Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Network and Computer Applications 28 (2005) 57–74
Journal of NETWORK and COMPUTER APPLICATIONS
www.elsevier.com/locate/jnca

Using service brokers for accessing backend servers for web applications

Huamin Chen*, Prasant Mohapatra

Department of Computer Science, Engineering II, One Shields Avenue, University of California, Davis, CA 95616, USA

Received 22 November 2003; received in revised form 21 February 2004; accepted 27 February 2004

Abstract

Web service infrastructures usually are comprised of front-end Web servers that accept requests and process them, and backend servers that manage data and services. Current Web servers use various API sets to access backend services. This model does not support service differentiation, overload control, caching of contents generated by backend servers. We have proposed a framework for using service brokers to facilitate these features. Service brokers are software agents that are the access points to backend services in Web servers. Unlike the current API-based scheme where accesses to backend services are through stateless and isolated APIs, in service broker framework, they are undertaken by passing messages to service brokers who gather all the requests and intelligently process them. We have prototyped this framework and validated its function in providing request clustering and service differentiation in accessing backend services. In addition, the performance in terms of the processing time is enhanced by this approach.

Keywords: HTTP; Web services; Service broker; Service differentiation; Overload control; Dynamic content caching

1. Introduction

Web servers have established their presence and usage in a variety of environment. More and more servers are being deployed for complex service environments, which also involve a variety of auxiliary servers. The platform independence and universal accessibility of Web servers have been leveraged to access other services like database,

* Corresponding author. Tel.: +1-5172829734; fax: +1-5172829734.
E-mail addresses: chenhua@cs.ucdavis.edu (H. Chen), prasant@cs.ucdavis.edu (P. Mohapatra).

1084-8045/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2004.02.001

mail, and directories. Web services like Microsoft.NET initiatives push such practice even further by facilitating more services accessibility through Web interfaces. It is conceivable that future Web servers will involve even more heterogeneous auxiliary service providers (hereafter, referred to as backend servers) to serve various tasks. Most large Web servers include a set of front-end servers that receive the requests from the clients. The requests are served by accessing a set of backend servers, which provide database, directory services, secure transactions, and other services. A schematic diagram of a typical Web server environment is shown in Fig. 1.

Backend servers can be categorized as tightly coupled or loosely coupled based on their connectivity and ownership with the initiating Web servers. Tightly coupled servers, like database and directory servers, are closely connected, usually in the same LAN, to the Web servers and belong to the same administrative authority. Tightly coupled servers are usually reliable and of high capacity. Loosely coupled servers represent Web servers belonging to other owners, which are not under control of the request initiating front-end servers. Web syndicates like My.Yahoo! and My.Netscape belong to loosely coupled Web servers. In accessing their services, the requests and response traffic must traverse WAN networks, which may incur higher latency and jitters than LANs. In more security-sensitive applications, authentication must proceed before further transactions. Since the loosely coupled servers are shared resources, service guarantee becomes an outstanding problem. We envision that in the future such services would be contract-based such that the service availability is honored only when the incoming traffic are within the contracted specifications. Loosely coupled services present a business model that has been existing in the current society. For instance, a travel agency has no sole control over airliners' ticketing services. Rather it contacts multiple airlines and selects the best deals for the customers.

The connectivity distinction between the two categories exposes different performance concerns. For tightly coupled services, the major performance issue is

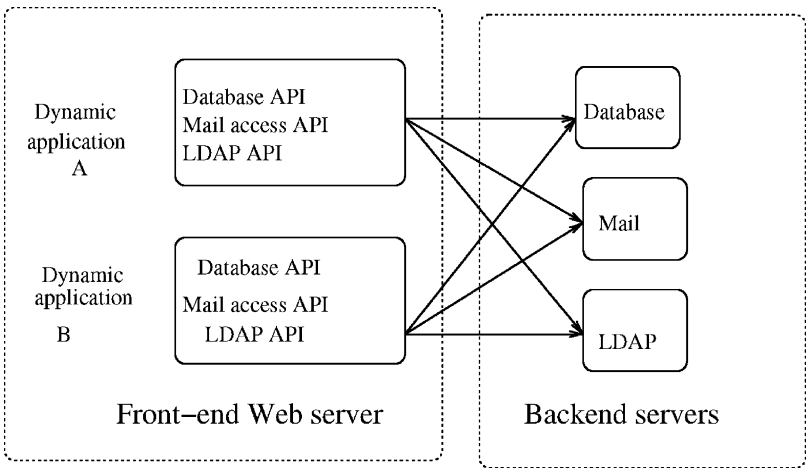

Fig. 1. API paradigm.

how to engineer the capacity to satisfy the front-end Web servers' requests. The inequivalence of the computational complexity and different levels of replicability between the front-end Web and backend servers make backend servers the likely performance bottleneck. For instance, a search operation involves traversal of database tables with many comparison operations, which only results in a few lines of output that are rendered in HTML pages. Moreover, many companies use centralized backend servers to serve multiple front-end Web servers to reduce the investment on the expensive backend servers. Thus the priority in this environment is how to prevent and control overload at the backend servers. Similar concerns were expressed by Zhang et al. (1999). In the loosely coupled environment, the volatile network conditions and the contract constraints demands efficient access schemes to ensure service availability and reliability.

In this paper we introduce a new concept and propose using *service brokers* to access backend services. Service brokers are software agents that act as access points to backend services in Web servers. Unlike the current API-based scheme where accesses to backend services are through stateless and isolated APIs, in the proposed scheme, backend services are accessed by passing messages to service brokers who gather all the requests and intelligently process them. In the tightly coupled environment, the service brokers can selectively drop some requests to reduce backend servers' load while facilitating service discrimination and QoS provision. In the loosely coupled environment, the brokers intelligently cluster the requests, cache the responses and prefetch the next possible queries in idle periods. We have prototyped this scheme and validated the feasibility in the testbed. We found that by properly clustering the backend server accesses, service brokers can significantly reduce the response time. In the service differentiation experiments, we demonstrate notable scalability improvement through fidelity variations. The experimental results depict a significant performance benefit through the usage of service brokers for backend services.

