instance manager clears the tenants local private data, terminates the virtual machine instance, and sends the result to the tenant manager to updates the tenants status.

In order to perform the above steps, the service instance manager module wraps the standard operations of the IaaS services (launch, start, stop, and terminate a virtual machine), as well as the remote controlling operations or commands (such as copy, move, service start, service stop and so on) on the virtual machine. Moreover, as the virtual machine instance launching time may be too long for an application that needs quick response time, in order to shorten the generation time of service instances, the A2SF also implements a virtual machine pool, managed by the service instance manager, to always keep a number of spare virtual machines waiting to host generated service instances.

3.2.4 Data access layer

The data access layer is responsible for controlling data access as from service instances to assure that tenants can only access their own data.

Generally speaking, an application has two essential parts, program code and user data. The program code is the common part which can be shared with all tenants, while the user data belongs to tenants individual which should be protected under the access control. Furthermore, the user data can be categorized into local data and database data. The local data are stored in local files. For instance, the configuration file is a typical example of local data, while the database data is always stored in a database.

The data proxy layer stores local data in the A2SF's data storage component (such as S3 in the Amazon Cloud), initializes local data before the service

instance runs, and stores and clears local data after the service instance stops. Whereas the data proxy layer provides two ways to deal with the database data. One way is to implement a data proxy server. All data accesses go through the data proxy server first. This method is for those applications in which the database connection strings are hard coded. The other way is to treat the data connection configuration file as a tenant's private local data. This method is only suitable for those database configurations that are separated from code and can be easily replaced.

3.2.4.1 Database data proxy process

The data proxy server provides a unified access point to tenants' databases, and forwards the data request to the proper database based on the provided tenants' mapping rules. Figure 3.8 shows the database data proxy process.

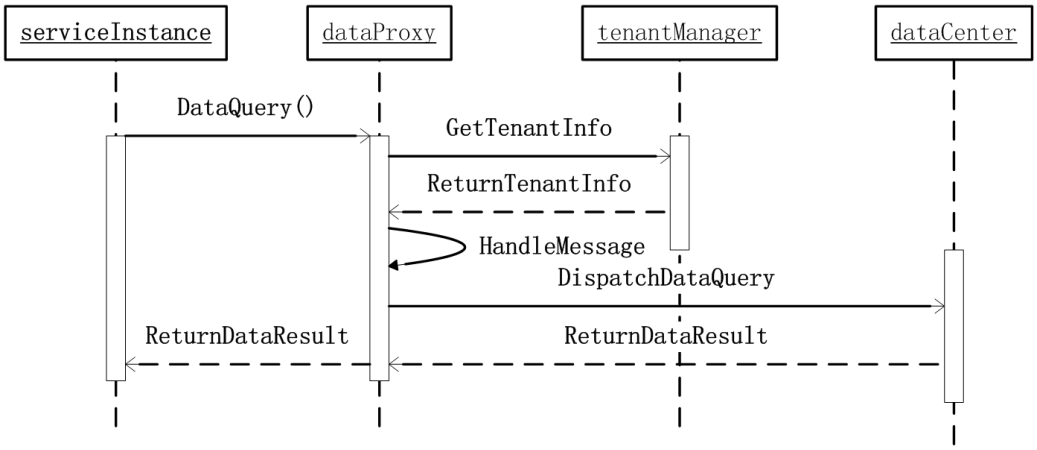

Figure 3.8: The process of database data proxy

As shown in Figure 3.8, when a service instance executes a data query, the data proxy will get the request first. Then the data proxy identifies which tenant this service instance belongs to. After that, based on the tenant information, the data proxy dispatches the data request to the proper database and passes the data result back to the service instance.

Chapter 4

Implementation

In this chapter, more details about the framework and system implementation will be expanded. Section 4.1 gives an overview of the implementation of A2SF framework as well as the developing environment. Section 4.2 shows more details of the key components, including the design and implementation and all modules.

4.1 Overview

As introduced in Chapter 3, the A2SF contains four major components: service proxy layer, tenant manager, service instance manager, and data access layer. As shown in Figure 4.1, in our prototype implementation of A2SF, there are also four parts: two runnable jar files named "Service Proxy Server" and "Data Proxy Server", one war file named "Application Man-

agement Centre" which is a web application, and one database scheme in MySQL database.

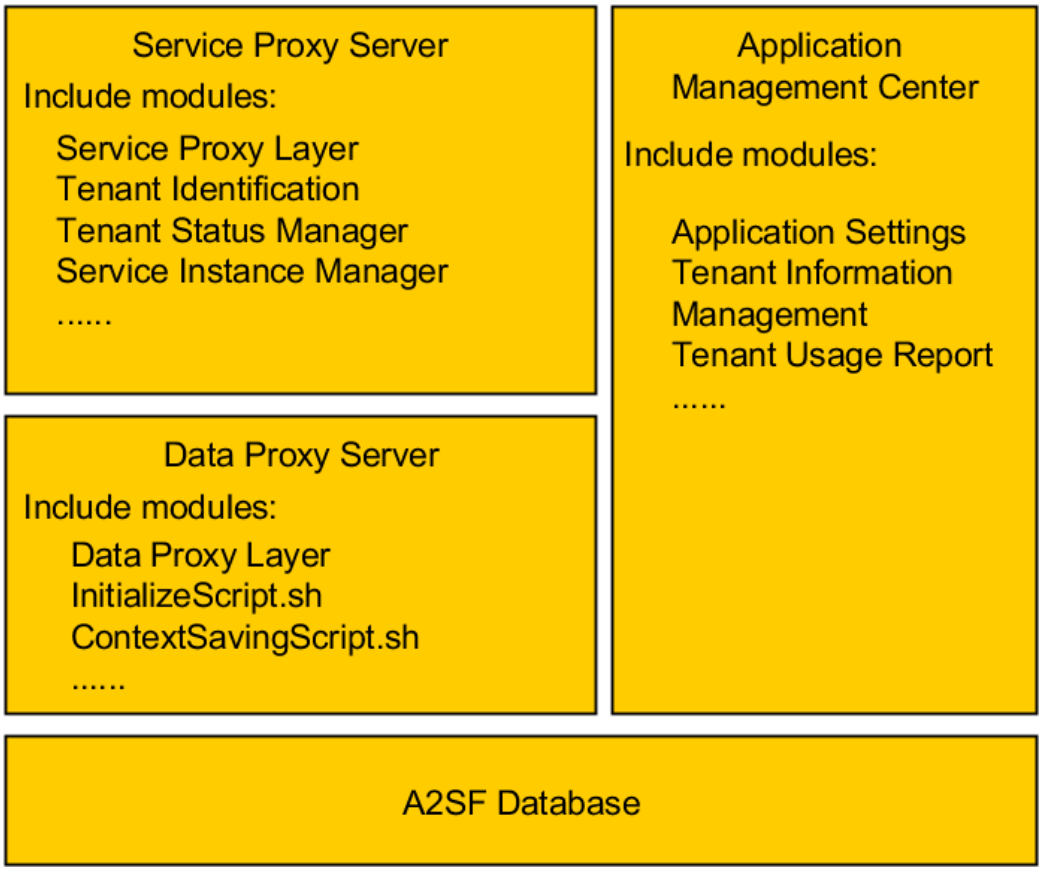

Figure 4.1: Main components in the implementation of A2SF

As shown in Figure 4.1, the modules of the service proxy layer, the service instance manager and part of the tenant manager in the design view are implemented in the service proxy server component. The data access layer is implemented in the data proxy server component which supports the two working modes: local proxy mode and database proxy mode. As proxy

servers, the both service proxy server and the data proxy server are implemented based on a TCP proxy core-server whose details will be explained in the following sections.

Application Management Centre is a web application for managing the migrated SaaS application, which has two main functions. One is for tenants to manage their information and view their service usages. The other is for the SaaS provider to manage tenants and application settings.

The following is the developing environment of A2SF. To implement A2SF framework, we chose Eclipse IDE 4.2 with Apache Tomcat 7, Java JDK 7, and Amazon AWS SDK1.6 as the developing environment. We chose MySQL as the database of A2SF. In the implementation, we used the programming language Java, and three Amazon cloud services applied: EC2 service [8], S3 service [23], and RDS service [21]. All the implemented framework components were running on the Amazon Linux operation system.

4.2 Component Implementation

As described in the previous section, the prototype of A2SF has three main parts: the service proxy server component, the data proxy server component, and the application management centre component. In the implementation, the service proxy server and the data proxy server components are similar and both of them are implemented based on a TCP proxy core-server. The application management centre component is a Java web application, which

is responsible for managing the migrated SaaS system, including tenant information and the framework settings. In this section, we will first describe the implementation of the TCP proxy server, and then introduce the composition of the service proxy server component, the data proxy server component and the application management centre component as well as their implementation.

4.2.1 Implementation of TCP proxy server

A client-server application is always a network based application. As known, the network model is layered. The well-known network model is the OSI framework which has seven layers (physical layer, data link layer, network layer, transport layer, session layer, presentation layer, and application layer), and the higher layer is implemented based on the lower layer [33].

The TCP/IP network is the most popular network, and the Internet is also a huge TCP/IP network. Although different applications have different application-level protocols, they often share the same underly TCP protocol. For example, some web applications are based on HTTP protocol, and some are based on Web Sock protocol. However, both of these two protocols are running on top of TCP protocol. So is the MySQL communication protocol. Therefore, a TCP proxy server can be a transparent proxy fitting for most of the client-server applications. Figure 4.2 shows the class diagram of the implemented TCP proxy server component for A2SF.

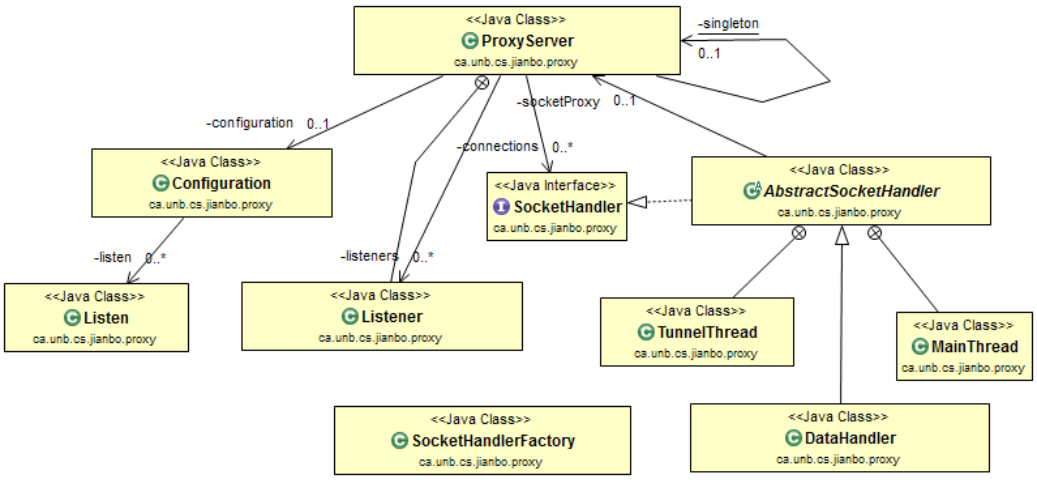

Figure 4.2: Class diagram of TCP proxy server

As shown in Figure 4.2, the TCP proxy server component consists of 11 classes and 1 interface. The class ProxyServer is the main entrance of the server. When the server starts up, it loads the server configuration and initializes the listeners which will listen to the pre-set connection ports in the configuration.

The class Configuration is used to get the configurations, such as the listening IP address, port number and the maximum number of the requests handling threads. When service requests come to the listeners, the class SocketHandlerFactory can create different SocketHandler objects to handle these requests based on different scenarios.

