Efficient QoS management for QoS-aware web service composition

Shangguang Wang*

State Key Laboratory of Networking and Switching Technology,
Beijing University of Posts and Telecommunications,
Haidian, Beijing, 100876, China
Email: sguang.wang@gmail.com
*Corresponding author

Xilu Zhu

The Research Institution of China Mobile,
China Mobile Limited,
Changsha, Hunan, 425200, China
Email: xilu.zhu@gmail.com

Fangchun Yang

State Key Laboratory of Networking and Switching Technology,
Beijing University of Posts and Telecommunications,
Haidian, Beijing, 100876, China
Email: fcyang@bupt.edu.cn

Abstract: In this paper, we propose an efficient QoS management approach for QoS-aware web service composition. In the approach, we classify web services according to theirs similarity and then design a QoS tree to manage the QoS the classified web services. Besides, by querying the managed QoS, we propose a QoS-aware web service composition via a particle swarm optimisation algorithm to perform fast web service composition. Experimental results based on two kinds of dataset show our proposed approach outperforms other schemes in terms of query cost, computation time and optimality.

Keywords: web service; QoS; QoS management; service composition; particle swarm optimisation.

Reference to this paper should be made as follows: Wang, S., Zhu, X. and Yang, F. (2014) 'Efficient QoS management for QoS-aware web service composition', *Int. J. Web and Grid Services*, Vol. 10, No. 1, pp.1–23.

Biographical notes: Shangguang Wang is an Assistant Professor in Beijing University of Posts and Telecommunications. He received his PhD degree in Computer Science and Technology from Beijing University of Posts and Telecommunications in 2011. He served as reviewer for *IEEE Transactions on Parallel and Distributed Systems*, *The Computer Journal*, *IET Software*, *Journal of Network and Computer Applications*, *International Journal of Web and Grid Services*, *International Journal of System Science*, etc. His research interests include service computing and cloud computing.

Xilu Zhu is a Senior Researcher in The Research Institution of China Mobile. He received his PhD degree in Computer Science and Technology from Beijing University of Posts and Telecommunications in 2011. His research interests include service computing and cloud computing.

Fangchun Yang is currently a Professor in the Beijing University of Posts and Telecommunication, China. He received his PhD degree in Communication and Electronic System from the Beijing University of Posts and Telecommunication in 1990. He is a fellow of the IET. He has published six books and more than 80 papers. His research interests include network intelligence, services computing, communications software, soft switching technology and network security.

1 Introduction

Service-Oriented Architecture (SOA) provides a flexible framework for web service composition (Zheng and Lyu, 2013). In SOA-based web service composition process, all web services produced by service providers are registered in service registry centres (such as UDDI). Then a service broker is used to help a customer select multiple web services (with different functional attributes but same Quality of Service (QoS)) to composite a new value-added service according to her QoS requirements. Hence, how to manage efficiently QoS of these web services is a critical problem in web service composition area (Jarma et al., 2013).

The UDDI as a common web service registry centre is often used to manage most data of web services such as service provider data, service implementation data and service metadata. Unfortunately, with the increasing number of web services, there are some disadvantages which are hardly solved in traditional central UDDI (ShaikhAli et al., 2003). For instance, it is difficult to manage and maintain mass web services. It is also vulnerable to denial of service attacks. Besides, its low performance such as high query cost, overload irritates its shortcoming.

Moreover, it is well known that an important usage of the UDDI is to provide QoS attributes (e.g., *response time*, *reliability*, *price*, etc.) of web services for QoS-aware web service composition. The reason is that the number of component services involved in this composite service may be large and the number of web services from which these component services are selected is likely to be even larger (Zeng et al., 2004). Then, the QoS of the resulting composite service executions is a determinant factor to ensure customer satisfaction. However, different users may have different requirements and preferences about this QoS. For instance, a customer may want lessening the response time while satisfying certain constraints with price and reliability, while another user may give more importance to the price than to the response time. A QoS-aware approach to web service composition is therefore needed, which maximises the QoS of composite services by taking into account the constraints and preferences set by the customers (Blanco et al., 2012).

However, the numbers of same or different category web services become larger and larger in composition process (Jiuyun and Reiff-Marganiec, 2008; Ma and Zhang, 2008; Su et al., 2008). The composition performance is much lower when it is based on

the traditional central UDDI. Moreover, it often merely considers the common QoS attributes, e.g., availability, reliability, cost, response time and reputation. Other non-functional attributes related to different business, e.g., performance, compensation, cannot be covered. Hence, it is very necessary to manage efficiently the QoS data of web services for improving the performance of web service composition (Zheng and Lyu, 2010; Zheng et al., 2010).

In this paper, we propose an efficient QoS management approach for QoS-aware web service composition in distributed computing environments. First, a P2P distributed hash table (called Chord) is used to classify web services. Each peer in the Chord maintains a category. The aim of classifying web services is to better manage QoS since it is great disparity among the QoS of web services from various businesses. By classifying web services, we can ensure the comparability among QoS and improve the query efficiency. Second, based on classifying web services, we design a QoS tree to manage the QoS data of all web services in Chord. The QoS tree can support efficient complex query. Finally, to lessen the QoS query in QoS tree, a load balance strategy is adopted. Moreover, relying on the managed QoS, a QoS-aware web service composition approach is proposed via particle swarm optimisation algorithm with customers' QoS constraints. By using a real QoS data of web service and a synthetic QoS data, we evaluate our proposed two approaches. Experimental results show that our approaches outperform other schemes.

We organise the paper as follows: Section 2 introduces the related work about QoS management of web services. Section 3 introduces our proposed QoS management approach, including classifying web services via Chord and Managing QoS via constructing a QoS tree. Based on the QoS management, a QoS-aware web service composition approach is proposed by using particle swarm optimisation in Section 4. Section 5 gives experimental results, including evaluating our proposed tow approaches and comparing them with other approaches. Finally, Section 6 ends the paper.

2 Related work

An efficient management approach for QoS can help customers to composite best suitable web services according to their QoS requirements. Obviously, QoS plays an important role in web service Composition. Hence, many notable QoS management scheme are proposed in an industry and academia.

Hongan et al. (2003) proposed a QoS management scheme. It is deployed in a QoS broker between web service clients and web service providers. The broker collects most QoS information about service providers that may offer qualified web services to a client. It is similar to this scheme; Singhera et al. (2004) integrated the QoS management into the UDDI. Then they developed a system to measure, collect and publish the QoS value of web services. However, they are not distributed UDDI. Hence, to avoid the drawbacks of the central UDDI, some distributed models are proposed. For instance, a distributed QoS registry architecture (Fei et al., 2008), a P2P system supporting distributed QoS register (Ragab et al., 2008) and Chord4S (Qiang et al., 2008).

Although these distributed models in the above literatures are worthy to learn. However, they cannot support an efficient complex query. Moreover, they did not consider the influence of QoS storage, query and load balance to QoS management.

Different previous related work, to improve the efficiency of QoS querying, we take the problem as a multi-attribute complex query. Unfortunately, we find that the traditional P2P with Distributed Hash Table (DHT) cannot effectively support the complex query. Although, the Content-Addressable Network (CAN) (Ratnasamy et al., 2001) can support multi-dimension query, the efficiency is much lower as the increasing dimension. Similar to the CAN, KD-tree (Bentley, 1975; Ganesan et al., 2004; Zhang et al., 2005) divided the space based on the equal data and uses the binary tree to index the space data. However, these distributed structures are only suitable for the specific P2P network. They cannot be used for web services. Hence, we propose a distributed QoS management approach based on the relation among the multi-dimension vector to support efficient QoS management. Moreover, different previous QoS-aware web service composition schemes, we cannot direct take the QoS data into service composition because of their high redundancy and low accuracy. We take the QoS query results into service composition to lessen computation cost and improve the optimality. Moreover, different our previous work (Fei et al., 2008; Zhu and Wang, 2010; Wang et al., 2012; Wang et al., 2013a; Wang et al., 2013b), in this paper, we not only consider the QoS management of web services, but also take the managed QoS data into QoS-aware web service composition process for value-add business applications.

