ReDAL: An Efficient and Practical Request Distribution Technique for Application Server Clusters

Kaushik Dutta, *Member*, *IEEE*, Anindya Datta, *Member*, *IEEE*, Debra VanderMeer, *Member*, *IEEE*, Helen Thomas, *Member*, *IEEE*, and Krithi Ramamritham, *Fellow*, *IEEE*

Abstract—Modern Web-based application infrastructures are based on clustered multitiered architectures, where request distribution occurs in two sequential stages: over a cluster of Web servers and over a cluster of application servers. Much work has focused on strategies for distributing requests across a Web server cluster in order to improve the overall throughput across the cluster. The strategies applied at the application layer are the same as those at the Web server layer because it is assumed that they transfer directly. In this paper, we argue that the problem of distributing requests across an application server cluster is fundamentally different from the Web server request distribution problem due to core differences in request processing in Web and application servers. We devise an approach for distributing requests across a cluster of application servers such that the overall system throughput is enhanced, and load across the application servers is balanced.

Index Terms—Distributed systems, performance evaluation, Web-based services, client/server and multitier systems, electronic commerce, middleware/business logic.

1 Introduction

Request distribution in clustered environments is an important problem that has been studied in a number of different contexts. In this paper, we are interested in developing effective techniques for distributing requests to a cluster of application runtimes such as the Java Virtual Machine (JVM) for Java Enterprise Edition (EE) applications and the Common Language Runtime (CLR) for Microsoft.NET applications.

Modern application infrastructures are based on clustered multitiered architectures. Fig. 1 shows a typical architecture for a Web-based application, one recommended as a "best possible" architecture [1].

In Fig. 1, there are two significant request distribution points. First, the Web switch must distribute incoming requests across a cluster of Web servers for HTTP processing. Subsequently, these requests must be distributed across the application server cluster for the execution of application logic. To distinguish between these two steps, we will refer to them as the *Web Server Request Distribution* (WSRD) problem and the *Application Server Request Distribution* (ASRD) problem, respectively. In this paper, we develop an effective ASRD technique for session-intensive applications. ASRD and WSRD differ greatly in the dynamics of work involved in serving a request (as described in [2]): Serving application requests require much more dynamic decision making than is required for Web server requests.

1.1 Related Work

Extensive literature dealing with the WSRD problem exists, and significant commercial value has been realized from this work such as Cisco's LocalDirector [3] and F5 Network's BIG-IP [4]. These approaches are variants of the *Weighted Round Robin* (WRR) approach [5].

In some commercial products [6], [7], *content-based routing* schemes are supported, which route requests based on information contained in the HTTP header. These techniques do not consider the impacts of statefulness in applications and are thus orthogonal to our work. Typically, content-based routing is used to segment requests across geographically dispersed environments, multiple application domains, etc.

The only strategies, to the best of our knowledge, that are not WRR variants are the *Locality-Aware Request Distribution* (LARD) algorithm [8] and the Client/Session Affinity schemes, both of which are based on some form of locality with respect to the servers. The LARD strategy attempts to route tasks to exploit the locality among the working sets of received requests (for example, cache sets on different Web servers), whereas the affinity-based schemes distribute requests to exploit the locality of session or state data. Elmeleegy et al. [9] consider the

- *K. Dutta and D. VanderMeer are with Decision Science and Information Systems, College of Business, Florida International University, 11200 SW 8th Street, Miami, FL 33199. E-mail: {Kaushik.Dutta, debra.vandermeer}@fiu.edu.*
- *A. Datta is with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902. E-mail: anindya@smu.edu.sg.*
- *H. Thomas is an independent consultant, 453 Alberta Court, Moab, UT 84532. E-mail: thomas.helen.m@gmail.com.*
- *K. Ramamritham is with the Department of Computer Science and Engineering, IIT Bombay, Powai Mumbai 400076. E-mail: krithi@iitb.ac.in.*

Manuscript received 26 Sept. 2005; revised 29 Dec. 2006; accepted 18 Apr. 2007; published online 7 May 2007.
Recommended for acceptance by X. Zhang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0413-0905.
Digital Object Identifier no. 10.1109/TPDS.2007.1105.

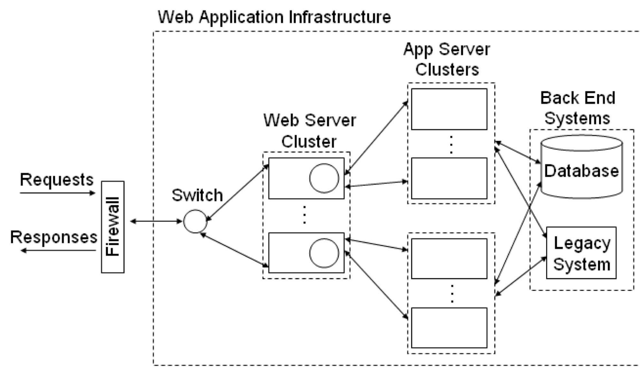


Fig. 1. Web application architecture.

extension of the LARD technique to the application server's Enterprise Java Beans (EJB) layer to take advantage of EJB data caching where possible.

The bulk of ASRD, in practice, is based on a combination of Round Robin (RR) and *Session Affinity* routing schemes drawn directly from WSRD techniques (for example, [10]). More specifically, the initial requests of sessions (for example, the login request at an airline Web site) are distributed in an RR fashion, whereas all subsequent requests are handled through *Session-Affinity-based schemes*, which route all requests in a particular session to the same application server. A user's session state, which stores information relevant to the interaction between the user and the Web site (for example, user profiles or a shopping cart), is usually stored in the process memory of the application server that served the initial request in the session and remains there while the session is active: Only the application server instance where the session resides can service requests for that session.

There is scant treatment of ASRD in the research literature. Approaching load balancing as a variant of the dynamic scheduling problem, techniques from the scheduling field (for example, [11]) may be applicable here. Some work in the literature [12] takes this approach, proposing the application of optimization techniques to the problem of providing different classes of services (for example, standard and premium services) in the context of Web services.

Although we can think of the ASRD problem as a variant of the dynamic scheduling problem at a high level (our technique will use a variant of the shortest-queue-first approach), a straightforward application is difficult. Virtually all dynamic scheduling techniques [13] presuppose some knowledge of the task (for example, duration or weight), the resource (queue sizes, service times, and so forth), or both. This assumption really does not work in our case because both the tasks and the resources are highly dynamic. Moreover, the scalability requirements of ASRD are such that any technique usable in practice must have only negligible overheads. The most direct work comparable to ours that we were able to discover is [2], in which the authors show that system resource usage is not a good indicator of load on an application. The authors suggest that a better basis for determining load might be the number of active requests on an application and propose a load balancing technique for application requests based on a "least-active-requests" routing policy. We refer to this as the HJ technique throughout the remainder of the paper. Although the authors make a good point in showing that system resource usage is not a strong basis for an ASRD technique, their load balancing technique has a significant limitation in that it is not applicable to stateful applications. Stateful session-based interactive applications form a large class of applications; for example, a login-based Web application is interactive and, therefore, stateful. Our approach considers the stateful case. To summarize, ASRD techniques, in practice, virtually always utilize WSRD policies, and there does not appear a good candidate for use in ASRD scenarios in the research literature.

1.2 Issues in Applying WSRD Strategies to the Application Layer

It is important to understand why *WSRD strategies at the ASRD layer are suboptimal and, in many cases, ineffective*. The key reason for this is that Web servers and application servers are fundamentally different entities, and therefore, the same notions of what constitutes a "loaded" server do not apply, as demonstrated in [2]. We highlight three key differences here to illustrate the reasons for this.

First, the biggest difference is in the *determinism of the work performed*. Web servers do well-defined and quantifiable work, for example, processing HTTP headers and serving static content. Application servers, on the other hand, run multilayer ad hoc programs, which might be dependent on data obtained from outside the application layer infrastructure. Thus, serving a request to an application server is significantly more complex than at the Web server layer, evidenced by the fact that the application server cluster saturates well before the Web server cluster in most dynamic applications.