The class AbstractSocketHandler is an abstract class which implements the interface SocketHandler. It implements a set of basic methods, such as establishing a communication tunnel between the client side and the server side.

This class can be inherited and extended by various kinds of handlers.

The class DataHandler inherits class AbstractSocketHandler, and is responsible for handling data messages based on the communication protocol and proxy rules. By implementing different types of handlers, we can easily support variety of proxy servers. For instance, in the service proxy server component and data proxy server component, we implemented the class HttpHandler and the class MysqlHandler to replace the class DataHandler.

4.2.2 Implementation of service proxy server

The service proxy server is the core component of the whole framework. It does not only handle service requests from tenants, but also manages cloud resources based on tenants' statuses.

Besides the TCP proxy core-server with the HttpHandler module, the service proxy server also has the tenant identification module, the tenant status manager module, and the service instance manager module. The tenant identification module provides identification services. And the tenant status manager loops the tenants' list and checks each tenant's status. The service instance manager is able to generate and recycle the customized service instances. Based on tenants' status, the service instance manager manages the service instances on the fly.

4.2.2.1 Implementation of HttpHandler

The HttpHandler module is responsible for processing and delivering HTTP messages between client side and service instance side. Figure 4.3 shows the class diagram of the HttpHandler module.

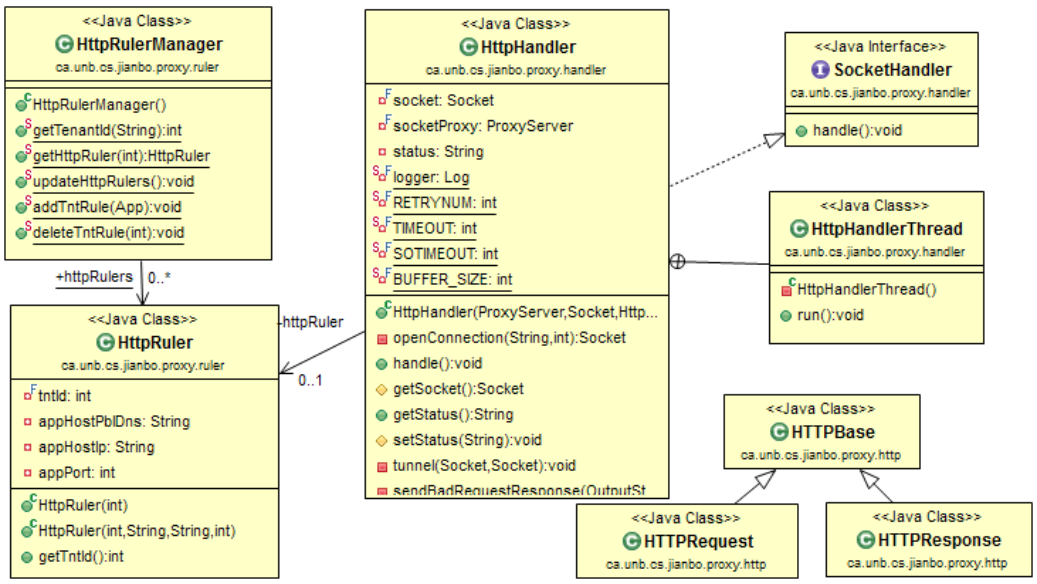

Figure 4.3: Class diagram of the HttpHandler module

As shown in Figure 4.3, the class HttpHandler implements the interface SocketHandler. Via class HttpHandlerThread, the service proxy server is able to create a handling thread for each http request.

The class HttpRulerManager and class HttpRuler are for keeping and managing the proxy rules of tenants. Each tenant has its own rule set, which is kept in the HttpRulerManager and to be updated regularly during runtime. The class SocketHandlerFactory creates an HttpHandler object with HttpRuler

object based on the tenant's information. And in order to parse the Http messages, the class HTTPRequest and HTTPResponse are implemented which are inherited from the class HTTPBase.

4.2.2.2 Implementation of tenant manager

The tenant manager component is responsible for retrieving tenant information, tenant proxy rules, tenant status, and so on. Figure 4.4 is the class diagram of the tenant manager component.

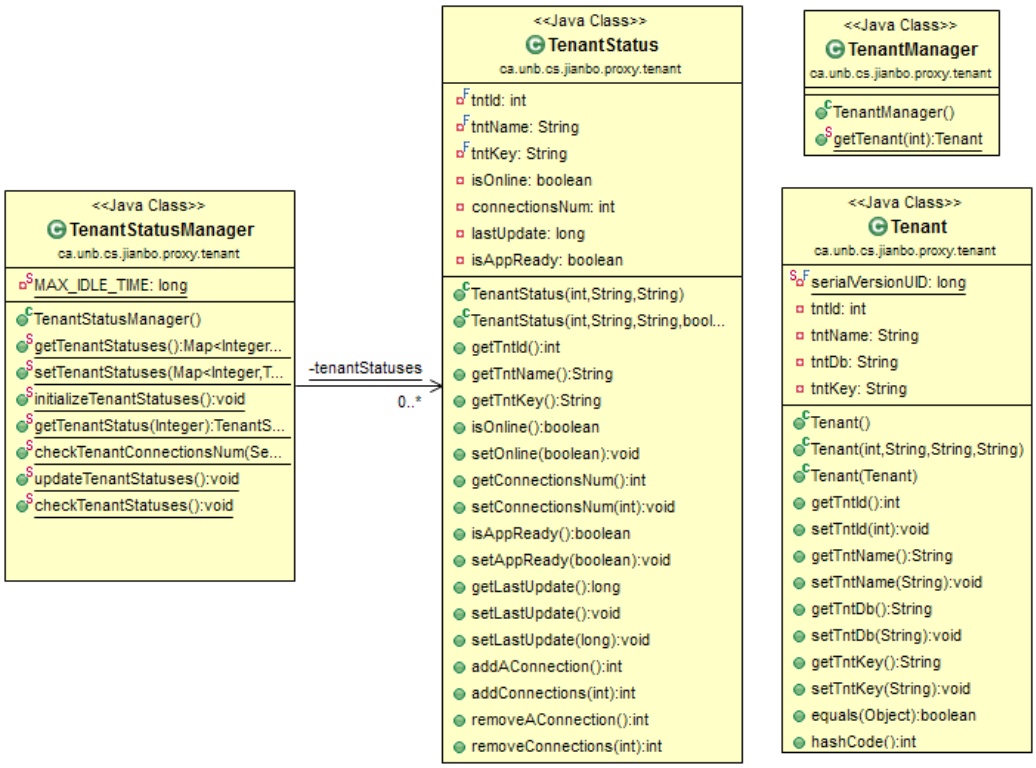

Figure 4.4: Class diagram of the tenant manager module

As shown in Figure 4.4, the tenant manager component includes the classes TenantManager, Tenant, TenantStatusManager, and TenantStatus.

The class TenantManage is to implement the tenant information management and tenant identification. The class TenantStatusManager has a hash map to store of all the tenants statuses. When a new client request arrives, the service proxy server updates the tenant status hash map. The tenant status manager loops the tenants' list and check each tenant status. Based on a tenant's status, the tenant status manager invokes the service instance manager to manage service instances. For example, if a tenant is on-line, but there is no activity in the last 30 minutes, the tenant status manager invokes the method "stopServiceInstance" of the service instance manager to recycle the service instance for this tenant.

The tenant identification is implemented in the tenant manager component. When a tenant subscribes to the service, the tenant administrator needs to set a tenant key as the tenant's token, which will be carried by each service request. When the service proxy server receives the service request, it captures the token first and invokes the tenant identification service to validate it. Once the identification of this request is confirmed, the service proxy server takes the next steps to process and forward the request to the proper service instance. In the implemented prototype system of A2SF, in order to achieve the goal of no modification of original code, the IP address was taken as the tenant's token (key).

4.2.2.3 Implementation of service instance manager

The service instance manager component is responsible for managing service instances on cloud, for example, generating or recycling a service instance atomically. In the A2SF framework, a service instance is a unit of cloud resources to be allocated and released. So the management of service instances also means the management of the allocated cloud resources. Figure 4.5 shows the class diagram of the service instance manager component. It shows the main classes and their relationships in the service instance management component.

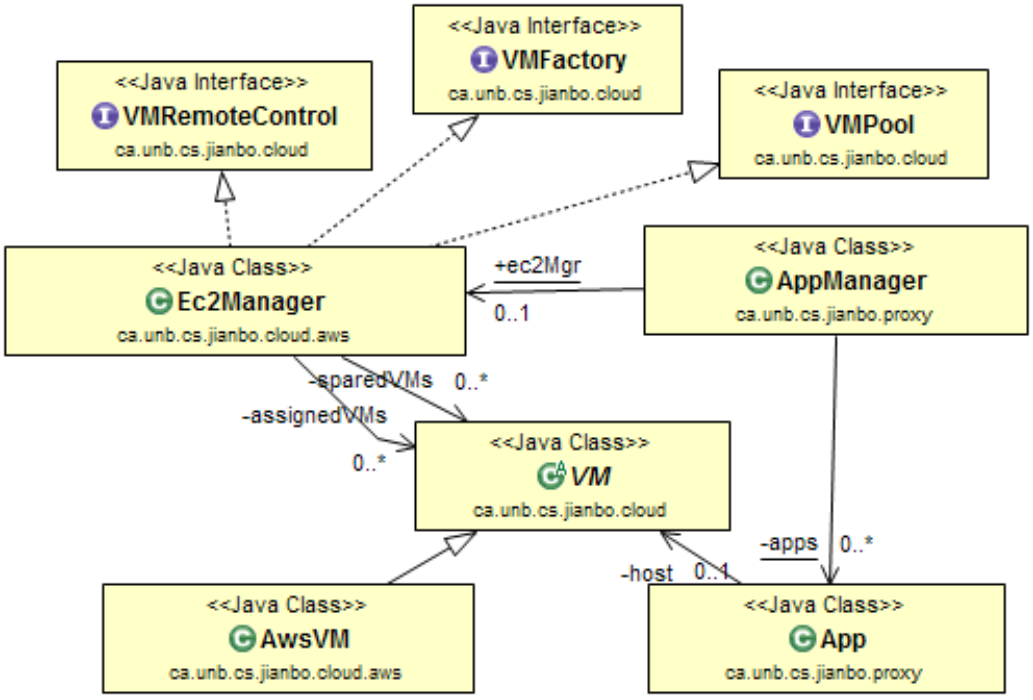

Figure 4.5: Class diagram of the service instance manager module

The class AppManager and class EC2Manager are the core classes in the service instance manager component. As known, a service instance is a customized application instance running on a virtual machine. The class AppManager is responsible for generating, customizing, and recycling service instances, while the class EC2Manager is responsible for providing cloud resources, like virtual machines in the Amazon cloud. The class EC2Manager implements the interfaces of VMFactory, VMPool and VMRemoteControl. These interfaces defined several methods of manipulating remote virtual machines. For example, the interface VMFactory has two methods, getVM() and returnVM(). By invoking these two methods, the AppManager can get a vm instance from the Amazon EC2 cloud and also can return or release the vm instance back.

Though currently the launching time of a new vm instance in the Amazon EC2 has been shortened to around one minute, it is still a little bit long for real time web response. In order to get a shorter time of application instance generation, the A2SF implements a vm pool in class EC2Manager, which always keeps a pre-setting number of spare and ready virtual machines. When a vm is requested, the EC2Manager returns the prepared virtual machine from vm pool immediately, and then replenish the vm pool again.

