An Efficient Service Discovery Algorithm for Counting Bloom Filter-Based Service Registry

Shuxing Cheng, Carl K.Chang
Department of Computer Science
Iowa State University
Ames, IA 50011, USA
{scheng, chang}@cs.iastate.edu

Liang-Jie Zhang
IBM T.J. Watson Research Center
Hawthorne, NY 10532, USA
zhanglj@us.ibm.com

Abstract

The Service registry, the yellow pages of Service-Oriented Architecture (SOA), plays a central role in SOA-based service systems. The service registry has to be scalable to manage large number of services along with their requirements on storage and discovery. Based on our previous work on feature-based services quantification, we characterize services according to their diverse functional and non-functional requirements, and represent them as string formats which can be stored, probed, and indexed by efficient data structures, such as hash table and Bloom filter. Then, we propose a comprehensive service-storage solution using the counting Bloom filter (CBF). The application of CBF enables us to structure candidate services into separate groups, resulting in an accelerated services discovery process. The contributions of this research work include a new approach to manage large number of services based on quantified service features, and a storage architecture design to support service discovery. Experimental results strongly support these claims.

1. Introduction

The SOA-based design enables service vendors to produce flexible business applications by reusing and aggregating existing service assets in an on-demand manner [14]. Guided by the SOA design philosophy, a service provider packages existing applications into services with standardized interfaces and publishes them into the service registry. The service registry maintains a record for each registered service, which contains various types of pertinent information such as the corresponding service provider and Quality of Service (QoS) attributes. Such information enables the published service to be discovered and invoked by a service composition engine. However, the proliferation of Web services and their diverse applications cause the problem of service explosion: the number of registered services keeps increasing, making management of the service registry an extremely challenging task. This issue has not been fully explored in the current service registry design [14].

In our previous work [13], we characterized the services through quantifiable features, and classified services into separate clusters based on well-established pattern recognition algorithms. This *services clustering* leads to a hierarchical services management scheme to support the large-scale SOA applications. In this paper, we continue our work by applying feature-based service characterization to service discovery and service registry design. Based on feature-based service characterization, we encode both the registered services and incoming service requests as bit strings, and thus provide a new approach to solve the service discovery problem using string matching. Furthermore, we propose a service-storage architecture based on efficient yet widely-applicable data structures that support the storage and fast lookup of string data types. During the structural design of the storage architecture, we keep track of the relationships between services and their respective providers, and take into account the influences of both the functional and non-functional requirements in the services discovery process. All of these design considerations are reflected in the service discovery protocol based on the proposed storage architecture.

The remainder of the paper is structured as follows: In section 2 we briefly introduce basic concepts of Web services in relation to business processes and associated requirements analysis in order to provide the necessary background of our research. Specifically, in section 2.2, we review the previous work on quantifying services as the foundation of the current work. In section 3, we introduce a services-storage architecture and describe its design in detail. Finally, in section 4, we delineate our research contributions, and conclude the paper with discussions and future work.

2. Preliminaries

In the SOA-based service systems, the registry services can be classified into different service groups with each group sharing the same functionality. From the perspective of delivered functionality, each service group is identified as a service task, denoted as ST_i. For clarification purposes, we use $\mathcal{S}_i = \{AS_1^i, \ldots, AS_j^i, \ldots, AS_N^i\}$ to denote the service group corresponding to ST_i, where AS_j^i represents the j-th service being able to finish the service task ST_i, and N is the size of $\mathcal{S}_i$. The scenario of $N > 1$ indicates that the service task ST_i is able to be realized by multiple candidate services, which can differ in their respective QoS attributes and other service features.

2.1. Requirements Analysis of Service Discovery

SOA design principles enable the creation of a platform independent composable service infrastructure that facilitates collaboration across different business domains. This collaboration is realized through service composition, which aggregates a variety of services into a business process to meet the customer's requirement [15]. Figure 1 illustrates a business process consisting of eight service tasks. The arcs in the graph represent the business logic embedded in this business process.