The second issue is the *degree to which observing the system yields insights into its load level*. System observation is a key component of most effective WSRD policies such as WRR policies. Consider, for instance, the fact that a Web server that is running at 30 percent of CPU would be considered "lightly loaded" (compared to one running, say at 50 percent) by most WSRD policies. Although such a judgment is quite accurate in the case of a Web server, it often breaks down when applied to an application server. For instance, an application server running at 30 percent of CPU might be experiencing low CPU utilization simply because a bulk of its active threads are "blocked" (for example, waiting for database query results). In contrast, another application server in the same cluster running at 50 percent of CPU may actually be less loaded, as it might possess a greater number of free threads. Note that although we used CPU utilization as the discussion metric in the above example, our arguments apply to any WSRD metric.

Third, since it is difficult, if not impossible, to determine the work required for a request based on the characteristics of the request or system resource utilization, most WSRD techniques that rely on such information simply will not work when applied to ASRD. For this reason, most ASRD techniques use simple RR to distribute requests representing new sessions. Thereafter, requests for existing sessions are distributed to the application server instance where the session's data resides.

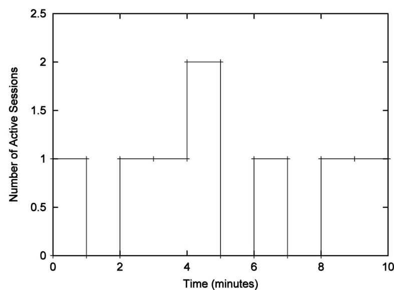


Fig. 2. Load distribution for application server A_1.

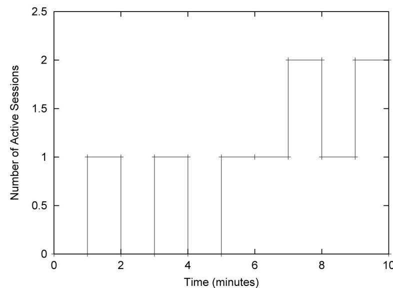


Fig. 3. Load distribution for application server A_2.

Clearly, Session Affinity schemes provide certain distinct advantages (such as state locality) identified previously. However, these policies often result in *severe load imbalances across the application cluster* due primarily to the phenomenon of the convergence of long-running or high-resource-demanding jobs in the same servers.

The problem of load imbalance due to session affinity is well known among practitioners and has received wide treatment in the literature (for example, [14] and [15]). Consider an application cluster having two application servers A_1 and A_2 configured identically. Consider a sequence of sessions arriving at the cluster such that sessions are of two types: a long session S, which lasts 3 minutes, and a short session s, which lasts 1 minute. Suppose that the following sequence of 10 sessions arrive to the cluster and are distributed to A_1 and A_2 according to the session affinity-RR policy: s_1, s_2, S_3, s_4, s_5, S_6, s_7, S_8, S_9, and s_{10}, where the interarrival time between new sessions is 1 minute. This policy results in the load distributions for A_1 and A_2, as shown in Figs. 2 and 3, respectively.

Both figures show load, in terms of the number of active sessions assigned, versus time (in minutes). During the time interval spanning (4, 5), A_1 reaches maximum capacity (two active sessions), whereas A_2 remains idle. A similar situation occurs during the (7, 8) time interval. As this simple example illustrates, a combined RR and Client/Session Affinity strategy can easily create load imbalances across the cluster.

Load imbalance is not the only issue inherent in a session affinity scheme. There is also the issue of the *lack of session failover*. This problem occurs because a session object resides on only one application server. When an application server fails, all of its session objects are lost, unless a session failover scheme is in place. The two main session failover schemes used are *session replication*, in which session objects are replicated at one or more application servers in the cluster, and *centralized session persistence*, in which session objects are stored in a centralized repository (for example, a database management system (DBMS)).

Effectively, these session failover mechanisms "virtualize" a session's data, making it available to any application server instance in the cluster, thus enabling any server in the cluster to service any incoming request. However, there is a cost associated with moving a session object from one server process to another, so it is beneficial to serve a request on the server instance where the session's data already resides. The Request Distribution for the Application Layer (ReDAL) approach attempts to optimize this trade-off by servicing a request on the server instance where the session data resides, unless a significant load imbalance situation is detected, in which case the workload may be transferred off a highly loaded server to a server experiencing lower load. We show the benefits of our approach experimentally in this paper.

The remainder of this paper is organized as follows: In Section 2, we present our ASRD approach. We evaluate the performance of our approach and compare it with existing ASRD policies experimentally in Section 3. Section 4 describes a real-life test of our proposed approach. Finally, we conclude in Section 5.

2 The ReDAL Approach

The ReDAL approach attempts to minimize load imbalances across application servers for stateful session-based applications. To accomplish this, ReDAL augments the traditional session-affinity-based schemes with three specific techniques.

First, we define a *nonintrusive load estimation measure* that senses the relative load of an application server without requiring instrumentation on the application server hardware or software.

Second, based on these load measurements, we propose a *request distribution scheme* that dispatches requests to the affined servers when possible and to less-loaded application servers when there is a significant load imbalance.

Third, in order to minimize the movement of session data between application servers, we introduce a *capacity reservation scheme* that attempts to estimate the near-future expected load on an application server based on the sessions residing on the server and to reserve future capacity sufficient to service those sessions' requests.

2.1 Intuition

In the *ReDAL* approach, we characterize an application server as being in one of two states: 1) *lightly loaded* or 2) *heavily loaded*. We explain these characterizations by using Fig. 4 (adapted from [7]), the upper portion of which shows a typical throughput curve for an application server as load is increased. Section A represents a *lightly loaded* application server, for which throughput increases almost linearly with the number of requests. This behavior is due to the fact that there is very little congestion within the application server system resource queues at such light loads. Section B represents a *heavily loaded* application server. Here, the response time increases proportionally to

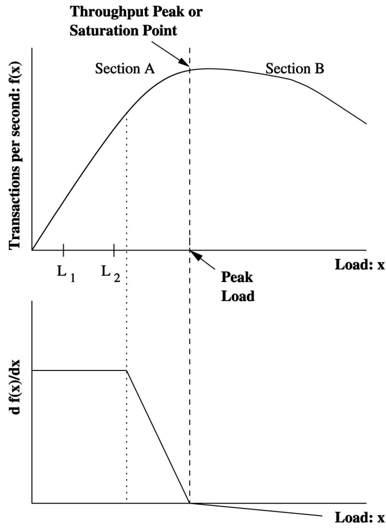


Fig. 4. Typical throughput curve for an application server and its first derivative.

the user load due to increased queue lengths in the application server. Thus, as soon as this *peak throughput point* or *saturation point* is reached, application server performance degrades. We refer to the load level corresponding to this throughput point as the *peak load*.

In order to determine the peak load at runtime, we do not need to find the exact peak throughput point: we need only determine where the *rate of change of throughput with load reaches zero* by looking at the first derivative of the throughput curve—effectively, the slope of the throughput curve. We can generate a close approximation of the slope of throughput curve at runtime by gathering two data values at a configurable interval: 1) transactions per second and 2) number of incoming requests. The lower portion of Fig. 4 shows an approximation of the first derivative $\frac{df(x)}{dx}$ of the throughput curve $f(x)$ shown in the upper part of Fig. 4. Here, $\frac{df(x)}{dx}$ is roughly linear in the early stages of Section A, where the server is very lightly loaded. As the server begins to experience congestion in the later stages of Section A, the slope of $f(x)$ begins to drop as load approaches its peak. In this stage, $\frac{df(x)}{dx}$ drops toward 0 as $f(x)$ approaches the peak load. When $f(x)$ reaches the peak load, $\frac{df(x)}{dx}$ reaches 0. With this, we can designate a server to be lightly loaded if $\frac{df(x)}{dx}$ is positive and heavily loaded if $\frac{df(x)}{dx}$ is 0 or negative.

