A Tiered System for Serving Differentiated Content

Huamin Chen*
Department of Computer Science
University of California at Davis
Davis, CA 95616.
Email: chenhua@cs.ucdavis.edu.

Arun Iyengar
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598.
Email: aruni@us.ibm.com

Abstract. Contemporary Web sites typically consist of front-end Web servers, application servers, and back-end information systems such as database servers. There has been limited research on how to provide overload control and service differentiation for the back-end systems. In this paper we propose an architecture called tiered service (TS) for these purposes. In TS, there are several heterogeneous back-end systems to serve the Web applications. The Web applications communicate with a routing intermediary to intelligently route the queries to the appropriate back-end servers based on various policies such as client profiles and server load. In our system the back ends may store different qualities of data; lower quality data typically requires less overhead to serve. The main contributions of this paper include (i) a tiered content replication scheme that replicates tiered qualities of data on heterogeneous back ends with different capacity to satisfy clients with diverse requirements for latency and quality of data (ii) an application-transparent query routing architecture that automatically routes the queries to the appropriate back ends. The architecture was implemented in our test bed, and its performance was benchmarked. The experimental results demonstrate that TS offers significant performance improvement.

Keywords: Web, back-end server, tiered system, load balancing, service differentiation.

1. Introduction

Web sites typically deploy front-end Web servers that process HTTP requests, application servers that facilitate deployment of customers' business applications, and back-end enterprise information systems, such as database servers, for information retrieval and storage. While there has been extensive research into load balancing to front-end Web servers to handle the ever increasing traffic, less attention has been given to load balancing to the back-end servers. Back-end servers are often the bottleneck at data-driven Web sites. The amount of processing time for back-ends to process the data queries is often significantly higher than that for the HTTP requests.

* The work described in this paper was done while H. Chen was a summer intern at IBM's T.J. Watson Research Center.

Back-end servers are critically important for serving data-driven Web contents. Even if only a small percentage of all requests are dynamically generated by back-end servers, the overhead resulting from these back-end requests can be significant because dynamic requests can consume orders of magnitude more CPU time to satisfy than static requests. In many situations, the back-end servers will be the bottleneck in the system. Therefore, efficient systems for managing back-end data processing are critically important.

In this paper, we propose an architecture called tiered service (TS) to improve the performance of back-end database server accesses. In TS, multiple heterogeneous servers are clustered together to serve the requests from the Web servers, and accesses to them are differentiated in the quality of content and server capacity.

The heterogeneity of the back-end systems gives rise to the imbalance of performance and capacity. The better configured back ends can store higher quality data, deliver better performance, and process more simultaneous requests than those with less capacity. However the heterogeneity can also be exploited to trade off performance and quality of data for users with different requirements [1, 20]. Better performance and higher quality of data are often competing goals. Generating data with higher quality (e.g. images with better resolution or data with higher accuracy) can result in higher latency. Trading off between them can satisfy users with different requirements as well as better utilize resources under different workloads. A key feature of the tiered system is that the back-end systems typically store contents of different qualities. When the system is not under heavy load and resources are plentiful, the back end will serve high quality data. When the system is heavily loaded and resources are scarce, the system can conserve resources by serving lower quality content.

More specifically, in the tiered system there is a primary database server with the highest quality content. The primary server satisfies requests by making queries which may have high overhead to a database. Clients obtaining responses from the primary database get information which is both the most current and has the greatest level of detail.

A secondary back-end server might contain information which is less current or less detailed than the information provided by the primary server. For example, the secondary server might be a cache for the information contained in the primary server. It may be updated less frequently than the primary server. However, data would be accessed from the secondary server with considerably less overhead than would be required for accessing the data from the primary server.

The secondary server might also provide less detailed information than the primary server by using lighter weight queries, for example. In situations where I/O bandwidth is limited, the primary server might contain high resolu-

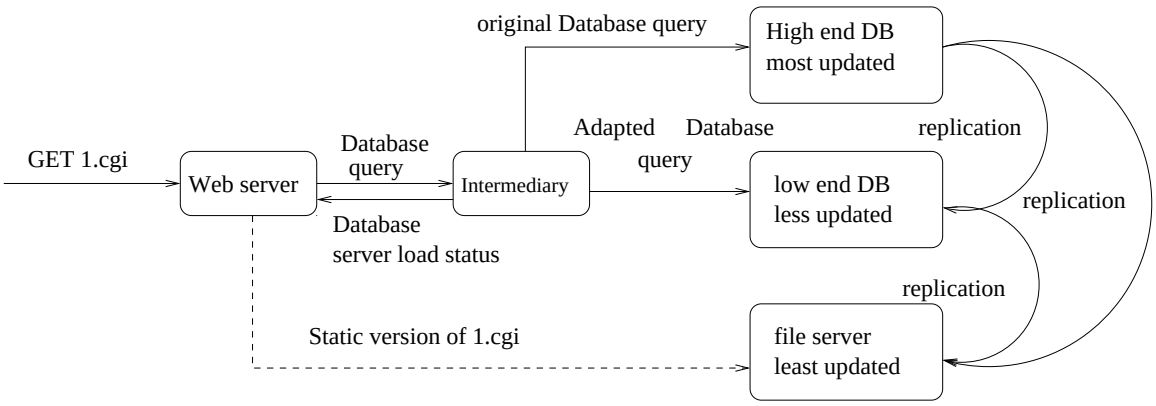

Figure 1. Architecture

tion images while the secondary server might contain lower resolution images which consume fewer bytes.

As illustrated in Figure 1, this architecture is composed of Web servers, an intelligent routing intermediary, and tiered back-end servers.

The high-end database server stores the most updated information with the highest level of quality. The update process could be initiated by HTTP clients or external programs. The response time could be adversely affected if the high-end database server is overwhelmed with too many client requests. Therefore, the high-end server only satisfies a fraction of the requests during periods of heavy loads.

The low-end server (periodically) replicates content from the high-end server. It contains less frequently updated or lower fidelity information. It starts serving requests when the traffic is beyond the high-end server's capacity, requests are from less favored clients, or the contents in the low-end server are sufficient for satisfying the clients' requirements (such as image resolution, document size, etc.). The low-end database server could be different from the high-end one to reduce the cost with different query types. For instance, the high-end server could run DB2 [10] and the low-end one could run MySQL [19]. The secondary database only serves selected queries to exploit MySQL's limitations in this type of operation. DB2 DataPropogator/DataJoiner could be used to propagate the data to MySQL. Naturally, it is necessary to resolve the incompatibility that may arise between the two systems.

