# Multi-Tenancy and Sub-Tenancy Architecture in Software-As-A-Service (Saas)

by

Peide Zhong

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved November 2015 by the
Graduate Supervisory Committee:

Hasan Davulcu, Chair
Hessam Sarjoughian
Dijiang Huang
Wei-Tek Tsai

ARIZONA STATE UNIVERSITY

May 2017

ABSTRACT

Multi-tenancy architecture (MTA) is often used in Software-as-a-Service (SaaS) and the central idea is that multiple tenant applications can be developed using components stored in the SaaS infrastructure. Recently, MTA has been extended where a tenant application can have its own sub-tenants as the tenant application acts like a SaaS infrastructure. In other words, MTA is extended to STA (Sub-Tenancy Architecture ). In STA, each tenant application not only need to develop its own functionalities, but also need to prepare an infrastructure to allow its sub-tenants to develop customized applications. This dissertation formulates eight models for STA, and proposes a Variant Point based customization model to help tenants and sub-tenants customize tenant and sub-tenant applications. In addition, this dissertation introduces Crowd- sourcing to become the core of STA component development life cycle. To discover fit tenant developers or components to help building and composing new components, dynamic and static ranking models are proposed. Further, rank computation architecture is presented to deal with the case when the number of tenants and components becomes huge. At last, an experiment is performed to prove rank models and the rank computation architecture work as design.

# DEDICATION

*Firstly, I dedicate my dissertation work to my family and many friends. A special feeling of gratitude to my loving parents, Jiacan and Zhuowen and elder sister, Xiaoling who were always supporting me and encouraging me with their best wishes. Secondly, I dedicate this dissertation to my wife, Zhixin Dong who was always there cheering me up and stood by me through the good and bad times, my son, Arthur Zhong who was always bringing me lots of funs and interesting stories, and my little baby Annie who gives me hope and courage.*

*Finally, I dedicate this dissertation to my many friends, Song Xiang, Feng Guo, Wei Lu, Ke Bai, Chong Sun, Jun Xie, Frank Yang, Xiaoyong Chai, Sha Liu, Tianyi Xing who have supported me throughout the process. I will always appreciate all they have done and extraordinary suggests they made.*

ACKNOWLEDGMENTS

*I would never have been able to finish my dissertation without the guidance of my committee members, help from friends, and support from my family, my wife, my son and another little baby on the way. I would like to express my deepest gratitude to my advisor, Professor Hasan Davulcu, for his excellent guidance, caring, patience, and providing me with excellent research topics. I also would like to thank him to patiently corrected my writing and financially supported my research. I would also like to thank Professor Sarjoughian, Professor Huang and Professor Tsai for guiding my research for the past several years and helping me to develop my background in software engineer and are willing to participate in my committee. I would like to thank Jay Elston, who as a good friend, was always willing to help and give his best suggestions. It would have been a lonely research journey without him. Many thanks to Qian Huang, Xin Sun, Guanqiu Qi, Wu Li, Xingyu Zhou, Le Xu, Zhibin Cao and JingJing Xu for helping me and giving me suggests. My research would not have been possible without their helps.*

TABLE OF CONTENTS

Page

LIST OF TABLES ..... ix

LIST OF FIGURES ..... x

CHAPTER

1 INTRODUCTION AND RELATED WORK ..... 1

1.1 Introduction ..... 1

1.2 Related Work ..... 2

1.2.1 MTA in SaaS ..... 2

1.2.2 Crowdsourcing ..... 3

1.2.3 Scalability in SaaS ..... 4

1.2.4 Security in SaaS ..... 5

1.2.5 Variation Point ..... 5

1.2.6 Customization in SaaS ..... 6

2 STA MODELS ..... 7

2.1 SSTA Models ..... 9

2.2 Two Level STA Models (TSTA): ..... 10

2.3 Multi-level STA (MSTA) ..... 15

2.4 STA Properties ..... 18

3 STA SECURITY CONSIDERATION AND ACCESS PERMISSION MODELS ..... 20

3.1 STA Security Consideration ..... 20

3.1.1 SSTA ..... 20

3.1.2 TSTA ..... 20

3.1.3 MSTA ..... 21

3.2 Permission Access Models ..... 21

CHAPTER Page

3.2.1 Permission Access Model for Applications . . . . . 21
3.2.2 Permission Access Model for Data . . . . . 22
4 STA CUSTOMIZATION MODELS . . . . . 24
4.1 STA Customization Techniques . . . . . 24
4.1.1 For SSTA . . . . . 26
4.1.2 For TSTA . . . . . 26
4.1.3 For MSTA . . . . . 28
4.2 STA Customization Deduction . . . . . 29
4.3 Variant Point Model . . . . . 31
4.3.1 VP Classification . . . . . 31
4.3.2 VP Relationships . . . . . 34
4.3.3 VP Properties . . . . . 38
4.3.4 VP Options and Rules Deduce Algorithm . . . . . 38
4.4 STA Customizations with VP models . . . . . 40
4.4.1 Server-Customers STA Customization . . . . . 41
4.4.2 Software-Data STA . . . . . 41
4.4.3 Master-Slaves STA . . . . . 43
4.4.4 Slave-Masters STA . . . . . 45
4.4.5 Partner-Partner STA . . . . . 46
5 STA IMPLEMENTATION STRATEGIES . . . . . 48
5.1 By Traditional Approaches . . . . . 48
5.2 By Template with VPs . . . . . 49
5.2.1 Service Management and Composition . . . . . 49
5.2.2 SaaS Application Templates . . . . . 51

CHAPTER Page

5.2.3 Extensions to Allow Users Designate Specific Services ..... 53
5.2.4 Code Generation Support ..... 55
5.2.5 Testing Workflows Of Service Composition ..... 55
5.2.6 Oracle Generation of Composite Services ..... 55
5.2.7 Unit Testing ..... 56
5.2.8 Integration Testing ..... 56
5.2.9 Continuous Testing ..... 57
5.2.10 Metadata-Driven Test Input Generation ..... 57
5.2.11 Execute Testing Processing by Service-Level MapReduce Way 58
6 TENANT-CENTRIC STA ..... 60
6.1 Life Cycles of Tenant-Centric Application Development ..... 60
6.2 Component and Tenant Rank ..... 63
6.2.1 Static Ranking Model ..... 63
6.2.2 Dynamic Ranking Model ..... 65
6.2.3 Rank Computation Architecture ..... 72
6.3 Feature Implementation Selection Model ..... 74
6.4 Rapid Application Building Process ..... 77
6.5 Experiment ..... 78
6.6 Conclusion ..... 82
7 DEPENDENCY-GUIDED SERVICE COMPOSITION ..... 83
7.1 Introduction ..... 83
7.2 Related Work ..... 85
7.3 Ontology Relationships ..... 88
7.3.1 Relationships in Ontology ..... 89

CHAPTER Page

7.3.2 Relationships Representation ..... 89
7.4 Dependency Analysis ..... 90
7.4.1 Axioms ..... 91
7.4.2 Property Definitions ..... 92
7.4.3 Formal Notation Definition ..... 93
7.4.4 Operations ..... 94
7.4.5 Theorems ..... 95
7.4.6 Algorithms ..... 96
7.5 Composition With Dependency Support ..... 97
7.5.1 Composition Process ..... 101
7.5.2 Key Techniques ..... 103
7.6 Case Study - Shipping Domain Tracking System ..... 106
7.6.1 Existing Items ..... 107
7.6.2 Specifications ..... 108
7.6.3 Notification Way Change Workflow ..... 108
7.7 Conclusion ..... 109
8 SERVICE REPLICATION WITH MAPREDUCE IN CLOUDS ..... 110
8.1 Introduction ..... 110
8.2 Related Work ..... 112
8.3 Cloud Architecture ..... 113
8.4 Service Replication Strategies ..... 116
8.4.1 Service-Level MapReduce ..... 116
8.4.2 Number of Replications Needed ..... 118
8.4.3 Passive Service Replication Strategy ..... 119

CHAPTER Page

8.4.4 Active Service Replication Strategy . . . . . 121
8.5 Application Illustration . . . . . 124
8.5.1 Data Sorting . . . . . 124
8.5.2 Keyword Search in Large Documents . . . . . 124
8.6 Case Study . . . . . 125
8.7 Conclusion . . . . . 127
9 STA EXPERIMENT AND CASE STUDY . . . . . 128
9.1 Experiment - STA Online Shopping System . . . . . 128
9.1.1 STA Online Shopping System Requirements . . . . . 128
9.1.2 STA Online Shopping System Experiment . . . . . 129
9.2 STA Online Shopping System Case Study . . . . . 131
9.2.1 VP Experiment . . . . . 143
REFERENCES . . . . . 146

LIST OF TABLES


Table Page

2.1 TSTA Summary ..... 12

2.2 TSTA Model Comparison ..... 16

2.3 Examples of STA Properties ..... 19

4.1 Server-Customers STA Customization ..... 42

4.2 Software-Data STA Customization ..... 44

4.3 Master-Slave STA Customization ..... 45

4.4 Slave-Master STA Customization ..... 46

4.5 Partner-Partner STA Customization ..... 47

6.1 Connected Graph with Weights ..... 78

6.2 Subgraph with Weights ..... 80

7.1 Corresponding Relationship ..... 93

7.2 Existing Items in Different Applications ..... 107

9.1 Tenant Information ..... 132

9.2 SubTenantSharingPermissions ..... 136

9.3 Tenant - Subtenants ..... 136

9.4 Tenant- Subtenants ..... 137

LIST OF FIGURES


Figure Page

2.1 Single Level STA Example . . . . . 7

2.2 SingleOrg-STA Example . . . . . 8

2.3 Tenant and Sub-tenant Relationship Example . . . . . 9

2.4 MultiOrg-STA Example . . . . . 10

2.5 Server-Customers Example . . . . . 11

2.6 Software-Data Example . . . . . 11

2.7 Master-Slaves Example . . . . . 14

2.8 Slave-Masters Example . . . . . 15

2.9 Partner-Partners Example . . . . . 17

2.10 Server-Customers MSTA Example . . . . . 17

2.11 STA Property Relationships . . . . . 19

4.1 E-Science Ontology Example . . . . . 25

4.2 UI VP Example . . . . . 33

4.3 Service VP Example . . . . . 33

4.4 Workflow VP Example . . . . . 33

4.5 Data VP Example . . . . . 34

4.6 Restrict Relationship . . . . . 34

4.7 Inherit Relationship . . . . . 35

4.8 Extend Relationship . . . . . 36

4.9 Compose Relationship . . . . . 37

4.10 Implement Relationship . . . . . 37

4.11 VP Properties . . . . . 39

4.12 VP Relationship DAG Example . . . . . 39

5.1 STA Architecture Overview . . . . . 49

Table Page

5.2 Domain Ontology Example . . . . . 50
5.3 Workflow Example . . . . . 51
5.4 Service Template Example Mapping . . . . . 52
5.5 Template Example in PSML-S . . . . . 52
5.6 Service Binding by Programming Example . . . . . 53
5.7 Ground Service Binding Example . . . . . 54
5.8 A GroundProfile Example . . . . . 54
5.9 Generated Source Code of Workflow . . . . . 55
5.10 Oracle Generation Service Level MapReduce process . . . . . 59
6.1 Application Development Life Cycle . . . . . 60
6.2 Community of Interests Example . . . . . 62
6.3 Static Ranking Example . . . . . 63
6.4 Component Rank Example . . . . . 66
6.5 Tenant Rank Example . . . . . 67
6.6 Rank Computation Architecture . . . . . 72
6.7 Feature Implementation Selection Model . . . . . 75
6.8 Result of Static Rank . . . . . 79
6.9 Staic Rank With Dynamic Rank Update . . . . . 79
6.10 Result of Dynamic Rank . . . . . 81
6.11 Final Score vs Final Static Score . . . . . 81
7.1 Shipping Domain Service Ontology . . . . . 88
7.2 Property Illustration . . . . . 90
7.3 User Centric SOA Composition Architecture . . . . . 97
7.4 A Composition Operation Sequence . . . . . 101

Table Page

7.5 Services with Dependencies . . . . . 104
7.6 Initial Dependency Likelihood Estimates . . . . . 106
7.7 Dependency Information . . . . . 107
7.8 Notification Way Change Workflow . . . . . 108
7.9 Notification Way Change Workflow with Its Dependency . . . . . 108
7.10 Notification Way Change Workflow and Its Mapping . . . . . 109
8.1 High Level Cloud Architecture . . . . . 114
8.2 Service-Level MapReduce Process . . . . . 116
8.3 Details of passive SRS process . . . . . 119
8.4 Details of active SRS process . . . . . 121
8.5 Instantiated Framework of SLMR . . . . . 126
9.1 STA Online Shopping Data Model . . . . . 130
9.2 SingleOrg-SSTA Flow Example . . . . . 131
9.3 STA Architecture Overview . . . . . 132
9.4 STA Provider's Default Templates . . . . . 137
9.5 SingleOrg-SSTA Tenant's Templates . . . . . 139
9.6 SC-STSTA Tenant's Templates . . . . . 139
9.7 SD-STSTA Tenant's Templates . . . . . 139
9.8 STA Customization Data Models . . . . . 143

# Chapter 1

# INTRODUCTION AND RELATED WORK

## 1.1 Introduction

Cloud platforms often have three main components: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS). SaaS is the software deployed over the internet [126], where users subscribe services from SaaS providers and pay by a way of "pay-as-you-go". In SaaS, software is maintained and updated on a cloud, and presented to the end users as services on demand. Multi-Tenancy Architecture (MTA) of saaS allows tenant developers to develop applications using the same code based stored in the SaaS infrastructure. MTA is often designed by integration with databases. MTA supports tenant application customization by composition of existing or new software components stored in the SaaS or supplied by tenant developers.

However, current MTA has the following limitations:

1. While a SaaS infrastructure support tenant applications using services and data stored in the infrastructure, a tenant application does not allow its users to use its own services or data to develop new applications.

2. It is difficult for a tenant application to share service or data with other tenant applications. Often, a SaaS platform provides security mechanisms to isolate tenant applications so that tenants cannot access data that belong to other tenants. Even though tenant code and data are stored in the same database, the SaaS security mechanism isolates a tenant from other tenants.

3. Most SaaS systems do not support tenants to customize their applications already customized by other tenants.

To address those issues, Tsai in [104] introduced a STA (Sub-Tenancy Architecture) to allow tenants to offer services for sub-tenant developers to customize their applications. As SaaS component building often needs different technologies such as frontend UI and database, tenants or sub-tenants often not good at all those technologies. Therefore, it is still difficult for them to build SaaS components from the scratch. Hence, this paper introduce Crowdsourcing to make use of public wisdom and assign tasks to specific experts who are good at. To help find fit tenants, sub-tenants and components to complete component building, ranking models are introduced.

## 1.2 Related Work

### *1.2.1 MTA in SaaS*

MTA may be implemented via the following ways:

1. Integration with Databases: Weissman and Bobrowski proposed a database-based and metadata-driven architecture to implement MTA in [120]. In [120], heavy indexing is used to improve the performance, and a runtime application generator is used to dynamically build applications in response to specific user requests. As all tenants share the same database, flexible schema design is used. Aulbach [11] experiments five techniques for implementing flexible schemas for SaaS.

2. Middleware Approach: In this approach, an application request is sent to a middleware that passes the request to databases behind the middleware. As all databases are behind the middleware and all application requests to databases are managed and directed by the middleware, applications can be transformed

into a MTA SaaS rapidly with minimum changes to the original applications. Cai [21] described a transparent approach of making existing Web applications to support MTA and run in a public cloud.

3. Service-oriented SaaS: This is an approach to implement MTA by SOA. SaaS domain knowledge is separated from SaaS infrastructure to facilitate different domains. EasySaaS [95] proposed a development framework to simplify SaaS development by harnessing both SOA and SaaS domain ontology. Azeez [12] proposed an architecture for achieving service-oriented MTA that enables users to run their services and other SOA artifacts in a MTA service-oriented framework as well as provides an environment to build MTA applications. As this MTA is based on SOA, it can harness both middle and SOA technology.

4. PaaS-based approach: The SaaS developers use an existing PaaS such as GAE [41], Amazon EC2 [3], or Microsoft Azure [61] to develop SaaS applications. In this approach, developers use the MTA features provided by a PaaS to develop SaaS applications, and most of SaaS features such as code generation, and database access are implemented by the PaaS. Tsai [97] proposed a model-driven approach on a PaaS to develop SaaS.

5. OO approach: Workday [129] proposed an object-oriented approach for tenant application development and configuration. In addition, [129] also conducts a study on MTA models, specifically it addresses the architecture of MTA and its impact on customization, scalability, and security.

### *1.2.2 Crowdsourcing*

The purpose of Crowdsourcing is to make use of public wisdom and let crowd with domain knowledge complete specific tasks. Howe first defined the term "crowd-

sourcing" in a companion blog post [43]. [60] defines Crowdsourcing as the practice of obtaining needed services, ideas, or content by soliciting contributions from a large group of people, and especially from an online community, rather than from traditional employees or suppliers. Kittur in [52] investigated the utility of a micro-task market for collecting user measurements, and discussed design considerations for developing remote micro user evaluation tasks. Peng in [73] provided an overview of current technologies for crowdsourcing.

### *1.2.3 Scalability in SaaS*

In MTA SaaS, components may be shared by tenants. In addition, each tenant may have a large number of users, and the number of concurrent accesses from users can be huge. Therefore, scalability in SaaS is important. In general, there are two solutions to scale a software system: scale-up and scale-out. In [100], scale-up is defined as running the application on a machine with a better configuration, including more computing resource, more memory, higher disk bandwidth and larger disk space; and scale-out is defined as running the application distributed on multiple machines with similar configurations. Tsai [94] identified scalability factors and discuss their impacts on the scalability of SaaS applications. In addition, evaluating scalability of SaaS application is also an important topic. Tsai [96] described unique features and challenges in testing SaaS applications, and introduce scalability metrics that can be used to test the scalability of SaaS applications. Service replication is another way to scale SaaS applications. Therefore, Tsai [110] proposed a way to replicate services for making use of MapReduce. Resource allocation becomes an issue as all tenants share same SaaS applications. Therefore, Espadas [36] proposed a resource allocation model to deploy SaaS applications over cloud computing platforms to create a cost-effective scalable environment. In addition, each tenant may have different SLA requirements.

Take SLA into account, Wu [130] proposed resource allocation algorithms for SaaS providers to minimize infrastructure cost.

### *1.2.4 Security in SaaS*

Security is an important topic in SaaS as all tenants share the same SaaS infrastructure. Compare to tradition software engineer, it introduce new challenges such as authorization and authentication. Rashmi [74] analyzed the status of cloud computing security. Data protection is also important as all tenants may share same database schema in some SaaS implementations. Chou [26] introduced security policies for SaaS data protection.

As STA requires supports for multi-level tenants, all methods including MTA implementations, customization and scalability discussed need to be extended.

### *1.2.5 Variation Point*

Variation points are locations that variation occurs, and variants are the alternatives that can be selected. Software product families introduce variability management to deal with these difference by handling variability. Kang [50] describes a method for discovering commonality among different software systems. Coplien [27] describes how to perform domain engineering by identifying the commonalities and variabilities within a family of products. Webber [118] describes a systematic method for providing components that can be extended through variation points, which allows the reuser or application engineer to extend components at pre-specified variation points to create more flexible set of components. Mietzner [63] presented a variability descriptor and describe they can be transformed into a WS-BPEL process model to guide customizations. In addition, Mietzner [65] explained how variability modeling techniques can support SaaS providers in managing the variability of SaaS

applications and proposed using explicit variability models to derive customization for individual SaaS tenants.

### *1.2.6 Customization in SaaS*

Customization is an important SaaS feature as tenants may have different business logic and interface yet they share the same code base. Chong [25] proposed a SaaS maturity model that classifies SaaS into four levels including ad-hoc/custom, customizable or configurable, multi-tenant efficient, and scalable. Tsai [101] introduced ontology into SaaS to help customize applications. In [101], a SaaS tenant application has components from four layers: GUI, workflow, service and data. For each layer, there is an ontology to help tenants customize SaaS applications. Variability modeling and management techniques have been widely employed in software product-line engineering and SaaS providers can potentially use those technologies. SaaS customization not only affects tenants but also provide new requirements for SaaS vendors that tenant-specific configuration may become an issue as all SaaS tenants share the same code base. Therefore, Sun [85] proposed a methodology framework to help SaaS vendors to plan and evaluate their capabilities and strategies for service configuration and customization. Truyen [89] proposed a context-oriented programming model to overcome tenant-specific variations so that all tenants can share the same code base. Service composition is another important approach for implementing SaaS application customization. Through service composition, tenants can quickly build new customized SaaS application. Tsai [105, 93] proposed a dependency-guided user centric service composition approach.

Chapter 2

# STA MODELS

A SaaS application can have multi-level tenants with following models:

- Single-Level STA (SSTA): One SaaS infrastructure supports multiple tenant applications, and each tenant application supports multiple end users. This is the same as tradition MTA.
- Two-Level STA (TSTA): One SaaS infrastructure supports multiple tenant applications, and a tenant application supports multiple sub-tenants. Both tenant and sub-tenant applications may support multiple end users.
- Multi-Level STA (MSTA): This is an extension of TSTA where a sub-tenant

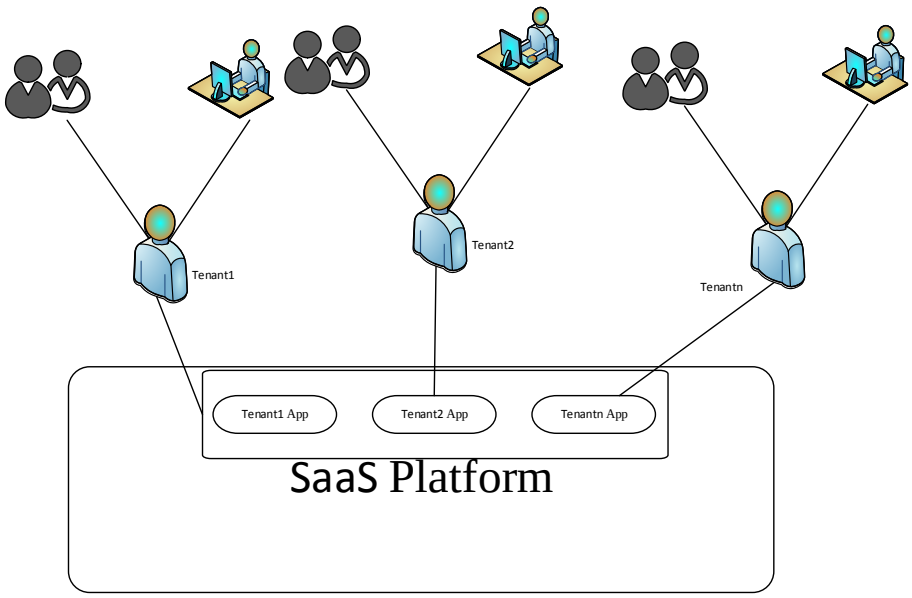



Figure 2.1: Single Level STA Example

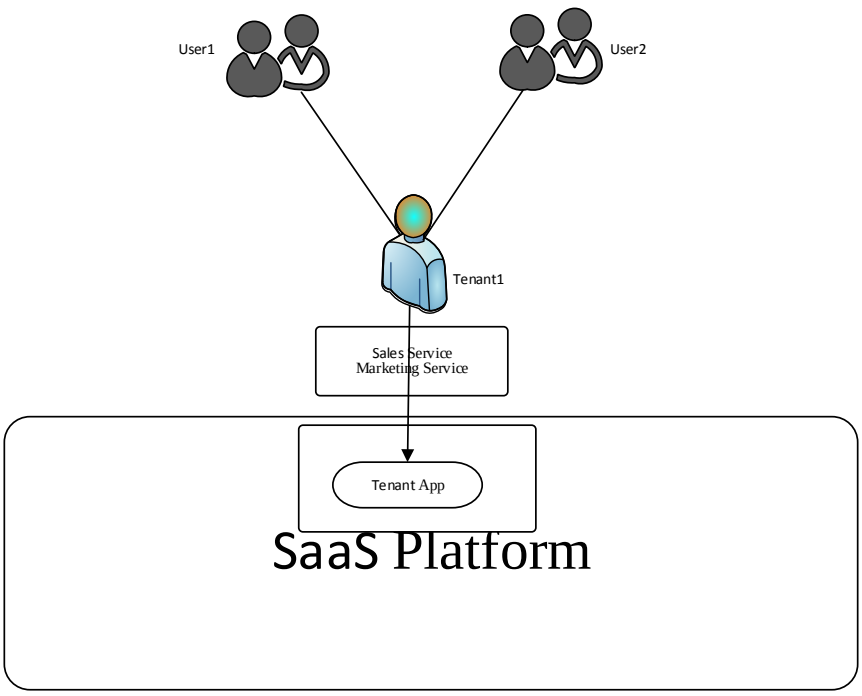



Figure 2.2: SingleOrg-STA Example

application can have its own tenants (sub-sub-tenants).

Tenants and their components and data are represented as $T = \{T_{C1}...T_{Cn}\}$ $\bigcup\{T_{D1}...T_{Dn}\}$. Then, sub-tenants are presented as $S = \{S_{C1}...S_{Cn}\}\bigcup\{S_{D1}...S_{Dn}\}$. Here, $T_{C1}$ represents tenant T has component C1 and $T_{D1}$ represents tenant T has data D1. $S_{C1}$ and $S_{D1}$ have similar concept except they represent sub-tenants. Additionally, $\widehat{T_{C1}S_{C2}}$ represents tenant T's component C1 shares the same instance with sub-tenant S's component C2; and C2 is a customized version of C1 while $\widetilde{T_{C1}S_{C2}}$ represents the two components C1 and C2 do not share the same instance. Further, $\overrightarrow{T_{C1}S_{C2}}$ represents components C1 and C2 share the same component but have different component instances. Tenant and sub-tenant relationships are shown in Figure 2.3.

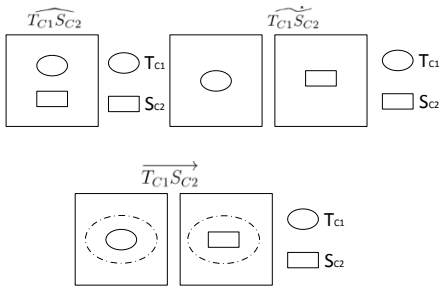



Figure 2.3: Tenant and Sub-tenant Relationship Example

## 2.1 SSTA Models

One SSTA model example is shown in Figure 2.1 with the following models:

1. Single-Organization Model (SingleOrg-SSTA): In this case, all tenants belong to the same organization with different customizations. This model is suitable when the organization is large with many divisions, and each division needs customized applications, but the resources can be shared among all these tenants as they belong to the same company. Furthermore, this approach is suitable if the company wishes to enforce overall company policies by supplying standardized services that tenants must use but cannot modify. Formally, this can be described as $\exists C \in T1_C, \exists C' \in T2_C \mid \widehat{CC'}$ or $\exists D \in T1_D, \exists D' \in T2_D \mid D \cap D' \neq \emptyset$. A SingleOrg-SSTA example is shown in Figure 2.2.

2. Multi-Organization Model (MultiOrg-SSTA): This is the case where each tenant may belong to different organizations. In this model, each tenant may compose its applications by customizing services in the SaaS infrastructure. This is traditional MTA. Formally, this can be described as $\forall C \in T1_C, \forall C' \in T2_C \mid \widetilde{CC'}$ and $\forall D \in T1_D, \forall D' \in T2_D \mid D \cap D' = \emptyset$. A MultiOrg-SSTA example is shown in Figure 2.4.

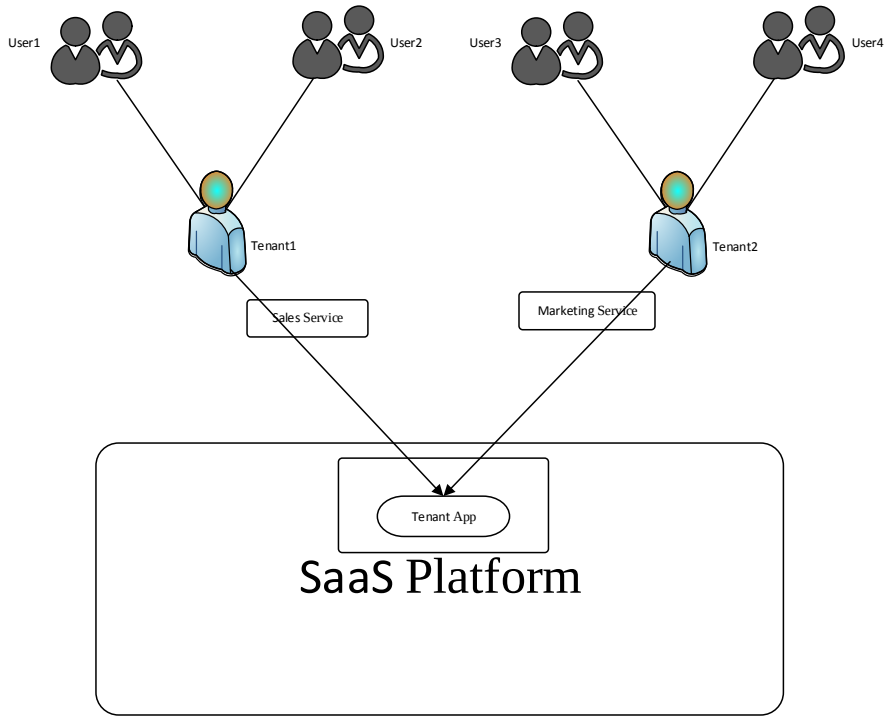



Figure 2.4: MultiOrg-STA Example

## 2.2 Two Level STA Models (TSTA):

TSTA is a model where a tenant can have both sub-tenants and end users as its customers while a sub-tenant can have end users as its customers only. There are mainly three actors in this model and their responsibilities are shown in Table 2.1. Depending on the sharing content between tenants and sub-tenants, this model has the following five sub-models.