The rest of the paper is organized in this way. Section 2 discusses the current API-based framework and its drawbacks. Section 3 introduces the service broker model and its advantages in various aspects. Section 4 proposes two implementation approaches of service broker model. Section 5 presents the implementation of the service broker model and its ability in request clustering and service differentiation. It is observed that by varying response fidelity in different QoS levels, service brokers can improve responsiveness and scalability. The related works were presented in Section 6 followed by the concluding remarks in Section 7.

2. API-based backend service access

Currently dynamic Web applications are in forms of CGI executables that run in separate processes or use Servlet or scripting languages like PHP, JSP, and ASP which usually run in Web server processes. They access backend servers (Database, LDAP, mail or even other Web servers) through specific APIs like socket, ODBC or modules like COM. The APIs reside inside the application process space and share no information among different processes. The paradigm adopted in the contemporary Web servers is

shown in Fig. 1. In this paradigm, applications A and B need to access database, mail and directory servers. They use the respective API sets to accomplish the tasks: in order to access database server, a connection needs to be established before any queries are initiated which are followed by the connection tear-down; similar procedures are applied in accessing the directory server. Since A and B are in different process space, their use of API sets is independent and they do not share anything. Thus even though A and B may access database or directory service simultaneously, they each need to establish a connection before using any services.

The drawbacks of this paradigm are:

- *QoS is not guaranteed*: Accesses to backend service are served on the FCFS basis. Unless QoS specifications propagate through all backend servers, there is no guarantee that they can be honored. Currently most of the backend servers like database servers, directory servers, and file servers do not provide QoS support. It is also likely that heterogeneous backend servers may have different QoS notations. For example, the file servers may cluster requests whose accesses are in adjacent disk layout. Database servers may cluster queries that access the same table. Thus this architecture may not be able to enforce QoS specifications throughout the entire systems and is also subject to the priority inversion problem.[1]
- *Hot spot unawareness*: Although overload control in Web servers has received increasing attention recently, there has been limited research work on backend servers' overload control. In fact, backend servers are more likely to be overloaded: backend servers are usually centralized legacy systems; they are not as replicable as the frontend Web servers. When the traffic to the same backend server is beyond its capacity, a hot spot is generated and this backend sever is likely to become bottleneck of the entire request handling process. Hot spots generated in backend servers are at most known to those who are using the service. Other processes are unaware of the overload due to the independence of the request handling processes. Thus overload could spread to other processes. The increased number of processes that are trapped in hot spots could impose serious threat to the overall server performance in some server architectures. For example, in Apache Web server, each request is handled by a dedicated server process. In order to process incoming requests, more child processes must be forked if others are busy. Thus processes trapped in accessing overloaded backend resources essentially exacerbate the overall performance.
- *Accesses are isolated, and thus not optimized globally*: Most of API libraries for backend server accesses do not share states or results among individual instances. Each application send requests and launch I/O operations separately even for identical operations. This drawback has been recognized and techniques like connection pooling are proposed: the front ends maintain a connection pool with the database server so the applications can use the open connections directly. This technique, however, is limited to database. Our service broker framework can make it available to applicable services.

[1] A low priority request could get service earlier than a high priority request.

3. Service broker paradigm

Motivated by the shortcomings of the contemporary model of backend accesses, we propose a service broker model in which instead of using APIs to access backend servers, Web servers pass requests onto intermediate dedicated processes (referred as service brokers). The schematic architecture is shown in Fig. 2. In this architecture, dynamic applications A and B do to have to invoke APIs to access backend services. They only pass messages to individual service brokers in some formats that contain their QoS specification and queries. Service brokers receive, sort and rewrite these messages according to their QoS levels and carry out the real query across connections. These connections can be established in advance to reduce the setup overhead, which is especially beneficial to loosely coupled environment. Upon receiving replies from backend servers, service brokers send the results to the dynamic applications and make a local copy, if possible, to serve similar requests. When hot spot occurs, the brokers can take appropriate actions globally across all the requests. In short, service brokers depart from current decentralized backend server access to a moderated control model.

As shown in Fig. 3, service brokers are independent of the Web application logic and are built on top of the API sets. It is per service based. Though schematically similar to COM and Enterprise Java Bean, service brokers have more meaningful attributes.

At the first glance, this new proposal appears to incur extra overhead for inter-process communication between service brokers and web processes. However, to access backend services, such overhead is insignificant. For a database access, database connection and tear-down, which are required in API model for each access, would be more expensive than inter-process communication. In the proposed approach, DB brokers maintain persistent connection thus saving the cost of connection setup.

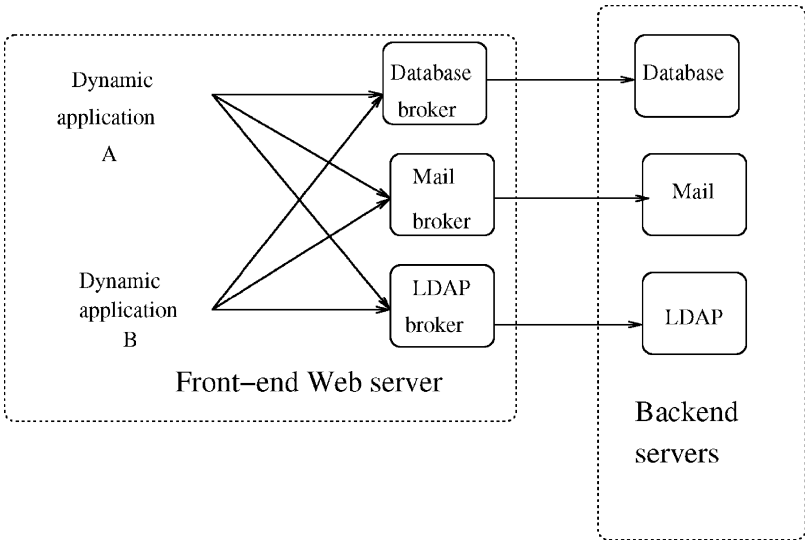

Fig. 2. Service broker framework.

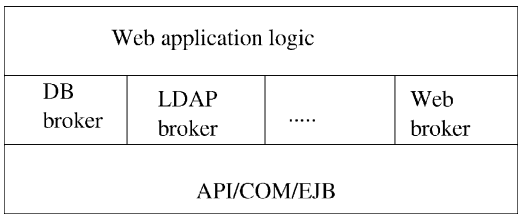

Fig. 3. Service broker layer.

