Available online at www.sciencedirect.com

ScienceDirect

CAAI Transactions on Intelligence Technology 1 (2016) 150–161
http://www.journals.elsevier.com/caai-transactions-on-intelligence-technology/

Original Article

Tenant-centric Sub-Tenancy Architecture in Software-as-a-Service

Wei-Tek Tsai*, Peide Zhong, Yinong Chen

School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85287, USA

Available online 13 October 2016

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 to allow a tenant application to have its own sub-tenants, where the tenant application acts like a SaaS infrastructure. In other words, MTA is extended to STA (Sub-Tenancy Architecture). In STA, each tenant application needs not only to develop its own functionalities, but also to prepare an infrastructure to allow its sub-tenants to develop customized applications. This paper applies Crowdsourcing as the core to STA component in the development life cycle. In addition, to discovering adequate fit tenant developers or components to help build and compose new components, dynamic and static ranking models are proposed. Furthermore, rank computation architecture is presented to deal with the case when the number of tenants and components becomes huge. Finally, experiments are performed to demonstrate that the ranking models and the rank computation architecture work as design.

Keywords: SaaS; MTA; STA; Tenant; Sub-tenant; Crowdsourcing; Ranking

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 [1], 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 that is based stored in the SaaS infrastructure. MTA is often integrated with databases and 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 [2] introduced a STA (Sub-Tenancy Architecture) to allow tenants to offer services for sub-tenant developers for customizing their applications. As SaaS component building often needs different technologies such as frontend UI and database, the tenant or sub-tenant developers are often not good at those technologies. Therefore, it can be difficult for them to build SaaS components from the scratch. Hence, this paper introduces Crowdsourcing

* Corresponding author.
E-mail addresses: wtsai@asu.edu (W.T. Tsai), Peide.Zhong@asu.edu (P. Zhong), Yinong.Chen@asu.edu (Y. Chen).
Peer review under responsibility of Chongqing University of Technology.

http://dx.doi.org/10.1016/j.trit.2016.08.002

to make use of the public wisdom and assign tasks to specific experts who are good at those required technologies. To help find adequate tenants, we developed models in this paper. The rest of the paper is organized as follows. Section 2 reviews related SaaS models and technologies; Section 3 analyzes life cycles of tenant-centric application development; Section 4 introduces component and tenant rank; Section 5 presents feature implementation selection model; Section 6 describes rapid application building process. Section 7 presents the experiment that illustrates the rank models, and Section 8 concludes the paper.

2. Related work

2.1. MTA in SaaS

In the current practice, MTA are implemented via the following ways:

1) Integration with Databases: Weissman and Bobrowski proposed a database-based and metadata-driven architecture to implement MTA in Ref. [3]. In Ref. [3], where heavy indexing was 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 shared the same database, a flexible schema design was used. Aulbach [4] developed 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, the applications can be transformed into a MTA SaaS rapidly with minimum changes to the original applications. Cai [5] 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 (Service-Oriented Architecture) [6]. SaaS domain knowledge is separated from SaaS infrastructure to facilitate different domains. EasySaaS [7] proposed a development framework to simplify SaaS development by harnessing both SOA and SaaS domain ontology. Azeez [8] proposed an architecture for achieving service-oriented MTA that enabled users to run their services and other SOA artifacts in a MTA service-oriented framework as well as provided an environment to build MTA applications. As this MTA is based on SOA, it can harness both middleware and SOA technology.
4) PaaS-based approach: The SaaS developers use an existing PaaS such as GAE [9], Amazon EC2 [10], or Microsoft Azure [11] 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 [12] proposed a model-driven approach on a PaaS to develop SaaS.
5) Object-oriented approach: Workday [13] proposed an object-oriented approach for tenant application development and configuration. In addition [13], also conducted a study on MTA models, specifically it addressed the architecture of MTA and its impact on customization, scalability, and security.

2.2. Crowdsourcing

The purpose of Crowdsourcing is to make use of public wisdom and let the crowd with domain knowledge to complete specific tasks. Howe first defined the term “crowdsourcing” in a companion blog post [14]. Merriam-Webster [15] 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 the traditional employees or suppliers. Kittur in [16] 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 [17] provided an overview of current technologies for crowdsourcing.

2.3. Variation point

Variation points are locations where variation occurs, and variants are the alternatives that can be selected. Software product families introduce variability management to deal with these differences by handling variability. Kang [18] described a method for discovering commonality among different software systems. Coplien [19] presented how to perform domain engineering by identifying the commonalities and variabilities within a family of products. Webber [20] described a systematic method for providing components that could be extended through variation points, which allowed the user or application engineer to extend components at pre-specified variation points to create more flexible set of components. Mietzner [21] presented a variability descriptor and described that its transformation into a WS-BPEL process model to guide customizations. In addition, Mietzner [22] explained how variability modeling techniques could support SaaS providers in managing the variability of SaaS applications and proposed using explicit variability models to derive customization for individual SaaS tenants.

