ReDAL: Request Distribution for the Application Layer

Debra VanderMeer
Chutney Technologies
Atlanta, GA
deb@chutneytech.com

Helen Thomas
Carnegie Mellon University
Pittsburgh, PA
hthomas@andrew.cmu.edu

Kaushik Dutta
Florida International University
Miami, FL
kaushik.dutta@fiu.edu

Anindya Datta
Chutney Technologies
Atlanta, GA
adatta@chutneytech.com

Krithi Ramamritham
IIT-Mumbai
Mumbai, India
krithi@cse.iitb.ac.in

Abstract

Modern application infrastructures are based on clustered, multi-tiered 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 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 overall system throughput is enhanced, and load across the application servers is balanced. We compare the performance of our approach–with widely used industrial and recently proposed techniques from the literature– experimentally in terms of throughput and response time performance, as well as resource utilization. Our experimental results show a significant improvement of up to nearly 80% in both throughput and response time, with a very low additional cost in terms of CPU overheads, 0.7% to 1.5%, on the web server, and virtually no impact on CPU overheads on the application server.

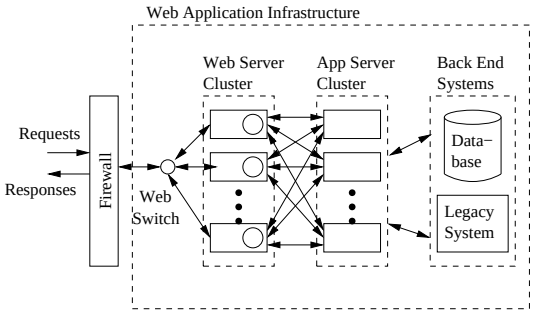

Figure 1. Typical Multi-Tiered Web Application Architecture

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 J2EE applications and the Common Language Runtime (CLR) for Microsoft .NET applications.

Modern application infrastructures are based on clustered, multi-tiered architectures. Figure 1 shows a typical such architecture for a web-based application. In Figure 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 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 [8]); serving application requests requires much more dynamic decision making than that required for web server requests.

Extensive literature dealing with the WSRD problem exists and significant commercial value has been realized from this work. WSRD approaches span both **content-blind** policies such as *random*, *round-robin* (RR), as well as **content-aware** policies such as *IP Address Routing* [2], *Least Loaded* (based on a metric called *server load index*) [2], *Least Connections* [3, 6], *Client Affinity* [7, 10], and *Session Affinity* [2]. Commercial Products such as Cisco's LocalDirector [3] and F5 Network's BIG/IP [6] are based on some of these approaches. Nearly all of the above-mentioned approaches are variants of the *weighted RR* (WRR) approach [10]. The only strategies, to the best of our knowledge, that are not WRR variants are the *Locality-Aware Request Distribution* (LARD) algorithm [13] and the Client/Session Affinity schemes, all 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 (e.g., cache sets on different web servers), while the affinity based schemes distribute requests to exploit the locality of session or state data.

The bulk of ASRD in practice is based on a combination of RR and *Session Affinity* routing schemes drawn directly from WSRD techniques (e.g., [1, 14, 12]). More specifically, the initial requests of sessions (e.g., the login request at an airline web site) are distributed in a RR fashion, while all subsequent requests are handled through *Session Affinity based schemes*, which route all requests in a particular session to the same application server. Session state, which stores information relevant to the interaction between the end user and the web site (e.g., 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. By routing requests to the application server "owning" the session, Client/Session Affinity routing schemes can avoid the overhead of repeated creation and destruction of session objects.

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 (e.g., [11, 5]) may be applicable here. While this is true 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 [15] presuppose some knowledge of either the task (e.g., duration, weight) or the resource (queue sizes, service times) or both. This assumption really does not work in our case, because both the tasks and the resources are highly dynamic. Moreoever, the scalability requirements of an 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 [8], in which the authors show that system resouce 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. While the authors make a strong 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, e.g., 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 doesn't appear to be a good candidate for use in ASRD scenarios in the research literature.*

1.1 Issues in Applying WSRD strategies to the Application Layer

It is important to understand why **WSRD strategies at the ASRD layer are sub-optimal, 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 [8]. 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 very well defined and quantifiable work, e.g., processing HTTP headers in packets and serving up static content. Application servers, on the other hand, run multi-layer 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.