The interface VMRemoteControl defines several methods to manipulate the remote virtual machines. For instance, one of the methods is "executeBatch" which is used to execute a batch of commands (A2SF scripts) in the remote virtual machine. In this way, as long as a pre-designed initialization script is

provided, the framework can initialize an application instance automatically. The A2SF scripts are responsible for loading and restoring the local private data between the service instance vm and the data storing bucket in the Amazon S3. There are two kinds of A2SF scripts now in the framework: one is the initialization script which is designed to copy the tenant private data like the configuration file from S3 to local before the application instance running; the other is the context saving script which is designed to copy the tenant private data from local to S3 after the application instance being stopped and before the virtual machine instance being terminated.

The class AwsVM inherits from the abstract class VM and implements its virtual machine functions based on AWS SDK, such as vmStart(), vmStop(), getHostPblDns() and so on.

As depicted in Figure 4.5, in this component, only the class EC2Manager and the class AwsVM are related with the AWS SDK. This design pattern makes the framework easy to switch to other cloud platforms and avoid being locked-in with one particular cloud service provider.

4.2.3 Implementation of data proxy server

Similar to the service proxy server, the data proxy server component is also implemented by extending the TCP proxy core-server. The data proxy server has two work modes: local proxy mode and remote proxy mode. Usually, in software development, there are two types of database connection: one is local connection, which means the database and the application are running

on the same host, and in the code, it can use "localhost" or "127.0.0.1" to represent the database location; the other is remote connection, which means the database and the application are deployed on the different hosts, and in the code, it uses the domain name with port number or the IP address with port number to represent the database location.

If the target application uses the local connection, the data proxy server will work in the local proxy mode. This means in the migrated SaaS system, the data proxy server will be deployed on the virtual host with the listening address being set to "127.0.0.1" in the configuration file. While, if the target application uses the remote connection, the data proxy server can be configured to be working in the remote proxy mode. This means that the data proxy server will be deployed on a separate host with listening the host's IP address.

The databases of application instances are stored in a MySQL instance in the Amazon RDS. In the migrated SaaS system, different tenants have different databases, whose names are the original database name such as "sugarcrm" with the tenant's name as suffix. For example, the database name for tenant "UNB" is "sugarcrm_unb" in the migrated SaaS system.

4.2.3.1 Implementation of MysqlHandler

The MysqlHandler component is responsible for processing and delivering the Mysql communication messages between the mysql client side and the mysql server side. Figure 4.6 shows the class diagram of MysqlHandler.

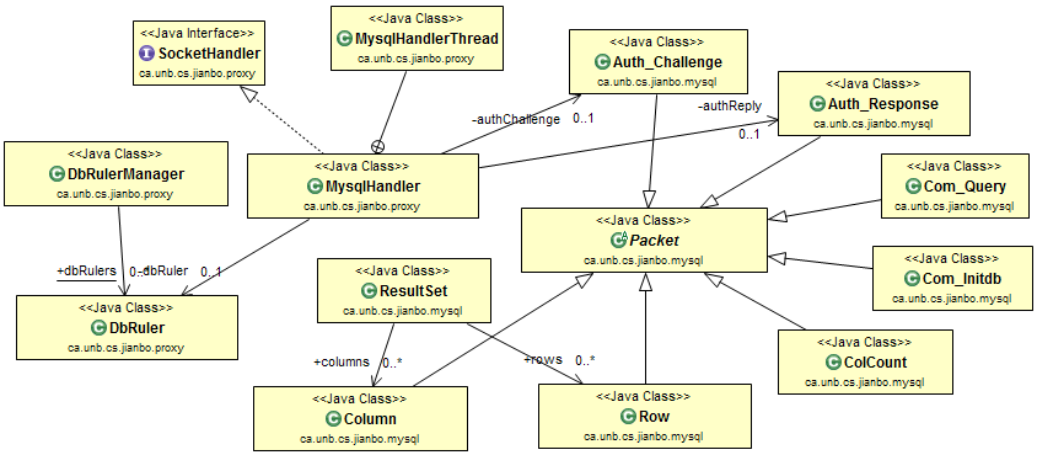

Figure 4.6: Class diagram of MysqlHandler

Similar as the HttpHandler in the service proxy server, the MysqlHandler is designed to process and exchange MySQL message between MySQL clients and MySQL server.

The class MysqlHandler is the core class of this component. For each data request, the class SocketHandlerFactory creates a Mysqlhandler object with a new thread to handle the data request. The class Packet is the base class for parsing MySQL message package. A whole interactive process has five steps between the MySQL client and server: initialization, hand shake, authentication, query and close. In order to handle these various types of messages, a number of classes in this component are implemented, which inherit from the base class Packet, such as Auth_Challenge, Auth_Response, Com_Query, Row, Column and so on.

The class DbRulerManager and class DbRuler are responsible for keeping the

proxy rules of the tenants' databases. The DbRuler helps the Mysqlhandler modify MySQL messages and deliver messages to the correct database server. For instance, a basic modification in the data proxy process is adding suffix to the database name. In the data proxy server, each tenant has its own DbRuler object which is kept by the object DbRulerManager and will be updated regularly.

4.2.4 Implementation of application management centre

The application management centre is a sub-system of the framework. It is responsible for managing the initial settings of the migrated SaaS application and the tenants' information as well. It is implemented as a web application. It has two types of users: service provider and tenant administrator.

The service provider can manage the application configuration using this web application, such as setting the initialization script and the context saving script of the migrated SaaS application. It can manage the tenants' information of the migrated SaaS system as well, such as adding a new tenant, modifying the tenant's information and deleting a tenant.

The tenant administrator can modify its own information and view its service usage report using this web application. Figure 4.7 shows a screenshot of the application management centre's homepage.

Figure 4.7: Screenshot of application management centre homepage

4.2.4.1 Implementation of application management

The application management component is responsible for managing the configuration of the migrated SaaS application in A2SF framework. These settings information is used for generating and recycling service instances automatically in the migrated SaaS system. Figure 4.8 shows a screenshot of application information edit page.

As shown in Figure 4.8, the setting information includes the customized virtual machine image, the database work mode, the initialization script and the context saving script. As discussed in the previous sections, in order to shorten the service instance generation time, a well-prepared virtual machine image is created for launching vm instances, instead of deploying the application from scratch each time. Such prepared virtual machine image includes the executable code of the original application as well as the runtime environment, such as Apache server, PHP, s3cmd tools [29] and so on.

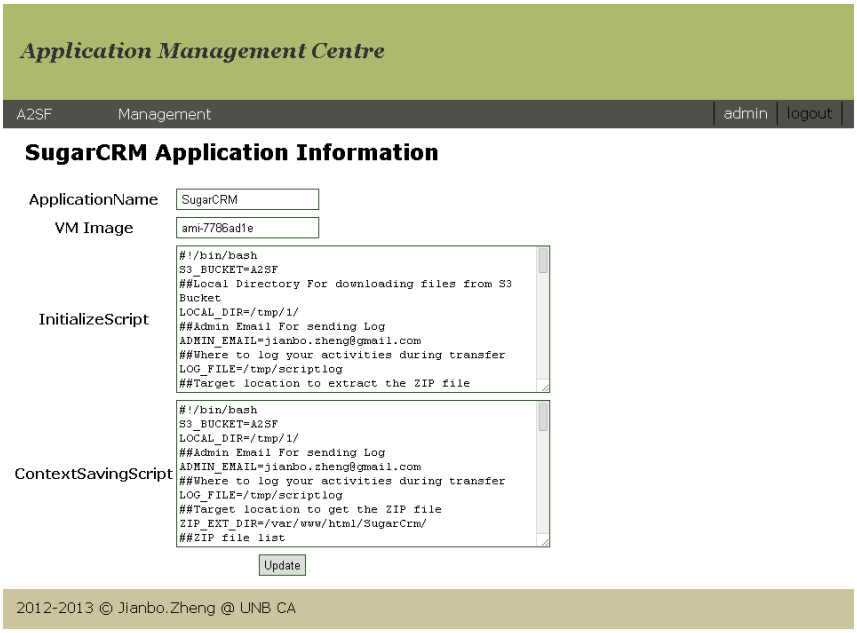

Figure 4.8: Screenshot of application information edit page

When the A2SF needs to generate a service instance for a new tenant, it can launch a new vm instance and load the pre-prepared vm image. Thus after the new vm instance being launched, the target application and the runtime environment are ready for work. The only thing that the A2SF needs is to customize the target application by executing the initialization script for this new tenant. This will shorten the generation time significantly. When the tenant becomes off-line, the A2SF runs the context saving script to save the tenant private data and current status before the service instance being terminated.

4.2.4.2 Implementation of tenant management

The tenant management component includes three features: the tenant information management, the initialization of tenant database, and the usage report of tenant service. Figure 4.9 shows a screenshot of the tenant listing page.

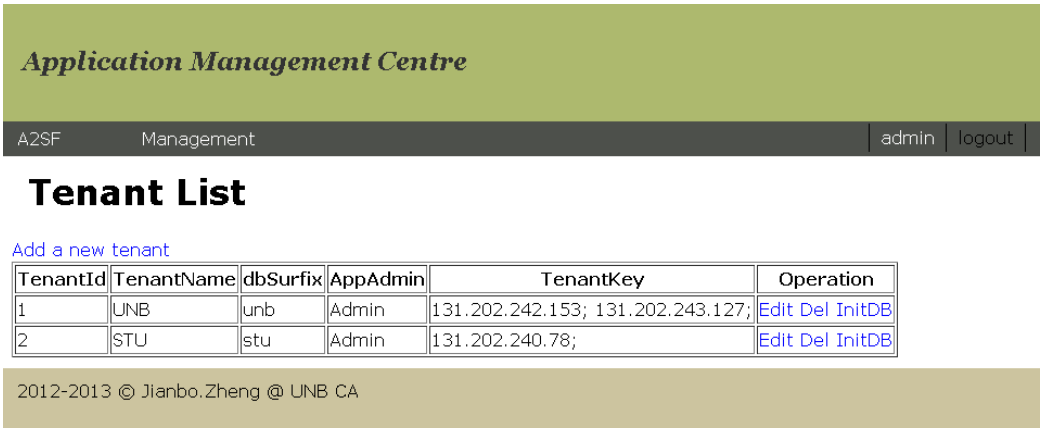

Figure 4.9: Screenshot of tenant list page

As shown in Figure 4.9, tenant information includes tenant id, tenant name, db suffix, application admin and tenant key. As discussed in the previous sections, the db suffix is used to create tenant's database name. The tenant key is used for tenant identification. In the implementation of A2SF, in order to avoid modifying the original code of the target application, the IP address was used as the token for identification. After adding a new tenant, the service provider needs to initialize the database for this tenant. Figure 4.10 shows a screenshot of the tenant information edit page. In this page, the service provider can add the initialization script and the context

saving script. If the tenant has its customized initialization script, the framework will execute this script after the application's initialization script being executed. And if the tenant has its own customized context saving script, the framework also executes this script after the application's context saving script being executed. In this way, the A2SF framework provides a flexible solution for the tenant customization.