At an implementation level, finding the point at which $\frac{df(x)}{dx}$ reaches 0 requires a few tweaks to account for burstiness in traffic, as well as the potential for differences in the average response times (ARTs; based on resource usage) across different types of requests. The following expression describes a more detailed view of throughput on a server to help account for these issues: $\frac{df(x)}{dx} \approx \sum_{\forall(p)} \frac{df_p(x)}{dx_p} = \sum_{\forall(p)} \sum_{i=1}^{m-1} \frac{th_p^{(i+1)} - th_p^i}{count_p^{(i+1)} - count_p^i}$, where $\frac{df_p(x)}{dx_p}$ is the slope of the throughput curve for request type p. In this expression, i represents an interval of time (where the typical interval duration on in the order of a second), and the sequence of m intervals represents the m most recent time intervals from the current time, th_p^i represents the throughput achieved for a specific request type p during an interval i, and $count_p^i$ represents the number of requests for request type p that arrived during the interval i. This expression allows us to normalize for burstiness by considering the slope of the throughput curve over a sliding window of time rather than at distinct points in time. It also enables us to generate a separate throughput curve slope per request type and sum these values to get an estimate of the overall health of the application server.

With respect to Fig. 4, we characterize a given application server as either *dispatchable* or *nondispatchable*. A *dispatchable* application server corresponds to a lightly loaded server, whereas a *nondispatchable* application server corresponds to a heavily loaded application server. At an implementation level, this maps to the scenario where $\frac{df_p(x)}{dx_p} \leq 0$ for most or all of the throughput curves for the individual request types p.

The goal of the ReDAL approach, intuitively, is to keep all application servers in its control under "acceptable" throughput thresholds; that is, *the goal here is not to "balance" load per se, but rather to keep the cluster in a stable (not overloaded) state as long as possible: balancing load is an ancillary effect*. Here, "balanced" load refers to the distribution of requests across an application server cluster such that the load on each application server is approximately equal.

The mechanism that the algorithm follows to achieve this goal is explained as follows. At decision time, that is, when a request needs dispatching, it attempts to send the request to an *affined* dispatchable server (that is, the server where the immediately prior request in the session was served), failing which it attempts to send the request to the "least loaded" dispatchable server, and finally, if the above two conditions cannot be met, sends it to the "least loaded" server overall. Clearly, we must first figure out the load levels of servers, which we can then map to dispatchability.

ReDAL follows a *capacity reservation* procedure to judge load levels. At an intuitive level, this capacity reservation mechanism is based on two key premises. First, it assumes that the think time or view time between user actions is predictable based on past behavior. This is a valid assumption: previous research (for example, [16]) on online user behavior shows that think time is highly predictable. Second, it assumes that session affinity, where consecutive requests in a given user session are handled by the same application server instance, will improve performance. We show the validity of this assumption through the experimental results in Section 3.

We now describe the capacity reservation procedure in detail. Consider an application server A_k processing

y sessions. Assume that it is desired to keep the server under a throughput of T. Further, it takes h seconds, on the average, between consecutive requests inside a session (this is referred to as think time) and that the system, at any given time, considers the state of this application server G seconds into the future. Given this information, for tractability, let us partition the look-ahead period G into C distinct time slices of duration d. Such partitioning allows us to make judgments effectively: Given that we are attempting to compute a decision metric (throughput in this case), it is easier and more reliable to monitor this metric over discrete periods of time rather than performing continuous dynamic monitoring at every instant.

In terms of the capacity reservation procedure, given y sessions in the current time slice, we assume that each of these sessions will submit at least one more request. Clearly, these requests are expected to arrive in a time slice h units of time away from the current slice in time slice c_h. This prompts us to reserve capacity for the expected request in this application server in c_h. More accurately, when a request r arrives at an application server A_k at time t, assuming that this request belongs to a session $\mathcal{S}$, we reserve a unit of capacity (sufficient to service a request) on A_k for the time slice containing the time instant $t + h$. Note that this reflects our desire to preserve affinity: we assume that all requests for session $\mathcal{S}$ will ideally be routed to A_k. Such rolling reservations provide a basis for judging expected capacity at an application server. To dispatch a request, if dispatching the request to the affined server is not possible, then we check the different application servers in the cluster to see which ones have the property that the amount of reserved capacity in the current time slice is under the desired maximum throughput T and choose the least loaded server among them.

If all the application servers are found to be in the nondispatchable state, then we have two options: 1) we can send the request to the application server with the least load at the current time or 2) we can delay the dispatching decision until some server becomes dispatchable. Since the overall system is not fully deterministic, the first dispatchable server may not be the server that would have been chosen under option 1.

Thus, we consider two variations on our algorithm to handle the case where no server is in a dispatchable state: standard ReDAL and a modified version of ReDAL, which we will call *ReDAL-Wait (ReDAL-W)*. In the standard ReDAL, the request is queued at the application server to which it was dispatched until the application server has finished processing prior requests and has the capacity to service the request. In ReDAL-W, the new request is not sent immediately to any particular application server instance but rather placed into a queue on the ReDAL request dispatcher. Requests in this queue are dispatched to an application server only when an application server becomes dispatchable. Delaying the dispatching decision until a server becomes dispatchable takes advantage of additional information available in the future, specifically which application server instance becomes dispatchable first, allowing us to make a more accurate dispatching decision than in the standard ReDAL case.

The above discussion, of course, does not account for every practical issue. In reality, we have to account for various other issues, for example, the fact that the current request may actually be the last request in a session (in which case the reservation that we have made is actually an overestimation of the capacity required), and the fact that we may have misestimated the think time for a particular request. The full ReDAL algorithm takes care of these practical issues.

2.2 System Architecture

The architecture of our proposed approach is similar to that shown in Fig. 1. Our system consists of two main logical modules: 1) the *Application Analyzer* and 2) the *Request Dispatcher*. The Application Analyzer and Request Dispatcher reside together on the Web server as a plug-in (denoted as circles within the Web servers in Fig. 1).

The *Application Analyzer* is responsible for characterizing the behavior of an application server as *dispatchable* or *nondispatchable*. This module monitors each application server's throughput to generate a close approximation of the slope of the server's throughput curve $f(x)$ and designates a server as dispatchable if $\frac{df(x)}{dx}$ is positive and nondispatchable if $\frac{df(x)}{dx}$ is zero or negative (we remind the reader that, as noted earlier in this section, it is not necessary to generate the exact throughput curve for an application server: we only need the slope of the curve). These values are used by the Request Dispatcher module, which we describe next.

The *Request Dispatcher* is responsible for the runtime routing of requests to a set of application servers according to our proposed request routing policy. To accomplish this, the Request Dispatcher monitors the expected and actual load on each application server. Upon receiving a request, the Request Dispatcher first determines whether the request is part of an existing session. If so, it will direct the request to the application server owning the session, as long as the affined server is in a dispatchable state. Otherwise, it will send the request to the application server having the lowest expected load. Requests that initiate a new session are also routed to the least loaded application server. Though not shown in Fig. 1, we assume that there is a session virtualization mechanism (as described in Section 1) in place to enable session failover.[1]

2.3 Technical Details

We consider a set of application servers $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ configured as a cluster, where a cluster is a set of application servers configured with the same code base and sharing runtime operational information (for example, user sessions and EJBs). For the sake of simplicity, we assume that each application server A_k $(k = 1, \ldots, n)$ is identical, though our approach also applies in the case of heterogeneous application servers. A request r is a specific task to be executed by an application server. We assume that each request is part of a session $\mathcal{S}$, where a session is defined as a sequence of requests from the same user or client. In other words, $\mathcal{S} =< r_{1,\mathcal{S}}, r_{2,\mathcal{S}}, \ldots, r_{s,\mathcal{S}} >$, and $r_{j,\mathcal{S}}$ denotes the jth request in $\mathcal{S}$. A set of Web

1. Such mechanisms are provided with virtually every commercial application server, either as a native feature or through the use of a DBMS. Third-party solutions are also available.

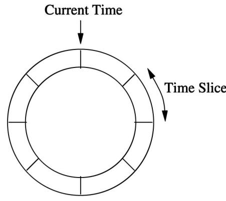


Fig. 5. Cycle of time slices.

