Electronic Commerce Research and Applications 9 (2010) 549–561

Contents lists available at ScienceDirect

Electronic Commerce Research and Applications

journal homepage: www.elsevier.com/locate/ecra

RDRP: Reward-Driven Request Prioritization for e-Commerce web sites

Alexander Totok [a,*], Vijay Karamcheti [b]

[a] Google Inc., 76 9th Ave., 6th Floor, New York, NY 10011, USA
[b] Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 715 Broadway, 7th Floor, New York, NY 10003, USA

ARTICLE INFO

Article history:
Received 30 July 2009
Received in revised form 16 February 2010
Accepted 2 March 2010
Available online 6 March 2010

Keywords:
e-Commerce services
Profit maximization
Performance
Quality-of-Service
Request scheduling
Admission control
Bayesian inference analysis

ABSTRACT

Meeting client Quality-of-Service (QoS) expectations proves to be a difficult task for the providers of e-Commerce services, especially when web servers experience overload conditions, which cause increased response times and request rejections, leading to user frustration, lowered usage of the service and reduced revenues. In this paper, we propose a server-side request scheduling mechanism that addresses these problems. Our *Reward-Driven Request Prioritization* (RDRP) algorithm gives higher execution priority to client web sessions that are likely to bring more service *profit* (or any other application-specific *reward*). The method works by predicting future session structure by comparing its requests seen so far with aggregated information about recent client behavior, and using these predictions to preferentially allocate web server resources. Our experiments using the TPC-W benchmark application with an implementation of the RDRP techniques in the JBoss web application server show that RDRP can significantly boost profit attained by the service, while providing better QoS to clients that bring more profit.

1. Introduction

In recent decade, the role of the Internet has undergone a transition from simply being a data repository to one providing access to a variety of sophisticated Internet services, such as e-mail, shopping, social networking, and entertainment. Various e-Commerce services (e.g., online banking, shopping) constitute a significant portion of the services offered on the Internet. Typical interaction of users with such services is organized into *sessions*, a sequence of related requests, which together achieve a higher level user goal. An example of such interaction is an online shopping scenario for a retail e-Commerce web site, which involves multiple requests that search for particular products, retrieve information about a specific item, add it to the shopping cart, initiate the check-out process, and finally commit the order. The *success of the whole session* now becomes the ultimate QoS goal, which contrasts with the *per-request success* performance metrics of the early Internet.

Providers of e-Commerce services frequently have to deal with *service overload* conditions. In such situations, clients see increased response times and their requests (and the containing sessions) may get rejected, which leads to user frustration, and as a consequence, to lowered usage of the service and reduced service revenues (Barnes and Mookerjee 2009). Recent studies showed that 33% of shoppers on a slow-loading e-Commerce web site abandoned the site entirely, while 75% of visitors would never shop on that site again (Moskalyuk 2006).

Numerous server-side performance management techniques have been proposed to deal with server overload situations. For example, Session-based Admission Control (SBAC) (Cherkasova and Phaal 2002) admits only as many sessions as can be served by the service. More complex service differentiation mechanisms have also been used to provide stable QoS guarantees (e.g., request throughput, response times) to different client groups, based on prenegotiated Service-Level Agreements (SLAs). Common to such schemes is the consideration that QoS received by a client is determined upfront by his association with a client group or by his service membership status. However, most of these schemes fall short of delivering the best performance in situations where it makes sense to differentiate among clients based on the (*dynamic*) activities these clients perform in a session, rather than on their (*static*) identity, in order to boost service revenues, or for other application-specific goals. Let's consider the following two examples:

- In the online shopping scenario introduced earlier, the service provider might be interested in giving a higher execution priority to the sessions that have placed something in the shopping cart (potential *buyer* sessions), as compared to the sessions that just *browse* product catalogs, making sure that clients that buy something (and so – bring profit to the service) receive better QoS.
- For a service, some of whose web pages contain third party-sponsored advertisements, the service provider's profits may

* Corresponding author. Tel.: +1 646 678 4321; fax: +1 212 995 4123.
E-mail addresses: totok@google.com, aatotok@gmail.com (A. Totok), vijayk@cs.nyu.edu (V. Karamcheti).

1567-4223/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.elerap.2010.03.001

increase with more visits to these pages, as it may increase the chances of the clients following the advertisement links. Consequently, the service provider may wish to provide better QoS to the sessions that visit web pages with advertisements more often.

These examples are unified by the idea that the service may benefit from providing better QoS to sessions, that bring more *profit* (give more *reward*), where the notion of profit or reward is defined in an application-specific fashion. What is important is that the information about the client's possible usage of a service (and its associated contribution to service reward) is not encoded in any static profile, so application-logic-independent SLA-based service differentiation approaches are not as beneficial here.

