Q-Peer: A Decentralized QoS Registry Architecture for Web Services*

Fei Li, Fangchun Yang, Kai Shuang, and Sen Su

State Key Lab. of Networking and Switching, Beijing University of Posts and Telecommunications
187#,10 Xi Tu Cheng Rd., Beijing,100876, P.R. China
pathos.lf@gmail.com, {fcyang, shuangk, susen}@bupt.edu.cn

Abstract. QoS (Quality of Service) is the key factor to differentiate web services with same functionality. Users can evaluate and select services based on their quality information. Traditionally, run-time QoS of web services is collected and stored in centralized QoS registry, which may have scalability and performance problem. More importantly, centralized registry can not operate across business boundaries to support global scale application of web services. In this paper, we propose a P2P (Peer-to-Peer) QoS registry architecture for web services, named Q-Peer. The architecture is a Napster-like P2P system, where query of QoS is naturally achieved by getting QoS storage address from service registry. Q-Peer employs object replication mechanism to keep load-balance of the whole system. We present two types of replication schemes and conduct comparison study. A prototype of Q-Peer has been implemented and tested on Planet-lab. Experimental results show that Q-Peer can automatically balance load among peers in different circumstances, so the system has good performance and scalability.

1 Introduction

Using web service to integrate business applications is one of the major trends in distributed computing. Web service, which specified by a set of XML (eXtensible Markup Language) based standards[1], is a standard way to improve interoperability between software running on different platforms over internet[2]. Web services can be published by providers and discovered by requesters based on their description. Services[1] can further be composed to a more powerful service to improve reusability. When using a certain web service, user experience is largely depends on quality of the service, so QoS information is essential in web service framework and should be properly processed.

* This work is supported by the National Basic Research and Development Program (973 program) of China under Grant No.2003CB314806; the Program for New Century Excellent Talents in University (No:NCET-05-0114); the Program for Changjiang Scholars and Innovative Research Team in University (PCSIRT); the Hi-Tech Research and Development Program (863 Program) of China under Grant No.2006AA01Z164; Collaboration Project with Beijing Education Committee.

[1] In this paper, we use *web service* and *service* interchangeably.

B. Krämer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 145–156, 2007.

Quality of service is non-functional properties of service, such as response time, availability, and price. It is a commonly accepted procedure that service requesters discover services by functional description and select services by QoS. Generally, service function is relatively stable throughout service lifetime, while QoS can change frequently with system status, load, network condition, etc. Maintaining the two types of information has different system requirements and design considerations. Thus service discovery and service selection are often accomplished on two entities respectively, called *service registry* and *QoS registry*. Currently, service registry in P2P manner is a hot research topic, but most of the published QoS registry works are still in centralized manner. They are sharing common shortcomings of centralized systems, like scalability, performance and single point failure. More importantly, because of business boundaries between different regions or management domains, a centralized system may not be able to support global scale web service interoperations.

In this paper, we propose a P2P QoS registry system for web services, named Q-Peer. The basic idea and a preliminary load-balance approach has been published as a work-in-progress paper in[3]. This work expanded our previous study by improving the load-balance approach and analyzing system performance in a series of experiments. Q-Peer is a P2P system which provides large-scale QoS storage, monitoring, collecting and query services. It can work with either centralized or decentralized service registries like UDDI (Universal Description, Discovery and Integration)[4] and other P2P service discovery system[5]. Every peer has its own policy to decide whether to accept a QoS registration request or a load-balance request. Q-Peer solves QoS object query problem by adding peer address into service registration information. User gets a peer address which storing the requested QoS object and accesses the peer directly, so that it does not need a query routing mechanism internally. QoS data of similar or identical services is clustered together, which makes query and comparison of QoS very efficient. Peers find other light-loaded peers to be neighbors by an autonomous load information dissemination scheme. Neighbors are expected to share load when needed. Data replication mechanism is applied on all peers to adjust load and improve availability. We propose two replication mechanisms and compare their effect by experiments. We have implemented the Q-Peer prototype and tested it on Planet-lab[6]. Experimental results show that Q-Peer has very good scalability and performance.