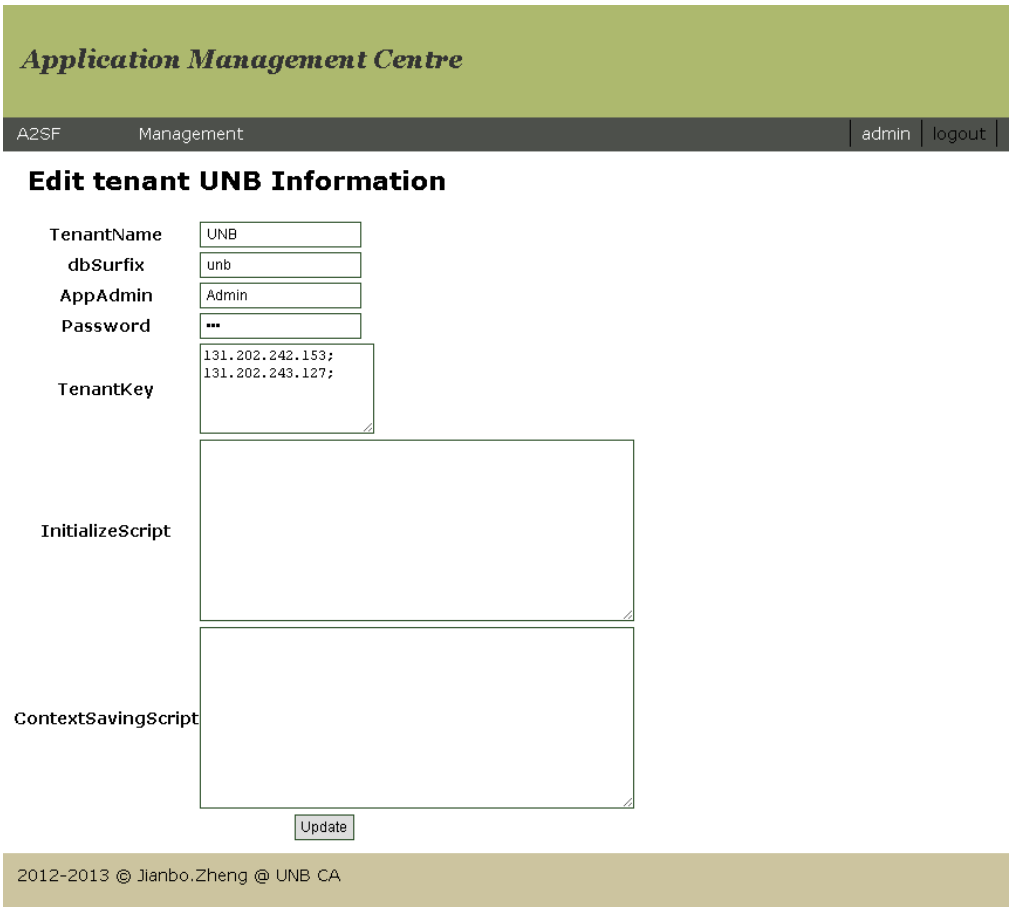

Figure 4.10: Screenshot of tenant information edit page

In the implementation of A2SF, the functionality of the service usage report is relatively simple for now. It only shows the count of the service requests from this tenant and the total time of the service instance running for this tenant. Figure 4.11 shows a screenshot of tenant usage report page.

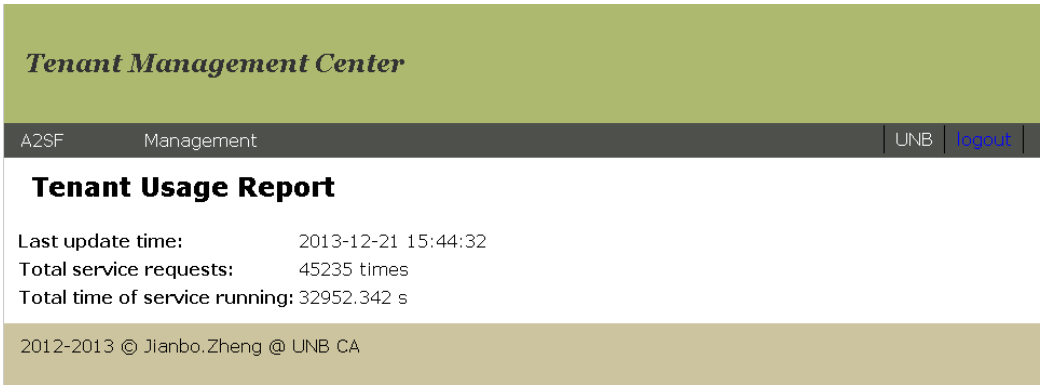

Figure 4.11: Screenshot of tenant usage report page

Chapter 5

Case Study: SugarCRM Cloud Migration

In this chapter, a case study of cloud migration based on the A2SF will be presented. Via this case study, firstly, the process steps of cloud migration based on the A2SF will be summarized and presented; secondly, the functionality of the A2SF prototype will also be verified.

Section 5.1 will briefly introduce the scenario of the case study as well as the target application we chose to be migrated to the cloud. Following section 5.1, the summarized steps of cloud migration based on A2SF as well as the details of each step will be presented in section 5.2. In section 5.3, we will add a new tenant to the migrated system and verify the basic functions of the A2SF components in the migrated SaaS system.

5.1 SugarCRM Client-Server Application

In this case study, a real-world client-server application is considered to be migrated to Amazon AWS cloud based on the A2SF. In order to find a typical and representative real-world client-server application for this case study, we searched for a popular open source CRM application, SugarCRM Community Edition.

The SugarCRM software [28] is among the most popular open source customer relationship management applications available today. SugarCRM provides many functions that allow maximum management for business and client relationships. Such functions include personal home pages, activity management, contacts, accounts, project management and so on. SugarCRM is implemented in the PHP programming language and supports multiple types of databases including MySQL. Figure 5.1 shows a screenshot of the customized demo homepage for UNB as a web based client-server application.

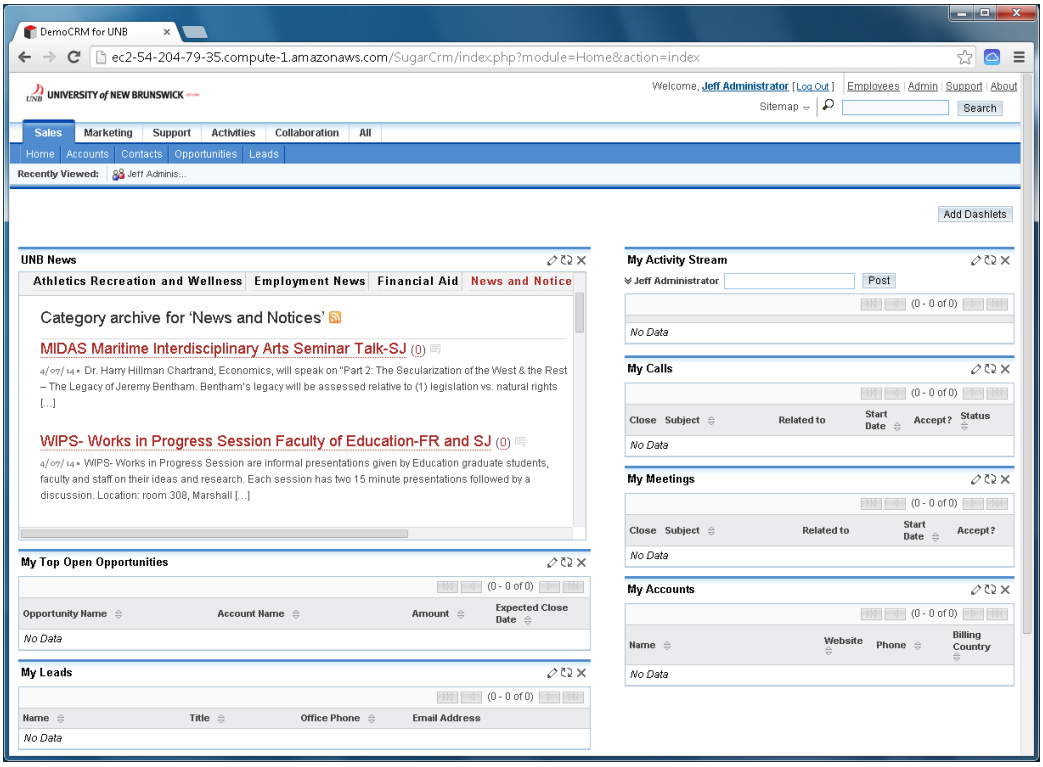

Figure 5.1: Screenshot of the customized demo homepage before being migrated to cloud

5.2 Software Migration and Deployment

This section demonstrates the process of the cloud migration of SugarCRM using A2SF. Generally the process of cloud migration can be divided into three steps: analyze the original application, create a migration package, and deploy the migration package to the cloud.

5.2.1 Analyze original application

Before starting to integrate the A2SF components and the original application, we need to analyze the original application first. In this case study, we do such analysis from three aspects.

5.2.1.1 Check compatibility

First, we need to determine whether or not the target application is supported by the A2SF. Limited by the implementation, so far our implemented prototype only supports the client-server software with MySQL database and can only run on the Linux operation system. In this case study, SugarCRM is a web application and supports MySQL database. Also, it can be deployed on an Apache server in a Linux machine. Thus, SugarCRM can be supported by our implemented prototype of the A2SF.

5.2.1.2 Separate user privacy related components

As discussed in Chapter 3, the components of the target application can be classified into three types: utility components, local private components (usually they are files), and remote private components (usually they are databases). After confirming the compatibility, we need to identify the user privacy related components of the target application and separate them from utility components. The A2SF provides different migration solutions for the different types of components. For example, utility components are deployed on the virtual machine image (in the Amazon cloud, it is also called as AMI) once for all the tenants, and private components are protected for individual tenants under the access control layers of the A2SF framework. Furthermore, the local private components (files) are saved in Amazon S3 and the remote private components (database) are stored in Amazon RDS. In this case study, after installing SugarCRM on an experimental environment, we classified the components of SugarCRM as shown in Table 5.1.

Table 5.1: The classification of SugarCRM components

Type	**Components**
Local private components	Folders[1]: custom, upload, cache, data, modules; Files[1]: .htaccess, sugarcrm.log, config.php, config_override.php
Remote private components	sugarcrm database
Utility components	The rest of components of SugarCRM

[1] All these folders and files are under the root of SugarCRM. In this case, the root of SugarCRM is "/var/www/html/SugarCrm".

As discussed in Chapter 3, because of SugarCRM have the configuration file "config.php", its remote database can be treated as local file data. However, in order to have a full verification of the A2SF components, in this case study we treat the SugarCRM database as the remote private components.

5.2.1.3 Determine cloud services

Limited by the implementation, our prototype only supports Amazon cloud platform, and uses the AWS services like EC2, S3, and RDS. For each service, the Amazon cloud provides various and flexible selections.

Classified by the functionality, there are two kinds of virtual machines that A2SF can use. One is a constant running virtual machine as the portal server, which supports the service proxy server and the application management center. The other is a dynamic generated virtual machine running as a service instance, which supports the data proxy server and the target application.

From the architecture point of view, with the growth of tenants, the portal server may become a bottleneck of the migrated SaaS system. However, in the cloud, the vm instance for the portal server can be easily shifted to a higher performance vm. This is one of the advantages of cloud computing. The SaaS provider can choose a proper service type after traded-off the system between demand and cost. In this case study, we chose the basic type of vm for both the portal server and the service instance server. Table 5.2 and Table 5.3 describe the detail specifications of service types that we chose for the case study. More information can be found in [8].

Table 5.2: Characteristics of Amazon EC2 and RDS used in the case study

	Amazon EC2	**Amazon RDS**
Platform	Amazon Linux Server 64-bit	MySQL 5.6.13 64-bit
CPU(type)	1 EC2 Compute Unit[1]	1 EC2 Compute Unit
Memory	613 MB	630 MB
Instance Storage	8GB EBS storage	20 GB
Number of Instances	Dynamic	1

[1] one virtual core with one EC2 compute unit.

Table 5.3: Characteristics of Amazon S3 used in the case study

Service Name	**Free Usage Limitation**
Amazon Simple Storage Service	5 GB of Amazon S3 standard storage, 20,000 Get Requests, 2,000 Put Requests, and 15GB of data transfer out each month for one year.

