Exploiting Service Usage Information for Optimizing Server Resource Management

ALEXANDER TOTOK, Google Inc.
VIJAY KARAMCHETI, New York University

It is often difficult to tune the performance of modern component-based Internet services because: (1) component middleware are complex software systems that expose several independently tuned server resource management mechanisms; (2) session-oriented client behavior with complex data access patterns makes it hard to predict what impact tuning these mechanisms has on application behavior; and (3) component-based Internet services themselves exhibit complex structural organization with requests of different types having widely ranging execution complexity. In this article we show that exposing and using detailed information about how clients use Internet services enables mechanisms that achieve two interconnected goals: (1) providing improved QoS to the service clients, and (2) optimizing server resource utilization. To differentiate among levels of service usage (service access) information, we introduce the notion of the *service access attribute* and identify four related groups of service access attributes, encompassing different aspects of service usage information, ranging from the high-level structure of client web sessions to low-level fine-grained information about utilization of server resources by different requests. To show how the identified service usage information can be collected, we implement a request profiling infrastructure in the JBoss Java application server. In the context of four representative service management problems, we show how collected service usage information is used to improve service performance, optimize server resource utilization, or to achieve other problem-specific service management goals.

Categories and Subject Descriptors: H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; *Commercial services*; K.6.4 [**Management of Computing and Information Systems**]: System Management—*Quality assurance*; K.6.2 [**Management of Computing and Information Systems**]: Installation Management—*Performance and usage measurement*; C.5.5 [**Computer System Implementation**]: Servers; C.4 [**Computer System Organization**]: Performance of Systems—*Modeling techniques*

General Terms: Design, Experimentation, Management, Performance

Additional Key Words and Phrases: Internet application, component middleware, quality-of-service, service usage information, client behavior, server resource management, optimization

ACM Reference Format:
Totok, A. and Karamcheti, V. 2011. Exploiting service usage information for optimizing server resource management. ACM Trans. Internet Technol. 11, 1, Article 1 (July 2011), 26 pages.
DOI = 10.1145/1993083.1993084 http://doi.acm.org/10.1145/1993083.1993084

1. INTRODUCTION

1.1 Motivation

In the last decade, the role of the Internet has undergone a transition from simply being a data repository to one providing access to a variety of network-accessible services

Authors' addresses: A. Totok, Google Inc., 76 9th Ave, 6th Floor, New York, NY 10011; email: totok@google.com; V. Karamcheti, Courant Institute of Mathematical Sciences, New York University, 715 Broadway, 7th Floor, New York, NY 10003; email: vijayk@cs.nyu.edu

DOI 10.1145/1993083.1993084 http://doi.acm.org/10.1145/1993083.1993084

such as e-mail, social networking, banking, shopping, and entertainment. The emergence of these web portals marked a shift from monolithically structured web sites with static read-only content, which were typical for the early Internet, to complex services that provide richer functionality, and dominate the modern Internet. These services share several commonalities in the way they are structured and the way they are used by their clients.

— *Session-oriented usage by clients.* The 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 an interaction is an online shopping scenario for an e-commerce web site, which involves multiple requests that search for particular products, retrieve information about a specific item (e.g., quantity and price), 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 client goal [Cherkasova and Phaal 2002], which contrasts with *per-request success* performance metrics of the early Internet.
— *Complex data access patterns.* Application data no longer has read-only access, as was typical for the older content-providing web sites. In scenarios such as the one just mentioned, certain service requests not only read but also write application data. Moreover, concurrent requests coming from different clients can access and modify shared application data. Data access patterns become even more complicated when the application data is replicated (e.g., for failover purposes) or partitioned (e.g., due to business requirements). The outcome of a request execution depends on the datasources it accesses, and may be influenced by concurrent user requests. For some services, it becomes crucial to preserve the *correctness* of a session's execution, w.r.t. the data it accesses.
— *Use of component middleware.* A growing number of services utilize *component middleware* such as the Java Platform Enterprise Edition (Java EE) framework [Java EE 2011] as their building platform. Such services are structured as aggregations of multiple application components communicating with each other and with the back-end databases, while the middleware provides commonly required support for communication, security, persistence, clustering, and transactions. Consequently, a web application's behavior depends not only on the way it is programmed, but also on the way it is assembled, deployed, and managed at runtime. As an example, changing middleware policies such as transaction demarcation may significantly impact not only application performance, but also certain aspects of application logic and the correctness of data presented to the users.

These characteristics have implications for how service providers ensure reasonable service quality for client requests. Providing good performance quality of service (QoS) has been a classic problem in the context of Internet services, and, the characteristics outlined above have made this problem even harder. Not only is providing performance guarantees more difficult for modern Internet services, but service providers now also need to take care of ensuring correctness of service (application) logic and providing expected data quality to the service clients.

1.1.1 Performance Quality. GVU WWW User Surveys [2001] showed that around 19% of the people surveyed attribute the bad experiences they had with Internet services to bad performance. The primary performance concern for service users is *request response time* [Barnes and Mookerjee 2009; Selvridge et al. 2001]. One study [Moskalyuk 2006] showed that 33% of shoppers on a slow-loading e-commerce web site abandoned the site entirely, and 75% of visitors would never shop on that site again.

The key contributing factor to long response delays is *service overload*, when the user load nears or exceeds server capacity, which causes request rejections and increases request response times, even in the absence of network disruptions. To improve service performance in the situation of service overload, service providers have traditionally used server-side resource management mechanisms to improve utilization of server resources. But this is harder to do for modern Internet services for the following reasons. First, middleware usually exposes several mechanisms that can be independently tuned in an attempt to improve application performance and optimize server resource utilization. However, these mechanisms do not provide a unique server configuration, which would be optimal for all request loads. Second, complex session-oriented client behavior makes it hard to predict what impact tuning a server resource management mechanism would have on server performance, which may vary for different incoming request mixes. Finally, component-based applications exhibit complex structural organization, where different sets of application components and middleware services are used to execute requests of different types. Some requests, for example, may need to access a back-end database, some may need CPU-intensive processing, while others may need exclusive access to a component or a critical resource.

The inability of service providers to predict the exact effects of using the server resource management mechanisms on service performance for a given client load accounts for the fact that these mechanisms are often used in an ad-hoc or "best-guess" manner. This results in suboptimal usage of server resources, not tailored for the specific incoming request load, and, as a consequence, the service clients do not get the best performance quality they could potentially get.

1.1.2 Service Logic and Data Quality. Clients of an Internet service expect that the service will operate according to the advertised functionality, present valid information, and correctly process and store the data submitted by its users. It is generally perceived that such correctness of the service (application) logic is solely the responsibility of the application developers. However, this may not hold true for component-based applications, where some functionality is delegated to the middleware. The behavior of the latter functionality is guided by configuration information provided by application assemblers, application deployers, and system administrators (e.g., through deployment descriptors and runtime server policies). This configuration information may significantly impact application behavior and the quality of data presented to its users; in the worst case, resulting in critically incorrect or abnormal application behavior.

An example of such undesirable service behavior is so-called "fare-jumping," when, during a shopping session, an item's price increases between the time a client first looks at it and the time he tries to checkout the order. One report showed that such an issue presents a problem for the e-commerce web sites selling airline tickets [Tedeschi 2005]. Such data quality problems stem from several facts. First, client requests read and write shared application data, potentially invalidating the data accessed by concurrent requests of different users. Second, the application data gets cached and replicated, which results in some requests returning out-of-date information.