1. Server-Customers Model (SC-TSTA): In this model, the server is a tenant of a SaaS component, distributes and supports its components. Sub-tenant developers can develop their own components using services provided by tenant components. An example shown in Figure 2.5 is ISVForce with Distributed Organization Model [10] where ISVForce supports Salesforce.com partners or Indepen-

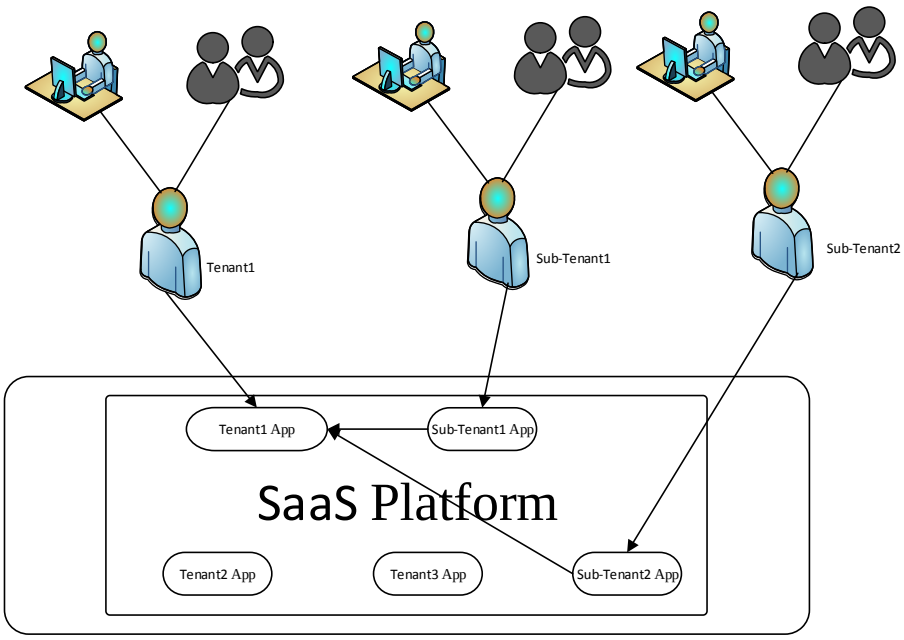



Figure 2.5: Server-Customers Example

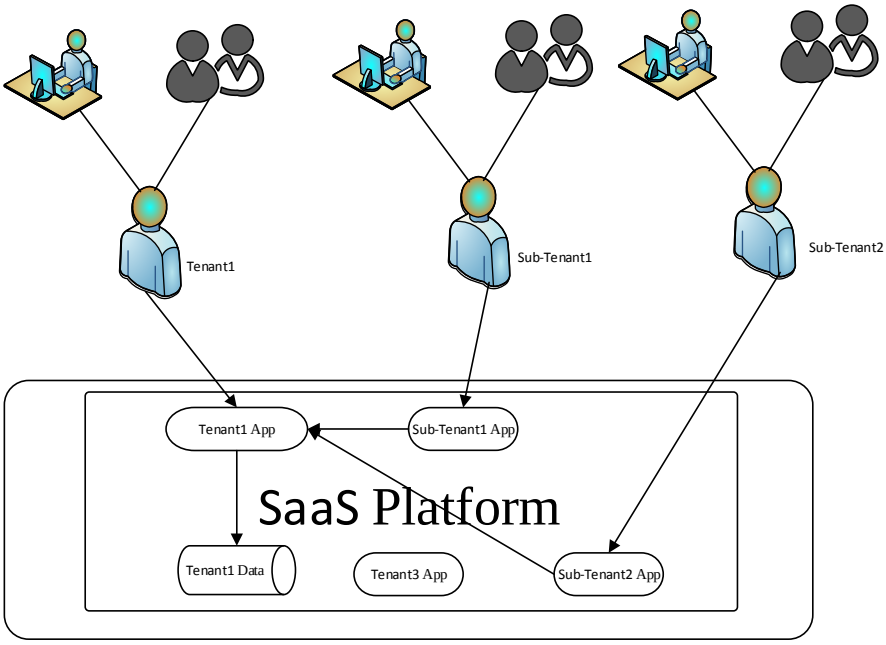



Figure 2.6: Software-Data Example

Table 2.1: TSTA Summary

| **Role** | **Responsibility** |
|---|---|
| SaaS Platform | • Allows tenant and sub-tenants developers to develop applications.<br>• Allows tenants to grant/remove/extend sub-tenants license to use tenant applications and data.<br>• Allows tenants to upgrade their tenant applications to support sub-tenants.<br>• Allows tenants to bill their sub-tenants.<br>• Gives the tenant the ability to support its customers. |
| Tenant Developers | • Develop and customize tenant applications on the SaaS platform.<br>• Publish customized applications to the platform to be used by both tenants and end users.<br>• Upgrade tenant applications and automatically push the update to all sub-tenant applications without interfering sub-tenants.<br>• Provide the license agreement for sub-tenants to use their tenant applications.<br>• Bill end users and sub-tenants.<br>• Support customers. |
| Sub-Tenant Developers | • Use the platform to develop sub-tenant applications.<br>• Subscribe customized tenant applications and data.<br>• Need both tenant applications and data with sub-tenant's data to complete applications. |

dent Service Vendors (ISVs) to build, sell and distribute their SaaS components and ISVs serve their customers and push upgrades to all of them automatically [11]. Its formal definition can be described as $\exists C \in T_C, \exists C' \in S_C \mid \widehat{CC'}$ and $\forall D \in T_D, \nexists D' \in S_D \mid D \cap D' \neq \emptyset$. This model is suitable when the SaaS provider wants to support its partners or ISVs to build, sell and support their customized SaaS components.

2. Software-Data Model (SD-TSTA): In this model, the tenant owns SaaS components and data, shared by its sub-tenants, and sub-tenants can customize tenant's components. One SD-TSTA is shown in Figure 2.6. Its formal definition can be described as $\exists C \in S_C, \exists C' \in T_C \mid \widehat{CC'}$ and $\exists D \in T_D, \exists D' \in S_D \mid D \cap D' \neq \emptyset$. This model is suitable when an organization who sells products has sub-organizations and the sub-organizations share same sale process and can sell the organization's products.

3. Master-Slaves Model (MS-TSTA): In this model, both the tenant and its sub-tenants have their isolated SaaS instances but share the same code base. However, the tenant can access the sub-tenants' data and sub-tenants can customize tenant components. One MS-TSTA example is shown in Figure 2.7. Its formal definition can be described as $\exists C \in T_C, \exists C' \in S_C \mid \overrightarrow{CC'}$ and $\forall D' \in S_D, \exists D \in T_D \mid D \cap D' \neq \emptyset$. This model is suitable when an organization has sub-organizations and wants to manage sub-organizations' data such as human resource information.

4. Slave-Masters Model (SM-TSTA): SM-TSTA is similar to MS-TSTA except sub-tenants can access tenant's data. Therefore, the data sharing flow will be top down, not bottom up as in MS-TSTA model. One SM-TSTA example is shown in Figure 2.8. Its formal definition can be described as $\exists C \in T_C, \exists C' \in S_C \mid \overrightarrow{CC'}$

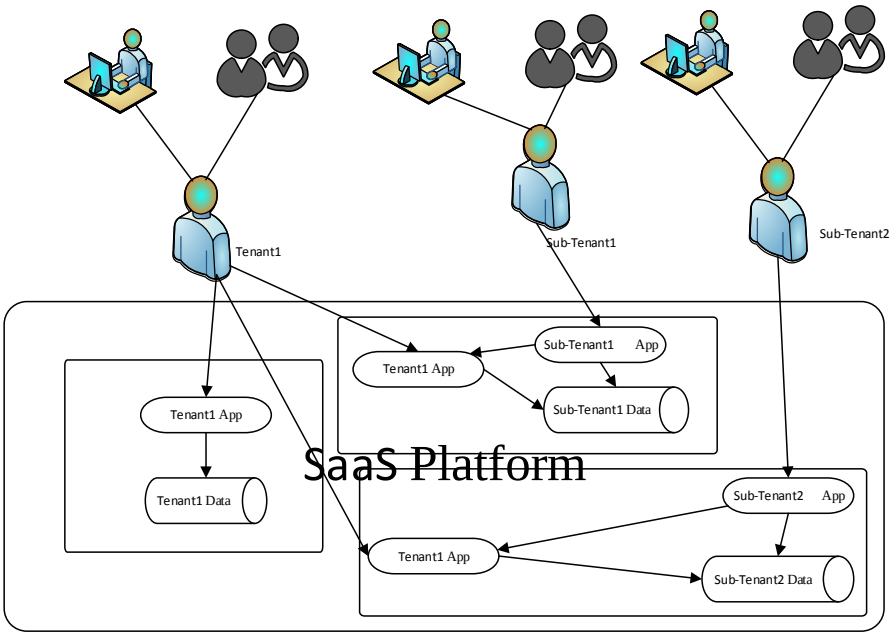



Figure 2.7: Master-Slaves Example

and $\exists D \in T_D, \exists D' \in S_D \mid D \cap D' \neq \emptyset$. This model is suitable when an organization does not have any products but wants to sell other's products, so it needs the product data to complete the sale process.

5. Partner-Partners STA (PP-TSTA): In this model, both the tenant and its sub-tenants have their isolated SaaS instances and data. In addition, both tenants and sub-tenants can customize each other's components and data. One PP-TSTA is shown in Figure 2.9. Its formal definition can be described as $\exists C \in T_C, \exists C' \in S_C \mid \overrightarrow{CC'}$ and $\exists C' \in S_C, \exists C \in T_C \mid \overrightarrow{C'C}$ and $\exists D \in T_D, \exists D' \in S_D \mid D \cap D' \neq \emptyset$ and $\exists D' \in S_D, \exists D \in T_D \mid D' \cap D \neq \emptyset$. This model is suitable when an organization is a partner with another organization that they want to share some components and data from each other.

All five TSTA models are different in sharing components and data between the tenant and its sub-tenants. A comparison of these five TSTA models is shown in

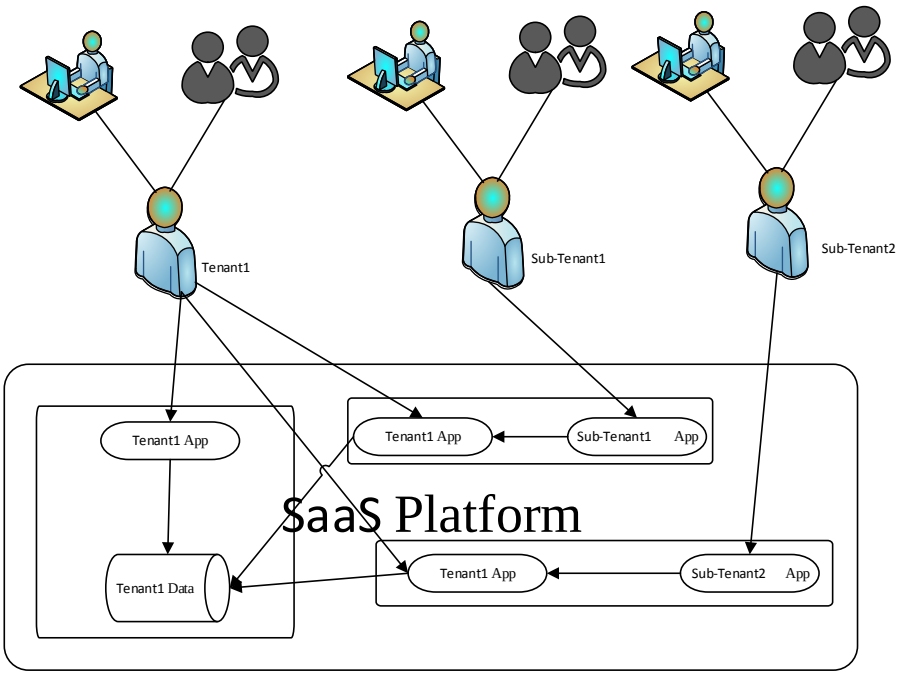



Figure 2.8: Slave-Masters Example

Table 2.2.

## 2.3 Multi-level STA (MSTA)

The MSTA model can be obtained by extending TSTA into more levels, and this means the sub-tenants can also have sub-tenants. Therefore, MSTA can be classified as following sub-models. Following the TSTA models, MSTA can have SC-MSTA, SD-MSTA, MS-MSTA, SM-MSTA, and PP-MSTA models. In these models, a tenant and its sub-tenants, a sub-tenant and its sub-sub-tenants share SaaS applications and data consistent with the corresponding TSTA models. Hybrid models are possible, but due to its complexity, they will not be emphasized. One SC-MSTA example is shown in Figure 2.10.

Table 2.2: TSTA Model Comparison

| Models | Tenant components | Data | Customization | Upgrade & Distribution |
|---|---|---|---|---|
| Server-Customers | • Tenants build, sell and distribute SaaS components.<br>• Tenants share the same component instance with their sub-tenants. | Tenants do not share their data with their sub-tenants and have no accesses to sub-tenants' data. | Sub-tenants can customize the tenant's components. | Tenants upgrade their components then propagate them to their sub-tenants. |
| Software-Data | • Tenants develop and own the SaaS component.<br>• Tenants share the same component instance with their sub-tenants. | • Sub-tenants can access the tenant's sharing data.<br>• Sub-tenants can only access the data related to them. | • Tenants can customize SaaS components.<br>• Sub-tenants can customize SaaS components and their tenant components.<br>• Tenants can define the scope that sub-tenants can customize. | Tenants upgrade the tenant applications and the SaaS platform propagates the update. |
| Master-Slaves | The tenants share components with sub-tenants but they have different component instances. | Tenants have access to the sub-tenants data and sub-tenants can not have access to the tenant data. | Sub-tenants can customize the shared tenant components. | Tenants upgrade their components and propagate them to their sub-tenants as they do not share the same component instance. |
| Slave-Masters | The tenants share components with sub-tenants but they have different component instances. | Tenants do not have access to the sub-tenants data but sub-tenants can have access to the tenants' sharing data. | Sub-tenants can customize the shared tenant components. | Tenants upgrade their components and propagate them to their sub-tenants as they do not share the same component instance. |
| Partner-Partner | • Tenants share components and data with sub-tenants.<br>• Sub-tenants share components and data with their tenants. | • Tenants share some data with their sub-tenants.<br>• Sub-tenants share some data with their tenants. | Both tenants and sub-tenants can customize shared SaaS components. | • Tenants upgrade their components and propagate them to their sub-tenants as they do not share the same component instance.<br>• Sub-tenants upgrade their SaaS components and propagate them to their tenant as they do not share the same component instance. |

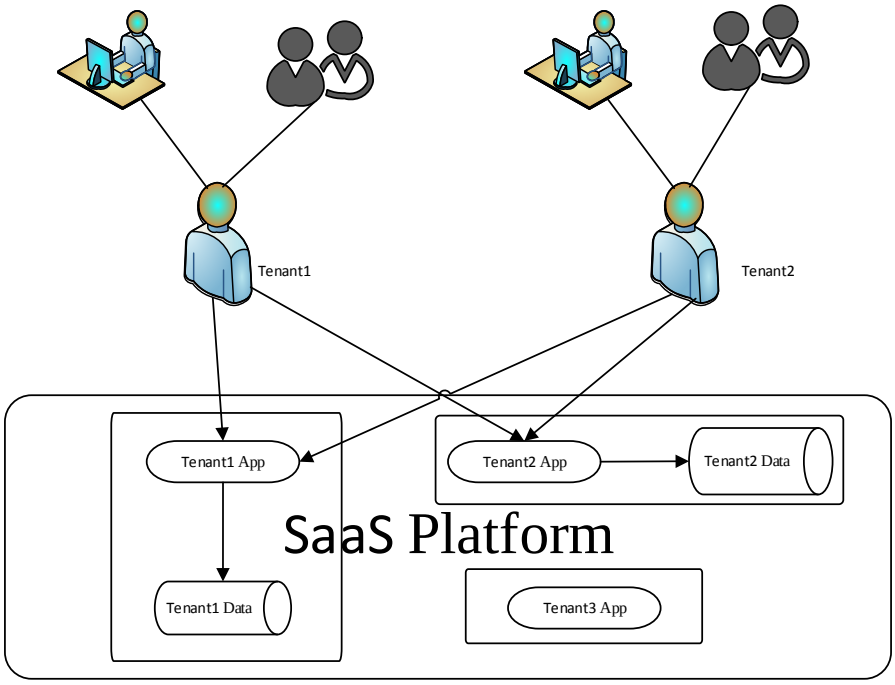



Figure 2.9: Partner-Partners Example

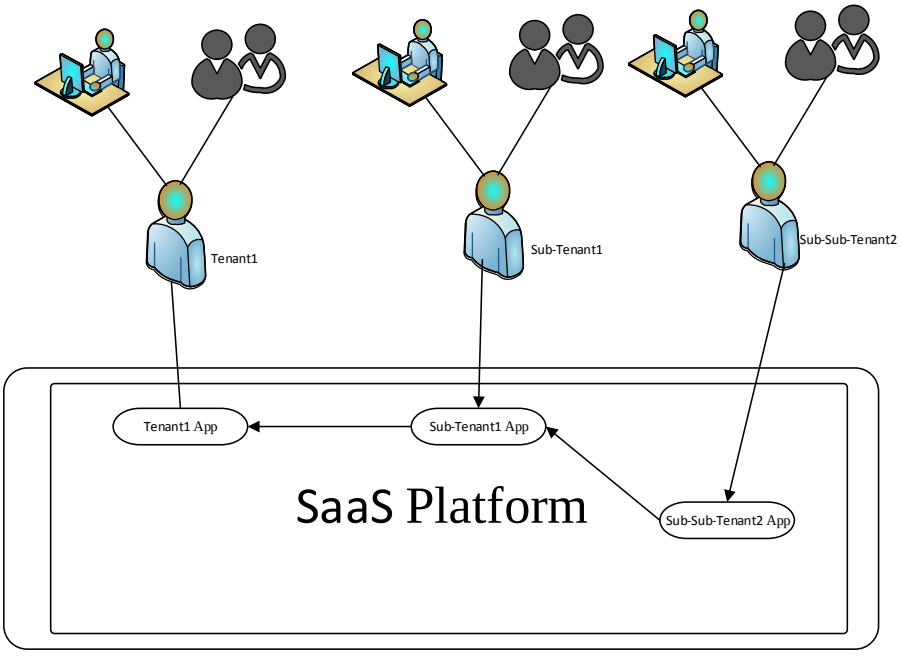



Figure 2.10: Server-Customers MSTA Example

## 2.4 STA Properties

There are four types of properties among STA models: transitive sub-tenant, symmetric reflective sub-tenant, implied sub-tenant and equivalent sub-tenant shown in Figure 2.11. And examples of each STA properties are shown in Table 2.3.

1. Transitive sub-tenants: a tenant, its sub-tenants and sub-sub-tenants have a transitive sub-tenant property if they have the following characteristics: If $A \xrightarrow{<M,P,D>} B$ and $B \xrightarrow{<M,P',D'>} C$ then $A \xrightarrow{<M,P'',D''>} C$. Here, $A \xrightarrow{<M,P,D>} B$ means A and B sharing application P and data D in a way of model M. Additionally, $P \cap P' \neq \emptyset$, $P'' = P \cap P'$ and $D'' = D \cap D'$. M can be SC-TSTA, SD-TSTA, MS-TSTA, SM-TSTA or PP-TSTA.

2. Symmetric reflective sub-tenants: a tenant, its sub-tenants and sub-sub-tenants have a symmetric reflective sub-tenant property if they have the following characteristics: If $A \xrightarrow{<M,P,D>} D$ then $D \xrightarrow{<M,P',D'>} A$. Here, $P' = P$ and $D' = D$. M can only be PP-TSTA.

3. Implied sub-tenants: a tenant, its sub-tenant, and sub-sub-tenants have an implied sub-tenant property if they have the following characteristics: If $A \xrightarrow{<M,P,D>} E$ and $F \xrightarrow{<M',P',D'>} E$ then $A \xrightarrow{<M'',P'',D''>} F$. Here, $\overrightarrow{PP''}$, $\overrightarrow{PP'P''}$ and $D'' = D \cup D'$. M, M' and M" can be PP-TSTA.

4. Equivalent sub-tenants: a tenant, its sub-tenants, and sub-sub-tenants have an equivalent sub-tenant property if they have the following characteristics: If $B \xrightarrow{<M,P,D>} G$ and $G \xrightarrow{<M',P',D'>} B$ then $B \xrightarrow{<M'',P'',D''>} G$. Here,$\overrightarrow{PP''}$, $\overrightarrow{PP'P''}$ and $D'' = D \cup D'$. M can be MS-TSTA, M' can be SM-TSTA and M" can be PP-TSTA only.

Table 2.3: Examples of STA Properties

| Property Names | Examples | Models With This Property |
|---|---|---|
| Transitive sub-tenants | In Figure 2.11, tenant A has Master-Slaves relationship with sub-tenant B; sub-tenant B has Master-Slaves relationship with sub-sub-tenant C. When the components and data shared by A and B, and B and C are the same, A also has Master-Slaves relationship with C. | • Server-Customers<br>• Software-Data<br>• Master-Slaves<br>• Slave-Masters<br>• Partner-Partners |
| Symmetric reflective sub-tenants | In Figure 2.11, tenant A has Partner-Partners relationship with sub-tenant D, which D automatically has Partner-Partners relationship with A. | • Partner-Partners |
| Implied sub-tenants | In Figure 2.11, tenant A has Partner-Partners relationship with sub-tenant E; tenant F has Partner-Partners relationship with sub-tenant E. When the components and data shared by A and E, and F and E are the same, A also has Partner-Partners relationship with F. | • Partner-Partners |
| Equivalent sub-tenants | In Figure 2.11, tenant B has Master-Slaves relationship with sub-tenant G. At same time, G has Slave-Masters relationship with sub-tenant B. By this way, B has Partner-Partners relationship with G. | • Master-Slaves<br>• Slave-Masters<br>• Partner-Partners |

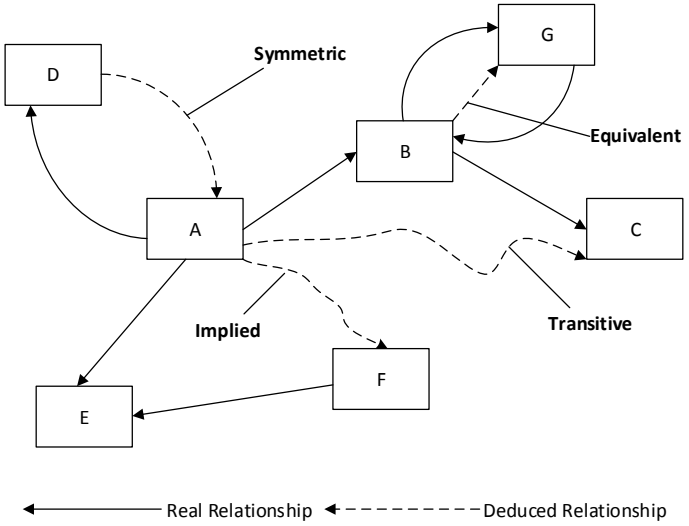



Figure 2.11: STA Property Relationships

# Chapter 3

# STA SECURITY CONSIDERATION AND ACCESS PERMISSION MODELS

STA has significant impact on both SaaS security and SaaS application and data access.

## 3.1 STA Security Consideration

### *3.1.1 SSTA*

As this is the regular MTA, any security mechanism used in current SaaS can be applied.

### *3.1.2 TSTA*

Tenants can use the SaaS provider's market to sell and distribute their SaaS applications. Normally, to protect the SaaS provider's reputation and customers, he has to review each SaaS application to make sure it follows the SaaS provider security standards and policies. The SaaS provider should publish those standards in the website so they can be easily accessed by tenants. They should also give the tenants a guidance on how to test the security of their SaaS applications, and point out where the SaaS provider will check on the SaaS applications. The SaaS provider may accept, reject, and require tenants to make changes to their SaaS applications. In addition, the sub-tenants can use built-in security of SaaS applications to define their end users and give them permissions when subscribe to the tenants' applications. At last, the best practices for building secure SaaS applications should be applied and the SaaS providers should make sure that the tenant and sub-tenant developers follow their security or policies on each SaaS application development.

### *3.1.3 MSTA*

This model is same as two-level STA except it has more level sub-tenancy. Therefore, both tenants developers and sub-tenant developers should follow the same process. In addition, they also need to follow the same security guidelines and policies of the SaaS provider when they build applications, make any customizations, and even define end users and assign permissions.

## 3.2 Permission Access Models

There are two types of permission access models: for applications and data.

### *3.2.1 Permission Access Model for Applications*

Both tenants and sub-tenants can own (O), subscribe (S) and use (U) SaaS applications. $T_{Ai_U}$ and $T_{Ai_O}$ are tenant T's access values for application $Ai$. Similarly, $S_{Ai_U}$, $S_{Ai_S}$ and $S_{Ai_O}$ are sub-tenant S's access values for application $Ai$. $T_{Ai_U}$, $T_{Ai_O}$, $S_{Ai_U}$ and $S_{Ai_S}$ can be used to deduce sub-tenants' possible use permission of the tenant's applications. The STA application access formulas for an application A can be presented as equation (3.1).

$$S_{Ai_U} = ((T_{Ai_U} \cup T_{Ai_O}) \cap S_{Ai_S}) \cup S_{Ai_O} \tag{3.1}$$

The formula shows that sub-tenant S has use permission of application Ai when one of following case happens:

- $((T_{Ai_U} \cup T_{Ai_O}) \cap S_{Ai_S})$ means the tenant T has own or use permission of the application Ai and sub-tenant S subscribes application Ai.
- $S_{Ai_O}$ means sub-tenant S has own permission.

$$\begin{cases} S_{Di_R} = ((T_{Di_R} \cup T_{Di_W} \cup T_{Di_D} \cup T_{Di_O}) \cap S_{Di_S}) \cup S_{Di_O} \\ S_{Di_W} = ((T_{Di_W} \cup T_{Di_O}) \cap S_{Di_S}) \cup S_{Di_O} \\ S_{Di_D} = ((T_{Di_W} \cup T_{Di_D} \cup T_{Di_O}) \cap S_{Di_S}) \cup S_{Di_O} \end{cases} \tag{3.2}$$

### *3.2.2 Permission Access Model for Data*

Both tenants and sub-tenants can own (O), read (R), write (W) and delete (D) data represented as $O$, $R$, $D$. $T_{Di_R}, T_{Di_W}, T_{Di_D}$ *and* $T_{Di_O}$ are tenant T's access values of data. As well, $S_{Di_R}, S_{Di_W}, S_{Di_D}$ *and* $S_{Di_O}$ are sub-tenant S's access values of data Di. In addition, $S_{Di_S}$ means sub-tenant subscribes the data Di. The formula shown in (3.2) is to deduce the data permission of a sub-tenant and is based on three assumptions below:

- When tenant or sub-tenant has own permission of a data, it also has read, write and delete permissions.
- When tenant or sub-tenant has write permission of a data, it also has delete and read permissions.
- When tenant or sub-tenant has delete permission of a data, it also has read permission.

Based on previous three assumption, the STA data access formulas of a data Di for sub-tenant S can be presented as equation (3.2):

The formula can be described as following:

- $((T_{Di_R} \cup T_{Di_W} \cup T_{Di_D} \cup T_{Di_O}) \cap S_{Di_S}) \cup S_{Di_O}$ means sub-tenant S has read permission of data Di when its tenant T has read, write, delete or own permissions and S subscribes Di from T. In addition, $S_{Di_O}$ means sub-tenant S also has read permission when S owns the data Di.

- $((T_{Di_W} \cup T_{Di_O}) \cap S_{Di_S})$ means sub-tenant S has write permission of data Di when its tenant T has write or own permissions and S subscribes Di from T. In addition, $S_{Di_O}$ means sub-tenant S also has write permission when S owns the data Di.

- $((T_{Di_W} \cup T_{Di_D} \cup T_{Di_O}) \cap S_{Di_S})$ means sub-tenant S has delete permission of data Di when its tenant T has write, delete or own permissions and S subscribes Di from T. In addition, $S_{Di_O}$ means sub-tenant S also has delete permission when S owns the data Di.

# Chapter 4

# STA CUSTOMIZATION MODELS

In STA, the tenants in upper STA hierarchy levels can specify the customization options and define what can be customized for the sub-tenants in lower levels.

## 4.1 STA Customization Techniques

SaaS application customization can be achieved by the following ways:

1. By coding: the developers develop the code, then they publish the code to the SaaS platform. An example of a SaaS application that uses this technique is Force.com [77] where Apex routines are written by the developers to add custom business logic for the application.

2. By variability points and options: the developers can choose several options of customization only and those options are called variability points. There are several types of variability points:

   (a) Fixed variability points with fixed options such that the options are already verified by SaaS infrastructure before deploying.

   (b) Fixed variability points that tenants can provide their options with verification mechanisms.

   (c) The SaaS provider provides list of templates for variation points with list of options and constraints. The tenant can create his variation points and options for the selected template. The variation template is stored in the SaaS database for other tenants to reuse. One approach for variability

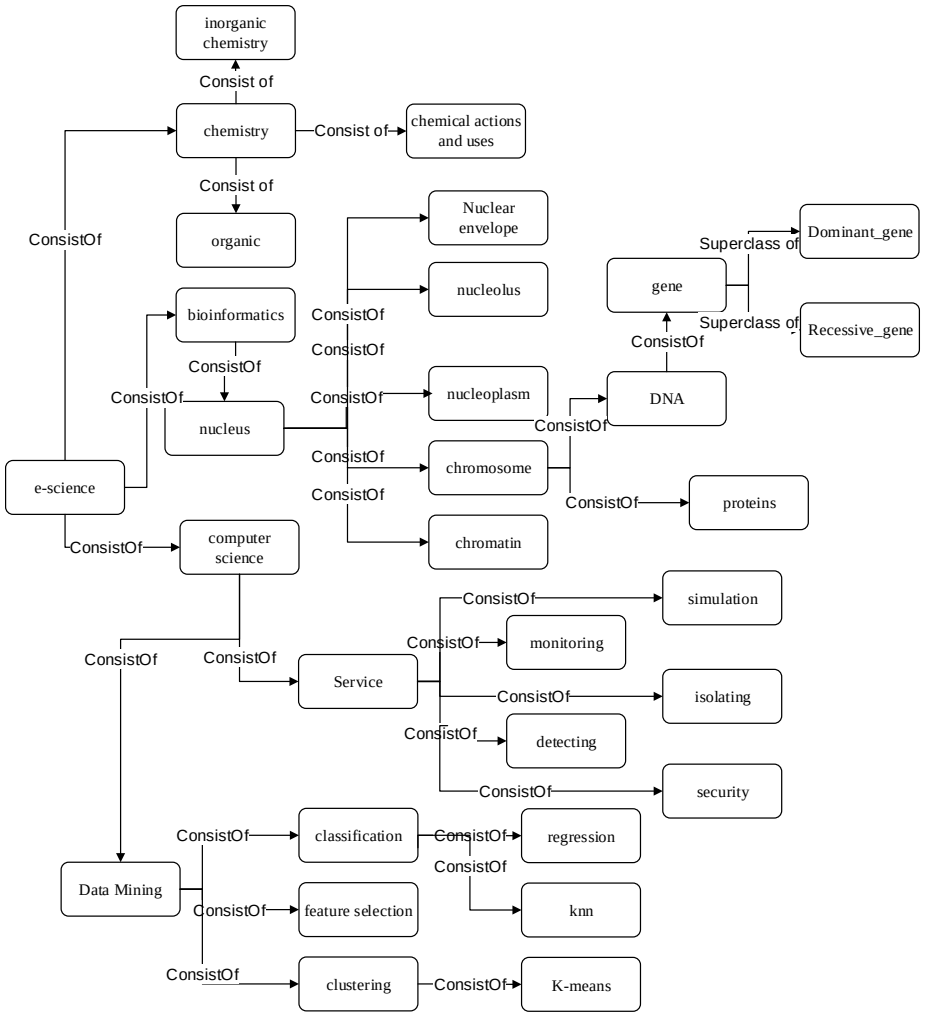



Figure 4.1: E-Science Ontology Example

points uses ontology information so that uses available variability points and their options can be discovered by other tenants [95]. One ontology example is shown in Figure 4.1 built on the data from myexperiment.org [66].

3. By composition: the developers build the entire tenant application in a service-oriented manner by composing GUI, workflow, service and data components with a recommendation system. An example of this approach is the OIC model

[13]. In addition, this approach is further improved by using the previous tenants applications to recommend components this is the Grapevine model [14].

4. By configuration: the developers do not need to make any coding. Metadata definitions are used to define the logic and the options that the customers can change. When the tenant selects or changes an option, the SaaS platform generates the application by interpreting the metadata. The key difference between the changes to metadata-based logic and changes made by code is that the SaaS application provider determines the various options or the ways the metadata can be changed.

5. By hybrid: The developer can customize their applications by using one or more of the previous approaches. In this way, the SaaS provider gives the tenants flexibility of customizations.