5.2.2 Create migration package

Based on the previous analysis, we are ready to integrate the A2SF components and the SugarCRM software to create a migration package.

Firstly, we need to have an Amazon AWS account before starting the migration and to create an access credential (Access Key ID and Secret Access Key) which can be used to access and control the AWS services through the AWS API.

Secondly, we update the configuration and the A2SF scripts files according to our Amazon Security Credentials and the list of private components.

Thirdly, a customized AMI (Amazon Machine Image) should be created for generating SugarCRM service instances, in which the data proxy server,

A2SF scripts, the target application (SugarCRM), and the running environments, like Apache server, PHP 5.3, JDK 1.6, and s3cmd tools [29], are already deployed. Table 5.4 shows the contents of the created migration package for SugarCRM to SaaS on Amazon cloud.

Table 5.4: Contents of SugarCRM migration package

Component	**Description**
Service proxy server	Java jar file to run on the portal server VM as the service entrance.
Management centre	Java war file to be deployed to Tomcat server running on the portal server VM.
Data proxy server	Java jar file to be pre-deployed in the AMI, and run as the local MySQL proxy server VM.
Script template	Bash script file to be instantiated by tenant subscriptions.
Customized AMI	EC2 VM machine image file with pre-deployed data proxy server image.
Database	Pre-defined MySQL database for A2SF framework.
SugarCRM Application	Original client-server application.
SugarCRM DB Creation Script	SugarCRMs database creation script file to be executed when a new tenants subscription is approved.

The Amazon machine image (AMI) is a vm template that contains a customized configuration for launching a customized EC2 vm instance. One simple way to create a customized AMI is to customize a vm instance first and then, based on this customized vm instance, create an AMI for our own use.

The AMI ID will be updated later by the A2SF management centre. Thus

when an off-line tenant becomes online, a new EC2 vm instance will be created first based on the customized AMI, then the framework will execute the initialization scripts on the vm instance, and assign this service instance to the tenant.

5.2.3 Deploy the migration package on Amazon cloud

After analyzing and creating the migration package for the target application SugarCRM, the next step is to deploy the integrated migration package on the Amazon cloud. Figure 5.2 shows the deployment diagram of the migrated SaaS SugarCRM system in the case study.

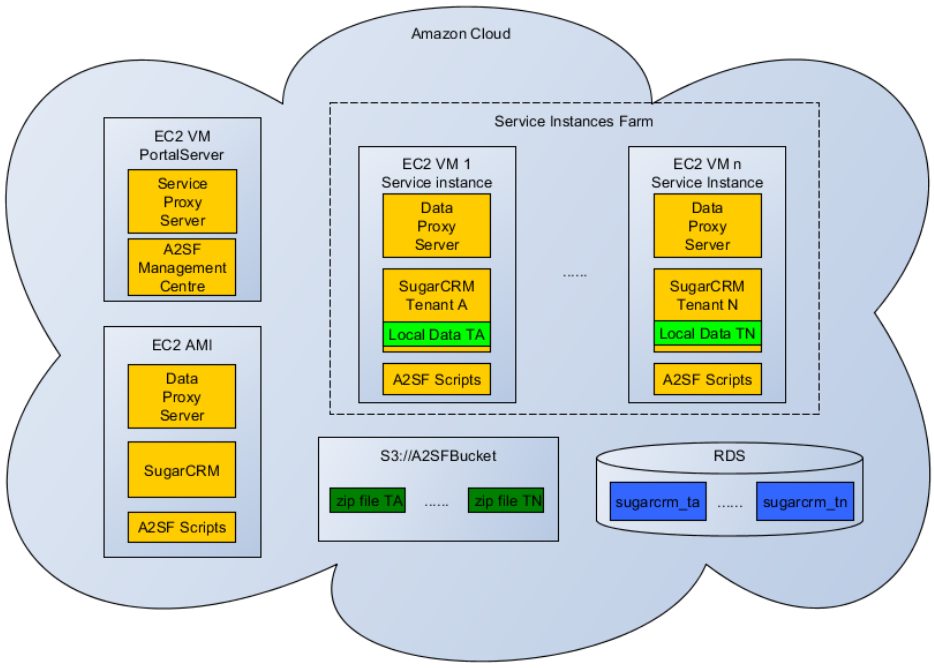

Figure 5.2: The deployment diagram of the migrated SaaS system

As shown in Figure 5.2, the components of the migrated SaaS system are deployed on several vm instances in the Amazon cloud.

Firstly, we need to create a new virtual machine from Amazon EC2 as the portal server. The portal server is the entrance of the migrated SaaS system, as well as the host of the service proxy server and the A2SF management centre.

As discussed in previous sections, there are multiple vm types in EC2 for us to choose as the portal server. Since in our case study, there will not be too much tenants in the migrated SaaS system, we chose a t1.micro type vm instance as the portal server. When the number of tenants quickly grows, the SaaS provider can easily alternate the portal server from the lower performance vm to a higher performance vm. Moreover, if the business keeps growing, and the high performance vm is not sufficient as well, we can take the load balancer solution which is also provided as a service in the Amazon EC2. That is one of advantages and benefits that SaaS providers can easily get from cloud computing.

Secondly, a database named "a2sf" should be created in the MySQL database instance of Amazon RDS. This database works for the A2SF components.

Thirdly, after running the portal server, we need to access the application management centre and update the application information, so that the framework can generate service instances correctly. As shown in Figure 4.8, currently, the application information for SugarCRM includes the application name, the virtual machine image id (AMI ID), and the A2SF scripts.

After completing all the steps in this section, we have successfully integrated and deployed the A2SF components and the original software of the target application SugarCRM to the Amazon cloud.

5.3 Test the Migrated System

This section illustrates testing the basic functions of the migrated system.
After starting up all the components in the Amazon cloud, the clients or end users of different tenants can access the SaaS from their web browsers on different client computers, where we use client computers IP addresses to identify which tenants the clients belong to.
In this case study, the public domain name of the portal server is EC2-50-16-48-150.compute-1.amazonaws.com. So the end users can access the migrated system by URL: "http://EC2-50-16-48-150.compute-1.amazonaws.com". Because we have not added any tenant in the system yet, the first time we access the URL in the browser, it shows "Tenant Not Found" page. Figure 5.3 shows a screenshot of the tenant not found page.
The result in Figure 5.3 means that the service proxy server cannot identify the source of the request.

1. **Add tenants to SugarCRM SaaS**

   In this step, we add a new tenant and access the migrated system again. We log into the A2SF management center by the URL: "http://EC2-50-16-48-150.compute-1.amazonaws.com:8080" and adds two tenants.

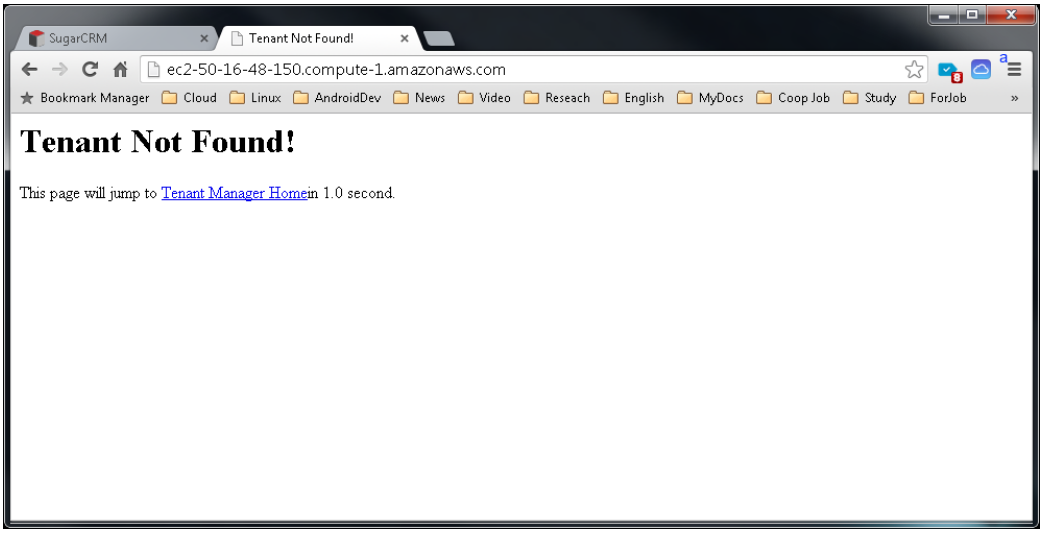

Figure 5.3: Screenshot of tenant not found page

The tenants' information is shown in Table 5.5:

Table 5.5: List of the adding tenants' information

TenantName	**TenantKey**	**dbSuffix**	**AppAdmin**	**Password**
UNB	131.202.242.153	unb	Admin	unb
STU	131.202.240.78	stu	Admin	stu

After adding the new tenants to the migrated system, we open client browsers on the machines, with IP addresses 131.202.242.153, 131.202.240.78, and 131.202.240.167, and access the portal server in the client browsers. Finally, we get the login pages, and use the user names and passwords of the tenants to login as shown in Figure 5.4 and Figure 5.5.

Figure 5.6 shows a screenshot of the proxy server logs on the portal server and application instance server. In Figure 5.6, the left side is