2.4. 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 [23] proposed a SaaS maturity model that classifies SaaS into four levels including ad-hoc/custom, customizable or configurable, multi-tenant efficient, and scalable. Tsai introduced ontology into SaaS to help customize applications [24]. A SaaS tenant application has components from four layers: GUI, workflow, services and data management. 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 [25] proposed a methodology framework to help SaaS vendors to plan and evaluate their capabilities and strategies for service configuration and customization. Truyen [26] 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 applications. Tsai [27,28] proposed a dependency-guided user centric service composition approach.

3. Life cycles of tenant-centric application development

The purpose of tenant-centric application development is to help tenants to find experts to develop components with domain knowledge requirements and to facilitate components creation and reuse. Generally, there are six steps the application development, as shown in Fig. 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:
    a) Feature requirements: they are all required features that tenants want to implement.

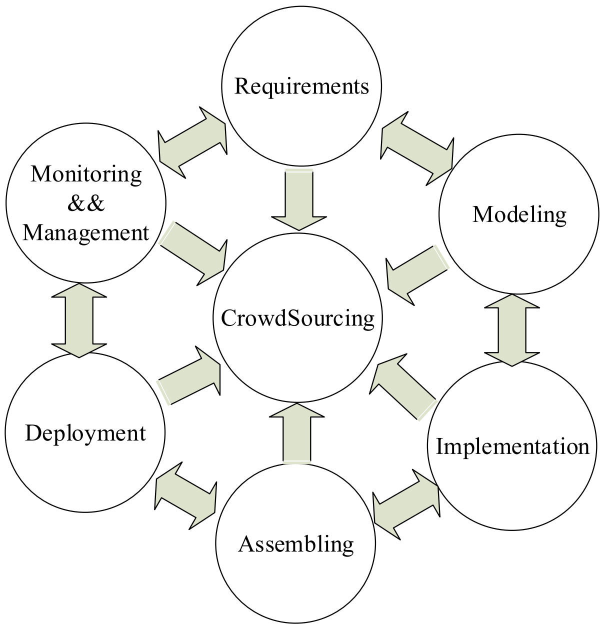


Fig. 1. Application development life cycle.

    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 test cases that modeling step proposes.
4) Assembling: it is the process that integrates all tenant applications, features, services and performs 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.
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 ways is through community of interests (COIs) shown in Fig. 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.

4. 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 [29,30] and Neural network [31,32] 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.

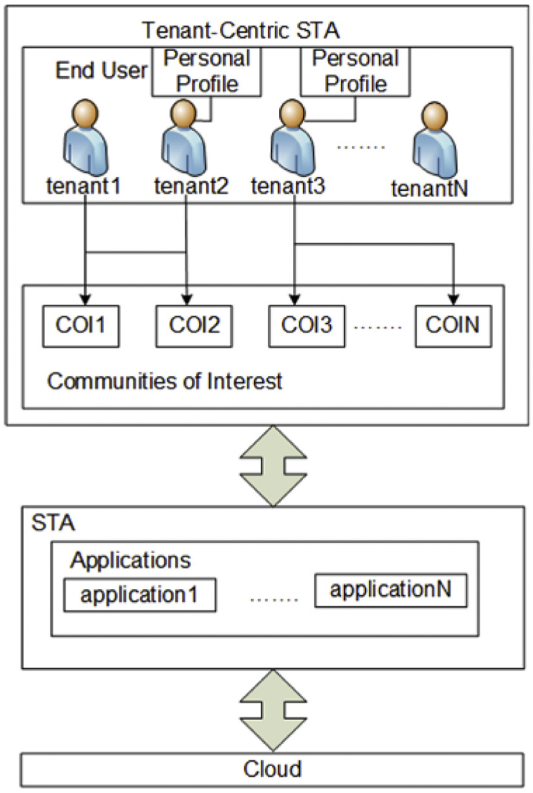


Fig. 2. Community of interests example.

4.1. 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 Fig. 3.

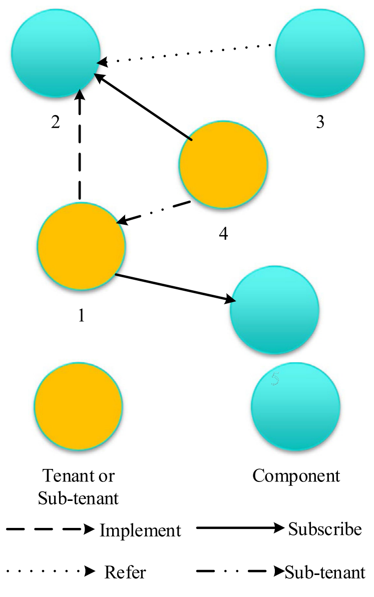


Fig. 3. Static ranking example.