Depending on the relationship between tenant and sub-tenants, different customization techniques may be offered in different STA models. Following is a discussion of the customization options for each STA model.

### *4.1.1 For SSTA*

In both the SingleOrg-SSTA and MultiOrg-SSTA models, a tenant can customize its application by using the customization techniques offered by the SaaS provider where the customization can use configuration, coding or other techniques.

### *4.1.2 For TSTA*

TSTA have different customization options.

1. SC-STA: In this model, the tenant either initially customize the SaaS provider's base application and templates or creates his SaaS application from the scratch.

Then, he sells licenses of his SaaS applications to sub-tenants. In addition, sub-tenants can further customize their applications. The way that the tenant customizes the base application and templates affects the flexibility of customization that the sub-tenants can have. The tenant can make customizations in three ways:

(a) Using customization techniques offered by SaaS providers: as these customization options provided by SaaS provider, it gives sub-tenants to reuse them. However, other factors such as the license or the edition the sub-tenants making agreement with the tenant, also affect the customization that sub-tenants can make.

(b) Coding: Customization techniques offered by SaaS providers sometimes does not satisfy the tenant's needs. Therefore, they develop custom code. However, using the coding to make customization leads to the limited options of customizations for the sub-tenants as they often cannot reuse customization options provided by the SaaS.

2. SD-STA: The customization options that applied to CS-STA can also be used in this model as sub-tenants share the same tenant's SaaS application. However, in this model, the tenant's data can also be shared by his sub-tenants. The changes to data object need be distributed to the sub-tenants that the tenant assigns. This process can be achieved or by using tools and code.

3. MS-STA: In this model, sub-tenants inherit their SaaS applications from the tenants SaaS application but have isolated application instances. Therefore, each sub-tenant can have any customization options inherited from the tenant application. The tenant can define the customization options for sub-tenants. In addition, the tenant may customize the SaaS application and have extra

logic, workflows, reports and dashboards that do not show to sub-tenants but share a different customized application to his sub-tenants. Due to inheritance relationship between the tenant and his sub-tenants, customizations to the tenant application logic, workflows, reports, and data may automatically affect the sub-tenants' applications. However, the tenant can select customizations to only apply to a selected group of sub-tenants. Thus, not all sub-tenants see the same customization. At last, any customizations made by the tenant should not change or affect the sub-tenants' UI customizations.

4. SM-STA: The customization options that applied to CS-STA can also be used in this model as sub-tenants inherit tenant's SaaS application with different application instances. In addition, sub-tenants need data from the tenant. Therefore, any customizations of the tenant data may be shared as the tenant can define the sub-tenants get which data customization. The changes to data object need be distributed to the sub-tenants that the tenant assigns. This process can be achieved or by using tools and code.

5. PP-STA: As one knows in STA property, two tenants imply PP-STA if they are MS-STA and SM-STA respectively. Therefore, any customization options applied in MS-STA and SM-STA can also be used in this model.

### *4.1.3 For MSTA*

In this model, due to the sharing between low and high levels' tenants, any customization made by low level tenants will affect the high-level tenants. Any customization made on data, workflows, or UI, program or tools should be distributed to other tenants. If the new customizations made by low level's tenant are not broadcast, other tenants in high levels may have data schema inconsistency problems when

$$S_{Ai_{FC}} = ((T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S}) \cup S_{Ai_O} \tag{4.1}$$

they want to share data.

## 4.2 STA Customization Deduction

The following three customization models will be used:

1. No customization model (NC): Tenant and sub-tenants cannot make any customizations to their applications.

2. Partial customization model (PC): Tenant and sub-tenants can customize some features in their applications such as GUI, workflow, services, and data but they cannot customize all these features.

3. Full customization model (FC): Tenant and sub-tenants can fully customize their applications.

The formula shown in (4.1) that is used to deduce full customization permission of an application and (4.2) that is used to deduce the Gui, workflow, service and data customization permission of an application for a sub-tenant and is based on two assumptions below:

- When tenant or sub-tenant has own permission of an application, it also has full customization permissions.

- When tenant or sub-tenant has full customization of an application, it also has all partial customizations for application's GUI, workflow, service and data.

Based on previous two assumption, the full customization permission of an application Ai for sub-tenant S is introduced at Equation 4.1.

From Equation 4.2, one can see follows:

$$\begin{cases} S_{Ai_{PC_G}} = ((T_{Ai_{PC_G}} \cup T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S}) \cup S_{Ai_{FC}} \cup S_{Ai_O} \\ S_{Ai_{PC_W}} = ((T_{Ai_{PC_W}} \cup T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S}) \cup S_{Ai_{FC}} \cup S_{Ai_O} \\ S_{Ai_{PC_S}} = ((T_{Ai_{PC_S}} \cup T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S}) \cup S_{Ai_{FC}} \cup S_{Ai_O} \\ S_{Ai_{PC_D}} = ((T_{Ai_{PC_D}} \cup T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S}) \cup S_{Ai_{FC}} \cup S_{Ai_O} \end{cases} \tag{4.2}$$

1. $(T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S})$ means sub-tenant has full customization permission of the application Ai when its tenant has full customization or own permission and S subscribe the application Ai.

2. $S_{Ai_O}$ means sub-tenant also has full customization permission of the application Ai when S has own permission of the application Ai.

In addition, the partial customization permission of an application Ai for sub-tenant S is presented at Equation 4.2

From Equation 4.2, one can see follows:

1. $((T_{Ai_{PC_G}} \cup T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S})$ represents a sub-tenant S has partial GUI customization permission of the application Ai when its tenant has partial GUI customization permission, full customization permission or own permission of the application Ai and it also subscribes the application Ai. In addition, sub-tenant S also has partial GUI customization permission when it has full customization permission or own permission of the application Ai.

2. $((T_{Ai_{PC_W}} \cup T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S})$ represents a sub-tenant S has partial workflow customization permission of the application Ai when its tenant has partial workflow customization permission, full customization permission or own permission of the application Ai and it also subscribes the application Ai. In addition, sub-tenant S also has partial workflow customization permission when it has full customization permission or own permission of the application Ai.

3. $((T_{Ai_{PC_S}} \cup T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S})$ represents a sub-tenant S has partial service customization permission of the application Ai when its tenant has partial service customization permission, full customization permission or own permission of the application Ai and it also subscribes the application Ai. In addition, sub-tenant S also has partial service customization permission when it has full customization permission or own permission of the application Ai.

4. $((T_{Ai_{PC_D}} \cup T_{Ai_{FC}} \cup T_{Ai_O}) \cap S_{Ai_S})$ represents a sub-tenant S has partial data customization permission of the application Ai when its tenant has partial data customization permission, full customization permission or own permission of the application Ai and it also subscribes the application Ai. In addition, sub-tenant S also has partial data customization permission when it has full customization permission or own permission of the application Ai.

## 4.3 Variant Point Model

STA allows sub-tenant to customize tenant's applications, which introduces new challenges. Therefore, a Variant Point model is introduced.

### *4.3.1 VP Classification*

Variation point (VP) is the place that can have multiple choices. One VP is composed by options that developers can select from and rules that options must obey. Normally, tenant developers define the options and rules that each VP can have and sub-tenant can only choose values from the options according to the rules. VP can be described by VP specification (VPs) and VP instance (VPi). VPs describe the interface of VP and VPi implements VPs. There are three types of VP to implement STA customizations: by fixed variation points and fixed options, by fixed variations but allow tenant options and by flexible variation points and options.

1. By fixed variation points and fixed options (FVPFO): this is an easy way for sub-tenants to customize applications but with less flexibility.

2. By fixed variations but allow tenant options (FVATO): this is the place in SaaS application that sub-tenant developers can add more options such as adding attributes and rules for data or fields for UI.

3. By flexible variation points and options (FVPO): there are multiple places where variation point can be put.

According to [101], a SaaS application can be classified into four layers, UI, Workflow, Service and Data. Based on the place where VP is set, VP has different content.

1. GUI: Tenant and Sub-tenant are able to add new attributes to the GUI forms as fields. The fields can be simple editable text or media object such as image and video. One example is shown in figure 4.2 on the following page. From figure 4.2 on the next page, one can see tenant developer or sub-tenant developer can choose different fields combination from user id, password, validation code, email address and cell phone for registration VP.

2. Service: Tenant and Sub-tenant are able to choose different services or compose services. Tsai proposes a dependency-guided service composition in [107] to compose services. One example is shown in figure 4.3 on the following page. From figure 4.3 on the next page, one can see tenant developers or sub-tenant developers can choose some features for shipping service VP such as choosing validation zip and address.

3. Workflow: Tenant and Sub-tenant are able to edit the business process by adding or deleting the steps that workflow has. One example is shown in figure 4.4 on the following page. From figure 4.4 on the next page, one can see

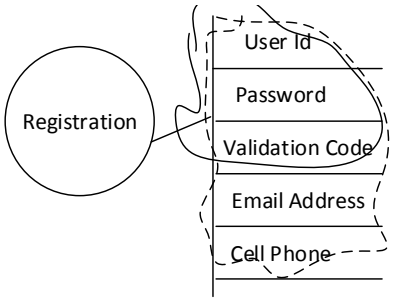



Figure 4.2: UI VP Example

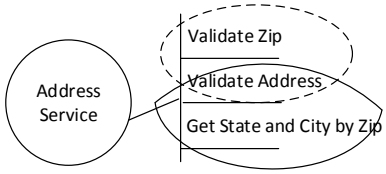



Figure 4.3: Service VP Example

tenant developers or sub-tenant developers can choose some steps such as fill address and validate address.

4. Data: Tenant and Sub-tenant are able to add new properties to the data and define rules for them. One example is shown in figure 4.5 on the following page. From figure 4.5 on the next page, one can see tenant developers or sub-tenant developers can choose blue for Color data VP.

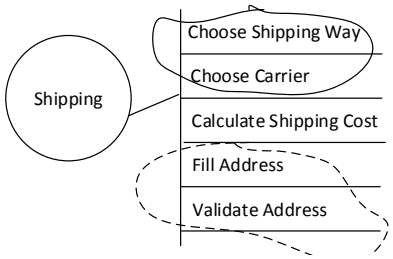



Figure 4.4: Workflow VP Example

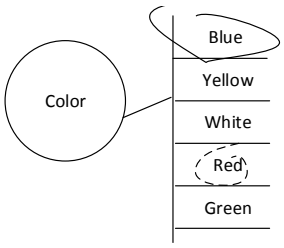



Figure 4.5: Data VP Example

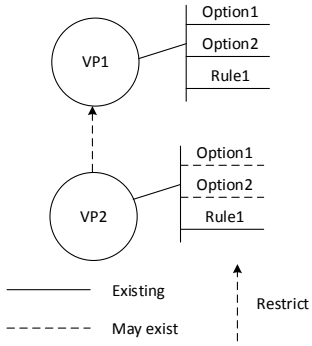



Figure 4.6: Restrict Relationship

### *4.3.2 VP Relationships*

There are five relationships among VPs.

1. Restrict Relationship: when two VPs have this relationship, child VP has less or equal options but may have more rules than parent VP. One example is shown in figure 4.6. From figure 4.6, one can see VP2 has less options than VP1 as VP2 is restricted to VP1.

2. Inherit Relationship: when two VPs have this relationship, child VP has all parent VP's options and rules. In addition, child VP can add more options or overwrite parent VP's options. One example is shown in figure 4.7 on the following page. From figure 4.7 on the next page, one can see VP2 has more options tham VP2 as VP2 inherit VP1.

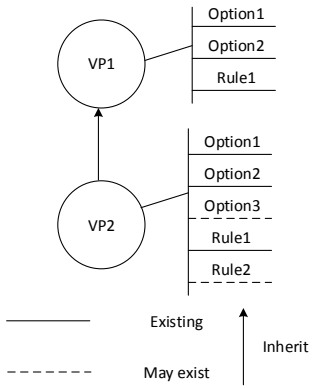



Figure 4.7: Inherit Relationship

3. Extend Relationship: this relationship has all features that inherit relationship has. In addition, it can change the type of VP from FVATO to FVPO or from FVPFO to FVPO by allowing change the place of VP. One example is shown in figure 4.8 on the following page. From figure 4.8 on the next page, one can see VP2 not only has more options than VP1 but also becomes a FVPO.

4. Compose Relationship: this relationship compose two or more VPs to become a new VP. The new composed VP has all VPs' options and rules except the types. There are two scenarios that affect the type of VP.

   (a) One of the composed VPs is fixed VP: the new composed VP becomes a fixed VP. In addition, if there is one of the composed VPs is FVATO or FVPO, the new comopose VP is FVATO. Otherwise, the composed new VP is FVPFO.

   (b) All composed VPs are FVPO: the new composed VP becomes a FVPO.

   One example is shown in figure 4.9 on page 37. From figure 4.9 on page 37, one can see one FVPFO VP1 and one FVPO VP2 compose VP1. Therefore,

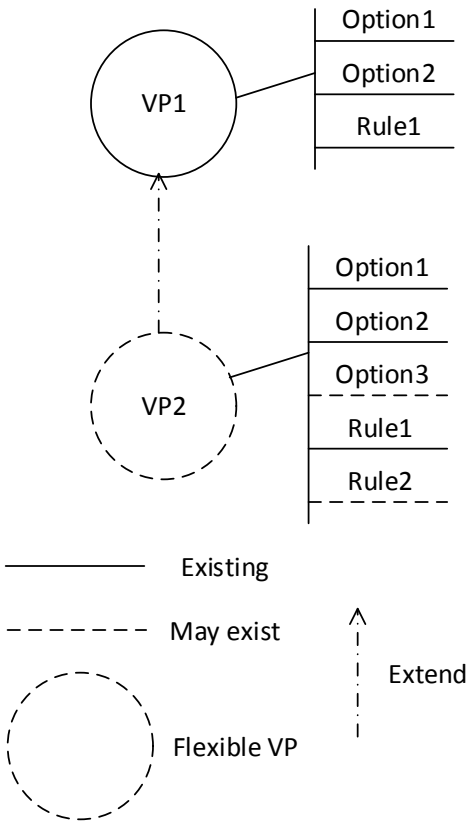



Figure 4.8: Extend Relationship

VP1 has all options and rules from both VP2 and VP1. At same time, VP1 is a FVPFO.

5. Implement Relationship: This relationship is between VP specification and VP instance. VP specification define what the VP is while VP instance implement VP specification. One example is shown in figure 4.10 on the following page. From figure 4.10 on the next page, one can see one VP specification can be implemented by more than one VP instances.

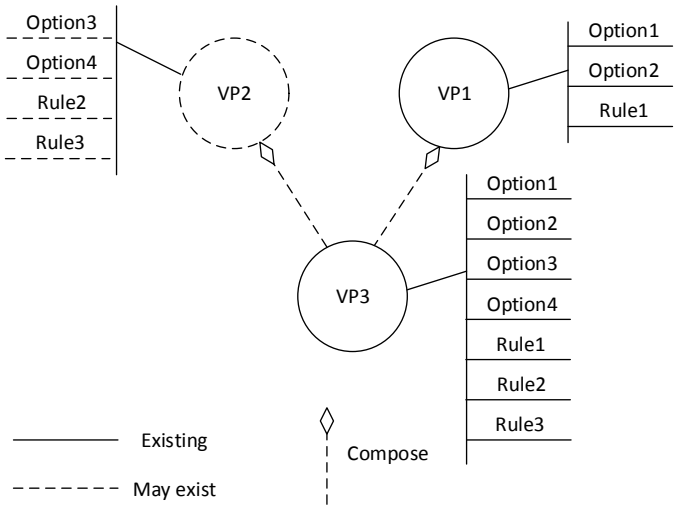



Figure 4.9: Compose Relationship

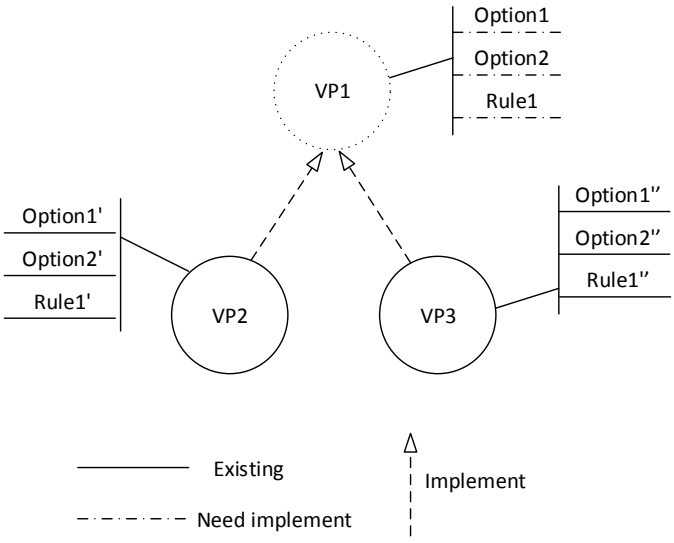



Figure 4.10: Implement Relationship

### *4.3.3 VP Properties*

There are following properties when VPs have relationships above. One example is shown in figure 4.11 on the following page.

1. Transitive: VP and another VP have a transitive property if they have following characters: $A \xrightarrow{<L,R>} B$ and $B \xrightarrow{<L,R'>} C$, then $A \xrightarrow{<L,R''>} C$. Here, $A \xrightarrow{<L,R>} B$ means VP A and VP B have R relationship in level R. R, R' and R" can be Inherit and Extend relationships. L can be UI, Service, Workflow and Data.

2. Weakest link effect: VP has a weakest link effect if this VP is composed by other VPs. If $A \xrightarrow{<L,R>} B$ and $A \xrightarrow{<L,R>} C$, then type weakest type of B and C determines type of A. Here defines FVPFO ¡ FVATO ¡ FVPO. L can be UI, Service, Workflow and Data. R can only be Compose relationship. For example: A can be FVATO if B is FVATO and C is FVPO.

3. Type changes: VP changes type if this VP has following characters: $A \xrightarrow{<L,R>} B$. Here, R can be Extend and Compose relationship.

4. Override: VP and another VP have a override property if they have following character $A \xrightarrow{<L,R>} B$. Here, override means options in A override options in B if the options have same name but different values. R can be Restrict, Inherit, Extend and Compose. L can be UI, Service, Workflow and Data. For example, B has a option name color and value is blue. Then, A has the option name color but can have value {white} or {blue, white}.

### *4.3.4 VP Options and Rules Deduce Algorithm*

VPs and their relationships can be described by a directed acyclic graph (DAG). One example is shown in figure 4.12 on the next page. From figure 4.12 on the

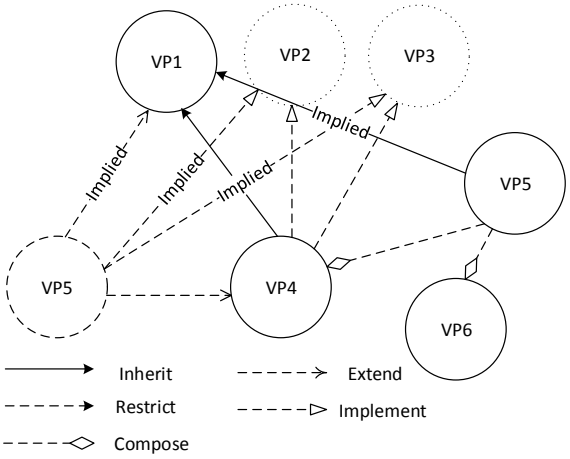



Figure 4.11: VP Properties

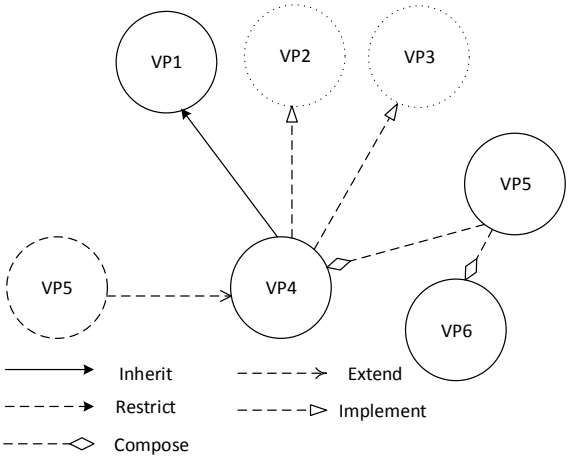



Figure 4.12: VP Relationship DAG Example

following page, one can see each node represents a VP and each edge represents a VP relationship. What is VP's options and rules can be deduced by Algorithm 1. In the Algorithm 1, the input is a DAG (G(V,E)) represents a VP relationship graph and the VP that need to look for options and rules.

From Algorithm 1, one can see followings:

1. The algorithm is implemented in a recursive way and it ends when the edge type is implement or restrict.

**Algorithm 1:** Deduce VP Options

**Algorithm** `algo(`*G(V, E), vp*`)`

```
1  Map < String, Set < Options >> vpOptions;
2  Map < String, Set < Rules >> vpRules;
3  proc(G(V, E), vp, vpOptions, vpRules);
4  return vpOptions and vpRules;
```

**Procedure** `proc(`*G(V,E), VP, Map < String, Set < Options >> vpOptions, Map < String, Set < Rules >> vpRules*`)`

```
1  vpOptions.get(vp.name).addAll(all vp's options);
2  vpRules.get(vp.name).addAll(all vp's rules);
3  foreach(Edge e : vp.edges) {
4  if(e.type == restrict || e.type == implement) return;
5  myproc(G(V,E), e.parent, vpOptions, vpRules);
6  }
7  return;
```

2. The complexity of the algorithm equals the number of VPs related to the input VP.

### 4.4 STA Customizations with VP models

Different STA Models have different way to implement customizations. To achieve STA customizations, all three VP models with five VP relationship models can be applied. In addition, three roles of SaaS application, infrastructure developers, tenant developers and sub-tenant developers are discussed.

### *4.4.1 Server-Customers STA Customization*

In Server-Customers STA model, the server is a tenant of a SaaS component, distributes and supports its components. One Server-Customers STA model example is shown in Figure 2.5. It has following characters:

1. Tenants develop, sell and distribute tenant components for sub-tenants, and sub-tenants share the same component instances.

2. Tenants do not share their data with their sub-tenants and have no accesses to sub-tenants data. For SOASaaS, this means that data components will be encrypted, thus invisible by infrastructure developers. Furthermore, sub-tenant developers will encrypt their data components, so that tenants or infra people will not be able to read.

3. Sub-tenants can customize the tenants components.

4. If a tenant upgrades its components, the changes will be propagated to its sub-tenants.

Server-Customers STA customization is shown in Table 4.1.

### *4.4.2 Software-Data STA*

In Software-Data STA model, both the tenant and its sub-tenants have their isolated SaaS instances but share the same code base. One software-Data STA model example is shown in Figure 2.6. It has following characters:

1. A tenant owns tenant components and data, and these are shared by its sub-tenants; furthermore sub-tenants can customize tenants components.

Table 4.1: Server-Customers STA Customization

| VP Type | SaaS Components | Tenant Components | Sub-tenant Components | Applied Relationships |
|---|---|---|---|---|
| Fixed variation points and fixed options | These are provided by infrastructure developers, but those options provided cannot be changed by tenant or sub-tenant developers. | 1. Tenant developers can choose those fixed options if the components are supplied by infra developers or other tenant developers (assuming the components can be shared).<br>2. Tenant developers can develop and upload their components with fixed variation and fixed options. But those uploaded components may be verified by infra developers to ensure system correctness. All components that can be used by sub-tenants do NOT touch the tenant data to ensure item (b) is satisfied. | As all variations points and options are fixed, sub-tenants can select options specified by infra or tenant developers only. | restrict and implement |
| Fixed variations but allow tenant options | Components and variations points may be provided by infrastructure developers, but tenant and sub-tenant developers can upload their software as options. Infra developers need to verify those options to ensure correctness, particularly related to item (b) above. | 1. Tenant developers can upload their components , but infrastructure people need to check if the tenant components follow the rules of variation points (such as input and output compatibility), specifically related to item (b) above.<br>2. Those options supplied by sub-tenants need to be verified by tenant developers first, and infra developers then. In addition, item (b) above should be verified. | As options may be provided by tenants or sub-tenants: in this case, both infra and tenant developers need to verify those new components satisfy the variation point rules before the options can be accepted, as related to item (b) above. | restrict and implement |
| Flexible variation points and options | This is similar to the cell above, except now the infra developers need to verify that the variation points can be placed as not all the variations can be placed arbitrarily. | This is similar to the cell above, except now the infra and tenant developers need to verify that the variation points can be placed as not all the variations can be placed arbitrarily. | This is similar to the cell above, except now the infra and tenant developers need to verify that the variation points can be placed as not all the variations can be placed arbitrarily. | restrict and implement |

2. Tenants develop and own the tenant components, and share the same component instance with their sub-tenants.

3. Sub-tenants can access the tenants sharing data and their own data.

4. Tenants can customize SaaS components, and sub-tenants can customize SaaS and tenant components.

5. Tenants can define the scope that sub-tenants can customize.

6. If a tenant upgrades its application, the changes will be propagated to sub-tenants.

Software-Data STA customization is shown shown in Table 4.2.

### *4.4.3 Master-Slaves STA*

In Master-Slaves STA, both the tenant and its sub-tenants have their isolated SaaS instances but share the same code base. One example is shown in Figure 2.7. It has following characters:

1. Both the tenant and its sub-tenants have their isolated SaaS instances but share the same code base.

2. The tenants share components with sub-tenants but they have different component instances.

3. Tenants have access to the sub-tenants data and sub-tenants can not have access to the tenant data.

4. Sub-tenants can customize the shared tenant components.

5. If a tenant upgrades its components, the changes will be propagated to sub-tenants as they do not share the same component instances.

Table 4.2: Software-Data STA Customization

| **VP Type** | **SaaS components** | **Tenant Components** | **Sub-tenant components** | **Applied Relationships** |
|---|---|---|---|---|
| Fixed variation points and fixed options. | These are provided by infrastructure developers, but those options provided can be changed by tenant or sub-tenant developers. | This cell is same to corresponding cell in Server-Customers model except all components that can be used by sub-tenants can touch the tenant data. | This cell is same to corresponding cell in Server-Customers model. | inherit |
| Fixed variations but allow tenant options. | This cell is same to corresponding cell in Server-Customers model. | This cell is same to corresponding cell in Server-Customers model. | This cell is same to corresponding cell in Server-Customers model. | inherit |
| Flexible variation points and options | This cell is same to corresponding cell in Server-Customers model. | This cell is same to corresponding cell in Server-Customers model except tenant developers can customize or replace options or rules developed by infrastructure developers or other tenant developers. | This cell is same to corresponding cell in Server-Customers model except sub-tenant developers can select, replace and customize options and rules. | inherit and extend |

Table 4.3: Master-Slave STA Customization

| VP Type | SaaS components | Tenant Components | Sub-tenant components | Applied Relationships |
|---|---|---|---|---|
| Fixed variation points and fixed options | These are provided by infrastructure developers, but those options provided can be changed by tenant or sub-tenant developers. | This cell is similar to the corresponding cell in Software-Data STA model except it is tenant developers that need to verify those options to ensure correctness. | This cell is similar to the corresponding cell in Software-Data STA model except it is sub-tenant developers that need to verify those options to ensure correctness. | tenant VPs inherit sub-tenant VPs |
| Fixed variations but allow tenant options | Components and variations points may be provided by infrastructure developers, but tenant and sub-tenant developers can upload their software as options. Infra developers need to verify those options to ensure correctness. | This cell is similar to the corresponding cell in Software-Data STA model except it is tenant developers that need to verify those options to ensure correctness. | This cell is similar to the corresponding cell in Software-Data STA model except it is sub-tenant developers that need to verify those options to ensure correctness. | tenant VPs inherit sub-tenant VPs |
| Flexible variation points and options | This is similar to the cell above, except now the infra developers need to verify that the variation points can be placed as not all the variations can be placed arbitrarily. | This is similar to the cell above, except now the infra and tenant developers need to verify that the variation points can be placed as not all the variations can be placed arbitrarily. In addition, tenant developers can fix the flexible variation points and change it to a fixed variations but allow tenant options. | This is similar to the cell above, except now the infra and tenant developers need to verify that the variation points can be placed as not all the variations can be placed arbitrarily. In addition, sub-tenant developers can fix the flexible variation points and change it to a fixed variations but allow tenant options or fixed variations and fixed tenant options. | tenant VPs inherit or extend sub-tenant VPs |

Master-Slave STA customization is shown in Table 4.3.

### *4.4.4 Slave-Masters STA*

Slave-Masters STA model is similar to Master-Slaves STA model except sub-tenants can access tenant's data. One example is shown in Figure 2.8. It has following characters:

1. Tenants share components with their sub-tenants, but they have different component instances.

Table 4.4: Slave-Master STA Customization

| VP Type | SaaS components | Tenant Components | Sub-tenant components | Applied Relationships |
|---|---|---|---|---|
| Fixed variation points and fixed options | This cell is similar to the corresponding cell in Slave-Masters STA model. | This cell is similar to the corresponding cell in Slave-Masters STA model except sub-tenant developers can access tenants' data. | This cell is similar to the corresponding cell in Slave-Masters STA model except sub-tenant developers can access tenants' data. | inherit and extend |
| Fixed variations but allow tenant options | This cell is similar to the corresponding cell in Slave-Masters STA model. | This cell is similar to the corresponding cell in Slave-Masters STA model except sub-tenant developers can access tenants' data. | This cell is similar to the corresponding cell in Slave-Masters STA model except sub-tenant developers can access tenants' data. | inherit and extend |
| Flexible variation points and options | This cell is similar to the corresponding cell in Slave-Masters STA model. | This cell is similar to the corresponding cell in Slave-Masters STA model except sub-tenant developers can access tenants' data. | This cell is similar to the corresponding cell in Slave-Masters STA model except sub-tenant developers can access tenants' data. | inherit and extend |

2. Tenants do not have access to the sub-tenants data but sub-tenants can have access to the tenants sharing data.

3. Sub-tenants can customize the shared tenant components.

4. If a tenant upgrades its components, the changes will be propagated to sub-tenants as they do not share the same instances.

Slave-Master STA customization is shown in Table 4.4.

### *4.4.5 Partner-Partner STA*

In Partner-Partner STA, both the tenant and its sub-tenants have their isolated SaaS instances and data. One example is shown in 2.9. It has following characters:

1. Tenants share components and data with sub-tenants.

2. Sub-tenants share components and data with their tenants.

3. Tenants share some data with their sub-tenants.

Table 4.5: Partner-Partner STA Customization

| VP Type | SaaS components | Tenant Components | Sub-tenant components | Applied Relationships |
|---|---|---|---|---|
| Fixed variation points and fixed options | These are provided by infrastructure developers, but those options provided can be changed by tenant or sub-tenant developers. | This cell is similar to the corresponding cell in Slave-Masters model except tenant can access sub-tenant data. | This cell is similar to the corresponding cell in Slave-Masters model except sub-tenant can access tenant data. | restrict, inherit and compose |
| Fixed variations but allow tenant options | This cell is similar to the corresponding cell in Slave-Masters model. | This cell is similar to the corresponding cell in Slave-Masters model except tenant can access sub-tenant data. | This cell is similar to the corresponding cell in Slave-Masters model except sub-tenants can access tenants' data. | restrict, inherit and compose |
| Flexible variation points and options | This is similar to the cell above, except now the infra developers need to verify that the variation points can be placed as not all the variations can be placed arbitrarily. | This cell is similar to the corresponding cell in Slave-Masters model except tenants can access sub-tenants' data. | This cell is similar to the corresponding cell in Slave-Masters model except sub-tenants can access tenants' data. | restrict, inherit, extend, compose |

4. Sub-tenants share some data with their tenants.

5. Sub-tenants share some data with their tenants.

6. If a tenant upgrades their components, the changes will be propagated to their sub-tenants as they do not share the same component instances.