The service broker framework has the following advantages:

- *Accesses can be clustered and optimized*: Multiple query clustering and optimization (Sellis, 1988) has been studied in database systems. Service brokers can provide similar optimization among requests in absence of the backend server support. For instance, two separate accesses to a remote Web server to get page 1.html and 2.html can be combined using MGET command (Franks, 1994) as 'MGET URI:1.html URI:2.html' at the broker and the results are appropriately split and sent to the request initiators. It appears that channel multiplexing may increase the workload at service brokers, which usually reside in the front-end Web servers. However, the transfer of computing load from backend servers to the front-end Web servers is viable since the front-end Web servers are easily replicable. Moreover, the same connection to backend servers can be multiplexed. In contrast to using independent connections to access backend servers in the API model, a single connection between the service broker and the backend server can be multiplexed to serve multiple applications and thus reduce the connection overhead while processing queries in bulk.
- *QoS awareness*: Service brokers are able to consistently honor the QoS priorities without propagating QoS specification to backend servers. QoS rules are fed into all the service brokers. Based on these rules, service brokers reshuffle the queued requests and schedule according to their priorities. When traffic intensities of QoS classes exceed their limits, their requests are dropped and other classes are not affected. Therefore lower priority requests give way to higher priority classes, thus avoiding the priority inversion problem. We demonstrate this functionality in Section 5.
- *Backend server overload control*: Unlike API-based architecture where each backend service access is unrelated, service brokers process all the requests and thus are aware of the states of the associated backend servers. Service brokers can notify request schedulers about the onset of hot spots or respond to the requests with lower fidelity results which would indicate that the system is busy. In some cases, it enables the use of cached results from the previous queries.
- *Caching of query results*: Since service brokers receive all the query results from the same backend servers, they can cache some of the results to serve similar requests. For example, consider an online Web site that provides movie schedules. All the schedule information is stored in a database. In the peak time, there would be a lot of requests for the same movie schedule. If the results are not cached, the database has to

process the same query repeatedly and will contribute to the response delay. In contrast, service brokers can be configured to cache the popular query results and promptly return them to the front-end Web servers. This approach reduces the number of requests to the backend servers and reduces the response time. This feature is especially beneficial in loosely coupled environment where accesses to backend service need to traverse high latency networks. This feature, however, gives rise to the problem of validating cached data when the original data change in the backend server. Several approaches (Luo et al., 2002) have been proposed to solve this problem based on request snooping. Similar techniques can be incorporated in the service broker framework.

- *Prefetching*: Service brokers enable forecasting of the next possible queries and prefetching the necessary information. For instance, a news provider website periodically updates the online headlines. Service brokers can be synchronized to prefetch them when the server load is not high. So the requests for the news can be served immediately without accessing the backend servers.
- *Transaction integrity assurance*: Service brokers can track the individual requests and enable transaction integrity assurance. Server brokers can be configured to properly allocate resources to ensure the completeness of sessions which consist of multiple requests. Currently, HTTP requests are stateless, the failure of individual requests is presumed to have little impact on others. In some occasions, especially e-commerce, single request failure would lead to broken sessions. For example, a computer manufacturer conducts an online purchase from multiple vendors. It first selects proper monitor models from a monitor vendor site (step 1), then video cards from the other vendors (step 2), then comes back to the monitor vendor again to match and purchase the best models (step 3). If somehow during step 3, the channel to the monitor vendor site is congested, the transaction could abort. The service brokers can recognize the subtlety of each access by proper tagging and gradually increase the priority of the subsequent accesses that belong to the same transaction. While accessing the monitor vendor, the broker would put more weight on those accesses whose transactions are in step 3 and selectively drop those whose transactions are in step 1 if the load is high. In API-based access models, each access within a transaction is stateless and is processed in the same way, i.e. access in step 3 is treated the same as that in step 1. If service brokers are enabled to communicate with each other, they can exchange state information to ensure that transactions involving different backend servers are properly protected.
- *Load balancing*: Load balancing have been widely used to improve front-end Web servers' performance (Bryhni et al., 2000; Cardellini et al., 1999). The basic idea is to select a candidate server and distribute the workload across multiple servers. Similar idea is also applicable to backend service accesses. In the API-based architecture, since no state information is shared in individual accesses, it can only work in a speculative manner. The service brokers can track the traffic and monitor their workload and accurately distribute the workload among the backend servers to achieve a balanced load.
- *Amortized context switching*: Accesses to backend servers are done in bulk at service brokers to reduce the number of context switchings. The service brokers can reside separately from the Web servers to facilitate the overall system optimization.

4. Implementation of service broker framework

The service broker scheme can be incorporated into current Web servers using two different approaches: centralized and distributed models.

Fig. 4 illustrates the centralized model. In this approach, the Web server manages all the load and QoS requirements. The load information from the service brokers is obtained through a listener thread and all the requested URLs' resource profiles are accessible to the Web server. For a particular incoming request, the Web server checks its resource requirements and current load status of the brokers before the request proceeds to the normal handling process. If overloading occurs at any of the processing stage, the request is aborted before any real processing starts and an error message is sent to the end user to indicate the resource unavailability at the server. While this approach is efficient, it is not very scalable. When the number of brokers or the update frequency of load information increases, the listener thread which resides in the same process space as the Web servers could be overwhelmed with update messages, which may erode away computing power from the Web server processes.

The distributed model is depicted in Fig. 5. In this model, the Web server imposes no admission control restrictions. Requests are forwarded to the brokers together with their QoS profiles. The brokers decide whether to send the requests to the backend servers or, in case the traffic intensity exceeds pre-defined thresholds, acknowledge the requests with some adaptive messages. Examples of such messages include cached results from previous queries with lower fidelity or simply an indication that the system is busy.

