SRMC: A Model for Web Services Registry with Multilevel Caches

Hao-peng Chen[1,2], Shu-jian Wang[1], Shao-chong Li[1]

[1] School of Software
Shanghai Jiao Tong University
Shanghai 200240, P.R. China

[2] School of Computer Science
Georgia Institute of Technology
Atlanta 30332, USA

E-mail: {chen-hp, ggoodd, lee.shaochong}@sjtu.edu.cn

***Abstract*—In order to improve the efficiency of service discovery and release the load of service registry, this paper proposes a service registry model named as SRMC (Service Registry with Multilevel Caches) which clusters the service consumers into groups according to their searching similarity and sets up a multilevel cache for all groups to improve the performance of service discovery. The multilevel caches of SRMC are refreshed by a hybrid mechanism which includes event-based refreshing and periodical refreshing. The basis of refreshing and clustering is the history records of service discovery requests issued by service consumers. The running results of an instance of SRMC deployed in an experimental environment have shown that SRMC is effective to reduce the times of accessing global storage and the amount of data searched in service discovery.**

Keywords-service registry; service discovery; multilevel cache; service consumer similarity

I. INTRODUCTION *(HEADING 1)*

Service-centric computing has been a trend for building enterprise applications in recent years. Since development of applications based on service-centric computing depends greatly on services published on web, more and more companies and organizations, even more individuals, publish their services on Internet as a kind of investment. Meanwhile, cloud computing greatly lowers the threshold for being a service provider since it provides infrastructure as a service (IaaS) and platform as a service (PaaS) for service providers [1]. Consequently, the number of web services has been increasing by exponential factor which brings a challenge for service consumers: how to discover the most suitable service cater to their requirements. As a result, the importance of service registry has been shown much clearer than several years before.

UDDI (Universal Description Discovery and Integration) is the current standard of service registry [2]. It provides a mechanism for service registering and discovering based on WSDL (Web Service Description Language) descriptors of services. WSDL is the specification for describing the features of web services, including the types element describing the kinds of messages that the service will send and receive, the interface element describing what abstract functionality the Web service provides, the binding element describing how to access the service, and the service element describing where to access the service [3]. We can find, however, WSDL does not include any semantic information about the functionality and quality of a web service, though it allows publishers to edit a description of service in natural language. The lack of semantic information about functionality and quality in WSDL makes that UDDI-complied service registries only can issue searches for services based on general keywords [2].

General keywords based searching ignores the services whose descriptions have no matched keywords but semantically equal to the functional requirements in the discovery request. Moreover, it considers that the services whose descriptions have all the keywords but don't semantically equal to the functional requirements in the discovery request are proper candidates. Consequently, peoples developed behavior-aware approaches to improve the accuracy of service discovery by adding semantic information into the descriptors of web services. As a result, Semantic Web Service (SWS) has been a research trend due to the knowledge-representation languages and ontology it brought. SWS also provides infrastructure for approaches to describe, discovery and invoke activities on the Web [4].Sheila A.McIlraith et al firstly indicated the importance and potential of bringing Semantic Web technologies to Web services in 2001 [5]. From then on, SWS emerged as a distinct research field, and a large number of initiatives began not long thereafter, including OWL-S [6], WSMO [7], SWSF [8], and WSDL-S [9]. Semantic Web services discovery(SWSD)， as defined by the Semantic Web Services Initiative Architecture(SWSA) committee, is the process of a service requestor identifies candidate services to achieve its objectives [10].

For the nonfunctional requirements, since WSDL-complied descriptor of a web service has no such information [3], people extend it to describe the static rating of QoS in order to support QoS-aware service discovery. For example, an OWL-S profile of a service has the information on its quality rating [6]. IBM's WSLA also supports describing static quality rating as assertions of a service provider to perform a service according to agreed guarantees for IT-level and business process-level service parameters [11]. However, the static rating of QoS is up to the service requester to use this information, to verify that it is indeed correct, and to decide what to do with it. Thus, the dynamic status of QoS is more important than static quality rating for service consumers to discover the desired services. To get the dynamic status of QoS, we need to monitor and measure the services at run-time. Monitoring may take different forms, depending on the QoS parameter. These include message interception, probing, value collection, and user feedback [12].No matter what form is adopted, it will generate enormous data to be kept and analyzed.