The rest part of this paper is organized as follows: Section 2 reviews some related works. Section 3 introduces the general model and design consideration of Q-Peer. Section 4 presents how to disseminate QoS and load information in Q-Peer. Section 5 proposes the load balancing approaches in Q-Peer. Section 6 presents the detail of experiments and analyzes the results. The paper is concluded in Section 7.

2 Related Works

QoS information processing is an important issue in web service framework. Most of the previous works are focused on how to evaluate and select web services, although they all mentioned certain kinds of QoS registries. Centralized QoS registry architecture has been proposed before. Maximilien et al. [7] proposed an agent based

architecture for processing QoS information. An ontology framework is builded to represent QoS knowledge. Serhani et al. [8] presented a QoS broker architecture and clarified its relationship with other entities in web service. Liu et al. [9] designed a QoS registry for a hypothetical phone service market place. The registry collect QoS information from two sources: one is active monitoring on service provider, another is user feedback. The registry can execute their QoS computation algorithm to rank services. Yu et al. [10] presented a broker based framework for QoS-aware web service composition. It maintains QoS information and integrates services on user's behalf. As far as we know, the only work mentioned a distributed QoS registry architecture is by Gibelin et al. [11]. They use hash-table based QoS indexing which is not efficient for QoS query problem. No detailed design information could be found in the paper.

In past several years, peer-to-peer paradigm has gained considerable momentum as a new model of distributed computing. P2P systems are created for file sharing at first, as Napster[12], Gnutella[13] and Kazaa[14]. P2P systems can be roughly divided to two categories by content distribution approach: structured and unstructured[15]. They have different query mechanisms, which should be chosen for different application scenarios. For scalability, autonomy and robustness of P2P systems, the P2P model has been introduced into distributed storage and information retrieving[16]. Some applications of P2P have already contributed to web service research, as distributed service discovery[17]. Replication is an important approach to improve P2P system performance. Cohen et al.[18] analyzed search efficiency by different replication strategies in unstructured P2P systems. Otherwise, we adopt replication mechanism to balance-load in Q-Peer.

3 System Model

Q-Peer is a peer-to-peer database system for register, storage and query of web service QoS. QoS data is recorded in XML documents. QoS query in Q-Peer is not conducted by query routing among peers, but by the support of service registry. For each service, service registry stores its functional description and at least one peer address containing its QoS. Users get QoS by directly access the address. This query mechanism can work with either centralized or decentralized service registry. The mechanism is suitable for QoS query, although different to common P2P database system. To query QoS without service description is meaningless, because no service user cares about service quality without known its function. The query mechanism in Q-Peer is similar to the most original P2P system-Napster[12], by a centralized index server cluster.

Services can be classified by their functionality, so corresponding QoS is naturally classified to QoS classes. QoS in a same class have same QoS metrics[7]. In Q-Peer, a QoS class is stored at one peer at first, but QoS data could be replicated when needed. Organizing storage by QoS class can improve efficiency because users often retrieve QoS of functional-identical services to compare and select from them. Different service selection algorithm can be deployed on peers to assist users[9][19]. If a service stored its QoS at a certain peer, the peer acts as its run-time monitor.

All peers are equal in Q-Peer. We do not use super-peer[20] because super-peer often intends to improve query efficiency, which is not a problem in our system. Peers employ a replication based load-sharing policy which utilizing spare resource on light loaded peers. Every QoS record may have several replicas on different peers. Service registry has a list of candidate peers for every service and chooses a random one when user query QoS. Peers have an autonomic mechanism to exchange run-time load information. Every peer keeps several other light-loaded peers as its neighbors for load sharing. The detailed mechanism will be presented in following sections.