Second, another significant 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% CPU would be considered "lightly loaded" (compared to one running, say at 50%) by most WSRD policies. While 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% CPU might be experiencing low CPU utilization simply because a bulk of its active threads are "blocked". In contrast, another application server in the same cluster running at 50% CPU may actually be less loaded, as it might possess a greater number of free threads. Note that while 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. 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 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 (e.g., [16, 4]). We illustrate this problem with the following example. 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, whichs last 3 minutes; or a *short session*, which lasts 1 minute. Suppose that the following sequence s of 10 sessions arrive to the cluster and are distributed to A_1 and A_2 according to the session affinity-RR policy (where new sessions are distributed according to RR, and all requests for an individual session are dispatched to the same server): $s_1, s_2, S_3, s_4, s_5, S_6, s_7, S_8, S_9, s_{10}$,

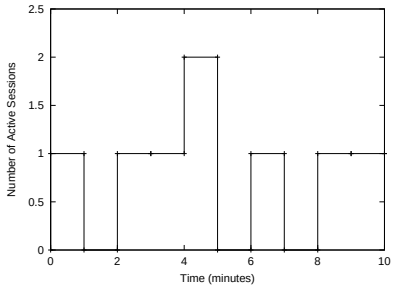

Figure 2. Load Distribution for App Server #1

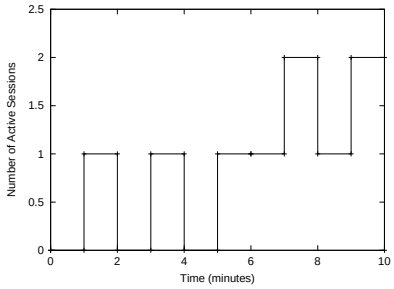

Figure 3. Load Distribution for App Server #2

where the interarrival time between new sessions is one minute. This policy results in the load distributions for A_1 and A_2 shown in Figures 2 and 3, respectively.

Both figures show load, in terms of the number of active sessions assigned, vs. time (in minutes). For example, during the time interval spanning (0,1), A_1 has 1 active session assigned (Figure 2) while A_2 has no active sessions assigned (Figure 3). The most interesting aspect of this example is the load imbalance created by Session Affinity. For example, during the time interval spanning (4,5), A_1 reaches maximum capacity (2 active sessions), while A_2 remains idle. A similar situation occurs during the (7,8) time interval, when A_2 reaches maximum capacity and A_1 remains idle. 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 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 to one or more application servers in the cluster, and *centralized session persistence*, in which session objects are stored in a centralized repository

(e.g., a 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 ReDAL approach attempts to optimize this tradeoff by servicing a request on the server instance where the session data resides unless a significant load imbalance situation is detected, in which case 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 then evaluate the performance of our proposed approach and compare it with that of existing ASRD policies experimentally, in Section 3. Finally, we conclude in Section 4.

2 The ReDAL Approach

In the *Request Distribution for the Application Layer* (ReDAL) approach, we characterize an application server as being in one of two states: (a) *lightly-loaded*, or (b) *heavily loaded*. We explain these characterizations using Figure 4 (adapted from [9]), 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, for which throughput remains relatively constant as load increases. However, the response time increases proportionally to 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. We can generate an approximation of the throughput curve at runtime

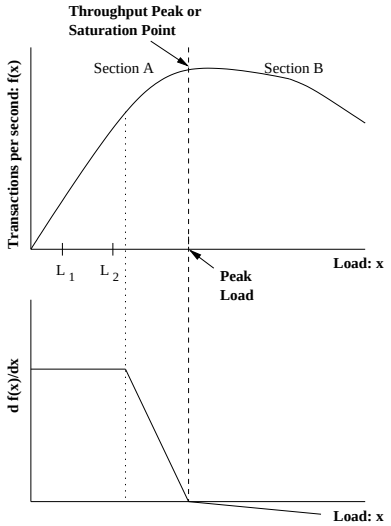

Figure 4. Typical Throughput Curve for an Application Server and its First Derivative

by gathering transactions per second data at a configurable interval. The lower portion of Figure 4 shows an approximation of the first derivative $\frac{df(x)}{dx}$ of the throughput curve $f(x)$ shown in the upper part of Figure 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 peak load. When $f(x)$ reaches 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.