7. If a sub-tenant upgrades its components, the changes will be propagated to their tenant as they do not share the same component instance.

Partner-Partner STA customization is shown shown in Table 4.5.

# Chapter 5

# STA IMPLEMENTATION STRATEGIES

There are two ways to implement STA: by traditional approaches and template with VPs.

## 5.1 By Traditional Approaches

By using traditional approaches, STA can be implemented by following ways:

1. Integration with DB: This supports customization (by coding), MTA (by de-normalization), scalability on top of a modified DB with two-levels of scalability mechanisms. One SaaS DB approach is proposed by Force.com [4].

2. SOASaaS: This is an SOA approach to STA, which provides model-driven code generation, and map code and data into different PaaS systems with different scalability mechanisms. One SOA SaaS example is shown in [95].

3. PaaS-based approach: This uses the existing PaaS such as GAE, EC2, and Azure as the infrastructure to develop STA.

4. OO approach: This uses an object-oriented approach for tenant application development and configuration. One SaaS OO approach example is proposed by Workday [129].

A STA architecture overview is shown in Figure 5.1. From Figure 5.1, one can see STA architecture needs to execute the following three tasks:

1. Route tenant requests: STA needs to distribute tenant application and data requests to right servers. Servers can be replicated and migrated for load balance.

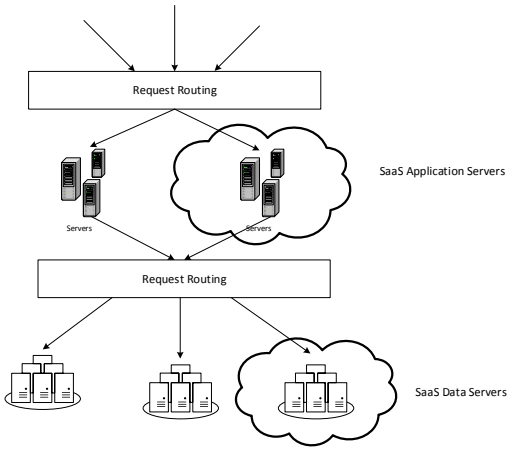



Figure 5.1: STA Architecture Overview

2. Add SaaS application servers: STA needs a way to add application servers or cloud application servers without interrupting existing applications and servers. In addition, SaaS servers are designed to be stateless that applications can be easily replicated.

3. Add SaaS data servers: STA can dynamically add data servers without interrupting other data servers and application servers. Data can be easily replicated when they need to be scaled.

## 5.2 By Template with VPs

With the help of template and VPs, the process of building SaaS application becomes building or discovering application templates with VPs and customize VPs. Application template with VPs are implemented by the way of SOA [23] to make use of its many good features.

### *5.2.1 Service Management and Composition*

Service specifications (SS) describe VP interfaces, and they may include the services' input, output, specification, test cases, and use scenarios. Services can be

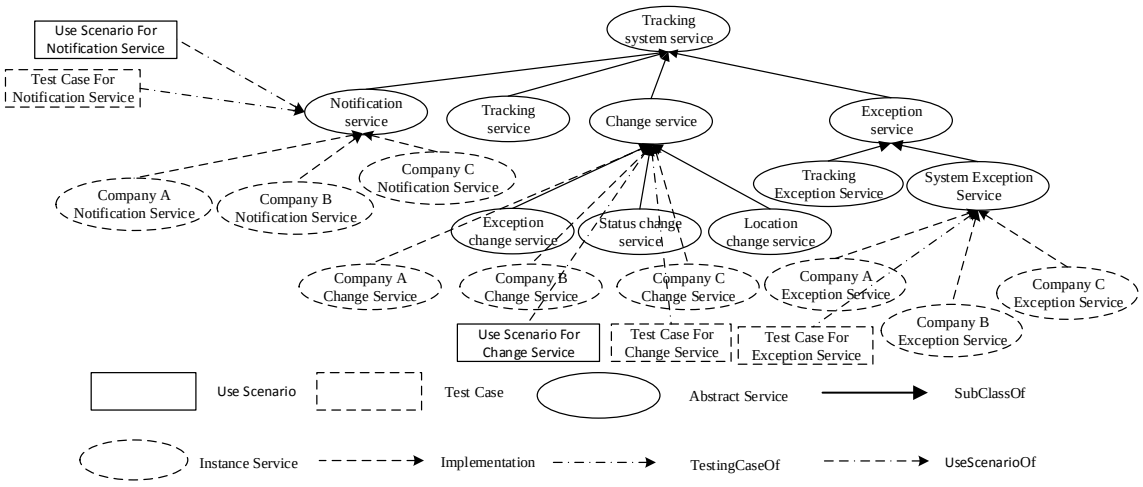



Figure 5.2: Domain Ontology Example

organized by a domain ontology, which maintains relationships among service interfaces and service implementations. The relationship between service specification and service instances is one-to-many. Service Instances (SI) implement an service specification.

Domain ontology: Domain ontology expresses domain information and represents entities (as nodes), relationships and constraints. It can organize and manage service interfaces. One example is illustrated as figure 5.2.

1. Nodes: A node is a unique entity that represent an service interface in domain ontology. Every node has zero or more corresponding implementations that have been verified. In addition, each nodes implementation must have the same input and output so that they can be dynamically replaced by each other.

2. Relationships: It is a connection between two nodes in the domain ontology.

3. Service Workflow or Application Template: they are composed by service interfaces represented by a domain ontology and control structures as illustrated as figure 5.3 on the following page.

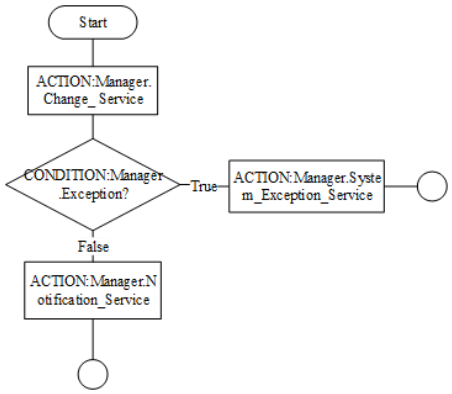



Figure 5.3: Workflow Example

### *5.2.2 SaaS Application Templates*

STA can use templates for static, dynamic, or hybrid composition approaches. Templates can be populated by service specification or service instances that come from service nodes of domain ontology. To be useful for actual execution, a template must be populated only by service instances. If a template is populated only by service instances, the composition is static. That is, it can be run without any further interaction by STA. If a template consists of only service specifications, STA needs to create an executable template by replacing service specifications with service specifications. This is a dynamic composition. Finally, if a template is populated by a mixture of implementation and interface services, a hybrid composition approach is taken where only service instances are replaced with service instances.

An application template of dynamic composition represented by domain ontology and control structures is shown in figure 5.4 on the next page. One benefit of this template-based approach is that service and test scripts or cases share the same template and it can be automatically completed with dependency support. STA can dynamically replace service specification with service instances. The service specifications that are used to compose the template can also be a template, as long as

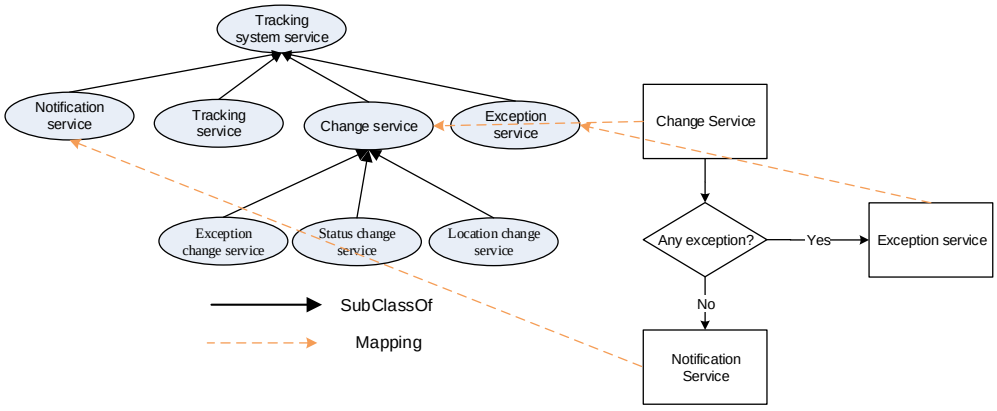



Figure 5.4: Service Template Example Mapping

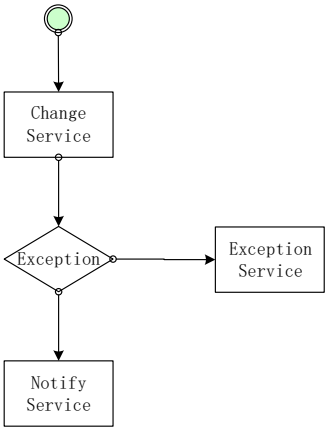



Figure 5.5: Template Example in PSML-S

this template has been registered and passed all tests. As a template is composed by service specifications, tenants or sub-tenants with limited programming knowledge can revise it. They can change the control flow in the template by adding or removing control structures or replace service instances with their implementations. In STA, templates can be described by PSML-S [99]. PSML-S provides many control constructs to help tenants to revise templates easily such as condition, parallel and sequence. Tenants or sub-tenants can revise the templates by drag and drop. One PSML-S template is shown in figure 5.5.

```
[WebMethod]
public void bind(Object IntS, Object ImpS)
{
   Bind binder = new Bind();
   binder.bind(Ints).to(ImpS);
}
```

Figure 5.6: Service Binding by Programming Example

### *5.2.3 Extensions to Allow Users Designate Specific Services*

In [111], service instances' selection mainly depends on dependency information among service specification and users' service selection history. Both of them do not allow user to designate specific service instance programmatically. This paper extends service selection mechanism using Service Injection (SI) that allows users to compose workflow or application templates by service specifications and inject service instance later. Developers have two options to specify SI:

1. By Configuration: Developers can fill all service selection information such as service name and method names. Inputs and other related information can be added into the the configuraion file groundProfile.xml. This approach is based on the Spring tool. It is convenient for users who want to do configuration rather than programming. figure 5.8 on the next page gives an example of the groundProfile.xml. In the upper part of groundProfile, a service interface NotificationService is mapped to the method notification of Company A Notification Service. Its input parameter is username whose value is Tommy. In the bottom part of this file, a workflow ChangeNotificationWay is defined. It uses the reference notificationServiceByCompanyA defined in the upper part. If developers want to use different service implementations, they can add another service interface mapping like Company A Notification Service and change the reference. By this way, developers can select service by injecting service into

```
CompanyANotification ImpS = new CompanyANotification();
GroudService myBinder = new GroudService();
myBinder.bind(NotificationService,ImpS.notification("Manager"));
```

Figure 5.7: Ground Service Binding Example

```
<?xml version="1.0" encoding="UTF-8"?>
<sevices xmlns="ASU service injection namespace"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<service id="notificationServiceByCompanyA" interface="NotificationService"
implementation="com.vaannila.SpringQuizMaster">
    <property serviceName="Company A Notification Service">
        <methodName>notification</methodName>
        <input>
            <org1 name="userName">Tommy<\org1>
        </input>
    </property>
</service>
<service id="changeNotificationWayWorkflow" interface="ChangeNotificationWay">
    <property serviceName="notificationService">
        <ref local="notificationServiceByCompanyA"/>
    </property>
</service>
</services>
```

Figure 5.8: A GroundProfile Example

a configuration file. The composed workflows or application templates can be reused by others.

2. By Programming: Developers can fill service selection information by calling GroundService.bind() method shown in figure 5.6 on the preceding page.

   From figure 5.6 on the previous page, one can see that, it mainly binds service interface to the service implementation. Developers can use GroundService like normal web service and fill in service specification and service instance shown in figure 5.7.

   In this way, application developers can designate their service implementations. This is similar to the class injection mechanism used in Google Guice [113].


Figure 5.9: Generated Source Code of Workflow

### *5.2.4 Code Generation Support*

In [111], developers compose workflows by dragging and dropping services from domain ontology, which is implemented by a tool [99]. One workflow example is illustrated as shown in Figure 3 and the generated source code is presented as figure 5.9. How to generate source codes of SI is similar with Spring [49] and Guice [113]. And it can be easily implanted by PSML-S.

### *5.2.5 Testing Workflows Of Service Composition*

Both the number of cloud services available and the size of data that cloud services need to handle are often large. Testing workflows composed of those cloud services is a challenge. This paper follows the service group testing [112, 20] to test workflows.

### *5.2.6 Oracle Generation of Composite Services*

A test case is a pair (test input, expected output), but often the expected output is difficult to obtain. As users compose services by using SInts and each SInts can be implemented by different providers (SImps), there is a large number of combinations for the same workflow if the cloud chooses different SImps. Oracle generation mechanisms in [112, 20] can be used to determine the expected output. Oracle generation uses a voting mechanism to establish an oracle with a confidence level, and if the confidence level is high enough, the corresponding oracle can be used to determine

the pass/fail of subsequent tests, i.e., a test case is established (input, established oracle with a confidence level).

### *5.2.7 Unit Testing*

Once a test case is formed with an established oracle, it can be used to test new service implementation. Test cases can also be ranked to help the cloud and users to select the most potent test cases to run first and often. After an oracle for a test input has been set up for a workflow, integration testing can be used to test each service by changing one service implementation at a time. If testing results are consistent, the new service is considered as correct with a confidence level with respect to the test case. When a sufficient large number of test cases pass the test, the new service implementation is considered as validated with another confidence level, otherwise the new implementation will be rejected.

### *5.2.8 Integration Testing*

Once the workflow is completed, one can apply its use scenarios [98] as test scripts to test the workflow. A user scenario of a workflow or a service is essentially an application that uses the workflow or the service respectfully. One can say that the workflow in figure 5.9 on the preceding page is a use scenario of all the participating services such as notification_service. Use scenarios for a service can be collected and served as a part of the service specification, and they can be used for service composition. A use scenario for a service may involve other services, and thus this use scenario provides a relationship between these two services, the relationship indicates these two services are linked to each other. During service composition, once a service is selected, the linked service becomes a candidate for composition. Furthermore, the use scenarios for a workflow can be collected and used later for service composition

or as the basis for test scripts.

### *5.2.9 Continuous Testing*

Continuous testing can be a part of the TDD (Test-Driven Development) process. Continuous testing is a testing process that is being applied during the development and execution stages. In traditional continuous testing, testing mostly in the form of regression testing is applied during the entire development time 24 hours a day [82]. In clouds, as new applications may be composed from existing services, continuous testing can be applied before and after application and service composition, and even during execution as a part of the service monitoring and/or policy enforcement processes. Continuous testing can be used to test SaaS applications [100] by embedding built-in test case generation with the metadata database associated with a tenant in the SaaS. In this case, test cases can be selected to test the SaaS applications continuously. If a test script detects a failure, the ranking of the test script with its associated test case will be increased so that it will be used early and more often in continuous testing. The cloud platform can run those test scripts continuously selecting most potent test case with dynamic ranking of test cases.

### *5.2.10 Metadata-Driven Test Input Generation*

For inputs and outputs of service interfaces, one can use metadata to define test inputs [100]. For example, if the length of user ID for a website must be 64 bits, the simple test inputs can be generated by randomizing the 64 bits. One can generate a collection of user IDs of 64 bits, another collection with 128 bits or any other bits.

### 5.2.11 *Execute Testing Processing by Service-Level MapReduce Way*

As the number of SImp can be large on a cloud, testing needs to be effective and efficient. So, group testing can be applied using service-level MapReduce [110] to run tests in parallel. The idea is to use different combinations of service implementations for the same workflow in the map step. The input of the workflow is either produced by developers or generated from the metadata. The majority of immediate results from the map step is reduced to generate an oracle in the reduce step by the voting mechanism [112]. It is shown in figure 5.10 on the next page. From figure 5.10 on the following page, one can see followings:

1. Each service combination for the workflow is running on different worker machine of the cloud with same inputData service.

2. All service combinations can be run in parallel.

3. Cache service can be used to shuffle the immediate results of map process and dispatch them to different reduce services to get final result. This process is controlled by service-level MapReduce [110].

4. If output data produced are consistent with limited deviation [82], i.e., a majority can be established, an oracle can be established with a confidence level. Otherwise, no such oracle can be established, and another test input need to be run to establish its own oracle.

The testing excution process is described as followings:

1. Users submit the composed service to the cloud;

2. Cloud management service (CMS) calculates how many workers needed based on configurations;

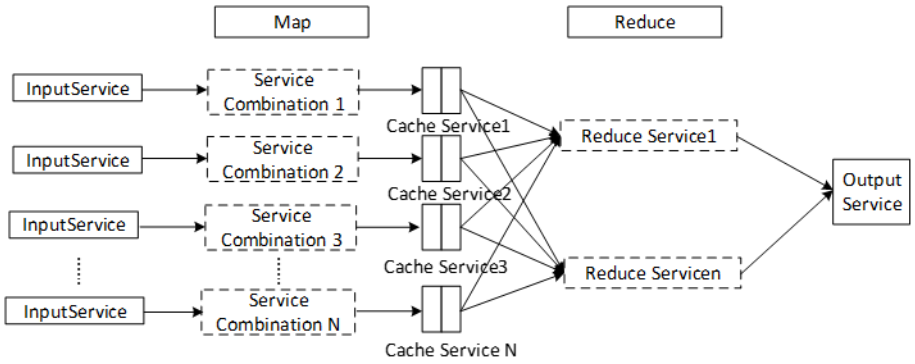



Figure 5.10: Oracle Generation Service Level MapReduce process

3. CMS finds enough available workers to run services.

4. CMS discovers service combinations for the composed service and dispatches them to the workers;

5. CMS calls the input service or provides test input from developers to start service-level MapReduce;

6. In the map step, service combinations are executed; results are sent to cache services for shuffling;

7. In the reduce step, the voting is done to establish an oracle;

8. If an oracle is found, it is sent back to users as an validated oracle. If not, the composed service fails to establish an oracle with this test input.

# Chapter 6

# TENANT-CENTRIC STA

In [109], MTA has been extended to allow a tenant application to have its own sub-tenants, where the tenant application acts like a SaaS infrastructure.

## 6.1 Life Cycles of Tenant-Centric Application Development

The purpose of tenant-centric application development is to help tenants find experts to develop components with domain knowledge requirements and facilitate components created and reused. Normally, there are six steps in general cases shown in figure 6.1: requirements, modeling, implementation, assembling, deployment and management

1. Requirements: they are the processes that tenants propose their business objectives. There are two types of requirements:

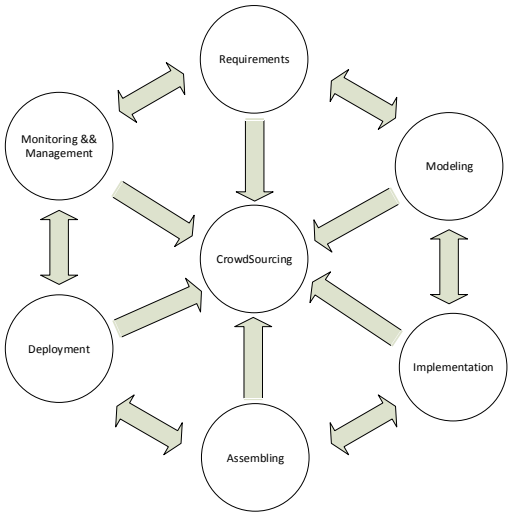



Figure 6.1: Application Development Life Cycle

(a) Feature requirements: they are all required features that tenants want to implement.

(b) Formal requirements: they are formal technique requirements that developers can implement.

2. Modeling: it is the process that translates tenant business requirements into a specification of business process and constraints. It may include following sub-steps:

   (a) Validating feature requirements: It is the process that verifies if feature requirements cover all business requirements.

   (b) Discovering current components: It is the process that discovers existing components to implement feature requirements.

   (c) Modeling feature and performance requirements: It is the process that simulate the feature and performance requirement. Any traditional simulation techniques can be applied.

3. Implementation: it is the process that implements all the features, functions, services and their testing cases that modeling step proposes.

4. Assembling: it is the process that integrates all tenant applications, features, services and does integration testing.

5. Deployment: it is the process that creates hosting environments and deploys assembled applications to different servers.

6. Monitoring && Management: it is the process that monitor the service execution and maintains operational environments and policies expressed in the assembling.

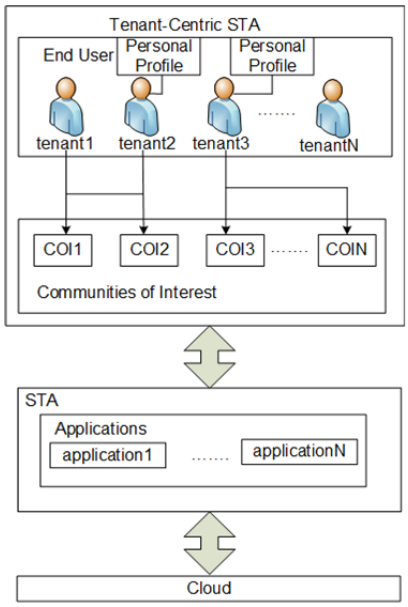



Figure 6.2: Community of Interests Example

7. Crowdsourcing: it is the process that tenant assigns tasks to tenants with domain knowledge. In other words, tenants do not need to develop applications by themselves but outsource some tasks to experts. Crowdsourcing is the center of all seven steps. All tasks in each step can be outsource to tenants in the same SaaS environment.

There are many ways that tenants can publish their requirements. One of the way is through community of interests (COIs) shown in Figure 6.2. COIs are composed by tenants in one or more domains that have common interests to exploit intelligence of crowd. Therefore, COIs are able to quickly finish domain related tasks with good quality. To get better quality, some tenants in the COI can implement the features while the others in the same COI can propose test cases. In addition, key words are used to describe COIs so STA can discover and recommend them when tenants have tasks.

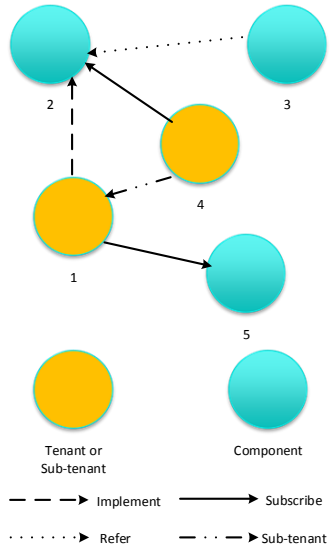



Figure 6.3: Static Ranking Example

## 6.2 Component and Tenant Rank

Normally, tenant proposes required technologies such as (Java and Cassandra) and let the STA system discover fit candidates. Machine learning technology such as KNN [2, 28] and Neural network [32, 16] can be applied to discover candidates. However, it is still difficult for a tenant to select candidate tenants if they are not ranked. It is also difficult for tenants to select components if components are not ranked. Therefore, this session propose a method to rank component and rank. There are two types of ranking models.

### *6.2.1 Static Ranking Model*

In STA, tenant, sub-tenant and their components form an relationship graph based on their implementation, subscription and reference relationships. One example is shown in figure 6.3.

In figure 6.3, one can see followings:

$$\begin{cases} R(r) = c \times \sum\limits_{s \in B_{r'}} \dfrac{R(s)}{N_s} \\ R(u) = \alpha \times \sum\limits_{v \in B_{u'}} \dfrac{R(v)}{N_v} + \beta \times \sum\limits_{w \in B_{u''}} \dfrac{R(w)}{N_w} + \gamma \times \sum\limits_{w \in B_{u'''}} \dfrac{R(x)}{N_x} \end{cases} \tag{6.1}$$

1. Tenant1 implements component2 and subscribes component5.

2. Tenant2 implements component5 and subscribes component2.

3. Component3 refers to component2. In this paper, reference can be translated as dependency, extending or other relationships existing between two components in STA.

By revising page rank algorithm [72], tenants, sub-tenants and components can get scores called static scores. Comparing to page rank model, this static ranking model has following characters:

1. There are two types of nodes in the relationship graph, tenants or sub-tenants and components while there is only page in page graph.

2. There are three types of links, implementation, subscription and reference.

To accommodate those characters, a simple revised page rank model is introduced in Equation (6.1).

Equation (6.1) can be described as following:

1. r is a component; u is a tenant or sub-tenant.

2. $B_r$ represents the sets of components that have reference relationships with component r.

3. $B_{u'}$ represents the sets of components that tenant or sub-tenant u has implementation relationships; $B_{u''}$ represents the sets of components that tenant or

$$\begin{cases} R'(r) = d \times (c \times \sum\limits_{s \in B_{r'}} \frac{R(s)}{N_s}) + (1-d) \times \frac{E_1}{k} \\ R'(u) = d \times (\alpha \times \sum\limits_{v \in B_{u'}} \frac{R(v)}{N_v} + \beta \times \sum\limits_{w \in B_{u''}} \frac{R(w)}{N_w} + \gamma \times \sum\limits_{w \in B_{u'''}} \frac{R(x)}{N_x}) + (1-d) \times \frac{E_2}{l} \end{cases} \tag{6.2}$$

sub-tenant u has subscription relationships or component u has reference relationship with; $B_{u''}$ represents the sets of sub-tenants or sub-sub-tenants that tenant or sub-tenant u has sub-tenant relationship.

4. $\alpha, \beta, \gamma$ and c are the weight factors to affect the importance of each types. For example, if $\alpha = 3$ and $\beta = 1$, the importance of tenant implementation is three times that of tenant subscription.

Considering components that have no relationship, this paper assumes those components have equally opportunity reference relationship with all other components in STA. For tenants and sub-tenants without sub-tenants or sub-sub-tenants, this paper assumes they have equally sub-tenant relationships with all other tenants or sub-tenants in STA. Therefore, Equation (6.1) can be revised to Equation (6.2).

In Equation (6.2), $E_1$ represents all components that have no reference relationships with other components and $E_2$ represents all tenants or sub-tenants have no sub-tenants and sub-sub-tenants. All elements of both $E_1$ and $E_2$ are ones. The parameter d is a factor that indicates components do not have reference relationships with other components or tenants and sub-tenants have no sub-tenants and sub-sub-tenants, which can be set between 0 and 1.

### *6.2.2 Dynamic Ranking Model*

There are two types of ranks: component and tenant ranks.

1. Component rank: there are two important factors, importance (I) and goodness (G), to describe a component.

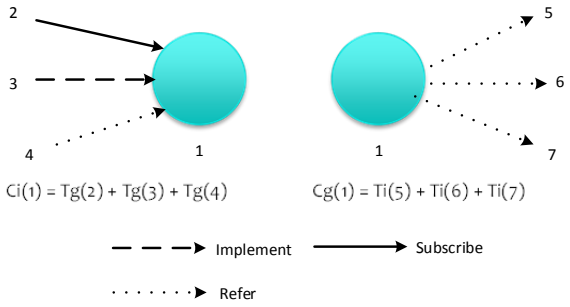



Figure 6.4: Component Rank Example

2. Tenant rank: same to component rank, importance (I) and goodness (G) are used to describe a tenant.

figure 6.4 shows how to calculate component's importance and goodness. From figure 6.4, one can see followings:

1. There are two tenants implement and subscribe a component 1; One component has reference relationship with the component 1; the component 1 has reference relationships with other three components.

2. Outdegree: number of components that a given component has reference relationship with, here it is used to measure the importance.

3. Indegree: number of tenants that implement or subscribe a given component and components that have reference relationship with the give component, used to measure the component's goodness.

figure 6.5 on the following page shows how to calculate tenant's importance and goodness. From figure 6.5 on the next page, one can see followings:

1. A tenant 1 has two sub-tenants; tenant 1 implements and subscribe one component; tenant 1 is sub-tenant of another tenant.

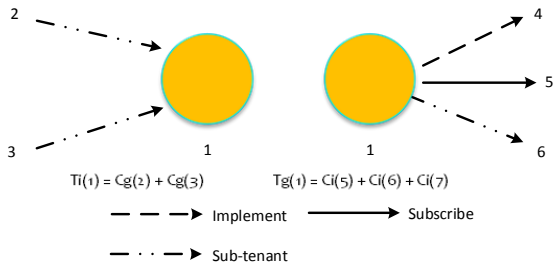



Figure 6.5: Tenant Rank Example

2. Indegree: number of sub-tenants to a give tenant, used to measure the tenant's importance.

3. Outdegree: number of components that a given tenant implements or subscribes and or tenants that the given tenant sub-tenant to, here it is used to measure the tenant's goodness.

Comparing the figure 6.4 on the preceding page and figure 6.5, one can see followings:

1. More good tenants implement the component, more importance the component has; more good tenants subscribe or components refer to the component, more goodness the component has.

2. More important tenants becomes sub-tenant of the given tenant, more goodness the tenant has; more good components the tenant subscribes and implements, more goodness the tenant has.

Formally calculating ranks are shown in Equation (6.3). And it can be describes as followings:

In upper part of Equation (6.3), a component's importance and goodness score is introduced.

$$\begin{cases} C_I = \sum_{i=1}^{n} C_{G_i} \; and \; C_G = \alpha * \sum_{i=1}^{n} T_{I_i} + \beta * \sum_{i=1}^{m} T'_{I_i} + \gamma * \sum_{i=1}^{k} C_{I_i} \\ T_I = \sum_{i=1}^{l} T_{G_i} and \; T_G = \alpha \times \sum_{i=1}^{j} C_{G_i} + \beta \times \sum_{i=1}^{o} C'_{G_i} + \gamma \times \sum_{i=1}^{p} T_{G_i} \end{cases} \tag{6.3}$$

1. A component's importance scores represented by $C_I$ are introduced by components that the component has reference relationships with represented by $C_{G_i}$.

2. A component's goodness scores represented by $C_G$ are introduced by following three parts:

   (a) $T_{I_i}$ are tenant's importance scores introduced by tenants implement the component.

   (b) $T'_{I_i}$ are tenant's importance scores introduced by tenants subscribe the component.

   (c) $C_{I_i}$ are component C's goodness scores introduced by components that the component has reference relationships with.

In lower part of Equation (6.3), a tenant's importance and goodness score is introduced.

1. A tenant's importance scores represented by $T_I$ are introduced by tenants or sub-tenants that are sub-tenants of a given tenant represented by $T_{G_i}$.

2. A tenant's goodness scores represented by $T_G$ are introduced by following three parts:

   (a) $C_{I_i}$ are component's importance scores introduced by the given tenant implements.

(b) $C'_{I_i}$ are component's importance scores introduced by the given tenant subscribes.

(c) $T_{I_i}$ are tenant's importance scores introduced by tenants that the given tenant has sub-tenant relationships with.

From the equation (6.3), one can see following objectives.

1. Initialization achieved by selecting set of components and tenants.

2. Importance and goodness of tenants and components can be set as a nonzero constant.

3. It is an iteration process to get importance and goodness of tenants and components. In other words, tenants and components get new values of importance and goodness each iteration.

4. The importance is computed from the current goodness weights, which are computed from the previous importance weights.

5. It can be proved that importance and goodness of tenant and application converge [53].

Base on the equation (6.3), Algorithm 2 is introduced. The Algorithm 2 performs a series of iterations and each consists of two basic steps:

1. Component Importance Update: Update each component's importance score to be equal to the sum of the goodness scores of components that the component has reference relationships with. That is, a component is given a high importance score by referring to components with high goodness scores.