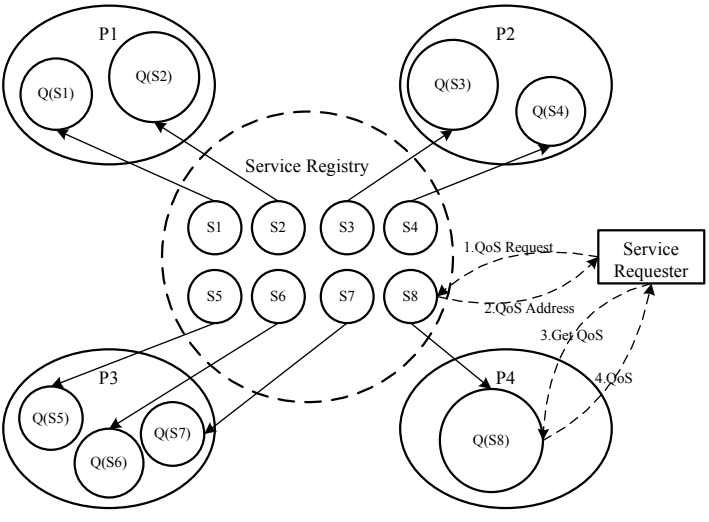

Fig. 1. General model of Q-Peer

Figure.1 illustrates a sample Q-Peer system containing 4 peers and 8 classes of services. Replicas are hided for illustrating our model clearly. Service registry in the figure can be either centralized or decentralized architectures. S_i is a service class which contains a number of same or similar service description. The QoS data set of a service class S_i is $Q(S_i)$. Each peer stores several QoS classes. Every service description contains the address of its QoS, as a pointer. When a service user needs to query QoS of a certain service, it sends a QoS request to service registry, then the registry will reply with a peer address. Service user can get QoS by direct accessing the peer.

4 Information Dissemination

In Q-Peer, two types of information change frequently which should be constantly updated and properly disseminated in the system. The first is service QoS. The second is load status of peers.

4.1 QoS Update

For a service *s* to be registered, which belongs to service class *S*, the service has functional and non-functional properties—service description *D(s)* and quality of the service *Q(s)*. *D(s)* is registered at service registry. If no service of *S* has been registered before, service registry will choose a random peer to store its QoS information. The selected peer is the *main peer* of *Q(S)* and the QoS data stored in this peer is the *main replica* of *Q(S)*. If *S* has been registered, *Q(s)* is added to its main peer. As soon as a peer decided to accept *Q(s)* of a new service, the peer contacts with the service and get current QoS for the first time.

For sharing load and improving availability, any *Q(s)* may have several replicas at different peers (The replication mechanism is presented in next section). These peers are called *replica peers* of *Q(s)*. Every time a service update its QoS, it only updates to the main peer. Then the other replicas are passively updated by the main peer.

4.2 Load Update

In Q-Peer, peer's load and capacity are characterized by the QoS access frequency on a peer. We assume every peer always has enough storage space for the cost of increasing storage is much lower than which of increasing CPU power or network bandwidth. QoS access comes from two major operations: one is update of QoS; another is query of QoS. For a peer *P* storing *n* services' quality information: $\{Q(s_1), Q(s_2), ..., Q(s_n)\}$, each service has an updating frequency $f^u(s_i)$ and a query frequency $f^q(s_i)$, the load of the peer is:

$$L(P) = \sum_{i=1}^{n} \left(f^u(s_i) + f^q(s_i) \right) \tag{1}$$

The estimated maxim allowed access frequency of *P* is the *maxim capacity* $C^M(P)$. The *available capacity* $C^A(P)$ is: $C^A(P) = C^M(P) - L(P)$.

Every peer has a list of several other peers' address, called Neighbor List (NL). The peers in NL are candidate peers to accept replication request of the NL owner. A neighbor item in NL is $N_i = \langle P_i, C^A(P_i) \rangle, (i = 1...m, a \le m \le b, 0 < a < b)$, where *m* is the total neighbor number, *a* and *b* are the lower and upper limit of *m*. Neighbor list is sorted by C^A in descent order. Items in NL can be dynamically added and deleted according to load change. When a new peer *P* adds to Q-Peer system, it will get a random list of peers as neighbors. *P* periodically sends its own $C^A(P)$ to neighbors and update NL by getting neighbors' C^A back from reply messages. If any peer received an unknown peer's C^A which was better than the last item in its NL or the NL was not full, the new peer is inserted as a neighbor. If NL exceeded the maxim number limit *b*, the last item is removed. A peer has a lowest capacity limitation *l* to take another peer as its neighbor. For any P_i in NL which $C^A(P_i) < l$, it is deleted. If item number in NL were lower than *a*, peer initiates a random walk process to find new neighbors. The random walk begins from a random peer in its NL. Random walk message contains the initiator's C^A for other peers to update NL if satisfied. For any

peer walked through, it sends a random item in its NL back to the initiating peer and forwards message to the item. The random walk stops for TTL limitation.