To support both behavior-aware and QoS-aware service discovery, the service registry need to store not only the descriptors the UDDI specification specified, such as WSDLs, but also the additional descriptor with either semantic descriptions or QoS descriptions, such as RDFs[13] and WSLAs. Moreover, it also needs to be able to receive and store the run-time status of QoS sent by service monitoring tools. As a result, there are massive data kept in service registry which turn service discovery into a data-intensive mission. When massive service consumers concurrently send service discovery requests to service registry, the service registry will be apt to be overloaded. If we cluster service consumers into groups according to the searching similarities among them and cache the services searched by the service consumers in same group, the most of service discovery requests would be processed by searching caches but not global storage of service registry. Consequently, the load of service registry would be reduced and the performance of service discovery would be improved greatly. This mechanism certainly would bring a flaw that some candidates would be probably ignored when the service discovery requests are processing. But for most service consumers, efficiency is the most important factor of service discovery. After all, it is much better for service consumers to get a sub-optimal service quickly than get a optimal service in an unacceptable time.

Aware of the advantages brought by caches, we propose a service registry model named as SRMC (Service Registry with Multilevel Caches) which clusters the service consumers into groups according to their searching similarity and sets up a multilevel cache for all groups to improve the performance of service discovery. The remainder of the paper is structured as follows. Section II briefly summaries the related works; Section III gives the principle of the SRMC; Section IV describes the method for service consumer clustering; Section V shows the instantiation of SRMC; and conclusion in Section VI.

II. Related Work

As the specification of service registry, UDDI has not been changed for more than five years, because as a keywords-based service registry specification, UDDI is rather complete. Unlike the original imagination of members of UDDI alliance, the global service registry providers are dead or dying. Meanwhile, some open source and free service registry implementations keep evolving now. For example, jUDDI is an open source Java implementation of the UDDI v3 specification for Web Services. It is used in many study and research projects due to its simplicity and efficiency [14]. Seekda [15] is a free web services search engine for Web API and their providers. It helps users to find web services based on a catalogue of more than 28,000 service descriptions. From the technical point of view, either jUDDI or Seekda is keywords-based service registry.

Aware of the conspicuous deficiency of key-words based search, researchers are attempting to design semantics matching search by add semantic description into the service descriptors. RDF is a language for representing information about resources in the World Wide Web [13] which facilitates the representation of semantics in service descriptor. The semantics in RDF could be represented by OWL-S, a language based on Semantic Web ontology language OWL [6].

The most influencing semantic matchmaking we are aware of is the Paolucci et al. algorithm [16], which has been cited extensively in subsequent proposals. In [16], Paolucci et al. proposed an ontology-based solution, which matching Inputs/Outputs of Services by evaluating their semantic similarity between them according to the hierarchical concept relationships defined in an ontology tree. In another frequently cited paper [17], Matthias Klusch et al. extended Paolucci's algorithm by adding new semantic similarity grades. Matthias also designed a matchmaker, called OWL-MX, to evaluate the semantic similarity grade between two services. There are some other algorithms derived from these two algorithms. In general, the maturity of ontology-based semantic matchmaking has become acceptable now. The service registries can adopt one of them to support behavior-aware service discovery.

Apart from no support for semantic matchmaking, existing UDDI-complied service registries, such as jUDDI and Seekda, has no support for QoS matchmaking. Researchers adopt two ways to support QoS-aware service discovery: to add static QoS rating and to obtain run-time status of QoS.

Kyriakos Kritikos et al. developed an OWL-S based (syntactical separation) solution, called OWL-Q, to describe static QoS rating [18]. However, no matter which way to be chosen among WSLA, OWL-S, and OWL-Q, it is up to the service requester to use this information, to verify that it is indeed correct, and to decide what to do with it. The dynamic status of QoS can be obtained by monitoring and measuring the services at run-time. MDS (Monitoring and Discovery System) of GT 4 aims to monitor and discover resource in a grid environment [19]. IBM's Tivoli software suite also provides the capability to monitor the status of web service hosted in Websphere [20]. Actually, service registries can gather both of the two kinds of information and provide the QoS-aware service discovery by analyzing the stored information. We has designed such a service registry, named as QMC [21].