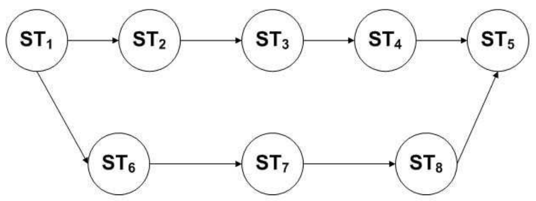

Figure 1. A Business Process Example

From the perspective of requirements analysis in the SOA design practice, service discovery is separated into two phases. In the first phase, a collection of service tasks is determined to fulfill the functional requirements of the requested business process based on the function-class decomposition [4]. As shown in Figure 1, $\{ST_1, ST_2, \ldots, ST_8\}$ is the set of selected service tasks. In the second phase, the SOA solution designers decide which service is going to be invoked for each service task selected in the first phase. The selection criterion in the second phase is to meet the non-functional requirements, which are framed as either global or local. The localized non-functional requirements focus on a single service task, whereas the global non-functional requirements relate to the performance metrics at the business process level, such as the end-to-end response time [12]. A variety of approaches have been proposed to decompose the global performance metrics into local performance thresholds. Within the realm of this research, we assume that all of the listed non-functional requirements have been either specified as or decomposed into the local ones [6]. Accordingly, after determining the collection of service tasks to constitute the requested business process, the service discovery is to iterate over the corresponding service groups and search a service within each group in order to match the associated localized non-functional requirements.

2.2. Quantifying Services With Features

Each service is characterized by a set of features with a wide range of attributes. In practice, typical service features include *Services Accessibility, Services Cost, Services Reliability and Services Response Time* [13]. The services belonging to the same service group share the same feature set used for characterization purposes. Each selected feature is associated with a numerical scale with which we can quantify a particular service. In the following example, we quantify the response time (RT) in the scale of "1-5". A service with smaller response time is assigned a higher score. This quantification procedure differs in accordance with the studied feature. In [13], we discussed a similar procedure for quantifying services reliability. The numerically scored service features can be represented as a bit string based on binary notation scheme. For instance, the l-th feature of AS_j^i, i.e., $bs_l^{i,j}$, stands for the service response time and gets a numerical score of 5 in the above quantification procedure, which is hence encoded as a binary bit string of 0101. Hereafter, we use $bs_l^{i,j}$ to denote the bit string corresponding to the l-th feature for characterizing service AS_j^i. The identity labels for AS_j^i is moved to the superscript of $bs_l^{i,j}$ for identification purpose. By considering all of the related features, the service is also represented as a bit string, the congregation of a collection of bit strings, each of which corresponds to a single feature.

Table 1. Quantification Procedure of RT

Quantitative range of RT	RT score
$RT \le 10s$	5
$10s < RT \le 20s$	4
$20s < RT \le 30s$	3
$30s < RT \le 40s$	2
$RT > 40s$	1

$$AS_j^i \leftrightarrow bs_1^{i,j} \ldots bs_l^{i,j} \ldots bs_n^{i,j} \qquad (1)$$

In Eq.1, the whole string $bs_1^{i,j} \ldots bs_l^{i,j} \ldots bs_n^{i,j}$ is called a *feature configuration*, which encodes every related feature for service AS_j^i. This is also the information published to the service registry and is available to the service discovery process.

This feature-based bit string representation scheme works not only for the registry services, but also for the service requests. The customers' various requirements are represented as the targeting score for the corresponding features used to characterize services. The service discovery process is hereby transformed into a classical string matching problem, which has been studied extensively [5]. From this point of view, the feature-based service representation scheme provides a new approach to solve the service discovery problem based on well-established algorithmic solutions. A string matching-based service discovery solution requires comparison of every candidate service against the request. Accordingly, the total computational cost depends on both the number of comparison operations and the performance of each string matching operation. The algorithm design for the string matching problem is beyond the research scope of this paper. Our research focus is the storage architecture design for the service registry that can reduce the number of comparison operations.