By this load information exchange approach, peers tend to take light-loaded peers as neighbors which are more likely to be able to accept replication requests. For peers with less available capacity which have not been taken as neighbor of any other peers, they still have chance to use other peers' resource. When they have more available capacity, they are added to its neighbor's NL.

5 Replication and Load Sharing

If a peer found itself under load pressure, it can request other peers to replicate some of its data for load-sharing. We present two replication schemes in this section: *Replicating QoS Class(RQC)* and *Replicating QoS Object(RQO)*.

5.1 Replicating QoS Class

Replicating QoS class considers load status of a whole QoS class and takes QoS classes as operation unit. This replication scheme makes service selection can be done at any peer containing a replica, because every replica is a whole class of QoS.

Every QoS class has $r\left(1 \le r \le K\right)$ replicas including the original one, where K is the maxim allowed replica number for a QoS class. If a peer's load were approaching threshold, it tries to replicate the most popular QoS class to the neighbor with most available capacity, which is the first neighbor in NL. A peer accepts replication request when all of the three conditions hold: the QoS class has less than K replicas, the spared capacity of the neighbor can satisfy load requirement, the neighbor has not stored this QoS class. Assume P_1 is the replication target peer, and $Q\left(S_i\right)$ is the class to replicate, the load satisfaction condition is:

$$C^A\left(P_1\right) > f^u\left(Q\left(S_i\right)\right) + \frac{r \times f^q\left(Q\left(S_i\right)\right)}{r+1} \text{ and } r < K \tag{2}$$

In (2), we can find that by replicating a QoS class, replication peer can share $1/r+1$ of the class' query load, but update load can not be leveraged because all replicas should keep consistency. With the growing of replica number, load sharing by replication can have less and less effect because $r/r+1$ is approaching 1. Furthermore, keeping more replicas consistent adds more load on the system. Thus, K should be a small number to make the approach effective.

If the available capacity of first neighbor $C^A\left(P_1\right)$ could not satisfy the replication requirement, the random walk process will be initiated to refresh the neighbor list. As soon as a replication peer is found, a replication request of $Q\left(S_i\right)$ is sent to new peer. Service registry is informed that a new replica can be selected to query QoS after replication is successfully performed.

If all QoS class in a peer had K replicas but it was still under load pressure, a random replica is chosen to delete. Service registry is informed before deletion, so that it would not get the class of QoS from this peer. Main peer of the QoS class is

also informed so that it would not update QoS to this peer. If the deleted replica was the main replica of the service class, another replica would be chosen as main replica and related service providers would be informed to update QoS to the new main peer.

5.2 Replicating QoS Object

The scheme of replicating QoS object replicates only one service's quality data every time. This replication scheme would not affect service selection function because a complete replica of a QoS class is still exists at the main peer which is responsible for updating QoS to all replicas.

When a peer finds itself under load pressure, it tries to replicate the most loaded QoS object. If the peer were still heavy loaded after a replication (This is highly possible if only one QoS object is replicated), it replicates the most loaded QoS object again. The new most loaded QoS object may or may not be the previous replicated one. This replication process would repeat until the peer's load is under predefined threshold. Every replication request is sent to the neighbor with most available capacity. We do not limit the replica number of a QoS object in this scheme, because for a single heavy loaded QoS object, its access frequency is always much higher than its update frequency.

In replication process, neighbor list may run out of neighbors because neighbor's available capacity is consumed. Random walk will be initiated when neighbor number is lower than limitation.

If neighbor list had been updated and run out again, and the peer were still under load pressure, peer begins to delete QoS object by descent order of object access frequency. Another deletion scheme is always taken periodically at all peers: if the query frequency of a QoS object was lower than its update frequency, it is deleted no matter what the peer status is. Only objects which are not main replica can be deleted.

6 Experiments

The Q-Peer prototype is developed in JAVA. We deploy our prototype on Planet-lab platform to test the performance and effectiveness of the system. "Planet-lab is a global research network that supports the development of new network services."[6] The platform can provide us nearly realistic distributed network environment.

6.1 Experiment Environment

The experiment environment is illustrated in Fig.2. Service registry is only a simulation program for testing the Q-Peer prototype, providing service classification and QoS address query function. Test agent simulates QoS access operations of service providers and requesters, which generates load to Q-Peer. We allocate a number of hosts in Planet-lab and deploy Q-Peer prototype on every host.