The file server contains the least updated contents that are generated from the database and dynamic applications to offload the excessive traffic to the database servers. It requires the least overhead for satisfying requests. Therefore, if the databases are overloaded, excess requests can be handled from the file server.

The intermediary's purpose is to identify the nature of the requests (the clients' priority and capacity), detect each database server's load, and direct the requests to the appropriate servers. The replication processes are content and load status aware. This architecture offers a good balance between performance and cost. Our architecture can handle more than three levels in the hierarchy if necessary with multiple servers at each level.

We have prototyped the architecture and demonstrated its improvement in several aspects. Our implementation focused on the methodology rather than on specific applications. The prototype can be adapted to domain-specific applications with their own valuation of data quality and constraints of tiered data replication.

The rest of the paper is organized in the following way. Section 2 discusses different instantiations of the tiered service. Section 3 outlines the proposed architecture. The implementation details and test bed set-up are presented in Sections 4 and 5 respectively with the experimental results in Section 6. These results demonstrate that the tiered database architecture significantly improves the overall performance in terms of scalability, fairness, and QoS provisioning. Related work is discussed in Section 7 followed by concluding remarks in Section 8.

2. Tiered Service

Based on data quality and system configuration, there are three approaches to realize the tiered services: quality differentiation, data partition, and query type differentiation.

In the quality differentiation approach, the back ends store contents with different overheads. The high-end stores the high quality contents while the lower-end contains data with lower overheads. The low-end's performance in processing the low overhead data can match or even exceed that of the high-end. In the data partition approach, the data in all of the back ends have the same overhead, but some back ends do not fully replicate the whole data set. In the third approach, back ends are optimized in different ways to serve specific query types.

The quality differentiation approach is applicable in the following types of environments:

– The high-ends store more detailed documents and high resolution images and the low-ends store the stripped down versions. Requests are routed to the high-ends when the traffic is light. Under heavy load, premium clients are still routed to the high-ends while the others are served by the low-ends. Client profiling can be based on their subscription (e.g. high paying customers enjoy better quality of data than low paying ones) or on client characteristics such as their devices' capacity. For instance,

those who use handheld devices are served with low resolution images that match the devices' rendering capability.

– In a streaming query service, the contents are frequently updated, and the freshness of the data determines the quality of service. The high-ends store the most updated data, and the low-ends periodically replicate content from the high-ends. When the traffic is high, the low paying customers are served with contents from low-ends to prevent the high-ends from becoming overloaded.

In the data partition approach, data sets are split and assigned to different back ends. For instance, in an e-commerce Web site, one back end stores the inventory data and the other stores the customer information. Queries that operate on different data sources are routed to the appropriate back ends. This approach enables efficient data caching and retrieval and benefits disk and memory-intensive operations. However, its effectiveness is contingent on the data dependency. If the operations frequently use both inventory and customer data simultaneously, then multiple connections are needed to retrieve the data from different back ends; as a result, performance will eventually suffer.

Applications using query type differentiation are as follows. The back ends are of different system types, some of which perform better on particular operations than others. The routing module exploits this difference to gain both in performance and cost. For instance, simple database implementations like MySQL excel in query types such as *select*. Hence some back ends can be customized to serve these type of queries. It is also beneficial to tailor the hardware configuration to expedite the execution of specific operations. For instance, more RAM can substantially expedite *join* operations. More powerful CPUs can accelerate the execution of computationally intensive queries. This approach should take into account the data layout to improve data locality.

In this paper, we propose a general architecture and implementation alternatives to realize the tiered service. Applications can adopt different policies on top of it. However, since the first two approaches (as defined in the beginning of this section) are largely application dependent, their effectiveness varies in different circumstances. The third approach can be adopted in a wide range of applications and is complementary to the other two. It is discussed in greater detail in this paper. Since databases are so widely used for back-end systems, we use them to instantiate the tiered architecture. The following sections describe the implementation of tiered database service (TDS). Note that the tiered service approach can also be applied to other back-end systems which do not use databases.

Our tiered database service (TDS) has the following properties:

– Transaction integrity. The routing decision is made at the URL level. Once a database is selected, all queries corresponding to a URL are routed to the same database.

– Layer-7 routing. Unlike the layer 3 and 4 load balancing algorithms that are used for front-end Web servers [6], TDS routing algorithms are in the application layer (ISO layer-7). They are aware of the query semantics and the database table layout.

– Query syntax variation resolution. Different databases have their own SQL variations. TDS can defer the query materialization till the actual database is selected.

– Data residency awareness. If the back ends are not full replicas, the routing algorithms should be aware of which back ends store which data. TDS can be explicitly configured to be aware of this or learn it based on feedback from the back ends.

– Adaptive configuration. TDS collects run-time statistics to detect performance differences and dynamically vary routing policies.

3. Tiered Database Service Architecture

We propose an implementation paradigm as illustrated in Figure 2. There are three components in TDS: a front-end connector, query adapter, and routing intermediary.

3.1. Front-end Connector

The front-end connector (FC) obtains the client information from the HTTP requests. Such information includes the identity of the client that determines its priority and the browser type that indicates the rendering capability. The client profile is used when the routing intermediary uses class-based service differentiation policies. The browser type information is used to select the back end whose data quality matches the browser's rendering capability. The FC also informs the Web applications of the load status of both the front-end and back-end servers so that the applications can rewrite some of the requests to retrieve the static version of the requested application or return to the client an alert message that indicates the system is under high load. In this regard, FC shares some similarity with content adaptation [1].

In practice, the FC can be implemented in the ISO layer-7 load balancer or be incorporated into the caller function that invokes the Web applications.