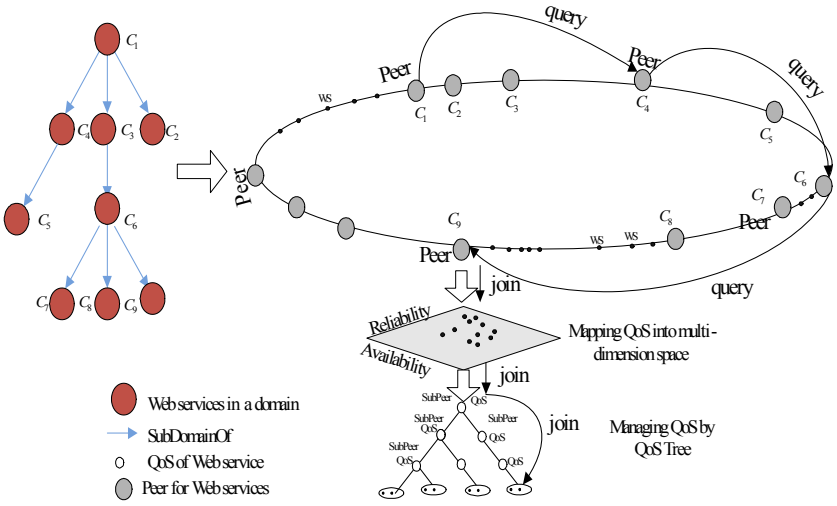
At present, the problem of QoS-aware web service composition has received a lot of considerable attention during the last years in the service computing community. Zeng et al. (2003) adopted the global optimisation approach to find the best service components for the composition by linear programming. Soon afterwards, Cardellini et al. (2007) also used linear programming to optimise user's end-to-end QoS constraints, but it differs in the solution of the optimisation problem. Ardagna and Pernici (2007) extended the linear programming model to include local constraints. These linear programming approaches are effective when the size of the problem is not very large. However, their scalability is very poor due to the exponential time complexity of the applied search algorithms with the increasing size of the problem. Cardinale (2011) presented a selection algorithm that satisfies the user query functional conditions expressed as input and output attributes, QoS requirements represented by weights over criteria and transactional properties expressed as a risk level. Blanco et al. (2012) proposed sampling-based techniques to accurately estimate QoS values that will be used in a hybrid composer, to identify the compositions that best meet a user request. Jiang et al. (2012) proposed an event driven continuous query algorithm for QoS-aware automatic service composition problem to cope with different types of dynamic services. The approach can solve the functional and non-functional parameters of web services dynamic change. However, the computation time of service composition remains out of real time requirements when web services increase.

Although efforts and results above have been made and obtained in web service composition area, existing technologies on web service composition little considered the computation time or time complexity. They are still not mature yet and require significant efforts. In this paper we propose an approach to composite web services with the QoS management results. By discarding redundant web services via our proposed QoS management approach, a particle swarm optimisation is used to find the optimal solution for web service composition with short computation time.

3 Proposed QoS management approach

As shown in Figure 1, our proposed QoS management approach contains the following two phases, i.e. classifying web services (Section 3.1) and Managing QoS (Section 3.2). As the foundation of QoS management, the first phase is very simple and its aim is to classify web services into a distributed network. The key of the approach is the second phase. After classified, we can quantify the QoS of web services and store them in a QoS tree. This phase is a bit more complicated and relying on three parts to manage QoS, that is, constructing QoS tree, designing rules and applying QoS tree.

Figure 1 QoS management approach of web services (see online version for colours)



3.1 Classifying web services via chord

In first phase, we adopt a P2P distributed hash table (called Chord) (Stoica et al., 2001) to classify web services. Chord is a protocol and algorithm for a P2P distributed hash table. It stores key-value pairs by assigning keys to different computers (known as 'nodes'). A node will store the values for all the keys for which it is responsible. Chord specifies how keys are assigned to nodes and how a node can discover the value for a given key by first locating the node responsible for that key.

By using Chord, each peer manages the web services which belong to the same domain. We use the domain ontology to represent their domain. Then, a node of the domain ontology is assigned to a peer in the Chord ring.

The Chord ring with positions numbered 0 to $2^m - 1$ is formed among nodes. Key k is assigned to node *successor(k)*, which is the node whose identifier is equal to or follows the identifier of k. If there are N nodes and K keys, then each node is responsible for roughly K/N keys. When a new node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands. If each node knows only the location of its successor, a linear search over the network could locate a particular key. This is a naive method for searching the network, since any given message could potentially have to be relayed through most of the network.

The search speed of the Chord is very fast. Chord requires each node to keep a "finger table" containing up to *m* entries. The *i-th* entry of node *n* will contain the address of successor *((n + 2^{i-1}) mod 2^m)*. With such a finger table, the number of nodes that must be contacted to find a successor in an N-node network is *O (logN)*.

From the description of the Chord ring, we found that the search process of web service's domain in the Chord ring is a complex query. The reason is that the search must consider the peer which manages the domain proposed by customer and the descendant node in the domain ontology tree. For example, when C_6 is required to search C_7, then C8, C_9 are also need to be involved. Moreover, when a web service joins the Chord ring, the first step is to locate the peer corresponding to the web service's domain ontology and then the QoS will be mapped to a normalised multi-dimension space composed by multiple attributes. Finally, the QoS of the web service will be mounted to a specific node in the QoS tree (Section 3.2). Except a peer in the Chord ring, there is another type peer, called subpeer, to manage the node in the QoS tree. Peer in the Chord will connect one subpeer to forward the QoS request to subpeers. When a peer quit the Chord, a subpeer is selected to replace its position.

3.2 Managing QoS via QoS tree

In this section, we give a description how to manage QoS of web services via QoS tree.

3.2.1 Constructing QoS tree

To construct the QoS tree, we adopt the quantitative method to describe the QoS values.

Different from the qualitative description which often uses OWL-S, the quantitative description's advantage is institutive, and the cost of storage is relatively small. Before adding the quantified QoS into the QoS tree, we map them to a normalised space to set the range of different types of QoS values. In this paper, we set the all QoS attributes range is [0,100]. The more the QoS value is close to 100, the better the web service in that attribute is. The threshold is set to reduce the computation cost. For example, any QoS value is greater or less than the threshold is deemed as 100 or 0. The relation between normalised QoS values can be considered as the Pareto dominance. To construct the QoS tree, we extend this dominance relation to the following three types, i.e., strong dominance, dominance and weak dominance.

Definition 1: *(Strong Dominance). There are two web services, WSA and WSB. The QoS value of WSA can be denoted as* $QoS_1 = (q_1, q_2 \cdots q_m)$. *The QoS value of WSB can be denoted as* $QoS_2 = (q_1^{'}, q_2^{'} \cdots q_m^{'})$. *If only if* $\forall i \in \{1, \cdots, m\} q_i > q_i^{'}$ *and* $QoS_1 \succ QoS_2$, *then* $QoS_1 = (q_1, q_2 \cdots q_m)$ *strong dominate* $QoS_1 = (q_1, q_2 \cdots q_m)$.

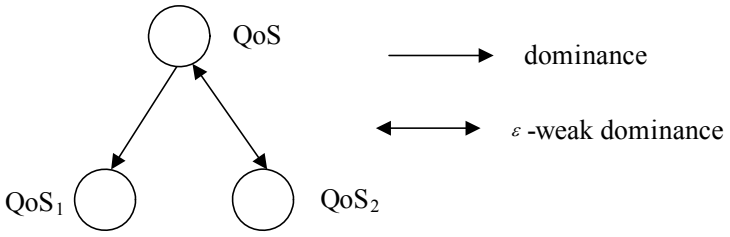
Definition 2: *(Dominance). There are two web services, WSA and WSB. The QoS value of WSA can be denoted as* $QoS_1 = (q_1, q_2 \cdots q_m)$. *The QoS value of WSB can be denoted as* $QoS_2 = (q_1^{'}, q_2^{'} \cdots q_m^{'})$. *If only if* $\exists i \in \{1, \cdots, m\} q_i = q_i^{'}$ *and* $QoS_1 \succeq QoS_2$, *then* $QoS_1 = (q_1, q_2 \cdots q_m)$ *dominate* $QoS_2 = (q_1^{'}, q_2^{'} \cdots q_m^{'})$.

