ServiceStore: A Peer-to-Peer Framework for QoS-aware Service Composition

Jun Jin[1], Yu Zhang[2], Yuanda Cao[1], Xing Pu[1], Jiaxin Li[1]

[1] Beijing Laboratory of Intelligent Information Technology, School of Computer Science, Beijing Institute of Technology, Beijing, China
[2] School of Computer Science, Beijing University of Civil Engineering and Architecture, Beijing, China
flea.miss@gmail.com

Abstract. Web service composition is to integrate component services for providing a value-added new service. With the growing number of component services and their dynamic nature, the centralized composition model can't manage them efficiently and accurately. In this paper, we proposed a distributed hash table (DHT)-based peer-to-peer (P2P) service composition framework, called ServiceStore. Compared with the central control in centralized model, in our ServiceStore, service selection and composition are distributed to the involved task brokers, requesting nodes and service nodes. Furthermore, a simple parallel service selection approach which can still satisfy global constraints is proposed and implemented in our multi-role cooperation (MRC) protocol. The results of experimental evaluation show that ServiceStore can achieve high scalability and efficiency

Keywords: P2P; DHT; service selection; task broker; MRC protocol.

1 Introduction

Standardized web service as a main solution of service-oriented computing provides a flexible and convenient way for applications to select and integrate basic services to form new value-added services. Many applications bring service composition into practice, Figure 1 shows a service composition example.

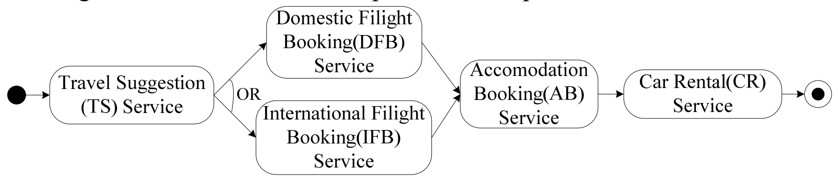

Travel Suggestion (TS) Service
OR
Domestic Filight Booking(DFB) Service
International Filight Booking(IFB) Service
Accomodation Booking(AB) Service
Car Rental(CR) Service

Fig. 1. Web service composition example

* This project supported by Provincial Natural Science Foundation of Hebei Province, China (Grant No. F2009000929).

In Figure 1, a traveler requests a tourism planning from a service provider and existing atomic services can't satisfy this composite request by themselves. Service composer can integrate the fight booking, accommodation booking and car rental by using BPEL and execute the BPEL file on an engine such as BEPL4J[6]. Furthermore, the traveler also associates the request with some global QoS constraints (e.g. total price $\leq$ \$5 and response time $\leq$ 1 minute). The service composer must ensure that the integrated QoS attributes satisfy the global constraints.

With the aim to realize efficient service composition and resource utilization in distributed environment, our contribution of this paper can be briefly stated as follows:

(1) A resource-efficient service selection approach. In order to pick out appropriate component services from the alternative ones that provide identical functionality but distinct QoS attributes and resource states. With the method proposed in [7], we design a simple local selection approach that not only satisfies the global constraints but also provides efficient resource utilization.

(2) A multi-role cooperation (MRC) protocol. Each peer in MRC protocol can plays four roles – query originator, query decomposer, task broker and coordinator. With the help of this protocol, a composite service request can be solved efficiently.

Assuming that the component service is atomic, the rest of this paper is organized as follows. Section 2 gives a brief overview of related work. Section 3 introduces the system architecture. Our multi-role cooperation protocol for distributed service selection and composition is presented in Section 4. Section 5 gives a simple proactive failure recovery approach. Experimental evaluations are presented in Section 6. Finally, Section 7 gives conclusions and our future work.

2 Related work

The problem of service composition has drawn many research institutes in recent years. As centralized orchestration lacks scalability and is easy to break down, more research work concentrates on decentralized orchestration. P2P system which is famous for its self-organizing and scalability has been adopted by many projects as their basic architecture.

SELF-SERV [4,5] adopts an orchestration model based on P2P interactions between service components in the composition which provides greater scalability than the approaches based on central scheduler. They propose the concept of service community and a decentralized execution model. But service discovery and selection are not considered.

WSPDS [7] uses an unstructured P2P system – Gnutella [2] as its infrastructure. With probability flooding technique and the subsequent content-based network, the overhead of query dissemination is significantly reduced. In WSPDS, each servent (acts as both server and client) is composed of two engines, communication engine and local query engine, having tasks for communication, collaboration and issue query. To discover a requested service, each servent receives the query and forwards it to the neighbor that has the most similar identity to the query. However, QoS-aware service selection is not considered in WSPDS.