A typical approach to cope with such problems is to retain as much control over data manipulation as possible in the application code. Another approach is to minimize the extent to which concurrent execution of user sessions is allowed. However, both of these approaches seem to be inadequate. First, contrary to the middleware paradigm, they place an unnatural burden on application developers and limit application modularity and reuse. Second, they worsen the service performance or restrict usage of the service by its clients. In order to provide reasonable service logic and data quality guarantees, while not limiting service performance, service providers need to

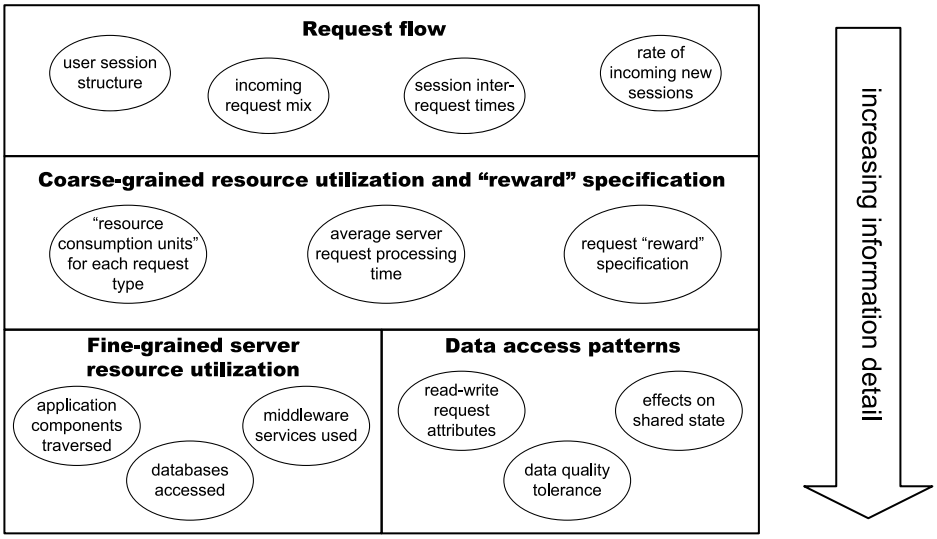

Fig. 1. Relationships between service access attributes.

understand how the data-quality-affecting server-side mechanisms they employ (e.g., transaction demarcation, data caching) impact the application logic and the quality of application data. On the other hand, the application developers need some guidelines for application development and structuring that would enable efficient use of the middleware mechanisms.

1.2 Approach and Methodology

In this work we focus on solutions that target the following three interconnected goals: (1) providing improved QoS guarantees to the clients of Internet services; (2) achieving optimal server resource utilization; and (3) providing application developers with the guidelines for natural application structuring, that enable efficient use of the proposed mechanisms for improving service performance. Specifically, we make the claim that exposing and using detailed information about how clients use component-based Internet services enables mechanisms that achieve the range of goals listed.

Service usage (or service access) information can be shown at different levels: from the high-level structure of user sessions, to low-level information about server resource consumption by different request types. Some of this information can be automatically obtained by request profiling, some can be obtained by statically analyzing the application structure, while others need to be specified by the service provider. To differentiate among various flavors of service usage information, we introduce the notion of the *service access attribute*, and identify the following four related groups of service access attributes, that correspond to different levels of service usage information. The relationship between the service access attributes is shown schematically in Figure 1. In Section 3 we describe the four service access attributes in greater detail.

— *Request flow.* This service access attribute provides the high-level information about the requests that are being invoked against the service. The information about an individual request is usually limited to its type, session (client) identity, and (optionally) the time of its arrival.
— *Coarse-grained resource utilization and "reward".* This service access attribute contains information about the high-level execution "cost" of requests of different types.

Table I. Service Access Attributes for Problems In this Study

	Request Flow	Coarse-Grained Resource Utilization	Fine-Grained Resource Utilization	Data Access Patterns
Problem 1: Maximizing reward	User session structures	Average request processing times and "rewards"		
Problem 2: Optimizing utilization of server resource pools	Incoming request mix (breakdown by request type)		Times spent in various stages of request processing	
Problem 3: Session data integrity	User session structures			The OP-COP-VALP model: specifies conflicting requests and data consistency constraints
Problem 4: Service distribution			Components invoked by different request types	Read-write behavior of requests

Service provider may also specify the so-called "reward" (or "profit") brought by each type of service request. This is an opportunity for providers of business-critical services to indicate which requests are more valuable according to the service business logic.

— *Fine-grained server resource utilization.* This service access attribute provides more detailed information about service requests of different types. It contains the information about how requests are processed in the application server. It may specify, for example, how long a request of a certain type spends in the database, on average, or which application components it accesses.

— *Data access patterns.* This service access attribute contains the information about how requests access application data. It may specify, for example, whether a request is read-only, read-write, or write-only. It may also specify what segments of application data are accessed by the request, and whether this data is shared among several clients or not.

To validate the claim that service usage information can be used to improve QoS guarantees and to better manage Internet services, we show its applicability to the following four problems: (1) maximizing reward brought by Internet services; (2) optimizing utilization of server resource pools; (3) providing session data integrity guarantees; and (4) enabling service distribution in wide-area environments. The problems were chosen to represent a wide range of challenges facing service providers in operating Internet services. In each problem we show how utilizing specific service access attributes helps to achieve the problem goal. Not all of the service access attributes are equally useful for all problems, which utilize different kinds of service usage information (see Table I) and exhibit various amounts of automatic exploitation of such information. Note also that the actual information specified in the service access attributes varies for different problems, QoS targets, and metrics being optimized.

In problems (1) and (2), we assume that the underlying computing infrastructure (hardware and software) is stochastic and that sampling enough service requests and measuring times spent in various stages of request execution is sufficient to reconstruct the broad picture of infrastructure behavior. The amount of time it takes to process a request depends on how loaded the underlying infrastructure is and on other factors such as delays in services provided by other machines, like the database server

or the JMS infrastructure. In the aforementioned two problems, we assume a dedicated infrastructure, for which the preceding assumptions indeed take place. Acknowledging that most modern Internet service deployments usually assume sharing of infrastructure resources and concurrent handling or requests from many other applications, in the future we plan to extend our work to shared and virtualized environments. Note also that in this study, we treat database as a black box and do not track database activities performed over database connections and in the database server. It is an interesting direction for future work to investigate how the techniques proposed in this article can be augmented and enhanced when profiling of the database server is added to the picture.

The solutions and techniques that we propose for each problem differ, but they span a representative range of mechanisms that researchers have proposed and used for predicting and improving performance of Internet services and server resource utilization. These mechanisms and techniques include mathematical modeling, statistical methods (event profiling and information gathering, Bayesian inference analysis), server resource management mechanisms (admission control, request prioritization and scheduling, concurrency control techniques), and application restructuring.

All of the proposed mechanisms, except for application restructuring, can be implemented in a modular and pluggable fashion as middleware services, which makes possible the voluntary use of such services that does not require changing the original application code of Internet services. To support this claim, we implemented and evaluated these mechanisms in the enterprise-level Java EE [Java EE 2011] application server JBoss [JBoss 2011]. The mechanisms showed their effectiveness, without bringing significant performance and management overheads. Although we show their utility in the context of the Java EE component middleware, we believe that the techniques and mechanisms described are general enough to be applicable to web servers utilizing other technologies.

Different parties involved in different stages of a component-based Internet service lifecycle could benefit from different aspects of the work in this article. Application developers could benefit from using the set of application design rules and optimizations for building component-based applications. Middleware architects and developers could benefit from utilizing the set of middleware mechanisms to introduce their functionality into the middleware systems. Service operators (e.g., system administrators) could benefit from using the models and techniques in order to boost performance of component-based Internet services and improve their manageability, given that these mechanisms and the corresponding functionality is provided by the underlying middleware.

The rest of the article is organized as follows. Section 2 provides necessary background information. Section 3 describes in greater detail the four identified service access attributes; while Section 4 talks about how this information is obtained with the implemented JBoss Request Profiling Infrastructure. The four problems chosen to validate the main claim made in this study are described separately in Sections 5, 6, 7, and 8. In each section we formulate the problem, present an overview of the proposed solution, mechanisms, and techniques, and point the reader to further publications dedicated to the problem. We conclude the article with Section 9.