**Algorithm 2:** Tenant and Application Rank

**Input:** n tenants T and m components C

**Output:** Goodness and importance of tenants and components

Initialize for all c ∈ C, $C_i = C_g = \frac{1}{n}$

e = 0.000001;

**while** $|C_{I_i} - C_{I_{i-1}}| + |C_{G_i} - C_{G_{i-1}}| + |T_{I_i} - T_{I_{i-1}}| + |T_{G_i} - T_{G_{i-1}}| > e$ **do**

  **foreach** *components in C* **do**

$$C_I = \sum_{i=1}^{n} T_{G_i}$$

$$C_G = \alpha * \sum_{i=1}^{n} T_{I_i} + \beta * \sum_{i=1}^{m} T'_{I_i} + \gamma * \sum_{i=1}^{k} C_{I_{C_i}}$$

$$C_I = C_I / c$$

`// normalize` $C_I$ `such that` $\sum_{i=1}^{m} (C_I/c)^2 = 1$

$$C_G = C_G / d$$

`// normalize` $C_G$ `such that` $\sum_{i=1}^{m} (C_G/d)^2 = 1$

  **foreach** *tenants in T* **do**

$$T_I = \sum_{i=1}^{l} T_{G_i}$$

$$T_G = \alpha \times \sum_{i=1}^{j} C_{G_i} + \beta \times \sum_{i=1}^{o} C'_{G_i} + \gamma \times \sum_{i=1}^{p} T_{G_i}$$

$$T_I = T_I / e$$

`// normalize` $T_I$ `such that` $\sum_{i=1}^{n} (T_I/e)^2 = 1$

$$T_G = T_G / f$$

`// normalize` $T_G$ `such that` $\sum_{i=1}^{n} (T_G/f)^2 = 1$

return all $A_I$, $A_G$, $T_I$ and $T_G$

2. Component Goodness Update: Update each component's goodness score to be equal to the sum of the importance scores of tenants that implement and subscribe it or components that have reference relationships with the given component. That is, a component is given a high goodness score by being implemented and subscribed by tenants with high importance scores or referred by components with high importance scores.

3. Tenant Importance Update: Update each tenant's importance score to be equal to the sum of the goodness scores of tenants that the given tenant has sub-tenant relationships. That is, a tenant is given a high goodness score by being sub-tenant to tenants with high goodness score.

4. Tenant Goodness Update: Update each tenant's goodness score to be equal to the sum of the goodness scores of components the tenant implements or subscribe and tenants that are sub-tenants to the tenant. That is, a tenant is given a high goodness score by implementing or subscribing many components with high importance scores or by tenants with high importance scores that are subtenants of the given tenant.

The Importance score and Goodness score for a component and a tenant is calculated with the Algorithm 2:

1. Start with each component having a Importance score and Goodness score of $\frac{1}{n}$.

2. For components, run the Component Importance Update; for tenants, run the Tenant Importance Update.

3. For components, run the Component Goodness Update; for tenants, run the Tenant Goodness Update.

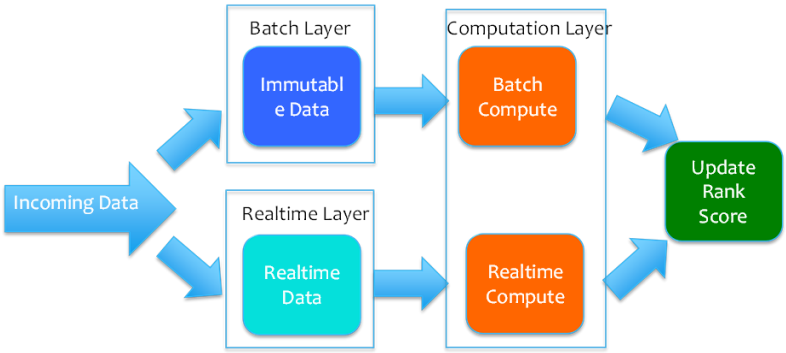



Figure 6.6: Rank Computation Architecture

4. Normalize the values by dividing each Importance score by square root of the sum of the squares of all Importance scores, and dividing each Goodness score by square root of the sum of the squares of all Goodness scores.

5. Repeat from the second step until there are small changes represented by $e$ for both tenant and component importance and goodness scores.

### *6.2.3 Rank Computation Architecture*

In the tenant and component rank algorithm, there are types of scores, static score and dynamic scores. However, the number of tenants and components can become huge. Therefore, it is difficult to calculate both static and dynamic score in realtime. As a result, a computation architecture is introduced to calculate both static and dynamic scores shown in figure 6.6.

From figure 6.6, one can see followings:

1. There are two layers to compute goodness and importance, batch layer and realtime layer. In this paper, batch layer means STA does the calculation after some period of time and does it in a batch way. Realtime layer means STA does the calculation when tenants and components need to change their rank scores.

2. Batch based calculation can compute large number of tenants or components and get very accurate results as it can take long time to finish calculation. In this layer, static ranking model is applied. After passing this layer, all tenants and components have static scores. As the number of tenants and components can become huge, famous big data framework such as hadoop [5] or spark [7] can be applied to accelerate computation.

3. Realtime based calculation can do calculation very fast but can only get proximate result. Only tenants and components that have relationships to tenants who implement or subscribe components and become sub-tenant to other tenants need to update their scores. Therefore, dynamic ranking model is applied. To apply dynamic ranking model, the first step is to retrieve the most relevant components and tenants by searching STA database and fetching tenants and components with changes. This set is called the root set and can be achieved by taking the top n tenants and components, where n can be huge. A base set is generated by augmenting the root set with all the tenants and components that subscribe, implement or refer to those components and tenants in root set. The tenants and components in the base set and all subscription, implementation and reference among those components and tenants form a subgraph. The subgraph can become large and complicate when the number of tenants and components is huge. Therefore, key words based search engine such as solr [81] and elastic search [35] can be introduced when searching tenant and component candidates or find augmenting information. In addition, graph databases such as neo4j [119] can be used to save subgraph information of the base set. In realtime environment, the time of computation must be short. Hence, famous realtime big data framework such as Apache Kafka [6] and storm [8] can be

$$S(i) = \begin{cases} \alpha \times R(i) + \beta \times (\gamma \times C_I + \xi \times C_G) & \text{if } i \text{ is a component} \\ \alpha \times R(i) + \beta \times (\gamma \times T_I + \xi \times T_G) & \text{if } i \text{ is a tenant} \end{cases} \tag{6.4}$$

integrated.

4. Batch based calculation can get static scores of all tenants and components. Realtime based calculation can get dynamic scores of tenants and components have changes. To integrate both static scores and dynamic scores, Equation (6.4) is applied. In Equation (6.4), $\alpha$, $\beta$, $\gamma$ and $\xi$ are weights to make static scores and dynamic scores comparable that can be adjusted.

## 6.3 Feature Implementation Selection Model

In STA, one component may have many features to be implemented. As one feature may be implemented by many tenants if Crowdsourcing is applied, it becomes import to choose fit tenants to implement features (X) of a component ($\tau$). This paper make following assumptions:

1. Feature is the smallest unit that cannot be further split.
2. Implementing feature X need time T and cost C.
3. A component $\tau*$ can be split into n features.
4. One tenant can implement more than one features for the same component.

One feature example is shown in figure 6.7 on the following page. In figure 6.7 on the next page, one can see followings:

1. One component can be split into n features presented by $X_1, X_2...X_n$.

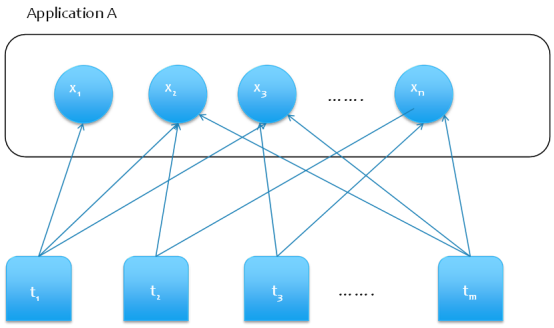



Figure 6.7: Feature Implementation Selection Model

2. One feature can be implemented by more than two tenants.

3. One tenant can implement many features at same time.

Formal feature implementation selection model can be shown in Equation 6.5. From Equation 6.5, one can see followings:

1. $\tau*$ represents an SaaS application.

2. The purpose of this equation is to find the minimal cost solution with time constraint.

3. $t_{i,j}$ represents if $tenant_i$ can implement the jth feature.

4. $x_j$ represents the jth feature.

5. $\sum_{i=1}^{n} t_{i,j} \times x_j = 1$ means only one tenant can implement the jth feature $x_j$.

6. $\sum_{j=1}^{n} t_{i,j} = m$ means n tenants can implement m features.

7. $\sum_{j=1}^{n} t_{i,j} \times t(x_j) < t$ means the total time that n tenants implement m features is less than the required time.

$$\begin{cases} \tau^* = argmin(\sum_{i=1}^{m}\sum_{j=1}^{n} t_{i,j} \times c(x_j)) \\ subject\ to: \\ \sum_{i=1}^{n} t_{i,j} \times x_j = 1, \sum_{i=1}^{m}\sum_{j=1}^{n} t_{i,j} = m\ and\ \sum_{i=1}^{m}\sum_{j=1}^{n} t_{i,j} \times t(x_j) < t \end{cases} \tag{6.5}$$

**Algorithm 3:** Algorithm for Feature Selection Problem

```
Input: m,n,c_1,...,c_n,t_1,...t_n,f_1,...f_m
Output: min cost, selected tenants
for i ← 1 to m do
    for j ← 1 to n do
        M[i,j] = 0 ;
OPT(i,j,ms,s) {
if i = 0 or j = 0 then
    if i = 0 then
        return 0 ;
    else
        return MaxNumber ;
else
    if t_i not implement f_i then
        M[i,j] = 0 ;
        return OPT(i,j-1,f,t-{ t_j }) ;
    else
        m_1 = OPT(i,j-1,f,t-{ t_j }) ;
        m_2 = c_j + OPT(i,j-1,f-{ f_i },t-{ t_j }) ;
        if m_1 < m_2 then
            M[i,j] = 0 ;
            return m_1 ;
        else
            M[i,j] = 1 ;
            return m_2 ;
}
```

To solve feature selection problem, Algorithm 3 is introduced.

The basic idea of Algorithm 3 is exhausting all possible solutions and find the best solution with minimal cost. Algorithm 3 can be described as followings:

- Algorithm 3 is a recursive algorithm and it explores every possible solutions.
- $\tau^*$ does not select $n_{th}$ tenant to implement the $m_{th}$ feature. So, $\tau^*$ will select the best tenant from $\{t_1, t_2, ..., t_{n-1}\}$ .
- $\tau^*$ select $n_{th}$ tenant for the $m_{th}$ feature. $\tau^*$ will choose tenants from $\{t_1, t_2, ..., t_{n-1}\}$ for $\{f_1,...,f_{m-1}\}$ .
- There is no features left. $\tau^*$ is optimized.

## 6.4 Rapid Application Building Process

This paper inherits those approaches proposed by Tsai in [111, 101, 108] to build application templates. When tenants or sub-tenants build application templates, the key words of those templates can be indexed by both elastic search [35] and solr [81]. By combining the relevance algorithm of elastic search and solr with components' rank discussed in 6.2.2, tenants or sub-tenants can quickly discover fit application templates. After selecting the application template, tenant or sub-tenants can customize or extend the application template to become an application or application template. The built application and application template can be published so that sub-tenants can subscribe and reuse. Therefore, the process of rapid application building become following two steps:

1. Tenants or sub-tenants discover fit application templates through key words based search engines.

Table 6.1: Connected Graph with Weights

| | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 |
| 4 | 2 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 |

2. Tenants or sub-tenants customize or extend the selected application templates. In addition, tenants or sub-tenants can publish customized applications or extended application templates so other sub-tenants can subscribe or reuse them.

## 6.5 Experiment

In this section, one experiment is to illustrate static and dynamic models. In static model, the relationships, implement, subscribe, reference and sub-tenant have different influence. In this experiment, implement is considered to have most influence and its weight is set to three. Sub-tenant is considered to have second influence and its weight is set to two. Both subscription and reference are considered to be equaled and their weights are set to one. Based on this assumption, figure 6.3 on page 63 can be translated to the connected graph with weights shown in Table 6.1.

Applying static model introduced by Equation (6.2), the result is shown in figure 6.8 on the next page.

From the static scores, one can see both tenant1 and tenant2 have higher static scores than those of components. By changing weights that changes the $\alpha$, $\beta$ and $\gamma$ in Equation (6.2), it will have different static scores.

Later, one tenant subscribes both component3 and component5. Applying dy-

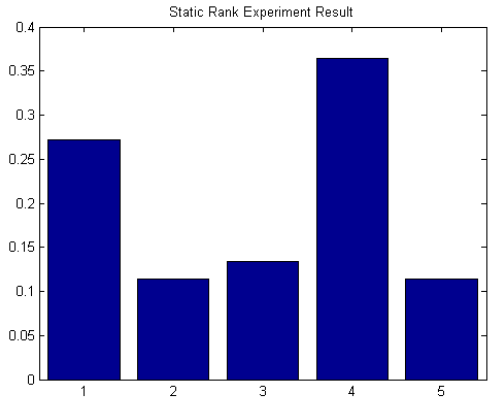
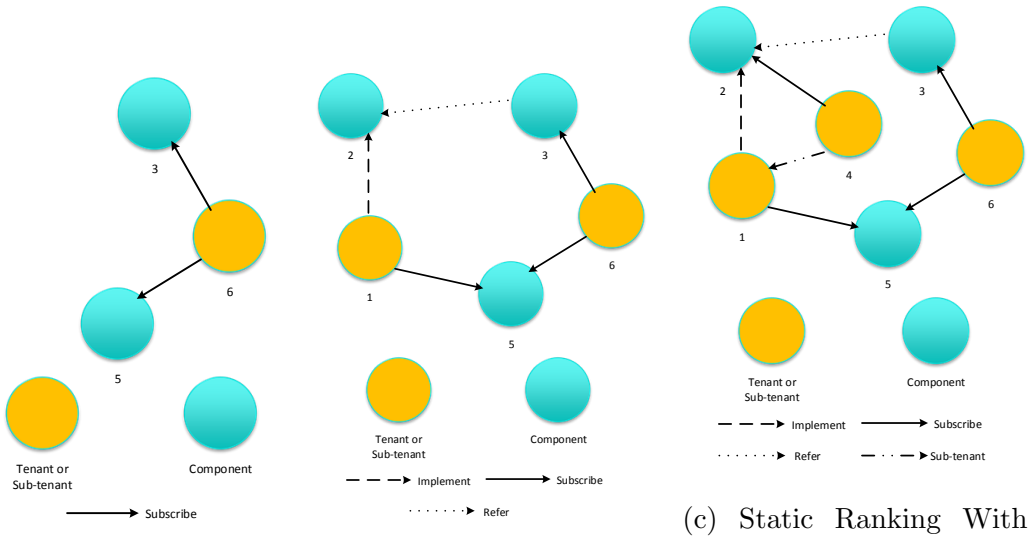



Figure 6.8: Result of Static Rank




(a) Root Set (b) Subgraph (c) Static Ranking With Update

Figure 6.9: Staic Rank With Dynamic Rank Update

Table 6.2: Subgraph with Weights

| | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| 1 | 0 | 3 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 1 | 0 | 1 | 0 |

namic model introduced by Equation (6.2), root set is shown in figure 6.9a on the preceding page. By augmenting relationships of component3 and component5, base set is discovered and it is composed of tenant1, tenant6, component2, component3 and component5. By adding their relationships, subgraph is shown in figure 6.9b on the previous page. To follow the same weights in static model, subgraph with weights is shown in Table 6.2. According to Algorithm 2, their importance and goodness scores are shown in figure 6.10 on the following page.

In figure 6.10 on the next page, tenant1 has highest importance score as tenant1 implements component1 and subscribes component5 where implement relationship has highes weight according to the assumption; component2 has highest goodness score as component2 is implemented by tenant1 with the highest importance score and referred by component3.

Combining static and dynamic scores, final scores are shown in figure 6.11a on the following page. Although static score, importance score and goodness score share same weights in this experiment, they can be different based on different requirements.

Finally, the final graph is formed by adding tenant6 and its subscriptions back to

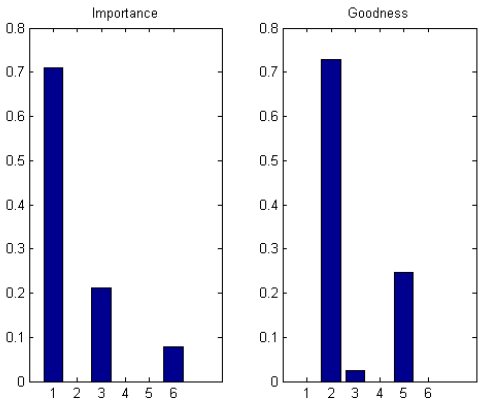
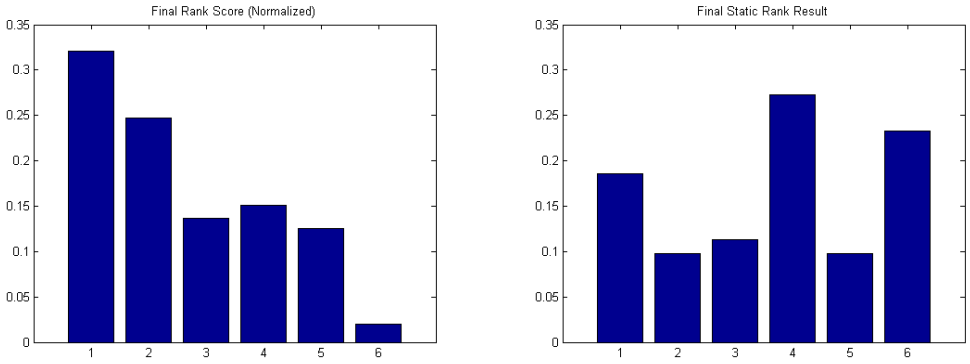



Figure 6.10: Result of Dynamic Rank




(a) Result of Final Rank

(b) Result of Final Static Rank

Figure 6.11: Final Score vs Final Static Score

the whole graph shown in fig:finalRankExample. And its corresponding final static scores are shown in figure 6.11b. Comparing final score with final static in figure 6.11, one can see followings:

1. Dynamic model boosts tenants or components with most relationships.
2. Static model boosts tenants with implementation relationships.
3. For other tenants or components, both dynamic model and static model have similar scores.

From the analysis of experiment result, both dynamic model and static model work as expect. Therefore, it proves rank computation architecture works well by applying static model to batch layer and dynamic model to realtime layer in figure 6.6 on page 72.

## 6.6 Conclusion

This paper provides a tenant centric STA to assist tenants to fast and easily build and publish customized components and data. To make use of public wisdom, Crowdsourcing is introduced to be the core of STA component development life cycle. In addition, static and dynamic models are proposed to rank tenants and components. Further, rank computation architecture is presented to handle the case when the number of tenants and components becomes huge. At last, an experiment is shown to demonstrate static model, dynamic model and rank computation architecture work as expected.

# Chapter 7

# DEPENDENCY-GUIDED SERVICE COMPOSITION

## 7.1 Introduction

Service-Oriented Architecture (SOA) [55, 64, 132, 29] imposes a composition-based approach for software application development by reusing existing software components available in the Internet which are wrapped into services and accessible through standard protocols. Service discovery and composition are central to service-based software development. Service discovery identifies a set of candidate services to meet the specified interface requirements. Composition is the process of creating an application by reusing and integrating the discovered services following the workflow requirements [57]. Various SOA protocols such as orchestration, chorography, and coordination have been designed for service composition, and many approaches have been proposed including model-based approaches [134, 46, 135], semantic-based approaches [38, 54, 19], or QoS-driven approaches [121, 58, 116]. In addition, SOA technology has been widely used by industry and research such as medical [131], healthcare [33], business process [79] and data mining [115, 24]. However, issues exist in current practices of service discovery and composition. First, it is hard to specify the needs of service. In the standard web services protocols, it often uses a static binding to pre-defined service implementations. It does not have a mechanism for the service users and applications builders to define their requirements. Second, it is difficult to match the service implementation to the application requirements. The services, published with XML-specified interface, may not have sufficient semantic information of their interface operations. Thus, even a direct XML-based syntac-

tic matching may not ensure the services' functions meet the users' expectations. In addition, the services discovered individually need to collaborate with others in an application context. To address these issues, Consumer-Centric SOA (CCSOA) and User-Centric SOA (UCSOA) framework [22, 102] have been proposed, on top of SOA, to allow the users to specify and publish their requirements with application templates. Taking the published requirements as domain knowledge, this paper extends the UCSOA framework to identify service dependencies from the domain model and the dependency information can be used to improve the intelligence and efficiency of service discovery and composition.

CCSOA allows the publishing of application, collaboration, and workflow templates that define the expected service functionalities and business processes. UCSOA further organizes the requirements into different groups called Community of Interests (COI) so that common solutions related to a certain domain can be reused for community members for rapid application development. In the UCSOA framework, the application builders select the application template first and modify the service and workflow to meet specific application requirements. The services implementations associated with the templates can be reused together as a package, together with the templates. In this paper, the ontology-based template specification is used to identify the potential dependencies between service functionalities. Relationships like hasInput, hasOutput, before, after, calledBy, hasCall are defined and presented, as complementary information to the domain model. The likelihood property is used to address uncertainties in dependency relationships. The algorithms are defined to calculate the dependencies between any two nodes. Such dependency information can be used to guide the process of service composition in the UCSOA framework. In spite of individual service matching and discovery, a group of dependent services can be identified together, evaluated and integrated into the composite workflow as

a whole.

Service-oriented techniques have been used in many critical systems including real-time mission-critical command-and-control systems. Specifically, the U.S. Department of Defense (DoD) has used service-oriented techniques to develop her network-centric operations since 2001. In fact, almost all major initiatives since then such as FCS (Future Combat Systems) [78], JBMC2 (Joint Battle Management Command and Control) [86], FORCEnet [68] are based on service-oriented computing. For those systems, service-oriented system engineering [90] is critical where systems need to be specified in a service-oriented manner, have an operational architecture that is compatible with service-oriented concepts, code can be deployed and executed in a service-oriented infrastructure, and system must be subject to service-oriented testing [14**?** , 108, 91]. Service-oriented system engineering is a new effort in system engineering [117, 15] where the system developed uses service-oriented specification techniques, design, languages, simulation, testing, verification and validation, and monitoring as well as other attributes such as reliability and security modeling and analysis.

This paper is organized as follows. Section II briefly reviews related SOA and service composition techniques. Section III analyzes the ontology relationships. Section IV introduces the definition and analysis algorithms of dependencies based on the ontology-specified domain model. Section V presents the process of service discovery and composition in the UCSOA framework. Section VI presents a case study to illustrate the proposed composition approach. Section VII concludes this paper.

## 7.2 Related Work

Current standard for describing Web services, WSDL, does not provide semantic information. WSDL 2.0, defines a set of extension attributes for the WSDL and XML

Schema definition language that allows description of additional semantics of WSDL components. As the Internet does not guarantee performance, Web services do not guarantee performance as they depend on the Internet to transfer messages between services

Dependency information is often used for compiler optimization and change management [75]. This paper further extends dependency with likelihood information to assess the weight of dependency relationships. The information is useful in identifying those items that are most likely to be selected for composition.

Service composition has been a difficult task. Dustdar [34] classified composition strategies into five categories: 1) static and dynamic composition strategies, 2) model-driven service composition, 3) business rule driven service composition, 4) declarative composition; and 5) automated and manual service composition. Static and dynamic composition concerns the time when services are composed. Static composition occurs at design time. Services are chosen, combined together, and finally compiled and deployed. Sun [84] defines Microsoft Biztalk and Bea WebLogic as examples of static composition engines and Stanford's Sword and HP's eFlow as examples of dynamic service composition. In static composition, it is difficult to replace services with equivalent new services. Dynamic composition was introduced to allow service composition to replace services dynamically. However, dynamic composition is difficult and one issue is identification of appropriate services at runtime.

Orriens [70] introduced model-driven dynamic service composition where UML is used to provide a high-level of abstraction that can be directly mapped to other standards, such as BPEL4WS. They use OCL (Object Constraint Language) to express business rules and describe the process flow. Gronmo [42] proposed a model-driven semantic web service composition. They use OWL-S and WSML as semantic web service description languages, and their method guides developers to compose ser-

vices through four phases, starting with the initial modeling, and ending with a new composite service that can be deployed and published. Aldin [1] proposes a survey to discuss the existing literature on the problem of business processing modeling reusability.

Ontology is often used for knowledge representation, sharing, classification, reasoning, and interoperability. [92] presents an ontology-based dynamic process collaboration and proposes a service collaboration ontology to exchange meaningful information between collaboration parties and perform collaboration workflow matching. Oh [69] proposes a novel metrics to measure ontology modularity. Ontology-based service composition is introduced in [87, 37]. Tosic [88] discussed the need for requirements for ontology, and provided ontology systems for the management of services and for Quality-of-Service (QoS) metrics. Process mining relates to the extraction of information is an important research discipline. Ingvaldsen [45] presents a framework to evaluate different aspects of enterprise process flows and address practical challenges of state-of-the-art industrial process mining.

This paper uses ontology systems to express domain information and ontology systems cross reference each other with dependency relationships. Ernestas [114] proposed a method of transforming ontology representation from OWL to relational databases and algorithms for transformation of domain ontology to relational databases. Bianchini [17] described an ontology design approach defined in the framework of the VISPO (Virtual-district Internet-based Service Platform) project to support knowledge sharing and service composition in virtual districts. Kim [51] presented a task dependency approach for Web service composition driven by business rules statically.

Current SOA composition emphasizes publishing and discovering services, and most of support is for service providers. UCSOA and CCSOA [22, 102] that run on top of SOA also support service consumers. These SOA allow various items such as

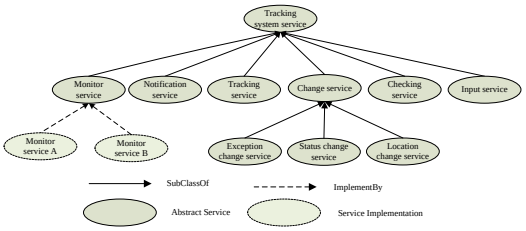



Figure 7.1: Shipping Domain Service Ontology

user requirements to be published so that service providers can discover and supply the needed services or workflows. UCSOA also allows end users to compose applications in a community. In this way, a non-technical person can compose applications easily like they use mashup. The proposed process is also useful for requirement analysis. An application engineer, who is developing the specification of a certain application, can discover similar application templates, workflows, and services. These provide significant guidance to all the needed features including services and workflows needed in the current project. The engineer may realize that other similar projects have features that were initially missing in the project requirements, and these missing requirements can be added to the current project requirements. Furthermore, those similar templates, workflows, or services identified may be reused or modified for the current application.

## 7.3 Ontology Relationships

Domain ontology is often used to express domain information and it represents entities, relationships and constraints. One kind of relationships is dependency relationship. A service ontology example is illustrated in figure 7.1.

### 7.3.1 Relationships in Ontology

Nodes in a ontology system may have some relationships with other nodes. At same time, nodes in different ontology systems may also be related to each other, i.e., an application node in an application ontology may use services in a service ontology. The following relationships can be in ontology systems:

- hasInput(A,B): This shows the input relationship between services, specifically A's output is the input of another service B.
- hasOutput(A,B): This shows output relationship between services, and A's output is the input of B, and this is the inverse relationship to hasInput.
- before(A,B): This shows service A must execute before service B.
- after(A,B): This shows service A must execute after service B. It is the inverse relation of before.
- calledBy(A,B): This shows service A is called by service B.
- hasCall(A,B): This shows service A call service B. It is the inverse relation of calledBy.
- mutualExclusion(A,B): This shows services A and B cannot be executed concurrently in one application.
- Concurrent(A,B): This shows services A and B can be executed concurrently. Other relationships are also possible.

### 7.3.2 Relationships Representation

This relationship can be expressed as following formal notation: $A \xrightarrow{<N,\Re,L>} B$.

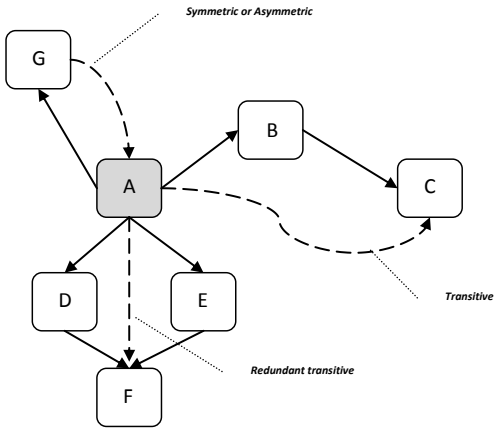



Figure 7.2: Property Illustration

- A , B represent service, workflow, application template or collaboration template, that have dependency relationship.
- N represents domain, that can be a service, workflow, application template or collaboration template and their subclasses
- $\Re$ represents one of relationships.
- L represents what is the likelihood that A has relationship $\Re$ with B at given domain N .
- $A \xrightarrow{<N,\Re,L>} B$ indicates that A and B have dependencies on each other in the domain .

## 7.4 Dependency Analysis

Three types of operation dependencies can be analyzed: Input Dependency (ID), Input/Output Dependency (IOD), and Output Dependency (OD) [46]. Service dependency was expressed as an AND/OR graph called Service Dependency Graph (SDG) [56]. SDG considers IOD and performs composition based on it. However,

these dependencies are not sufficient to express dependencies among services. These dependencies can be identified only in a composed application where the workflow is specified. If the workflow is not available, service dependencies cannot be deduced. However, if SOA follows the CCSOA (Consumer-Centric SOA) [102] in which application templates, workflows and collaboration templates can be published and discovered (in addition to services). In CCSOA, dependencies between services can be discovered by their association with workflows and templates. Thus, these two services have IOD. Dependency information is also useful in placing sensors for instrumentation [13]. As dependency is domain related, the same service may have different dependencies in different domains. For example, two services, multimedia trans-coding and slice services are related to each other because in a workflow, multimedia trans-coding service is called before slice service.

### *7.4.1 Axioms*

1. **Nodes**: A node represents an abstract service in an ontology. It is a unique entity.

2. **Relationships**: It is denoted as a connection between two nodes in an ontology. Relationships are directional. A relationship consists of the dependent node, the relationship, and the target node.

3. **Dependency**: For a designated relationship, it denotes that this relationship implies a dependency between one node and the other.

4. **Dependency Domains**: It represents collections of dependencies that share an analytical interest.

5. **Dependency Likelihood**: It is a measure of how likely a relationship is to induce a dependency.

6. **Likelihood estimation functions**: It is a collection of functions that estimate likelihoods. These can be initial estimation functions (for providing initial estimates), asymmetric initial estimation functions and estimation update functions.

### *7.4.2 Property Definitions*

**Transitive Dependent Relationship**

Two relationships have a transitive dependency through an intermediate node if they following characters: If $A \xrightarrow{<N,\Re,L_1>} B$ and $B \xrightarrow{<N,\Re',L_2>} C$ then $A \xrightarrow{<N,\Re'',L_3>} C$. $\Re$ , $\Re'$ and $\Re''$ can be any item of relationship set R. Here, A and B have relation $\Re$ in given domain N with likelihood $L_1$. B and C have relation $\Re'$ with likelihood $L_2$ in same domain. So A and C have relation $\Re''$ in same domain with likelihood $L_3$. Here, $\Re$ can be same as $\Re'$. And $\Re''$ must be $\Re$ if $\Re$ dominant $\Re'$ . Or $\Re''$ should be same as $\Re'$.