With respect to Figure 4, we characterize a given application server as either *dispatchable* or *non-dispatchable*. A *dispatchable* application server corresponds to a lightly loaded server, while a *non-dispatchable* application server corresponds to a heavily loaded application server.

The goal of the ReDAL approach, intuitively, is to keep all application servers under its control under "acceptable" throughput thresholds, i.e., **the goal here is not to "balance" load per se, but rather to keep the cluster in a stable 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 the algorithm follows to achieve the goal of enhancing performance is as follows: at decision times,

i.e., when a request needs dispatching, it attempts to send the request to an *affined* dispatchable server (i.e., 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, the meat of the procedure lies in figuring out the load levels of servers, which is then mapped to dispatchability.

ReDAL follows a *capacity reservation* procedure to judge loading levels. As an example, 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 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 lookahead 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 is 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 app server A_k at time t, assuming that this request belongs to a session $\mathcal{S}$, we reserve a unit of capacity 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, assuming dispatching the request to the affined server is not possible, 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.

This is of course a very simplistic view. In reality, we have to account for various other issues, e.g., the fact that the current request may actually be the last request in a session (in which case the reservation we have made is actually an overestimation of the capacity required), as well as the fact that we may have misestimated think time for a particular request. The full ReDAL algorithm takes care of these practical issues.

2.1 System Architecture

The architecture of our proposed approach is similar to that shown in Figure 1. Our ReDAL request distribution system consists of a software component that runs within the web server (similar to the circles in Figure 1). Our system consists of two main logical modules: (i) the **Application Analyzer**, and (ii) the **Request Dispatcher**.

The **Application Analyzer** is responsible for characterizing the behavior of an application server as *dispatchable* or *non-dispatchable*. This module monitors each application server's throughput to generate a close approximation of the server's throughput curve $f(x)$, and designates a server as dispatchable if $\frac{df(x)}{dx}$ is positive, and non-dispatchable if $\frac{df(x)}{dx}$ is zero or negative.

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 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 Figure 1, we assume that there is a session virtualization mechanism (as described in Section 1) in place to enable session failover.[1]

2.2 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 (e.g., 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

[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.

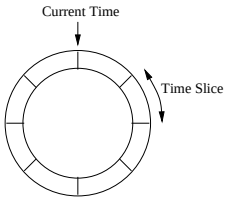

Figure 5. Cycle of Time Slices

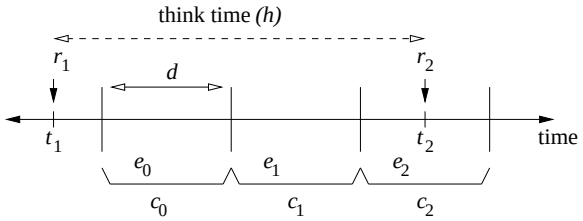

Figure 6. Load Metrics

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 j^{th} request in $\mathcal{S}$. A set of web servers $\mathcal{W} = \{W_1, W_2, \ldots, W_n\}$ dispatch 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 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 i^{th} time slice, and C allows sufficient time slices for reservations h seconds in the future, i.e., $C = \lceil \frac{h}{d} \rceil$. The C time slices are organized in a cycle of time slices for each application server, as shown in Figure 5. Each time slice will have an associated set of two load metrics, *actual load* and *expected load*, which are updated as new requests arrive and 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.)

The **Predicted time slice**. 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, i.e., 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. 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.

Figure 6 helps to illustrate how 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 Figure 6). Then e_2^k, the expected load for time slice c_2, is incremented by one. 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 (a) in a time slice prior to its predicted time slice, or (b) 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 Figure 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, while 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.

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).