Obviously, from the above two definitions, the strong dominance is an exception of the dominance. Hence, in this paper, we use the mark $\succeq_{all}$ to include the two types.

Definition 3: *(ε-weak Dominance). There are two web services, WSA and WSB. The QoS value of WSA can be denoted as* $QoS_1 = (q_1, q_2 \cdots q_m)$. *The QoS value of WSB can be denoted as* $QoS_2 = (q_1^{'}, q_2^{'} \cdots q_m^{'})$. *If only if* $\exists i \in \{1, \cdots, m\} q_i \geq q_i^{'}$ *and* $QoS_1 \succ QoS_2$, *where the number of* $q_i \geq q$ *is* ε *and* $\varepsilon < m$, *then* $QoS_1 = (q_1, q_2 \cdots q_m)$ *ε-weakly dominate* $QoS_2 = (q_1^{'}, q_2^{'} \cdots q_m^{'})$.

So according to the three definitions, we find that those satisfied QoS values must dominate or strongly dominate the QoS_{user} proposed by user. If the node in QoS tree can maintain these relations, it might reduce the searching cost and improve the search efficiency. Thus, as shown in Figure 2, we give a description of the data structure of a node in QoS tree.

Figure 2 The data structure of the node in QoS tree



From Figure 2, the $node_{QoS_1}$ managed by *LeftChild* is strongly dominate or dominate by the $node_{QoS}$. The $node_{QoS_2}$ managed by *RightChild* ε-weakly dominate by the $node_{QoS}$. The *QoS* denotes the non-functional vector ($q_1, q_2 \cdots q_n$) and $Val = \frac{1}{n}\sum_{i=1}^{n} q_i$ is the average value of the *QoS* vector. Then, according the node's data structure, we find that the QoS tree contains the following two properties.

Theorem 1: *When the* $node_{QoS_1}$ *is the left child of the* $node_{QoS}$ *and the* $node_{QoS_2}$ *is the right child of the* $node_{QoS_1}$, *then* $QoS \succeq_{all} QoS_1 \succeq_{all} QoS_2$. *When the Val of the* $node_{QoS}$ *is less than the customer's requirement* QoS_{user}, *the* $node_{QoS}$ *must not satisfy the customer's requirement.*

Proof: Known by the reduction ad absurdum, if *QoS* satisfies the condition: $\neg\exists j \in \{1, \cdots, m\} q_j < q_j^{'}$, $q_j \in QoS$, $q_j^{'} \in QoS_{user}$, then the *Val* of the $node_{QoS}$ must be more than customer's requirement QoS_{user}. However, it is contrary to the condition. Thus, we can conclude that the *Val* of the node $node_{QoS}$ is less than the customer's requirement QoS_{user}. Therefore, the $node_{QoS}$ must not satisfy the customer's requirement.

Theorem 2: *If the QoS is ε-weak dominated by customer's requirement* QoS_{user} *and ε is the maximum in the leftmost nodes, the left descendants of the* $node_{QoS}$ *must not strongly dominate or dominate* QoS_{user}.

Proof: Known by the reduction and absurdum, because the $QoS^{'}$ of the $node_{QoS}$'s left descendants $node_{QoS_1}$ dominate QoS_{user}, the value in the $QoS^{'}$ vector must be equal or

great than QoS_{user}. Therefore, the QoS' must ε-weakly dominate QoS. However, it's inconsistent with the condition that the QoS strongly dominate the QoS'. Thus, we conclude that the QoS of the $node_{QoS}$'s left descendant must not strongly dominate or dominate QoS_{user}.

Hence, according two Theorems, we find that the comparison cost for researching the QoS tree can be lessened.

3.2.2 Designing rules for QoS tree

According to the node's data structure, as shown in Algorithm 1, we propose a joining QoS tree algorithm to to help a new QoS vector join the QoS tree.

Algorithm 1. Joining QoS tree

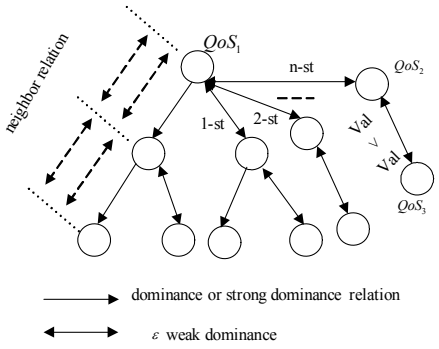
```
Input: node oldNode, node newNode;
Switch (compare(oldNode, newNode)
case 0: //ε-weakly dominated
   addInRightSubTree(oldNode, newNode);
case 1: // dominate or strong domination
   addInLeftSubTree(oldNode, newNode);
case 2: //dominated
   newNode replace the position of oldNode;
```

In Algorithm 1, the first step is to compare a new QoS_{new} vector with the *root*' QoS. If the QoS of the $node_{QoS}$ dominate QoS_{new}, the *addInLeftSubTree* function will be invoked. Then QoS_{new} will be continuous to compare with the leftmost node until it finds a node which has dominated or ε-weakly dominated relation. If the QoS_{new} dominate the QoS, it will become the $node_{QoS}$'s parent; otherwise, the *addInRightSubTree* function is invoked. In that function, the QoS_{new} continues to compare with the $node_{QoS}$'s rightmost child by using the Val until the node's Val is less than the $node_{QoS'}$. Thus, the QoS_{new} will be mounted on the $node_{QoS_{new}}$ by repeatedly invoking the *NodeJoin* function.

From Algorithm 1, we find that the left child node of a QoS tree can be considered as the index for a new QoS join, it's critical for the search efficiency. We call these nodes as index node, and it can be represented by $node_{index}$. The leftmost node of a QoS tree is the main index of the QoS tree, thus, it is denoted by $Mnode_{index}$.

Moreover, if we compare with $node_{index}$ step by step, the search efficiency will be very low. The time complexity of iterative comparison is $\Theta(n)$ and the time complexity cost of comparing rightmost node in the *addInRightSubTree* procedure is the same as that of comparing with $node_{index}$. As large number of web services joins to a peer, the construction and the search cost will grow. Thus, it's necessary to add some new element to the node structure. Since the leftmost node and the rightmost node have the transitive relation, we add the binary search tree to manage this transitive relation. Thus, each node not only stores its left child, right child and parent, but also stores the node which is immediate in the binary search tree. The time complexity will reduce to $\Theta(\log_2 n)$. We take Figure 3 as an example to design some rules for constructing QoS tree.

Figure 3 QoS tree



Rule 1: In the QoS tree, the $Mnode_{index}$ is critical for search efficiency. To simplify the index construction, we consider the diagonal of the hypercube as the index. That is, a strong dominance relation vector sequence is introduced $(0,\cdots,0) \prec (1,\ldots,1) \prec \ldots \prec (100,\ldots,100)$.

Rule 2: When a new QoS_{new} joins a $node_{QoS}$'s right-side subtree, $node_{QoS}$ must be maximum ε-weak dominated. For example, Comparing to the $Mnode_{index}$ (50,50,50) and (60,60,60), the new QoS vector (52,45,65) will be mounted on (50,50,50) 's right child due to the maximum ε-weak dominance.