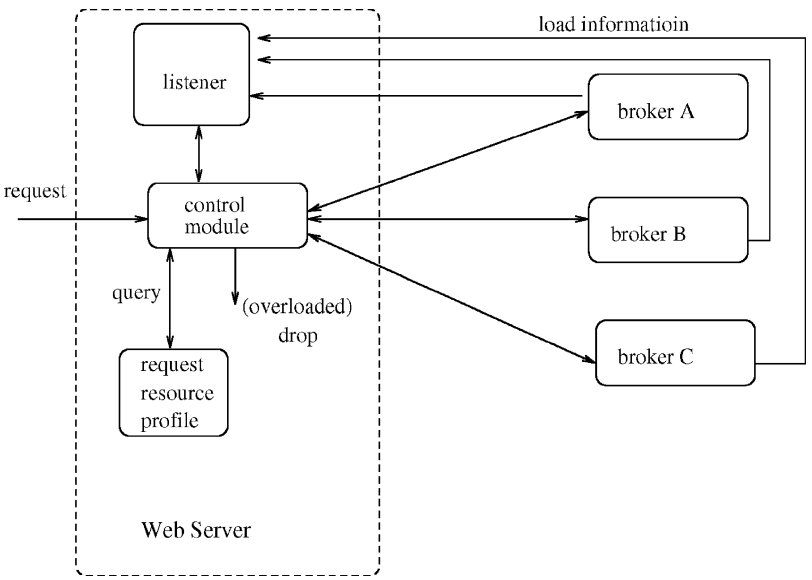

Fig. 4. Centralized model.

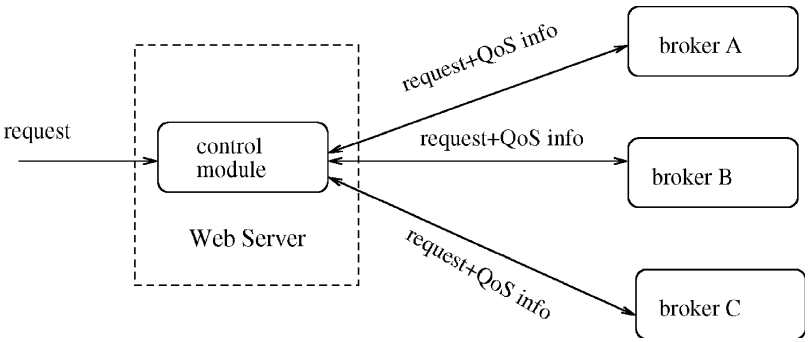

Fig. 5. Distributed model.

5. Experiment

We have prototyped the service broker framework to demonstrate its feasibility and ability in request clustering and service differentiation. The testbed configurations and results are discussed in this section.

5.1. Request clustering experiment

We evaluated the performance of the request clustering technique. The objective of this test is to investigate the effect of request clustering on the response time of Web servers.

5.1.1. Testbed configuration

Fig. 6 shows the testbed configuration. The front-end Web application that the client requested was to send a request to the backend server and relay the response to the client. The backend server's task was to look up a database table that contained 42,000 records and retrieve the appropriate records according the query conditions. The frontend and backend servers ran Apache for Linux, the database was MySQL, and the client ran ab (Apache benchmarking tool).

Since the request clustering is application specific, a general request clustering engine implementation is beyond the scope of this paper. We used the following scheme to simulate the clustering. The backend Web server access script was to generate a random query command and retrieve the corresponding results from the database. If the accesses to the backend server were not clustered, each single access to the script called only one database query. The service broker in the front-end Web server could gather all

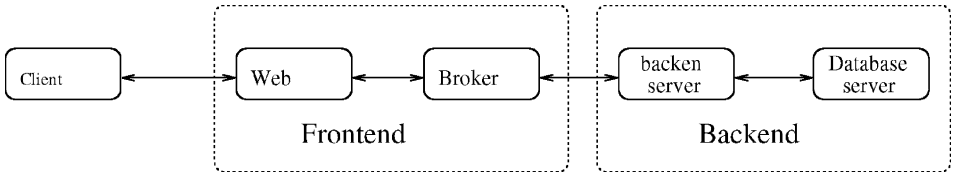

Fig. 6. Request clustering testbed.

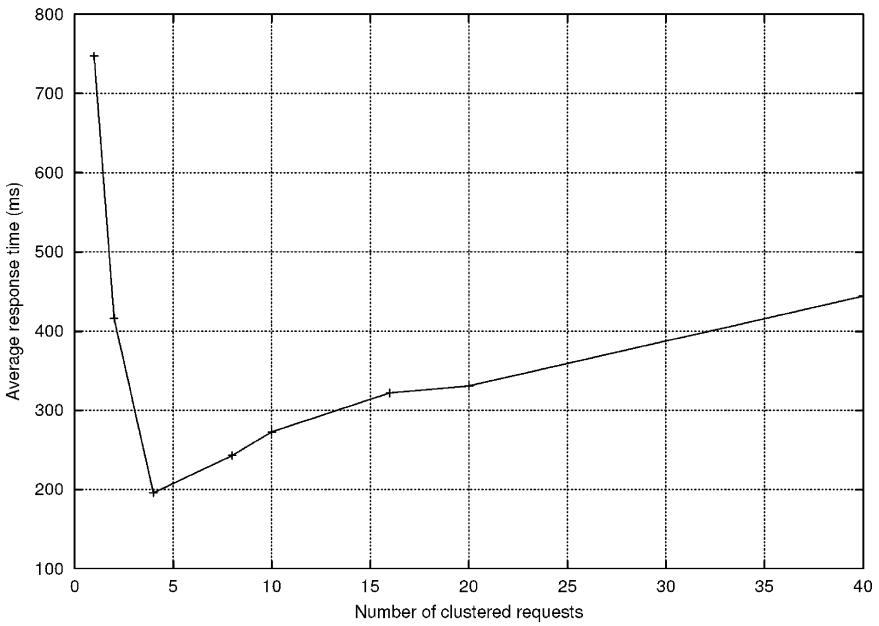

Fig. 7. Request clustering experimental results.

the requests and rewrite the query command to notify the script to repeat the same workload multiple times to achieve clustering.