2. BACKGROUND

2.1 Java EE Component Middleware

Among current day enterprise-level component middleware standards, Java Platform Enterprise Edition [Java EE 2011] is the most widely accepted and used component framework. Applications developed using the Java EE framework usually adhere

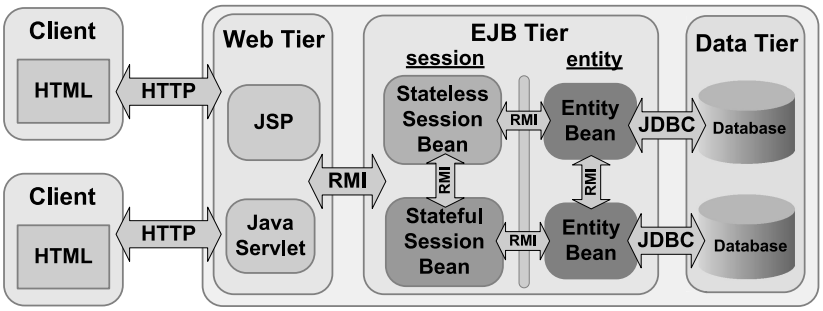

Fig. 2. Java EE component architecture.

to the classical 3-tier architecture, with *web tier*, *business tier*, and *data tier* (see Figure 2). Java EE components belonging to each tier are developed adhering to specific Java EE standards. Web tier deals with the presentation logic of the application. Components in this tier include Java Servlets and Java Server Pages (JSP) [Java EE Web 2011]. These components are invoked to process incoming HTTP requests, and are responsible for the generation of the response HTML pages, invoking components from the business tier or communicating directly with the data tier, to get application data from back-end datasources, if necessary. Business tier, sometimes called *middle tier* or, in the Java EE realm, *EJB tier*, consists of Enterprise Java Beans (EJB) components [EJB 2011], which have three flavors: *Session*, *Entity*, and *Message-Driven*. Session beans usually provide generic application-wide services, and also serve as façade objects in front of shared persistent datasources. Entity beans are transactional shared persistent entities, representing a synchronized in-memory copy of the database information. Message-driven beans are stateless components and serve the purpose of processing incoming asynchronous messages. Data tier serves the purpose of persistently storing the application data, and is usually represented by relational databases. Java EE application components usually communicate with relational datasources through JDBC (Java DataBase Connectivity) [JDBC 2011] interfaces. In this study, we work with the Java EE application server JBoss [JBoss 2011], augmented with the Jetty HTTP/web server [Jetty 2011].

2.2 Resource Consumption and Bottlenecks in Java EE Applications

There are several aspects of component middleware platforms in general, and of the Java EE component framework in particular, relevant to this study that influence performance of Internet applications built on these platforms.

— *Component invocations.* Component invocation is a relatively expensive operation compared to a plain Java object method invocation, especially in container implementations that use Java reflection mechanisms extensively (which is a common practice). CPU consumption of a service request depends on *how many* component invocations are involved in its execution and on the *types* of the invoked components. For example, invocation of a method on an entity EJB is typically more expensive than on a session bean (partly because of the need to synchronize an EJB's state with the database).
— *RMI serialization.* Early EJB container implementations used Java RMI (Remote Method Invocation) for all intercomponent communication, even for components residing within the same JVM. This approach imposes significant RMI serialization/deserialization overheads [Cecchet et al. 2002]. With the introduction of EJB local interfaces, it became possible to specify component *collocation*, which allows

EJB containers to avoid using RMI for intercomponent communication. To reduce the cost of marshalling, application servers usually provide communication optimizations for components residing in the same JVM (even if they don't use EJB local interfaces), by using local object references instead of going through RMI.

— *Communication with the database.* Accessing the database entails significant performance overhead of retrieving the data from disk, moving it to memory, and sending it over the network to the application server machine. Inefficiently structured JDBC code can significantly limit application performance. The most prominent example of this situation is synchronization of entity beans with the database, which, with inaccurate server configuration, may happen for every business method invoked on the entity bean [Cecchet et al. 2002]. CPU, memory, and network data transfers are much faster than reading and writing data from/to disk, so primary overhead in accessing the database is database disk I/O. Even if the data happens to be cached in memory on the database side, the overhead may still be significant. A common misconception is that the problem is bandwidth, while the problem is the CPU overhead for writing and reading an object's data to/from the wire. Modern application servers provide facilities to limit unnecessary entity bean database synchronization by, for example, updating the state of the bean only before the method call, if the call is read-only.

— *Contention for exclusively-held server resources.* Some of the server resources are shared among requests, while some are held exclusively by a request for the whole duration or a portion of it. Examples of the former include low-level OS resources, such as CPU and memory, while the latter are represented by such middleware resources as server threads and database connections. In the situation of server overload (static or transient), these resources become a source of request contention, with internal request queues building up. It is sometimes the case that the application performance is limited by such exclusively held "bottleneck" resources, even when there is enough CPU power to process more requests.

The application implementation method has a significant impact on application performance [Cecchet et al. 2002] as well. Usually, Java EE applications with session beans perform as well as Java servlets-only applications, and much better than most of the implementations based on entity beans. The fine-granularity access exposed by entity beans limits scalability, which, however, can be improved using session façade beans. For implementations using session façade beans, local communication cost is critically important, but EJB local interfaces (or application server optimizations substituting for the use of the latter) improve performance by avoiding the RMI communication layers for local communications.

2.3 Sample Java EE Applications

To test the techniques proposed in our work, we have used two sample Java EE applications. One of them, an implementation of the TPC-W benchmark application, was developed by ourselves, while the other, Java Pet Store, was developed elsewhere.

TPC-W application [TPC-W 2005] is a transactional web e-commerce benchmark, which emulates an online store that sells books. TPC-W specifies the application data structure and the functionality of the service; however it neither provides implementation nor limits implementation to any specific technology. The TPC-W specification describes in detail the 14 web invocations (requests types, in our terminology) that constitute the web site functionality, and defines how they change the application data stored in the database. A typical TPC-W session consists of the following requests: a user starts web site navigation by accessing the Home page; searches for particular products (Search); retrieves information about specific items (Item Details); adds some

of them to the shopping cart (Add To Cart); initiates the check-out process, registering and logging in as necessary (Register, Buy Request); and finally commits the order (Buy Confirm). We have developed our own implementation of the TPC-W benchmark, realized as a Java EE component-based application [TPC-W-NYU 2006]. The implementation utilizes the *Session Façade* design pattern [Marinescu 2002]. For each type of service request there is a separate servlet which, when necessary to generate the response HTML page, invokes business method(s) on an associated session bean(s), which, in turn, access application shared data stored in the database through a set of fine-grained invocations to the related entity EJBs.

Java Pet Store application [Java Pet Store 2006] is a well-known and widely adopted best-practices sample Java EE application. It represents an online store that sells pets. Java Pet Store aims at covering as much of the Java EE component platform as possible in a relatively small application. Its main focus is on design patterns and industry best practices that promote code and design reuse, extensibility, and modularity. Therefore, Java Pet Store is a relatively heavyweight application, compared to our TPC-W implementation. The fundamental design pattern used in Java Pet Store is the Model-View-Controller (MVC) architecture [Singh et al. 2002], which decouples the application's data structure, business logic, data presentation, and user interaction.

3. SERVICE ACCESS ATTRIBUTES

Information about service usage by clients can be exposed at different levels: from the high-level structure of incoming request flow, to low-level information about resource consumption and data access patterns of different request types. Some of this information can be automatically obtained by request profiling, some can be obtained by statically analyzing the application structure, while some information needs to be specified by the service provider. In our work, we identify four related groups of service access attributes, that correspond to different levels of service usage information. The relationship between different service access attributes, which are described in detail below, is shown schematically in Figure 1.

3.1 Request Flow

This service access attribute provides the high-level information about the requests that are being invoked against the service and represents the highest level of service usage information. The information about an individual service request is limited to its *type*, *session (client) identity*, and (optionally) *time of arrival*. *Request type* corresponds to the functionality of a request. It is often possible to determine the functionality of an HTTP request as well as its parameters by using the request's URL. For example, in the TPC-W application, HTTP request `http://tpcw.com/item?id=57` requests a web page with a detailed description of the item with `id` 57. Thus, the organization of HTTP requests makes it possible to infer a request's type and its parameters at the earliest stage of request execution—request preprocessing at the web tier—thus enabling collection of the request flow information through real-time profiling of incoming user requests in the web server.

Service usage information specified in the request flow service access attribute may come in different forms. For example, it may state the rate and the arrival pattern of the requests of certain types as they are received by the server. It may also describe the (typical) structure(s) of incoming user sessions. Different service management problems addressed in this article are sensitive to different aspects of the request flow information. In Table II we list the most important request flow properties that we are looking at in this work. In our work we distinguish between *request-oriented* service request flow specification, where session identity of requests is not important and is

Table II. Most Common Properties In the Request Flow Service Access Attribute

RATE	overall request rate
RATE_i	rate of requests of a particular type i
V_i	average number of requests of type i in a session
R_i	percentage of requests of type i among all requests
L_{av}	average session length (in requests)
T_{ir}	average session inter-request time
λ	rate of incoming new sessions
CBMG	structure of web sessions (Section *3.1.1)*

not taken into account, and *session-oriented* request flow specification, where session identity of requests is important and which usually comes with some information about web session structure(s). Note that models capturing the structure of request flow can be used to reproduce and simulate user activities, hence the problem of representation of request flow information is tightly coupled with the problem of web workload generation.