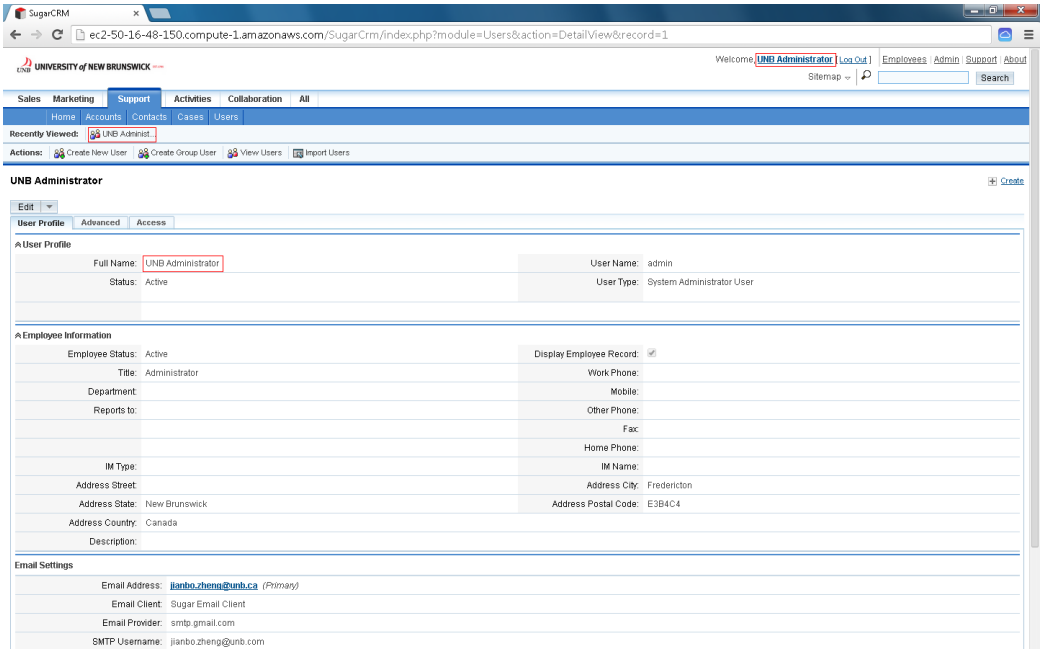

Figure 5.4: Screenshot of tenant "UNB" administrator's profile page

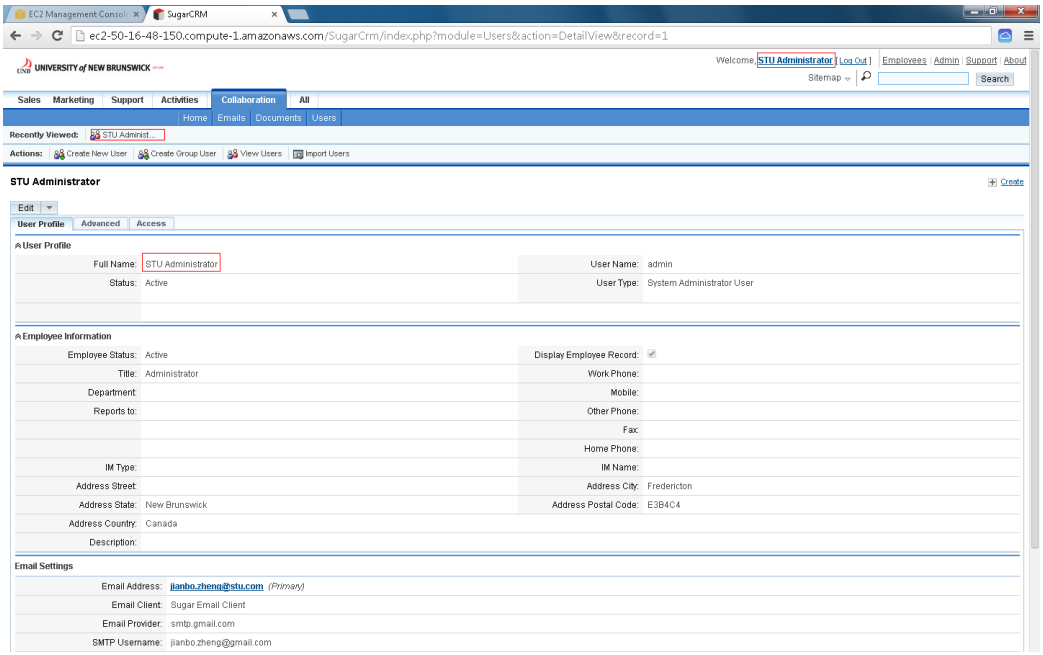

Figure 5.5: Screenshot of tenant "STU" administrator's profile page

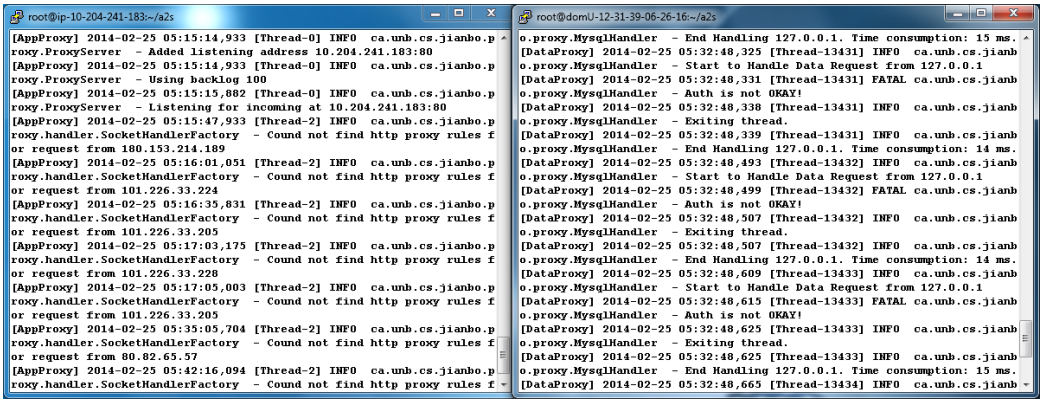

Figure 5.6: Screenshot of proxy server logs

the logs of the service proxy server and the right side is the logs of the data proxy server.

From the EC2 management console, there are four running VM instances as shown. Figure 5.7 shows a screenshot of the vm instances list in the EC2 management console.

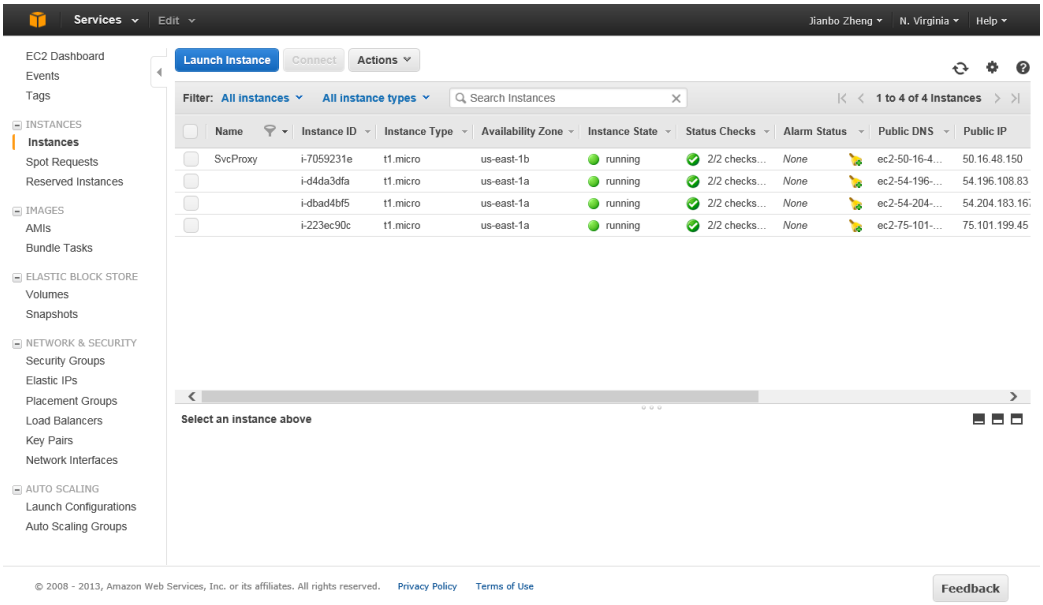

Figure 5.7: Screenshot of running instances in EC2 management console

The vm instance named “SvcProxy” is the portal server; the other three vm instances are the service instance servers. Two of them are for two tenants and one is launched and waiting in the vm pool as a spare vm instance.

2. **Delete the tenant "STU"**

   We next delete the tenant "STU" in the A2SF management center and access the migrated system from the machine with IP address 131.202.240.78. The result shows the "Tenant Not Found" page. Figure 5.8 is the list of vm instances in the EC2 management console. The vm instance for tenant "STU" has been terminated.

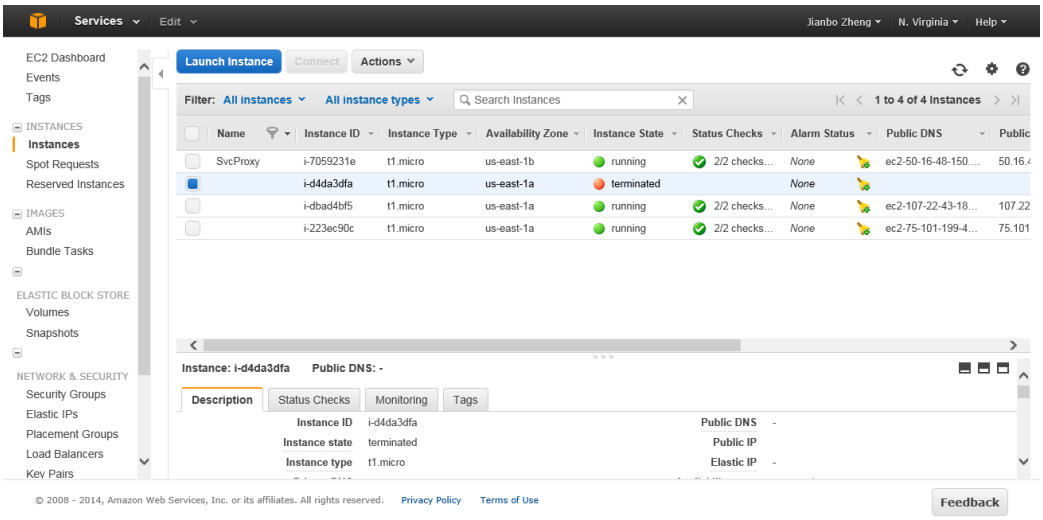

Figure 5.8: Screenshot of running instances after deleting tenant "STU"

3. **Keep the tenant "UNB" offline for more than 30 minutes**

   We next close the client browser on the machine with IP address 131.202.242.153 and wait for more than 30 minutes. The vm instance for tenant "UNB" is terminated. Figure 5.9 shows the list of vm instances in the EC2 management console after UNB was off-line for more than 30 minutes.

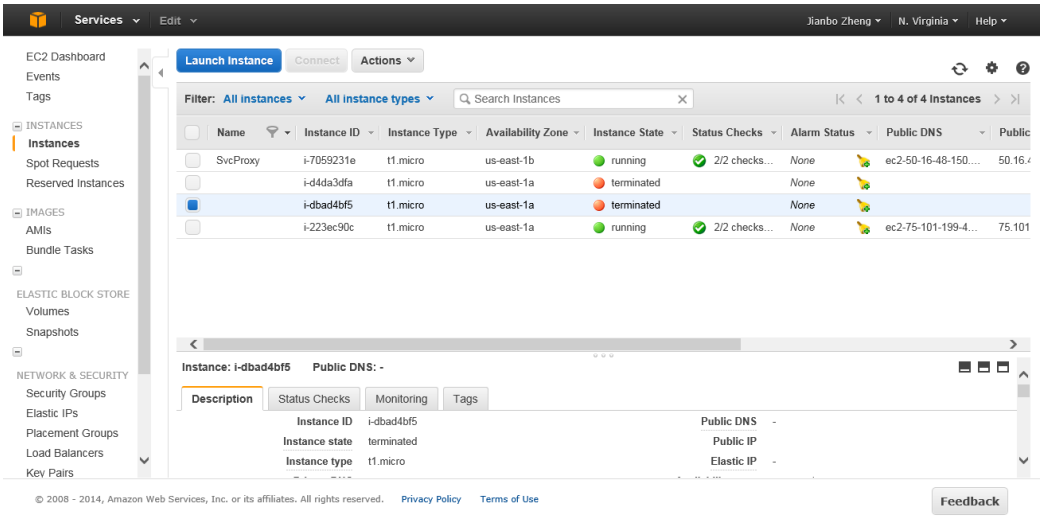

Figure 5.9: Screenshot of running instances after tenant "UNB" offline

All the test stages being presented above indicate that the A2SF components are working and the migrated SugarCRM SaaS is running well.

This chapter introduced and demonstrated the steps of how a real-world client-server application is migrated to the cloud based on the A2SF framework. It also demonstrated how to start the migrated SaaS system with dynamic tenants. In the next chapter, we will conduct more experiments and evaluate the quality of the A2SF framework.

Chapter 6

Experiments and Evaluation

In the previous chapter, a case study and the process of cloud migration were demonstrated. This chapter reports further experiments on A2SF to verify and evaluate the framework.

Section 6.1 describes several experiments to measure the time latency of the migrated SugarCRM system and analyze the performance and scalability of the A2SF based on the experiment results. Section 6.2 demonstrates more cloud migration practices on other kinds of client-server applications based on the A2SF. In the last section, we evaluate the A2SF from five quality aspects: usability, performance, scalability, security, and flexibility.

6.1 Experiments on Time Latency

In the case study, the open source web application SugarCRM was migrated to the Amazon cloud environment running as an SaaS application. As described in previous chapters, to make an existing client-server system support multi-tenancy, an effective approach is adding the multi-tenant awareness layers. However, more layers added also means more time the process will take. In this section, we describe several experiments to measure the time latency of the migrated SugarCRM system and analyze the performance of the A2SF based on the results.