SpiderNet [10] proposes a QoS-aware service composition framework that uses DHT based P2P system as its infrastructure. Using (key, value) pairs, service discovery is very efficient. SpiderNet, each peer acts the same as in WSPDS, both server and client. SpiderNet uses a probing protocol to collect needed information and perform parallel searching of multiple candidate service graphs. Although the authors use probing budget and quota to control each request's probing overhead, they don't consider the situation when dealing with large number of concurrent requests and the service session setup time can't be guaranteed.

Note that all the research work above treats service discovery and selection sequentially which is costly and unnecessary. Integer programming [8] can be used to find optimal selection of component services [15]. Alrifai et al. [9] adopts it and changes service selection into a parallel fashion. In this paper, we adopt this idea and propose a distributed broker-based framework with MRC protocol to achieve QoS-aware and resource-efficient service composition.

3 System Architecture

The ServiceStore system is implemented as a distributed middleware infrastructure, which can effectively map user's composite service request into a set of component services in the P2P service overlay.

3.1 Three-layer architecture

The architecture of ServiceStore is a three-layer structure (see Figure 2). The bottom service overlay is constructed by all service nodes with their registry component services and links mapped from underlying physical network. To facilitate node location in service overlay, a unique identifier *nodeID* is assigned to each service node. Each component service provides its functionality with advertised QoS attributes, however these non-functional values are mutative yet, e.g., the response time will be high when network congestion emerges, we classify them into two parts: (1) static metadata denoted as $MD_s(s_{ij})$, a profile of the component service, including function name, its location $LC(s_{ij})$ and IO parameter list; (2) dynamic metadata denoted as $MD_d(s_{ij})$, including recent statistical QoS attribute values $Q(s_{ij}) = [q_1(s_{ij}),\ldots,q_{M1}(s_{ij})]$ and instant workload $WL(s_{ij}) = [r_1(s_{ij}),\ldots,r_{M2}(s_{ij})]$, where M_1 and M_2 are the sizes of QoS vector and workload vector respectively, $Q(s_{ij})$ and $WL(s_{ij})$ are mutative to describe the performance of s_{ij}. All these component services can be classified into different *service class*es with each class sharing the same functionality. From the perspective of delivered functionality, each service class is identified as a *service task*, denoted as T_i. For clarification, we use $S_i = \{s_{i1},\ldots, s_{ij},\ldots, s_{iL}\}$ to denote the service class corresponding to T_i, where s_{ij} represents the *j*-th component service being able to fulfill the service task T_i, and L is the size of S_i. The scenario of $L > 1$ indicates that the service task T_i is able to be realized by multiple candidate services, which can differ in their respective QoS attributes.

To fast locate components services, we adopt Distributed Hash Table technique [13] to manage the component services in ServiceStore. DHT systems use (key, value) pairs to store and retrieve the value associated with a given key. We design a hash function to map a function name (keywords of function name) to a nodeID (GUID). After applying it, the metadata list of component services that own similar function names and thus belong to the same service class (including $MD_d(S_i)$ and $MD_s(S_i)$) are stored on the same service node, here called task broker. We use *Broker*(S_i) to denote the task broker corresponding to S_i. In Figure 2, the middle layer shows the task brokers of all component services on the service overlay.

The top layer is a function graph that comes from a user's composite service request. A request is denoted as $R = \langle F, Q^r \rangle$, where $F = \{T_1,\ldots,T_N\}$is a function graph composed by a set of tasks (composition relations); $Q^r = [q_1^r,\ldots, q_{M1}^r]$ shows a user's QoS requirements.