Request flow information usually contains various timing parameters describing the arrival patterns of service requests. In a request-oriented request flow specification, *overall request rate* RATE and *rate of requests of a particular type* RATE_i are usually of interest. In a session-oriented request flow specification, session interrequest times and arrival patterns of new sessions are the most commonly used timing parameters. It is usually assumed that session interrequest time is a random variable with a certain distribution, and its mean value T_{ir} (i.e., average session interrequest time) is one of the timing parameters of interest. For the arrivals of new user sessions, whatever this process looks like, we are interested in the average rate of incoming new sessions—λ, which reflects the intensity of the user load (Table II).

3.1.1 Web Session Structure Modeling. A user session consists of a sequence of service requests issued by a single user. These requests do not go in arbitrary order, because they adhere to the application logic of the service. In the realm of web-based services, the set of service requests that a user can make consists of the HTML links presented on the web page that was last displayed to the user; that is, it depends on the result of the previous request. Therefore, generally, session structure can be captured by a *state transition diagram*, where states denote results of service requests (web pages), and transitions denote possible service invocations. Whereas state transition diagrams producing possible web session structures can be arbitrarily complex, in most cases these diagrams have quite simple organization, or can be considerably simplified by only accounting for state transitions (and so certain session structures) that represent *typical* user behavior for a given Internet service. For example, session structures can often be represented by graphs, where the set of possible state transitions (possible service requests) depends only on the current state (previous request).

In this work we follow the adopted approach to model web session structures by state transition diagrams. Specifically, we adopt the classic *customer behavior model graph* approach [Menascé et al. 1999] to model user sessions. As originally proposed, the customer behavior model graph (CBMG) is a plain state transition graph, where the set of possible transitions (service requests) does not depend on the application (session or shared) state. In the CBMG model, state transitions are governed by transition probabilities $p_{i,j}$ of moving from state i to state j($\sum_{j=1}^{N} p_{i,j} = 1$, where N is the number of states in the CBMG. In our model we extend the classical CBMG model to also allow a finite number of finite-domain attributes for each state of the CBMG.

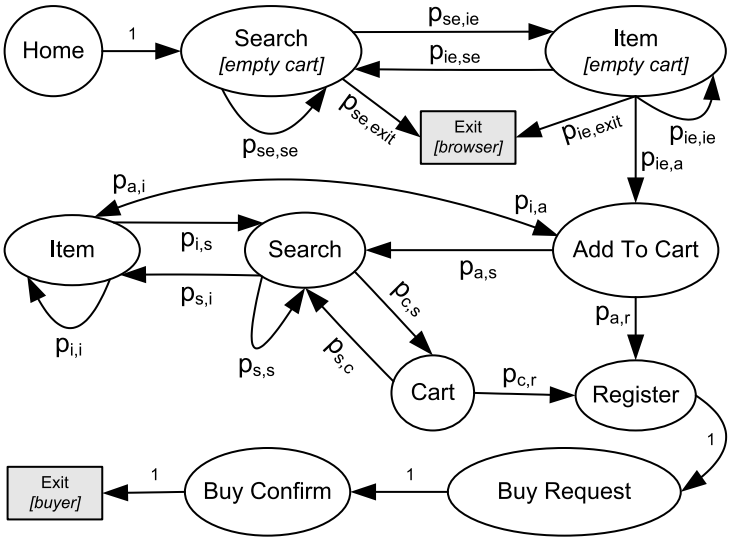

Fig. 3. CBMG of a sample TPC-W buyer session.

These attributes can be used to represent session state, that is, session events like signing-in and signing-out of an e-commerce Web site, or the number of items put into the shopping cart for an online store. The set of possible state transitions and their probabilities can in turn depend on the values of these attributes. Since the set of state attributes and their values is finite, each extended CBMG may be reduced to an equivalent CBMG, by duplicating states for each possible combination of state attribute values.

Figure 3 shows the CBMG of a sample *buyer* session for the TPC-W application (Section 2.3). This CBMG produces simplified user session structures, which use only a subset of the available TPC-W request types, but are rich enough to include essential application activities and represent requests with a wide range of functional and execution complexity. Each session modeled by this CBMG starts with the Home request, and may end either after several Search and Item Details (Item in short) requests (we refer to such sessions as *browser* sessions), or after putting one or more items in the shopping cart and completing the purchase (*buyer* session). This CBMG has one (boolean-valued) state attribute for the Search and Item states, which denotes the presence of items in the shopping cart, to model the assumption that once a user puts an item into the shopping cart, he never abandons the session and eventually commits the order. With such structure, this CBMG can be used to produce workloads that stress essential buyer activities. Other CBMGs can be used to model sessions that experience different behaviors.

It was shown in Menascé et al. [1999] that if one uses a mix of several CBMG session structures, then the resulting workload can approximate a given service request log as closely as desired by choosing the model parameters appropriately (i.e., the number of CBMGs and their transition probabilities), which can be obtained from the service request log by using a clustering algorithm. Following this approach, we would typically model a session-oriented web workload as one consisting of K CBMGs: CBMG_1, $\text{CBMG}_2, \ldots, \text{CBMG}_K$. The probability of a session having the structure of CBMG_k is p_k, $\sum_{k=1}^{K} p_k = 1$. For a session with structure CBMG_k, the probabilities of state transitions are denoted $p_{i,j}^k$.

Several studies analyzed web access logs obtained from web servers of organizations ranging from educational institutions to e-commerce web sites [Menascé et al. 2000; Cherkasova and Phaal 2002; Shi et al. 2002; Akula and Menascé 2007]. They report that session interrequest times usually have an exponential distribution, but have also been observed to have a lognormal or even a Pareto distribution. Following this finding, we model session interrequest (user think) times as either exponentially or log-normally distributed. It is generally believed that the times between *new session arrivals* are well modeled by an exponential distribution, which corresponds to a Poisson arrival process and fits well into the framework of classical Queueing theory [Kleinrock 1975]. The Poisson process produces a relatively smooth sequence of events, and fails to model the occasionally bursty traffic sometimes observed at web sites [Wang et al. 2002]. To better model the latter, we also use the B-model [Wang et al. 2002], which has been shown to produce synthetic traces with burstiness matching that of real web traffic.

The request flow parameters that we are looking at in this work (Table II) can be extracted from the parameters of a CBMG-based session-oriented web workload. The values of V_i, the average number of visits to state i, can be obtained by solving the following system of linear equations (this apparatus was originally developed in Menascé et al. [1999]):

$$\begin{cases} V_1 = 1 \\ V_i = \sum_{k=1}^{N} V_k \cdot p_{k,i} \quad \text{for all } i = 2, \ldots, N \end{cases} \tag{1}$$

where V_1 is the entry state (e.g., the Home request in the CBMG in Figure 3). The average session length is given by the equation

$$L_{\text{av}} = \sum_{i=1}^{N} V_i, \tag{2}$$

and the breakdown of requests by their type is given by

$$R_i = \frac{V_i}{L} \tag{3}$$

Finally, the overall request rate and request rates for specific request types are given by

$$\begin{aligned} \text{RATE} &= \lambda \cdot L \\ \text{RATE}_i &= \lambda \cdot V_i \quad \text{for } i = 1, \ldots, N \end{aligned} \tag{4}$$

In case the user load consists of several CBMGs, Eqs. (1) to (4) are generalized in a straightforward manner using the probabilities associated with different CBMGs as a weighting factor.

3.2 Coarse-Grained Resource Utilization and "Reward"

This service access attribute contains information about the high-level "cost" of executing requests of different types and the "profit" ("reward") that requests of different types bring to the service provider.