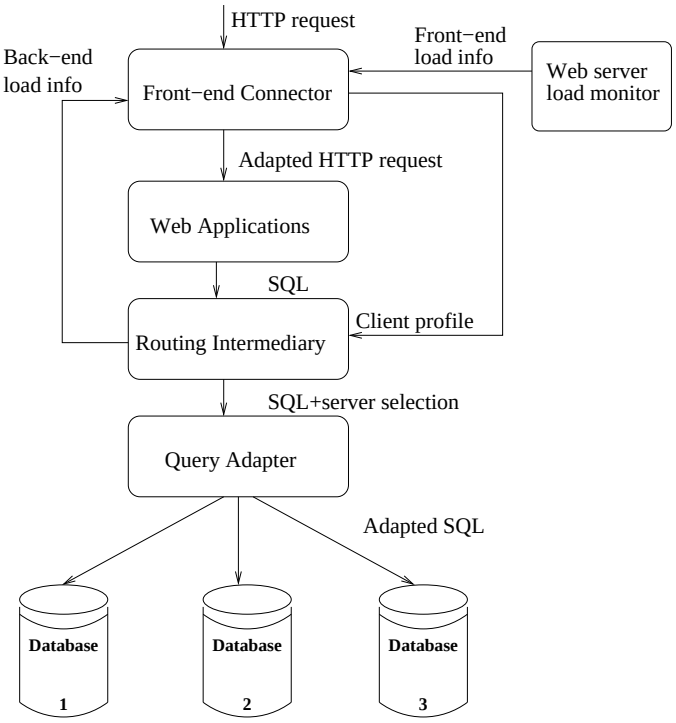

Figure 2. Tiered Database Architecture

3.2. ROUTING INTERMEDIARY

The routing intermediary routes queries to back-end servers based on specific policies. It monitors the traffic status and intelligently routes the queries based on routing policies, client profiles, traffic, and the back-end configurations. The routing polices are the instantiations of the tiered service variants in Section 2.

3.3. QUERY ADAPTER

There are many platform-dependent SQL schemas in different database servers. Therefore, it is necessary to provide appropriate SQLs to heterogeneous DBs. The query adapter (QA) component resolves this problem. There are two solutions for implementing a QA: deferred query materialization and query rewriting. Deferred query materialization provides an abstract layer for query languages. Instead of directly using SQL queries, the application logic uses QA APIs that are independent of the back-end database implementations to construct the query logic and then translate it into real platform-dependent database queries. Consequently, the application logic need not be aware of the actual database that it interacts with, and the programmers can be more

focused on the business logic itself. Examples of this kind include Sun's Java Data Object (JDO) specifications [18] and the sqlrelay project [25]. The drawback of this method is that existing applications have to be rewritten to adopt the tiered services.

Alternatively, a query can also be rewritten before being directed to the tiered service to resolve the syntactical differences; thus the overhead of modifying existing applications is amortized.

In addition to the syntactical differences, QAs are also responsible for resolving the semantic variations between the tiered back-end servers. One way to resolve the variations is to provide uniform interfaces using extensible data management (XDM) as proposed in [8]. Such mechanisms can be incorporated into our modules to resolve semantic inconsistencies among the back ends. This is an application-dependent problem which is more appropriately addressed in the application domain and is beyond the scope of this paper.

3.4. Implementation Alternatives

There are two implementation alternatives based on different combinations of the three components, namely centralized mode and distributed mode. In the centralized mode, the routing decision is made by a centralized routing intermediary which has knowledge about the traffic, system load, and configuration differences between the back ends. All the Web applications communicate with the routing intermediary to get routing instructions.

In the distributed mode, the routing decision is not made by a centralized entity. Each Web application routes queries to the appropriate back ends based on the local estimation of the load distribution in the back ends. The Web applications periodically converge their individual routing policies and revise them if necessary to keep the policies consistent.

3.4.1. *Centralized Mode*

Figure 3 illustrates the implementation. The Web applications use a special API to communicate with the routing intermediary. The intermediary is responsible for server selection, query transmission, and result relay to the Web applications.

The advantage is that the routing intermediary has a global view of the traffic to the back-end servers. It can thus ensure that routing decisions are consistent. However, the intermediary may become a bottleneck when traffic is heavy. This mode also requires changes to existing applications.

3.4.2. *Distributed Mode*

Figure 4 illustrates the distributed routing mode. The Web applications make their own routing decisions locally and periodically communicate with an arbitrator. The arbitrator merges the global routing and traffic information

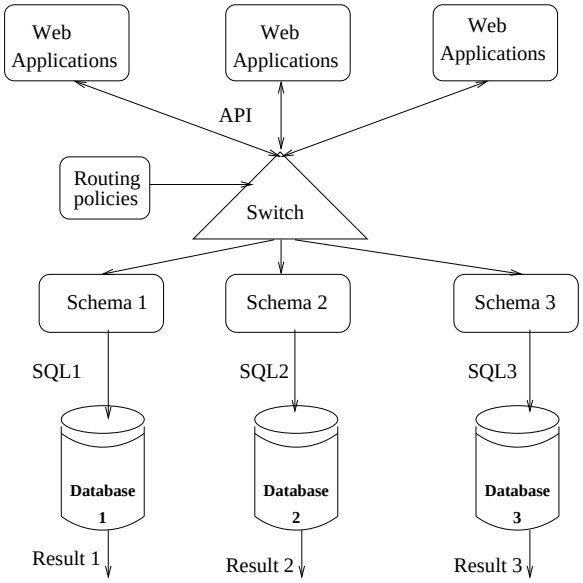

Figure 3. Centralized Mode

and instructs the Web applications to adjust their routing strategies. This mode distributes the routing functionality and is thus more scalable than the centralized mode.

4. Implementation

A prototype of the proposed architecture was implemented based on Tomcat [17]. It can be easily deployed in other Web systems that support Java servlets. As depicted in Figure 5, the implementation consists of the invoker servlet that is responsible for identifying the HTTP clients, a virtual JDBC driver that intercepts and routes database queries, and a QoS policy coordinator that maintains the consistency of QoS policies across clustered Web servers. These modules interact among themselves with little modification to existing applications. The architecture can thus can be easily applied to production systems. The functionalities of these modules are described in the following sections.

4.1. Invoker Servlet

The invoker servlet is invoked by the Tomcat servlet container and obtains the HTTP requests. The QoS identity of the client is enveloped either within the HTTP request or in a cookie. The invoker servlet parses the HTTP query, decodes and stores the QoS identity, and passes the request to the appropriate