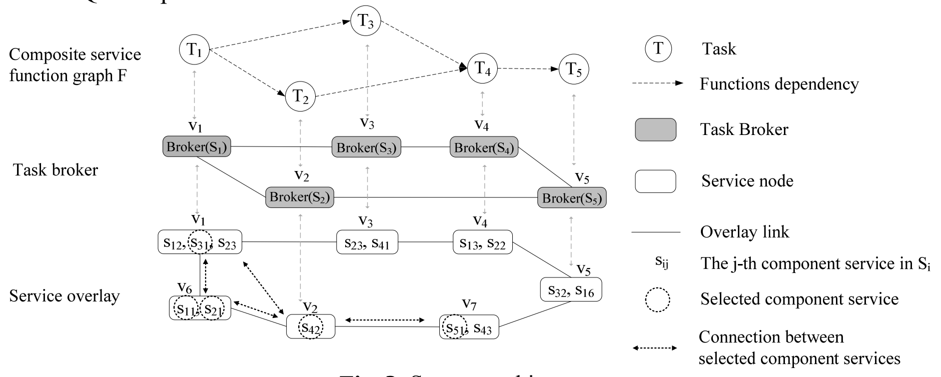


Fig. 2. System architecture

3.2 QoS-aware service composition problem

In ServiceStore, given a composite service request R with $F = \{T_1,\ldots,T_N\}$ and $Q^r = [q_1^r,\ldots, q_{M1}^r]$, the aim of service composition is to find a list of component services, we call it *Execution Plan* (*EP*), that can realize each task in F and satisfy each quality attribute in Q^r.

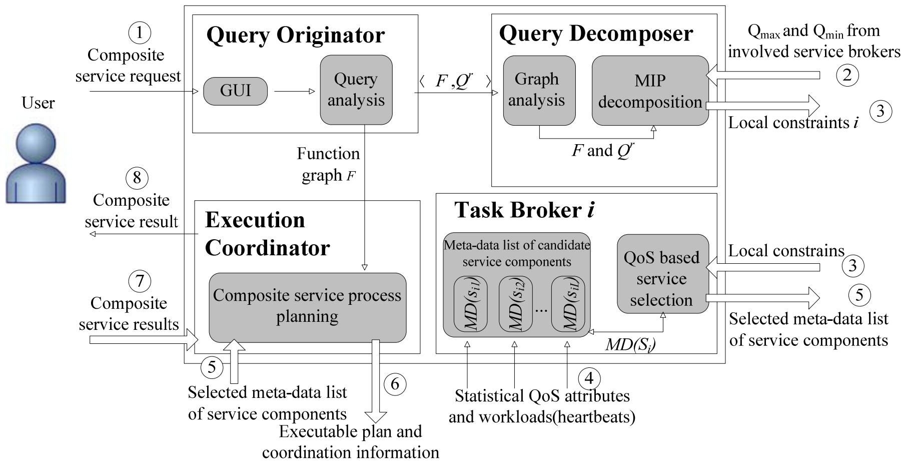
Figure 2 shows a few of selected components services and their links with dotted lines and circles respectively, which make up an *EP* for R.

4. Multi-role Cooperation (MRC) Protocol

4.1 Four roles in MRC protocol

In MRC protocol, each peer plays four roles: (1) **Query originator** provides visualized specification environment [1] to help users issue composite service requests. (2) **Query decomposer** receives the quality levels and converts the global constraints Q^r into local constraints and sends them to the involved task brokers. (3)

Task broker maintains the meta-data list of alternative component services, and receives local constraints from requesting node and returns the most appropriate component service according to the heartbeats from every component service. (4) **Execution coordinator** the execution of the selected component services using composition pattern (e.g. sequential, conditional, concurrent, loop) and return the results to the user.



1
Composite service request
User
Query Originator
GUI
Query analysis
⟨F, Q^r⟩
Query Decomposer
Graph analysis
MIP decomposition
F and Q^r
Q_max and Q_min from involved service brokers
2
Local constraints i
3
Function graph F
8
Composite service result
Execution Coordinator
Composite service process planning
7
Composite service results
5
Selected meta-data list of service components
6
Executable plan and coordination information
Task Broker i
Meta-data list of candidate service components
MD(s_i1)
MD(s_i2)
...
MD(s_iL)
QoS based service selection
MD(S_i)
Local constrains
3
5
Selected meta-data list of service components
4
Statistical QoS attributes and workloads(heartbeats)

Fig. 3. A peer plays four roles in ServiceStore

4.2 MRC protocol

Our MRC protocol includes five major steps shown in Figure 3:

Step 1. Issue a composite service request. With the help of prestored composite service templates, a user at a host specifies a composite service request *R* using GUI. After query analysis, $\langle F, Q^r\rangle$ is sent to the query decomposer and *F* is sent to execution coordinator.

Step 2. Decompose global QoS constraints into local ones. Through graph analysis, the involved tasks and global QoS constraints Q^r are sent to the MIP decomposition [7]. Then the query decomposer achieves *N* local constraints and sends them to the corresponding task brokers.