3. A Scalable Storage Architecture to Support Service Storage and Service Discovery

With respect to a given service infrastructure, the number of service tasks in terms of offered functionalities are limited; however,the number of services in terms of their feature configurations can become much bigger. This is a reasonable hypothesis because a service provider is able to provide multiple services with different feature configurations. For instance, a server can serve requests with different response time values. Consider a service task characterized by 6 features, and each feature is scored on a scale of "1-5". Potentially, there will be 5^6 different feature configurations for this single service task. The large number of various services resulting from the combinatorial feature configurations are published to the service registry as candidates for the service discovery. Formally, we use $\mathcal{T} = \{ST_1, \ldots, ST_m\}$ to denote the set of service tasks under a service infrastructure. According to the discussion at the beginning of section 2, every service task ST_i corresponds to a service group, denoted as $\mathcal{S}_i$. The union of $\mathcal{S}_i$, i.e., $\bigcup_{i=1}^{i=m} \mathcal{S}_i$, is denoted as $\mathcal{G}$.

From the perspective of storage architecture design, $\mathcal{T}$ and $\mathcal{G}$ are stored separately, and the storage requirement for $\mathcal{T}$ is less demanding than the one for $\mathcal{G}$, due to the fact that a single service task can have multiple candidate services. The mapping relationship between $ST_i \in \mathcal{T}$ and $S_i \subseteq \mathcal{G}$ suggests that the former can be used to index the latter. In the storage architecture design, keeping track of the mapping relationship between service task and the corresponding service group is important to support the service discovery process, which is separated into two sequential steps as discussed in section 2.1.

In this section, we construct a hybrid storage architecture (HSA) that is able to store a large amount of services while supporting an efficient service discovery process. HSA, illustrated in Figure 2, is composed of an service task array used to store the set $\mathcal{T}$, and a collection of storage units used to store the set $\mathcal{G}$. Due to the limited number of service tasks, the service task array is based on the normal data structure, such as array. The i-th entry of the service task array stores the identity of a service task, e.g., ST_i, and maintains a pointer to the storage unit i, which stores the service group S_i. Due to the large size of S_i, the focus of this section centers around the design of these storage units. In our research, we propose to use counting Bloom filter, a probabilistic data structure, to build the storage unit for every service group while supporting the service discovery process.

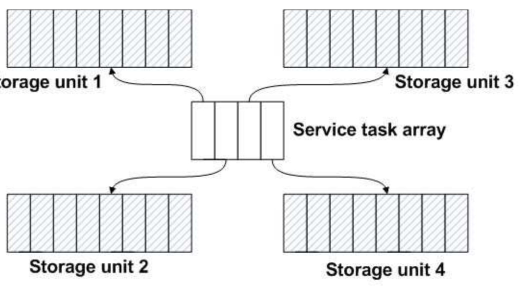

Figure 2. Hybrid Storage Architecture for Service Registry

3.1. Standard Bloom Filters and Counting Bloom Filters

A standard Bloom filter (BF) is a hashing-based data structure representing a set of elements [1]. Compared to the hash table, BF can reduce the space requirement further and allows the use of simpler hash functions, which saves computational cost for lookup-intensive operations [2][7]. A BF is composed of a bit array denoted as B and g independent hash functions $\mathcal{H} = \{h_1(\ ), \ldots, h_g(\ )\}$. The bit array B is of length q and is initialized as 0 for all of the entries. Each hash function maps an element in $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ to an array entry. Hereafter, we use $BF^{\mathcal{A}}$ to denote the BF for set $\mathcal{A}$, and use $B^{\mathcal{A}}$ to denote the related bit array. Algorithm 1 lists the procedures of constructing a Bloom filter representing a given set, and Figure 3 shows the constructed $BF^{\mathcal{A}}$ for set $\mathcal{A}$.

Algorithm 1 Constructing Bloom filter for set $\mathcal{A}(\|\mathcal{A}\| = n)$

```
1:  for $i = 0$ to $q$ do
2:      $B^{\mathcal{A}}[i] = 0$;
3:  end for
4:  for $i = 0$ to $n$ do
5:      for $j = 0$ to $g$ do
6:          $p = h_j(a_i)$;
7:          if $B^{\mathcal{A}}[p] = 0$ then
8:              $B^{\mathcal{A}}[p] = 1$;
9:          end if
10:     end for
11: end for
```

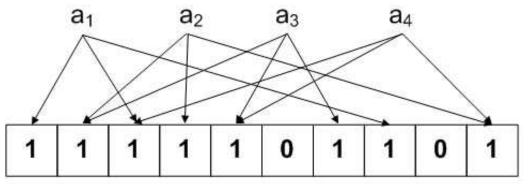

Figure 3. A Bloom Filter Example

BF has two structural properties resulting from its construction scheme.