In Fig. 3, one can see followings:

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 [33], tenants, sub-tenants and components can receive scores called static scores. Comparing to page rank model, this static ranking model has following characteristics:

1) There are two types of nodes in the relationship graph, tenants or sub-tenants and components, while there is only one page in page graph.
2) There are three types of links, implementation, subscription and reference.

To accommodate those characteristics, a simple revised page rank model is introduced in Equation (1).

$$\begin{cases} R(r) = c \times \sum_{s \in B_{r'}} \frac{R(s)}{N_s} \\ R(u) = \alpha \times \sum_{v \in B_{u'}} \frac{R(v)}{N_v} + \beta \times \sum_{w \in B_{u''}} \frac{R(w)}{N_w} + \gamma \times \sum_{w \in B_{u'''}} \frac{R(x)}{N_x} \end{cases} \quad (1)$$

Equation (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 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) N_s, N_v, N_w, N_x represents the number of components that tenant s, v, w and x implement.
5) α, β, γ 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 equal opportunity in 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

$$\begin{cases} R'(r) = d \times \left(c \times \sum_{s \in B_{r'}} \frac{R(s)}{N_s} \right) + (1-d) \times \frac{E_1}{k} \\ R'(u) = d \times \left(\alpha \times \sum_{v \in B_{u'}} \frac{R(v)}{N_v} + \beta \times \sum_{w \in B_{u''}} \frac{R(w)}{N_w} + \gamma \times \sum_{w \in B_{u''\prime}} \frac{R(x)}{N_x} \right) + (1-d) \times \frac{E_2}{l} \end{cases} \tag{2}$$

relationships with all other tenants or sub-tenants in STA. Therefore, Equation (1) can be revised to Equation (2).

In Equation (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. k and l are the column numbers of E_1 and E_2 respectively.

4.2. Dynamic ranking model

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

1) Component rank: there are two merit factors, importance (I) and goodness (G), to describe a component.
2) Tenant rank: same to component rank, importance (I) and goodness (G) are used to describe a tenant.

Fig. 4 shows how to calculate component's importance and goodness. From Fig. 4, one can see followings:

1) There are two tenants that 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.

Fig. 5 shows how to calculate tenant's importance and goodness. From Fig. 5, 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.
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 Figs. 4 and 5, one can observe the followings:

1) The more good tenants implement the component, the more important the component is; the more good tenants subscribe or components refer to the component, the more goodness the component has.
2) The more important the tenant becomes sub-tenant of the given tenant, the more goodness the tenant has; more good components the tenant subscribes and implements, more goodness the tenant has.

Formally calculating ranks is shown in Equation (3), and it can be describes as followings:

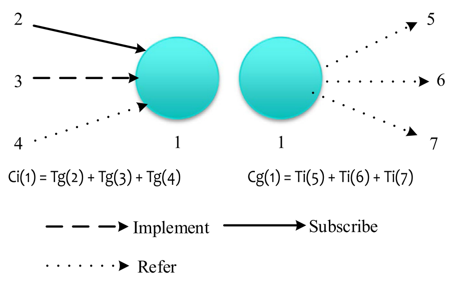


Fig. 4. Component rank example.

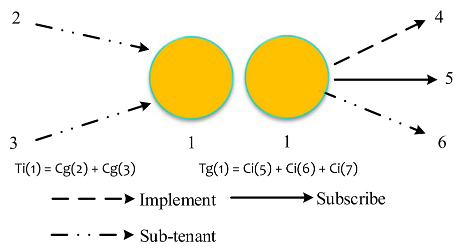


Fig. 5. Tenant rank example.

$$\begin{cases} C_I = \sum_{i=1}^{n} C_{G_i} \, and \, C_G = \alpha * \sum_{i=1}^{n} T_{I_i} + \beta * \sum_{i=1}^{m} T'_{I_i} + \gamma * \sum_{i=1}^{k} C_{I_i} \\ T_I = \sum_{i=1}^{l} T_{G_i} and \, T_G = \alpha \times \sum_{i=1}^{j} C_{G_i} + \beta \times \sum_{i=1}^{o} C'_{G_i} + \gamma \times \sum_{i=1}^{p} T_{G_i} \end{cases} \tag{3}$$

In the upper part of Equation (3), a component's importance and goodness score is introduced.

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 implementing the component.
   b) T'_{I_i} are tenant's importance scores introduced by tenants subscribing the component.
   c) C_{I_i} are component C's goodness scores introduced by components that the component has reference relationships with.

In the lower part of Equation (3), a tenant's importance and goodness score are 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 implementations.
   b) C'_{I_i} are component's importance scores introduced by the given tenant subscriptions.
   c) T_{I_i} are tenant's importance scores introduced by tenants that the given tenant has sub-tenant relationships with.