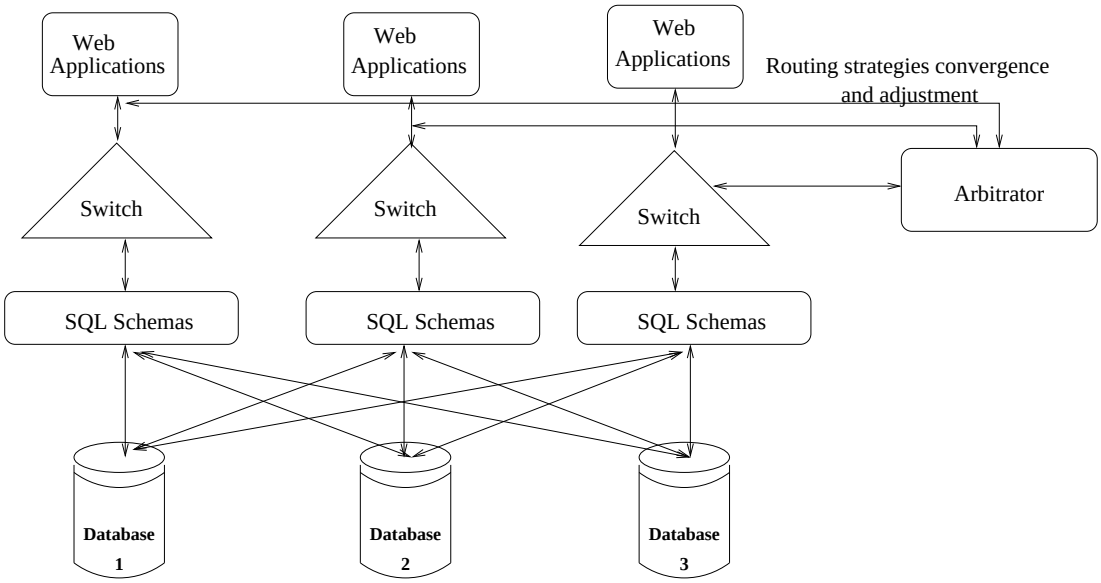

Figure 4. Distributed mode implementation

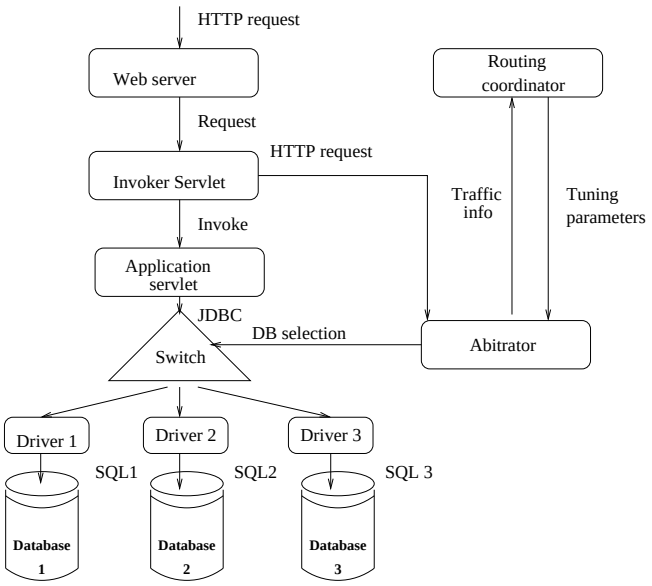

Figure 5. Prototype implementation

servlet to execute the application logic. After the application servlet finishes the processing, the invoker servlet proceeds to do some post-processing tasks including updating the processing and database access time of this HTTP request and sending this information to the QoS policy coordinator to keep the QoS policy consistent.

4.2. VIRTUAL JDBC DRIVER

The virtual JDBC driver is invoked when the application servlet initiates database access, usually by calling the function *connect()*. Upon calling *connect()*, the virtual driver determines which database server the application should interact with. We have implemented the following routing policies in the test-bed.

– Data affinity

If the application needs to access database tables that do not reside in the secondary database server, the application uses the primary one. This policy applies to situations where data are not fully replicated between primary and secondary database servers. The virtual driver is configured to understand the data residency in the servers.

– Data consistency

If the application is known to include data insert, update, or deletion operations, the virtual driver selects the primary server as its data source to keep the primary server updated. The virtual driver can also be programmed to schedule such queries to update the secondary database upon completion of the application.

– Query complexity

If the application contains queries that are not well handled in one database server such that the processing of these queries would incur significant latency as compared to the other server, this application will use the primary server. Examples of these operations include join and recursive select. Detection of this situation does not require analyzing the application code. It can be identified by comparing the previous database access time from the two database servers.

– QoS policy

In the QoS-based policy, the database selection is a function of the traffic composition and the priority of the request as described later in this section.

– Database load

In the load balancing policy, if the primary database server is projected to be highly loaded, the following requests use the secondary one. The load is detected using the following scheme.

A common way to detect server load is use response time variation. However, because the access time of a single database query can vary significantly, it is not a reliable indicator of server load. The load detection techniques used in our experiments are based on the variation of the ratio of the total database access time per URL and the associated HTTP request processing time. The database access time is determined by the size of the result set, the query complexity, and the degree of query concurrency, which is highly volatile. However, the corresponding servlet needs to process the result set and construct the query command, the complexity of which is approximately linear with that in accessing the database server. Thus, when the data set does not vary significantly, the ratio can signify which server is overloaded; a higher ratio that results from long database access times means that the database is more loaded; conversely, a lower ratio means that the Web server is less loaded.

More specifically, the database server selection algorithm for those URLs that have no data update operations is as follows. If the previous accesses reveal that a URL is database intensive (characterized by the total database access time), then upon completion, the ratio of its total database access time and the HTTP processing time is evaluated and compared to the previous values. If their difference is beyond a certain threshold, the switch is then instructed to route some traffic to the secondary database server. Otherwise, current traffic load is considered acceptable for the primary database's capacity, and the current traffic rate and the ratio value are recorded for future reference.

The load detection algorithm determines the maximum number of simultaneous requests *Max* that use the primary database server. It is formally described as the following:

$$Max = \begin{cases} a * Max^{'} + b * Rate, \; r > (1+\Delta) * \widetilde{r} \\ Max + 1, \; otherwise \end{cases} \tag{1}$$

where positive real numbers a and b are tuning parameters and $a+b=1$. A higher value of a leads to a more gradual decrease in the traffic to the primary server, while a lower value is more effective in preventing the primary server from being overloaded. Their values were both set at 0.5 in the experimental evaluation. The rationale is that the primary server is more powerful than the secondary one. The performance the primary server delivers under increased load is still comparable with that from the secondary server for some URLs. Note that the traffic rate is not the sole factor that determines the load at the back-end servers. Other issues, such as the traffic composition and the server's implementation, also play an important role. Incorporation of these

factors, however, leaves the routing algorithm complicated and reliant on the system configuration, thus resulting in administrative overhead.

Max' is the previous value of the maximum number of simultaneous requests.

Rate is the current traffic rate to the primary server.

r is the ratio of the database access time and the HTTP processing time of the monitored query, and $\widetilde{r}$ is its average value.

Δ is the variation threshold that controls how much variation of r is considered normal. Variation beyond that is regarded as occurrence of overload.