6.2 Evaluation Methodology

We expect that by replication, we can balance load among peers in Q-Peer which will result in better system performance. We characterize the Q-Peer performance by *system utility*, *balance degree,* and *request loss ratio*.

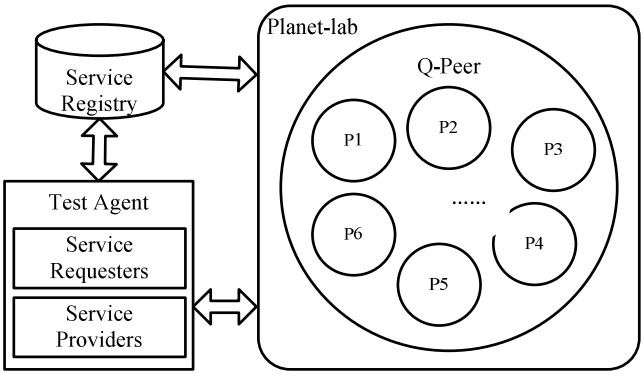

Fig. 2. Experiment environment

- System utility: the percentage of total successful query frequency to the total allowed capacity of all peers.
- Balance degree: the standard deviation of utility at all peers.
- Request loss ratio(RLR): the percentage of refused requests to total requests. A peer will refuse QoS access request when the load achieves allowed capacity.

The performance have been evaluated in three replication schemes: without replication, QoS class replication and QoS object replication. We allocated 50 hosts as 50 peers. Each host had a maxim allowed capacity randomly set from 500 to 10,000 (access/minutes). QoS data was added by the service provider simulator from test agent. Every provider updated its QoS once a minute. QoS query requests was simulated following Zipf distribution[21]. That is, a small portion of the QoS objects is queried much frequently.

Different scenarios was tested to find out the effect of replication schemes in different system status: 1, fixed QoS object number with growing request; 2, different QoS object number and class number with growing request; 3, different parameter configuration and their effect on load-balance. In the first scenario, 200 classes of QoS was generated and randomly distributed in peers. There were randomly 100-500 QoS objects in each class. Average request frequency was set at 50,000 requests/minutes., and grew 1000 every minute. The maxim replica number in RQC was set at 5. In the second scenario, two cases were tested. One was that we tested the performance when total QoS object number increased with a step of 1,000, but keeping the total class number at 200. The other was that we increased total class number by adding new classes to the system, but average QoS object number in each class did not change. In both cases, we grew requests frequency until RLR achieves 10%. The third scenario tested the effect of different system configuration. The maxim neighbor list length was set from 6 to 20 with a step of 2. Random walk TTL was set at 10. For the sake of peers' policy, not every peer is so generous to accept replication request. We tested the system utility when 10%-70% of peers would refuse other peers' replication request. In each configuration, we still tested system utility with growing request frequency until RLR achieves 10%.

6.3 Results and Analysis

In Fig.3, we illustrate the Q-Peer performance in different replication schemes. Figure 3(a) shows request loss ratio when request frequency grows. The no replication case has a linear growth of RLR because some of the peers are unable to response all the request at the beginning. With the request frequency growing, more and more peers are reaching capacity limits and more requests will be dropped. Obviously, replication cases can make the system scale up easily. The RQO case has better scalability because it balance load more accurately. Only the QoS object which needs replication will be replicated in RQO while the RQC replicates some light-loaded QoS object with the whole class. Figure 3(b) also shows that RQO has better balance degree. The balance degree keeps high at the beginning because requests are not evenly distributed in all peers and most of the peer has no need to replicate. When request frequency increases to system limit, the balance degree is decreased under 0.1 by RQO.