3.2.1 Coarse-Grained Resource Utilization. Requests of different types may exhibit different execution complexity and show different server resource consumption, since they tend to utilize different sets of application components and middleware services. Some requests, for example, may need to access a back-end database, while others may need CPU-intensive processing. Information about the coarse-grained "cost" of a

request execution can help in an approximate comparison of resource consumption by different requests. Processing times for individual requests in typical Internet services can vary widely by as much as two-to-four orders of magnitude. However, there tends to be much more variation *across request types* than for requests within the same type but with different request parameters [Chen et al. 2001; Elnikety et al. 2004]. Therefore, *request execution cost* is usually specified on the basis of request type. This cost can be specified by the service provider in the form of abstract *resource consumption units* (called *computational quantums* in Chen et al. [2001]). However, such static specification can be quite inaccurate for the following reasons. First, request execution times tend to depend on actual user load—request processing times under heavy load are much higher that those measured in isolation. Second, execution of complex SQL queries in the database, especially those involving merging and sorting, depend on the volume of the data processed, which may vary considerably during service lifetime. An alternative approach, which we adopt in this work, is to specify request execution cost as the average request processing time of the requests of a certain type. This approach is also attractive because it allows automated collecting and updating the required information through online request profiling in real time.

3.2.2 Request Reward. Service providers of business-critical services are interested in boosting service revenues. However, different user sessions make different contributions to the profit attained by the service. The specification of profit (or, generally speaking, "reward") brought by service requests is an opportunity for service providers to indicate which requests are more valuable according to the service logic, or to indicate which requests are crucial for the service. This information may be used by server-side resource management mechanisms to allocate server resources preferentially to requests. For example, in the online shopping scenario, 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. In another example, for Internet services whose web pages contain third-party-sponsored advertisements, the service provider's profits may (directly) depend on the number of visits to those pages. Consequently, the service provider may wish to provide better QoS to the sessions that visit these pages more often. It is the service provider's responsibility to define the reward function associated with the session. The model we adopt in this study is simple yet general enough to encompass several possible applications: a reward value is defined for every request type of the service. The reward of the session is the sum of rewards of the requests in the session.

3.3 Fine-Grained Server Resource Utilization

This service access attribute provides detailed information about how service requests are processed in the application server. The actual information about the way a request gets processed by the server may vary for different problems, QoS targets, and metrics being optimized. Here are the examples of information that could be specified in this service access attribute, and which could be useful in determining server resource utilization by service requests.

— Components invoked during a request's execution, their types and time spent in each component. Component invocation is a relatively expensive operation for the application server (Section 2.2), so a request's server resource consumption is directly affected by how many component invocations are involved in the execution of a service request.

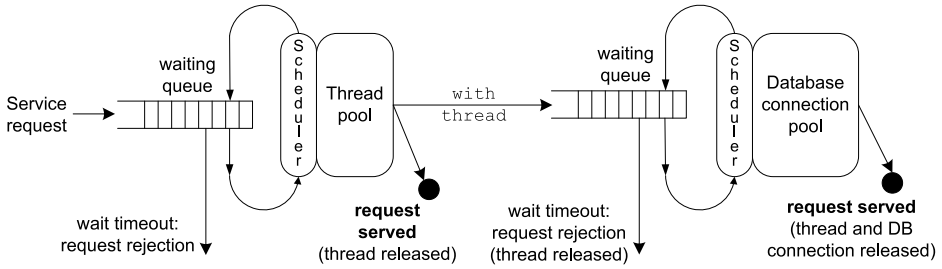

Fig. 4. Request execution model with 2-level exclusive resource holding threads and database connections.

— Databases accessed by a request and time spent processing SQL directives. Processing complex database queries is a major performance bottleneck in data-centric Internet services, hence this information can be used in assessing the execution complexity of a request.
— Communication with auxiliary middleware services, such as the JNDI naming service or the transaction manager service. This information may be used in assessing the request's resource consumption and in identifying middleware services that become performance bottlenecks.

The low-level information about how service requests are processed in the application server can be obtained through a fine-grained profiling of server-side request processing (see Section 4 for the description of our JBoss Request Profiling Infrastructure) by statically analyzing the application structure (application source code and deployment descriptors), or by a combination of both approaches.

3.3.1 Request Execution Model with 2-Level Exclusive Resource Holding. It is often the case that middleware performance is limited by several "bottleneck" resources that are held exclusively by a service request for the whole duration or some significant portion of it (such as server threads or database (DB) connections), as opposed to low-level shared OS resources. In the absence of application errors, failing to obtain such a resource is the major source of request rejection. In this work 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 not only guarantees that a request is either served within a time limit or unambiguously rejected, but also helps to more efficiently free server resources of the requests that cannot be handled due to the server capacity limitations.

Figure 4 schematically illustrates our model of request execution and the flow of a request through the system. Requests compete for two critical exclusively-held server resources: server threads and DB connections; these resources are pooled by the web application server. If the timeout value for obtaining a thread or a DB connection expires, the request is rejected with an explicit rejection message. An acquired database connection is cached, and is used exclusively by the request until it is processed (we leave the discussion of the rationale behind this to a separate publication [Totok and Karamcheti 2010a]). When the request is processed, the thread and cached database connections are returned to their respective pools. Note that some requests do not require access to database(s), so they can be served successfully just by acquiring a server thread. Note that Figure 4 shows a simplified picture where a request needs

at most one DB connection during its execution (e.g., if it accesses only a single database). In our request processing model, the request execution time can be represented as follows:

$$t = w^{\mathrm{THR}} + p + w_1^{\mathrm{DB}} + q_1 + \cdots + w_n^{\mathrm{DB}} + q_n \tag{5}$$

where w^{THR} is the time waiting for a thread; p, time doing request processing before getting DB connection(s); w_i^{DB}, time waiting for a i-th DB connection; q_i, time processing the request after the i-th DB connection was obtained by the request, but before $i+1$-th DB connection was requested from its respective pool, or the request completed its execution. This q_i time includes the time spent in making SQL queries, retrieving the results, processing them, and doing all other request processing while the DB connection is cached by the request. Note that in this work we treat the database as a black box.

The data in the request execution model with 2-level exclusive resource holding (e.g., the values of p and q_i) is an example of the actual service usage information specified in the *fine-grained server resource utilization* service access attribute. In Section 6 we will discuss specifically how in this model the knowledge of the values of p and q_i can be used to identify the optimal configuration of the thread and DB connection pools in a web application server environment.

3.4 Data Access Patterns

This service access attribute contains information about how service requests access application data. The information specified at this level varies for different problems, QoS targets, and metrics being optimized, but typically it would specify the read-write behavior of a request w.r.t. the data it accesses, and information about whether this data is shared among multiple users. This service access attribute may also specify, based on the needs of a specific service management problem, more detailed information, for example, what segment(s) of application data is (are) accessed and what the consequences of accessing this data are. It may also specify how tolerable a certain request is to application data quality (Section 1.1). This information may be used in managing data replication and caching, as well as scheduling the execution of concurrent client requests that access shared business-critical application data and that are critical to application data quality.

A concrete example of information specified in this service access attribute is the `OP-COP-VALP` model, a flexible model for specifying web session data consistency (integrity) constraints, which we introduced in a previous work [Totok and Karamcheti 2007]. In the `OP-COP-VALP` model, potential shared data conflicts are identified by specifying pairs of conflicting service requests (operations): OPERATION (`OP`, for short) and CONFLICTING OPERATION (`COP`, for short). For simplicity, one may think of `OP` and `COP` as READ and WRITE operating with associated conflict semantics. The model also provides ways to specify the tolerable data inconsistency. The `OP-COP-VALP` model then allows us to specify the situations when the middleware should validate a web session's correctness of execution (w.r.t. the defined conflicting operations). This is done by specifying that certain service requests represent VALIDATION POINTS (`VALP`, for short), which act like database checkpoints or commits. This model draws its ideas from several areas of advanced transaction processing, such as semantics-based concurrency control [Garcia-Molina 1983] and relaxed consistency [Wong and Agrawal 1992], but is specifically tailored for the case of concurrent web sessions. Note also that while a request's read-write data access patterns can be obtained through request profiling or static code analysis, such information as the tolerance to application data quality needs to be specified by the service provider, on the basis of some business

or QoS requirements, which are not "encoded" in the application structure or logic and cannot be inferred automatically.