The ideas behind Equation 1 share certain similarities with those behind calculation of round trip times in the TCP protocol [14]. The current maximum degree of simultaneous connections is determined by its historical values and current traffic rate. The tuning parameters a and b determine how much each component affects the value of Max; larger a and smaller b make Max less subject to traffic variations and vice versa. If the ratio r is within the variation scope, the primary server is considered to be underutilized such that additional traffic can be handled without penalty. Once r is beyond its variation scope, the current traffic rate and the historical value of Max approximate the capacity of the primary server.

The advantage of this scheme is that it requires little modification to current applications and is adaptable for various servlet applications.

The algorithm is formulated in Algorithm 1. The function *DB_select* is invoked when the servlet establishes a connection to access the database. It returns a database connection. Function *Postprocessing* is invoked when the servlet finishes. It collects all the runtime statistics and recomputes the global variables that are used to select the database connections. The complexity of the algorithm is linear with the number of URL's that are tracked, and most of the operations are string matching. The routing overhead is relatively low.

4.3. QoS Policy Coordinator

The QoS policy coordinator is a long-running process. It receives the traffic composition and database selection information from all the Web servers and checks if the collective effect of database scheduling is consistent. If necessary, the QoS policy coordinator notifies Web servers to change their local decision function parameters.

We use two factors to route the database queries: request priority and the traffic composition from all classes. The algorithm description is the following.

For a request r that belongs to class i, the boolean variable S_i represents whether r can access the primary database server.

Algorithm 1 Load Balancing Algorithm

```
FACILITIES:
boolean contain_update(url): boolean function that tells whether the given url
contains insert/deletel/update queries.
long db_processing_time(url, db): function that returns the processing time
(ms) of the given url at the database db.
float ratio(url): function that return the URL's ratio of database access time to
the HTTP processing time.
R: current traffic rate to the primary server.
delta: threshold of ratio variation.
Max: maximum concurrent connections to the primary server.
Conn DB_select(url)
{
if (contain_update(url))
  return primary; /* update queries are routed to the primary server */
if (R>Max)
  return secondary; /* if the traffic to the primary server is high, the secondary
starts to function */
if (db_processing_time (url, primary) > db_processing_time (url, secondary))
  return secondary; /* the url is served by the server that can best handle the
queries */
}
Postprocessing(url)
{
compute and store the average database access time of the URL at the selected
database db.
compute the ratio of the database access time to the HTTP processing time,
denoted as r.
if the URL is served by the secondary database, then exit.
if (r > ratio(url)*(1+delta))
  Max = a*Max+b*R;
else
   Max ++;
}
```

$$S_i = 1(\frac{\lambda_i * W_i * Max}{\sum_j \lambda_j * W_j} > C_i), \tag{2}$$

where

$1()$ is the boolean function.

λ_i is the number of requests which have arrived since time point t that belong to class i.

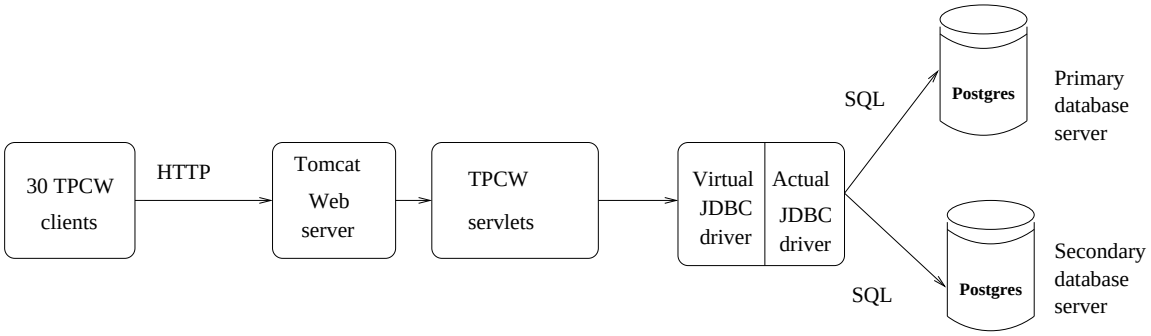

Figure 6. Test bed

W_i is the priority weight.

Max is the maximum number of simultaneous requests that the primary database can accept.

C_i is the number of requests in class i that have used the primary server since t.

It is inferred from the equation that in a clustered Web server environment where each server makes database selections based on its own traffic conditions, the request routing as a whole across all of the Web servers may be incorrect. Therefore it is necessary to set up an agent that mediates the distributed selection algorithms to approximate the ideal value.

The usage of the primary database at Web server p, denoted as N^p, during time frame T is:

$$N^p = \frac{\lambda_i^p * W_i^p * Max^p}{\sum_j \lambda_j^p * W_j^p}, \tag{3}$$

where λ_i^p and Max^p are the local observed value at Web server p. W_i^p is the local priority weight value used by this server.

The ideal collective usage of the primary database server, denoted as N^{ideal}, from all the Web servers is:

$$N^{ideal} = \frac{\sum_p \lambda_i^p * W_i * \sum_p Max^p}{\sum_j (\sum_p \lambda_j^p * W_j)}. \tag{4}$$

In most cases, $W_j = W_j^p$ cannot ensure $N^{ideal} = \sum_p N^p$. Thus it is necessary to adjust the local priority weight vector W^p to approximate the ideal case.

5. Experimental Setup

Figure 6 illustrates the system that was used to generate experimental results. The two database servers have different hardware capacity and database

table configurations. The purpose is to investigate how to maximize performance on the heterogeneous systems. The primary server is a high-end one but its database tables are not indexed. It thus incurs longer latencies for search-intensive queries, i.e. URL 2, 4, and 5 in Table I. The secondary server has lower hardware capacity but uses indexed database tables. As a result, the secondary server performs well on search-intensive queries. We use this configuration distinction to simulate the following situations: (i) back ends have different levels of overheads, and the low-end stores data with lower overhead to trade off between performance and quality. (ii) heterogeneous back-end systems with different processing optimization, e.g. some are optimized for search and the others are optimized for update. The experimental results demonstrate that our routing algorithm can identify and exploit this optimization distinction. The benchmark suite used is the Java-based TPC-W [5].

TPC-W [9] models transactions of an online bookstore. It stresses a Web server and database server system with representative e-commerce workloads. The benchmark client generates multiple concurrent sessions with remote browser emulators (RBE) for both dynamic and static contents. The dynamic contents are generated by referencing and updating a database with many tables with different sizes and attributes that contain inventory, purchasing, and shipping information.

The RBEs can be configured by setting parameters to generate different workloads. These parameters include thinking time, warm-up, and cool-down time. We ran stress tests with no thinking time, i.e. no pause between consecutive requests that are from the same RBE.