To improve the performance of service discovery, UDDI specification v3.0.2 includes replication APIs to support distributed multi-node architecture of service registry. P2P is also an architecture adopted by many service registries. For example, in a frequently cited paper [22], K. Verma et al. use a P2P approach to organize registries into domains, enabling domain based classification of all web services. For the load of service registry and the amount of data to be searched when processing service discovery request, replication architecture can reduce the former but not latter. Meanwhile, P2P can effectively reduce the latter but not former. To the best of our knowledge, there is no existing service registry adopting multilevel cache architecture to reduce both of them. In [23], we proposed multilevel cache architecture to achieve this aim. But it supports only QoS-aware service discovery and but not behavior-aware service discovery. This paper proposes a new multilevel cache architecture to add the support for behavior-aware service discovery.

III. Principle of SRMC

A. Architecture of SRMC

As shown in Figure 1, SRMC is a cloud application.

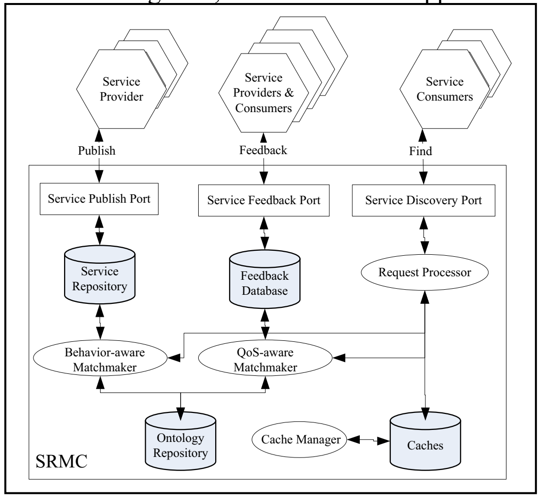

Figure 1. Architecture of the SRMC

SRMC has three ports, namely Service Publish Port, Service Discovery Port, and Service Feedback Port. The Service Publish Port enables service providers register their services into the Registry. It is an extended UDDI-complied port, which means the service providers can not only register WSDL files but also semantic description, such as RDF files, into the service registry. The Service Feedback Port is designed for collecting feedbacks from Service Customers and Service Providers. With the Service Discovery Port, users can find and locate services they are interested in. Service consumers can discover services not only by keyword-based constraints but also by semantic-based constraints and QoS-based constraints.

The Service Repository holds the static description of services, such as WSDLs and RDFs, and the Feedback Database stores the dynamic feedbacks of QoS sent by Service Consumers and Providers. The Ontology Repository stores all the ontologies, including Domain Ontology and QoS Ontology. Both of the Behavior-aware Matchmaker and QoS-aware Matchmaker will access the Ontology Repository to accomplish their tasks. The Caches are multilevel caches which are stored in a database. We will give the details of Caches in Section III.B.

When a Service Consumer proposes a service discovery request, the request is dispatched to the Request Processor. It collaborates with Caches, Behavior-aware Matchmaker, and QoS-aware Matchmaker to accomplish the mission of service discovery. The details of the process of service discovery will be described in Section III.C.

The Cache Manager maintains Caches by executing cache refreshing. The details will be described in Section III.D.

B. The structure of caches

The structure of caches is a tree shown in Figure 2.

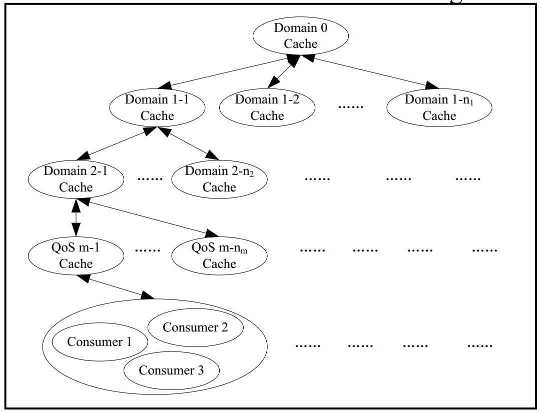

Figure 2. Structure of the Caches

All the service consumers are clustered in groups according to their similarity. The similarity is calculated by the history of service discovery requests of service consumers. The algorithm for calculating similarity will be described in Section VI. Since functional requirements are prior to nonfunctional requirements, the high level of cache tree is organized by application domains. The height of cache tree depends on the subdivision level of application domains which is specified by provider of service registry. The leaves of cache tree each of which relates to a group of service consumers are clustered by the QoS preference of service consumers. That is, for the service consumers who focus on a same domain, they will be clustered by their QoS preference. A service consumer is assigned to at least one group which is related to a leaf node of cache tree. If a service consumer focuses on several domains, it can be assigned to multiple groups.