The preceding discussion focused on the load metrics maintained by a single web server. In a multi-web server environment, 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 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, the new web server can retrieve the time of the last response and use it to compute think time.

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: **ReDAL-ALPHA=0.9** and **ReDAL-ALPHA=0.5** to show the impact of varying α[2]; 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 throughput performance compare across the* **RR**, **HJ** **ReDAL-ALPHA=0.5**, *and* **ReDAL-ALPHA=0.9** *algorithms?*, (2) *How does response time performance compare across the* **RR**, **HJ**, **ReDAL-ALPHA=0.5**, *and* **ReDAL-ALPHA=0.9** *algorithms?*, (3) *How do each of these policies impact CPU resource utilization on the web server?*, (4) *Does ReDal impact CPU on the application server?* and (5) *How is application server scaling affected by ReDal?*

[2]We have found that $\alpha = 1$ is effective only in the case where there is no error in 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.

3.1 Experimental Architecture

Our experiments were run using the general architecture described in Figure 1, with the addition of a load generation tool to simulate user requests, and a session clustering mechanism. This experimental environment consists of a load generator, LoadRunner v6 (www.mercuryinteractive.com), which simulates client requests; a single web server instance, the Apache HTTP Server v2.0 (www.apache.org); and ten application server instances, running WebLogic Server v7.1 (www.bea.com). In order to ensure session objects are available from every application server, our architecture requires some mechanism for clustering sessions (as discussed in Section 1). For this, our experimental architecture uses a relational database, running Oracle v8 (www.oracle.com); sessions are stored and retrieved from it through an implementation of the HttpSession object that connects to the database.

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 plugin, adding statefulness (not addressed in [8]) through calls to an external session object repository.

The application servers support two types of requests, *high.jsp* and *low.jsp*, which generate high and low load levels, respectively. These scripts are inspired by a simple human resources information system. *High.jsp* invokes a servlet that accesses an *employee_id* in the stored session object, and queries a database for the employee's salary, insurance, and vacation benefits. This operation is a multi-way join in the database. *Low.jsp* accesses the session object for the *employee_id* and returns the employee's name. This operation is a simple projection on a single table. On an unloaded (i.e., no other active requests) application instance, *high.jsp* returns a response in approximately 500 ms, while *low.jsp* returns a response in about 100

ms.

Sessions are stored to the external session repository as well as in the application server's memory space. If a request arrives at an application server for a locally-stored session object, read speed is dramatically reduced over the external retrieval case. If a request updates a session object that resides on another application server, an invalidation message is sent to the server where the object resides to remove the object. 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 from a uniform distribution across *high.jsp* and *low.jsp*.

The load generator, web server, application server instances, and external session repository all run on separate hardware. All machines are configured with a single-CPU (900 MHz), 1 GB RAM, and 20 GB disk, and run Windows 2000 Advanced Server (SP 3). All communication takes place on a local area network.

In these experiments, we measure three performance metrics: (1) *Average Throughput per Application Server* (ATAS) refers to the average number of transactions per second an application server in the cluster provides. We obtain this value by dividing the overall throughput rate provided by the cluster by the number of application servers in that cluster. (2) *Average Response Time* (ART) refers to the average request response time that the ten application servers can provide. Throughput of the cluster and Average Response Time are measured from the perspective of the end user. (3) *Web Server CPU Utilization* (WSCU) refers to the percentage CPU utilization on the web server, as measured by O/S utilities.

3.2 Throughput Performance

For our throughput experiments, Figure 7 shows how ATAS varies for **ReDAL-ALPHA=0.9**, **ReDAL-ALPHA=0.5**, **HJ**, and **RR** as the number of simultaneous sessions increases from 5 to 100.

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

We now consider each curve relative to one another. For the **ReDAL-ALPHA=0.5** curve, where throughput/server peaks at 80 simultaneous sessions with 50 transactions per second per server. **HJ** and **RR** do not perform as well as **ReDAL-ALPHA=0.5**, peaking at a lower simultaneous session load of 60 sessions and providing only 32 transactions per second per server in the **RR** case, and **HJ** peaking at 80 sessions, with 42 transactions per second. 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%) 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, **ReDAL-ALPHA=0.9** outperforms **ReDAL-ALPHA=0.5**, peaking at the same number of simultaneous sessions (80), but providing higher throughput, at 57 transactions per second per server. This shows the benefit of **ReDAL**'s reservation planning capability, which has greater impact as α is increased.