**Symmetric Reflective Dependency**

Two relationships have a symmetric reflective dependency if they have following characters: If $A \xrightarrow{<N,\Re,L_1>} B$ then $B \xrightarrow{<N,\Re,L_2>} A$. $\Re$ can be relationship parallelWith. Here, if A and B have relationship $\Re$ in a given domain N with likelihood $L_1$, B should have the same relationship with A in the same domain with likelihood $L_2$.

Table 7.1: Corresponding Relationship

| | | | | |
|---|---|---|---|---|
| $\Re$ | hasInput | before | subOf | calledBy |
| $\Re$ | hasOutput | after | parentOf | hasCall |

**Asymmetric Reflective Dependency**

Two relationships have an asymmetric reflective dependency if they have following characters: If $A \xrightarrow{<N,\Re,L_1>} B$ then $B \xrightarrow{<N,\Re',L_2>} C$. $\Re$ and $\Re'$ can be any item illustrated as table 7.1, where $\Re$ and $\Re'$ are corresponding to each other. Here, A and B has relationship $\Re$ in a given domain with likelihood $L_1$, which means B has relationship $\Re'$ with A in the same domain with likelihood $L_2$.

**Redundant Transitive Dependency**

This type of dependency is where there are multiple transitive dependency paths between two nodes such that the likelihood that the two nodes are dependent increases, but the number of relationships between the nodes decreases. It has following character: If $A \xrightarrow{<N,\Re,L_1>} B$,$A \xrightarrow{<N,\Re',L_2>} D$,$D \xrightarrow{<N,\Re'',L_3>} B$ and $B \xrightarrow{<N,\Re''',L_4>} C$ then $A \xrightarrow{<N,\Re'''',L_5>} C$ . Here, A and B have relation $\Re$ in given domain N with likelihood $L_1$ . A and D have relation $\Re'$ with likelihood $L_2$ in same domain. D and B have relation $\Re''$ with likelihood $L_3$ in same domain. B and C have relation $\Re'''$ with likelihood $L_4$ in same domain. So A and C have relation in same domain with likelihood $L_5$. Here $\Re$,$\Re'$,$\Re''$,$\Re'''$ and $\Re''''$ have same meaning in Transitive character. These three properties can be illustrated as Fig.7.2.

### *7.4.3 Formal Notation Definition*

Some formal notation will be defined here:

**Complete Dependency Set**

A dependency set in which all dependencies and derived dependencies have been added.

**Dependency Likelihood Threshold**

The minimum likelihood value of dependencies in a complete dependency set.

**Complete Dependency Set at Level Z**

A dependency set that has all dependencies and derived dependencies that are above this value. Derived dependencies below this threshold are not in the set.

**Degree of Dependency**

the smallest number of intermediate dependency relationships between two nodes. That is, if $A \xrightarrow{<N,\Re_1,L_1>} B$ , then A is 1 degree from B. If $A \xrightarrow{<N,\Re_1,L_1>} B$ and $B \xrightarrow{<N,\Re_2,L_2>} C$, then A is 2 degrees from C.

### *7.4.4 Operations*

**Create Transit Dependency Relation**

$A \xrightarrow{<N,\Re_3,L_1\times L_2>} B$=$A \xrightarrow{<N,\Re_1,L_1>} B < t > B \xrightarrow{<N,\Re_2,L_2>} C$. Here, $< t >$ represents two relations have transitive dependency property.

**Create Symmetric Dependency**

$B \xrightarrow{<N,\Re_2,L_1>} A = < s > A \xrightarrow{<N,\Re_1,L_1>} B$. Here, $< s >$ represents two relations have symmetric dependency property.

**Create Asymmetric Dependency**

$B \xrightarrow{<N,\Re_2,L_1>} A =< a > A \xrightarrow{<N,\Re_1,L_1>} B$. Here $< a >$ represents two relations have asymmetric dependency property.

**Use Node**

This operation (or these operations) updates likelihood estimates based on node usage. This includes rules reporting actual dependencies and for updating likelihood estimates. It will be detailed in section VI.

### *7.4.5 Theorems*

- Transitivity is associative: If $A \xrightarrow{<N,\Re_1,L_1>} B < t > B \xrightarrow{<N,\Re_2,L_2>} C$ and $B \xrightarrow{<N,\Re_3,L_3>} C < t > D \xrightarrow{<N,\Re_4,L_4>} E$ , then $A \xrightarrow{<N,\Re_1,L_1>} B < t > B \xrightarrow{<N,\Re_2,L_2>} C < t > D \xrightarrow{<N,\Re_4,L_4>} E$

- When adding a new node to a complete dependency set, only the nodes within 1 or 2 degrees must be evaluated when adding the new node. Because all dependencies and derived dependencies have been added to complete dependency set, only 1 or 2 degrees are possible new. Nodes that have more than 3 degrees are already in the complete dependency set.

- If two relations have symmetric dependency property $B \xrightarrow{<N,\Re_1,L_1>} A =< s > A \xrightarrow{<N,\Re_2,L_2>} B$, then $\Re_2$ must equal $\Re_2$, $L_2$ must equal $L_1$.

- If two relations have asymmetric dependency property $B \xrightarrow{<N,\Re_1,L_1>} A =< a > A \xrightarrow{<N,\Re_2,L_2>} B$, then $\Re_2$ must not equal $\Re_2$, $L_2$ may equal $L_1$.

- $\Re$,$\Re'$,$\Re''$,$\Re'''$ and $\Re''''$ in redundant dependency property can be same, which means $\Re = \Re' = \Re''$=$\Re''' = \Re''''$ .

### *7.4.6 Algorithms*

The Algorithm 1 identifies all the relationships between node A and B based on transitive, symmetric and asymmetric properties on relationships. In other words, it will identify those dependency relationship not specified but can be derived by the existing relationships. The algorithm can be used in a modified Warshall's algorithm to identify all the dependency relationship in a set of ontology systems.

---

**Algorithm 4:** identifies all the relationships between node A and B

---

**Input:** Starting Node A, Target B, a likelihood threshold z, Set of Dependency Relationship D, where $d_i \in D$ is a dependency $x \xrightarrow{<N,\Re,L>} y$

**Output:** $D' = D \bigcup D^{+}_{AB}$

set R= D

**while** *there is a path between A and B* **do**

- Put all the nodes in the path into S1 **while** $a \in S_1$ **do**
  - **while** $b \in S_1$ **do**
    - **if** $r = a \xrightarrow{<n,r,l>} b$ **then**
      - **if** *r has Reflexive character* **then**
        - add $b \xrightarrow{<n,r,l>} a$ into R
      - **if** *r has Symmetric character* **then**
        - add $b \xrightarrow{<n,r,l>} a$ into R
      - **if** *r has Transitive character* **then**
        - **if** $A \xrightarrow{<n,r,l_1>} a \in R$ *and* $B \xrightarrow{<n,r,l_2>} a \in R$ *has Transitive character and* $l \times l_1 \times l_2 > z$ **then**
          - add $A \xrightarrow{<n,r,l \times l_1 \times l_2>} B$ into D

---

The Algorithm 2 is used to merge two dependency sets. The Algorithm 3 is used to merge two complete dependency sets. The Algorithm 4 is used to make an incomplete

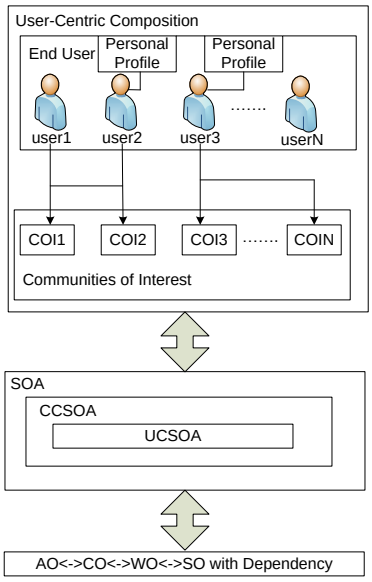



Figure 7.3: User Centric SOA Composition Architecture

dependency set to complete dependency set.

## 7.5 Composition With Dependency Support

This proposed framework is shown in figure 7.3. The architecture has six layers:

- Dependency layer: This provides dependency among services, application and workflow. Some of them directly come from ontology. The others are derived from those dependencies.

- Ontology layer: This classifies and represents relationships among templates, workflows and services. In figure 7.3, AO is Application Ontology, CO Collaboration Ontology, WO Workflow Ontology, and SO Service Ontology.

- SOA layer: This layer provides conventional SOA services such as publishing, discovery and broker services.

**Algorithm 5:** Add new node to a complete dependency set

**Input:** A set of nodes N Complete Dependency Set D that covers N which has a likelihood threshold z new node A, where $A \notin N$ with a set of dependencies $D_{new}$ to nodes in N Note that D and $D_{new}$ are sets of dependencies where each $d_i \in (D \bigcup D_{new})$ is a dependency between two nodes X and Y of the form: $d_i = X \xrightarrow{<D,\Re,L>} Y$ , where $X \in N$ or $X = A$, $Y \in N$ or $Y = A$ , $D$ is the domain of the dependency relationship, $\Re$ is the relationship, and L is the likelihood that the relationship is dependent.

**Output:** $D^{'} = D \bigcup D_{new} \bigcup D_A$, where $D_A$ is the set of derived dependencies of node A. that exceed the given likelihood threshold z.

**Assumptions:** $f_{prop}(R_1, R_2, p)$ is a relationship property function where $R_1$, $R_2$ and p is one of the relationship properties { Transitive, Symmetric, Asymmetric }. $f_{prop}(R, p)$ is defined to be 1 if the relationship R has property p, 0 otherwise.

*steps:*

1. Add symmetric and asymmetric dependencies between A and nodes that are 1 degree from it
2. Add transitive dependencies between A and the nodes 2 degrees from A
3. Add any symmetric and asymmetric dependencies that can be derived from the new transitive dependencies.

**Algorithm 6:** Merge two complete dependency sets

**Input:** A set of nodes M and N Dependency Set D that covers M which has a likelihood threshold z and Dependency Set E that covers N which has a likelihood threshold z.

**Output:** $D^{'} = D \bigcup E$ and $M^{'} = M \bigcup N$.

**Assumptions:** $f_{prop}(R_1, R_2, p)$ is a relationship property function where $R_1$, $R_2$ and p is one of the relationship properties { Transitive, Symmetric, Asymmetric }. $f_{prop}(R, p)$ is defined to be 1 if the relationship R has property p, 0 otherwise.

*Steps:*

1. Take two nodes A,B from nodes M
2. if $D_{AB} \in D$ and $D_{AB} \notin E$
3. Add $D_{AB}$ to E
4. Call add a new node to a complete dependency set algorithm to add node A and B to N
5. repeat 1 to 4 steps until M is empty or one node left
6. if there is one node in M, Call add a new node to a complete dependency set algorithm to add the left node
7. return set $D^{'}$ and $M^{'}$

**Algorithm 7:** Make an incomplete dependency set to a complete dependency set

**Input:** A set of nodes M Dependency Set D that covers M which has a likelihood threshold z.

**Output:** $D^{'}$, $D^{'}$ is the complete set of $D$

**Assumptions:** $f_{prop}(R_1, R_2, p)$ is a relationship property function where $R_1$, $R_2$ and p is one of the relationship properties { Transitive, Symmetric, Asymmetric }. $f_{prop}(R, p)$ is defined to be 1 if the relationship R has property p, 0 otherwise.

*Steps:*

1. Create an empty set $D^{'}$ and $M^{'}$
2. Take two nodes A,B from nodes M
3. if $D_{AB} \in D$ and $D_{AB} \notin D^{'}$
4. Add $D_{AB}$ to $D^{'}$
5. Call add a new node to a complete dependency set algorithm to add node A and B to $M^{'}$
6. repeat 1 to 4 steps until M is empty or one node left
7. if there is one node in M, Call add a new node to a complete dependency set algorithm to add the left node
8. return set $D^{'}$

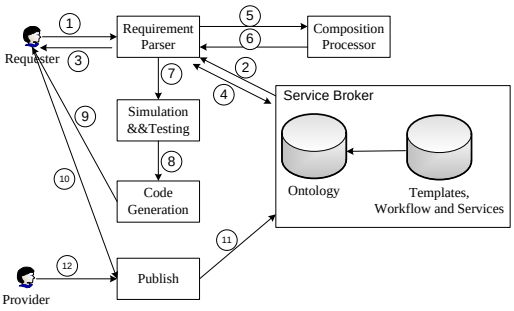



Figure 7.4: A Composition Operation Sequence

- CCSOA layer: CCSOA publishes not only service specification, but also application, collaboration, and workflow templates.
- Community of Interests (COI) layer: This is the place where common solutions related to a domain are stored and classified for community members to reuse.
- User profiling layer: Each individual user has his or her preference and behaviors, and thus each user may have customized solutions. This information helps in identifying best fit services.

### *7.5.1 Composition Process*

An operational sequence for the composition process is shown in figure 7.4. Note that the framework allows many different ways for composition, and this is just one of many possible ways.

The process has following steps:

1. A user submits a specific application requirement and it may include specific workflow processes, potential service description and attributes, and related information to an appropriate COI.
2. The requirement parser analyzes user's requirements and identifies those related application, collaboration and workflow templates in the ontology that closely

match the needs of the requirements using dependency analysis. The system returns a set of candidate application templates, ranked by closeness and other factors to the user. These templates come with a set of associated workflows and services.

3. The candidate templates are presented to the user, and the user can make informed decisions on the selection or modification of these templates. If no such application template is available, a new application template needs to be created. The user may need to modify a candidate template to fit the current application. In this step, a set of application, collaboration and workflow templates will be chosen to fit the application. Simulation can be performed to evaluate the overall system at this time even though not all the services and workflows have been finalized.

4. Once the templates are obtained, the set of candidate workflows and services are automatically identified by dependency analysis. The set of candidate workflows and services ( stored in workflow ontology and service ontology) with their likelihood information are presented to the user for final selection. The user finalizes the selection based on various criteria. It may be necessary to create new services and/or workflows, or modify existing services and/or workflows to meet the user requirements. Those new or modified services and workflows can be published with dependency information updated.

5. After all the decisions are made, the selected application template with all the selected workflows and services will be packaged together to form a new application.

6. The composed application will be sent back to requirement parser.

7. The composed application will be simulated and tested before making final decision. If there is any problem, repeated the previous steps to obtain a new composition.

8. Finally, the code can be generated from the final packaged composition.

9. The code will be returned to the user with links to published items such as services.

10. The user may decide that the newly composed application will be a good candidate for reuse, and publish the application as an application template.

11. The system will save this application template, map it to application ontology and update dependency likelihoods related to this.

12. After seeing the newly published application template, a provider can submit services and workflows to be associated with this application template. The dependency analysis can be used to verify the validity of these associations.

    Note that the proposed 2-steps composition process is described as steps 2, 3 and 4 in the above process. Also, a service consumer may become a contributor in the community if the consumer performs step 10. Furthermore, this consumer acts as a virtual service provider by publishing an application template.

### *7.5.2 Key Techniques*

The proposed framework needs many techniques to implement the design. Note that UCSOA already provides COI organization and management including verification and validation, rapid application generation, and publishing and discovery mechanisms.

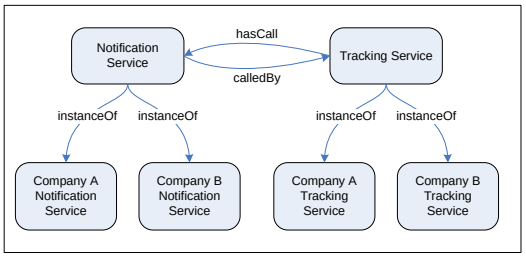



Figure 7.5: Services with Dependencies

### Initialize and Update Likelihood

Section 7.4 introduced the dependency likelihood L. This section defines how L can be initialized and updated. Consider an example show in figure 7.5.

In this example, Notification services collaborate with Tracking services to get tracking information to their customers. This example show 6 direct dependencies: NotificationService calls TrackingService, NotificationService hasCall from TrackingService, CompanyANotificationService and CompanyBNotificationServices are instances of NotificationService, and CompanyATrackingService and CompanyBTrackingService are instances of TrackingService. Likelihoods for these 6 dependencies are initialized and updated using a recursive Bayes filter. Initializing the Filter. The initial likelihood for dependencies can be assigned to a predefined constant such as 0.5. Note that this assumes 50service is selected, the other service will be selected in the same application. In the example shown in figure 7.5, the direct dependency likelihood estimates will be initialized as shown in figure 7.6 on page 106. Where T is the shipping service application domain. Updating the Filter. As services selected to be packaged into an application, the likelihood dependency estimate can be updated based on historical usage patterns. Consider the dependency:

After n usages of item A and the next time A is used, L can be updated using follows:

$L_{n+1} = \frac{n*L_n+u_{n+1}}{n+1}$, where $u_{n+1}$ is 1 if A is dependent on B, 0 otherwise.

In other words, as two services are selected to participate in an application, their dependency likelihood increases. Furthermore, this information can be customized with respect to a community of users or individual users. For example, a specific user prefers CompanyA over other companies, thus with respect to this user, the dependency likelihood between CompanyATrackingService and other shipping-related services is high, but the dependency likelihood between other tracking services and shipping-related services will be low. If the user decides to change the preferred company, the user can either reset the likelihood to the initial value or let the system corrects itself by following the update process.

### Ranking

Once an item is published, it can be ranked by all the users including service consumers, brokers, and providers. Not only instance services of each abstract service can be ranked, but also test scripts, test cases and templates. Ranking can be based on test and evaluation such as reliability evaluation or personal opinions, i.e., social ranking. Furthermore, trust information is also important for ranking and ordering. In [106] that talks how to build service trust model. These ranking information can be used together with dependency information in composition.

### Publication with Dependency Analysis

Published items can be analyzed by dependency analysis at publication time. For example, a published service specifies its attributes such as inputs, outputs, preconditions, effects, grounding, and description. Similar existing services can be discovered based on these attributes. These existing services can be used as a basis for performing dependency analysis on the newly published service. In this way, a

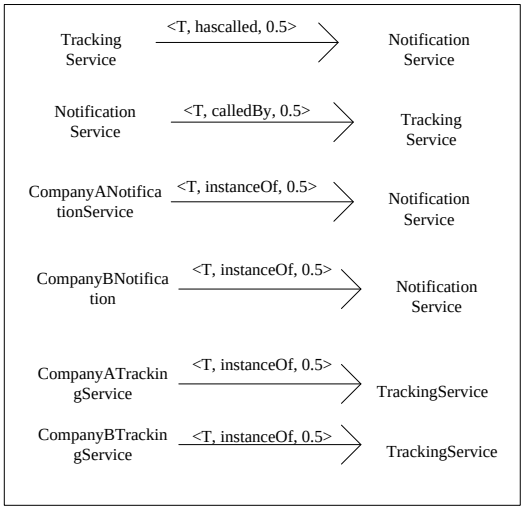



Figure 7.6: Initial Dependency Likelihood Estimates

newly published service may be associated with workflows, collaboration templates, and application templates automatically. These associations can be re-confirmed by contributing providers or by users. Furthermore, dependency analysis can be used as a T&E (Test and Evaluation) mechanism to support service publication. For example, if a provider publishes a service, and indicates that it is useful for certain applications and/or collaboration templates. However, dependency analysis on the service does not match well with the indicated templates. Thus, the associations with these templates need to be rejected. This can be done automatically if tool support is available. If this mechanism is available for every party, a provider can perform dependency analysis before publishing their services or workflows.

## 7.6 Case Study - Shipping Domain Tracking System

This section uses a shipping domain service composition system example to illustrate the composition process. We get the system requirements from three different companies. According to applications' requirement, three different application sys-

Table 7.2: Existing Items in Different Applications

| | **Company A** | **Company B** | **Company C** |
|---|---|---|---|
| Service | 16 | 17 | 13 |
| Workflow | 20 | 21 | 17 |
| Application Template | 5 | 6 | 5 |

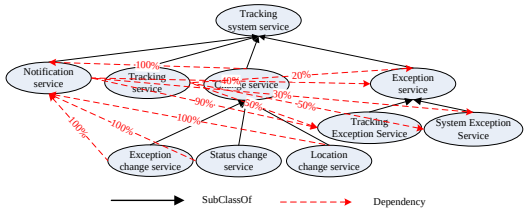



Figure 7.7: Dependency Information

tems have been developed. Many services, application template, workflows and ontology have been published in [23]. For illustrating, a fourth company's requirements have been proposed. The system consists of four participants, a company manager who wants to see statistics data such as the profit of the company in the most recent month, a system administrator who will manage the system need to make sure that system works well, a carrier who wants to change the status of shipment and a user who wants to track the status of shipment.

### *7.6.1 Existing Items*

Forty-six services, fifty-eight workflows and sixteen application templates have been published, which table 7.2 illustrates how they distribute.

Shipping domain ontology has been presented as 7.1. Dependency information of the service ontology can be illustrated as figure 7.7.

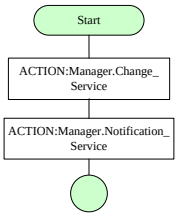



Figure 7.8: Notification Way Change Workflow

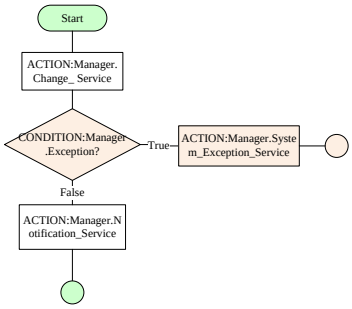



Figure 7.9: Notification Way Change Workflow with Its Dependency

### *7.6.2 Specifications*

The mission is to let a manager, Jerry quickly composes an application according to their requirements. Jerry wants to get informed by cell phone if there is any tracking exception happens in the system. The key point is he does not know about software design or programming and the system does not have any available service or application that can be used. They will use PSML-S [99] to publish their requirements or compose their applications.

### *7.6.3 Notification Way Change Workflow*

The notification way change workflow can be constructed by PSML-S and the shipping domain ontology. For demonstration, manager's notification way change is illustrated as figure 7.8.

As manager does not need to consider any dependency information, he can focus

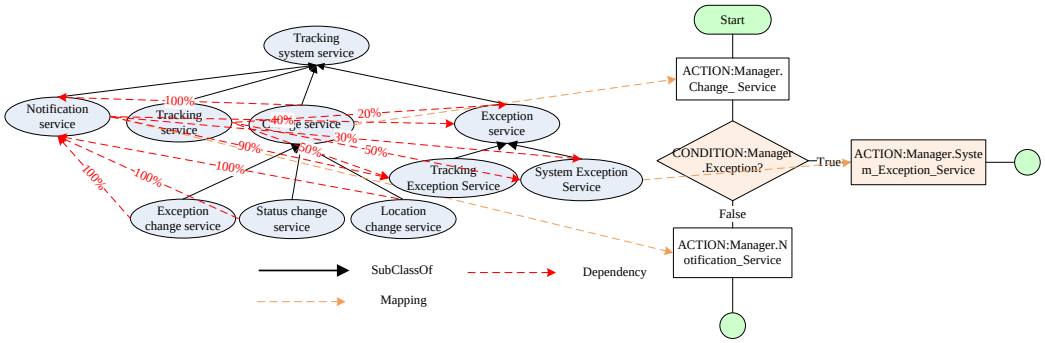



Figure 7.10: Notification Way Change Workflow and Its Mapping

on what he wants to do. He can ask system to do the work. The composition system analysis the *Change_Service* and finds that has dependency. So, composition system will automatically choose its dependency with *Change_Service* for the workflow. It can be illustrated as figure 7.9 on the previous page.

According to the description before, user-centric composition process allow users focus on what they need and let the composition system select their dependent components automatically. So, the system will help users with not much programming background to do composition. Notification way change workflow and its corresponding mapping service can be illustrated as Fig.7.10.

## 7.7 Conclusion

This paper provides a user-centric service composition process to assist people to compose applications. One key technique is dependency analysis among published application templates, collaboration templates, workflows, and services. The dependencies identify those associated items quickly. The dependency relationships can be formalized and analyzed to ensure coverage.

Chapter 8

# SERVICE REPLICATION WITH MAPREDUCE IN CLOUDS

In a typical cloud environment, services wait to serve users' request. If a service receives more requests than it can handle, it needs to acquire additional resources. This paper proposes a new service replications that allows a cloud to adjust its service instance deployments in response to existing and projected service requests. This approach is called Service-Level MapReduce (SLMR) as it is based on MapReduce, a parallel processing mechanism commonly used in cloud environments such as GAE(Google App Engine). SLMR includes dynamic service replication and predeployed service replication. Furthermore, a passive SLMR approach that depends on the cloud management service (CMS) and an active SLMR approach that does not need the support from CMS will be introduced.

## 8.1 Introduction

Recently, cloud computing has received significant attention. Many companies have started their cloud projects including Software-as-a-Service (SaaS) [126], Platform-as-a-Service (PaaS) [125], and Infrastructure-as-a-Service [124]. Some example systems include Microsofts Azure [61], Googles App Engine (or GAE) [41], Amazons EC2 [3] and SimpleDB [4]. Traditional service replication is passive, i.e., the service being replicated does not participate in the decision on where to replicate, when to replicate, or the number of copies to replicate. This passive service replication is useful from the separation of concerns point of view, as it separates service deployment from service functionality. Thus, service replication is not a standard feature in service-oriented architecture (SOA) [23]. However, services deployed in a cloud

can be replicated as service replication is common to support scalability and elastic computing. Furthermore, cloud platforms often have mechanisms for distributing and replicating data among many processors in the cloud. For example, cloud platforms such as GAE provide automated triplicate redundancy to support data and service availability and reliability. In a cloud, users can request services to deal with a large amount of data. In this case, a service may not be able to complete users' requests in a timely manner, and thus the service may be unavailable for other users. For the users' requests that have timing constraints, this can be an issue. Traditional passive service replication may not be able to guarantee that a request will be handled with a timing constraint. This paper introduces a MapReduce-based approach to service replication that addresses this problem. The MapReduce splits a task to smaller tasks, and executes them in distributed nodes in parallel. There are two main phases in MapReduce: map and reduce. An input set is split into certain number of segments. For each segment, a job is created to run the map function on that segment. The map function produces intermediate results. Once the map phase is complete, the reduce phase starts by handling a portion of the intermediate results. MapReduce has been widely used in cloud computing. Furthermore, the map and reduce functions are implemented as library functions at the code level. They are written by programmers for a specific MapReduce platform in a specific language. For Googles MapReduce functions are written in C++ [30]. Ideally, in a cloud, every computation is a service, rather than just code. A service has its code, but it also has service specification (such as IOPE or input, output, preconditions, and effects) that can be published, discovered, and composed visually. Map and reduce functions can be implemented as services. This paper develops the following strategies for composing service-level MapReduce (SLMR) Applications: i. Passive Service Replication Strategy (SRS): In this approach, a management service controls the composition

and service replication, as well as manages MapReduce service requests. ii. Active SRS: In this strategy, services manage themselves. For both strategies, the following problems for MapReduce are studied: Splitting the input set for performance, and Fault Tolerance.

## 8.2 Related Work

Many commercial cloud platforms are available. For example, Google has GAE and it is a platform for Web applications. Developers do not need to worry about the load of web applications, and they can be balanced by GAE. Based on the service agreement, GAE can impose storage size and computing usage limitations [41]. Microsoft Azure provides a cloud environment for .NET-based web applications [61]. Amazon has a range of cloud-based products including EC2 and SimpleDB. EC2 is a web service platform that provides resizable compute capacity in a cloud [3]. SimpleDB is also a web service providing core database functions of data saving, indexing and querying in a cloud [4]. MapReduce is widely used to support parallel computing on large data sets in distributed systems. It was inspired by map and reduce function in functional programming such as Lisp [128] and ML [127]. However, MapReduce has a different meaning. Here, the map is initiated by a master node, the input is split into many small sub-problems, and then distributed to worker nodes. Worker nodes process the sub-problems and produce intermediate results in the form of key-value pairs. Once the map step is complete, the reduce step begins. The master creates certain number of workers to perform the reduce operations. Intermediate data from the map step is processed by the reduce workers based on the keys. Intermediate data with the same key is handled by the same reduce worker. The main advantage of MapReduce is that it allows map and reduce operations to be distributed so they can occur on different processors in parallel. This can substantially decrease the

processing time required for large data sets [30]. Service replication is an important concept in cloud computing. It can occur when there is a large backlog of queued requests in the service queue. The service can be replicated so as to provide additional processing capacity. In service replication, data replication can be an issue as data will be processed by different copies of the same service. Gao [39] proposed an application-specific data replication for edge services e-commerce. It took advantage of application-specific semantics to design distributed objects to manage a specific subset of shared information using simple and effective consistency models. Another issue is to choose replicated services to serve users' request. Zegura [133] proposed an application layer approach to choose servers. It does not replicate services but replicate servers so as to replicate services as services hosted by different servers. Stantchev [83] proposed an OS-level replication strategy to handle dynamic service replication. They proposed two ways to replicate web services: per-process replication and per-thread replication. Nevertheless, dynamic service replication is still difficult because services normally cannot get a full authorization of servers. However, pre-deployed services strategies can be useful. In this strategy, replicated services are pre-deployed on different servers but remain hibernating until they are needed. Service replication is also used for other purposes. For example, Zheng [136] proposed a distributed replication strategy evaluation for fault-tolerant web services. Their purpose is to provide reliable services by replicate services when service cannot perform well. Service replication is a useful strategy. In most cases, service replication is a passive selection.

## 8.3 Cloud Architecture

This section presents a high-level cloud architecture containing elements that support service replication and dispatching. This is not a complete architecture; instead,

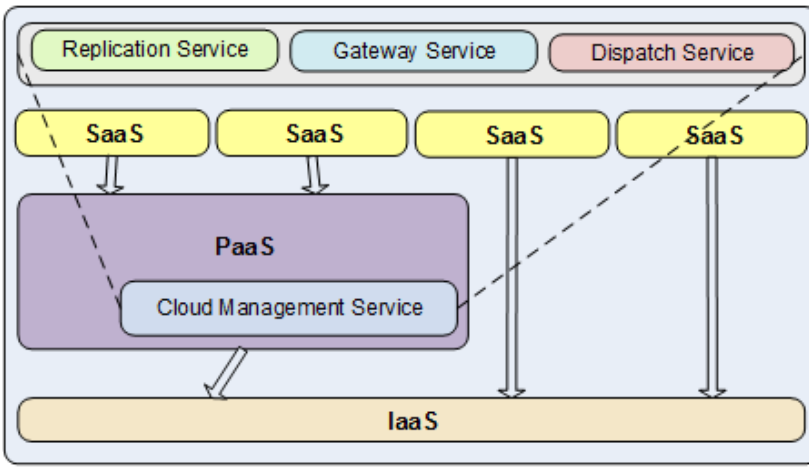



Figure 8.1: High Level Cloud Architecture

it focuses on the dispatching and replication services. The dispatching, replication, and the related services in the cloud infrastructure are illustrated in Figure 8.1.

1. Cloud Management Service (CMS) has the following functions:

   (a) Monitoring services: The CMS monitors the execution of services to ensure that they are functioning, making progress, and completing normally. If it detects certain services are overloaded, it can replicate additional services and dispatch requests to relieve their loads.

   (b) Dispatching services: When services have been replicated, the CMS determines the location replicated services to be deployed, replicates them, and updates the service information so that computation requests can be directed to the newly replicated services.

   (c) Complete the map process: In passive SRS, the CMS manages the map and reduce processes. For the map process, the CMS manages splitting, and the map service replication, and dispatches data to the replicated services.