There are two global tables keep the data of structure of caches and the relationship between service consumers and cache nodes:

- *NodesofCaches*: It holds the data of tree structure of caches. Each record has the structure as <id, URL, father> in which *id* field is the unique identity of node, *URL* field is a link to the physical storage location of the cache, and *parent* field refers to its father node. The value of *parent* field of root node is *null*.
- *ServiceConsumers*: It holds all the necessary information of service consumers which at least includes id and password. The fields of this table can be extended by provider of service registry.
- *RelationC2C*: It holds the relationship between service consumers and caches. Since a service consumer can be assigned to multiple groups, the relationship is a many-to-many one which needs a association table to hold it.

All the service consumers in same group share a unique cache to record their requests and received responds. All the data of a cache are stored in a physical location which is referred by *NodesofCaches* table. There are four local tables kept in each leaf node of cache:

- *FeaturesList*: It holds features of the cache, including a *domain* field and a *QoS* field. The *domain* field is one of the finest-grained domains which relate to the nodes of second lowest level of cache tree. The *QoS* field is a vector each component of which represents a range of a quality attribute in the form of <*attribute, minValue, maxValue*>. The statuses of QoS of all the services cached in this cache node are in the ranges specified by *QoS* vector.
- *RequestsList:* It holds different service discovery requests sent by the service consumers in this group. Each request has two parts: functional requirements and nonfunctional requirements. Since we use RDF to describe functional requirements, the functional requirements are represented by a vector, named as *func*, each component of which is in form of <*subject, {predicator}, object*>. The nonfunctional requirements are also represented by a vector, named as *nonfunc*, which is in the same form of *QoS* field of *FeaturesList* table.
- *RespondsList:* It holds the discovery results relate to requests in *RequestsList* table. It has at least three fields. The first one is *id*. The second one is a foreign id refers to a request in *RequestsList*. The third one is a vector to hold the result set, named as *result*, each component of which is a candidate service meets all the requirements in the request.
- *HistoryRequestsList:* It holds all history records of service discovery requests sent by the service consumers in this group. Each record has two foreign keys: the first one refers to a service consumer record in *ServiceConsumers* to indicate the sender of the request; the second one refers to a service discovery request in *RequestsList*. Each record also has a timestamp which is used in a fading memory method to calculate the similarity among service consumers.

The above structure of Caches is the basis for process service discovery requests.

C. *The process of service discovery*

When a service consumer sends a Service discovery request via Service Discovery Interface, it is processed by Request Processor as following steps:

1. Request Processor gets *id* of the *ServiceConsumers* table, and uses it to locate the caches related to the groups of the Service Consumer by accessing *NodesofCaches* and *RelationC2C* tables. For a new service consumer who has no history records in SRMC, it will be assigned to a temporary group, named as *freshmen*. Moreover, Request Processor respectively inserts a new record into *ServiceConsumers* and *RelationC2C* tables, and jump to step 4. For a returned service consumer, since it can be assigned into multiple groups, Request Processor may find multiple caches.
2. For each located cache, Request Processor matches the request against all the requests in *RequestsList* table. At first, the functional requirements will be matched against the *func* vector of each request by Paolucci algorithm [16]. The provider of service registry can specify an acceptable minimum degree of match, such as *Plug-in*. If the degree of match is not lower than the minimum degree, we say they match on functionality. If there is no request matches the functional requirements, jump to step 4. Otherwise, Request Processor will get a set of matching requests. Next, the nonfunctional requirements will be matched against *nonfunc* vector of each matching request. For each request, if the ranges of all quality attributes in *nonfunc* vector meet the nonfunctional requirements, we say they match on QoS. If there is no request matches the nonfunctional requirements, jump to step 4. Otherwise, the matching requests are candidate requests.
3. Request Processor iterates the set of candidate requests and gets the corresponding responds from *RespondsList* table. The joined set of all responds will be sent back to service consumer as the final discovery result. Jump to step 6.
4. Request Processor creates instances of Behavior-aware and QoS-aware matchmakers, divides the request into functional requirements and QoS requirements, and respectively passes them to the instances of Behavior-aware and QoS-aware matchmakers. The Behavior-aware and QoS-aware matchmakers respectively find a set of function-matched services and a set of QoS-matched services. Request Processor calculates the intersection of the two result sets returned by Behavior-aware and QoS-aware matchmakers. The resulting intersection is the final result which is sent back to the Service Consumer.
5. Request Processor inserts a new record into *RequestsList* table to record a new request which is different with all records in *RequestsList* table.
6. Request Processor inserts a new record into *HistoryRequestsList* table to record the action of service discovery of the service consumer.