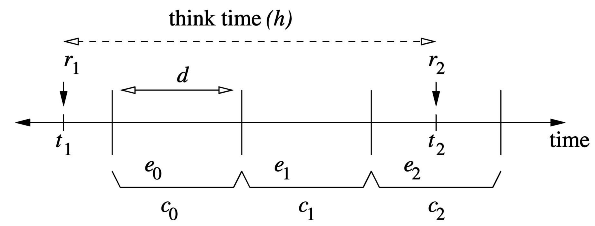


Fig. 6. Load metrics.

servers $\mathcal{W} = \{W_1, W_2, \ldots, W_n\}$ dispatches application requests to the application servers in $\mathcal{A}$. Based on this foundation, let us define some notions that will be used in our algorithmic description.

Think time (h) is defined as the time between two consecutive requests $r_{j,\mathcal{S}}$ and $r_{j+1,\mathcal{S}}$, measured in seconds. Think time is computed as a moving average of the time between consecutive requests from the same session arriving at the cluster. The moving average considers the last g requests arriving at the cluster, where g represents the window for computing the moving average and is a configurable parameter.

A *time slice* (c_i) is defined to be a discrete time period of duration d (in seconds, where d is greater than the time required to serve an application request), over which we record measurements for throughput on each application server. We consider a finite number of such time slices $\mathcal{C} = \{c_0, c_1, \ldots, c_{C-1}\}$, where c_0 represents the current time slice, each c_i $(i = 0, \ldots, C-1)$ represents the ith time slice, and C allows sufficient time slices for reservations h sec in the future, that is, $C = \lceil \frac{h}{d} \rceil$. The C time slices are organized in a cycle of time slices for each application server, as shown in Fig. 5. Each time slice will have an associated set of two load metrics, *actual load* and *expected load*, which are updated as new requests arriving and as existing requests are served.

The *actual load* (l_k^t) of an application server A_k at time t is defined as the number of requests arriving at A_k within a time slice c_i such that $t \in c_i$ (we drop the t superscripts when t is implicit from the context). Intuitively, l_k^t maintains the count of requests that have been assigned to application server k within the current time slice c_i. For example, if 10 requests have been assigned to application server k since the start of c_i, then $l_k^t = 10$.

Consider a request r_j of a session $\mathcal{S}$ arriving at time t_p. The *predicted time slice* c_q of the subsequent request in the session, that is, r_{j+1}, is the time slice containing the time instant $t_p + h$ such that the request r_{j+1} is predicted to arrive at the time instant $t_p + h$.

The *expected load* (e_i^k) of an application server A_k for the time slice c_i is defined as the number of requests expected to be served by A_k during the time slice c_i. The expected load is determined by accumulating the number of requests that a given application server should receive during c_i based on the predicted time slices for future requests for each active session associated with A_k.

Fig. 6 helps illustrate how the expected load is determined. The figure shows a linear view of a partial cycle of time slices. Each time slice has an expected load counter. For instance, consider the cycle for A_k. Here, e_0^k represents the expected load counter for the current time slice (c_0), e_1^k the expected load counter for time slice c_1, and so on. Suppose that request r_1 in a particular session occurred at time t_1, as shown in the figure. From the think time (h), we can determine the time slice in which request r_2 is expected to arrive. Suppose that, based on the think time, it is determined that request r_2 will arrive at time t_2, which occurs in time slice c_2 (refer to Fig. 6). Then, e_2^k, which is the expected load for time slice c_2, is incremented by 1. This effectively reserves capacity for this request on A_k during c_2.

Since predicted time slices are not guaranteed to be correct, we may need to adjust the expected load to account for incorrect predictions. An incorrectly predicted request may arrive either 1) in a time slice prior to its predicted time slice or 2) in a time slice subsequent to its predicted time slice. In the former case, we simply decrement the expected load counter for the predicted time slice upon observing the arrival of the request in the current time slice. For example, referring to Fig. 6, suppose that request r_2 actually arrives during the current time slice (c_0). In this case, the actual load l for the current time slice is incremented, whereas the expected load e_2^k for time slice c_2 is decremented. This effectively cancels the reservation for this request on the application server during the future time slice.

In the case where a request arrives subsequent to its predicted time slice, we have no way of knowing about this error until we reach the end of the predicted time slice. We can only estimate that this type of error will occur with a certain frequency. We account for this type of error in our *modified load* metric m_k for application server A_k, defined as $m_k = l_k^t + \alpha e_0^k$, where α $(0 < \alpha \leq 1)$ is an expected load factor, which adjusts for requests that arrive after their predicted time slices.

Setting an optimal value of α requires first estimating the think time and then adjusting α for the correctness of that estimate. There are multiple methods of estimating the think time in the literature, for example, based on the analysis of Web logs [17] or logs generated with an HTTP packet sniffer [18]. Such logs provide data on the interarrival times for user requests, that is, think time. Finding the correctness of the think time estimates generated from such logs can be done with standard train and test techniques drawn from artificial intelligence, that is, using two data samples, where an estimate of the think time is generated using the first sample and tested using the second sample to obtain a value for correctness of the think time estimate. For applications where the think time estimate is very accurate, higher values of α can be used (for example, $\alpha = 0.9$). The value of α should be reduced as the correctness of the think time estimate becomes less accurate.

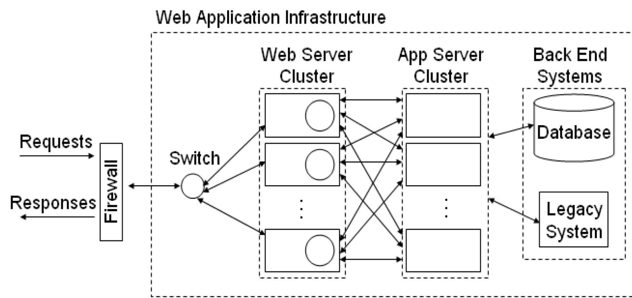


Fig. 7. Web application architecture with a single application server cluster.

We briefly summarize the above-described load metrics. For a given application server, we maintain an expected load counter for each time slice. For the current time slice, we record the actual load by observing the number of requests served by the application server. We then compute the modified load for the current time slice by summing the actual load and the adjusted expected load (adjusted to account for prediction errors).

2.4 Handling Multiple Web Servers

The preceding discussion focused on the load metrics maintained by a single Web server. In the best practices recommended architecture [1], each Web server dispatches requests to a separate cluster, so there is no need to share load metrics across Web servers. However, in practice, there are multi-Web server environments (for example, as depicted in Fig. 7) in which multiple Web servers dispatch requests to the same application server cluster. Since each Web server runs its own instance of the Request Dispatcher, we must ensure that each Request Dispatcher accesses the same global view of load metrics. To accomplish this, each Request Dispatcher maintains a synchronized copy of the global view of load metrics. This global view is updated via a multicast synchronization scheme, in which each Request Dispatcher periodically multicasts its changes to all other Request Dispatcher instances. This data sharing scheme allows all Request Dispatcher instances to operate from the same global view of load on the application servers and yet allows each instance to act autonomously. Another issue that arises in a multi-Web server environment is computing the think time, given that consecutive requests from the same session may be sent to a different Web server. To address this issue, each Web server, upon sending an HTTP response, records the time that the response is sent in a cookie. Thus, if a subsequent request from this session is sent to a different Web server, then the new Web server can retrieve the time of the last response and use it to compute the think time.

It should be noted that this synchronization scheme adds very little overhead to the system, both in terms of network communications overhead and processing overhead. The communications overhead depends on the number of application servers, the number of time slices, and the storage space needed for the load metrics. The number of Web servers is not included in this computation because in a multicast network, the number of recipients of a message (here, Web servers are the recipients) does not matter: the message is broadcast once, and all recipients receive it. To reduce the potential for multiple Web servers to assign requests to the same "least loaded" server, the multicast interval should be set to ensure that synchronization of load metrics occurs multiple times per time slice.