Table I lists the servlets' execution time complexity as a function of the database table size (n). These servlets are database driven and frequently referenced. Servlet TPCW_best_sellers_servlet contains a complex query that involves join operations over several database tables and query criteria. Therefore it takes significant processing time in the PostgreSQL database [22]. The database complexity is not programmed into our prototype. Instead, our back-end server selection algorithm is based on monitoring the database access time variations. This approach is more independent of system and application configuration and thus helps to reduce administrative overhead. Table II lists the server configurations in this test bed. Table III presents the database reference patterns and the traffic composition in the single database server configuration. In each test, the database server serves queries from 30 RBEs.

Table I. Servlet execution time complexity as a function of database table size (n)

URL	Servlet	execution time
1	TPCW_best_sellers_servlet	$O(n^3)$
2	TPCW_execute_search	$O(n^2)$
3	TPCW_home_interaction	$O(n)$
4	TPCW_new_products_servlet	$O(n^2)$
5	TPCW_product_detail_servlet	$O(n^2)$
6	TPCW_search_request_servlet	$O(n)$

Table II. Server configuration

Server	CPU	Memory
Web server	PIII 733MHZ	256M
Primary database server	PIV 1.8GHZ	512M
Secondary database server	PIII 500MHZ	128M

Table III. DB server access characteristics.

	DB Access time (sec)		
URL	Primary	Secondary	Traffic Proportion
1	26.24	49.33	11%
2	3.16	2.86	11%
3	1.35	1.32	30%
4	2.31	2.44	11%
5	1.63	1.75	21%
6	1.44	1.54	12%

6. Experimental Results

6.1. Workload Description

We benchmarked the test bed under different workloads. The number of RBEs ranged from 20 to 60. Two static traffic partition settings were used along with the load balancing routing to compare the performance and fairness. Under static partitioning, traffic is statically routed in fixed proportions to the back ends. We have tested different static settings and selected the results that represent two different service strategies: *under- and over-utilization* of the primary DB. Traffic ratio 2:1 (66% of the traffic to the primary server, underutilization) allocates less traffic to the primary server. Thus the primary server can deliver better service for these requests. But the secondary DB is overwhelmed by the traffic with increasing intensity and thus delivers poor service. This scenario may be used to provide differentiated service based on server affinity. Traffic ratio 5:1 (83% to the primary server, over-utilization) represents the intention to deliver uniform service to all the requests by dragging down the primary server's performance and improving the service from the secondary one. The performance of the single database server architecture is also measured and compared against the different clustering schemes

6.2. Results

We compare different routing schemes and strategies in light of performance (latency and throughput), skewness, and the ability to exploit heterogeneous optimizations in the back ends. Throughput is the number of completed HTTP requests per second. Latency is the average accrued database access time. Skewness is to evaluate the service's fairness. It is quantified by using the ratio of the latency difference from the two back ends. During each test, these performance metrics were collected under various workloads.

– Latency

Figure 7 plots the latency with respect to various workloads under different routing policies. The curves represent the average latency obtained by using such routing policies as load balancing, static over- and under-utilization routing, and the single database server (the primary server). It is observed that the load balancing routing policy can achieve better performance with lower latency than the static routing policies and single database server. In Figure 7, it is observed that when the workload is light, the over-utilization policy achieves lower latency than the under-utilization policy because the primary database server is not highly loaded and performs better than the secondary server. When the workload increases, the load at the primary server increases

at a faster pace than the secondary one and eventually overcomes their capacity difference. As a result, the primary server's performance deteriorates more sharply under the over-utilization routing policy. The load balancing routing policy takes into account the performance difference between the two database servers. It is able to detect the performance differences under varying workloads and use the appropriate server to serve the queries. Hence it utilizes the system more efficiently. As a result, the latency is smoother and lower than the static policies.

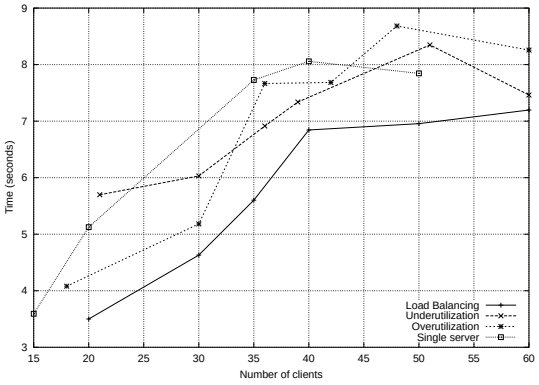

Figure 7. Latency. Traffic ratios between primary and secondary servers are 2:1 (underutilization) and 5:1 (over-utilization).

– Throughput

Figure 8 presents the throughput results. The system is highly loaded with heavy weight database queries so the throughput is relatively low. The throughput is determined by the traffic intensity and the service rate. When the workload is low, throughput is mostly determined by the traffic rate because the servers have enough capacity to accommodate the workload. When the workload increases, the throughput grows at a slower pace. At this stage, the throughput is determined mostly by the processing time because the servers are saturated with the traffic. Static routing policies deliver lower throughput because they cannot accurately determine how much traffic should go to each server under different workloads.

– Skewness

Skewness is defined as follows.

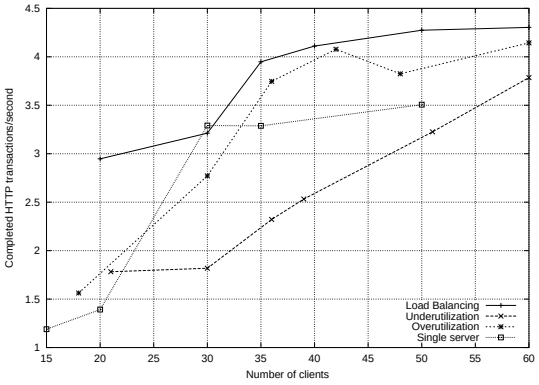

Figure 8. Throughput

Definition *Skewness*: The skewness between the response time from back ends A and B is $\frac{|r_A - r_B|}{max(r_A, r_B)}$, where r_A and r_B are the response time from A and B respectively.

Skewness is used to evaluate the fairness of the routing schemes. Higher skewness means that services from different back ends diverge significantly. Figure 9 plots the skewness of different routing schemes. As discussed earlier, over-utilization of the primary server offers more balanced service while the under-utilization scheme is more biased toward the primary server. Under low workloads, load balancing exhibits more fairness than static routings. When the workload increases, the skewness in all three settings approaches zero. That is because both the primary and secondary servers are overloaded, and the performance gap between the two servers narrows.