All the service consumers in group *freshmen*, will be reassigned into other group(s) when the Cache is refreshed. The mechanism of Cache refreshing will be described in next section.

In such a process, Request Processor searches local caches at first. Only when it receives a new request, it will search the global storage of service registry. As a result, the mean amount of data to be searched when processing service discovery requests is reduced. Meanwhile, the load of accessing global storage is also reduced.

D. Cache refreshing

There are two kinds of cache refreshing in SRMC. The first one is an event-based one. When an existing service is updated or removed from SRMC, a message of the event will be sent to all caches via Cache Manager. This message will trigger the event-based refreshing. All the caches will update the cached information with the data in message when they receive an update message. When they receive a delete message, all the caches will delete the service if they cached it.

The second kind of cache refreshing is periodical refreshing. The work of periodical refreshing has four parts. The first part is to process the service consumers of *freshmen* group who are new users of SRMC. We can set a threshold of the number of history records for single service consumer. In periodical refreshing, Cache Manager scans the *HistoryRequestsList* of group *freshmen* to find out all the service consumers whose number of history records has greater than the threshold. For each of such service consumers, the Cache Manager will assign it into group(s) by the clustering method in Section IV and send its history requests records to the assigned group(s). The assigned group(s) will parse the records and add them to *RequestsList* and *HistoryRequestsList* tables according to their content. Finally, the Cache Manager will delete the service consumer and all the records relate to it in group *freshmen*.

The second part of periodical refreshing is to fade the history records. Since we use fading memory method to cluster service consumers, we need to fade the records in *HistoryRequestsList* table. The Cache Manager deletes all the expired records according their timestamps. The period of validity of records is specified by provider of service registry. The *RequestsList* table will be scanned as long as any record of *HistoryRequestsList* table is deleted. If a record of *RequestsList* table is not be referred by any record of *HistoryRequestsList* table any more, it will be deleted.

The third part of periodical refreshing is to process all requests in *RequestsList* table in order to refresh the records in *RespondsList* table. This task mainly focuses on the new registered services because they cannot be discovered without this task. For most service discovery requests, they can be processed by accessing caches. As a result, the new registered services are hardly discovered because they are not in any caches but in global storage. Since it is a time-consuming task, it should be executed at idle time.

The forth part of periodical refreshing is to re-cluster service consumers. Since each record of *HistoryRequestsList* table has a timestamp and the expired records will be deleted, the algorithm of re-clustering service consumers adopted by Cache Manager is a fading memory method. This task is also a time-consuming one, so it also should be executed at idle time.

The four parts of periodical refreshing can be executed together. But more reasonable mechanism is they have their own periods which may be different with each other, because some of them are time-consuming tasks which should have longer periods and others are not time-consuming tasks which may have shorter periods.

IV. Clustering Service Consumers

There are two steps in service consumer clustering. The first step is to find the application domains in which the service consumers are interested, that is, to find the concerns of service consumers. We use an OWL-S based description model defined in [24] to describe services, because it facilitates behavior-aware service discovery:

$$S = \{ I^C, O^C, \Phi(I^C, O^C, P^P), Ct \} \qquad (1)$$

Where $I^C = \{I_1, \dots, I_n\}$ represents a set of inputs with types of concepts; $O^C = \{O_1, \dots, O_m\}$ represents a set of outputs with types of concepts; $\Phi(I^C, O^C, P^P)$ is the semantic relationship holding between I^C and O^C variables, and is represented in the form of OWL triples; $P^P = \{P_1, \dots, P_m\}$ is a set of ontology properties represent predicates that relating I^C and O^C; Ct is the constraints set imposed on S including QoS constraints. This model is also used by service consumer to describe request of service discovery.