For example, consider an application environment having 50 application servers and a think time (h) of 60 seconds.[2] If we assume a time slice duration (d) of 5 seconds, then the number of time slices (C) is $60/5 = 12$. The load metric value and the current throughput value can each be stored as 1-byte integers. Since there is only a single value for each of the actual load and current throughput values, synchronizing this data across the Web server plug-ins requires transmitting 2 bytes for each of the 50 application servers and thus incurs 100 bytes of synchronization overhead. Transmitting the expected load requires sending 12 bytes (1 byte for each time slice) for each of the 50 application servers, incurring 600 bytes of synchronization overhead. Thus, the total synchronization overhead incurred for a Web server cluster, summing the overheads for actual load, current throughput, and expected load, is 700 bytes per transmission per Web server. Considering the overhead for the Ethernet protocol (42 bytes), Internet Protocol version 4 (IPv4; 20 bytes), and User Datagram Protocol (UDP; 8 bytes) [19], the total network overhead per Web server becomes 770 bytes per transmission. If we assume a UDP multicast interval of 1 second and five Web servers (to serve the 50-application-server cluster), then the maximum overhead possible at any given time is 30.8 kilobits per second (Kbps), which is negligible (less than 0.03 percent) in the context of the total capacity of a network of 100 megabits per second (Mbps; and far less on gigabit networks, which are becoming increasingly prevalent in enterprise application infrastructures).

With regard to processing overhead, a given Request Dispatcher performs $n \times C$ operations to apply the updates that it receives from another Request Dispatcher. Since each Request Dispatcher applies the changes that it receives to its own copy of the global view array, there is no locking contention.

A second potential issue can arise in the scenario where multiple Web servers dispatch requests to the same application cluster (as depicted in Fig. 7). In such a scenario, if a very large number of requests for new sessions arrive within this time interval, all the new requests will be sent to the same least loaded application server. To prevent this from occurring, we implement a simple estimation scheme in the Web servers, assuming a uniform distribution of this large number of requests for new sessions across the Web server cluster. Here, each Web server in the cluster maintains a separate *estimated actual load* value between updates. For each request requiring a new session arriving at a given Web server instance, the instance assumes that a similar request has arrived on all other clusters and increments the actual load for the least loaded application server by 1 and the estimated actual load by the size of the cluster. This estimate is reset every time the updated load values arrive from the other Web servers in the cluster.

2. These values were obtained from a major Web retailer.

3 Experimental Results

In this section, we show the runtime performance of the ReDAL algorithm with a set of experimental results, comparing it to a widely used existing technique, specifically a commercial implementation of the RR scheme and the HJ load balancing scheme. We consider two cases for the ReDAL algorithm with two different settings for the α parameter, $\mathrm{ReDAL} - \mathrm{ALPHA} = 0.9$ and $\mathrm{ReDAL} - \mathrm{ALPHA} = 0.5$, to show the impact of varying α.[3] Here, higher values of α take greater advantage of ReDAL's reservation scheme than lower values.

In this, we are interested in five particular questions:

1. How does the throughput performance compare across the RR, HJ, $ReDAL - ALPHA = 0.5$, $ReDAL - ALPHA = 0.9$, and $ReDAL - W - ALPHA = 0.9$ algorithms?
2. How does the response time performance compare across the RR, HJ, $ReDAL - ALPHA = 0.5$, $ReDAL - ALPHA = 0.9$, and $ReDAL - W - ALPHA = 0.9$ algorithms?
3. How does each of these policies impact the CPU resource utilization on the Web server?
4. How does ReDAL impact CPU overheads on the application server?
5. How is application server scaling affected by ReDAL?

3.1 Experimental Architecture

Our experiments were run using the general architecture described in Fig. 1, with the addition of a load generation tool to simulate user requests and a session clustering mechanism. As the topology described in Fig. 1 is described as the "best possible topology" by the IBM WebSphere scalability documentation [1], we primarily focus on this topology. Later in this section, we also demonstrate the impact of the topology depicted in Fig. 7 on our ReDAL approach.

The experimental environment consists of a LoadRunner v6 load generator [20], which simulates client requests, several Apache HTTP server v2.0 [21] Web server instances, and several WebLogic server v7.1 [10] application server instances. The numbers of Web server and application server instances that we use in our experiments are described in Table 1. We use two Oracle 10-Gbyte [6] database servers.

Of the two database servers, one stores application data. The other serves as a session object repository, which ensures that all session objects are accessible from each application server instance. Session access is implemented as an override of the HttpSession object, which connects to the database to read and write the session data, if not already residing in the application server's memory space.

We have implemented the ReDAL algorithm as an Apache Web server plug-in module, written in C++. For the RR algorithm, we use the WebLogic Apache plug-in module, which implements a round-robin dispatching policy. We have implemented HJ as an Apache plug-in, adding support for statefulness (not addressed in [2]) through calls to an external session object repository.

3. We have found that $\alpha = 1$ is effective only in the case where there is no error in the think time prediction in capacity reservation. This is not a realistic scenario; thus, we consider values of α up to 0.9 in these experiments.

TABLE 1
Experimental Parameters

Parameter	Values	Base Value
Number of Web Servers	1, 2, 5	2
Number of Application Servers	10, 20, 30	20

The Transaction Processing Performance Council benchmark W (TPC-W) [22] is used in all experiments. The TPC-W application is an online bookstore. One Oracle database 10-Gbyte server is used to store the book and transactional data, as described in TPC-W. The cardinality of the ITEM table in TPC-W in these experiments is 100,000. Fourteen user activities are defined in the TPC-W benchmark, of which six activities fall under the "browse" classification, and eight activities fall under the "order" classification. Three different mixes ("Browsing," "Shopping," and "Ordering") of these activities are defined in TPC-W. For our experiments, we used the "Shopping" mix, where 80 percent of the user actions fall under "browse," and 20 percent of the user actions fall under "order."

The load generator is configured to simulate a varying number of simultaneous user sessions, with each session submitting a stream of requests to the Web server. Each request is chosen randomly, as defined in the TPC-W benchmark. The think time (h) between requests is set to a small number of milliseconds to allow us to minimize the number of threads required on the load generator while still simulating significant loads on the experimental architecture.

Due to the short think times between requests, we must also use a small window size. For all experiments, the window size for the ReDAL algorithm is set to 100 ms, that is, $d = 0.1$ (in real life, the think time is significantly longer than millisecond time frames: typically, d is in the multi-second range).

Sessions are stored to the external session repository (in an Oracle 10-Gbyte database) and in the application server's memory space. If a request arrives at an application server for a locally stored session object, then the read speed is dramatically reduced over the external retrieval case. If a request updates a session object that resides on another application server, then an invalidation message is sent to remove the object. This configuration is used in the HJ and ReDAL cases, where session virtualization is required (RR does not require session virtualization).

All machines used in the experiments are configured with a dual-core dual-CPU (1.5 MHz), 2-Gbyte RAM, and 20-Gbyte disk, and run a Windows 2003 server. All communications take place on a local-area 100-Mbps Ethernet network. All application server instances are installed in 10 machines, deployed as follows: When we use 30 application servers, each machine is running three

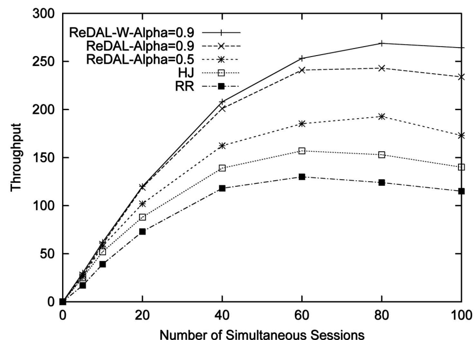


Fig. 8. Average throughput per application server.

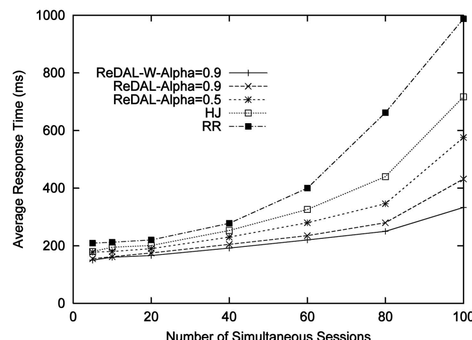


Fig. 9. Average response time.

application server instances. When we use 20 application servers, each machine is running two application server instances, and when we use 10 application servers, each machine is running one application server instance. Each Web server instance is run on a separate machine. Thus, when we are running five Web server instances, we have five machines dedicated for Web servers. Three machines are used to simulate user sessions, and one of these three machines additionally hosts the LoadRunner console, which displays consolidated performance data.