5.1.2. Experimental results

Fig. 7 presents the experiment results from the tests in which 40 simultaneous requests were initiated to the front-end Web server. The broker was configured to cluster various numbers of requests (termed as degree of clustering) each time. The backend Web server was configured to accept at most five simultaneous requests from the front-end Web server. It is observed that the response time measured at the client side first declines with respect to the number of the clustered requests. This is because backend server's capacity (no more than five simultaneous requests could be served) could not meet the traffic intensity from the front-end Web server application, thus some of the accesses were queued resulting in long response time. By clustering the accesses, the service broker initiated fewer simultaneous accesses to the backend server, thus reducing the queueing time and improving the overall performance. When the degree of clustering increased, however, the backend server's capacity could meet the traffic intensity and the processing time began to dominate the performance. Since the script that processed the clustered requests needed to repeat the same workload multiple times, it took more time to serve the requests with a large degree of clustering.

It infers from the observation that clustering must be configured according to the backend server's capacity to achieve the maximum performance benefit. Nevertheless, the performance benefits from clustering is significant as is evidenced from Fig. 7.

5.2. Service differentiation experiment

As discussed in the previous sections, there have been no consistent schemes to provide service differentiation in accessing backend servers. We designed an experiment to demonstrate how service differentiation could be supported in the service broker framework.

5.2.1. Testbed configuration

The prototype of the service broker scheme using the distributed model was implemented in the *boa* Web server. Boa is an open sourced, light-weighted, and efficient Web server. Similar implementation can be done on other Web servers. The objective of this experiment is to demonstrate the service broker's capabilities for service differentiation and the performance benefits.

As illustrated in Fig. 8, in the testbed, there were three service brokers (brokers 1, 2 and 3), each connected to a Web server as backend server (backend server 1, 2 and 3, all ran Apache Web servers.). The brokers and the front-end Web server exchange request and response messages through lightweight UDP. The brokers communicate with the backend servers in HTTP. All the front and backend servers and brokers ran on Redhat Linux 7.2. The client side used WebStone 2.5 on 4 Sun UltraSparc workstations. Three workstations A, B, and C were designated as Web clients with QoS levels 1, 2 and 3, respectively.

The backend services provided by each backend servers are CGI requests with bounded processing time. The processing time of each of the services is 1, 2 and 3 s at the backend servers 1, 2 and 3, respectively. The QoS specification used in the testbed is just a binary mode of forward or drop: QoS level i means that the request is forwarded to the backend servers if the number of the outstanding requests is $(10 - x)10\%$ of the threshold. For instance, a request with QoS level 2 is allowed to send query to the corresponding backend server only when the number of outstanding requests is below 80% of the threshold. The thresholds at each broker were set to be 20, i.e. at most 20 requests are allowed to be outstanding in each of the backend servers. The thresholds of queues ensure bounded queueing time. The maximum number of server processes in each of the backend end Web servers is set to be 5, therefore only 5 requests can be processed simultaneously in each of these servers and the rests are queued. If a request was dropped at the broker, a short message was sent to the front-end Web server immediately. Therefore, longer the processing time a request undergoes, higher the fidelity it receives.

5.2.2. Experimental results

We set up an experiment to demonstrate the QoS assurance provided by service brokers. The backend servers were assured to be QoS-unaware. A normal Web request consists of three stages which take approximately 6 s to complete. In the service broker scheme, if a request's QoS level does not meet the current load status, a low fidelity response is replied immediately. Thus the longer is the processing time of a request, the higher is the fidelity of service it receives.

The experiments were conducted for both API-based accesses and service broker-based configurations. Fig. 9 presents the results reported by WebStone. It is observed that in API-based accesses, the average processing time is linear with respect to the number of requests. In the service broker model, the processing time first rises when the number of

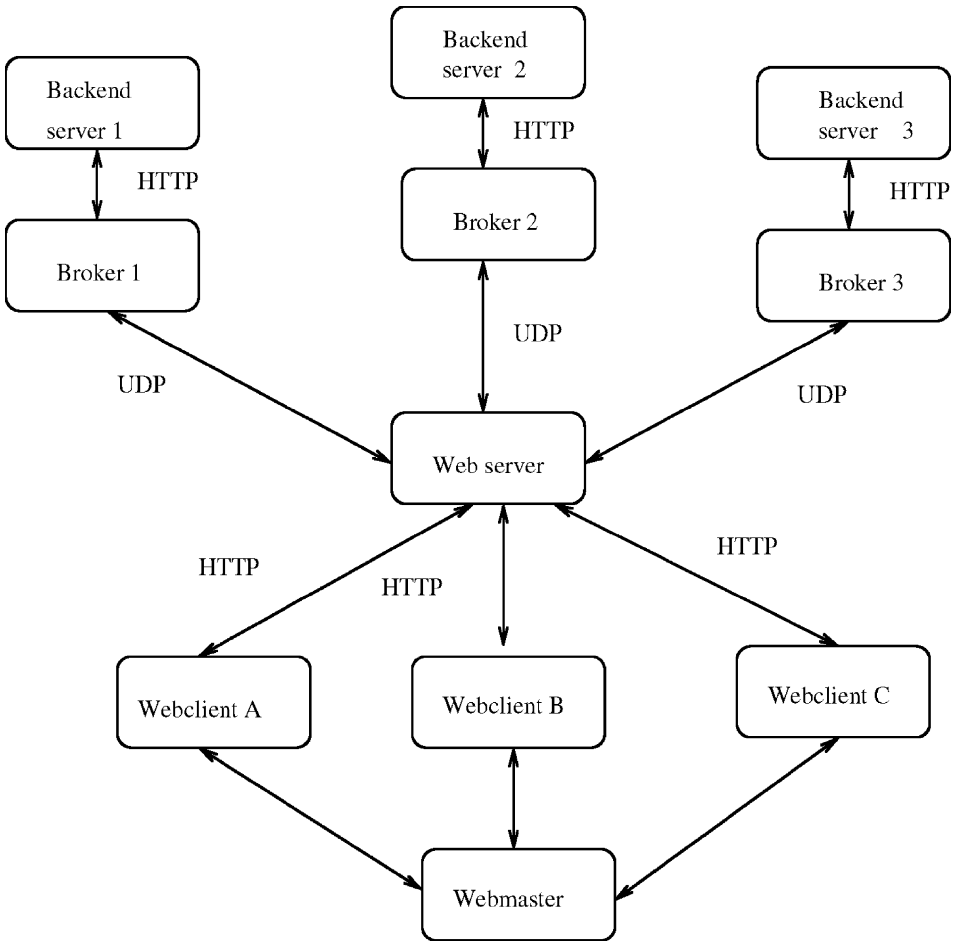

Fig. 8. Service differentiation testbed configuration.