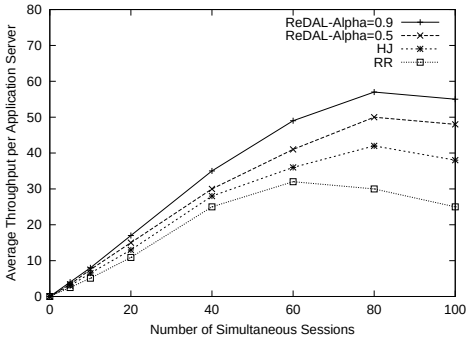

Figure 7. Average Throughput Per Application Server

3.3 Response Time Performance

Our response time experimental results, shown in Figure 8, show how ART varies for **ReDAL-ALPHA=0.5**, **ReDAL-ALPHA=0.9**, **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, 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

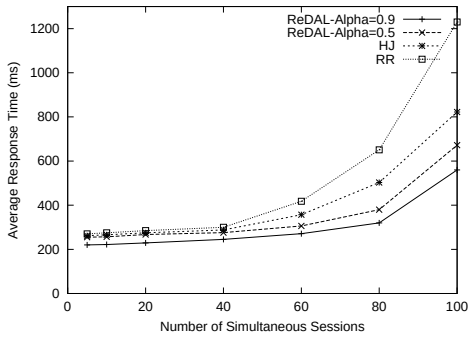

Figure 8. Average Response Time

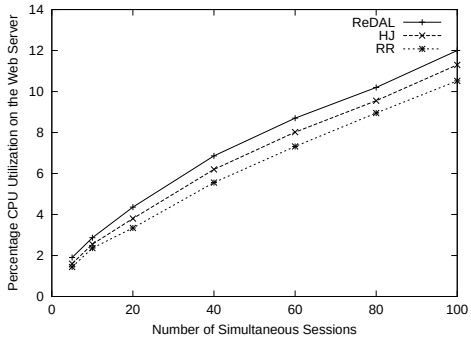

Figure 9. Average CPU Utilization on the Web Server

ATAS curves. Here, as the arrival rate of requests increases, 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 **ReDAL-ALPHA=0.5** curve, where response time begins to increase sharply at 80 simultaneous sessions with a response time of 380 ms. **RR** does not perform as well as **ReDAL-ALPHA=0.5**; here, response time begins to increase sharply at a lower simultaneous session load of 60 sessions, and providing a response time of 418 ms. This underscores the point made with regard to throughput – maintaining balanced load across the application server cluster provides significant benefit. For the **HJ** case, response time is also higher than the **ReDAL-ALPHA=0.5** case, i.e., 503 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, **ReDAL-ALPHA=0.9** outperforms **ReDAL-ALPHA=0.5**. While it begins to rise sharply at the same number of simultaneous sessions (80), it also provides a lower average response time of 320 ms. This reiterates our point regarding the benefits of **ReDAL**'s reservation mechanism.

3.4 CPU Overheads 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. Figure 9 shows shows how WSCU varies for **ReDAL**, **HJ**, and **RR** as the number of simultaneous sessions increases from 5 to 100. Here, we show only the results for $\alpha = 0.9$ for the **ReDAL** case, as the value of α does not impact the work required for **ReDAL**.

For all approaches, the WSCU curves are linear with a positive slope, i.e., CPU utilization increases with increasing simultaneous sessions. The **RR** approach shows the lowest overall WSCU, rising from 1.55% at 5 sessions to 10.52% at 100 sessions. The **HJ** case shows slightly higher values than **RR**, rising from 1.6% to 11.3%, 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 1.9% to 12.0%. 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 cost 1.5% additional CPU over **RR**, and 0.7% over the **HJ** case, 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, i.e., **HJ**, **RR**, and **ReDaL**, at each number of simultaneous sessions, we note the peak % CPU across ten application servers. We plot this peak % CPU vs. the number of simultaneous sessions in figure 10. Due to highly unbalanced load distribution, the peak 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% earlier than in the **ReDal** case. This is reflected in the increase in ART in figure 8.