Rule 3: When a new QoS_{new} joins a $Mnode_{index}$'s right-side subtree, and it will be classified according to similar position in the vector. For example, when the $node_{QoS_1}$ is QoS_1 (50, 60, 70) and the $node_{QoS_2}$ is QoS_2(60, 50, 70). They will join the right tree of $Mnode_{index}$, which has *QoS* (50, 50, 50). QoS_1, QoS_2 has the same value with *QoS* vector at 1-th place, 2-th place, respectively. Then $node_{QoS_1}$ will be classified in first class and $node_{QoS_2}$ will be classified in second class.

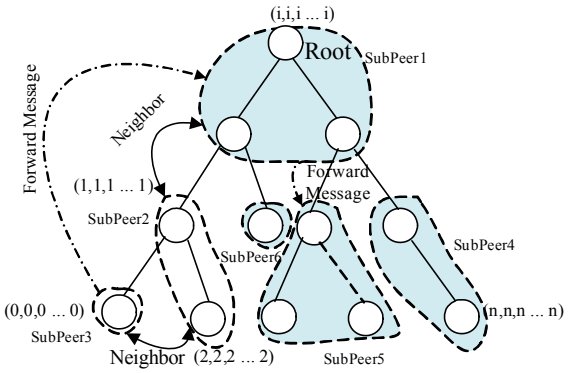
3.2.3 Applying QoS tree to manage QoS

The above section describes the construction of QoS tree. To avoid the shortcomings of the central model, it's necessary to partition the QoS tree, and distributed the parts into subpeers in practical web service application. Hence, in this section, we propose Range Query algorithm, Subpeer Join and Quit algorithm and the load balance strategy.

Range Query

The main characteristic of QoS tree is to support the range query, any strongly dominate or dominate the user's requirement QoS_{user} can be accepted. To support this kind of complex query, the content of the QoS tree route table includes the neighbours of the main index vector sequence and the immediate relation node in binary search tree. In addition, the subpeer which maintains the root in binary search tree is also included by the other subpeers. Therefore, the cost of the exact query which locates the start subpeer of range query is less than $\lceil \log n \rceil + 1$. A QoS tree partition sample graph is depicted as shown in Figure 4.

Figure 4 Range query on QoS tree (see online version for colours)



In this sample graph, the route table of subpeer2 which maintains node (1,1,1,…1) and node (2,2,2,…2) contains the subpeer3 and subpeer6. It has the immediate relation with (1,1,1,…1) in binary search tree. The neighbours relation in main index vector sequence (0,0,0…0) and (3,3,3…3) is irrespective, and the subpeer1 is also included. The maximum depth of the route table is 6, if each subpeer only maintains one node in the binary search tree. The communication cost of subpeer which maintains the node (0, 0, 0…0) is the least since the query cost is zero. Thus, it is selected to communicate with the peer in Chord ring. According to the route table, the Range Query algorithm can be described in Algorithm 2.

Algorithm 2: Range query

Step 1: use the exact query to find the maximum ε-weak dominant QoS_{user}'subpeer, each query's startpoint is subpeer maintains (0,0,0…0), it will compare the QoS_{user} with the root to judge whether the destination is at the left-side subtree or right-side subtree.

Step 2: When the message is forward to the destination subpeer, it will broadcast the message to the neighbours in terms of the route table. Each subpeer which receives range query message does the local search according to the search algorithm of QoS tree.

Step 3: return the query result, and the range query stops.

Subpeer join and leave

In practical application, some web services are hardly inquired by customers due to the some poor quality attributes in QoS vector. Thus, it leads the subpeers which manage them to pay less query load. According to the distribution of QoS values in the QoS tree, the more reach to the root node, the more query load the subpeers will pay. Therefore, we classify the subpeers into two types: the query subpeer and the storage subpeer. The storage subpeer mainly manages the QoS which is hardly inquired by user and the query subpeers share the query load. The partition of nodes among storage peers in terms of the quantity, thus, the storage load of a subpeer is number of its QoS node. The partition of the query subpeers rely on the query load. The query load of a node can be considered as

the number of query times in a unit time, which is denoted by f_Q, and the query load of a $Mnode_{index}$ is $\sum f_Q$. Hence, the Subpeer Join algorithm can be described in Algorithm 2.

Algorithm 3: Peer Join

```
Input: Peer newPeer;
1.  IF(QueryLoad( )==0 ){
2.     p= FindMostCrowdedPeer( );
3.  }
4.  ELSE{
5.       IF ( !IsQueryLoadBalance( )){
6.          p=FindMaxLoadInQuerySubpeer( );
7.       }ELSE{
8.          p= FindMaxLoadInStorageSubpeer ( );
9.       }
10.  }
```

According to Algorithm 3, if the QoS tree hasn't been inquired yet, it will select the subpeer which maintains the most number of nodes to split. Otherwise, a new subpeer relies on the query subpeers's load imbalance to select the type of subpeer it will join. In both join procedures, the new coming node will accept some node from the other subpeer. The QoS node partition principle is to transfer a half type of node classification defined in Rule 2. When receiving the QoSs, the new coming subpeer will construct a new QoS tree for management. Contrary to the subpeer's join, the leave procedure the leaving procedure is invoked when a subpeer quit. Then the query subpeer will find whether there is a storage subpeer can replace its position. If not, it will shed the load to its neighbours which next to each other in main index vector sequence. As shown in Algorithm 4, in our proposed Peer Leave algorithm, if the storage subpeer leaves its position, it merely transfers its data to the neighbours and quit the system.

Algorithm 4: PeerLeave

```
Input: Peer lPeer;
1.  IF( !IsQueryPeer(lPeer)){
2.       AssginNodeToNeighbour( );
3.  }ELSE{
4.        IF(numOf(RelaxPeer))>1
5.  &&Nodein(least)< Nodein(lPeer)){
6.       TakeOverBy (least, lPeer);
7.        }
8.        ELSE{
9.          TakeOverByNeighbour(lPeer);
10.       }
11.      AcceptNewNode(p,newPeer);
12.  }
```

Load balance

According to the literature (Ganesan et al., 2004), the load balanced P2P system has an imbalance factor δ. That is, the ratio of the maximum load peer and the minimum load peer is less than δ. The peerleave procedure mentioned above shows us that the storage

subpeers' function is not only to manage some node, but also to help the overflowing subpeer. Thus, these procedures is designed to ensure the query load balance among the subpeers, $\frac{\mu_q}{l_q} \le \delta_q$. We use amortised analysis method to discuss the load balance. The potential function is $\Phi = \Phi_q + \Phi_s$, where

$$\Phi_q = \sum_{p \in P_q} \varphi_q(p) \tag{1}$$

$$\Phi_s = \sum_{p \in P_s} \varphi_s(p) \tag{2}$$

$$\varphi_q(p) = \begin{cases} \frac{c_q}{l_q}(l_q^0 - |\, p.ld\,|)^2 & l_q \le |\, p.ld\,| \le l_q^0 \\ 0 & l_q^0 \le |\, p.ld\,| \le u_q^0 \\ \frac{c_q}{l_q}(|\, p.ld\,| - u_q^0)^2 & u_q^0 \le |\, p.ld\,| \le u_q \end{cases} \tag{3}$$

$$\varphi_s(p) = \frac{c_s}{l_s}(|\, p.nu\,|)^2 \tag{4}$$

where

$$l_q^0 = (\frac{\delta_q}{4}) l_q \text{ and } u_q^0 = \frac{3\delta_q}{4} l_q .$$

When a QoS vector adds in QoS tree, it might be inserted into storage subpeer or query subpeer. Thus, the maximum increase in the potential is $\Delta\Phi = \max(\Delta\Phi_q, \Delta\Phi_s)$ where

$$\begin{aligned} \Delta\Phi_q &= \frac{c_q}{l_q}(|\, p.ld\,| + f_Q - u_q^0)^2 - \frac{c_q}{l_q}(|\, p.ld\,| - u_q^0)^2 \\ &= \frac{c_q}{l_q}(2\,|\, p.ld\,| f_Q + f_Q^2 - 2\mu_q^0 f_Q) \le \frac{3c_q l_q \delta_q^2}{2} \end{aligned} \tag{5}$$

with

$$\Delta\Phi_s = \frac{c_s}{l_s}((|\, p.nu\,| + 1)^2 - (|\, p.nu\,|))^2 = \frac{c_s}{l_s}\left(2\,|\, p.nu\,| + 1\right) \le 2c_s\delta_s + c_s \tag{6}$$