Property 1: If $a \in \mathcal{A}$, then the construction process should mark every hashed position in the bit array with 1. Accordingly, for a given query a, if any of the hashed positions in $B^{\mathcal{A}}$ for a is found to be 0, then $a \notin \mathcal{A}$.

Property 2: With respect to $BF^{\mathcal{A}}$ representing set $\mathcal{A}$ and a query $a \notin \mathcal{A}$, it is possible to find that every hashed position of a having been marked with 1 in $B^{\mathcal{A}}$.

Given an element a and a set $\mathcal{A}$, determining whether $a \in \mathcal{A}$ is called *set membership evaluation*, which is used in a lot of applications. A trivial approach for the set membership evaluation is to compare each element belonging to $\mathcal{A}$ with a until we find a match, which involves a computational complexity of $\mathcal{O}(\|\mathcal{A}\|)$, where $\|\mathcal{A}\|$ represents the size of set $\mathcal{A}$. Instead of comparing each element of $\mathcal{A}$ with a, the BF-based set membership evaluation computes every hashed position of a and checks whether it has been marked with 1. In accordance with **Property 1**, an all negative answer can be used to remove the possibility of $a \in \mathcal{A}$ with certainty. On the other hand, implied by **Property 2**, an all positive answer cannot ensure that $a \in \mathcal{A}$ due to the possibilities of overlaps in the hashed positions among different elements belonging to $\mathcal{A}$. This issue is known as *false positive* yielded by BF-based set membership evaluation [2]. The false positive rate will decrease exponentially when g increases, which makes the influences of false positive negligible in real applications [2].

The overlaps in the hashed positions among different elements make it unable to support BF-based deleting operation on a set. For instance, deleting a_i from $\mathcal{A}$ requires the BF to reset every hashed position $B^{\mathcal{A}}[h_l(a_i)] : \forall l = 1, \ldots, g$ to zero. If $h_l(a_j) = 1$ for $j \neq i$, then the deleting operation of a_i will affect the associated hashed position of another element a_j, which results in an incorrect BF representation for a_j, whose g hashed positions should always be marked "1". To address this issue, counting Bloom filter (CBF), an extension to the BF, is developed by setting the entry attribute as a counter value rather than a bit as in BF [8]. During the construction of a CBF, every hashed position for an element is increased by "1" rather than marked with "1". A counterpart of Figure 3 is shown in Figure 4, which illustrates the CBF representation for set $\mathcal{A}$. The value of each array entry equals the number of input arcs, i.e., the number of elements hashed into this entry. Besides the capability of supporting the deleting operation, the introduction of counter values also enables the CBF to keep track of the number of elements having been hashed into a given position. This property will be used in the latter for the CBF-based storage unit design. The set membership evaluation based on a CBF is similar to the BF: if any of the g hashed position of query a is found to be 0 in $B^{\mathcal{A}}$, then $a \notin \mathcal{A}$.

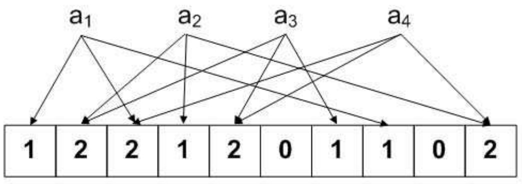

Figure 4. A Counting Bloom Filter Example

3.2. Storage Unit Design Based on Counting Bloom Filter

In this section, we will discuss the storage unit design for the service groups. Each unit stores a service group for a given service task, and consists of two parts: CBF-based counter value array and a collection of linked lists. Each linked list is indexed by an array entry of CBF. Compared to the normal CBF, each entry not only stores the counter value, but also contains a pointer to the associated linked list. This storage design is based on the concept proposed in [11], which is called *pruned fast Hash table* in the original paper. The major structural property of this design is the placement of element. With respect to an element and the set of corresponding hashed positions, the element will be stored in the linked list whose length is the smallest according to the counter values of these hashed positions. In Figure 5, we use the organization of the storage unit for service group $\mathcal{S}_1$ as an illustrative example.