4. REQUEST PROFILING INFRASTRUCTURE

Tracing user requests is a well-known and widely used technique in computer systems, utilized for various purposes, such as accounting, debugging, and performance analysis. The request logging feature is available in all mature web application servers, however, the information traced by the standard request logging functionality has a very limited scope. In order to be able to gather more fine-grained information about request execution, we need a request profiling infrastructure, such as the one used in Pinpoint [Chen et al. 2002] for problem determination and root-cause analysis in dynamic Internet services. To this end, we implemented our own request profiling infrastructure for the JBoss/Jetty web application server. The implementation takes advantage of the microkernel architecture of JBoss [Fleury and Reverbel 2003], and, overall, contributes to less than 1% of the server codebase. The infrastructure (as well as all other middleware mechanisms injected in JBoss) is implemented in a modular, extensible, and pluggable fashion. Where necessary, certain functionality modules are substituted with ones also augmented with the profiling execution hooks. Only absolutely necessary changes were made to the original JBoss/Jetty code, which are backward compatible with the original server configuration. We also show that performance overheads imposed by the infrastructure are rather small.

4.1 JBoss/Jetty Instrumentation

Various JBoss/Jetty modules are augmented with additional functionality and execution hooks to gather information about service request execution. While a request is being processed, all the information associated with it is kept in the local *Request Context*, associated with the request through a dedicated `ThreadLocal` Java object (a request is executed by a single thread). When the request completes, this data is sent to the *Request Profiling Service*, where it is added to a server-wide in-memory service usage information storage. Figure 5 schematically shows the architecture of the profiling infrastructure.

The *Request Profiling* middleware service acts as a centralized storage of information about completed service requests. The service keeps track of currently active sessions, as well as the aggregated information about recently completed service requests. The former is used to keep histories of session requests and inter-request times for currently active sessions, while the latter is used to extract various parameters of service usage from the history of recent requests executed against the service.

Request profiling in JBoss/Jetty is performed at all three Java EE tiers (Section 2.1). Request profiling at the *web tier* is performed by the modified Jetty HTTP/web server. It is used to gather high-level request flow information about incoming client requests, which are classified by their *type* (based on the URL pattern) and session affiliation. To inject profiling functionality, we substituted the default Jetty's *socket listener* module with an augmented socket listener implementation, which for each request creates a *request context* object and associates it with the request's thread. Profiling at the *EJB tier* is performed by adding two JBoss EJB interceptors [Marrs and Davis 2005; Fleury and Reverbel 2003], that is, the *client profiling interceptor* and *server profiling interceptor* (each at the client and the server sides correspondingly), which record in the request context the information about the EJB components and business methods invoked. The interceptors are also responsible for propagating the request context between the JVMs by putting it in the serializable part of the invocation object, which travels over the wire, in case of a remote invocation. We injected the profiling

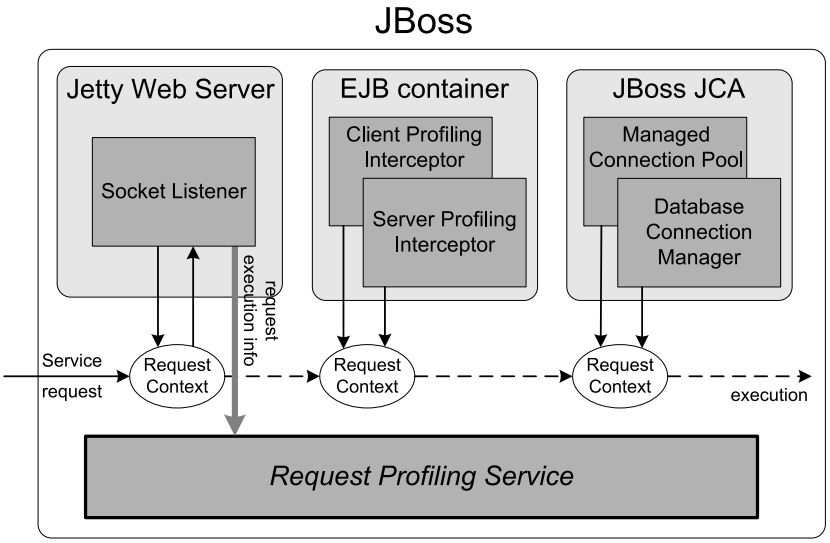

Fig. 5. Architecture of the JBoss/Jetty profiling infrastructure.

functionality into the *data tier* by modifying the *database connection manager* and *managed connection pool* modules. This allowed us to gather information about how database connections are assigned to requests and record various connection management events, for example, when connections are requested from the pool, granted, closed, and returned to the pool (Section 3.3.1). Note that we do not profile how specifically DB connections are used by requests (i.e., what JDBC queries are executed). Such, more detailed, information can only be obtained with additional profiling hooks injected into the database-specific JDBC driver code. However, information gathered by our profiling mechanisms is sufficient for the server resource management mechanisms that we propose for the problems described in this work.

4.2 Gathering and Analyzing the Service Usage Information

The Request Profiling middleware service not only gathers the information about recent service requests, but also provides mechanisms to manage this information and methods to extract parameters of service usage that we are interested in. To keep track of only *recent* service usage, we implement an information-gathering mechanism where events are stored in so-called *shifting epochs*. A currently *open* epoch records events (e.g., new session arrivals) either for a specified time interval or until it accumulates a certain number of events, after which this epoch *closes*, a new one opens and starts to record events, and the oldest epoch is discarded. This mechanism simplifies phasing out the aging epochs, imposes a limit on the memory used for storing the information, and also reduces data management overheads of recording an event (one does not need to discard the oldest event on every event arrival).

To extract various service usage parameters (e.g., average session inter-request time) we perform statistical analysis of the accumulated data. Each request for a parameter estimate indicates the number of recent epochs to be used for it, and each calculated parameter estimate is accompanied by the confidence interval with a confidence level value of 95%, computed using the Student's T-test [Roussas 1997]. The confidence interval contains the actual value that we are trying to estimate with a probability of 95%. Based on the specific problem at hand, a computed parameter

Table III. Comparative Performance of JBoss/Jetty Web Application Server Augmented with Profiling Infrastructure (**orig.**: original server, **prof.**: server with the profiling infrastructure)

User load	Average request response time (ms)		CPU utilization		Memory utilization (MB)	
(in λ)	**orig.**	**prof.**	**orig.**	**prof.**	**orig.**	**prof.**
$\lambda = 1$	40	52	9.7%	9.9%	104	125
$\lambda = 2$	61	75	25.9%	27.4%	109	135
$\lambda = 3$	83	104	32.0%	35.9%	114	143
$\lambda = 4$	127	201	40.3%	45.1%	142	164
$\lambda = 5$	187	320	57.4%	64.2%	154	173
$\lambda = 5.5$	670	n/a	68.8%	n/a	160	n/a

estimate can be deemed invalid if its confidence interval is larger than a predefined threshold (e.g., ± 0.01, or $\pm 10\%$ of the estimated value). In this case, the parameter estimate can be discarded or recomputed taking into account a greater number of epochs (and a greater number of events), which will likely decrease the computed confidence interval.

4.3 Performance Overheads of the Profiling Infrastructure

To evaluate the performance overheads that our profiling infrastructure imposes we conducted a series of experiments with two server configurations with the TPC-W application deployed on them: (1) original JBoss/Jetty application server and (2) JBoss/Jetty augmented with our profiling infrastructure. The server environment for these tests consisted of two dedicated workstations (one with JBoss/Jetty web application server, another with MySQL database server [MySQL 2011]), connected by a high-speed LAN. A separate workstation was used to produce artificial session-oriented user load, with different λ, the rate of new session arrivals (Section 3.1). In order to better evaluate the overheads of the profiling infrastructure, we used the TPC-W application configuration with the smallest database population size, and therefore with the highest sustainable request throughput. In the experiments, we measured average request response times and CPU and memory utilization. The latter two parameters were only measured for the JBoss/Jetty server, because MySQL server performance did not depend on the presence of the JBoss/Jetty profiling infrastructure. In both tested server configurations, the MySQL database server was the performance bottleneck. The results of the experiments are shown in Table III, which represents an example of comparative performance of our web application server infrastructure, with and without the implemented request profiling infrastructure.