The increase potential $\Delta\Phi$ is bounded by a constant by the following:

$$\Delta\Phi = \max\left(\frac{3c_q l_q \delta_q^2}{2},\ 2c_s\delta_s + c_s\right) \tag{7}$$

The increase potential of deleting a QoS from DQoS tree is also bounded by the same constant $\Delta\Phi$. Then, we find that the cost of load balance algorithm is less than the

potential decrease by setting a proper c_q and c_s. When a query subpeer overflows, it will transfer a half of query load $\frac{\delta_q l_q}{2}$ to the new coming subpeer q. The potential decrease by the following:

$$\Delta\Phi_q = \frac{c_q}{l_q}(|\ p.ld\ | - u^0_{query})^2 \le c_q l_q (\frac{\delta_q}{4})^2 \tag{8}$$

The potential decrease of the storage subpeer is as follows:

$$\Delta\Phi_s = \frac{c_s}{l_s}((|\ p_1.nu\ | + l)^2 - (|\ p_1.nu\ |)^2) \le c_s(2\delta_s + l_s) \tag{9}$$

Since the maximum cost of this rebalance procedure is $\frac{\delta_q l_q}{2f_Q} + \delta_s l_s$. To ensure the cost is less than the potential's decrease, it must satisfy the inequality by the following:

$$c_q l_q (\frac{\delta_q}{4})^2 - c_s(2\delta_s + l_s) > \frac{\delta_q l_q}{2f_Q} + \delta_s l_s \tag{10}$$

In the load balance procedure of underflow, the inequality is as follows:

$$c_q l_q (\frac{\delta_q}{4})^2 \ge (\frac{\delta_q}{4}) l_q \tag{11}$$

Therefore, by setting the constants c_q and c_s, the cost is less than the potential decrease. We can prove that the amortised cost of inserts and deletes is a constant.

4 QoS-aware web service composition

4.1 What is service composition?

In this section we introduce several definitions for understanding service composition.

Definition 4: *A composite service can be defined as an abstract representation of a composition request* $\mathbb{S}=\{S_1,\cdots,S_n\}$, *where* $\mathbb{S}$ *refers to n (n<1) required service classes without referring to any concrete service.*

In Definition 4, a service class S_j ($S_j \in \mathbb{S}$, and $S_j = \{s_{j1},\cdots s_{jl}\}$) contains $l(l>1)$ service candidates which have same functional attributes and different QoS attributes. In this paper, a service candidate denotes a concrete service, e.g., s_j ($s_j \in S_j$) is a transcoding service from Huawei.

Definition 5: *QoS vector of a composite service* $\mathbb{S}$ *can be defined as* $Q\mathbb{S} = \{q_1(\mathbb{S}),\cdots, q_r(\mathbb{S})\}$, *and* $q_i(\mathbb{S})$ *represents the estimated value of the i-th QoS attribute of* $\mathbb{S}$.

Definition 6: *QoS vector of a service candidate s can be define as* $Qs = \{q_1(s),\cdots,q_r(s)\}$, *and* $q_i(s)$ *represents the estimated value of the i-th QoS attribute of s, which determines the quality of each service candidate.*

In this paper, service candidates have a number of different QoS attributes, and the difference often result in large variation in their QoS values or scopes. Obviously, prefect computing or evaluating QoS is very different for each service candidate. Hence, Yu et al. (2007) proposed a QoS utility function to overcome the problem. The function maps the quality vector Q_S into a single real value, and enables sorting and ranking of service candidates and simplifying choosing to satisfy QoS constraints of the service components. The QoS utility function adopted by this paper is similar to Yu et al. (2007). What is the QoS utility function? For example, in the sequential composition model, the throughput of QoS attributes can be calculated by $q(\mathbb{S}) = \min_{j=1}^{n} q(s_j)$ and the reputation can be calculated by $q(\mathbb{S}) = \sum_{j=1}^{n} q(s_j)$. Because other models (e.g., parallel, conditional and loops) can be reduced or transformed to the sequential model with several techniques for handling multiple execution paths and unfolding loops from (Jang et al., 2006), and here omitted. Hence, in this paper, we also focus on the sequential composition model, and then in the sequential composition model, the overall utility of a composite service $\mathbb{S}$ is computed by the following:

$$U(s) = \sum_{k=1}^{r} \frac{Q_{j,k}^{\max} - q_k(s)}{Q_{j,k}^{\max} - Q_{j,k}^{\min}} . w_k \tag{12}$$

$$U(\mathbb{S}) = \sum_{k=1}^{r} \frac{Q_k^{\max} - q_k(\mathbb{S})}{Q_k^{\max} - Q_k^{\min}} . w_k \tag{13}$$

with

$$Q_{j,k}^{\max} = \max_{\forall s_{ji} \in S_j} q_k(s_{ji}), Q_{j,k}^{\min} = \min_{\forall s_{ji} \in S_j} q_k(s_{ji}), Q_k^{\max} = \sum_{j=1}^{n} Q_{j,k}^{\max} \text{ and } Q_k^{\min} = \sum_{j=1}^{n} Q_{j,k}^{\min}$$

where $w_k \in R^+ (\sum_{k=1}^{r} w_k = 1)$ represents users' preferences; $Q_{j,k}^{\min}$ is the minimum value of the *k*-th attribute in all service candidates of the service class S_j; $Q_{j,k}^{\max}$ is the maximum value of the *k*-th attribute in all service candidates of the service class S_j; $Q_k^{\min}$ is the minimum value of the *k*-th attribute of $\mathbb{S}$; $Q_k^{\max}$ is the maximum value of the *k*-th attribute of $\mathbb{S}$.

In service computing, service composition with global QoS constraints is a well-know optimisation process. The optimal composition for a given service composition $\mathbb{S}$ must satisfy the following two conditions (Wang et al., 2011):

1 For a given vector of global QoS constraints $\mathbb{CS} = \{C_1, \cdots, C_m\}$ $(0 \le m \le r)$, $q(\mathbb{S}) \le C$ $(\forall C_k \in \mathbb{CS})$ where $q(\mathbb{S})$ is the aggregated QoS value of the composition service;

2 The maximum overall utility value $U(\mathbb{S})$ in the composition service.

In order to solve the optimisation problem, we find that finding the optimal composition requires enumerating all possible combinations of service candidates, which can be very expensive in terms of computation time. Based on the previous work, we find that particle swarm optimisation is an alternative scheme to solve the optimisation problem based on the managed QoS results.

4.2 Particle swarm optimisation

As it is well known that Particle swarm optimisation (PSO) is based on information sharing and cooperation between particles in a swarm (Kennedy and Eberhart, 1995). A particle is attracted by its personal best position (local optimisation) p_{best} and the best position (global optimisation) of all particles in its neighbourhood p_{gbest}. By randomly changing the magnitude of these attractions, particles can search for better position in the regions around p_{best} and p_{gbest}. It assumes that the search space is d-dimensional and the particle population is *Np*. Every particle in the swarm has a position $x_{id}\,(i \in Np)$, a velocity v_{id} and a memory p_{best} for its best found position, which are all updated in every iteraction step of PSO. The best solution p_{gbest} depends on the iteration number ($it\max$). At the beginning, the *Np* particles are initialised with a random position. The fitness of all initial position is evaluated by the fitness function, leading to an initial p_{gbest}.