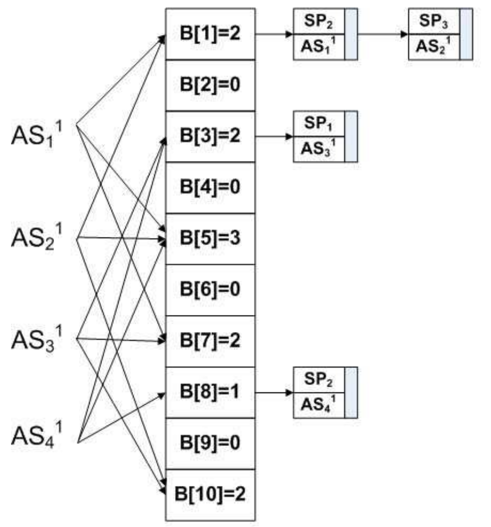

Figure 5. CBF-based Storage Unit for Service Group $\mathcal{S}_1$

$\mathcal{S}_1$ has four different services, which are denoted as AS_1^1, AS_2^1, AS_3^1 and AS_4^1 respectively. AS_2^1 is provided by SP_3, AS_3^1 is provided by SP_1, and SP_2 is able to deliver two different services AS_1^1 and AS_4^1. Here, SP_i stands for the identity of a service provider. In order to fulfill the functionality of service discovery, the HSA needs to keep track of the chaining relationships connecting service task, service, and service provider. This motivates our extension of the classical CBF configuration. As shown in Figure 5, each node in the attached list stores the bit string representing a service and its associated service provider. There are three hashed positions for AS_3^1, which are $3, 7$ and 10. The associated counter values are $B[3] = 2, B[7] = 3$ and $B[10] = 2$ respectively. The smallest counter value among the hashed positions of AS_3^1 is $B[3]$, and thus the element is placed in the linked list pointed by $B[3]$. Note that the counter value stored in an array entry does not necessarily reflect the true length of its attached linked list. For instance, $B[5] = 3$, but its attached linked list is empty. Algorithm 2 illustrates the operation of inserting a service into the storage unit for its corresponding service group.

In Algorithm 2, i is the identity of the targeting storage unit for the service group $\mathcal{S}_i$ corresponding to ST_i. Due to variances among different service groups, the corresponding CBFs are heterogeneous in their respective parameter settings. Thus, we use g^i to represent the number of hash functions used by BF^i and h_y^i to represent the y-th hash function. The operation of line 7 is to locate the linked list with the smallest counter value among g^i hashed positions for service AS_j^i. In line 8, $list_m$ represents the linked list pointed by the m-th array entry, and $SP(AS_j^i)$ stands

Algorithm 2 Insert service and index it with the corresponding service task

```
1: Input: $ST_i$, $AS_j^i$
2: Initialize $p$ as a zero array of size $g^i$;
3: for $y = 1$ to $g^i$ do
4:     $p[y] = h_y^i(AS_j^i)$;
5:     $B^i[p[y]] = B^i[p[y]] + 1$;
6: end for
7: $m = \mathbf{argmin}(B^i[p[y]]), \forall y = 1, \ldots, g^i$;
8: Attach a node consisting of $AS_j^i$ and $SP(AS_j^i)$ to $list_m$
   of $BF^i$;
```

for the service provider for AS_j^i, which also needs to be stored. Algorithm 3 lists the HSA-based service discovery protocol, which is divided into two phases: determining the storage unit for the service group that can fulfill the functional requirements of a service request, and discover the service that can fulfill the non-functional requirements of a service request from the storage unit obtained in the first phase. This two-phase protocol matches the service discovery process discussed at the end of section 2.1.