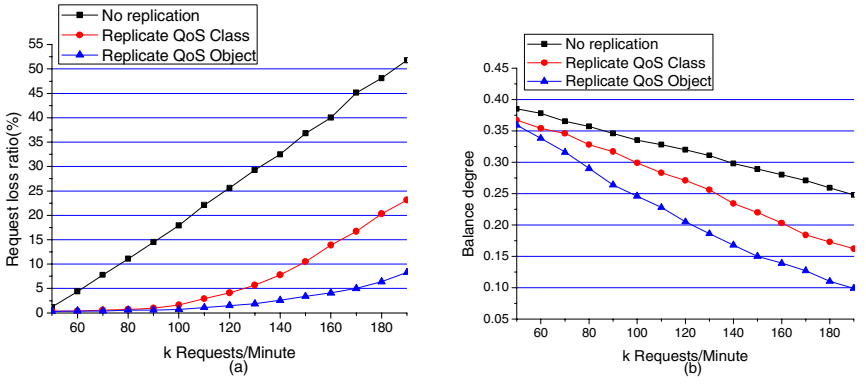

Fig. 3. (a) Request loss ratio with request frequency growing. (b) Balance degree with request frequency growing.

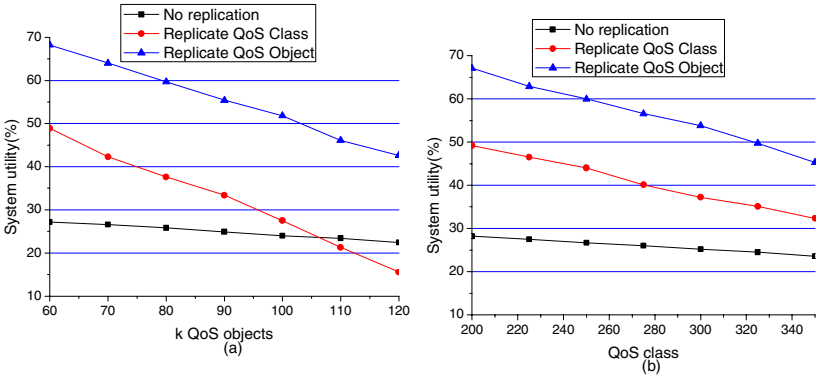

Fig. 4. (a)System utility with different QoS object number. (b) System utility with different QoS class number.

The total QoS number and its classification also has significant impact on Q-Peer performance because when QoS number growing, the QoS update operation will consume a significant part of total system capacity. Replications will increase update operations because we have to keep consistency of all replicas. In Fig 4, we can find that no replication case keeps a very low system utility at 20%-30%. The RQO case's system utility decreases with QoS number growth and drops to about 40% when QoS object number achieves 120,000. Considering our total system capacity is about 250,000, the system utility is still very good. A problem of ROC is illustrated in Fig 4(a). The system utility drops rapidly with total QoS number growing by expanding every class. The reason is that RQC has to replicate more QoS object when QoS number in each class growing. Surprisingly, system utility of RQC is worse than no replication case when total QoS number is approaching 110,000.

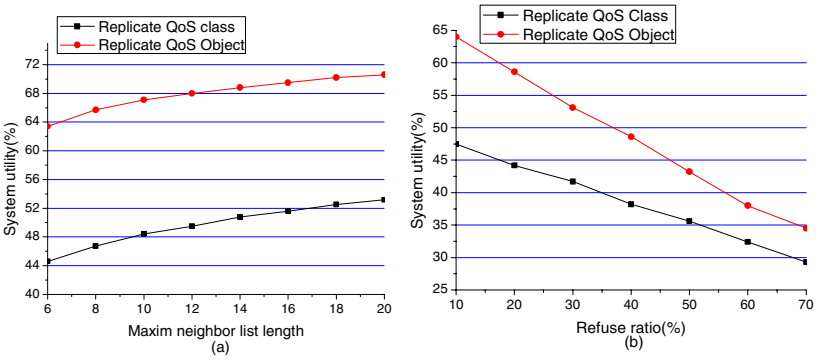

Fig. 5. System utility with different configuration

Figure 5 illustrates the impact of neighbor list length and refuse ratio. With neighbor list length growing, system utility can be improved. But long neighbor list can generate more network overhead for neighbor load update. And system performance improves slowly when neighbor list is long enough, because the neighbors in last part of the list can not provide much capacity for replication. The refuse ratio has obvious impact on system utility but this case is similar to realistic environment, where some of the registries have their own serving area.

While our experiments show that the RQO can give system better utility and balance peers, the RQO also has a problem that it balances system slower than RQC. Because RQC replicates a large number QoS objects every time, it can response to load change rapidly, which is important for burst requests.