clients is small so service brokers can admit most of the requests to access backend servers. When the number of clients increases, more and more requests in lower priority classes are dropped at the service brokers and they are informed promptly without any backend service, so the processing time declines. Lower priority requests are dropped to give way to process requests in higher priority classes. Thus the fidelity of each QoS class is differentiated. In this regard, the priority inversion abnormality is avoided. Table 1 presents the number of requests completed in each QoS class from the Web server's access logs. The numbers of completed requests in API-based settings ranged between 740 and 750. Since WebStone clients were best-effort based, with shorter processing time, more number of requests were initiated. As a result, more requests were processed from lower QoS levels. We believed that the traffic intensity among QoS levels is inversely proportional to their priorities; higher priority should be provisioned with lower request rates such that the lower priority requests do not starve.

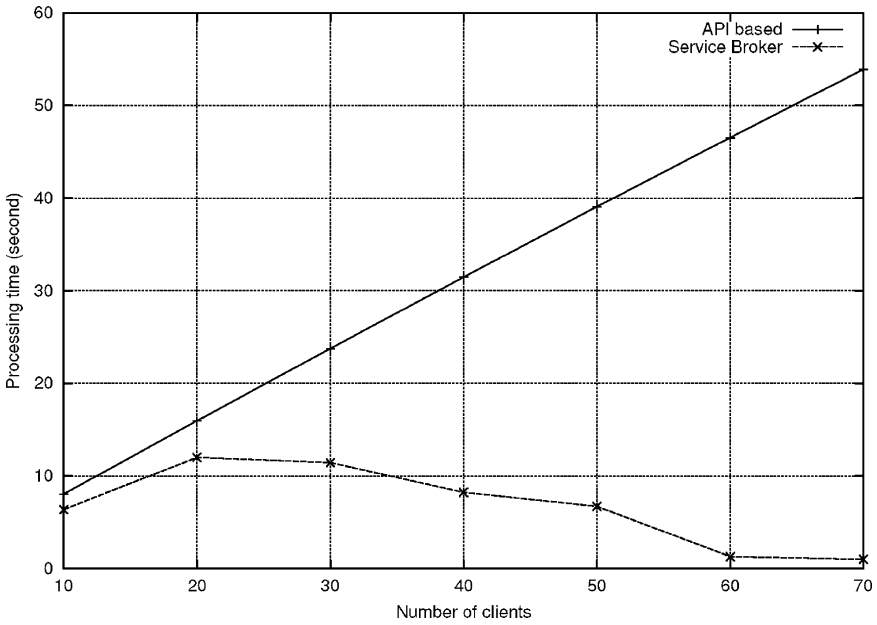

Fig. 9. Processing time of API and service broker-based settings.

Fig. 10 depicts the average processing time of each QoS class under various loads. The requests with higher QoS level experienced longer processing time, which means that the fidelity of the response is higher. For each curve, an increase in processing time is observed followed by a decline when the number of clients reaches a certain point. The reasons behind this phenomena are: when the number of clients was low (from 10 to 20), only a few requests were queued in service brokers, thus the increase of requests could be accommodated; after some points, the traffic intensity exceeded the capacity and some lower prioritized requests were responded by service brokers with lower fidelity messages without being forwarded to the backend servers. High priority requests still enjoy high fidelity service but the increase in traffic intensity contributes to the queueing time. When the number of clients exceeded 50, more requests from the high QoS classes were dropped at the service brokers and thus their average processing time declined as well.

Table 1
Number of completed requests at each QoS level

Number of clients	QoS 1	QoS 2	QoS 3
10	378	281	288
20	355	353	302
30	395	466	723
40	479	494	1950
50	541	820	3107
60	732	1154	25,914
70	1303	3256	37,050

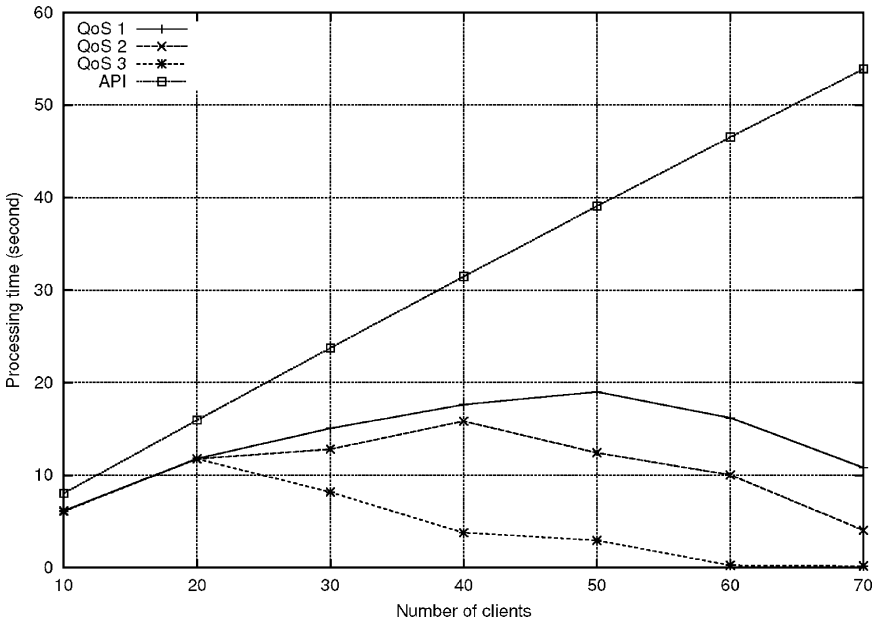

Fig. 10. Average processing time for each QoS level.

Figs. 11–13 present the drop ratios from each QoS class at the three brokers. It is observed that the when traffic was light (number of clients < 20), no drops occurred. When the traffic intensified, more lower priority requests were dropped. The drop ratios were mostly consistent with their associated QoS levels.