   (d) Complete the reduce process: After the map step is completed, the CMS

collates the results, replicates the reduce services, and dispatches requests to the reduce services.

2. Replication Service manages service replication in two ways:

   (a) Dynamic Service Replication: Services will be dynamically replicated and deployed as determined by the CMS. As a cloud often duplicates or triplicate data already [40, 18], a cloud can duplicate a service by deploying another copy of code in one server, and distinguish the new copy from existing ones. However, the cloud often does not allow services to make its own decision to duplicate itself as this can be a source of computer virus. Thus, any service duplication will be managed by the CMS to prevent this kind of security attack.

   (b) Pre-deployed Service Replication: Services are pre-deployed on specific servers. Services remain in a hibernating state when they are not processing any tasks. When the CMS needs to replicate services, it activates hibernated services to complete the replication process.

   After replicating services, they can be released. As the cost of dynamic service replication may be high, so a replicated service created by dynamic service replication can enter a hibernation state once it has no tasks to execute so that it can be re-activated later.

3. Gateway Service: This serves as the entry point of the cloud, and it gets users requests and transfers them to a specific server for proceeding.

4. Dispatch Service: When the CMS performs dynamic service replication , replicated services need to be deployed onto different servers in the cloud. This service performs this task.

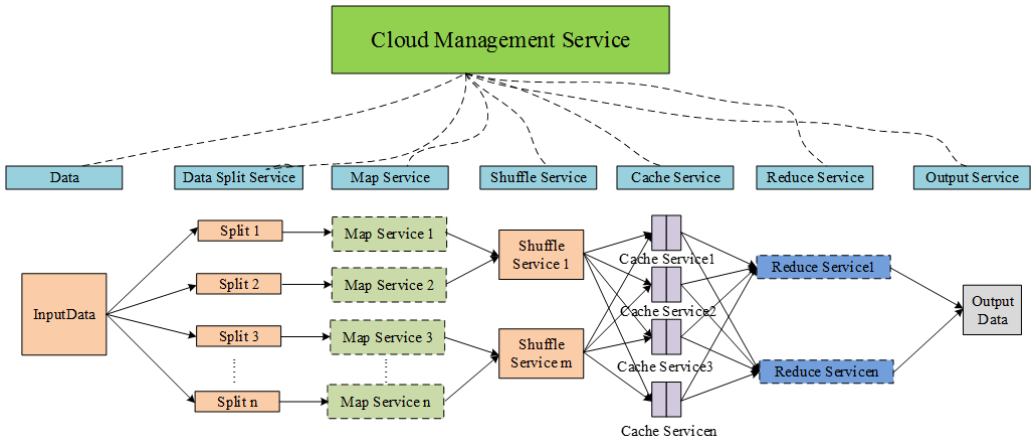



Figure 8.2: Service-Level MapReduce Process

## 8.4 Service Replication Strategies

### *8.4.1 Service-Level MapReduce*

MapReduce [30, 122] is popular, and it can be extended so that it can work at the service level. Figure 8.2 illustrates SLMR:

1. The CMS manages the SLMR process. The CMS will create an appropriate number of map services, split services, shuttle services, cache services, reduce services and output services dynamically based on processor capacity and timing constraints.

2. Data Split Services: These will split input into n partitions according to users requirements, and current cloud status such as server availability.

3. Map Services: These will perform the map method

4. Shuffle Services: These will reorder the intermediate data and send them to suitable cache services. Normally, the cloud will provide a shuffle service to data of key-value structure. In addition, users can provide their shuffle services

for specific data type.

5. Cache Services: These will store intermediate data from the map services but not necessarily in the form of key-value pairs. Before these data come to cache services, they would be sent to shuttle services to reorder. Users may select or provide their own cache services.

6. Reduce Services: These services take data from cache services, collate them and send them to an output service.

SLMR has the following features:

1. Map and reduce services can be replaced by other similar services. That is, the map and reduce are not coupled at a code level, only at the interface level.

2. Service providers can focus their concern with their services and may be less concerned with the overall MapReduce process. In Hadoop [122], users need to write map and reduce functions, while Hadoop finds workers to run map and reduce functions. In SLMR, the CMS is in charge of the MapReduce process.

3. SLMR is service-oriented. This makes it more flexible, and services can be published, discovered, and composed. This feature also enables dynamic composition of MapReduce operations, as long as map and reduce services are available. In this way, all SLMR services can be published, and reused by others.

4. Users that used to write map and reduce functions can now compose these services or use existing services from service providers.

5. SLMR can support more data types than the traditional MapReduce. In the MapReduce framework, users must design data carefully so that they can match the key-value structure. However, sometimes, data cannot be easily represented

by the key-value structure such as multimedia data. Here, input data of service can be any type, not necessarily key-value pair. If users need to handle special data, they can write split services and shuffle services.

6. Map service and reduce service can be same. So, the cloud can clone services and let them play as map and reduce services. In this way, execution can be accelerated.

7. The framework allows different reduce services so that more than two types of results can be introduced while traditional MapReduce can have only one type of results.

8. Both dynamic service replication and pre-deployed service replication can tolerate some faults. When services are replicated, redundant services are created, and these redundant services can replace those failed services in case of faults. SLMR also can allow these extra services to run concurrently.

9. Both dynamic service replication and pre-deployed service replication can get better accuracy by a voting mechanism [103]. The idea is to take majority result from all responses that running replicated services.

### *8.4.2 Number of Replications Needed*

It is not good if replicas are over supplied, as they consume more resources than needed in a cloud. However, if services replicas are insufficient, the MapReduce process may not perform well. So, it is necessary to determine the number of replication services needed. The following formula is a possible solution:

$$N = \frac{RD}{SPC * T}$$

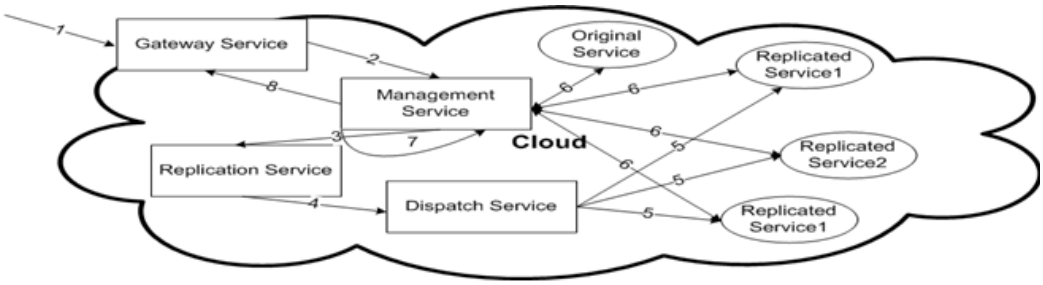



Figure 8.3: Details of passive SRS process

Here, RD represents the size of request data (in number of records). SPC represents the rate at which a service processes records. T represents users timing constraints. N represents the number of services needed. If formula gets a float number, N must be up-bounded. For example, if the calculation result is 2.01, N should equal 3. Here, three constraints, size of request data, service processing capability and time are considered as major reasons affecting the number of replications.

### *8.4.3 Passive Service Replication Strategy*

This strategy has three sub-processes as illustrated in Figure 8.3.

1. The CMS controls service replication and data splitting. Service replication is managed by replication service while data partition is charge.

2. Replicated services work alone, get input data from the CMS and return results to CMS for completing the Map process.

3. CMS collates results from replicated services to get final results by completing reduce process.

Figure 8.3 shows the eight steps for completing the whole process.

1. Users submit their requests to the cloud. A gateway service at the cloud boundary receives these requests.

2. The gateway service transfers users requests to the CMS. If users have special requirements such as users have preferred services, the gateway service will pre-process these requests before passing to the CMS.

3. The CMS asks the replication service to replicate specific services, and dispatch service to dispatch the service requests. This process can use Formula 1 to calculate the number of replicated services needed based on the requests timing constraints. The original service then asks the replication service to replicate N-1 copies of the map services.

4. After the replication service finishes map service replication, the dispatch service will deploy the replicated services (if dynamic service replication is being used) or activate the replicated services (if passive SRS is being used).

5. The dispatch service deploys or activates the replicated services.

6. The CMS splits the request data into N parts and dispatches them to N map services. After that, it calls replicated services to work on the data and wait until it gets all the results.

7. The CMS collates the results to complete the reduce process.

8. The CMS returns the final results to the gateway service and it in turn returns them to the users.

he CMS is responsible for managing the map and reduce services in case of faults. If a replicated map service stops working, The CMS can detect this, replicate another service to replace the failed service. Here, one can see passive SRS is a centralized process strategy, and it has advantages and disadvantages of a centralized architecture. It is easier to manage because the MapReduce services are managed by a central

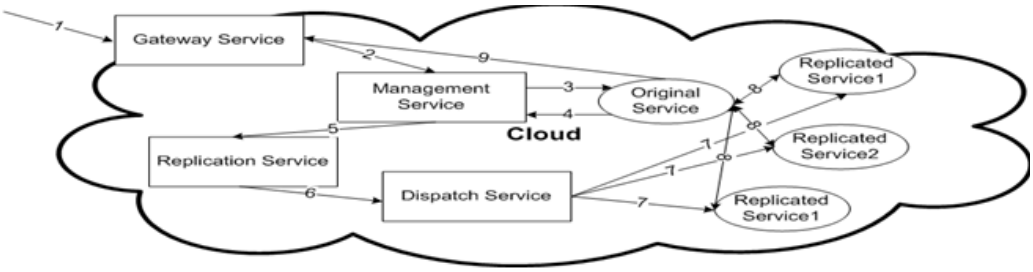



Figure 8.4: Details of active SRS process

manager. However, the CMS becomes a critical service of the cloud. If the CMS becomes overloaded, it will degrade the performance of the cloud. This is the similar problem that the original GFS (Google File System) [40] faced before. One common solution is to use some P2P technology such as DHT [67], where several CMS services are provided and DHT is used to dispatch services onto those services.

### *8.4.4 Active Service Replication Strategy*

The active SRS has three sub-processes as shown in Figure 8.4:

1. Each service determines its data partitioning strategies and service replication strategies. The CMS still manages the actual replication and dispatching, but is does so at the behalf of the services in the composed MapReduce application.

2. The CMS manages service replication and service deployment if dynamic service replication is used, or the activation of hibernating services if passive SRS is used.

3. Replicated services communicate with the original services to get input data and finish the map process. Then, the original service collates results from services and completes the reduce step in a similar manner. As shown in Figure 8.4, this process has nine steps:

(a) Users submit their requests to the gateway service, and this that is similar as passive SRS Step A.

(b) The gateway service transfers users requests to the CMS, and this is similar as passive SRS Step B.

(c) Instead of replicating original service in passive SRS Step C, The CMS calls the original service to deal with users requests.

(d) After the original service determines the partitioning strategy using a strategy such as Formula 1, splits the data, and requests that the CMS replicate and dispatch the map services.

(e) The CMS asks the replication service to finish service replication or activate the required number of services.

(f) After the replication service finishes services replication, it asks the dispatch service to dispatch the replicated services.

(g) The dispatch service deploys replicated services. This step is same as steps E and F.

(h) Each service communicates with the original service and gets input data to finish the map process. After that, all of them return results to the original service. It is the original service that initiates the reduce step in a similar manner.

(i) After the original service collates all returned results for getting final result, it will return the final results to the gateway service and in turn to the user.

In active SRS, fault tolerance is the services responsibility. This responsibility can be shared across one or more of the replicated services. Similar to a centralized fault

tolerance strategy, if individual map services fail, new map services are replicated and processing is requested of them.

Active SRS has its own advantages and disadvantages. It is a hybrid architecture in between a centralized architecture and a distributed architecture. It alleviates the burden of the CMS, which requires the original service to be in charge of map and reduce processes instead. It will also make the original service complicated to finish the MapReduce process. However, as this functionality can be packaged as a service, MapReduce compositions can include this functionality from previous published MapReduce management services. While passive SRS and active SRS share similar strategies such as service replication, and the ability to support service-level MapReduce, they are different with respect to the following aspects:

1. The main difference between passive SRS and active SRS is the location of the management of MapReduce process. passive SRS places the responsibility with the CMS, while active SRS with the services implementing the MapReduce.

2. The algorithms behind the passive SRS and active SRS are different, as illustrated in Sections 8.4.3 and 8.4.4.

3. The algorithm and logic of the original service are different. Active SRS requires additional logic for management and fault tolerance. However, with active SRS, this logic can be packaged as services, providing flexibility with respect to the level of fault tolerance needed.

## 8.5 Application Illustration

### *8.5.1 Data Sorting*

In this scenario, a user request contains a large number of data records that need be sorted by a sorting service. A service-level MapReduce service can reduce the overall time of sorting using the algorithm 8:

**Algorithm 8:** Data Sorting

**Input:** D is an input set that has a large number of records, service processing rate SPC and timing constraints T

**Output:** The sorted data

1 Calculate the number

2 Replicate N map services

3 Split data according to $D_1.....D_N = \frac{D}{N}$

4 Deliver the splitting data and call the replicated service to sort sub-data

5 Collate the sub-data to final sorted data

6 Return the final sorted data

From this algorithm, one can see the time complexity is O(nlogn), and this depends on how one sorts the sub-data and collate the sub-data. If one uses O(nlogn) sorting algorithm such as merge sort, the time complexity is O(n log n). If one uses O(n2) sorting algorithm such as Bubble sort [10], the time complexity will be O(n2).

### *8.5.2 Keyword Search in Large Documents*

In this scenario, a user requests a number of items matching certain keywords. This problem does not require much computation but requires service to handle a large number of documents. Just as the scenario given in Section 5.1, it is difficult

from a single service to finish keyword searching in a short time in traditional ways. Therefore, algorism 9 is introduced.

**Algorithm 9:** Keyword Search

**Input:** Large number of documents D that need to be searched, keywords K, service processing rate SPC and timing constraints T.

**Output:** The list of matching documents

1 Calculate the number

2 Replicate N Keywords Searching Services

3 Split documents D according to $D_1.....D_N = \frac{D}{N}$

4 Deliver the splitting documents and call the replicated service to search keywords in the splitting documents

5 Collate the sub-documents from replicated services to final sub-documents

6 Return the final sub-documents

From the basic ideas, one can see the time complexity depends on the search keywords in sub-documents and collate the sub-documents, which can be O(n) if comparing keywords with each string in documents, O(log n) if using search algorithm such as binary search tree [80] while not consider how to build up binary search tree. Search speed is also related to the document types. If document types are XML, some useful package such as JAXP [48], JAXB [71], JDOM [44] and JAX-RPC [47] in java and namespace such as System.XML [62] in C# can be helpful.

## 8.6 Case Study

In this section, word counting and inverted table for words will be used to illustrate service replication strategies for MapReduce in clouds. For demonstration purposes, passive SRS will be used. Replicated services can be pre-deployed onto

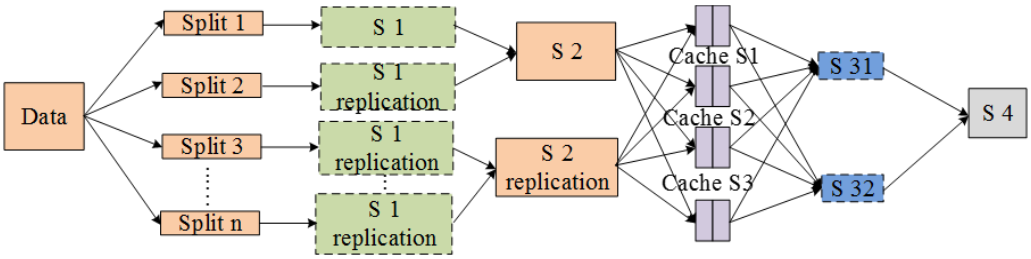



Figure 8.5: Instantiated Framework of SLMR

several windows servers using windows communication foundation (WCF) [59]. By using SLMR, one can design reduce services elaborately to get two results at same time.

1. We designed a map service S1 to count words in documents and get the result by list of ¡word, counts, and document¿.

2. We designed a shuttle service S2 so it can reorder the lists based on keyword word.

3. We designed two reduce services, S31 and S32, which service S31 is used to collate counts of words and other service S32 is used to get list of documents for the same word.

4. We also designed an output service, S4, so it can collate all results from reduce services and output the result.

Now, instantiated framework of SLMR can be illustrated as Figure 8.5. From Figure 8.5, one can see:

1. Users do not need to write split service but just use default service provided by SLMR.

2. Users can reuse other services provided by SLMR or service providers.

3. Designing the MapReduce process becomes service composition.

4. Intermediate data type do not necessarily be ¡key, value¿ structure. In this case, it is a list.

5. Reduce services can be different and all of them can be replicated.

6. Through service replication and running them in parallel, service execution can be accelerated.

## 8.7 Conclusion

This paper proposed services replication strategies for the MapReduce process in a cloud environment. The strategies focus on efficient use of the cloud resources with large requests. This problem cannot be addressed only by allocating more computation resource to the service from the cloud or simply replicating services and let the cloud balance the load. This paper proposed a SLMR, and introduces two strategies: passive SRS and active SRS, and illustrated these with examples. The examples demonstrate that SLMR service replication can decrease the execution time in a cloud.

# Chapter 9

# STA EXPERIMENT AND CASE STUDY

As Force.com cannot support all STA models, MultiOrg-SSTA, MS-TSTA, SM-TSTA, PP-TSTA and MSTA are introduced in the case study while SingleOrg-SSTA, SC-TSTA and SD-TSTA are implemented on Force.com. In addition, VP model is also implemented in Force.com.

## 9.1 Experiment - STA Online Shopping System

A STA online shopping system is introduced and built to illustrate STA models including requirements, implementation and customizations. Three STA models, SigleOrg-SSTA, SC-TSTA and SD-TSTA, are implemented in Force.com platform. Currently, not all the STA models can be implemented in Force.com easily, for example, Force.com does not provide a way to run the same application in different instances.

### *9.1.1 STA Online Shopping System Requirements*

In this STA online shopping system, tenants, sub-tenants and end users can perform following actions:

1. Tenant and its sub-tenants can add, update, delete and sell items.
2. End users can browse, search and buy items.
3. Tenant and its sub-tenants can customize buying process. And buying process includes following styles:
   (a) Make order → shipping → pay when deliver the order

(b) Add items to cart → Make order → shipping → pay when deliver the order.

(c) Add items to cart → Make order → pay online → shipping

### *9.1.2 STA Online Shopping System Experiment*

In this experiment, an application on Force.com called STA online shopping is built. To build the application, following custom objects are created.

1. Merchandise: it is used to describe what products tenants can sell including two fields: price and quantity.

2. Cart: it is used to describe the number of products and products that end users want to buy including three fields: product id, product name and quantity.

3. Order: it is used to describe the detail information about the products that end users have bought including four fields: product id, product name, quantity and customer id.

4. Payment: it is used to describe how the end users pay their orders including pay online and pay when deliver the order including two fields: pay online and pay when deliver. In addition, another custom object is created to support online payment that includes credit card number, expire date and billing address.

5. Shipping: it is used to describe how the end users want tenants provide their orders including shipping address.

Their relationships are showed in Figure 9.1. In Force.com, one can achieve role-based permission control by some tenant's applications such as the permissioner [9]. As this experiment employ the permissioner, the SC-TSTA is applied. In this experiment, roles, SingleOrg-SSTA tenant, SC-TSTA sub-tenant and SD-TSTA, are created to

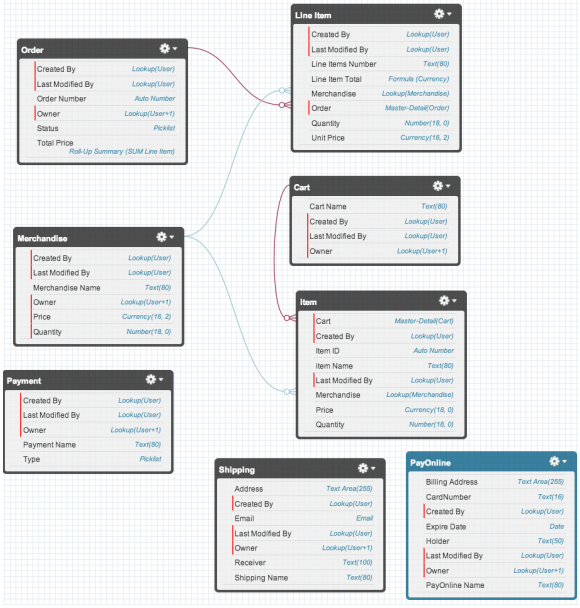



Figure 9.1: STA Online Shopping Data Model

manage all permissions. Permission sets, merchandise, cart, order, payment and shipping are built to assign corresponding permissions. According to requirement, three workflows are created:

1. W1:order → shipping → pay when deliver.
2. W2:cart → order → shipping → pay when deliver.
3. W3:cart → order → pay online → shipping.

By using the permissioner, permissions are assigned to different roles which W1 is assigned to SingleOrg-SSTA tenant, W2 is assigned to SC-TSTA sub-tenant and W3 is assigned to SD-TSTA sub-tenant. As workflow has multiple meanings in Force.com such as workflow rules [77], like events in event-driven architecture[123], and flows [76] that equals to workflow in this experiment. SingleOrg-SSTA flow example is showed in Figure 9.2. The customization options are achieved by combination of roles and their permission sets.

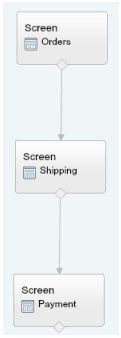



Figure 9.2: SingleOrg-SSTA Flow Example

## 9.2 STA Online Shopping System Case Study

In this case study, the STA online shopping system is built on extending the OIC architecture [101] to support the tenant and its sub-tenants, which adds subtenancy management related core system services shown in Figure 9.3. The subtenancy management services are a set of services such as subtenant information, management services, data sharing management services, upgrade and distribute services, subtenant subscriptions management services, subtenant monitoring services and subtenant billing services. Compare to tradition SaaS, STA need more tenant information such as tenant type. In this paper, the TenantType field is added to the Tenant table and the license type is also stored for the subtenants of ISV. One example of the tenant table is shown in Table 9.1.

The tenant type decides whether the tenant can use subtenancy management services. In this paper, the tenant type could be one of the following types:

1. Tenant: the customer will be assigned this type when it is an isolated organization with no relation to other organizations, or it could be an organization that has sub-tenants which makes it the parent tenant. Therefore, a tree can be used to describe the relationships between the tenant and other organizations where the tenant will be in the root of the tree. In this tree, the tenant has no

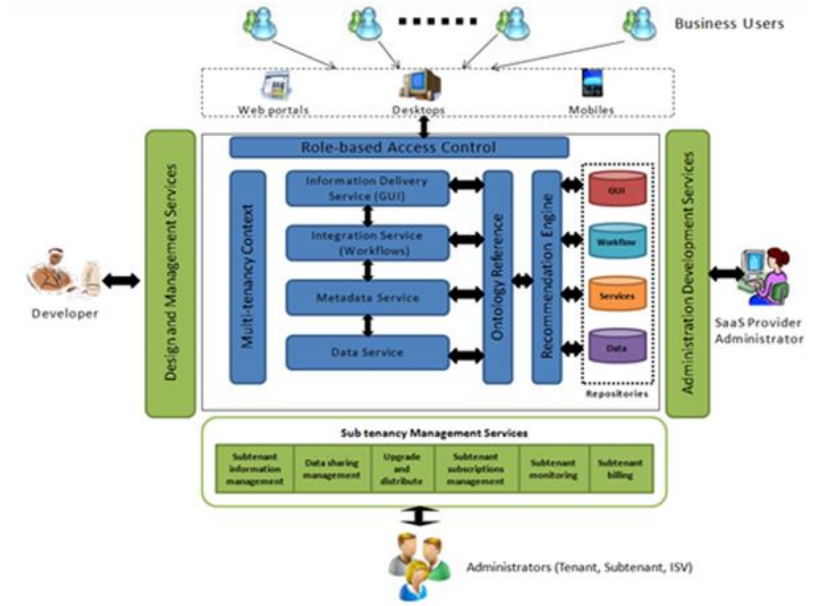



Figure 9.3: STA Architecture Overview

Table 9.1: Tenant Information

| **Tenant ID** | **Name** | **Tenant type** | **Enable Sharing** | **License type** |
|---|---|---|---|---|
| 00001 | Company ABC | Tenant | NO | - |
| 00002 | Company ABC 1 | Sub-tenant | NO | - |
| 00003 | Company ABC 2 | Sub-tenant | NO | - |
| 00004 | Company ABC 3 | Sub-tenant | NO | - |
| 00005 | Company ABC 4 | Sub-tenant | NO | - |
| 00006 | Company DEF | ISV | - | - |
| 00007 | Company HIJ | Sub-tenant | NO | Enterprise |
| 00008 | Company KLM | Sub-tenant | NO | Basic |
| 00009 | Company OPQ | Tenant | NO | - |
| 00010 | Company RST | Sub-tenant | YES | - |
| 00011 | Company UVW | Tenant | YES | - |
| 00012 | Company X | Sub-tenant | NO | - |
| 00013 | Company Y | Sub-tenant | NO | - |

parent but it may have children and grandchildren.

2. Independent Service Vendors (ISV): The customer will be assigned this type when they are partners of the SaaS provider, and use the SaaS provider platform to develop, sell, distribute, and support their SaaS applications.

3. Sub-tenants: This type can be assigned to different type of customers such as:

   (a) An external organization that has a sharing relationship with another tenant.

   (b) An internal department inside the tenant's organization.

   (c) Customers of an ISV.

Each tenant type will have different sub-tenant management services, which is achieved by adding permissions for each tenant type in the SaaS application security system, the followings are examples of required roles and permissions:

1. Tenants or Subtenants Administrators: they perform the following operations:

   (a) Manage SaaS application security: the administrators are able to add new end users, assign roles, data, and make field level access control.

   (b) Customize the SaaS application: the administrators are able to customize the SaaS application based on the tenant license agreement type with the SaaS provider. For example: the SaaS provider could have three license types:

      i. Basic: it offers simple customization on small number of features.

      ii. Enterprise: it offers full customization to all the features.

      iii. Professional: it offers partial customization by the tenant. In addition, by using the subtenant management services in some multilevel

MTA models, the tenant can control the customization level of its sub-tenants that it can select the components that its sub-tenants can customize and what type of customization it can do.

(c) Enable data sharing with their sub-tenants: data sharing services should be added to the system and the administrator has the permissions to manage the data sharing with other tenants. This can be achieved by enabling or disabling the data sharing that could be implemented by adding a field called EnableSharing to the tenant data permission table shown in Table 9.1. In some STA models, the data sharing is needed between tenants and sub-tenants. When sharing data, the tenant selects the fields to be shared in each data object, and it can also select the type of sharing offered on each object or field to the other tenants. Examples of the sharing types are read only, or read and write. This can be achieved by adding SubTenantSharingPermissions table that contains each shared field and the subtenant sharing type on them shown in Table 9.2.

(d) Sub-tenant management: the tenant administrator adds sub-tenants information and links it to the tenant by using this service. A new table Tenant-Subtenants is created to store this information in the database where Table 9.3 is the relational database schema example, and Table 9.4 is the MTA database example. In addition, they can build the components and templates that are inherited by the sub-tenants in some STA models. Further, for each inherited component, the administrators can set the permissions and the level of customization that the sub-tenants have. An example of customization levels are full customization, partial customization, or no customization permissions.

(e) Distribute customization upgrades to the sub-tenants: The tenant makes customizations to its SaaS application and select some components of these customizations to push them to its sub-tenants. In a meta data driven MTA, the customizations are described by meta data tables so that the upgrade process can be achieved by simply running a database script job that copies the updated components data from the tenant to its sub-tenants. In some STA models, the sub-tenant approval is needed before the upgrade is performed. However, in other STA models, the approval is not needed and upgrade is performed automatically.

(f) Integrate SaaS application with external systems: The tenant uses this service to integrate the SaaS application with external systems. Here, external systems are systems where other SaaS providers or organizations in internal system offer applications.

2. ISV administrators: they have the same permissions of the tenants or subtenants administrators as they are able to customize the SaaS applications, manage subtenants, and to distribute customization upgrades. In addition, they also have permissions to perform following tasks:

3. Mange subtenants subscriptions: ISV needs to add extra information related to its customers subscriptions that include the start and end date of subscription, the type of subscription, license type. SubTenantSubscription table is created in the database to store this information.

4. Monitor sub-tenants: ISV is able to monitor its sub-tenants' activity on the SaaS applications. The information monitored can be related to security, performance, etc.

Table 9.2: SubTenantSharingPermissions

| **Tenant ID** | **Sub-Tenant ID** | **Object ID** | **FieldId** | **Read** | **Write** |
|---|---|---|---|---|---|
| 00001 | 00002 | 00002 | 00001 | Yes | NO |
| 00001 | 00002 | 00002 | 00002 | Yes | Yes |
| 00001 | 00002 | 00002 | 00003 | NO | NO |
| 00001 | 00002 | 00002 | 00004 | NO | NO |
| 00001 | 00002 | 00002 | 00005 | Yes | Yes |
| 00001 | 00002 | 00002 | 00006 | Yes | No |

Table 9.3: Tenant - Subtenants

| **Tenant ID** | **Sub-Tenant ID** |
|---|---|
| 00001 | 00002 |
| 00001 | 00003 |
| 00001 | 00004 |
| 00001 | 00005 |
| 00006 | 00007 |
| 00006 | 00008 |
| 00009 | 00010 |

5. Billing services: The ISV is responsible for entering the billing information and registering all payments to its subtenants. This can be achieved by adding one or more tables to store sub-tenants billing information.

Different scenarios of customization for STA models are illustrated in this section. It is assumed that the SaaS provider of STA online shopping system has a set of default templates for UI, workflow, services, and data schema and those templates

Table 9.4: Tenant- Subtenants

| Tenant ID | Sub-Tenant ID1 | Sub-Tenant ID2 | Sub-Tenant ID3 | Sub-Tenant ID4 | Sub-Tenant IDn |
|---|---|---|---|---|---|
| 00001 | 00002 | 00003 | 00004 | 00005 | ..... |
| 00006 | 00007 | 00008 | | | |
| 00009 | 00010 | | | | |

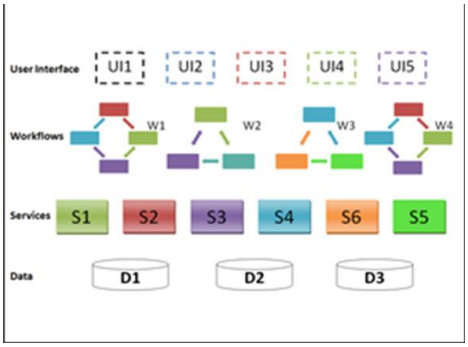



Figure 9.4: STA Provider's Default Templates

are used to create shopping applications.

Here, the SaaS provider default templates can be described as followings:

- UI template set U = {UI1, UI2, UI3, UI4, UI5}.
- Workflow template set W={W1, W2, W3, W4}.
- Service template set S= {S1, S2, S3, S4, S5, S6}.
- Data template set D= {D1, D2, D3}.

Figure 9.4 shows an example for each layer the set of the SaaS providers' default templates. It is also assumed that several companies are using the STA system to customize their SaaS application with diverse requirements. The customization options are achieved by combination of roles and their permission sets.

1. SingleOrg-SSTA Customization: In this scenario, it is assumed that tenent1 is a small company that needs a making order application, in the initial phase tenant1's administrator selects the required templates from the SaaS provider default templates. Then, the system composes the required making order application based on the selected templates. Tenant1 select the following templates:

   - $UI_1$= {UI1, UI2} $\subseteq$ UI.
   - $W_1$= {W1} $\subseteq$ W.
   - $S_1$= {S1, S2, S3, S4} $\subseteq$ S.
   - $D_1$= {D1} $\subseteq$ D.