When a service consumer issued a certain number of requests of service discovery and obtained the corresponding number of responds, we can get its history by accessing the *HistoryRequestsList* table and cluster its history records to find its concerns. Since the inputs and outputs in $\Phi(I^C, O^C, P^P)$ are subset of I^C and O^C, we only need focus on I^C and O^C. Each element in I^C and O^C is a concept of ontology which is a taxonomy tree of an application domain. To cluster history records of service consumers, we assign a unique value to each concept of ontology which indicates its position in depth-first traversal of taxonomy tree. Then we calculate the average position of each request in *HistoryRequestsList* table as equation (2):

$$P_R^d = \frac{\sum_{i=1}^{n} P_{I_i}^d + \sum_{j=1}^{m} P_{O_j}^d}{n+m} \qquad (2)$$

Where P_R^d represents the average position of request R; $P_{I_i}^d$ represents the position of input I_i; $P_{O_j}^d$ represents the position of output O_j; *n* represents the number of inputs; *m* represents the number of outputs.

For each request, we make a pair $P_R^t = < P_R^d, t_R >$, where t_R represents the timestamp of request R. For a single service consumer, we get a dataset $P_R^t = \{P_{R_1}^t, P_{R_2}^t, \dots, P_{R_n}^t\}$, $P_{R_i}^t$ represents the pair of request R_i. We use OPTICS algorithm [25] to cluster $P_{R_i}^t$. The reason we choose OPTICS algorithm is that it can filter the noise. What we need is to find the concerns of a service consumer. But it is common that a service consumer issues requests to find services which are not in its concerned domain. We need to filter such requests because they are noise for finding concerns. Each pair is mapped onto a point on coordinate, where $< P_R^d, t_R >$ is mapped onto the <x, y> value. We use the euclidean distance to measure the distance between two points. For the two parameters of OPTICS, ε , which describes the maximum distance (radius) to consider, and *MinPts*, describing the number of points required to form a cluster, provider of service registry can specify it according to the subdivision level of application domains. The clusters obtained by OPTICS algorithm are the concerns of a service consumer. The clusters determine which the second lowest caches the service consumer should be assigned into.

The second step of service consumer clustering is to further cluster the service consumers in same second lowest cache into different groups by their QoS preference. The QoS constraints in a service discovery request make up a high-dimension data structure because the multiple quality attributes are orthogonal. Since the high-dimension data clustering is a quite complex problem, we use multiple *k*-means clustering instead. Suppose the number of the quality attributes we focus on is *n*. We use *k*-mean clustering to divide i^th attribute into K_i ranges, thus, the whole data space is divided into $\prod_{i=1}^{n} K_i$ subspaces.

Before QoS clustering, we need to preprocess data by merging the ranges of quality attributes in the requests issued by same service consumer. For each service consumer, we get all the history requests in the cache and iterate them. For each of quality attribute, we iterate the QoS constraints in all history requests and merge the acceptable range by join them. Then, we get the middle point of merged range as the prefer point of the service consumer.

After preprocess, we use *k*-mean clustering to cluster the service consumers. Consequently, the service consumers are clustered into $\prod_{i=1}^{n} K_i$ groups. Hereto, the mission of service consumer clustering is accomplished.

The whole process is as the following pseudo-code:

```
function Domain_Clustering(Collection cons)
  // cons: the service consumers
  Collection<Collection> doms = {};
  //doms: 2-dimensional collection to hold the relationship
  // between service consumers and application domains.
  for each consumer in cons
    Collection history = getHistory(consumer);
    // his: history records of consumer
    Collection positionpairs ={};
    // positionpairs: position pairs of all history records
    for each $R_i$ in history
      $P^d_{R_i}$ = position($R_i$);
      $P^t_{R_i} = makepair < P^d_{R_i}, t_{R_i} >$;
      positionpairs = positionpairs $\cup \{P^t_{R_i}\}$;
    endfor
    Collection domains = {};
    domains = opticsClustering(positionpairs);
    // domains: domains of the service consumer
    for each dom in domains
      if dom is not in doms
        add(doms, {dom});
        // add new collection into doms to indicate
        // a new application domain
      add(doms[dom], consumer);
      // add consumer into doms[dom] to indicate the
      // consumer is assigned to this application domaim.
    endfor
  endfor
endfunction
```

```
function QoS_Clustering(Collection<Collection> doms)
  // doms: the result of Domain_Clustering
  for each dom in doms
    for each consumer in dom
      Collection history = getHistory(consumer);
      // his: history records of consumer
      for each $q_i$ //$q_i$ is a quality attribute
        range = mergeRange(history);
        // range: the merged range of $q_i$
        float p = concernPoint(range);
        // p: the concern point of $q_i$
      endfor
    endfor
    Collection<Collection> groups = {}
    // groups: the result of clustering by a single attribute.
    for each $q_i$
      add(groups,{k-mean($q_i$)});
      // k-mean($q_i$): clustering consumers by $q_i$
    endfor
    groups = orthogonalize(groups)
    // the final result comes from the groups orthogonalizing
  endfor
endfunction
```