In Algorithm 3, RS_t denotes the requested service task. RS_f denotes the bit string representing the requested service, which stands for the customer's requirement on the service. The operations listed between line 2 and line 6 belong to the first phase of HSA-based service discovery protocol. As we discussed above, the i-th entry of the service task array stores the identity of a service task. The operation of **FindIndex** iterates over the service task array and returns the task identity that matches the requested service task. A returned *null* value of c indicates that there does not have a service group being able to meet the functional requirements of request. The array of p stores the hashed positions for the request. In accordance with the membership evaluation criterion of CBF, we check whether the request RS_f belongs to the service group based on the smallest counter values in array p, which is the operation listed at line 11. A positive answer indicates that it is possible to have a candidate service matching the request. This operation is a fast lookup without 100% guarantee due to the false positive of CBF. A positive feedback leads to the searching operation by traversing the associated linked list. Note that the smallest element of p not only works for evaluating the element membership but also helps selecting the linked list to be searched. **LinkedListSearch** represents a standard searching operation over a linked list. The returned node is denoted as d, and $d.sp$ stands for the service provider identity stored in d. A returned null node indicates that there is no match to the request and the original positive feedback is caused by false positive.

Algorithm 3 is composed of three major operations: (1) the operation of **FindIndex** determines the particular ser-

Algorithm 3 Service discovery based on HSA

```
1: Input: $RS_t$, $RS_f$
2: $c = \mathbf{FindIndex}(RS_t)$;
3: if $c = null$ then
4:     print "There is no match to the requested service task."
5:     return
6: else
7:     Initialize $p$ as a zero array of size $g^c$;
8:     for $i = 1$ to $g^c$ do
9:         $p[i] = h_i^c(RS_f)$;
10:    end for
11:    $y = \mathbf{argmin}(B^c[p[m]]), \forall m = 1, \ldots, g^c$;
12:    if $p[y] = 0$ then
13:        print "There is no match to the requested service."
14:        return
15:    else
16:        $d \leftarrow \mathbf{LinkedListSearch}(list_{p[y]}, RS_f)$;
17:        if $d \neq null$ then
18:            print "d.sp is the service provider."
19:            return
20:        else
21:            print "There is no match to the requested service."
22:            return
23:        end if
24:    end if
25: end if
```

vice task for fulfilling the requested functional requirement, the computational cost of which depends on the number of registered service tasks; (2) the operations at lines $8-14$ perform the membership evaluation and chooses the linked list to be searched, the computational cost of which depends on the number of hash functions, i.e., g^c; (3) the computational cost of searching operation of **LinkedListSearch** depends on the length of the linked list bounded by $p[y]$. Among these three operations, **LinkedListSearch** contributes the most to the total computational cost because neither the number of registered service tasks nor the number of hash functions will be a big value.

3.3. Performance Analysis of the Service Discovery Process Based on HSA

As we discussed at the end of Section 2, the bit string-based service discovery process requires the comparison operations conducted between the service request and every candidate service. Our design motivation is hereby to reduce the number of these kinds of comparison operations. We use the expected number of comparison operations as the evaluation metric. The benchmark service discovery process is performed on a normal storage unit. Without any assisted information, the position of a candidate service on the linked list is random and assumed to follow the uniform distribution. Suppose there are n candidate services, then the expected number of comparison operations is $\frac{n}{2}$. On the other hand, the proposed storage unit design is a combination of a linked list with the CBF-based counter value array, which decides which linked list to be placed for a given service based on the hashed positions. As shown in Algorithm 3, if every hashed position is discovered to be non-empty, then we will search the linked list pointed by the entry whose counter value is the smallest among all of the hashed positions. Accordingly, the number of comparison operations for our approach is bounded by the counter values. In the following, we list the derivation procedure for the expected counter value based on probabilistic analysis.

$$\mathbf{E}[V] = \mathbf{E}[\mathbf{E}[V|\tilde{g} = i]] \tag{2}$$

Eq.2 computes $\mathbf{E}[V]$ using iterated expectation [3]. $\mathbf{E}[V|\tilde{g} = i]$ represents the conditional expectation of $\mathbf{E}[V]$ when there have i different hashed positions, and is estimated as $\frac{ng}{i}$ since the hashed positions are assumed to follow uniform distribution. We use $\tilde{g}$ to denote the number of different hashed positions in order to differentiate it with g, i.e., the number of hash functions to be used. $\mathbf{P}(\tilde{g} = i)$ represents the probability that g hashed functions produce i different positions, which is computed in Eq.3.

$$\mathbf{P}(\tilde{g} = i) = \binom{q}{i} \frac{\sum_{a=0}^{i} (-1)^a \binom{i}{a} (i-a)^g}{q^g} \tag{3}$$