From the Equation (3), one can observe the 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 obtain importance and goodness of tenants and components. In other words, tenants and components obtain new values of importance and goodness in each iteration.
4) The importance is computed from the current goodness weights, which are being computed from the previous importance weights.
5) It can be proved that importance and goodness of tenant and application converge [34].

```
Algorithm 1: Tenant and Application Rank
   Input: n tenants T and m components C
   Output: Goodness and importance of tenants and
           components
 1 Initialize for all c ∈ C, $C_i = C_g = \frac{1}{n}$
 2 e = 0.000001;
 3 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
 4     foreach components in C do
 5         $C_I = \sum_{i=1}^{n} T_{G_i}$
           $C_G = \alpha * \sum_{i=1}^{n} T_{I_i} + \beta * \sum_{i=1}^{m} T'_{I_i} + \gamma * \sum_{i=1}^{k} C_{I_{C_i}}$
           $C_I = C_I / c$
           // normalize $C_I$ such that $\sum_{i=1}^{m} (C_I/c)^2 = 1$
 6         $C_G = C_G / d$
           // normalize $C_G$ such that $\sum_{i=1}^{m} (C_G/d)^2 = 1$
 7     foreach tenants in T do
 8         $T_I = \sum_{i=1}^{l} T_{G_i}$
           $T_G = \alpha \times \sum_{i=1}^{j} C_{G_i} + \beta \times \sum_{i=1}^{o} C'_{G_i} + \gamma \times \sum_{i=1}^{p} T_{G_i}$
           $T_I = T_I / e$
           // normalize $T_I$ such that $\sum_{i=1}^{n} (T_I/e)^2 = 1$
 9         $T_G = T_G / f$
           // normalize $T_G$ such that $\sum_{i=1}^{n} (T_G/f)^2 = 1$
10 return all $A_I$, $A_G$, $T_I$ and $T_G$
```

Base on the Equation (3), Algorithm 1 is introduced, which 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.

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 1:

1) Start with each component having an Importance score and Goodness score of $\frac{1}{n}$.
2) For each component, run the Component Importance Update; for tenants, run the Tenant Importance Update.
3) For each component, run the Component Goodness Update; for each tenant, run the Tenant Goodness Update.
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.

4.3. Rank computation architecture

In the tenant and component ranking algorithm, there are two types of scores, static scores and dynamic scores. However, the number of tenants and components can become huge. Therefore, it is difficult to calculate both static and dynamic scores in realtime. As a result, a computation architecture is introduced to calculate both static and dynamic scores shown in Fig. 6.

From Fig. 6, one can observe the followings:

1) There are two layers to compute goodness and importance, batch layer and real-time layer. In this paper, batch layer means STA does the calculation after certain 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 a large number of tenants or components and obtain 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, well-established big data frameworks such as hadoop [35] and spark [36] can be applied to accelerate computation.
3) Realtime based calculation can do calculation fast but can only obtain proximate result. Only those tenants and components need to update their scores that have relationships to tenants who implement or subscribe components and become sub-tenant to other tenants. Therefore, dynamic ranking model is applied in this case. 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

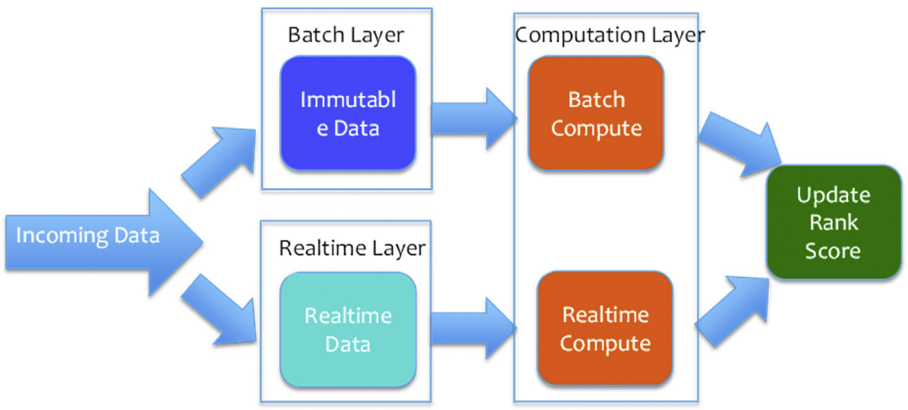


Fig. 6. Rank computation architecture.

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 numbers of tenants and components are huge. Therefore, key words based search engine such as solr [37] and elastic search [38] can be introduced when searching tenant and component candidates to find augmenting information. In addition, graph databases such as neo4j [39] can be used to save subgraph information of the base set. In realtime environment, the time of computation must be short. Hence, well-establised realtime big data framework such as Apache Kafka [40] and storm [41] can be integrated.