7 Conclusion and Future Works

The web service infrastructure is evolving, so as the QoS registration architecture of web services. Most of the early works are centralized systems, but we believe that decentralized system is more suitable for global service oriented environment. In this paper, we presented a distributed web service QoS registry—the Q-Peer architecture.

The architecture is based on Napster-like unstructured peer-to-peer model. Every QoS object address is stored in service registry with its service description. Same or similar services' QoS is clustered together to conveniently expand other QoS operation like service selection. We designed simple but effective mechanisms to exchange load information between peers. Every QoS has several replicas to share load on different peers. Replication is based on load status exchange mechanism among peers. We presented two replication schemes—replicating QoS class and replicating QoS object, which have different granularity of replication. We tested the system performance with two replication schemes. RQO showed good load-balance effect in various system statuses.

We are still improving Q-Peer on the replication scheme. A more accurate and rapid load sharing approaching is needed. We are conducting theoretical analysis on replication behavior and effect of different configurations, which will make the system more adaptive on different scale and network status.

References

1. Tsalgatidou, A., Pilioura, T.: An Overview of Standards and Related Technology in Web Services. Distributed and Parallel Databases 12(2), 135–162 (2002)
2. Web Services Architecture, W3C (February 2004)
3. Li, F., Yang, F.C., Shuang, K., et al.: Peer-to-Peer based QoS Registry Architecture for Web Services. In: DAIS 07. The Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems. LNCS, vol. 4531, Springer, Heidelberg (2007)
4. UDDI version 3.0, OASIS
5. Verma, K., Sivashanmugam, K., Sheth, A., et al.: METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. Information Technology and Management 6(1), 17–39 (2005)
6. Planet-Lab Homepage, http://www.planet-lab.org/
7. Maximilien, E.M., Singh, M.P.: A Framework and Ontology for Dynamic Web Services Selection. IEEE Internet Computing 8(5), 84–93 (2004)
8. Serhani, M.A., Dssouli, R., Hafid, A., et al.: A QoS broker based architecture for efficient Web services selection. In: ICWS'05. Proceedings of IEEE International Conference on Web Services, pp. 113–120. IEEE Computer Society Press, Los Alamitos (2005)
9. Liu, Y., Ngu, A.H., Zeng, L.Z.: QoS computation and policing in dynamic web service selection. In: Proceedings of the 13th International Conference on World Wide Web, pp. 66–73. ACM Press, New York (2004)
10. Yu, T., Lin, K.J.: A Broker-based Framework for QoS-Aware Web Service Composition. In: EEE-05. Proceeding of IEEE International Conference on e-Technology, e- Commerce and e-Service, Hong Kong, China, IEEE Computer Society Press, Los Alamitos (2005)
11. Gibelin, N., Makpangou, M.: Efficient and Transparent Web-Services Selection. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 527–532. Springer, Heidelberg (2005)
12. Napster Homepage, http://www.napster.com
13. Gnutella Homepage, http://www.gnutella.com
14. KaZaA Homepage, http://www.kazaa.com

15. Lua, E.K., Crowcroft, J., Pias, M., et al.: A Survey and Comparison of Peer-to-Peer Overlay Network Schemes, IEEE Communications Survey and Tutorial (March 2004)
16. Koloniari, G., Pitoura, E.: Peer-to-peer management of XML data: issues and research challenges. In: ACM SIGMOD Record, vol. 34(2), ACM Press, New York (2005)
17. Schmidt, C., Parashar, M.: A peer-to-peer approach to Web service discovery. In: Proceedings of the 13th International Conference on World Wide Web, pp. 211–229 (2004)
18. Cohen, E., Shenker, S.: Replication Strategies in Unstructured Peer-to-Peer Networks. In: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 177–190. ACM Press, New York (2002)
19. Li, F., Su, S., Yang, F.C.: On Distributed Service Selection for QoS Driven Service Composition. In: Bauknecht, K., Pröll, B., Werthner, H. (eds.) EC-Web 2006. LNCS, vol. 4082, Springer, Heidelberg (2006)
20. Nejdl, W., Wolpers, M., Siberski, W., et al.: Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In: Proceedings of the 12th international conference on World Wide Web, pp. 536–543. ACM Press, New York (2003)
21. Adamic, L.A., Huberman, B.A.: Zipf's Law and the Internet. Glottometrics 3, 143–150 (2002)