The presence of the profiling infrastructure decreased the maximum sustainable session throughput, but only slightly: $\lambda = 5.5$ was the approximate maximum session throughput for the original server configuration, while $\lambda = 5$, for the server augmented with the profiling infrastructure (interval $[0 - 5.5]$ used in Table III therefore represents the operational range of our test server envronment as a function of user load, measured in λ). As the results show, CPU and memory overheads are small and are consistent for various user loads. Request response times for the server with the profiling infrastructure are only marginally higher if the server operates well below maximum sustainable user load. The overhead margin becomes higher as the load approaches server capacity, but under such a load the server shows deteriorating performance in any case, as request response times for both the original JBoss/Jetty server configuration and the one with the profiling infrastructure skyrocket.

5. PROBLEM 1: MAXIMIZING REWARD BROUGHT BY INTERNET SERVICES

In a typical setting, a web application server hosting an Internet service processes the incoming user request on a first-come-first-served basis. This approach provides fair access to the service for all clients. When a need emerges to provide some clients with a better service (e.g., based on their predefined customer status), the request scheduling and processing is governed by the service level agreements (SLA) or other analogous mechanisms that differentiate between different client groups. A common element in all these schemes is that QoS received by a client is determined upfront by its association with a client group.

While trying to provide its clients with reasonable or prenegotiated QoS, the service provider running a commercial service also wants to boost its revenues. Different user sessions bring different levels of profit to the service provider. For example, in the online shopping scenario introduced earlier, the service provider might be interested in giving a higher execution priority to the sessions that end up buying something buyer sessions), and a lower priority to the sessions that don't buy anything (browser sessions), making sure that clients that buy something receive better QoS. However, the information about user intentions to buy products is not encoded in its client group's profile, so SLA-based approaches are not as beneficial here.

To be able to provide better QoS to the sessions that bring more profit (reward), the service provider now needs to predict the behavior of a client. If the client is a returning customer and his identity can be determined (e.g., using cookies), then 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 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 via dynamically assigning higher execution priority values to the requests whose sessions are likely to bring more reward. Our RDRP algorithms work with the assumption that the following service usage information, (logically belonging to the *request flow* and *coarse-grained resource utilization* service access attributes) is available. This information is obtained by the request profiling infrastructure (Section 4) in real time. We assume that the user load (request flow) consists of K CBMGs: CBMG_1, $\text{CBMG}_2, \ldots,$ CBMG_K. The probability of a session having the CBMG_k structure is p_k, $\sum_{k=1}^{K} p_k = 1$ (Section 3.3.1). For each request type i, its execution cost, cost_i, is defined as the average execution time of requests of this type. We also assume that reward is defined for each request type. In the online shopping scenario, the profit of the service is reflected by the volume of items sold. So we define a reward function by assigning a reward value of 1 for the Add To Cart request: the shopping cart will contain as many items in it as the number of times the Add to Cart request was executed.

Given the above, the RDRP mechanism works in the following way. For every incoming request, it looks at the sequence of requests already seen in the session and compares this sequence with the known CBMG structures of the session types comprising the user load. A Bayesian inference analysis estimates the probability that the given session is of type CBMG_k, for each $k = 1, \ldots, K$ (step 1). For each session type CBMG_k, the algorithm computes the values of expected reward and execution cost, resulting from the future requests of the session, assuming it had the structure CBMG_k

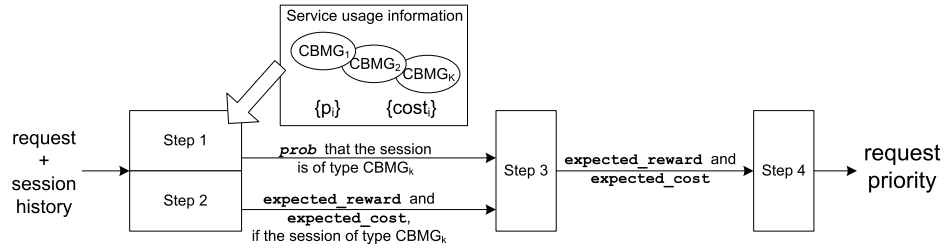

Fig. 6. Logical steps of the Reward-Driven Request Prioritization method.

(step 2). This information is used to get the nonconditional values of expected reward and execution costs of the future session's requests (step 3). We then define the *priority* of the request, as its session's reward (attained and expected) divided by the expected execution cost of the future session's requests (step 4). The assigned request priorities govern the scheduling of available server threads and DB connections to incoming requests. The logical sequence of the RDRP algorithm steps is depicted in Figure 6.

We implemented our proposed RDRP methods as a set of middleware mechanisms, which are seamlessly and modularly integrated in the web application server JBoss, and which make use of the information gathered by our request profiling infrastructure. We evaluated our approach on the TPC-W benchmark application, and compared it with both the session-based admission control [Cherkasova and Phaal 2002] and the aforementioned per-client history-based approach. 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 reward, which is crucial for ensuring return customers. This is important because it is often the case that the bulk of service customers are returning clients, so providing good QoS to long-time customers is a key factor in service success [Pecaut et al. 2000; VanBoskirk et al. 2001]. In overload situations, when some of the requests get rejected, the mechanisms ensure that sessions that bring more reward are more likely to complete successfully, and that the aggregate profit attained by the service increases compared to other solutions. Additionally, our experiments show that the history-based approach matches the performance of our RDRP mechanisms on the amount of reward attained and response times only if the correlation between the clients' past and future behavior is, respectively, 75% or greater and 50% or greater. Further details on the RDRP techniques, implemented middleware infrastructure, and the performance results can be found in another publication [Totok and Karamcheti 2010b].

6. PROBLEM 2: OPTIMIZING UTILIZATION OF SERVER RESOURCE POOLS

It is often the case that Internet service performance is limited by certain "bottleneck" server resources (such as server threads or database connections) that are held exclusively by a service request for the whole duration of its execution (or for some significant portion of it), as opposed to low-level *shared* OS resources (Section 2.2). In the absence of application errors, failing to obtain such a resource constitutes a major source of request rejections, especially in the situation of *server overload*. Optimizing utilization of web server threads and database connections proves to be a nontrivial task because for different client loads, different configurations of the thread and database connection pools provide the optimal performance. This happens because different sets of application components and middleware services are used to execute requests of different types. Some requests need to access a database (so they need to obtain and exclusively hold a database connection), while some do not.

To come up with a solution to this problem, we propose a methodology that computes the optimal number of threads and database connections for a given Internet application, its server and database environment, and specific user load (request mix). The methodology is built on the model of request execution with 2-tier exclusive resource holding (1st tier: threads, 2nd tier: database connections) which we introduced in Section 3.3.1. The methodology consists of the following steps.

(1) In the *first step*, a limited set of offline experiments are conducted, where the actual application (Internet service) and its server environment are subjected to an artificial user load. The user load is chosen to be representative of the actual (anticipated) user load, that is, it is close to the load that we expect the system will experience during its real-life operation. We use a different number of threads and database connections for each test run, and only a sparse subset of possible values for these resource pools is used throughout the experiments. During this series of such "profiling tests," information logically belonging to the *fine-grained server resource utilization* service access attribute is obtained with the use of the request profiling Infrastructure (Section 4). More specifically, we are interested in the average values of p and q_i for different requests types, that is, the time a request spends in the different stages of its execution (Section 3.3.1).
(2) In the *second step*, the values obtained for these timing parameters (considered as functions of the number of threads and database connections) are used as data points for function interpolation in order to get the values of these parameters for all possible combinations of the number of threads and database connections.
(3) The *third step*, unlike the first two, is performed in operating conditions and in real time, when the server environment is subjected to actual user load. The parameters of the incoming request mix—the values of RATE_i (Section 3.1)—are obtained through online request profiling by the request profiling infrastructure (Section 4). Then, special mathematical model of request execution takes as input the interpolated functions p and q_i from step 2 and the parameters of the actual incoming request flow and computes the number of threads and database connections that provides the best request throughput, thus achieving optimal utilization of web server threads and database connections.

If the user load changes but stays close enough to the load used in the "profiling tests" in step 1, the optimal number of threads and database connections for the new load can be recomputed in real time by just applying step 3 above, using the new values of incoming request mix (RATE_i) and the old functions p and q_i (from step 2), without the need to rerun the "profiling tests" of step 1 for the new request flow. This happens because dependence of functions p and q_i on the incoming request mix (RATE_i) is very weak, that is, their values change insignificantly when the request flow parameters RATE_i stay close enough to their initial values. This consideration enables *dynamic adaptation* of web application server environments to changing user load conditions.