4) Batch based calculation can obtain static scores of all tenants and components. Realtime based calculation can obtain dynamic scores of tenants and components have changes. To integrate both static scores and dynamic scores, Equation (4) is applied. In Equation (4), α, β, γ and ξ are weights to make static scores and dynamic scores comparable that can be adjusted.

$$S(i)=\begin{cases}\alpha\times R(i)+\beta\times(\gamma\times C_I+\xi\times C_G) & \text{if } i \text{ is a component}\\ \alpha\times R(i)+\beta\times(\gamma\times T_I+\xi\times T_G) & \text{if } i \text{ is a tenant}\end{cases} \tag{4}$$

5. 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 adequate tenants to implement features (X) of a component (τ). 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 τ* 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 Fig. 7. In Fig. 7, from which one can observe the followings:

1) One component can be split into n features presented by $X_1, X_2 \ldots X_n$.
2) One feature can be implemented by more than one tenants.
3) One tenant can implement many features at same time.

Formal feature implementation selection model can be shown in Equation (5). From Equation (5), one can observe the followings:

1) τ* 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.

$$\begin{cases}\tau^* = argmin\left(\sum_{i=1}^{m}\sum_{j=1}^{n} t_{i,j}\times c(x_j)\right)\\ subject\ to:\\ \sum_{i=1}^{n} t_{i,j}\times x_j = 1, \sum_{i=1}^{m}\sum_{j=1}^{n} t_{i,j} = m\ and \sum_{i=1}^{m}\sum_{j=1}^{n} t_{i,j}\times t(x_j) < t\end{cases} \tag{5}$$

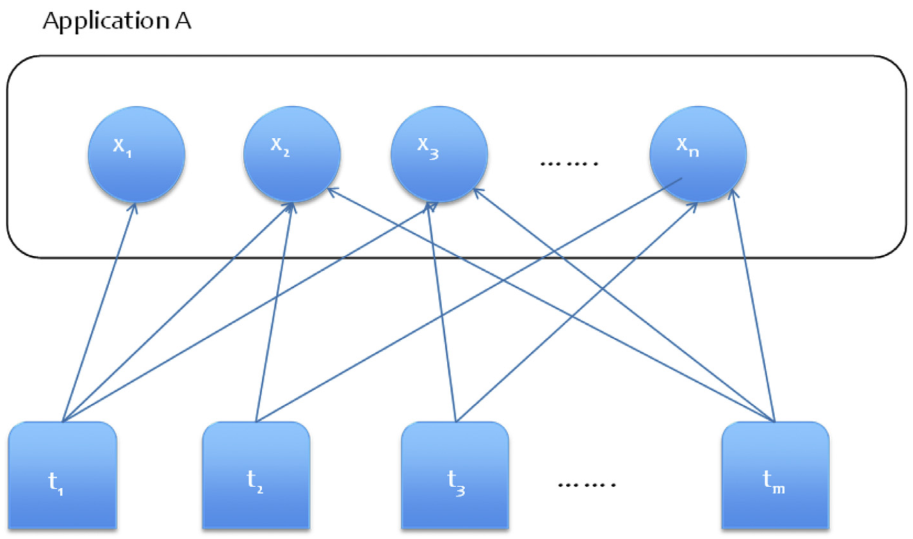


Fig. 7. Feature implementation selection model.

To provide the feature selection a solution, Algorithm 2 is introduced.

```
Algorithm 2: 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 $i \leftarrow 1$ to $m$ do
    for $j \leftarrow 1$ 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 $m_2$ ;
}
```

The basic idea of Algorithm 2 is exhausting all possible solutions and find the best solution with the minimal cost. Algorithm 2 can be described as followings:

- Algorithm 2 is a recursive algorithm and it explores every possible solutions.
- τ^* does not select n_{th} tenant to implement the m_{th} feature. So, τ^* will select the best tenant from $\{t_1, t_2, \ldots, t_{n-1}\}$.
- τ^* select n_{th} tenant for the m_{th} feature. τ^* will choose tenants from $\{t_1, t_2, \ldots, t_{n-1}\}$ for $\{f_1,\ldots,f_{m-1}\}$.
- There is no features left. τ^* is optimized.

6. Rapid application building process

This paper inherits those approaches proposed by Tsai in [42,24,43] 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 [38] and solr [37]. By combining the relevance algorithm of elastic search and solr with components' rank discussed in Section 4.2, tenants and sub-tenants can quickly discover adequate 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 and sub-tenants discover adequate application templates through key words based search engines.
2) Tenants and sub-tenants customize or extend the selected application templates. In addition, tenants and sub-tenants can publish customized applications or extended application templates so that other sub-tenants can subscribe or reuse them.

7. Experiment

Experiments are conducted and used to illustrate static and dynamic models. In static model, the relationships, implement, subscribe, reference and sub-tenant have different influence. In this experiment, implementation is considered to have the most influence and its weight is set to three. Sub-tenant is considered to have the second-most influence, and its weight is set to two. Both subscription and reference are considered to be equal and their weights are set to one. Based on these assumptions, Fig. 3 can be translated into the connected graph with weights shown in Table 1.