– Ability to exploit heterogeneous optimizations

Our system can exploit heterogeneous optimizations. Figures 10 and 11 show the breakdown of database access time and number of accesses when 60 RBEs are used. Since the routing intermediary can recognize the best server for each URL, it always tries to route the queries to the appropriate servers with lower latency. This is demonstrated by the observation that, in each URL, the worse performing server is assigned fewer requests and vice versa.

6.3. CLASS-BASED SERVICE DIFFERENTIATION

In this test, the RBE clients were assigned priorities that were enveloped in the HTTP requests. There were 3 priority levels: 0, 1, and 2. Level 0 is the highest. Figures 12 and 13 plot the cumulated number of accesses to each database

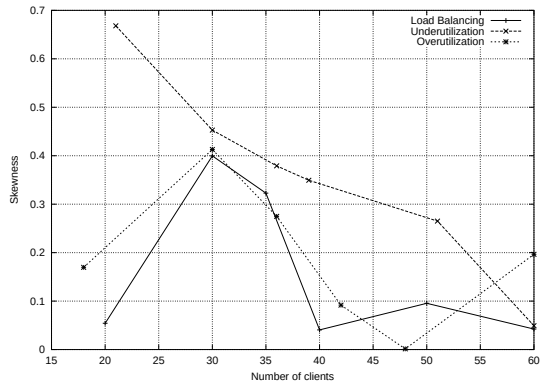

Figure 9. Skewness

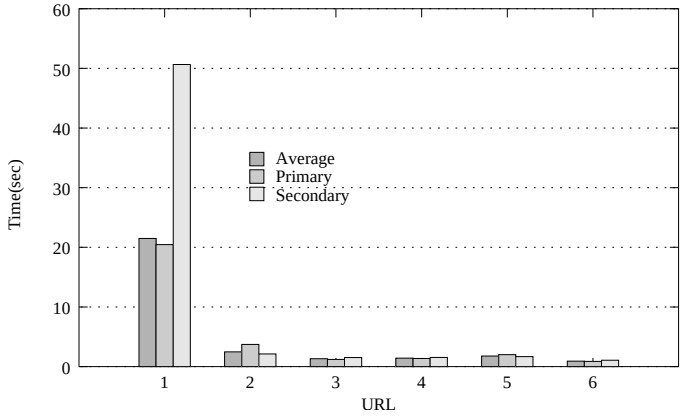

Figure 10. Database access times

server. It is observed from the figures that the higher priority classes have more chances to access the primary server; they thus receive better service.

Figure 14 plots the latency of each QoS class under various workloads. Classes with higher priority consistently receive better performance. Figure 15 plots the latency of the highest QoS class under various traffic ratios. The latency of the QoS 0 class increases with respect to the load.

7. Related Work

Load balancing in computationally intensive distributed applications is presented in [12]. A distributed industrial web clustering architecture is presented in [16], which provides distributed transactional service. It uses tiered

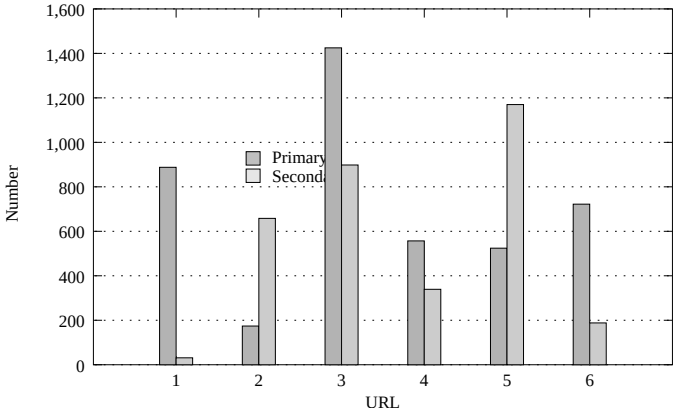

Figure 11. Number of database accesses

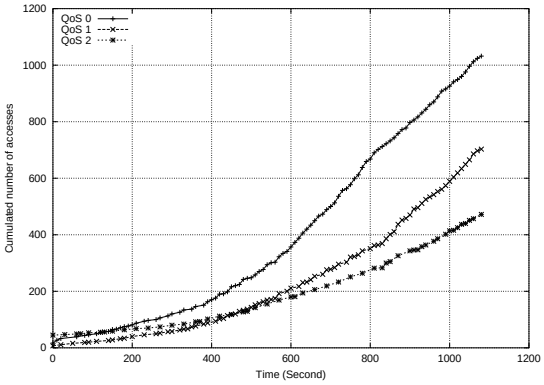

Figure 12. Primary database server

Servlet engines to provide fail-over when the primary server is not reachable. Algorithms for managing workloads to back-end systems in clustered environments are discussed in [3]. This work does not discuss improving performance with different qualities of data in back ends.

There has been considerable work in the area of load balancing to front-end Web servers [6, 7, 15]. Many front-end load balancers will route requests at layer 4 of the OSI protocol stack which means that the load balancer is not aware of the contents of the request [11, 13]. Layer 7 routing in which the load balancer is aware of the contents of the request has also been studied [21, 4]. While content-aware routing increases the functionality of the load balancer, it also adds significantly to the overhead resulting in the load balancer becoming a bottleneck at lower throughput levels [23, 24].

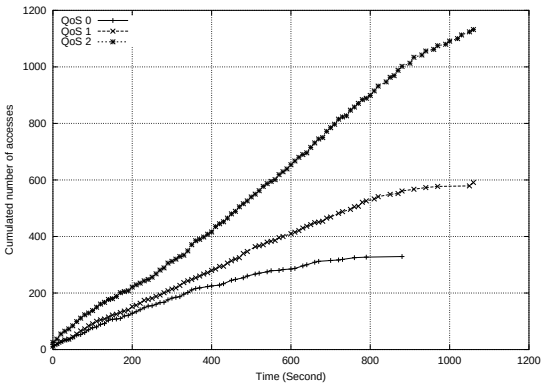

Figure 13. Secondary database server

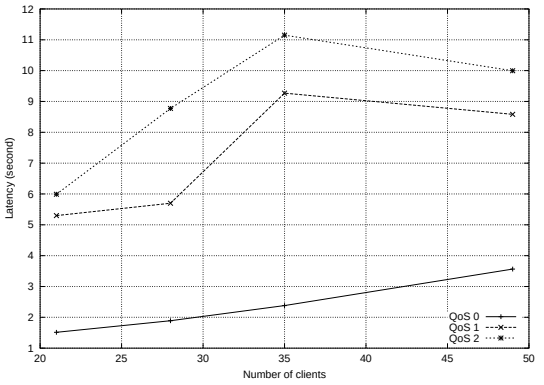

Figure 14. Latency of each class. The ratios of the number of requests corresponding to QoS classes 0, 1, and 2 respectively are: 1:2:4.