6. Related works

The related work can be categorized into component-based systems, content adaptation, and Web server clustering.

Pupeteer (Lara et al., 2001, 2002) is a component-based system for mobile devices. It shares the concepts of service brokers. In Pupeteer, network activity of mobile applications is governed by a common middleware which determines bandwidth allocation, content caching, and content transcoding. While the service broker framework applies to different system environment: interaction between front-end Web and backend servers. In addition to the aspects addressed in Pupeteer, more issues that are particular to Web services can be solved in this framework.

Content adaptation has been proposed for Web server overload control (Abdelzaher and Bhatti, 1999a) and QoS provision (Abdelzaher and Bhatti, 1999b; Fox et al., 1996). The basic idea of content adaptation is to render contents of different levels of fidelity. The adaptiveness can be based on the server load, available network bandwidth, and the end user's rendering capacity. Abdelzaher and Bhatti (1999a) proposed content adaptation for web servers under various server load. Chandra et al. (2000) proposed adaptive

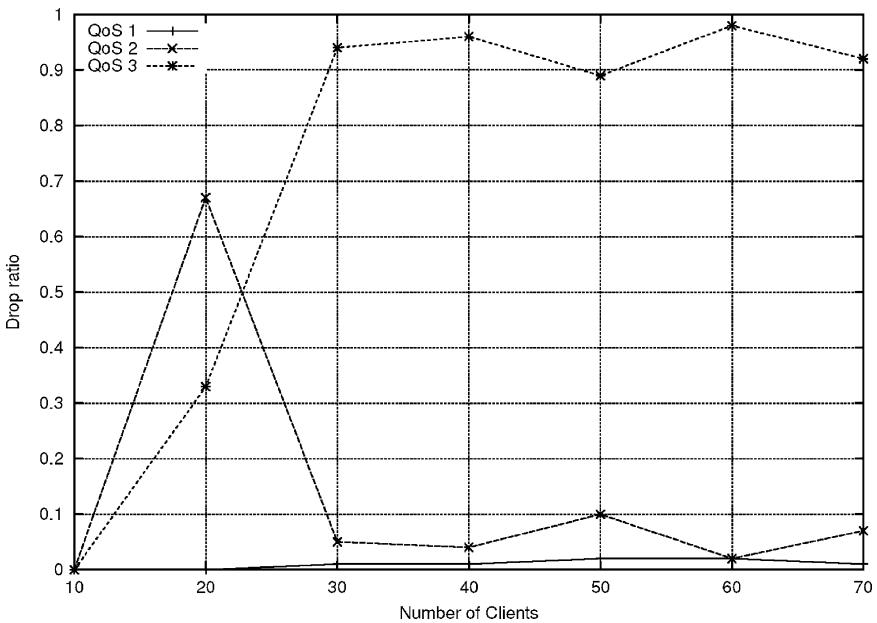

Fig. 11. Drop ratios at broker 1.

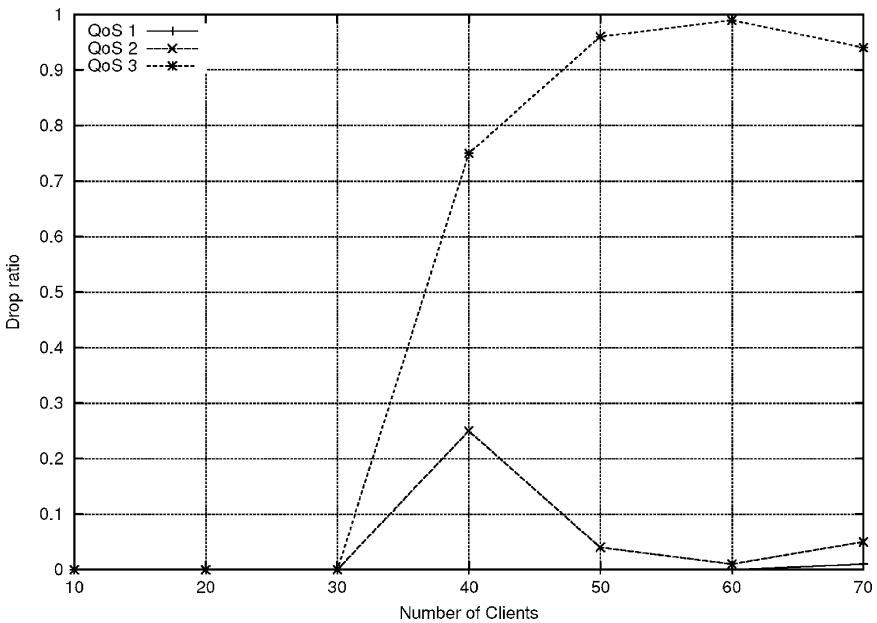

Fig. 12. Drop ratios at broker 2.

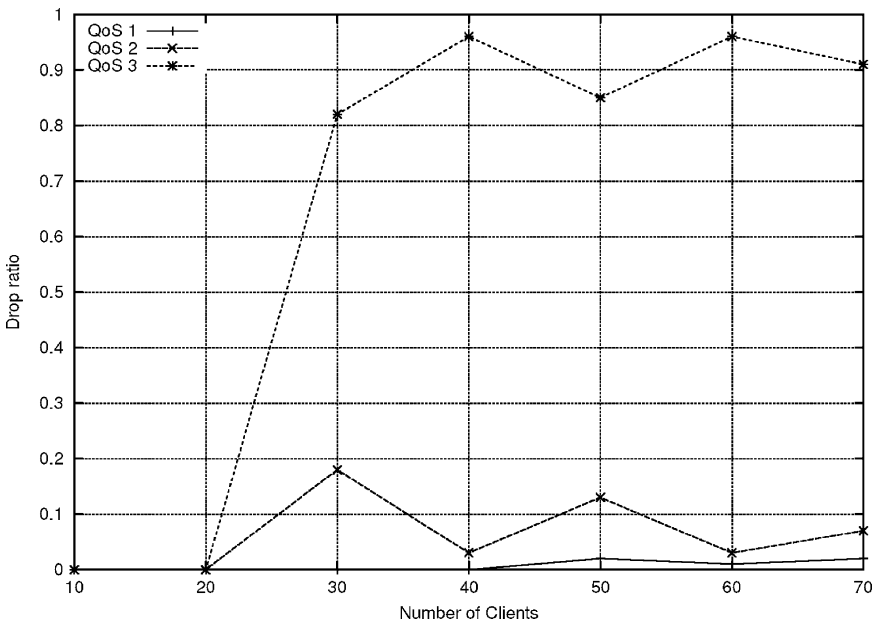

Fig. 13. Drop ratios at broker 3.

transcoding scheme based on user type (mobile, stationary) and network capacity. Both of these schemes provide QoS through service differentiation. However, none of these schemes has addressed how to apply content adaptation to relieve backend server overload or reduce cost associated with accessing backend servers. These issues can be resolved in the proposed service broker framework.