In these experiments, we measure three performance metrics: 1) *throughput* refers to the average number of transactions per second that the cluster of application server provides, 2) *Average Response Time (ART)* refers to the average request response time that the cluster of application servers can provide (the throughput of the cluster and ART are measured from the perspective of the user), and 3) *Web Server CPU Utilization* (WSCU) refers to the percentage of CPU utilization on the Web server, as measured by operating system utilities.

3.2 Throughput Performance

Fig. 8 shows how throughput varies for $\mathrm{ReDAL-W-ALPHA} = 0.9$, $\mathrm{ReDAL-ALPHA} = 0.9$, $\mathrm{ReDAL-ALPHA} = 0.5$, HJ, and RR as the number of simultaneous sessions increases from 5 to 100 for 20 application servers and two Web servers.

For all approaches, the throughput shows an inverted "U" shape; that is, the throughput rises initially, peaks, and then falls. The throughput rises initially as the arrival rate of requests increases and then peaks when a resource on the server reaches maximum utilization (for example, CPU reaches 100 percent). Once a resource reaches its maximum usage, queuing for that resource begins, causing the throughput to drop.

We now consider each curve relative to one another. For the $\mathrm{ReDAL-ALPHA} = 0.5$ curve, the throughput/server peaks at 80 simultaneous sessions, with 192 transactions per second per server. HJ and RR do not perform as well as $\mathrm{ReDAL-ALPHA} = 0.5$: both peak at 60 simultaneous sessions, providing only 130 transactions per second per server in the RR case and with 157 transactions per second in the HJ case. The lower throughput in the RR case results from one or more of the application servers in the cluster reaching a resource bottleneck (in this case, CPU utilization reaching 100 percent) due to unbalanced load, bringing down the overall throughput on the cluster. This clearly shows the impact of maintaining balanced load across the application server cluster that ReDAL provides. The lower throughput in the case of HJ stems from the fact that HJ does not take advantage of session affinity and needs to retrieve the session from external storage on every request. On the other hand, $\mathrm{ReDAL-ALPHA} = 0.9$ outperforms $\mathrm{ReDAL-ALPHA} = 0.5$, peaking at 80 simultaneous sessions and providing higher throughput, with 243 transactions per second per server. This shows the benefit of ReDAL's reservation planning capability, which has greater impact as α is increased. The $\mathrm{ReDAL-W-ALPHA} = 0.9$ performs the same as $\mathrm{ReDAL-ALPHA} = 0.9$ until all servers reach the saturation point (peak throughput). At the saturation point, all servers become nondispatchable. Here, the ReDAL-W algorithm performs better than the ReDAL, with the peak at 269 transactions per seconds at 80 simultaneous sessions. The nominal improvement of our ReDAL-W is also seen at the load of 100 sessions, where the throughput in the case of $\mathrm{ReDAL-W-ALPHA} = 0.9$ is 264 transactions per second compared to 234 transactions per seconds in the case of $\mathrm{ReDAL-ALPHA} = 0.9$.

3.3 Response Time Performance

Our response time experimental results, shown in Fig. 9, show how ART varies for $\mathrm{ReDAL-W-ALPHA} = 0.9$, $\mathrm{ReDAL-ALPHA} = 0.9$, $\mathrm{ReDAL-ALPHA} = 0.5$, HJ, and RR as the number of simultaneous sessions increases from 5 to 100.

For all approaches, the ART curves are exponential. Here, response time is relatively flat initially and then begins to increase with each successive value for simultaneous sessions. The points where the slopes of these curves begin to increase sharply are closely correlated to the peaks in the throughput curves. Specifically, these "knee points" map exactly to the peaks in the average throughput per application server (ATAS) curves. Here, as the arrival rate of requests increases, the response time begins to increase sharply when a resource on the server reaches maximum utilization, at which point queuing begins, causing rising response times.

We now consider each curve relative to one another. For the $\mathrm{ReDAL-ALPHA} = 0.5$ curve, the response time begins to increase sharply at 80 simultaneous sessions, with a response time of 346 ms. RR does not perform as well as $\mathrm{ReDAL-ALPHA} = 0.5$. Here, the response time begins to

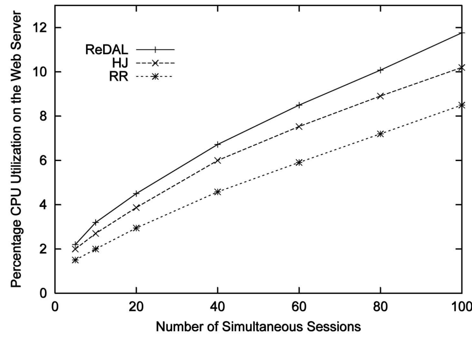


Fig. 10. Average CPU utilization on the Web server.

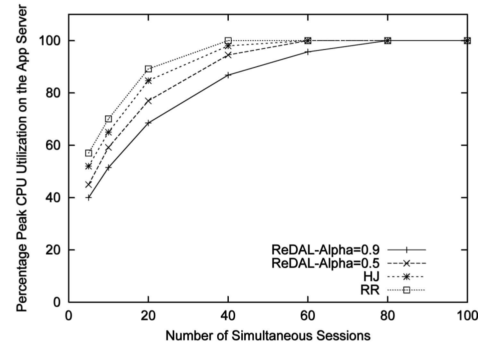


Fig. 11. Peak percent of CPU on the application servers.

increase sharply at a lower simultaneous session load of 60 sessions, providing a response time of 662 ms at 80 simultaneous sessions. This underscores the point made with regard to throughput: maintaining balanced load across the application server cluster provides significant benefit. For the HJ case, the response time is higher than the $\mathrm{ReDAL} - \mathrm{ALPHA} = 0.5$ case, that is, 440 ms at 80 sessions, reinforcing the points shown in the throughput experiment that there is significant advantage in utilizing session affinity. On the other hand, $\mathrm{ReDAL} - \mathrm{ALPHA} = 0.9$ outperforms $\mathrm{ReDAL} - \mathrm{ALPHA} = 0.5$. Although it begins to rise sharply at the same number of simultaneous sessions (that is, 80), it provides a lower ART of 280 ms for 80 simultaneous sessions. This reiterates our point regarding the benefits of ReDAL's reservation mechanism. Further, at high loads when all servers reach the saturation point, that is, the nondispatchable state, our ReDAL-W algorithm performs better than ReDAL. The ART value for 80 simultaneous users in the case of $\mathrm{ReDAL} - \mathrm{W} - \mathrm{ALPHA} = 0.9$ is 250 ms compared to 280 ms in the case of $\mathrm{ReDAL} - \mathrm{ALPHA} = 0.9$. Similarly, the ART value for 100 simultaneous users is less in the case of $\mathrm{ReDAL} - \mathrm{W} - \mathrm{ALPHA} = 0.9$ than $\mathrm{ReDAL} - \mathrm{ALPHA} = 0.9$. This demonstrates the benefit of dispatching a request only when an application server is ready to serve, which is done in our algorithm ReDAL-W. In most real-life scenarios, application servers are seldom operated at the saturation point, that is, in the nondispatchable states, so we focus our further experimental and real-life studies on our standard ReDAL algorithm.

3.4 Peak CPU Usage on the Web Server

We show that the response time and throughput benefits of ReDAL come at a very low computational cost by considering the average CPU overheads on the Web server, which is where the three approaches differ. Fig. 10 shows how WSCU varies for ReDAL, HJ, and RR as the number of simultaneous sessions increases from 5 to 100 for 20 application servers and 2 Web servers. Here, we show only the results for $\alpha = 0.9$ for the ReDAL case, since the value of α does not impact the work required for ReDAL. In addition, experiments showed that the impact of the single additional thread in the case of ReDAL-W is so minimal that the percentage CPU utilization in the case of ReDAL-W is almost same as the ReDAL case. Thus, we do not report the CPU utilization of the Web server in the case of ReDAL-W separately.