A Web server architecture that features a collaborative caching system is proposed in [2]. In this architecture, the traditional file system is replaced by a distributed database system.

Overload control through content adaptation was proposed in [1]. It mainly works on the front-end Web servers with degraded versions of static content. Our work targets dynamic Web applications that use back-end systems.

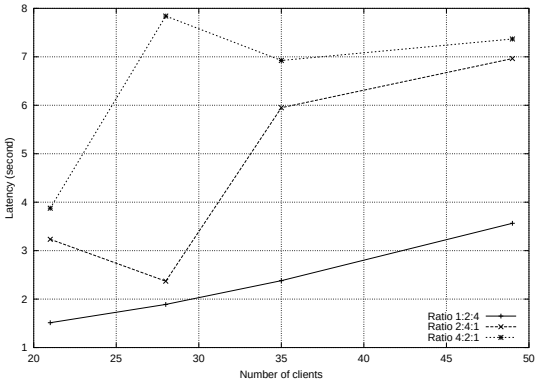

Figure 15. Latency for QoS 0 under various traffic ratios

8. Conclusion

In this paper, we proposed a tiered back-end architecture for serving dynamic Web data. Our architecture supports heterogeneous back ends with different qualities of data. The routing intermediary that resides between the front and back ends intelligently forwards requests to appropriate back ends based on certain policies. We described how we implemented our system. Performance results demonstrate significant improvement over traditional architectures in both response time and throughput.

One area of future work is to implement specific mechanisms to evaluate trade-offs between satisfaction and latency. It is largely a psychological problem with significant application dependencies. Satisfaction can be fulfilled with either better content or shorter latency. For a specific application, it is necessary to quantify what the best combination of these two factors is and implement such quantification in the routing policies.

Acknowledgments

Isabelle Rouvellou provided important management support for this work. Thomas Mikalsen and Lakshmish Ramaswamy provided helpful comments on an earlier draft of this paper.

References

1. Abdelzaher, T. F. and N. Bhatti: 1999, 'Web Content Adaptation to Improve Server Overload Behavior'. In: *Proceedings of the 8th International World Wide Web Conference, Toronto, Canada*.
2. A.Belloum, E. Kaletas, H. Afsarmanash, and L. Hertzberger: 2002, 'A Scalable Server Architecture'. *World Wide Web: Internet and Web Information Systems* **5**, 5–23.
3. Aman, J., C. Ellert, D. Emmes, P. Yocom, and D. Dillenberger: 1997, 'Adaptive Algorithms for Managing a Distributed Data Processing Workload'. *IBM Systems Journal* **36**(2), 242–283.
4. Aron, M., D. Sanders, P. Druschel, and W. Zwaenepoel: 2000, 'Scalable Content-Aware Request Distribution in Cluster-based Network Servers'. In: *Proceedings of the 2000 USENIX Technical Conference*. pp. 90–101.
5. Bezenek, T. et al., 'TPC-W in Java'. http://www.ece.wisc.edu/ pharm/tpcw.shtml.
6. Cardellini, V., E. Casalicchio, M. Colajanni, and P. Yu: 2002, 'The Sate of the Art in Locally Distributed Web-Server Systems'. *ACM Computing Surveys* **34**(2), 263–311.
7. Cardellini, V., M. Colajanni, and P. Yu: 1999, 'Dynamic Load Balancing on Web-Server Systems'. *IEEE Internet Computing* pp. 28–39.
8. Cooper, B., N. Sample, M. Franklin, J. Olshansky, and M. Shadmon: 2001, 'Middle-Tier Extensible Data Management'. *World Wide Web: Internet and Web Information Systems* **4**(3), 209–230.
9. Council, T. P. P., 'TPC-W'. http://www.tpc.org/tpcw/default.asp.
10. DB2, I., 'DB2 Database'. http://www.ibm.com/software/data/db2/.
11. Dias, D., W. Kish, R. Mukherjee, and R. Tewari: 1996, 'A Scalable and Highly Available Web Server'. In: *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*.
12. Eager, D. L., E. D. Lazowska, and J. Zahorjan: 1986, 'Adaptive Load Sharing in Homogeneous Distributed Systems'. *IEEE Transactions on Software Engineering* **12**(4), 662–675.
13. Hunt, G., G. Goldszmidt, R. King, and R. Mukherjee: 1998, 'Network Dispatcher: A Connection Router for Scalable Internet Services'. In: *Proceedings of the 7th International World Wide Web Conference*.
14. IETF, 'Transmission Control Protocol'. http://www.ietf.org/rfc/rfc0793.txt.
15. Iyengar, A., E. Nahum, A. Shaikh, and R. Tewari: 2002, 'Enhancing Web Performance'. In: *Proceedings of the 2002 IFIP World Computer Congress (Communication Systems: State of the Art)*.
16. Jacobs, D.: 2003, 'Distributed Computing with BEA WebLogic Server'. In: *First Biennial Conference on Innovative Data Systems Research (CIDR)*.
17. Jakarta, 'Tomcat'. http://jakarta.apache.org/tomcat/.
18. Micro., S., 'Java Data Object'. http://http://access1.sun.com/jdo/.
19. MySQL, 'MySQL Database'. http://www.mysql.com.
20. Olston, C. and J. Widom: 2000, 'Offering a Precision-Performance Tradeoff for Aggregation Queries over Replicated Data'. In: *Twenty-Sixth International Conference on Very Large Data Bases (VLDB), Cairo, Egypt, pp. 144-155.*
21. Pai, V. et al.: 1998, 'Locality-Aware Request Distribution in Cluster-based Network Services'. In: *Proceedings of ASPLOS-VIII*.
22. PostgreSQL, 'PostgreSQL Database'. http://www.postgresql.org.
23. Song, J., A. Iyengar, E. Levy, and D. Dias: 2000, 'Design Alternatives for Scalable Web Server Accelerators'. In: *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*.
24. Song, J., A. Iyengar, E. Levy, and D. Dias: 2002, 'Architecture of a Web Server Accelerator'. *Computer Networks* **38**(1).

25. Sqlrelay, ‘Sqlrelay’. http://www.firstworks.com/sqlrelay.html.