6.1.1 Objective of the experiments

For client-server applications, the response time is a very important quality requirement, especially for web-based applications. The response time can be defined as the amount of time that the application takes from the client sending a service request to the server to the client computer receiving and displaying the response to the end user.

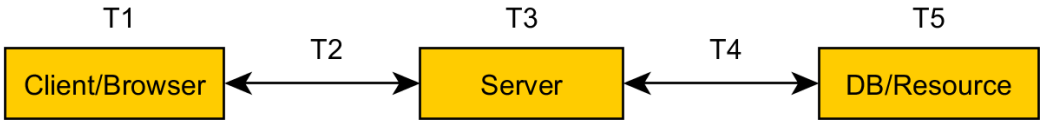

Figure 6.1: The Composition of a response time before migrated to the cloud

The response time can be broken down into several segments as illustrated in Figure 6.1.

After SugarCRM is migrated to the cloud based on the A2SF, two multi-tenant awareness layers are added in the process. Figure 6.2 shows the composition of response time for the migrated SugarCRM SaaS by the A2SF.

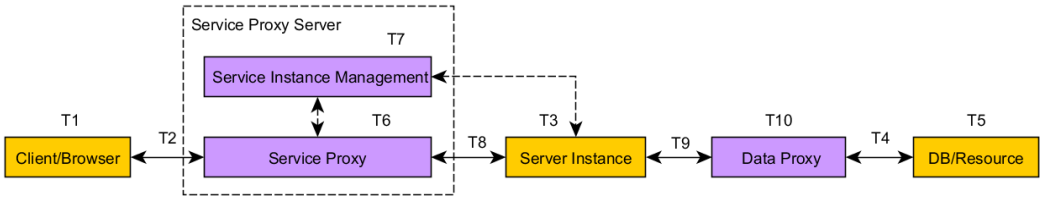

Figure 6.2: The Composition of a response time after migrated to the cloud by the A2SF

The following are the explanation of the time slips:

T1 The time of the client sending a service request and receiving the response.

T2 The time of the service request/response travelling between the client and the server.

T3 The time of the server handling the service request and sending the response back.

T4 The time of the data request/response travelling between the server and the database.

T5 The time of the database processing the data request.

T6 The time of the service proxy server processing and dispatching the service request/response.

T7 The time of generating a service instance.

T8 The time of service request/response travelling between the service proxy server and service instance.

T9 The time of data request/response travelling between the service instance and the data proxy server.

T10 The time of the data proxy handling the data request/response and dispatching them.

Compared with Figure 6.1 and 6.2, we can see that the response time of the migrated SugarCRM SaaS theoretically should be slower than that of the original system before being migrated. Moreover, the service instances for multiple tenants are dynamic, which means only when a tenant sends a service request, the service instance of the tenant would be generated and started up. So the first response time to an off-line tenant is longer than the response time for online tenant. Other important time segments are the processing time of the service proxy and the processing time of the data proxy. The service proxy server is a key component of the A2SF. All clients of tenants need to access the migrated SaaS application through the service proxy server. So the efficiency of the service proxy server has a big effect on the performance and scalability of the migrated SaaS application. As in

the migrated SaaS application, the data server is deployed on each service instance, theoretically its performance should always be relatively stable.
In this experiment, the following four time segments were measured: the first response time, the average time of the normal response, the processing time of the service proxy, and the processing time of the data proxy.

6.1.2 Experiments and results

With the migrated SugarCRM SaaS, we created a testing program which is a local webpage designed to send HTTP requests to the specified remote server regularly at specific times, as shown in Figure 6.3.

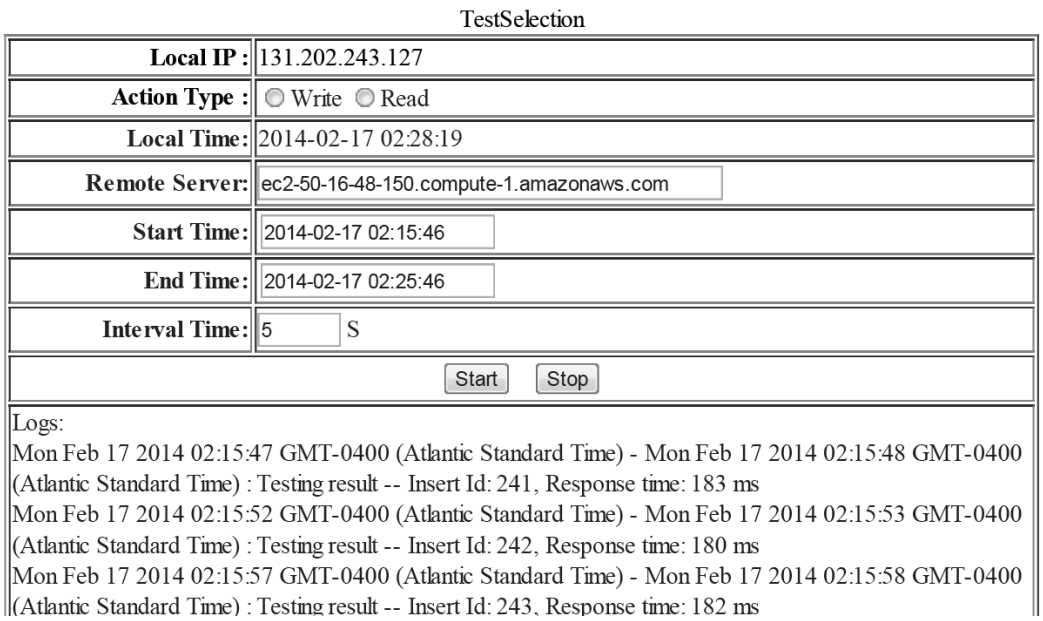

Figure 6.3: The testing program

The experiments were conducted in a computer lab at University of New

Brunswick. Ten computers with different IP addresses were chosen to simulate ten tenants of the migrated SugarCRM SaaS. Table 6.1 shows detailed tenant-IP assignments.

Table 6.1: Tenant key (IP) list

Tenant Name	**Tenant Key**
Tenant1	131.202.243.127
Tenant2	131.202.243.128
Tenant3	131.202.243.129
Tenant4	131.202.243.130
Tenant5	131.202.243.131
Tenant6	131.202.243.136
Tenant7	131.202.243.138
Tenant8	131.202.243.139
Tenant9	131.202.243.140
Tenant10	131.202.243.141

6.1.2.1 Experiment on the original client-server system

We launched a micro-type virtual machine in the Amazon EC2, which is the same capability of the vm for service instance of migrated SugarCRM SaaS. Then, we deployed the original SugarCRM software on this vm server and deployed its database on the database instance in the Amazon RDS. After deployed the SugarCRM application on Amazon cloud, we ran the testing script on this original SugarCRM and the measured response times without the A2SF. The average response time of the original system was 100.75ms.

6.1.2.2 Experiment on the migrated system

Before starting the experiments, we created 10 tenants with different assigned IPs in the Application management center, and then carried the following experiments.

1. **Experiment with single tenant**

   We took tenant1 as the single tenant in this experiment and ran the test script on the computer with IP 131.202.243.127. We measured the first response time, the average normal response time, the average service proxy processing time, and the average data proxy processing time. They are listed in Table 6.2.

   Table 6.2: The result of the experiment with a single tenant

Measure Item	**Value (millisecond)**
The first response time	3428
The average stable response time	169.05
The average service proxy time	62.39
The average data proxy time	27.04

2. **Experiment with 10 tenants**

   We started up the ten computers listed in Table 6.1, and ran the testing script on each computer. the time period for this experiment is 10 minutes. During the testing period, we let each tenant computer to send a request to the server every 5 seconds to simulate multiple client connections and requests from each tenant. Table 6.3 shows the tenants' settings for the experiment.

Table 6.3: The list of tenants settings in the experiment

Tenant Name	**Start Time**	**Stop Time**	**Interval Time**
Tenant1	2014-02-17 03:30:57	2014-02-17 03:40:57	5 s
Tenant2	2014-02-17 03:30:30	2014-02-17 03:40:30	5 s
Tenant3	2014-02-17 03:30:58	2014-02-17 03:40:58	5 s
Tenant4	2014-02-17 03:30:04	2014-02-17 03:40:04	5 s
Tenant5	2014-02-17 03:30:09	2014-02-17 03:40:09	5 s
Tenant6	2014-02-17 03:30:22	2014-02-17 03:40:22	5 s
Tenant7	2014-02-17 03:30:14	2014-02-17 03:40:14	5 s
Tenant8	2014-02-17 03:30:11	2014-02-17 03:40:11	5 s
Tenant9	2014-02-17 03:30:04	2014-02-17 03:40:04	5 s
Tenant10	2014-02-17 03:30:59	2014-02-17 03:40:59	5 s

The measured results of the experiment with 10 tenants are shown in Table 6.4.

Table 6.4: The results of experiment with 10 tenants

TenantName	**FRT**[1]	**ASRT**[2]	**ASPT**[3]	**ADPT**[4]
Tenant1	20438	172.38	70.32	25.92
Tenant2	52014	180.42	62.83	24.68
Tenant3	24863	177.50	70.10	30.30
Tenant4	3509	161.30	62.03	23.43
Tenant5	60271	172.47	69.19	29.59
Tenant6	52924	166.03	61.02	21.94
Tenant7	59248	174.30	69.89	30.59
Tenant8	54631	175.94	61.08	25.15
Tenant9	3417	163.46	61.81	22.90
Tenant10	65741	173.40	70.37	31.05
Average	39705.6	171.72	65.86	26.56

[1] FRT stands for the first response time.
[2] ASRT stands for the average stable response time.
[3] ASPT stands for the average service proxy time.
[4] ASRT stands for the average data proxy time.
[5] The unit of time value in this table is millisecond.

6.1.3 Result analysis

The results of the experiments are following:

The experiments show that the average response time of the original system before migration is around 100 ms, and the average response time of the migrated SaaS system is around 170 ms. So the time latency of using A2SF is around 70 ms.

The first response time of the migrated SaaS system in the best case was 3417 ms, and in the worst case was 65741 ms in the experiments. These two values also can be taken as the times of generating a service instance on a pre-prepared vm instance and generating a service instance from launching a new vm instance.

The average time of service instance generation is related to the size of the vm pool and the size of target application. Adding a vm pool would improve the efficiency of service instance generation. The experiments show that when the size of vm pool in A2SF was set to 2 the average time of the first response became 39705.6 ms which is significantly lower than the first response time of the worst case.

The experiments show that the average service proxy processing time is around 65ms, and the average data proxy processing time is around 27 ms. Under the same computing capability, the service proxy processing time and the data proxy processing time are stable. In case of supporting more tenants, the A2SF can choose to increase the number of CPU allocate to the service proxy server or add a load balancer layer with multiple service proxy

servers to process service requests.

6.2 Other Experiments

In order to verify that the A2SF has the capability of supporting variable types of client-server applications, we also migrated several other applications to the cloud based on the A2SF.

In the case study, based on the A2SF, SugarCRM as a PHP based web application was migrated to the Amazon EC2. Using the migration process, we also migrated a Java web application which is implemented in J2EE SSH (Spring, Struts, and Hibernate) framework and a conventional client-server application. The experiments show that these two migrated systems work well. The conventional client-server application is implemented in Java and also uses MySQL as its database.

These two experiments show that the A2SF has good flexibility and extensibility to support variety of client-server applications. Even for those applications which have different protocols, the application vendors can easily extend the implementation of the A2SF to meet their requirements.

6.3 A2SF Evaluation

In this section, we evaluate the A2SF framework from five quality aspects: usability, performance, scalability, security, and flexibility.