During every iteration step of PSO, the new position $x_{id}^{t+1}(t = 1, 2, \cdots, it\max)$ and new velocity v_{id}^{t+1} of each particle are calculated by the following:

$$v_{id}^{t+1} = w * v_{id}^{t} + c_1 * r_1 * (p_{best}(t) - x_{id}^{t}) + c_2 * r_2 * (p_{gbest}(t) - x_{id}^{t}) \tag{15}$$

$$x_{id}^{t+1} = x_{id}^{t} + v_{id}^{t+1} \tag{16}$$

where the parameter w (called the inertia weigh) is a measure for the sensitivity to influences of p_{gbest} and p_{best}, and controls the exploration behaviour of the swarm; The parameters c_1 and c_2 are the cognitive ratio and the social ratio and used to control the influence of p_{gbest} and p_{best} on a particle's new velocity; The random variables r_1 and r_2 are uniformly distributed $U(0,1)$ (Demarcke et al., 2009).

4.3 Service composition based on QoS management

In this paper, binary decision variables are used in the problem to represent the service candidate. A service candidate s_{ji} is selected in the optimal composition if its corresponding variable x_{ji} is set to 1 and discarded otherwise. Then a fitness function (f) is designed to maxim**ise t**he overall utility value. By re-writing (12) to include the decision variables, the fitness function can be expressed by the following:

$$f = \sum_{k=1}^{r} \frac{Q_k^{\max} - \sum_{j=1}^{n}\sum_{i=1}^{l} x_{ji} \cdot q_k(s_{ji})}{Q_k^{\max} - Q_k^{\min}} . w_k \tag{17}$$

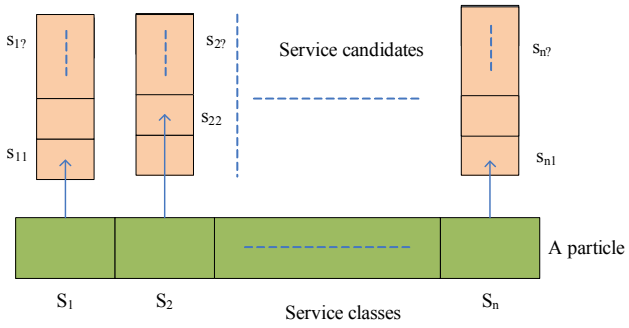
In order to ensure that global QoS constraints are satisfied in the composition, we add the following set of constraints to the (17):

$$\begin{cases} \sum_{j=1}^{n}\sum_{i=1}^{l} q_k(s_{ji}) \cdot x_{ji} \geq C_k, 1 \leq k \leq m \\ \sum_{j=1}^{l} x_{ji} = 1, 1 \leq j \leq n \end{cases} \tag{18}$$

Then by solving (17) and (18) using PSO, a list of service candidates are obtained and returned to service broker providing a composition service for service users.

In order to find best services by PSO, a suitable coding scheme should be designed. We use integer array coding scheme on the basis of the characters of service candidate. The number of items in the array denotes the number of service classes. Each element of the array denotes the index of service candidates. The maximum of the element is the number of service candidates, and the minimum is 1. Therefore, all particles are coded as n-dimensional arrays as shown in Figure 5. The velocities of the particles are also encoded as n-dimensional arrays, and each element value of the velocity is an integer that is over $[-V_{\max}, +V_{\max}]$.

Figure 5 The particle coding scheme (see online version for colours)



5 Experiment

In order to prove the efficiency of the DQoS-tree, we implement a simulation platform using JDK 1.6. The algorithms in this platform include the subpeer' join and leave, range query and the load balance. We compare QoS tree with KD-tree (Jang et al., 2006; Jiang et al., 2012; Jarma et al., 2013) in the query performance by using two kinds of experiment data. One is the real dataset, QWS which comes from the literature (Ma and Zhang, 2008). QWS includes about 2500 web services with 11 types of non-functional attributes. We choose some to do out experiments. Moreover, we use the synthetic data, it contains 10000 QoS values, which are all range from [0, 100]. The experiment environment is Pentium CPU 2.0G, 2G RAM, Windows XP.

Moreover, based on the QoS management, we also compare proposed service composition approach with other approaches (Ardagna and Pernici, 2007; Wang et al., 2010). The same parameters are set, including the number of service classes from 5 to 50 and the number of service candidates per class from 100 to 1000. The number of QoS attributes and global QoS constraints in all these test cases are fixed to 3 and 2, respectively. Other parameters are set such as $w = 0.7$, $c_1 = 2.0$, $c_2 = 2.0$, itmax = 30, $Np = 12$. In the experiments, the capital letters 'WSC' represent our service composition approach. The capital letters 'MIP' represent the approach in (Ardagna and Pernici, 2007). The capital letters 'QoST' represent the approach by Wang et al. (2010). All results were collected in average after each approach running for 20 times.

5.1 Query cost

The query cost experiment includes two types of query cost, the exact query cost and the range query cost. The range query cost is to compute the cost of the compare times. The exact query cost is to compute the hops which the message forward from the start subpeer to the destination subpeer. Since the exact query in QoS tree is start from the subpeer which manage the QoS vector (0, 0…0). Thus, the exact query in D-tree is to count the hops from random subpeer to the subpeer which manage the QoS vector (0, 0…0).

From Figure 6, the increasing dimension poses little affection to our approach with respect to the two datasets. This can be explained by the reason that the nodes in QoS tree of our approach maintain the strong-dominance or dominance relation. This can reduce some compare cost to some extents. Moreover, the classification management of rule 2 contributes to reduce compare cost. On the contrary, the peer in the KD-tree doesn't manage its data. Therefore, the compare cost increase sharply as the dimension increase.

Figure 6 Range query cost (see online version for colours)

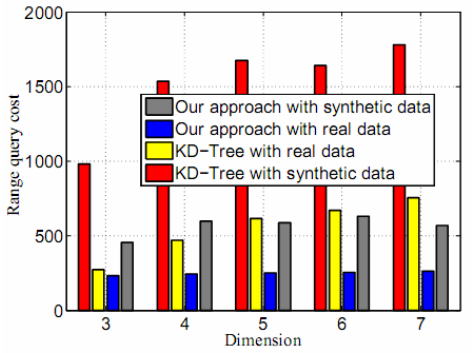
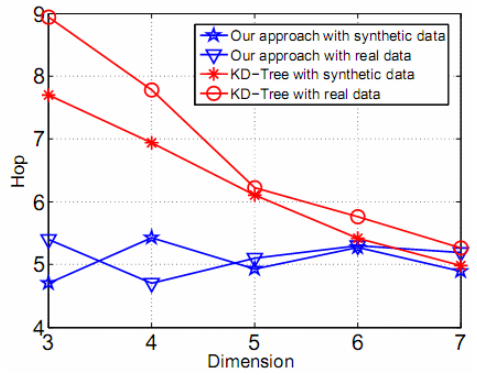


Figure 7 shows the comparison results in term of the exact query cost where 200 subpeers are invoked. From the experimental results, we find that the exact query cost of our approach remains in a steady state. The reason is that the route table consists of the neighbours which are the parent and child node in the binary search tree. The exact query cost nearly is close to the average search length, and it will not increase as the dimension or the number of subpeer increase. The exact query cost of KD-tree is lower than that of QoS tree only when the dimension increases into 7-dimensions. The reason is that the subpeer's route table will grows as the dimension increases, especially, the skewed data. Thus, it contributes to reduce the exact query cost. However, with the increasing numbers of subpeers, the exact query cost will be not steady as the QoS tree does.

5.2 Query and storage load

There are two types of subpeer in our approach, query subpeer and storage subpeer. To prove the query load satisfy the load imbalance factor δ, we use the real data, and assign them to 100 subpeers. The query probability refers to the route algorithm which the higher quality QoS has higher probability to be inquired. We continue to inquire the QoS tree 1000 times. The f_{query} is 0.3, 0.4, respectively.

Figure 7 Cost of exact query (see online version for colours)



From Figure 8, the query load is relatively smooth due to the load balance operation. This means that the compare times of our approach gradually increases with the increasing numer of QoS vector of web service. Figure 9 shows that the storage load, i.e. the hops gradually decreases with the increasing number of QoS vector of web service. Hence, from Figures 8 and 9, the query and storage load are controlled within a reasonable scope.