V. Instantiation of SRMC

Since we are studying on another model for building service registry which is deployed as a cloud application, we shared the experimental environment with it. The shared experimental environment is a 10-computer environment in which we runs one Cloud Controller, three Cluster Controllers, and six Cluster Nodes. We install Apache Hadoop [26] on the cloud to store all the necessary data.

We simulate the semantic descriptors and QoS feedbacks of more than 10,000 web services to accumulate necessary data. All the web services belong to five domains: *travel, shopping, e-learning, sport*, and *accounting*. We designed more than 1,000 request templates. They are used by 500 service consumers we simulate to access SRMC with different frequency. After accumulating more than 100, 000 requests, we find that cache hit ratio varies from 40% to 80% when we set different values to the parameters ε and *MinPts* of clustering and period of periodical refreshing.

The running results of this instance of SRMC have shown that SRMC is effective to reduce the times of accessing global storage and the amount of data searched in service discovery. However, our experiments are not enough to precisely measure and analyze the performance of SRMC. Since the hosting environment we establish is an experimental one which is quite different from the true cloud platform, there is no real user to access it. As a result, we have no enough valid data to do more analysis. The data we simulate are not massive enough to check the performance of SRMC. After all, when the amount of data is not massive enough, the improvement of performance is not notable. So in next work, we will deploy it into a commercial environment and compare it with other service registry to precisely measure and analyze its performance.

VI. CONCLUSION

In order to improve the efficiency of service discovery and release the load of service registry, this paper proposes a service registry model named as SRMC (Service Registry with Multilevel Caches) which clusters the service consumers into groups according to their searching similarity and sets up a multilevel cache for all groups to improve the performance of service discovery. The multilevel caches of SRMC are refreshed by a hybrid mechanism which includes event-based refreshing and periodical refreshing. The basis of refreshing and clustering is the history records of service discovery requests issued by service consumers.

The running results of an instance of SRMC deployed in an experimental environment have shown that SRMC is effective to reduce the times of accessing global storage and the amount of data searched in service discovery. In next work, we will deploy it into a commercial environment and compare it with other service registry to precisely measure and analyze its performance.

ACKNOWLEDGMENT

As a visiting scholar of Georgia Institute of Technology, Hao-peng Chen thanks Georgia Institute of Technology and Prof. Ling Liu, a Professor of College of Computing at Georgia Institute of Technology, for their supports.

This paper is also supported by the Project of Daystar of Shanghai Jiao Tong University.

REFERENCES

[1] Michael Armbrust, Armando Fox, et.al, Above the Clouds: A Berkeley View of Cloud Computing, http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf, 2009

[2] Luc Clement, Andrew Hately, Claus von Riegen, Tony Rogers, *UDDI Version 3.0.2*, http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm, 19 October 2004.

[3] David Booth, Canyang Kevin Liu, *Web Services Description Language (WSDL) Version 2.0 Primer,* http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/, 26 June 2007

[4] David Martin, John Domingue, Michael L.Brodie, Frank Leymann, *Semantic Web Services, Part1*, IEEE Intelligent Systems, September/October 2007, volume 22, no.5, pp.12-17.

[5] McIlraith, S., Son, T.C. and Zeng, H. *Semantic Web Services*, IEEE Intelligent Systems. Special Issue on the Semantic Web. 16(2):46--53, March/April, 2001

[6] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila Mcllraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, Katia Sycara, *OWL-S: Semantic Markup for Web Services*, http://www.w3.org/Submission/OWL-S/, 22 November 2004.