Step 3. Select feasible component services locally. To achieve accurate selection, each component service sends heartbeat message with $MD_d(s_{ij})$ indicating the states of s_{ij} to $Broker(S_i)$. Then each corresponding task broker performs local selection and returns the selected service candidates to the requesting peer separately. And for efficient resource utilization, task broker updates $WL(s_{ij})$ of the optimal selected candidate service(e.g. its concurrent connection plus one). The details of this step will be described in the next subsection.

Step 4. Form executable plans. Upon the receipt of all service candidates from the corresponding task brokers, the execution coordinator begins to compose them into *EP* according to *F* and sends to the first component service in *EP*.

Step 5. Coordinate the execution of the *EP*. When receives an *EP*, a component service checks if its function is contained in the *EP*. If its function exists, the component service begins to execute and output results to the next component service according to *EP*. Finally, after all component services complete their executions, the last component service sends the results to the requesting peer (user) and each involved task broker recovers $WL(s_{ij})$ of the component services in *EP* .

4.3 Service selection

As each task broker keeps the information of all alternative component services, upon the receipt of local constraints, it uses them as the bound and performs service selection for the corresponding service class independently.

Given the following parameters: (1) The received local constraints for service class S_i: $Q^c(S_i) = [q_k(S_i) \mid 1 \le k \le M_1]$; (2)The dynamic metadata of component service s_{ij}: $MD_d(s_{ij}) = <Q(s_{ij}), WL\ (s_{ij})>$, where $Q(s_{ij}) = [q_k(s_{ij}) \mid 1 \le k \le M_1]$ and $WL\ (s_{ij}) = [r_k(s_{ij}) \mid 1 \le k \le M_2]$. We compute the utility $U(s_{ij})$ of the *j*-th service candidate in class S_i as

$$U(s_{ij}) = \sum_{k=1}^{M_1} \frac{Q_{\max}(i,k) - q_k(s_{ij})}{Q_{\max}(i,k) - Q_{\min}(i,k)} \times \omega_k \qquad \textbf{(1)}$$

where $\sum_{k=1}^{M_1} \omega_k = 1$, $Q_{\max}(i,k)$ and $Q_{\min}(i,k)$ represent the max and min value of the *k*-th quality attribute in class S_i.

Generally, the component service with the highest $U(s_{ij})$ is always selected as it provides the best capability. However, with the increasing number of invocation, its actual performance may become poor. Thus, a resource utility function $UR(s_{ij})$ is needed for representing a component service's resource utility.

$$UR(s_{ij}) = \sum_{k=1}^{M_2} \frac{ra^{v_{ij}}(k)}{r_{\max}^{v_{ij}}(k)} \times \omega_k \qquad \textbf{(2)}$$

where $\sum_{k=1}^{M_2} \omega_k = 1$, $r_{\max}^{v_{ij}}(k)$ and $ra^{v_{ij}}(k)$ represent the max and available value of the *k*-th resource (e.g. memory) in node v_{ij} (suppose s_{ij} is on the node v_{ij}).

Here three requirements need to be considered in service selection: The selected component services (1) satisfy the global constraints; (2) achieve a large resource utility value; (3) achieve the optimal utility. The first requirement is very essential, as our major aim is to achieve a feasible solution for the user. The last requirement is set the lowest priority, even if big value does please the user, it may cause the system unstable yet for resource competition. Hence, for better stability and resource utility, the second requirement has higher priority than the last one. We apply these requirements in the algorithm 1.

```
  Input: Decomposed local constraints $Q^c = [q_1, q_2, \dots, q_{M1}]$ and
metadata list of the service class $S$ $MD_d(S) = \{<[q_1(s_j), \dots,
q_{M1}(s_j)], [r_1(s_j), \dots, r_{M2}(s_j)]> \mid 1 < j < L, s_j \in S\}$
  Output: Sorted list of component services $S_{out} = \{s_1, s_2, \dots\}$
  1. Initialize list $S_{out}$
  2. for all $s_j$ in $S$ do
  3.   for all $k$, $1 \le k \le M_1$, that $q_k(s_j) \in MD_d(S)$ do
```

```
4.      if $q_k(s_j) > q_k$ then break
5.      end if
6.      set $k = k + 1$
7.    end for
8.    if $C$ then
9.      add $s_j$ to $S_{out}$
10.     compute $U(s_j)$ and $UR(s_j)$
11.   end if
12. end for
13. Sort $S_{out}$ according to $UR(s_j)$
14. return $S_{out}$
```