Figure 8 Query load (see online version for colours)

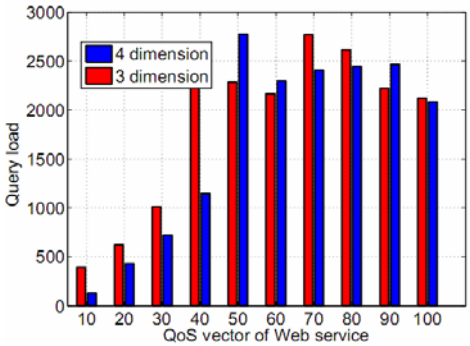
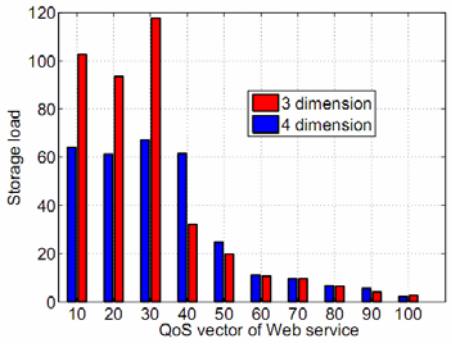


Figure 9 Storage load (see online version for colours)



5.3 Load balance

The load balance experiments mainly describe the changes of the load imbalance factor δ_{query} and $\delta_{storage}$. For this experiment, there are total 100 subpeers managing 2500 real QoS. This experiment is divided into 3 phases, the first phase is add 1000 QoS to the QoS tree, the second phase is to add or delete QoS alternatively, and the final phase is delete 1000 QoS from the QoS tree.

From Figure 10, the load-imbalance factor retains steady level which has minor fluctuation, and the load-imbalance of the storage subpeers has a significant fluctuation without the specific load balance operation. Figure 11 shows the communication cost of the subpeers join and leave when the overflow or underflow occurs. We find that when the overflow or underflow occurs, the number of adjusted nodes is fewer than 50 nodes in most instances.

5.4 Computation time of service composition

In this section, we compare our service composition approach with MIP on computation time.

As shown in Figure 12, we compare the computation time of WSC and MIP with respect to the number of service candidates. In the experiment, the number of service classes is 5–15 (random) for the QWS dataset and the synthetic data in all test cases.

Figure 10 Load balance of query and storage (see online version for colours)

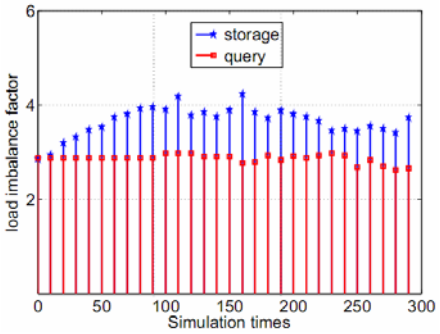


Figure 11 Load balance cost (see online version for colours)

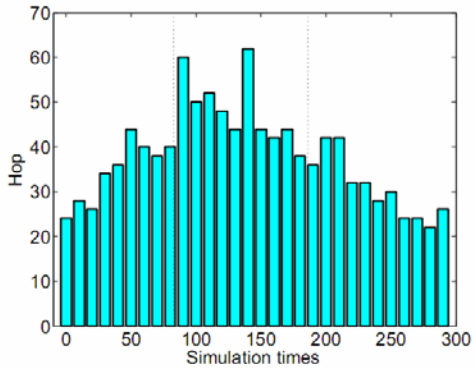
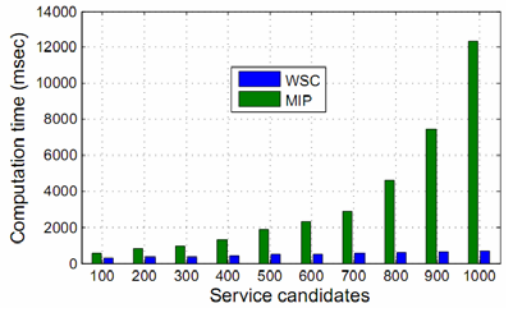
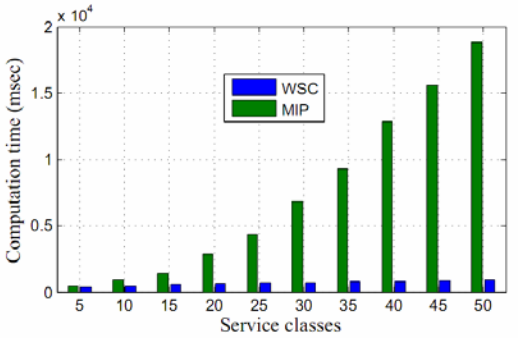


Figure 12 The computation time with respect to the number of service candidates (see online version for colours)



As shown in Figure 13, we compare the computation time of WSC, MIP and QoST with respect to the number of service classes. In this experiment, the number of service candidates is 100–500 (random) for the QWS dataset and 100–500 (random) for the synthetic data.

Figure 13 The computation time with respect to the number of service classes (see online version for colours)



From the two figures above, regardless of the QWS real dataset, the synthetic data, the computation time of our approach is obviously shorter than that of MIP with respect to the number of service classes and service candidates. The reason is that many redundant QoS data are removed in our QoS tree. The search space of our service composition approach is less than MIP.

5.5 Optimality of service composition

In this experiment, we evaluate the quality of the results obtained. 'optim1' represent the overall utility value of the composition service by WSC. 'optim2' represent the overall utility value of the optimal composition services by MIP. Then the optimality of our approach can be calculated by optim1/optim2. The optimal degree of MIP is 100% (because of optim2/optim2).

Figures 14 and 15 shows the optimal degree of our approach and MIP with respect to the number of service candidates and service classes, respectively.

From the simulation results, the optimality of our approach is 94.8% on average and almost close to the optimal solution of MIP (the optimal degree is always 100%). Because this gap (5.2%) is very little, it is acceptable for the service composition based on global QoS constraints. What's more, in user-centric web service environment, the optimality of composition result of our approach is almost the same as that of MIP, but the computation time of our approach is much fewer than MIP. Hence, the QoE (Tasaka and Yoshimi, 2008) of our approach is much better than MIP.

6 Conclusions

In order to efficiently and correctly search the qualified QoS of web service and avoid the shortcomings of the central web service register for QoS management, we propose a two phase QoS management approach. The key of the approach is to construct a QoS tree for manage the QoS of web service. Moreover, based on QoS management results, we also propose a QoS-aware web service composition approach via a particle swarm optimisation. Extensive experiments show that our proposed two approaches can outperform better performance than other approaches in terms of query cost, computation time and optimality.

Acknowledgements

The work presented in this study is supported by the National Natural Science Foundation of China under Grant No.61202435; National Natural Science Foundation of China under Grant No. 61272521; Natural Science Foundation of Beijing under Grant No.4132048; Specialized Research Fund for the Doctoral Program of Higher Education under Grant No. 20110005130001; Program for New Century Excellent Talents in University of China under Grant No.NCET-10-0263; Innovative Research Groups of the National Natural Science Foundation under Grant No.61121061.

References

Ardagna, D. and Pernici, B. (2007) 'Adaptive service composition in flexible processes', *IEEE Transactions on Software Engineerin*g, Vol. 33, No. 6, pp.369–384.

Bentley, J.L. (1975) 'Multidimensional binary search trees used for associative searching',*Communications of the ACM*, Vol. 18, No. 9, pp.509–517.

Blanco, E., Cardinale, Y. and Vidal, M.E. (2012) 'Experiences of sampling-based approaches for estimating QoS parameters in the web service composition problem', *International Journal of Web and Grid Services*, Vol. 8, No. 1, pp.1–30.