For all approaches, the WSCU curves are linear with a positive slope; that is, CPU utilization increases with increasing simultaneous sessions. The RR approach shows the lowest overall WSCU, rising from 1.55 percent at five sessions to 8.5 percent at 100 sessions. The HJ case shows slightly higher values than RR, rising from 2 percent to 10.2 percent, due to the fact that it tracks more information about the application cluster than RR, essentially a count of active requests on each application instance. ReDAL also shows slightly higher WSCU values than RR, rising from 2.2 percent to 11.6 percent. These values are higher than that of RR and HJ because ReDAL not only maintains load information for application servers but also exchanges that data across the Web server cluster. Overall, this costs 3 percent of additional CPU over RR and 1.4 percent over the HJ case, which is a very low cost to pay to obtain the throughput and response time benefits shown above.

3.5 CPU Overheads on the Application Server

Here, we demonstrate how the peak CPU of application server varies for different load distribution schemes. For each load distribution scheme, that is, HJ, RR, and ReDaL, at each number of simultaneous sessions, we note the peak percent of CPU across 20 application servers with two Web servers. We plot this peak percent of CPU versus the number of simultaneous sessions in Fig. 11. Due to highly unbalanced load distribution, the peak percent of CPU is higher in the case of RR and HJ than in the case of ReDal. Also, for RR and HJ, the peak CPU reaches 100 percent earlier than in the ReDal case. This is reflected in the increase in ART in Fig. 9.

3.6 Scaling with Additional Application Servers

Fig. 12 shows CPU usage on the Web server for simultaneous sessions increasing from 5 to 100, for 10, 20, and 30 application servers running behind the two Web servers. Each case uses ReDal, with $\alpha = 0.9$, to distribute the request load across the application servers. The curves all increase as the number of simultaneous sessions increases: each additional session increases the number of requests that must be distributed across the application server set. The curves for 10, 20, and 30 application servers all show very similar CPU growth rates as the number of simultaneous sessions increase, with the 10-server case showing slightly lower CPU usage than the 20-server case and the 20-server case showing slightly higher CPU usage than the 30-server case. Clearly, increasing the number of application servers results in increased CPU usage on the Web server due to the

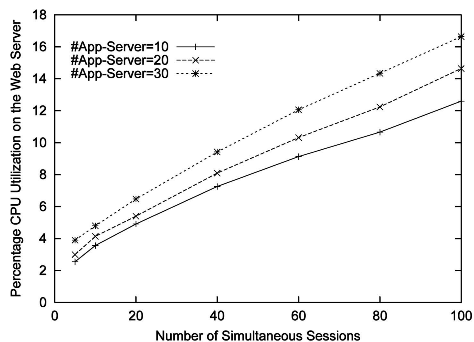


Fig. 12. Scaling with application servers.

increased complexity in tracking the load states of more servers; however, this increase is very small: the difference between the 20-server and 30-server cases is about 4 percent.

3.7 Scaling with Additional Web Servers

Fig. 13 shows how the ReDAL approach performs with varying numbers of Web servers. In this experiment, we kept the number of application servers constant at 20 and varied the number of Web servers to 1, 2, and 5, and measured the end-to-end response time for each case. We consider two topologies in this experiment: 1) the hierarchical topology shown in Fig. 1, in which each Web server dispatches requests to a separate application server cluster (for one, two, and five Web servers) and 2) the nonhierarchical topology shown in Fig. 7 (for five Web servers, marked "NH" in Fig. 13).

The first noticeable point about this set of plots is that the curves are clustered tightly together, showing that the end-to-end response time does not vary significantly with the number of Web servers. This is due to the fact that the primary bottleneck in the TPC-W benchmark is the application server processing. The slight improvement in the performance with the increased number of Web servers is due to the fact that the HTTP protocol and image request processing (which is the responsibility of the Web server) is spread across a larger number of Web servers.

We now consider the differences between the curves in Fig. 13 based on topology. Topology begins to impact the ReDAL experimental setup when the number of Web servers is greater than 2. In both the hierarchical (Fig. 1) and nonhierarchical (Fig. 7) topology cases, when two Web servers are deployed, they are typically fully synchronized for failover purposes, including any plug-in information. Thus, we do not show a curve here for "$\#\text{Web Server} = 2\ \text{NH}$" because it is exactly the same as in the hierarchical case.

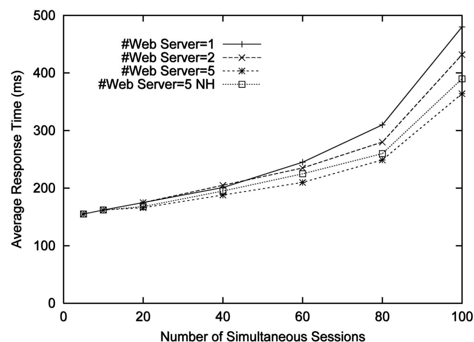


Fig. 13. Scaling with Web servers.

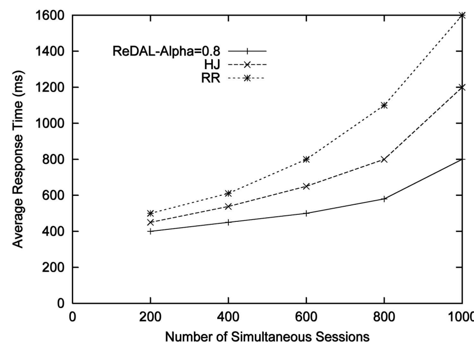


Fig. 14. Performance on a real-world application.

We are therefore interested in the curves for "$\#\text{Web Server} = 5$" and "$\#\text{Web Server} = 5\ \text{NH}$." The curve for "$\#\text{Web Server} = 5\ \text{NH}$" shows a slight decrease (8 percent at the highest load levels) in performance as compared to the case of "$\#\text{Web Server} = 5$." This is due to the additional overhead of sharing load metrics across the Web server cluster when the load and throughput data must be synchronized, as well as the slight increase in processing required to compute the load matrix for all the application servers.

4 Case Study

To validate our experimental results, we tested the performance of our ReDAL algorithm in the staging environment of an online application at a major US credit card issuer. In this staging environment, the application runs on 30 instances of the WebLogic application server running on RedHat Linux 9.0, where the WebLogic cluster receives requests through an Apache HTTP server 2.0 running on Linux RedHat 9.0.

Load is distributed across the WebLogic cluster using WebLogic's Apache plug-in, which uses the RR algorithm to distribute requests. User sessions are synchronized across the cluster using the WebLogic session synchronization mechanism.

We implemented two additional Apache plug-ins, representing the ReDAL and HJ algorithms, for performance comparison purposes. Based on anonymized Web server logs from the application, we generated LoadRunner 6.0 [20] scripts to emulate user behavior. In Fig. 14, we show the response time of the system as recorded by LoadRunner versus the number of sessions connected to the system for ReDAL, HJ, and RR.

For ReDAL, we varied α and found that we obtained the optimal performance at $\alpha = 0.8$ for the application. Thus, in Fig. 14, we plot ReDAL for $\alpha = 0.8$. As can be seen in Fig. 14, the response time is lowest in the case of ReDAL. The rate of change in the slope of the curve, that is, the increase of response time as the number of sessions increases, is also much lower for ReDAL than for HJ and RR. Though RR is the most widely used ASRD logic, it is a very rudimentary algorithm. In this study, HJ improves the response time by 25 percent at 1,000 sessions, whereas our

ReDAL improves the overall performance by 50 percent over RR. This demonstrates the applicability of our algorithm in a real-life case.

5 Conclusion

We devise an approach for distributing requests across a cluster of application servers such that the overall system throughput is enhanced, and load across the application servers is balanced. Our approach considers two cases: one suited to clustered servers that are sized to handle peak load on the site and a second case that is best suited for a highly loaded cluster of servers that operates beyond the saturation point. We compare the performance of our approach with widely used industrial and recently proposed techniques from the literature experimentally in terms of the throughput and response time performance, as well as resource utilization.