Algorithm 1. Local service selection

Our aim is to get a list of feasible component services from each involved task broker for the completion of *EP*. All feasible component services must meet the aforementioned three requirements. Algorithm 1 shows the service selection process. With the decomposed QoS constraints for service class *S* and meta-data of *S* as input parameters, the *Broker*(*S*) begins to run this algorithm. *Broker*(*S*) checks every QoS constraint (e.g. price $\leq$ \$2) for every candidate service. If any QoS attribute was beyond the upper bound of the according given QoS constraint, *Broker*(*S*) would discard that component service. Therefore, the first requirement is satisfied during service selection. After S_{out} filled with all feasible component services, we sort it according to $UR(s_j)$ of every feasible component service. For fault tolerance, each involved task broker returns more than one candidate services.

5 Proactive failure recovery

Failure recovery is very essential in dynamic environment [12]. ServiceStore provides a proactive failure recovery mechanism to maintain the quality of service composition during system runtime. As task broker is very crucial in MRC protocol, we discuss the situation when task brokers fail.

ServiceStore maintains a small number of backup task brokers for each service class, for fault tolerance, when a peer publishes a component service, the metadata of this component service are stored into more than one task brokers [10][1].

(1) **Backup task broker computation.** Applying the secure hash algorithm to the strings formed by concatenating two or three the component service's function name, we can achieve different resourceIds, thus different task brokers to maintain the meta-data list of this service class. For clarification, we give the following simple functions:

String Concatenate(*String functionname, int n*) : concatenate the *function name* for *n* times. *GUID SH*(*String functionname*) : apply the secure hash algorithm to the *functionname.* And these three task brokers of the service class S_i with the function name *name_i* are calculated as the flowing:

Broker1(S_i) = *SH*(*Concatenate*(*name_i*, 1))

[1] Current implementation keeps two additional backups.

$Broker2(S_i) = SH(Concatenate(name\_i, 2))$

$Broker3(S_i) = SH(Concatenate(name\_i, 3))$

For example, three task brokers of car rental service class are calculated by *SH*(*Concatenate* ("carrental",1)), *SH*(*Concatenate* ("carrental",2)) and *SH*(*Concatenate* ("carrental",3)) respectively.

(2) **Backup task broker selection.** If one of the task brokers failed, we would adopt the rule depicted in Figure 4.

(3) **Backup task broker synchronization.** As service selection depends on the meta-data list of the candidate services, each component service periodically sends the MD_d to the task brokers (including the backup ones).

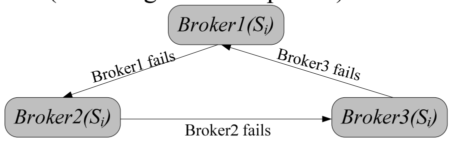


Fig. 4. A simple rule for task broker selection

6 Implementation and evaluation

The experiment is carried out on PeerSim [3] and the decentralized service overlay is implemented based on DHT based P2P system Pastry [11,14]. Please note that we set the same parameters during each round of simulation: 50 service functions in service overlay, 2 composite service requests during each time unit and 3000 time units each round of simulation lasts.

First, we evaluate the resource utility of our MRC protocol. For simplicity, we use the number of each candidate component service's concurrent link to measure the resource efficiency of our selection approach. We use a 1000 nodes service overlay, with each node provides component services whose function is selected from 50 pre-defined functions and each function has 4 duplicates. Each composite service request contains 3 functions. Every component service's number of concurrent link will be increased by one when it is selected and reduced by one after working for 80-90 time units. Figure 5 illustrates the 4 duplicates almost have the same number of concurrent connections during 2000 running time units.

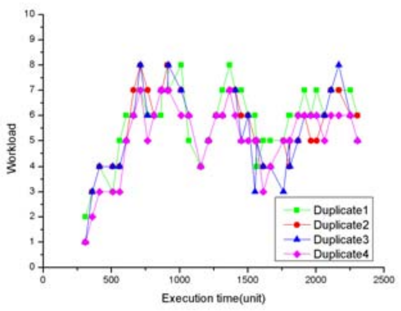


Fig. 5. Concurrent connections of each service duplicate

Second, we evaluate the max and average hops when discovering a component service in different size service overlay. As the expected number of routing hops in DHT based P2P system Pastry is O(logN) [14], in Figure 6 although the service overlay size grows from 500 to 20000, the number of average hop increased very slowly and only few hops reach the max number.