Table 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

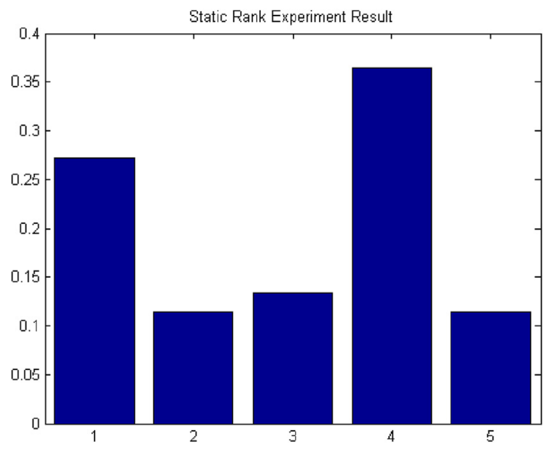
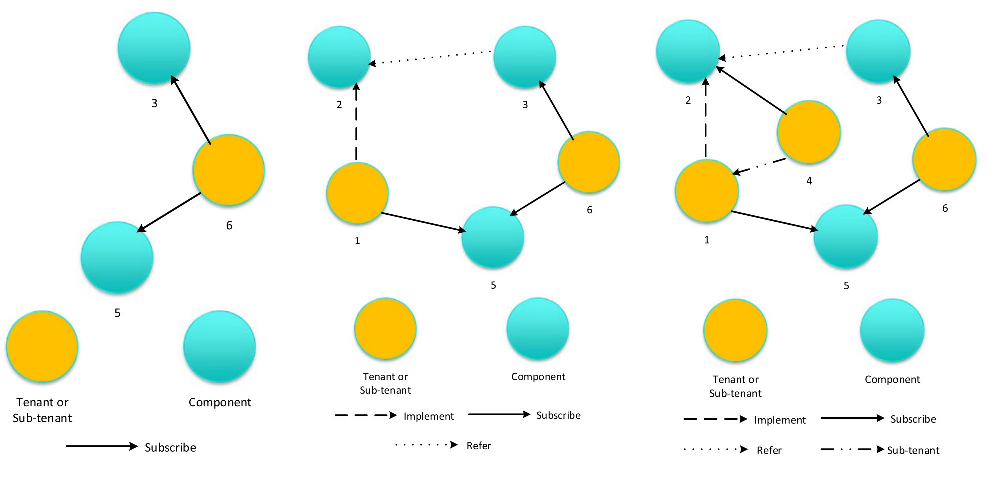


Fig. 8. Result of static rank.



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

Fig. 9. Static rank with dynamic rank update.

Table 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

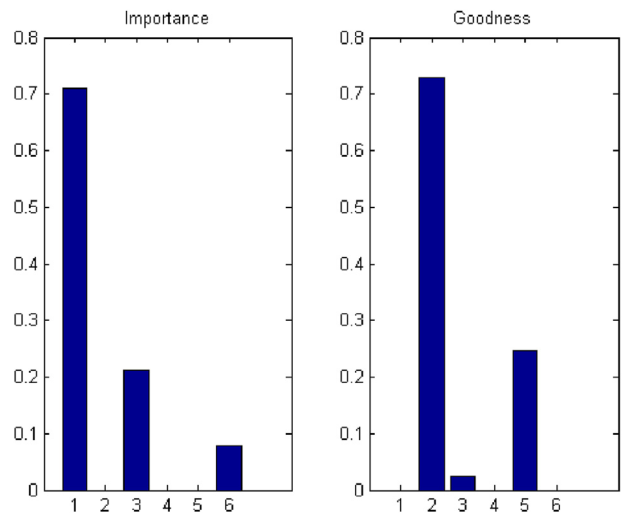


Fig. 10. Result of dynamic rank.

Applying the static model introduced in Equation (2), the result is shown in Fig. 8.

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 α, β and γ in Equation (2), it will have different static scores.

Next, one tenant subscribes to both component3 and component5. Applying dynamic model introduced by Equation (2), root set is shown in Fig. 9a. 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 Fig. 9b. To follow the same weights in static model, subgraph with weights is shown in Table 2. According to Algorithm 1, their importance and goodness scores are shown in Fig. 10.

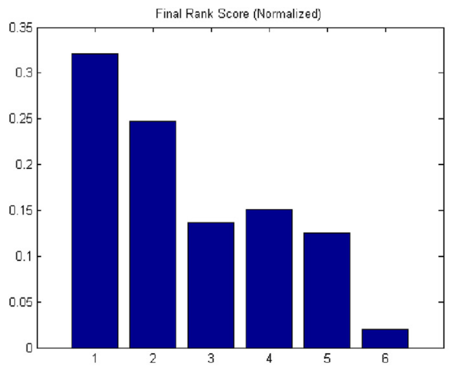
In Fig. 10, tenant1 has the highest importance score as tenant1 implements component1 and subscribes component5 where implement relationship has the highest weight according to the assumption. The component2 has highest goodness score as component2 is implemented by tenant1 with the highest importance score and referred by component3.