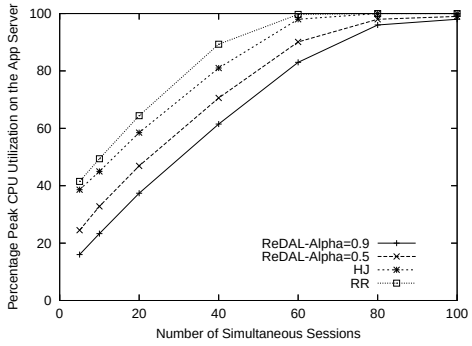

Figure 10. Peak % CPU on the Application Servers

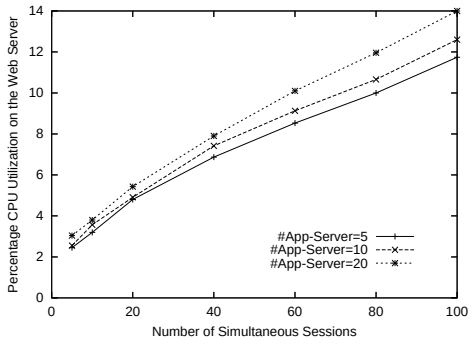

Figure 11. Scaling with Application Servers

3.6 Scaling with Additional Application Servers

Figure 11 shows CPU on the web server for simultaneous sessions increasing from 5 to 100, for 5, 10, and 20 application servers running behind the web server. 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 5, 10, and 20 application servers all show very similar CPU growth rates as simultaneous sessions increase, with the 5-server case showing slightly lower CPU usage than the 10-server case, and the 20-server case showing slightly higher CPU usage than the 10-server case. Clearly, increasing the number of application servers results in increased CPU usage on the web server, due to the increased complexity in tracking the load states of more servers; however, this increase is very small – the difference between the 10-server and 20-server cases is less than 2%.

4 Conclusion

We devise an approach for distributing requests across a cluster of application servers such that overall system throughput is enhanced, and load across the application servers is balanced. We compare the performance of our approach with widely used industrial and recently proposed techniques from the literature experimentally, in terms of throughput and response time performance, as well as resource utilization. Our experimental results show a significant improvement of up to nearly 80% in both throughput and response time, with a very low additional CPU cost, ranging from 0.7% to 1.5%.

References

[1] BEA. Weblogic server. www.bea.com.

[2] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server systems. Technical Report RC22209 (W0110-048), IBM Research Division, October 2001.

[3] Cisco. Localdirector. www.cisco.com.

[4] Cisco. The effects of distributing load randomly to servers. Cisco White Paper, 1997.

[5] M. Colajanni, P. Yu, and D. Dias. Scheduling algorithms for distributed web servers. In *Proc. ICDCS'97*, pages 169–176, May 1997.

[6] F5-Networks. BIG-IP. www.f5.com.

[7] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee. Network dispatcher: A connection router for scalable internet services. *Computer Networks*, 30(1-7):347–357, 1998.

[8] S. Hwang and N. Jung. Dynamic scheduling of web server cluster. In *Proc of the ICPDS*, 2002.

[9] IBM. Websphere application server. www.ibm.com.

[10] Linux-Virtual-Server-Project. Linux virtual server. www.linuxvirtualserver.org.

[11] D. Menasce, D. Saha, S. da Silva Porto, V. Almeida, and S. K. Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architecture. *Journal of Parallel and Distributed Computing*, 28(1):1–18, 1995.

[12] Microsoft. Internet information services (iis). www.microsoft.com/iis.

[13] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proc. of the 8th ASPLOS*, 1998.

[14] Sun. iplanet server. wwws.sun.com.

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

[16] V. Viswanathan. Load balancing web applications. http://www.onjava.com/pub/a/onjava/2001/09/26/load.html?page=1, September 2001.