   Tenant2 is another company that needs the adding item to cart and making order services. At same time, the company want to share an application instance as some of its employees may need to access both services. As well, some collaboration is needed between the two services. Therefore, both of the services are created in the same application instance and the customization is made according to tenant2's requirements. The administrator selects following templates to create the two required services.

   - $UI_2$= {UI2, UI3, UI4, UI5} $\subseteq$ UI.
   - $W_2$= {W3, W4} $\subseteq$ W.
   - $S_2$= {S1, S2, S3, S4, S5, S6} $\subseteq$ S.
   - $D_2$= {D2, D3} $\subseteq$ D.

   Figure 9.5 shows template choices of each layer for the SingleOrg-SSTA model.

   In this scenario, both tenants have subsets of the SaaS provider templates. However, there is a possible case that two tenants have the same subset or need

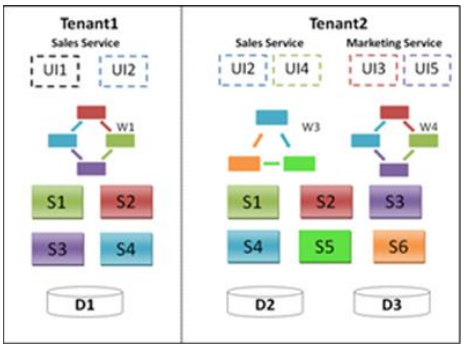



Figure 9.5: SingleOrg-SSTA Tenant's Templates

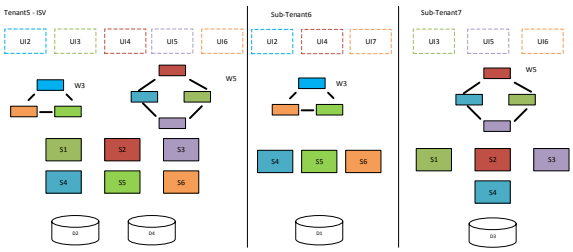


Figure 9.6: SC-STSTA Tenant's Templates

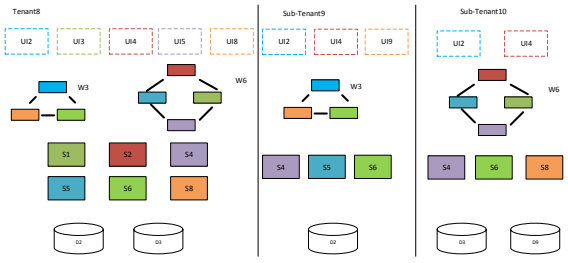


Figure 9.7: SD-STSTA Tenant's Templates

to create their own custom templates which is different from the SaaS provider templates as the two tenants represents different organizations.

2. SC-TSTA Customization: In this scenario, the tenant is an ISV, the ISV administrator customizes its SaaS application by selecting the proper templates from the SaaS provider's default templates or creates new one. Then, it starts selling subscriptions of its customized application to its subtenants, Figure 9.6 shows an example of template choices in each layer for the SC-TSTA model. The tenant5 selected some of the SaaS provider templates and added some new templates which may be one of the SaaS provider templates but with some extended functionality, the following are the tenant5 templates:

   - $UI_5$= {UI2, UI3, UI4, UI5, UI6} $\subseteq$ UI.
   - $W_5$= {W3, W5} $\cap$ W = {W3}.
   - $S_5$= {S1, S2, S3, S4, S5, S6} $\subseteq$ S.
   - $D_5$= {D2, D4} $\cap$ D = {D2}.

   Tenant templates UI6, W5, and D4 are a customized templates created by the tenant5, the sub-tenants of the tenant5 can use those templates to customize their SaaS application. At same time, sub-tenants can add their customized templates. In this scenario, sub-tenant6 used the following templates:

   - $UI_6$= {UI2, UI4, UI7} $\cap$ $UI_5$= {UI2, UI4}.
   - $W_6$= {W3} $\subseteq$ $W_5$.
   - $S_6$= {S4, S5, S6} $\subseteq$ $S_5$.
   - $D_6$= {D1} $\cap$ $D_5$ = $\emptyset$.

Sub-tenant6 adds UI7 as a customized template as it does not exist as a part of the tenants templates. All other sub-tenant6's templates are subsets of the tenant templates. In this scenario, the sub-tenant6 adds only one template in UI layer but in some scenarios it may add more custom templates to other layers if needed. As well, sub-tenant7 has the following templates:

- $UI_7$= {UI3, UI5, UI6} $\subseteq UI_5$.
- $W_7$= {W5} $\subseteq W_5$.
- $S_7$= {S1, S2, S3, S4} $\subseteq S_5$.
- $D_7$= {D3} $\cap D_5 = \emptyset$.

This scenario shows that sub-tenant6 and sub-tenant7 selects different subsets of templates for each layer and sub-tenant6 adds some customized templates but both of them do not use tenant5's data. At same time, they can also have the same templates to the tenant (ISV) or extend their selected templates based on their requirements and the type of licenses they make with the tenant. In this scenario, the tenant uses templates from the same template subsets of the SaaS provider's but it can create its custom templates on all the layers when the SaaS applications provided by the SaaS provider cannot satisfy its requirements.

3. SD-TSTA Customization: In this scenario, sub-tenants not only inherit the tenant's templates and components but also use the tenant's data. They can customize their own SaaS application from the default templates of SaaS provider or can create new custom templates. In this scenario, the tenant has to enable data sharing and give its sub-tenant the permissions on a selected data objects. As a result of enabling data sharing, the sub-tenants are able to view and use the data related to them inside SaaS application to complete its work or to

make decisions depending on. Figure 9.7 shows example of template choices in each layer for the SD-TSTA model. In this scenario, the followings are tenant8 templates:

- $UI_8$= {UI2, UI3, UI4, UI5, UI8} $\cap$ UI = {UI2, UI3, UI4, UI5}.
- $W_8$= {W3, W6} $\cap$ W = {W3}.
- $S_8$= {S1, S2, S4, S5, S6, S8} $\cap$ S ={S1, S2, S4, S5, S6}.
- $D_8$= {D2, D3} $\subseteq$ D.

Some components of tenant8 are subset of the SaaS provider's templates but it also created extra templates like UI8, W6, S8 that those templates may relate to the operations of sharing data to its sub-tenants. Sub-tenant9 uses the following templates:

- $UI_9$= {UI2, UI4, UI9} $\cap$ $UI_8$= {UI2, UI4}.
- $W_9$= {W3} $\subseteq$ $W_8$.
- $S_9$= {S4, S5, S6} $\subseteq$ $S_8$.
- $D_9$= {D2} $\subseteq$ $D_8$.

In this scenario, sub-tenant9 subscribes tenant8's templates and creates its customized templates UI9 that is not a subset of the tenant's or the SaaS provider's templates. At same time, it also uses some of data objects from tenant8. Sub-tenant10 also uses some data from tenant8 but it uses the following subset templates from the tenant8's templates:

- $UI_{10}$= {UI2, UI4} $\subseteq$ $UI_8$.
- $W_{10}$= {W6} $\subseteq$ $W_8$.
- $S_{10}$= {S4, S6, S8} $\subseteq$ $S_8$.

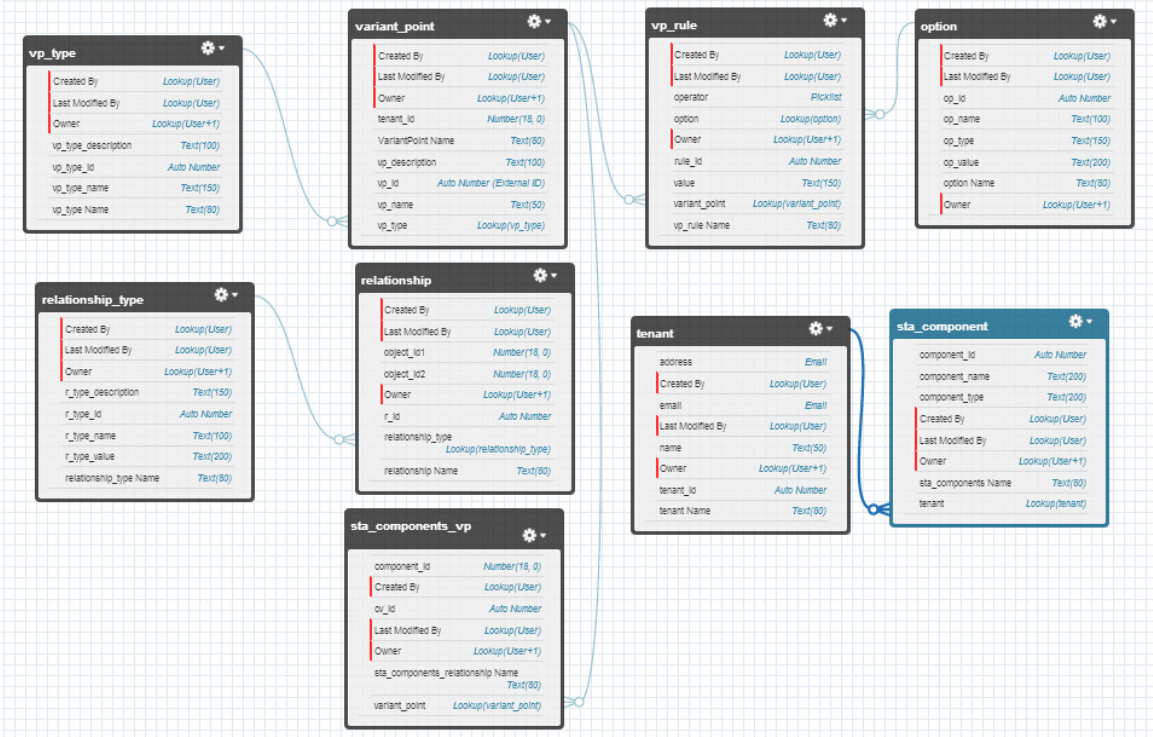



Figure 9.8: STA Customization Data Models

- $D_{10}$= {D3, D9} $\cap$ $D_8$ = {D3}.

In this scenario, sub-tenant10 inherit all templates from tenant8 except it has its own data D3. Compare previous two scenarios, one can see the major difference between CS-TSTA and SD-TSTA is sub-tenants of CS-TSTA do not use their tenant's data.

### *9.2.1 VP Experiment*

In this experiment, STA VP models are proposed. With the help of VP models, STA customization can be easily achieved. Here, an application shown in figure 9.8 on Force.com called STA variant point models. To build the application, following custom objects are created.

1. variant_point: it defines VP properties including name, id, tenant id that owns this VP, rules, options and types. What are rules, options and types are described in Section 4.3.1.

2. vp_type: it defines VP type properties and there are mainly three VP types, fixed variation points and fixed options, fixed variations but allow tenant options and flexible variation points and options that are introduced in Section 4.3.1.

3. vp_rule: it defines rule properties that describe constraints of VP and its options.

4. option: it defines option properties that describe VP options. Both VP rules and options are introduced in Section 4.3.4.

5. relationship: it defines relationship properties that describe the relationship among VPs. This paper introduces five relationships: restrict, inherit, extend, compose and implement introduced in Section 4.3.2.

6. sta_component_vp: it describes the VPs of components including GUI, service, workflow and data.

By joining different table, one can easily achieve his purpose such as searching what options a VP has. To get full list of a VP's options and rules, the algorithm introduced in Section 4.3.4 is needed. Followings are steps to deduce the VP's options and rules.

1. Step 1: Find all VPs has direct relationships with the VP need to be searched and build or add VPs and their relationships to the graph.

2. Step 2: Recursively do previous steps until meet relationship restrict or implement.

3. Step 3: Apply the deduction algorithm introduced in Section 4.3.4.

If VP, options, rules and their relationships are stored in graph database such as Neo4j [31], Step 1 and 2 can be omitted as graph information has been saved.

REFERENCES


[1] L. Aldin and S. de Cesare. A literature review on business process modelling: new frontiers of reusability. In *Enterprise Information Systems*, volume 5, pages 359–383. Taylor & Francis, 2011.

[2] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[3] Amazon. *EC2*. `http://aws.amazon.com/ec2/`.

[4] Amazon. *SimpleDB*. `http://aws.amazon.com/simpledb/`.

[5] Apache. *Apache Hadoop*. `http://hadoop.apache.org/`.

[6] Apache. *Apache Kafka*. `http://kafka.apache.org/`.

[7] Apache. *Apache Spark*. `http://spark.apache.org/`.

[8] Apache. *Apache Storm*. `http://storm.incubator.apache.org/`.

[9] I. Arkus. *The Permissioner*. `https://appexchange.salesforce.com/listingDetail?listingId=a0N30000008XYMlEAO`.

[10] O. Astrachan. Bubble sort: an archaeological algorithmic analysis. In *ACM SIGCSE Bulletin*, volume 35, pages 1–5. ACM, 2003.

[11] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold. A comparison of flexible schemas for software as a service. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 881–888. ACM, 2009.

[12] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle. Multi-tenant SOA middleware for cloud computing. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 458–465. IEEE, 2010.

[13] X. Bai, S. Lee, W. Tsai, and Y. Chen. Collaborative Web services monitoring with active service broker. In *32nd Annual IEEE International Computer Software and Applications, COMPSAC'08*, pages 84–91, 2008.

[14] X. Bai, M. Li, B. Chen, W. Tsai, and J. Gao. Cloud testing tools. In *IEEE 6th International Symposium on Service Oriented System Engineering (SOSE)*, pages 1–12, 2011.

[15] H. R. Berenji and M. Jamshidi. Fuzzy reinforcement learning for system of systems (SOS). In *FUZZ-IEEE*, pages 1689–1694, 2011.

[16] D. P. Bertsekas and J. N. Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[17] D. Bianchini and V. De Antonellis. Ontology-based integration for sharing knowledge over the web. In *CAiSE International Workshop on Data Integration over the Web*, pages 82–89, 2004.

[18] D. Borthakur. The hadoop distributed file system: Architecture and design. *"http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf"*, 2007.

[19] A. Brogi, S. Corfini, and R. Popescu. Semantics-based composition-oriented discovery of Web services. In *ACM Transactions on Internet Technology (TOIT)*, volume 8, pages 1–39, 2008.

[20] R. C. Bryce, Y. Chen, and C. J. Colbourn. Biased covering arrays for progressive ranking and composition of web services. *International Journal of Simulation and Process Modelling*, 3(1):80–87, 2007.

[21] H. Cai, N. Wang, and M. J. Zhou. A transparent approach of enabling SaaS multi-tenancy in the cloud. In *Services (SERVICES-1), 2010 6th World Congress on*, pages 40–47. IEEE, 2010.

[22] M. Chang, J. He, W. Tsai, B. Xiao, and Y. Chen. UCSOA: User-centric service-oriented architecture. In *IEEE International Conference on e-Business Engineering, 2006. ICEBE'06.*, pages 248–255, 2006.

[23] Y. Chen and W. Tsai. *Service-Oriented Computing and Web Software Integration (Fourth Edition)*. Kendall Hunt Publishing, 2014.

[24] D. Chiang, C. Lin, and M. Chen. The adaptive approach for storage assignment by mining data of warehouse management system for distribution centres. In *Enterprise Information Systems*, volume 5, pages 219–234, 2011.

[25] F. Chong and G. Carraro. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation*, pages 9–10, 2006.

[26] Y. Chou, J. Oetting, and O. Levina. Building the security foundation to embrace public Software-as-a-Service (SaaS) - security policies for SaaS data protection. In *SECRYPT*, pages 227–232, 2012.

[27] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *Software, IEEE*, 15(6):37–45, 1998.

[28] T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[29] L. Da Xu. Information architecture for supply chain quality management. In *International Journal of Production Research*, volume 49, pages 183–198, 2011.

[30] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[31] N. Developers. Neo4j. *Graph NoSQL Database [online]*, 2012.

[32] S. Dominic, R. Das, D. Whitley, and C. Anderson. Genetic reinforcement learning for neural networks. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 71–76. IEEE, 1991.

[33] L. Duan, W. Street, and E. Xu. Healthcare information systems: data mining methods in the creation of a clinical recommender system. In *Enterprise Information Systems*, volume 5, pages 169–181, 2011.

[34] S. Dustdar and W. Schreiner. A survey on Web services composition. In *International Journal of Web and Grid Services*, volume 1, pages 1–30, 2005.

[35] Elasticsearch. *Elasticsearch*. `http://www.elasticsearch.org/`.

[36] J. Espadas, A. Molina, G. Jiménez, M. Molina, R. Ramírez, and D. Concha. A tenant-based resource allocation model for scaling software-as-a-service applications over cloud computing infrastructures. *Future Generation Computer Systems*, 29(1):273–286, 2013.

[37] R. Fileto, L. Liu, C. Pu, E. Assad, and C. Medeiros. POESIA: An ontological workflow approach for composing Web services in agriculture. In *The VLDB Journal - The International Journal on Very Large Data Bases*, volume 12, pages 352–367, 2003.

[38] K. Fujii and T. Suda. Semantics-based context-aware dynamic service composition. In *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, volume 4, pages 1–31, 2009.

[39] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proceedings of the 12th international conference on World Wide Web*, pages 449–460. ACM, 2003.

[40] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[41] Google. *Google App Engine*. `https://developers.google.com/appengine/`.

[42] R. Grønmo and M. Jaeger. Model-driven semantic Web service composition. In *12th Asia-Pacific Software Engineering Conference, APSEC'05*, pages 79–86, 2005.

[43] J. Howe. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

[44] J. Hunter. Jdom makes xml easy. In *Suns 2002 Worlwide Java Developer Conference*, 2002.

[45] J. Ingvaldsen and J. Gulla. Industrial application of semantic process mining. In *Enterprise Information Systems*, volume 6, pages 139–163, 2012.

[46] E. Jackson, D. Seifert, M. Dahlweid, T. Santen, N. Bjørner, and W. Schulte. Specifying and composing non-functional requirements in model-based development. In *Software Composition*, pages 72–89, 2009.

[47] A. Java. for xml-based rpc (jax-rpc), 2005.

[48] A. Java. for xml processing (jaxp). *Sun Microsystems http://java. sun. com/webservices/jaxp/. Access April*, 2006.

[49] R. Johnson, J. Hoeller, A. Arendsen, and R. Thomas. *Professional Java Development with the Spring Framework*. John Wiley & Sons, 2009.

[50] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[51] J. Kim and R. Jain. Web services composition with traceability centered on dependency. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences, HICSS'05*, pages 89–89, 2005.

[52] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 453–456. ACM, 2008.

[53] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[54] S. Kona, A. Bansal, L. Simon, A. Mallya, and G. Gupta. USDL: A service-semantics description language for automatic service discovery and composition. In *International Journal of Web Services Research (IJWSR)*, volume 6, pages 20–48, 2009.

[55] S. Li, L. Xu, X. Wang, and J. Wang. Integration of hybrid wireless networks in cloud services oriented enterprise information systems. In *Enterprise Information Systems*, volume 6, pages 165–187, 2012.

[56] Q. Liang and S. Su. AND/OR graph and search algorithm for discovering composite Web services. In *International Journal of Web Services Research (IJWSR)*, volume 2, pages 48–67. IGI Global, 2005.

[57] C. Lin, L. Minglu, and C. Jian. Eca rule-based workflow modeling and implementation for service composition. In *IEICE transactions on information and systems*, volume 89, pages 624–630, 2006.

[58] B. Liu, S. Cao, and W. He. Distributed data mining for e-business. In *Information Technology and Management*, volume 12, pages 67–79, 2011.

[59] A. Mackey. Windows communication foundation. In *Introducing. NET 4.0*, pages 159–173. Springer, 2010.

[60] Merriam-Webster.com. Crowdsourcing - definition and more, August 31, 2012.

[61] Microsoft. *Azure*. `http://www.windowsazure.com/en-us/`.

[62] Microsoft. *System.XML*. http://msdn.microsoft.com/enus/library/system.xml(VS.71).aspx.

[63] R. Mietzner and F. Leymann. Generation of BPEL customization processes for SaaS applications from variability descriptors. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 2, pages 359–366. IEEE, 2008.

[64] R. Mietzner, F. Leymann, and T. Unger. Horizontal and vertical combination of multi-tenancy patterns in service-oriented applications. In *Enterprise Information Systems*, volume 5, pages 59–77, 2011.

[65] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25. IEEE Computer Society, 2009.

[66] Myexperiment. *My Experiment*. http://www.myexperiment.org/.

[67] M. Naor and U. Wieder. A simple fault tolerant distributed hash table. In *Peer-to-Peer Systems II*, pages 88–97. Springer, 2003.

[68] National Research Council (US) Committee on the FORCEnet Implementation Strategy. *FORCEnet implementation strategy*. Natl Academy Pr, 2005.

[69] S. Oh, H. Yeom, and J. Ahn. Cohesion and coupling metrics for ontology modules. In *Information Technology and Management*, volume 12, pages 81–96, 2011.

[70] B. Orriëns, J. Yang, and M. Papazoglou. Model driven service composition. In *Service-Oriented Computing - ICSOC 2003*, pages 75–90, 2003.

[71] E. Ort and B. Mehta. Java architecture for xml binding (jaxb). *Sun Developer Network*, 2003.

[72] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[73] X. Peng, M. A. Babar, and C. Ebert. Collaborative software development platforms for crowdsourcing. *IEEE software*, 31(2):30–36, 2014.

[74] R. Rai, G. Sahoo, and S. Mehfuz. Securing software as a service model of cloud computing: Issues and solutions. *arXiv preprint arXiv:1309.2426*, 2013.

[75] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. In *IEEE Transactions on Software Engineering*, volume 27, pages 58–93, 2001.

[76] Salesforce.com. *Flows*. https://help.salesforce.com/HTViewHelpDoc?id=vpm_designer_overview.htm&language=en_US.

[77] Salesforce.com. *Workflow Rules.* `https://help.salesforce.com/apex/HTViewHelpDoc?id=creating_workflow_rules.htm&language=en`.

[78] J. Schroeder. Future combat systems. *Department OF The Army Washington DC*, 2001.

[79] C. Shen and C. Chou. Business process re-engineering in the logistics industry: a study of implementation, success factors, and performance. In *Enterprise Information Systems*, volume 4, pages 61–78, 2010.

[80] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[81] D. Smiley and E. Pugh. *Solr 1.4 Enterprise Search Server*. Packt Publishing Ltd, 2009.

[82] E. Smith. Continuous testing. In *Proceedings of the 17th International Conference on Testing Computer Software*, 2000.

[83] V. Stantchev and M. Malek. Addressing web service performance by replication at the operating system level. In *Internet and Web Applications and Services, 2008. ICIW'08. Third International Conference on*, pages 696–701. IEEE, 2008.

[84] H. Sun, X. Wang, B. Zhou, and P. Zou. Research and implementation of dynamic Web services composition. In *Advanced Parallel Processing Technologies*, pages 457–466, 2003.

[85] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su. Software as a service: Configuration and customization perspectives. In *Congress on Services Part II, 2008. SERVICES-2. IEEE*, pages 18–25. IEEE, 2008.

[86] A. Technology, Logistics, and USJFCOM. Joint battle management command and control (jbmc2) roadmap. Technical report, Office of the Under Secretary of Defenseand U.S. Joint Forces Command, 2004.

[87] V. Tosic, B. Esfandiari, B. Pagurek, and K. Patel. On requirements for ontologies in management of Web services. In *Web services, E-business, and the semantic web*, pages 237–247, 2002.

[88] V. Tosic, K. Patel, and B. Pagurek. WSOL - Web service offerings language. In *Web Services, E-Business, and the Semantic Web*, pages 57–67, 2002.

[89] E. Truyen, N. Cardozo, S. Walraven, J. Vallejos, E. Bainomugisha, S. Günther, T. D'Hondt, and W. Joosen. Context-oriented programming for customizable SaaS applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 418–425. ACM, 2012.

[90] W. Tsai. Service-oriented system engineering: a new paradigm. In *IEEE International Workshop on Service-Oriented System Engineering, SOSE 2005*, pages 3–6, 2005.

[91] W. Tsai, Z. Cao, X. Wei, R. Paul, Q. Huang, and X. Sun. Modeling and simulation in service-oriented software development. In *Simulation*, volume 83, pages 7–32, 2007.

[92] W. Tsai, Q. Huang, J. Xu, Y. Chen, and R. A. Paul. Ontology-based dynamic process collaboration in service-oriented architecture. In *IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2007*, pages 39–46, 2007.

[93] W. Tsai, Y. Huang, and X. Bai. Grapevine model for template recommendation and generation in SaaS applications. *Arizona State University, Tempe, AZ, USA*, 2011.

[94] W. Tsai, Y. Huang, X. Bai, and J. Gao. Scalable architectures for SaaS. In *2012 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 112–117. IEEE, 2012.

[95] W. Tsai, Y. Huang, and Q. Shao. EasySaaS: A SaaS development framework. In *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–4. IEEE, 2011.

[96] W. Tsai, Y. Huang, and Q. Shao. Testing the scalability of SaaS applications. In *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–4. IEEE, 2011.

[97] W. Tsai, W. Li, B. Esmaeili, and W. Wu. Model-driven tenant development for PaaS-based SaaS. In *2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 821–826. IEEE, 2012.

[98] W. Tsai, R. Paul, H. Huang, B. Xiao, and Y. Chen. Semantic interoperability and its verification & validation in c2 systems. Technical report, DTIC Document, 2005.

[99] W. Tsai, R. Paul, B. Xiao, Z. Cao, and Y. Chen. PSML-S: A Process Specification and Modeling Language for Service-Oriented Computing. In *Software Engineering and Applications*, pages 160–167, 2005.

[100] W. Tsai, Q. Shao, Y. Huang, and X. Bai. Towards a scalable and robust multi-tenancy SaaS. In *Proceedings of the Second Asia-Pacific Symposium on Internetware*. ACM, 2010.

[101] W. Tsai, Q. Shao, and W. Li. Oic: Ontology-based intelligent customization framework for SaaS. In *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8. IEEE, 2010.

[102] W. Tsai, B. Xiao, R. Paul, and Y. Chen. Consumer-centric service-oriented architecture: a new approach. In *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006*, pages 175–180, 2006.

[103] W. Tsai, D. Zhang, P. Raymond, and Y. Chen. Stochastic voting algorithms for Web services group testing.

[104] W. Tsai and P. Zhong. Multi-Tenancy and Sub-Tenancy Architecture in Software-as-a-Service (SaaS). In *8th international Symposium on service-Oriented System Engineering (SOSE), Oxford, 2014*.

[105] W. Tsai, P. Zhong, X. Bai, and J. Elston. Dependency-guided service composition for user-centric SOA. In *IEEE International Conference on e-Business Engineering, ICEBE'09*, pages 149–156. IEEE, 2009.

[106] W. Tsai, P. Zhong, X. Bai, and J. Elston. Role-based trust model for community of interest. In *IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8, 2009.

[107] W. Tsai, P. Zhong, X. Bai, and J. Elston. Dependence-guided service composition for user-centric soa. *Systems Journal, IEEE*, PP(99):1–11, 2013.

[108] W. Tsai, P. Zhong, J. Balasooriya, X. Chen, Y.and Bai, and J. Elston. An approach for service composition and testing for cloud computing. In *in 10th International Workshop on Assurance in Distributed Systems and Networks (ADSN)*, pages 631–636, March 2011.

[109] W. Tsai, P. Zhong, and Y. Chen. Tenant-centric sub-tenancy architecture in software-as-a-service. *In CAAI Transactions on Intelligence Technology, Volume 1, Issue 2, Pages 150-161*, 2016.

[110] W. Tsai, P. Zhong, J. Elston, X. Bai, and Y. Chen. Service replication strategies with mapreduce in clouds. In *2011 10th International Symposium on Autonomous Decentralized Systems (ISADS)*, pages 381–388. IEEE, 2011.

[111] W. Tsai, P. Zhong, J. Elston, Y. Chen, and X. Bai. Ontology-Based Dependency-Guided Service Composition for User-Centric SOA. In *SEKE*, pages 462–467, 2010.

[112] W. Tsai, X. Zhou, R. A. Paul, Y. Chen, and X. Bai. A coverage relationship model for test case selection and ranking for multi-version software. In *High Assurance Services Computing*, pages 285–311. Springer, 2009.

[113] R. Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework*. Apress, 2008.

[114] E. Vysniauskas and L. Nemuraite. Transforming ontology representation from owl to relational database. In *Information Technology and Control*, volume 35, pages 333–343, 2006.

[115] K. Wang, X. Bai, J. Li, and C. Ding. A service-based framework for pharmacogenomics data integration. In *Enterprise Information Systems*, volume 4, pages 225–245, 2010.

[116] P. Wang, K. Chao, C. Lo, and R. Farmer. An evidence-based scheme for Web service selection. In *Information Technology and Management*, volume 12, pages 161–172, 2011.

[117] J. Warfield. A proposal for systems science. In *Systems Research and Behavioral Science*, volume 20, pages 507–520, 2003.

[118] D. L. Webber and H. Gomaa. Modeling variability in software product lines with the variation point model. *Science of Computer Programming*, 53(3):305–331, 2004.

[119] J. Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 217–218. ACM, 2012.

[120] C. D. Weissman and S. Bobrowski. The design of the Force.com multitenant internet application development platform. In *SIGMOD Conference*, pages 889–896, 2009.

[121] B. Wetzstein, P. Leitner, F. Rosenberg, S. Dustdar, and F. Leymann. Identifying influential factors of business process performance using dependency analysis. In *Enterprise Information Systems*, volume 5, pages 79–98, 2011.

[122] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[123] Wikipedia. *Event Driven Architecture*. `http://en.wikipedia.org/wiki/Event-driven_architecture`.

[124] Wikipedia. *Infrastructure As a Service*. `http://en.wikipedia.org/w/index.php?title=Infrastructure_as_a_service&oldid=341154594`.

[125] Wikipedia. *Platform as a service*. `http://en.wikipedia.org/wiki/Platform_as_a_service`.

[126] Wikipedia. *Software as a service*. `http://en.wikipedia.org/w/index.php?title=Software_as_a_service&oldid=578705617`.

[127] Å. Wikström. *Functional programming using Standard ML*. Prentice Hall International (UK) Ltd., 1987.

[128] P. H. Winston and B. K. Horn. *Lisp*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[129] Workday. *Workday's technology strategy*. `http://www.workday.com/landing_page/workday_technology_strategy_whitepaper.php`.

[130] L. Wu, S. K. Garg, and R. Buyya. SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 195–204. IEEE, 2011.

[131] E. Xu, M. Wermus, and D. Bauman. Development of an integrated medical supply information system. In *Enterprise Information Systems*, volume 5, pages 385–399, 2011.

[132] L. Xu. Enterprise systems: state of the art and future trends. In *Industrial Informatics, IEEE Transactions on Industrial Informatics*, number 99, pages 630–640, 2011.

[133] E. W. Zegura, M. H. Ammar, Z. Fei, and S. Bhattacharjee. Application-layer anycasting: a server selection architecture and use in a replicated web service. *Networking, IEEE/ACM Transactions on*, 8(4):455–466, 2000.

[134] H. Zhao and H. Tong. A dynamic service composition model based on constraints. In *Sixth International Conference on Grid and Cooperative Computing, GCC 2007*, pages 659–662, 2007.

[135] X. Zheng and Y. Yan. An efficient syntactic Web service composition algorithm based on the planning graph model. In *IEEE International Conference on Web Services, ICWS'08*, pages 691–699, 2008.

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