Combining the static and dynamic scores, the final scores are shown in Fig. 11a. Although the static score, importance score and goodness score share same weights in this experiments, they can be different based on different requirements.

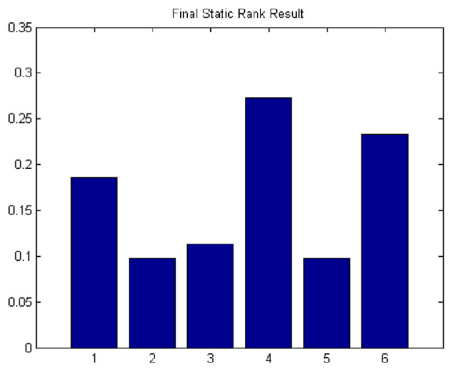
Finally, the final graph is formed by adding tenant6 and its subscriptions back to the whole graph shown in fig:finalRankExample. And its corresponding final static scores are shown in Fig. 11b. Comparing final score with final static in Fig. 11, one can observe the followings:

1) Dynamic model boosts tenants or components with the 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 worked as expect, and it illustrates rank computation architecture works well by applying static model to batch layer and dynamic model to realtime layer in Fig. 6.



(a) Result of Final Rank



(b) Result of Final Static Rank

Fig. 11. Final score vs final static score.

8. Conclusion

This paper proposed a tenant centric STA to assist tenants to rapidly 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 were developed to rank tenants and components. Furthermore, a ranking architecture is presented to handle the cases when the number of tenants and components becomes huge. Finally, experiments were conducted to demonstrate that the static model, dynamic model and rank computation architecture. . The results showed that they work as expected.

Acknowledgment

The work is supported by Fujitsu Laboratories.

References

[1] Wikipedia, Software as a Service. http://en.wikipedia.org/w/index.php?title=Software_as_a_service&oldid=578705617.
[2] W.T. Tsai, P. Zhong, Multi-tenancy and sub-tenancy architecture in software-as-a-service (saas), in: 8th IEEE International Symposium on Service Oriented System Engineering, SOSE 2014, Oxford, United Kingdom, April 7–11, 2014, 2014, pp. 128–139.
[3] C.D. Weissman, S. Bobrowski, The design of the Force.com multitenant internet application development platform, in: SIGMOD Conference, 2009, pp. 889–896.
[4] S. Aulbach, D. Jacobs, A. Kemper, 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, ACM, 2009, pp. 881–888.
[5] H. Cai, N. Wang, M.J. Zhou, A transparent approach of enabling SaaS multi-tenancy in the cloud, in: 2010 6th World Congress on Services (SERVICES-1), IEEE, 2010, pp. 40–47.
[6] Y. Chen, W.T. Tsai, Service-oriented Computing and Web Software Integration, fifth ed., Kendall Hunt Publishing, 2015.
[7] W.T. Tsai, Y. Huang, Q. Shao, EasySaaS: a SaaS development framework, in: 2011 IEEE International Conference on Service-oriented Computing and Applications (SOCA), IEEE, 2011, pp. 1–4.
[8] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, P. Fremantle, Multi-tenant SOA middleware for cloud computing, in: 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD), IEEE, 2010, pp. 458–465.
[9] Google, Google App Engine. https://developers.google.com/appengine/.
[10] Amazon, EC2. http://aws.amazon.com/ec2/.
[11] Microsoft, Azure. http://www.windowsazure.com/en-us/.
[12] W.T. Tsai, W. Li, B. Esmaeili, W. Wu, Model-driven tenant development for PaaS-based SaaS, in: 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2012, pp. 821–826.
[13] Workday, Workday's technology strategy. http://www.workday.com/landing_page/workday_technology_strategy_whitepaper.php.
[14] J. Howe, Wired Mag. 14 (6) (2006) 1–4.
[15] Merriam-Webster.com, Crowdsourcing - Definition and More, August 31, 2012.
[16] A. Kittur, E.H. Chi, B. Suh, Crowdsourcing user studies with mechanical turk, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2008, pp. 453–456.
[17] X. Peng, M.A. Babar, C. Ebert, IEEE Softw. 31 (2) (2014) 30–36.
[18] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, Feature-oriented Domain Analysis (Foda) Feasibility Study, tech. rep., DTIC Document, 1990.
[19] J. Coplien, D. Hoffman, D. Weiss, IEEE Softw. 15 (6) (1998) 37–45.
[20] D.L. Webber, H. Gomaa, Sci. Comput. Program. 53 (3) (2004) 305–331.
[21] R. Mietzner, F. Leymann, Generation of BPEL customization processes for SaaS applications from variability descriptors, in: IEEE International Conference on Services Computing, 2008. SCC'08, vol. 2, IEEE, 2008, pp. 359–366.
[22] R. Mietzner, A. Metzger, F. Leymann, 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, IEEE Computer Society, 2009, pp. 18–25.
[23] F. Chong, G. Carraro, Architecture Strategies for Catching the Long Tail, MSDN Library, Microsoft Corporation, 2006, pp. 9–10.
[24] W.T. Tsai, Q. Shao, W. Li, Oic: ontology-based intelligent customization framework for SaaS, in: 2010 IEEE International Conference on Service-oriented Computing and Applications (SOCA), IEEE, 2010, pp. 1–8.
[25] W. Sun, X. Zhang, C.J. Guo, P. Sun, H. Su, Software as a service: configuration and customization perspectives, in: Congress on Services Part II, 2008. SERVICES-2. IEEE, IEEE, 2008, pp. 18–25.
[26] E. Truyen, N. Cardozo, S. Walraven, J. Vallejos, E. Bainomugisha, S. Günther, T. D'Hondt, W. Joosen, Context-oriented programming for customizable SaaS applications, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, ACM, 2012, pp. 418–425.
[27] W.T. Tsai, P. Zhong, X. Bai, J. Elston, IEEE Syst. J. 8 (3) (2014) 889–899.