References

[1] K. Ueno, T. Alcott, J. Blight, J. Dekelver, D. Julin, C. Pfannkuch, and T. Shieh, *WebSphere Scalability: WLM and Clustering, Using WebSphere Application Server Advanced Edition (IBM Redbook).* IBM Int'l Technical Support Organization, Sept. 2000.

[2] S. Hwang and N. Jung, "Dynamic Scheduling of Web Server Cluster," *Proc. Ninth Int'l Conf. Parallel and Distributed Computer Systems (ICPADS '02),* 2002.

[3] Cisco, LocalDirector, www.cisco.com, 2007.

[4] *F5-Networks,* BIG-IP, www.f5.com, 2007.

[5] *Linux Virtual Server Project,* Linux virtual server, www.linuxvirtualserver.org, 2007.

[6] Oracle 10g database server, Oracle, Inc. http://www.oracle.com, 2007.

[7] WebSphere, IBM, http://www.ibm.com, 2007.

[8] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-Aware Request Distribution in Cluster-Based Network Servers," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '98),* 1998.

[9] H. Elmeleegy, N. Adly, and M. Nagi, "Adaptive Cache-Driven Request Distribution in Clustered EJB Systems," *Proc. 10th Int'l Conf. Parallel and Distributed Systems (ICPADS '04),* pp. 179-186, July 2004.

[10] Weblogic server v7.1, BEA, Inc., http://www.bea.com, 2007.

[11] M. Colajanni, P. Yu, and D. Dias, "Scheduling Algorithms for Distributed Web Servers," *Proc. 17th Intl' Conf. Distributed Computing Systems (ICDCS '97),* pp. 169-176, May 1997.

[12] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance Management for Cluster Based Web Services," *Proc. Eighth IFIP/IEEE Int'l Symp. Integrated Network Management (IM '03),* pp. 247-261, Mar. 2003.

[13] A. Tannenbaum, *Modern Operating Systems.* Prentice Hall, 2001.

[14] V. Viswanathan, *Load Balancing Web Applications,* p. 1, http://www.onjava.com/pub/a/onjava/2001/09/26/load.html?, Sept. 2001.

[15] *The Effects of Distributing Load Randomly to Servers,* white paper, Cisco, 1997.

[16] J. Heer, "Mining the Structure of User Activity Using Cluster Stability," *Proc. SIAM Conf. Data Mining, Web Analytics Workshop,* 2002.

[17] I. Nino, B. de la Ossa, J. Gil, J. Sahuquillo, and A. Pont, "Carena: A Tool to Capture and Replay Web Navigation Sessions," *Proc. Fifth Workshop End-to-End Monitoring Techniques and Services (E2EMON '05),* May 2005.

[18] F. Smith, F.H. Campos, K. Jeffay, and D. Ott, "What TCP/IP Protocol Headers Can Tell Us About the Web," *Proc. Joint ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '01),* pp. 245-256, June 2001.

[19] W. Stevens, B. Fenner, A. Rudoff, and R. Stevens, *UNIX Network Programming, Vol. 1: The Sockets Networking API,* third ed. Addison-Wesley Professional, 2003.

[20] Mercury Interactive, Loadrunner v6, http://mercuryinteractive.com, 2005.

[21] The Apache HTTP Server Project, Apache http server v2.0, http://www.apache.org, 2007.

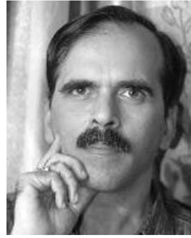
[22] *TPC benchmark W,* 2001, Transaction Processing Performance Council, http://www.tpc.org/tpcw/default.asp, 2007.

Kaushik Dutta received the BS degree in electrical engineering from Jadavpur University, the MS degree in computer science from the Indian Statistical Institute, and the PhD degree from Georgia Institute of Technology. He was the director of engineering at Chutney Technologies, a software company that develops solutions to improve the scalability and performance of enterprise Web applications. He has almost a decade of experience in software product development in India, Europe, and the US. He worked as a senior software engineer at Intarka Inc, a New Enterprise Associates (NEA)-funded company that was acquired by Update.com. He also worked for Wipro Ltd. He is currently an assistant professor at the College of Business, Florida International University. His research interests include the design and development of emerging technologies, electronic commerce, mining Web log data, building scalable e-business infrastructures, database systems, and data management. He has published articles in the *ACM Transactions on Database Systems*, *IEEE Transactions on Mobile Computing*, *Management Science*, *VLDB Journal*, and *INFORMS Journal on Computing*. He also has several publications in various IEEE conferences proceedings. He is a member of the IEEE and the IEEE Computer Society.

Anindya Datta received the BS degree from the Indian Institute of Technology, Kharagpur, and the MS and PhD degrees from the University of Maryland, College Park. He is a former faculty member of the Georgia Institute of Technology (Georgia Tech) and the University of Arizona. He advised large corporations such as AT&T, US West, IBM, the government of Israel, and a number of venture capital firms on high-performance database systems. He cofounded Chutney Technologies in 1999, a software company that develops solutions to improve the scalability and performance of enterprise Web applications, and was the chief executive officer (CEO) of the company. He founded the iXL e-Commerce Center, College of Management, Georgia Tech. He has also served as a visiting scientist at AT&T Laboratories. He is currently a visiting associate professor in the School of Information Systems, Singapore Management University. His research interests include database systems, data warehousing, data mining, and e-commerce. He has published more than 50 papers in various IEEE and ACM journals and is the holder of several patents in a variety of technologies. He is a member of the IEEE and the IEEE Computer Society.

Debra VanderMeer received the BA degree from Georgetown University, the MS degree in management information systems (MIS) from the University of Arizona, and the PhD degree from the Georgia Institute of Technology (Georgia Tech). She has served in engineering and managerial roles in large companies such as Tandem, as well as early-stage venture-funded software enterprises. She is currently an assistant professor in the College of Business, Florida International University. Her research interests focus on applying concepts developed in computer science and information systems to solve real-world problems. She has published widely in well-known journals such as *Management Science*, *ACM Transactions on Database Systems*, and *IEEE Transactions on Knowledge and Data Engineering*, as well as prestigious conference proceedings, including the International Conference on Data Engineering, International Conference on Distributed Computing Systems, and the Very Large Database Conference. She is a member of the IEEE and the IEEE Computer Society.

Helen Thomas received the BS degree in decision and information sciences from the University of Maryland, College Park, the MSE degree in operations research and industrial engineering from the University of Texas at Austin, and the PhD degree from the Georgia Institute of Technology (Georgia Tech). She previously served as an information systems faculty member at Carnegie Mellon University. She was also a cofounder and the director of product strategy at Chutney Technologies, a software company that develops solutions to improve the scalability and performance of enterprise Web applications. She has significant consulting experience, including several years of working with the American Management Systems. Her research interests include data management in e-commerce and decision support databases. She has published articles in well-known information systems journals such as the *Information Systems Research* and *Management Science*. She is a member of the IEEE and the IEEE Computer Society.



Krithi Ramamritham received the BS degree in electrical engineering and the MS degree in computer science from the Indian Institute of Technology (IIT), Madra, and the PhD degree in computer science from the University of Utah. Presently He is currently theVijay and Sita Vashee Chair Professor at the Department of Computer Science and Engineering, IIT, Bombay. He is also the Dean R&D at IIT, Bombay. His research explores timeliness and consistency issues in computer systems, in particular databases, real-time systems, and distributed applications. His recent work addresses these issues in the context of dynamic data in sensor networks, embedded systems, and mobile environments. During the last few years, he has been interested in the use of information and communication technologies for creating tools aimed at socioeconomic development. He is a coauthor of 5 books, 27 book chapters, 71 journal publications including various IEEE and ACM journals, and 156 conference proceedings. He is a fellow of the IEEE, the IEEE Computer Society, the ACM, and the Indian National Academy of Engineering.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.