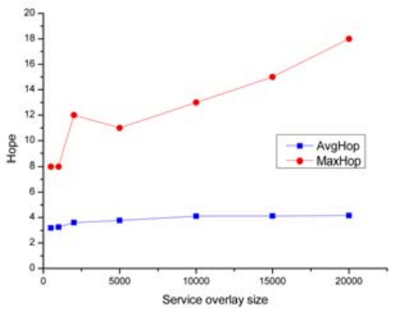


Fig. 6. Max and average hops for discovering a component service

Third, we measure the average composite service session setup time with different service overlay size and different function number of each request. Figure 7 illustrates the average composite service session setup time when the function number is 3, 5, 10, 20 and the service overlay size varies from 50-20000. Thanks to parallel service selection, with the increasing service overlay size and function number in each request, composite service setup time increases slowly and doesn't multiples with the function number.

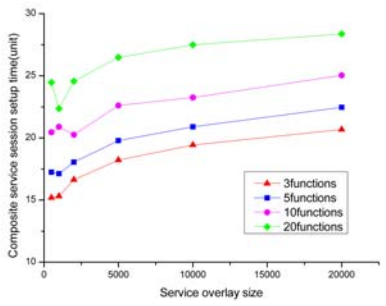


Fig. 7. Composite service session setup time

7 Conclusion and future work

In this paper, we have presented a P2P service composition framework called ServiceStore. The main contributions are: 1) ServiceStore provides a fully decentralized architecture implemented by using distributed task brokers as coordinators; 2) ServiceStore provides a simple MRC protocol for service composition; 3) Our evaluation shows that ServiceStore scales well with service overlay size increasing and achieves good resource efficiency.

Since the task broker is a critical role in MRC protocol and the failure recovery needs more time especially near the end of composite service execution, in the future we will integrate behavior prediction into our service composition framework which can help us to improve system stability.

Preference

1. Svend Frolund and Jari Koistinen.: Quality of service specification in distributed object systems design. Distributed Systems Engineering Journal, 5(4), December (1998).(QML, which is a language for the description of QoS using XML.)
2. Gnutella. http://gnutella.wego.com/.
3. PeerSim. http://peersim.sourceforge.net/
4. B. Benatallah, M. Dumas, Q.Z. Sheng, and A.H. Ngu.: Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services, Proc. Int'l Conf. Data Eng. (ICDE), pp. 297-308. Feb. (2002).
5. Benatallah, B., Sheng, Q., and Dumas, M.: The Self-Serv environment for web services composition. IEEE Internet Computing, 7(1), pp. 40–48, (2003).
6. Business Process Execution Language for Web Services Java Run Time (BPWS4J). http://www.alphaworks.ibm.com/tech/bpws4j.
7. Farnoush Banaei-Kashani, Ching-Chien Chen and Cyrus Shahabi.: WSPDS: web services peer-to-peer discovery service, In Proc. of the 5th Int'l Conference on Internet Computing (IC), pp. 733-743. Las Vegas, Nevada, June (2004).
8. G. L. Nemhauser and L. A.: Wolsey, Integer and Combinatorial Optimization, Wiley-Interscience, New York, NY, USA, (1988).
9. Mohammad Alrifai and Thomass Risse.: Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition, In Proc. of the 18th Int'l World Wide Web(WWW), Madrid, Spain, April (2009).
10. Xiaohui Gu, Klara Nahrstedt, and Bin Yu.: SpiderNet: An Integrated Peer-to-Peer Service Composition Framework, In Proc. of the 13th Int'l Symp. on High-Performance Distributed Computing (HPDC), IEEE Computer Society, pp. 110-119. Honolulu, Hawaii, (2004).
11. PeerSim-Pastry, http://code.google.com/p/peersim-pastry/
12. B. Raman and R. H. Katz.: Load Balancing and Stability Issues in Algorithms for Service Composition, Proc. of IEEE INFOCOM 2003, San Francisco, CA, April (2003).
13. Distributed hash table, http://en.wikipedia.org/wiki/Dist-ributed_hash_table.
14. Antony Rowstron and Peter Druschel.: Pastry:Scalable, distributed object location and routing for large-scale peer-to-peer systems, Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms(Middleware). Heidelberg, Germany, November (2001).
15. D. Ardagna, B. Pernici.: Global and Local QoS Constraints Guarantee in Web Service Selection, 3rd IEEE International Conference on Web Services (ICWS), pp.805-806. Orlando, FL, USA, July (2005).