6.3.1 Usability

In Chapter 2, the challenges faced by the cloud migration of conventional applications were summarized as tenant identification, data security, and dynamic scalability. In Chapter 5, we verified the functionality of the A2SF by the case study of the SugarCRM cloud migration. The case study shows that the A2SF has overcome these challenges. Furthermore, Chapter 5 also summarized a general solution based on the A2SF as well as the operatable and detailed steps of the cloud migration of the client-server application. The experiments in Section 6.2 indicted that the A2SF not only supports the cloud migration of various web applications but also supports the cloud migration of a non-web application.

In order to have a friendly user interface, the A2SF also provides a web-based management center. By this management center, the service provider and the tenant administrators can easily configure the tenants' information and the application services. The A2SF integrates the cloud migration solution and multi-tenancy framework together, which makes the cloud migration solution more easily operatable.

6.3.2 Performance

In section 6.1, several experiments on the time latency has been tested. From the results of these experiments, we conclude that the time latency is stable and stays in a tolerantable range. The results also indicate that the design of the vm pool is useful and can effectively shorten the response time. Furthermore, when the scalability of the migrated system is scaled up quickly, in order to keep the performance, the A2SF could be able to apply two solutions: one is that the service provider chooses a vm instance that has the better performance for the portal server; and the other is that the service provider creates a load balancer tier for the A2SF. These two solutions are easily to be implemented in the Amazon cloud.

6.3.3 Scalability

From the architecture point view, the service instances in the A2SF are independent and can be dynamically generated and recycled. Because of running on the cloud, the capability of computing and the scalability of service instances are no longer problems. The service provider can add or delete a tenant via the A2SF management center, and the migrated system manages these service instances based on the statuses of the tenants.

Another solution to achieve more efficient scalability is that one vm instance hosts multiple application instances. This solution should be researched in the future.

6.3.4 Security

Security is always the key point of cloud migration. The efforts of data security in the A2SF are in two aspects: the access control and data isolation. In the aspect of the access control, first of all, the A2SF limits and identifies the tenants by their IP addresses. Although this restriction limits the convenience (such as anytime and anywhere access) for the end user, it gives the first secure protection. The security and convenience tradeoff is a common problem. Furthermore, the IP address issue can be solved through other technology such as VPN. Secondly, by adding the multi-tenant awareness layer, the A2SF acquired the effective authentication and access control of the system module and tenant data. In the migrated system, the tenants can only access their own resources.

The data isolation in the A2SF can be divided into two aspects: the database and the runtime private data. In the database aspect, the A2SF stores the tenant database separately. The databases of different tenants can be configured by the tenant and stored in different locations. In the runtime private data aspect, the A2SF stores the runtime private data in different vm instances. When the tenant goes off-line, its private data will be first copied to the tenant's bucket in the Amazon S3, and then deleted from the terminating vm instance. In this way, the tenant's privacy gets maximum protection.

6.3.5 Flexibility

The flexibility of the A2SF is mainly reflected in three aspects. Firstly, in the framework design, as we discussed in chapter 3, the multi-tenant awareness layers shared a common TCP proxy core-server which can be easily inherited and extended to support different application requirements. Secondly, the data proxy server provides two work modes: the local work mode and the remote work mode. For the different types of database connections, the A2SF provides different work modes accordingly. Last but not least, the A2SF allows tenants to add their own customized scripts in the A2SF management center, such as the initializing script and context-saving script.

Chapter 7

Conclusions

This chapter concludes the thesis. Section 7.1 summarizes the thesis work. The contributions of the thesis are listed in Section 7.2, and the last section contains some future work.

7.1 Summary

This thesis studied the current research status on cloud migration and multi-tenancy. The basic challenges were presented, which were faced by the cloud migration of conventional client-server applications.

In this thesis, a framework named A2SF was proposed, which aims to make the cloud migration in an easy way and without modifying the original system. A prototype of the A2SF was implemented on the Amazon cloud using several Amazon cloud services, like EC2, S3, and RDS in the A2SF. In the

case study, we developed a general cloud migration process based on the A2SF and presented the steps of the cloud migration process. Furthermore, a series of experiments were designed and conducted to verify and measure the functionality and performance of the implemented A2SF. Finally, the A2SF was evaluated from five quality aspects: usability, performance, scalability, security, and flexibility.

7.2 Contributions

First, we summarized the key challenges faced by the cloud migration of the conventional applications and discussed the solutions of these challenges.

Second, we proposed a general multi-tenancy supported framework for the cloud migration of client-server software. The framework provides an easy way for software vendors to migrate their applications to the cloud without revising the original code.

Third, we implemented a prototype of the A2SF on the Amazon cloud. The prototype gained the scalability and flexibility on the capacity of computing and storage by applying multiple Amazon cloud services.

Finally, we developed a general cloud migration process based on the A2SF and presented the migration steps via a case study. We also verified and evaluated the A2SF by a series of experiments.

7.3 Future Work

Further work to continue the research on the A2SF can be considered in three aspects: usability, security and efficiency.

From the usability aspect, though the implemented prototype of the A2SF is working well, there is still room for improvement. Currently, our framework only supports the Amazon cloud and MySQL database. In future, the framework needs to support more types of cloud engines and databases and provide a more friendly configuration interface. More experiments and studies are also needed to be conducted and analyzed to improve the performance of the A2SF.

From the security aspect, the solutions of tenants' privacy isolation need further research and improve. The current prototype of the A2SF has isolated the private data of the tenants and also provides the access control in the inner framework. However, if the invaders know the public IP addresses of service instances and access them directly, they can still have the chance to crack the service instances and access tenants' privacy data. One solution for this is to establish a virtual LAN for each tenant, and forbid direct accesses from the outside and from other tenants as well.

From the efficiency aspect, the current framework assigns each tenant service instance a virtual machine. In order to improve cloud resource utilization, in future, deploying application instance for multiple tenants on same virtual machine instance should be studied.

Bibliography

[1] Amazon, *The total cost of (non) ownership of web applications in the cloud*, Tech. report, Technical report, Amazon Web Services, 2012.

[2] Rajkumar Buyya, James Broberg, and Andrzej M Goscinski, *Cloud computing: Principles and paradigms*, vol. 87, John Wiley & Sons, 2010.

[3] Hong Cai, Ning Wang, and Ming Jun Zhou, *A transparent approach of enabling saas multi-tenancy in the cloud*, Services (services-1), 2010 6th world congress on, IEEE, 2010, pp. 40–47.

[4] Hong Cai, Ke Zhang, Ming Jun Zhou, Wei Gong, Jun Jie Cai, and Xin-Sheng Mao, *An end-to-end methodology and toolkit for fine granularity saas-ization*, Cloud Computing, 2009. CLOUD'09. IEEE International Conference on, IEEE, 2009, pp. 101–108.

[5] Muhammad Aufeef Chauhan and Muhammad Ali Babar, *Towards process support for migrating applications to cloud computing*, International Conference on Cloud and Service Computing (CSC 2012), 2012.

[6] Frederick Chong and Gianpaolo Carraro, *Building distributed applications architecture strategies for catching the long tail*, MSDN architecture center (2006).

[7] Hala N . Dajani, *Client-server component architecture for scientific computing*, Master's thesis, Rice University, April 2003.

[8] Amazon EC2, *Amazon elastic compute cloud (amazon ec2)*, http://aws.amazon.com/ec2, October 2013.

[9] Cloud ERP, *business accounting software, crm, ecommercen - netsuite*, http://www.netsuite.com/portal/home.shtml, October 2013.

[10] Gartner, *Worldwide software-as-a-service revenue to reach 14.5 billion in 2012*, http://www.gartner.com/newsroom/id/1963815, April 2013.

[11] Chunye Gong, Jie Liu, Qiang Zhang, Haitao Chen, and Zhenghu Gong, *The characteristics of cloud computing*, Parallel Processing Workshops (ICPPW), 2010 39th International Conference on, IEEE, 2010, pp. 275–279.

[12] Google, *Google app engine*, http://code.google.com/appengine/, October 2013.

[13] Chang Jie Guo, Wei Sun, Ying Huang, Zhi Hu Wang, and Bo Gao, *A framework for native multi-tenancy application development and management*, E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007.

CEC/EEE 2007. The 9th IEEE International Conference on, IEEE, 2007, pp. 551–558.

[14] Google Inc., *Google office productivity suite*, https://docs.google.com/, October 2013.

[15] Li Jiang, Jin Cao, Peifeng Li, and Qiaoming Zhu, *A mixed multi-tenancy data model and its migration approach for the saas application*, Services Computing Conference (APSCC), 2012 IEEE Asia-Pacific, IEEE, 2012, pp. 295–300.

[16] Ali Khajeh-Hosseini, David Greenwood, and Ian Sommerville, *Cloud migration: A case study of migrating an enterprise it system to iaas*, Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on, IEEE, 2010, pp. 450–457.

[17] Daichao Lu, *A cloud broker design for cloud-based it planning*, Master's thesis, The University of Texas at Dallas, December 2012.

[18] Peter Mell and Tim Grance, *The nist definition of cloud computing*, National Institute of Standards and Technology **53** (2009), no. 6, 50.

[19] Xin Meng, Jingwei Shi, Xiaowei Liu, Huifeng Liu, and Lian Wang, *Legacy application migration to cloud*, Cloud Computing (CLOUD), 2011 IEEE International Conference on, IEEE, 2011, pp. 750–751.

[20] Microsoft, *Windows azure*, http://www.microsoft.com/windowsazure/, October 2013.

[21] Amazon RDS, *Amazon relational database service (amazon rds)*, http://aws.amazon.com/rds, October 2013.

[22] Fred Rowe, *Migrating legacy software applications to cloud computing environments: A software architect's approach*, Master's thesis, East Carolina University, 2011.

[23] Amazon S3, *Amazon simple storage service (amazon s3)*, http://aws.amazon.com/s3, October 2013.

[24] SalesForce, *Crm and cloud computing - salesforce.com*, http://www.salesforce.com/, October 2013.

[25] Taraneh Baradaran Seyed, *Client/server technology*, Master's thesis, Southern Connecticut State University, March 2006.

[26] G. Shroff, *Enterprise cloud computing-technology, architecture, applications*, Cambridge University Press, 2010.

[27] Jie Song, Feng Han, Zhenxing Yan, Guoqi Liu, and Zhiliang Zhu, *A saasify tool for converting traditional web-based applications to saas application*, Cloud Computing (CLOUD), 2011 IEEE International Conference on, IEEE, 2011, pp. 396–403.

[28] SugarCRM, *Crm software and online customer relationship management*, http://www.sugarcrm.com/community/, October 2013.

[29] S3 tools, *Amazon s3 tools: s3cmd*, http://s3tools.org/s3cmd, October 2013.

[30] Jinesh Varia, *Migrating your existing applications to the aws cloud*, A Phase-driven Approach to Cloud Migration (2010).

[31] Xun Xu, *From cloud computing to cloud manufacturing*, Robotics and Computer Integrated Manufacturing **28** (2012), no. 1, 75–86.

[32] Jianfeng Yang and Zhibin Chen, *Cloud computing research and security issues*, Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on, IEEE, 2010, pp. 1–3.

[33] Hubert Zimmermann, *Osi reference model–the iso model of architecture for open systems interconnection*, Communications, IEEE Transactions on **28** (1980), no. 4, 425–432.

Vita

Candidate's full name:
Jianbo Zheng
University attended (with dates and degrees obtained):
Shandong University of Science and Technology, China
Southeast University, China
Poster:
Jianbo Zheng, "Poster: Cloud Computing Migration Tool," Proc. of the 10th UNB Computer Science Research Exposition, April 11, 2013, Fredericton, NB, Canada