Cardellini, V., Casalicchio, E., Grassi, V. and Lo Presti, F. (2007) 'Flow-based service selection for web service composition supporting multiple QoS classes'. *Proceedings of IEEE International Conference on Web Services, ICWS 2007*, 9–13 July, Salt Lake City, UT, USA, pp.743–750.

Cardinale, Y. (2011) 'CPN-TWS: a coloured petri-net approach for transactional-QoS driven web service composition', *International Journal of Web and Grid Services*, Vol. 7, No. 1, pp.91–115.

Demarcke, P., Rogier, H., Goossens, R. and De Jaeger, P. (2009) 'Beamforming in the presence of mutual coupling based on constrained particle swarm optimization', *IEEE Transactions on Antennas and Propagation*, Vol. 57, No. 6, pp.1655–1666.

Fei, L., Fangchun, Y., Kai, S. and Sen, S. (2008) 'A policy-driven distributed framework for monitoring quality of web services', *Proceedings of IEEE International Conference on Web Services, ICWS 2008*, 23–26 September, Beijing, China, pp.708–715.

Ganesan, P., Yang, B. and Garcia-Molina, H. (2004) 'One torus to rule them all: multi-dimensional queries in P2P systems', *Proceedings of the 7th International Workshop on the Web and Databases, WebDB 2004*, 17–18 June, Paris, France, pp.19–24.

Hongan, C., Tao, Y. and Kwei-Jay, L. (2003) 'QCWS: an implementation of QoS-capable multimedia web services', *Proceedings of the 5th International Symposium on Multimedia Software Engineering, MSE 2003*, 10–12 December, Taichun, Taiwan, pp.38–45.

Jang, J.H., Shin, D.H. and Lee, K.H. (2006) 'Fast quality driven selection of composite web services', *Proceedings of the 4th European Conference on Web Services, ECOWS 2006*, 4–6 Decemeber, Zurich, Switzerland, pp.87–96.

Jarma, Y., Boloor, K., de Amorim, M.D., Viniotis, Y. and Callaway, R.D. (2013) 'Dynamic service contract enforcement in service-oriented networks', *IEEE Transactions on Services Computing*, Vol. 6, No. 1, pp.130–142.

Jiang, W., Songlin, H., Lee, D., Gong, S. and Zhiyong, L. (2012) 'Continuous query for QoS-aware automatic service composition', *Proceedings of the 19th International Conference on Web Services, ICWS 2012*, 24–29 June, Honolulu, Hawaii, USA, pp.50–57.

Jiuyun, X. and Reiff-Marganiec, S. (2008) 'Towards heuristic web services composition using immune algorithm', *Proceedings of IEEE International Conference on Web Services, ICWS 2008*, 23–26 September, Beijing, China, pp.238–245.

Kennedy, J. and Eberhart, R. (1995) 'Particle swarm optimization', *Proceedings of IEEE International Conference on Neural Networks, ICNN 1995*, 27 November–1 December, Perth, Australia, pp.1942–1948.

Ma, Y. and Zhang, C. (2008) 'Quick convergence of genetic algorithm for QoS-driven web service selection', *Computer Networks*, Vol. 52, No. 5, pp.1093–1104.

Qiang, H., Jun, Y., Yun, Y., Kowalczyk, R. and Hai, J. (2008) 'Chord4S: a P2P-based decentralised service discovery approach', *Proceedings of IEEE International Conference on Services Computing, SCC 2008*, 7–11 July, Honolulu, HI, USA, pp.221–228.

Ragab, K., Haque, A.U. and Zahrani, M. (2008) 'Wait time management for efficient web service discovery service with P2P architecture', *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications, HPCC 2008*, 25–27 September, Dalian, China, pp.923–928.

Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Shenker, S. (2001) 'A scalable content-addressable network', *Computer Communication Review*, Vol. 31, No. 4, pp.161–172.

ShaikhAli, A., Rana, O.F., Al-Ali, R. and Walker, D.W. (2003) 'UDDIe: an extended registry for Web services', *Proceedings of the International Symposium on Applications and the Internet Workshops, SAINT-W 2003*, 27–31 January, Orlando, Florida, USA, pp.85–89.

Singhera, Z.U. (2004) 'Extended Web services framework to meet non-functional requirements', *Proceedings of International Symposium on Applications and the Internet Workshops, SAINT-W 2004*, 26–30 January, Tokyo, Japan, pp.334–340.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. and Balakrishnan, H. (2001) 'Chord: a scalable peer-to-peer lookup service for internet applications', *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computers Communications, SIGCOMM 2001*, 27–31 August,San Diego, CA, USA, pp.149–160.

Su, S., Li, F. and Yang, F.C. (2008) 'Iterative selection algorithm for service composition in distributed environments', *Science in China Series F-Information Sciences*, Vol. 51, No. 11, pp.1841–1856.

Tasaka, S. and Yoshimi, H. (2008) 'Enhancement of QoE in audio-video IP transmission by utilizing tradeoff between spatial and temporal quality for video packet loss', *Proceedings of IEEE Global Telecommunications Conference, GLOBECOM 2008*, 30 November–4 December, New Orleans, LA, USA, pp.1–6.

Wang, S.G., Liu, Z.P., Sun, Q.B., Zou, H. and Yang, F.C. (2013a) 'Pruning redundant services for fast service selection', *International Journal of Computational Method*s, Vol. 10, No. 6.

Wang, S.G., Sun, Q.B. and Yang, F.C. (2010) 'Towards web service selection based on QoS estimation', *International Journal of Web and Grid Service*s, Vol. 6, No. 4, pp.424–443.

Wang, S., Sun, Q., Zou, H. and Yang, F. (2013b) 'Particle swarm optimization with skyline operator for fast cloud-based web service composition', *Mobile Networks and Applications*, Vol. 18, No. 1, pp.116–121.

Wang, S.G., Zheng, Z.B., Sun, Q.B., Zou, H. and Yang, F.C. (2011) 'Reliable web service selection via QoS uncertainty computing', *International Journal of Web and Grid Services*, Vol. 7, No. 4, pp.410–426.

Wang, S., Zhu, X., Sun, Q., Liu, Z. and Yang, F. (2012) 'A distributed quality of service index framework', *Advanced Science Letters*, Vol. 7, No. 1, pp.98–102.

Yu, T., Zhang, Y. and Lin, K.J. (2007) 'Efficient algorithms for web services selection with end-to-end QoS constraints', *ACM Transactions on the Web*, Vol. 1, No. 1, pp.1–26.

Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J. and Sheng, Q.Z. (2003) 'Quality driven web services composition'. *Proceedings of the 12th international conference on World Wide Web, WWW 2003*, 20–24 May, Budapest, Hungary.pp.411–421.

Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J. and Chang, H. (2004) 'QoS-aware middleware for web services composition', *IEEE Transactions on Software Engineering*, Vol. 30, No. 5, pp.311–327.

Zhang, C., Krishnamurthy, A. and Wang, R. (2005) 'Brushwood: distributed trees in peer-to-peer systems. in Castro, M. and Renesse, R. (Eds): *Peer-to-Peer Systems IV*, Springer Berlin Heidelberg.

Zheng, Z. and Lyu, M.R. (2010) 'Collaborative reliability prediction of service-oriented systems', *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010*, 1–8 May, Cape Town, South Africa, pp.35–44.

Zheng, Z. and Lyu, M.R. (2013) 'Personalized reliability prediction of web services', *ACM Transactions on Software Engineering and Methodology*, Vol. 22, No. 2, pp.1–28.

Zheng, Z., Zhang, Y. and Lyu, M.R. (2010) 'Distributed QoS evaluation for real-world web services', *Proceedings of the 8th International Conference on Web Services, ICWS 2010*, 5–10 July, Miami, FL, USA, pp.83–90.

Zhu, X. and Wang, B. (2010) 'A distributed quality of service index framework', *Proceedings of 2010 IEEE Asia-Pacific Services Computing Conference, APSCC 2010*, 6–10 December, Hangzhou, China.pp.123–130.