In Eq.3, the item of $\binom{q}{i}$ represents the number of combinations of choosing i positions from q available entries of the counter value array; the item of $\sum_{a=0}^{i} (-1)^a \binom{i}{a} (i-a)^g$ represents the number of permutations of putting g hashed results into i positions and none of these i positions is empty, which is derived based on the *generating function for permutation* [10]; the item of q^g represents the number of combinations of g hashed positions. Submitting Eq.3 into Eq.2, we have

$$\begin{aligned}
\mathbf{E}[V] &= \mathbf{E}[\mathbf{E}[V|\tilde{g} = i]] \\
&= \sum_{i=1}^{g} \mathbf{E}[V|\tilde{g} = i] \mathbf{P}(\tilde{g} = i) \\
&= \sum_{i=1}^{g} \frac{ng}{i} \binom{q}{i} \frac{\sum_{a=0}^{i} (-1)^a \binom{i}{a} (i-a)^g}{q^g} \qquad (4)
\end{aligned}$$

Eq.4 shows that $\mathbf{E}[V]$ is a function of n and g, which are the parameters of the CBF. Due to the complexities of Eq.4, it is quite difficult to get the analytical solution of $\mathbf{E}[V]$ and compare it with $n/2$ directly when n is large. Therefore, we conduct a simulation- based experimental study to compare the expected counter value with the benchmark value of $n/2$. For illustration purposes, we report both the expected counter value and the expected length of the linked list to be searched.

Table 2. Expected Counter Value

20000 services (n=20000)					
	q = 90	q = 95	q = 100	q=105	q=110
g = 4	848	794	761	729	695
g = 6	1269	1211	1120	1099	1036
g = 8	1688	1603	1515	1427	1367
40000 services (n=40000)					
	q = 90	q = 95	q = 100	q=105	q=110
g = 4	1725	1598	1540	1471	1407
g = 6	2562	2449	2303	2204	2109
g = 8	3352	3229	3092	2867	2776

Table 3. Expected Length of the Linked List to be Searched

20000 services (n=20000)					
	q = 90	q = 95	q = 100	q=105	q=110
g = 4	493	459	442	438	431
g = 6	672	673	583	563	558
g = 8	887	851	804	806	740
40000 services (n=40000)					
	q = 90	q = 95	q = 100	q=105	q=110
g = 4	997	968	842	873	825
g = 6	1343	1323	1285	1105	1064
g = 8	1707	1704	1653	1587	1437

In the experiment, we investigate a variety of parameter settings for the CBF along with two different numbers of services. Table 2 reports the expected counter value, whereas Table 3 reports the expected length of the linked list to be searched. The experimental results show that each expected length of the linked list to be searched is smaller than the corresponding expected counter value, and in some cases the differences are large. Therefore, it is a conservative yet applicable approach to use the expected counter value as the estimation of the expected length of the linked list to be searched, which directly controls the number of comparison operations.

Based on Table 2 and Table 3, a larger size of counter value array for the CBF, i.e., q, reduces both the expected counter value and the expected length of the linked list to be searched. This is due to the fact that a larger counter value array allows more linked lists to be built, which leads to a more distributed storage configuration. Although the total storage requirement is still the same, the expected number of comparison operations can be reduced accordingly. In practice, the counter value array of CBF can be maintained in a high-bandwidth and small on-chip memory [11]. This makes the computation of hash functions very fast, which is critical to a lot of hashing-based applications. For instance, we can block the request very quickly if we get a negative answer from the CBF-based set membership evaluation. This feature enables the customer to decide whether to modify the request or just quit the market in a very short time period, which is a desirable characteristic in a competitive business environment. On the other hand, the size of on-chip memory limits the size of counter value array that we can use.

The construction scheme of CBF results in an increase in the counter values when we use more hash functions. This fact is reflected in the results shown in both tables. Nevertheless, Bloom filter theory suggests that the false positive rate decreases exponentially when we increase the number of hash functions. According to Algorithm 3, the HSA-based service discovery protocol decides whether to block the request based on the CBF-based set membership evaluation. A small false positive rate is always desirable for this kind of functionality. Therefore, choosing the number of hash functions should be treated as a design trade-off that balances different factors.