Instead, to be able to provide better QoS to the sessions that bring more reward, the service provider now needs to predict the behavior of a client. If a client has used the service before and his identity can be determined (e.g., using cookies), than decisions on QoS provided to this client can be based on the history of his service usage (e.g., history of previous purchases). However, the success of this per-client *history-based approach*, is, not unexpectedly, highly dependent on the correlation between the past and the future behavior of a client, and may not work well if such a correlation is absent or weak.

Instead of focusing on individual client behavior, we advocate the approach of predicting a session's activities by associating it with aggregated client behavior or broader service usage patterns, obtained for example through online request profiling. Specifically, we propose *Reward-Driven Request Prioritization* (RDRP) mechanisms that try to maximize reward attained by the service, by dynamically assigning higher execution priority values to the requests whose sessions are likely to bring more reward. Our methods compare the sequence of a session's requests seen so far with aggregated information about client behaviors, and use a Bayesian inference analysis to statistically predict the future structure of a session, and so – the reward the session will bring and the execution cost it will incur. The predicted reward and execution cost values are used to compute each request's priority, which is used in scheduling "bottleneck" server resources, such as server threads and database connections, to incoming client requests.

We have implemented our proposed methods as a set of middleware mechanisms, which are seamlessly and modularly integrated in the open-source Java web application server JBoss (JBoss 2010). A *Request Profiling* module performs automatic real-time monitoring of client requests to extract parameters of service usage and to maintain the histories of session requests. It also performs fine-grained request profiling to identify execution times for different service request types. The *RDRP module* uses the information gathered by the Request Profiling module to compute and assign request priorities, that in turn influences queueing behavior for various application server resources.

We evaluate our approach on the TPC-W benchmark application (TPC-W 2005), emulating an e-Commerce web site selling books, and compare it with both the session-based admission control and per-client history-based approaches. Our experiments show that RDRP techniques yield benefits in both underload and overload situations, for both smooth and bursty client behavior. In underload situations, the proposed mechanisms give better response times for the clients that bring more profit to the service, thus helping to secure their satisfaction and future return to the web site. Note that it is often the case that the bulk of a service customers are returning clients, so providing good QoS to long-time customers is a key factor in service success (VanBoskirk et al. 2001, Barnes and Mookerjee 2009). In overload situations, when some of the requests get rejected, the mechanisms ensure that sessions that bring more profit are more likely to complete successfully and that the aggregate profit attained by the service increases compared to other solutions. Additionally, we show that the history-based approach matches the performance of our RDRP mechanisms on the amount of profit gained and response times only if the correlation between the clients' past and future behavior is 75% or greater, and 50% or greater, respectively.

The rest of this paper is organized as follows: Section 2 presents models and assumptions used throughout the paper. Section 3 describes the Reward-Driven Request Prioritization techniques. Section 4 presents our testing methodology and experimental results. In Section 5 we discuss related work, and we conclude in Section 6.

2. Models and assumptions

2.1. Web application server architecture

We work with e-Commerce services implemented on top of modern middleware platforms, such as the Java EE component framework (Java EE 2010). Such services are usually built as complex (and often distributed) software systems, consisting of several logical and physical tiers (e.g., web tier, application tier, and database tier) and accessing multiple backend data sources. We present our request prioritization algorithms in a centralized setting however, to focus on the benefits of the proposed request prioritization techniques. We expect that the methods will show their utility in a distributed setting as well, where they can be independently applied at every system resource contention point that sees concurrent requests competing for server resources.

Our work uses middleware-level mechanisms for server performance optimization, specifically control over request scheduling policies. We adopt this approach, because the middleware itself often does not have control over the low-level OS resources (e.g., CPU and memory), and uses higher-level mechanisms, such as request scheduling, component pool management, transaction demarcation, etc., to improve server performance. It is often the case that middleware server performance is limited by several "bottleneck" resources, that are *held exclusively* by a request for the whole duration or some significant portion of it (as opposed to low-level *shared* OS resources), such as server threads, or database (DB) connections. The default allocation policy of these resources to requests is FIFO. In the absence of application errors, failing to obtain such a resource is the major source of request rejection.

We advocate and use a request execution model, where a request is rejected (with an explicit message) if it fails to obtain a critical server resource within a specified time interval. This approach is shared by a vast majority of robust server architectures that bound request processing time in various ways (e.g., by setting a deadline for request completion), as opposed to a less robust approach, where a request is kept in the system indefinitely, until it is served (or is rejected by lower-level mechanisms such as TCP timeout). The former approach has the advantage of more efficiently freeing up server resources held by requests whose processing cannot be completed because of server capacity limitations.

Fig. 1 illustrates this application server architecture and the flow of a request through the system. Requests compete for two critical exclusively-held server resources: server threads and database (DB) connections; these resources are pooled by the web server and the application server respectively. Scheduling of requests to available threads and DB connections is done according to the request priority set by the RDRP module. The request with the highest priority is served first, with FIFO used as a tiebreaking policy. Timeout values for obtaining a thread and a DB connection are set to be 10 s. If this timeout expires, the request is rejected with an explicit message. Note that some requests do not require database access, so they can be successfully served just by acquiring a server thread.