[7] Jos de Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Uwe Keller, Michael Kifer, K.Birgitta, Jacek Kopecky, Ruby Lara, Holger Lausen, Eyal Oren, Axel Polleres, Dumitru Roman, James Scicluna, Michael Stollberg, *Web Service Modeling Ontology(WSMO)*, http://www.w3.org/Submission/WSMO/, 3 June 2005.

[8] Steve Battle, Abraham Bernstein, Harold Boley, Benjamin Grosof, Michael Gruninger, Richard Hull, Michael Kifer, David Martin, Sheila McGuinness, Jianwen Su, Said Tabet, *Semantic Web Services Framework (SWSF) Overview*, http://www.w3.org/Submission/SWSF/, 9 September 2005.

[9] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth, Kunal Verma, *Web Service Semantics - WSDL-S Version 1.0*, http://www.w3.org/Submission/WSDL-S/, 7 November 2005.

[10] Mark Burstein, Christoph Bussler, Michal Zaremba, Tim Finin, Michael N.Huhns, Massimo Paolucci, Amit P.Sheth, Stuart Williams, *A Semantic Web Services Architecture*, IEEE Internet Computing, September 2005, volume 9, issue 5, pp.72-81.

[11] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, Richard Franck, *Web Service Level Agreement (WSLA) Language Specification*, http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf, 2003/01/28

[12] Qi Yu, Xumin Liu, Athman Bouguettaya, Brahim Medjahed, *Deploying and managing Web services: issues, solutions, and directions*, The VLDB Journal (2008) 17:537–572

[13] Frank Manola, Eric Miller, *RDF Primer*, http://www.w3.org/TR/2004/REC-rdf-primer-20040210/, 10 February 2004

[14] Tom Cunningham, Kurt Stam, and The jUDDI Community, *jUDDI Dev Guide*, http://ws.apache.org/juddi/docs/3.0/devguide/html/index.html, 2009

[15] Seekda, http://webservices.seekda.com/, May 2010

[16] Massimo Paolucci, Takahiro Kawamura, Terry R.Payne, and Katia Sycara, *Semantic Matching of Web Services Capabilities,* The First International Semantic Web Conference on The Semantic Web(ISWC 2002), 2002, pp.333-347.

[17] Matthias Klusch, Benedikt Fries, Katia Sycara, *Automated Semantic Web Service Discovery with OWLS-MX*, The Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, 8-12 May 2006, pp.915-922.

[18] Kyriakos Kritikos, Dimitris Plexousakis, *OWL-Q for Semantic QoS-based Web Service Description and Discovery*, First International Joint Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web, PP123-137, 2007

[19] *GT Information Services: Monitoring & Discovery System (MDS)*, http://www.globus.org/toolkit/mds/

[20] *Tivoli Monitoring,* http://www-01.ibm.com/software/tivoli/products/monitor/

[21] Siming Xiong, Haopeng Chen, *QMC: A Service Registry Extension Providing QoS Support*, Proceedings of 2009 International Conference on New Trends in Information and Service Science (NISS 2009), PP.145-151, Beijing, China, 2009.6.30-2009.7.2, ISBN:978-0-7695-3687-3/09.

[22] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar and J. Miller, *METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services*, Journal of Information Technology and Management, Special Issue on Universal Global Integration, Vol. 6, No. 1 (2005) pp. 17-39. Kluwer Academic Publishers.

[23] SHU-JIA WANG, HAO-PENG CHEN, *A Web Service Selecting Model Based on Measurable QoS Attributes of Client-Side*, Proceedings of 2008 International Conference on Computer Science and Software Engineering (CSSE 2008), PP.385-389, Wuhan, China, 2008.12.12-2008.12.15, ISBN: 978-0-7695-3336-0/08.

[24] Barhamgi, M. Benslimane, D. Ouksel, A.M. LIRIS Lab., Claude Bernard Univ., Villeurbanne, *SWSMS: A Semantic Web Service Management System for Data Sharing in Collaborative Environments*, Proceedings of 3rd International Conference on Information and Communication Technologies: From Theory to Applications (ICTTA 2008), pp.1-5, April 2008.

[25] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, Jörg Sander. *OPTICS: Ordering Points To Identify the Clustering Structure*. ACM SIGMOD international conference on Management of data 1999. ACM Press. pp. 49–60.

[26] Apache, "*Apache Hadoop*." http://hadoop.apache.org/ . 2010.