[28] W.T. Tsai, Y. Huang, X. Bai, Grapevine model for Template Recommendation and Generation in SaaS Applications, Arizona State University, Tempe, AZ, USA, 2011.
[29] N.S. Altman, Am. Stat. 46 (3) (1992) 175−185.
[30] T. Cover, P. Hart, IEEE Trans. Inf. Theory 13 (1) (1967) 21−27.
[31] S. Dominic, R. Das, D. Whitley, C. Anderson, Genetic reinforcement learning for neural networks, in: International Joint Conference on Neural Networks, 1991., IJCNN-91-Seattle, vol. 2, IEEE, 1991, pp. 71−76.
[32] D.P. Bertsekas, J.N. Tsitsiklis, Neuro-dynamic programming: an overview, in: Proceedings of the 34th IEEE Conference on Decision and Control, 1995, vol. 1, IEEE, 1995, pp. 560−564.
[33] L. Page, S. Brin, R. Motwani, T. Winograd, The Pagerank Citation Ranking: Bringing Order to the Web, 1999.
[34] J.M. Kleinberg, J. ACM (JACM) 46 (5) (1999) 604−632.
[35] Apache, Apache Hadoop. http://hadoop.apache.org/.
[36] Apache, Apache Spark. http://spark.apache.org/.
[37] D. Smiley, E. Pugh, Solr 1.4 Enterprise Search Server, Packt Publishing Ltd, 2009.
[38] Elasticsearch, Elasticsearch. http://www.elasticsearch.org/.
[39] J. Webber, A programmatic introduction to neo4j, in: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, ACM, 2012, pp. 217−218.
[40] Apache, Apache Kafka. http://kafka.apache.org/.
[41] Apache, Apache Storm. http://storm.incubator.apache.org/.
[42] W.T. Tsai, P. Zhong, J. Elston, Y. Chen, X. Bai, Ontology-based dependency-guided service composition for user-centric soa, in: SEKE, 2010, pp. 462−467.
[43] W.T. Tsai, P. Zhong, J. Balasooriya, Y. Chen, X. Bai, J. Elston, An approach for service composition and testing for cloud computing, in: 10th International Workshop on Assurance in Distributed Systems and Networks (ADSN), March 2011, pp. 631−636.

Peide Zhong received the M.S. degree in 2007 in Department of Software Engeering Department of Tsinghua University, Beijing, China. He now is a Ph.D student in the department of Computer Science and Engineering of Arizona State University, Tempe, AZ, U.S.A.

Yinong Chen received his Ph.D. from the University of Karlsruhe (KIT), Germany, in 1993. He was a postdoctoral research fellow at KIT in 1993 and at the LAAS−CNRS, France in 1994. From 1994 to 2000, he was with the Wits University at Johannesburg, South Africa. He was the funding director of the Highly Dependable System Research Program at the university and a rated research fellow at South African National Science Foundation. Dr. Chen joined Arizona State University in 2001. He is a senior lecturer, a Ph.D. student advisor in the Computer Engineering program, the director of the Internet of Things and Robotics Education Laboratory, and an honors faculty in the Barrett Honors College of the university. Dr. Chen is an area editor of the Elsevier Journal: Simulation Modeling Practice and Theory, an associate editor of the International Journal of Simulation and Process Modelling, and an editorial board member of the Journal of Systems & Software. Dr. Chen's primary research interests are in service-oriented computing, Robot as a Serve, Internet of Things, and computer science education. He (co−) authored over ten books and 200 technical papers in these areas.