Most of the studies on multiple servers environment in Web research are focused on how to improve performance through load balancing techniques. SWEB (Andresen et al., 1996) investigated how to use DNS to implement load balancing and corresponding support in operating systems. Cluster Reserves (Aron et al., 2000) proposed a technique to provide resource isolation among different classes. Locality-aware request dispatching algorithms were proposed by Pai et al. (1998) and Zhang et al. (1999) to enable requests with related URL to be sent to the same server to achieve locality. Our work is complementary to these front-end Web server research.

7. Conclusions

Web servers have been used widely for applications that need to access backend services. The current API-based access schemes are not well suited for various reasons. We proposed a service broker framework that facilitates performance improvement and QoS provisioning. In the service broker framework, every backend service access is carried out by middleware agents that reside between front-end Web servers and backend servers. Service brokers can cluster and optimize queries and cache the results according

to a set of pre-defined rules. We proposed two implementation approaches and compared their performance implications. We have prototyped the framework to demonstrate their feasibility and evaluate their capacity to provide service differentiation. Experimental results demonstrated that service brokers can effectively use QoS specifications and provide scalable performance and service differentiation.

Acknowledgements

This research was supported in part by the National Science Foundation through the grants CCR-0296070 and ANI-0296034.

References

Abdelzaher T, Bhatti N. Web content adaptation to improve server overload behavior. International World Wide Web Conference, Toronto, Canada; 1999a.

Abdelzaher T, Bhatti N. Adaptive content delivery for web server QoS. International Workshop on Quality of Service, London, UK; 1999b.

Andresen D, Yang T, Holmedahl V, Ibarra OH. SWEB: toward a scalable world wide web server on multicomputers. Proceedings of the 10th International Parallel Processing Symposium, Honolulu, Hawaii, USA; 1996.

Apache HTTP Server Project, http://www.apache.org.

Aron M, Druschel P, Zwaenepoel W. Cluster reserves: a mechanism for resource management in cluster-based network servers. Proceedings of the ACM SIGMETRICS 2000 Conference, Santa Clara, California, ACM; 2000.

Boa Web server, http://www.boa.org.

Bryhni H, Klovning E, Kure O. A comparison of load balancing techniques for scalable web servers. IEEE Netw 2000;July/August:58–64.

Cardellini V, Colajanni M, Yu PS. Load balancing on web-server systems. IEEE Internet Comput 1999;3(3):28–39.

Chandra S, Ellis C, Vahdat A. Differentiated multimedia web services using quality aware transcoding. Proceedings of the IEEE Infocom 2000 Conference, Tel-Aviv, Israel; 2000.

Fox A, Brewer E, Gribble S, Amir E. Adapting to network and client variability via on-demand dynamic transcoding. ASPLOS 1996;.

Franks J. An MGET proposal for HTTP; October 1994. WWW-talkmailing list. http://www.webhistory.org/www.lists/www-talk.1994q4/0479.html

Lara ED, Wallach D, Zwaenepoel W. Puppeteer: component-based adaptation for mobile computing. Proceedings of the Third Usenix Symposium on Internet Technologies and Systems; 2001.

Lara ED, Wallach D, Zwaenepoel W. HATS: hierarchical adaptive transmission scheduling. Proceedings of the 2002 Multimedia Computing and Networking Conference (MMCN'02), San Jose, CA; 2002.

Luo Q, Krishnamurthy S, Mohan C, Pirahesh H, Woo H, Lindsay B, Naughton JF. Middle-tier database caching for e-business. SIGMOD 2002;.

MySQL, http://www.mysql.com/

Pai VS, Aron M, Banga G, Svendsen M, Druschel P, Zwaenepoel W, Nahum E. Locality-aware content distribution in cluster-based network servers. Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA; 1998.

Sellis T. Multiple query optimization. ACM Trans Database Syst 1988;13(1):23–52.

WebStone, http://www.mindcraft.com/webstone/

Zhang X, Barrientos M, Chen J, Seltzer M. HACC: an architecture for cluster-based web servers. Proceedings of the Third USENIX Windows NT Symposium; 1999.

Huamin Chen, received the B.S. and M.S. degrees from Huazhong University of Science and Technology, China, in 1996 and 1999, respectively. He obtained Ph.D. degree in the Department of Computer Science, University of California at Davis in 2003. His research interests are Internet server performance improvement and QoS provisioning, computer networks, and enterprise applications.

Professor Prasant Mohapatra, is currently a Professor in the Department of Computer Science at the University of California, Davis. In the past, he was on the faculty at Iowa State University and Michigan State University. He has also held Visiting Scientist positions at Intel Corporation and Panasonic Technologies. Dr. Mohapatra received his Ph.D. in Computer Engineering from the Pennsylvania State University in 1993. Dr. Mohapatra is was on the editorial board of the IEEE Transactions on Computers from 1999 to 2003, and has been on the program/organizational committees of several international conferences. He was the Program Chair for the PAWS Workshop during 2000 and 2001, and the Program Vice-Chair for the ICPP-2001, and INFOCOM 2004. He was also the Co-Editor of the January 2003 issue of IEEE Network. Dr. Mohapatra's research interests are in the areas of wireless mobile networks, Internet protocols and QoS, and Internet servers. Dr. Mohapatra's research has been funded though grants from the National Science Foundation, Intel Corporation, Panasonic Technologies, Hewlett Packard, and EMC Corporation.