Overall, as shown in Table 2 and Table 3, both the expected counter value and the expected length of the linked list to be searched are much smaller than the number of comparison operations of the benchmark case, which is $n/2$. This shows that a carefully designed service storage architecture can provide a better lookup structure than a trivial one.

4. Conclusions and Discussions

From the perspective of service science and engineering, our contributions in this paper are threefold. First, after discussing our previous work on feature-based service quantification, we distinguish between the concepts of service tasks and services in terms of functional requirements and non-functional requirements. The quantification result is used to represent services as bit strings that can be used to reframe the service discovery process as a string matching problem, which provides a new approach to support the service discovery. Further, we propose a hybrid storage architecture and detail its design based on CBF. Rather than focusing on the storage solution only, we also formalize the service discovery protocol based on the proposed architecture. Overall, our work centers on two essential functionalities that have to be considered in a systematic service registry design: the structure of service storage and the related service discovery protocol.

View publication stats

The proposed storage architecture still has several limitations that need to be lifted in the future research. Regarding the HSA, the counter values and the associated linked lists for each storage unit are configured when building CBF based on the existing service group. If a service is newly created and is required to be added into the related storage unit, then the CBF and its related linked lists have to be rebuilt in order to allow the service discovery protocol perform correctly. This rebuilding operation will cost a lot of computational resources if there are frequent service insertions. An incremental insertion algorithm, proposed by Song et.al [11], can be applied to support dynamic element insertion. Another approach is to design storage unit based on dynamic Bloom filter, which supports the dynamic set management and aims to control the false positive rate even the set size increases [9].

In [13], we developed a pattern recognition-based service clustering scheme to organize the services in an hierarchical structure to accelerate the service discovery process. Service clustering enables us to classify a group of services into different clusters based on a collection of features for characterizing the services. This technique can be used to refine the HSA design. Instead of using a single CBF-based storage unit to store the whole service group corresponding to a given service task, we can build a set of storage units with each of which storing a service cluster rather than a larger service group. Hence, the service discovery process will be performed on a smaller sized storage unit. This increased granularity can improve the performance of service discovery by reducing the number of comparison operations further.

In the future research, we will test these designs on an experimental Web services management platform and evaluate the performance using real-world examples, which can offer a variety of challenges. These challenges can lead us to develop a more scalable and robust storage architecture for the service registry.

References

[1] B.Bloom. Space/time tradeoffs in hash coding with allowable errrors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2002.

[3] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury Press, 2001.

[4] C. K. Chang, J. Cleland-Haung, S. Hua, and A. Kuntzmann-Combelles. Function-Class Decomposition: A Hybrid Software Engineering Method. *IEEE Computer*, 34(12):87–93, 2001.

[5] C. Charras and T. Lecroq. *Handbook of Exact String Matching Algorithms*. College Publications, 2004.

[6] Y. Chen, S. Lyer, X. Liu, D. Milojicic, and A. Sahai. Translating Service Level Objectives to Lower Level Policies for Multi-tier Services. *Cluster Computing*, 11(3):299–311, 2008.

[7] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching Using Bloom Filters. *IEEE/ACM Trans. Netw.*, 14(2):397–409, 2006.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: a Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[9] D. Guo, J. Wu, H. Chen, and X. Luo. Theory and Network Applications of Dynamic Bloom Filters. In *INFOCOM*, 2006.

[10] P. A. MacMahon. *Combinatory Analysis*. Dover Publications, 2004.

[11] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *SIGCOMM*, pages 181–192, 2005.

[12] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-Aware Middleware for Web Services Composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327, 2004.

[13] L. J. Zhang, S. Cheng, Y.-M. Chee, A. Allam, and Q. Zhou. Pattern Recognition Based Adaptive Categorization Technique and Solution for Services Selection. In *Proceedings of the 2nd IEEE Asia-Pacific Service Computing Conference*, pages 535–543, 2007.

[14] L. J. Zhang, J.Zhang, and H.Cai. *Services Computing*. Springer and Tsinghua University Press, 2007.

[15] L. J. Zhang and B. Li. Requirements Driven Dynamic Services Composition for Web Services and Grid Solutions. *Journal of Grid Computing*, 2(2):121–140, 2004.