We evaluated our methodology on the TPC-W benchmark application. We used our method to compute the optimal number of threads and database connections and the corresponding value of the maximum sustainable request throughput. We also determined these values experimentally, trying different sizes of the thread and database connection pools. Our results show that the proposed method is always able to accurately compute the optimal number of threads and database connections, and the value of the maximum sustainable request throughput computed by the method always lies within a 5% margin of the actual value determined experimentally. Further details on this methodology to compute the optimal number of web server threads and database

connections, the mathematical models behind it, and the performance results can be found in another publication [Totok and Karamcheti 2010a].

7. PROBLEM 3: SESSION DATA INTEGRITY GUARANTEES

This problem deals with the previously described situation of the need to ensure application data quality when multiple concurrent user sessions involve requests that read and write a shared application state and potentially invalidate each other's data (Section 1.1.2). Depending on the nature of the business represented by the service, allowing the session with invalid data to progress can lead to potential financial penalties incurred by the service (e.g., selling an item which has gone out of stock, or selling it at a lower price), while blocking the session's execution might result in user dissatisfaction. In the latter case, the session execution is deferred, and the case is relayed to customer service or awaits the intervention of system administrators. A compromise would be to tolerate some *bounded data inconsistency* [Wong and Agrawal 1992], denote it q (measured in some units, e.g., price or item quantity difference), which would allow more sessions to progress while limiting the potential financial loss to the service. The current dominant approach in web-based shopping systems is to satisfy the client at all costs, and never defer its session (which corresponds to tolerating $q = \infty$), but one could envision scenarios where imposing some limits on tolerable session data inconsistency (and so limiting the possible financial loss) at the expense of a small number of deferred sessions might be a more preferable alternative. Besides online shopping, examples of the systems where such tradeoffs might prove beneficial are online trading systems and auctions.

To enforce that the chosen degree of data consistency is preserved, the service can rely on various concurrency control algorithms. Several such algorithms (e.g., two-phase locking, optimistic validation at commit) have been developed in the context of classical database transaction theory. However, these algorithms need to be modified to be able to enforce session data consistency constraints due to the substantial differences between classical transactions and web sessions. To address this shortcoming, we came up with three versions of the classical concurrency control algorithms specifically tailored for web sessions: *optimistic validation*, *locking*, and *pessimistic admission control*. The algorithms work by rejecting the requests of the sessions for which they cannot provide data consistency guarantees (so these sessions become deferred). However, they utilize different strategies in doing so, which leads to different numbers of deferred sessions, which are not known to the service provider in advance. In order to meaningfully tradeoff having to defer some sessions for guaranteed bounded session data inconsistency, the service provider can benefit from models that predict metrics such as the percentage of successfully completed sessions, for a certain degree of tolerable data inconsistency (the value of q), based on service particulars and information about how clients use the service.

To this end, we propose analytical models that characterize execution of concurrent web sessions with bounded shared data inconsistency for the aforementioned three concurrency control algorithms. The models operate in an application-independent manner using the `OP-COP-VALP` data access model, which we introduced in Section 3.4. Mapping service requests to the operations in the `OP-COP-VALP` model is done by the service provider, who uses the information about application data access patterns to identify how the service requests access and changes the shared application state. To compute the metrics of interest (such as the percentage of successfully completed sessions), the proposed analytical models take as input request flow information obtained through real-time profiling of incoming client requests, performed by our Request Profiling Infrastructure (Section 4). To validate the models, we demostrated their applicability in the context of a sample TPC-W buyer scenario. We compared the results of

the analytical models with those of concurrent web session executions in a simulated and in the real JBoss web application server environment. The three sets of results closely matched each other, thus validating the models.

Besides allowing one to quantitatively reason about tradeoffs between the benefits of limiting tolerable session data inconsistency and the drawbacks of deferring some sessions to enforce data consistency, the models also permit comparison between concurrency control algorithms regarding the chosen metric of interest (e.g., the percentage of successfully completed sessions). In particular, since the proposed models use as input the service usage parameters that are easily obtained through profiling of incoming client requests, one can build an automated decision-making process as a part of the service or of its server environment (e.g., middleware platform), which would choose an appropriate concurrency control algorithm in real time in response to changing service usage patterns.

To test this claim we implemented such an automated decision-making infrastructure as a part of our JBoss web application server environment. Session data consistency is enforced by our infrastructure, which is capable of intercepting (and so rejecting, if need be) the service requests, and deciding which concurrency control method is the best to use, based on the analytical models and the parameters of service usage, obtained by the request profiling infrastructure. Our experiments show that the infrastructure is always able to pick the best algorithm. During a test run with the dynamic adaptation in place, the infrastructure achieved a higher value of the metric of interest (e.g., 75.6% of successful sessions) as compared to a scenario where the concurrency control algorithm is fixed (67.1% of successful sessions). Further details on the analytical models, the implemented automated decision-making infrastructure, and the experiment results can be found in another publication [Totok and Karamcheti 2007].

8. PROBLEM 4: SERVICE DISTRIBUTION IN WIDE-AREA ENVIRONMENTS

In the recent decade, application distribution and replication has become a noticeable trend in the way Internet services are designed and utilized. These techniques bring application data and data processing closer to remote clients and help cope on the network level, with the unpredictable nature of Internet traffic, especially in wide-area environments, and, on the application level, with high-volume, widely varying, disparate client workloads. Examples of this approach vary, from old-fashioned web caching of static content to web content delivery using content-distribution networks (CDN), to distributed (edge) service deployment [Akkerman et al. 2005; Gao et al. 2003, 2005].

Internet services built as component-based applications are natural candidates for service distribution, because component frameworks offer mechanisms enabling distributed application deployments [Akkerman et al. 2005]. Despite their nominal suitability, component-based applications are traditionally deployed only in a centralized fashion in high-performance local area networks. In the rare cases when these applications are distributed in wide-area environments, the systems tend to be highly customized and handcrafted. When a component-based application is distributed in wide-area environments, intercomponent communication, otherwise "invisible" in local area networks, results in dramatically increased request response times, whose impact on overall service performance depends on what application components and back-end datasources are accessed during a service request's execution. Information of this kind belongs to the fine-grained resource utilization service access attribute, and needs to be available for the service provider to be able to assess the performance quality of the application being distributed.

On the other hand, in order to ensure that business-critical service requests experience small response delays, the application should be engineered in a way that limits unnecessary wide-area intercomponent communication. To achieve this, the application developer needs to be aware of (1) the "read-write" data access behavior of service requests; and (2) whether or not the application state accessed in a request is shared among several clients. In other words, while developing the application, the developer needs to take into account the application's data access patterns. Our approach to enabling beneficial and efficient distribution of component-based applications in wide-area environments is to (1) take into account the information about read-write shared data access patterns and fine-grained resource utilization by service requests of different types and (2) based on this information, provide guidelines for application (re)structuring, which limits wide-area intercomponent communication. To this end, we identified and recommended a small set of design rules and optimizations for application structuring that enable distribution of component-based applications: (1) the remote façade design pattern; (2) stateful component caching; (3) query caching; and (4) asynchronous updates.

We validated the applicability of these design rules by applying them to the Java Pet Store sample Java EE application (Section 2.3). We deployed Java Pet Store in a fixed, simulated wide-area environment, applied the identified design patterns and optimizations in different combinations, and measured the performance of the application. Configuration with all the applied design patterns achieved the best overall performance and scalability by accumulating all improvements. The remote façade design pattern, in which collections of related entity EJB components accessed by a single service request are wrapped with a thin layer of façade objects [Marinescu 2002], avoids redundant wide-area communication, because requests from remote clients, who have access only to a façade object, delegate their execution in just one network call to the remote façade, which in turn performs multiple fine-grained local calls against collocated EJB components. The use of the remote façade pattern is required if communicating components are separated by a wide-area network, regardless of the nature of user requests served by these components. The use of this pattern also helps to implicitly define the optimal application partitioning granularity. Read-only entity EJBs, which implement the stateful component caching design pattern and query caches deployed in edge servers absorb the load generated by remote clients and save expensive trips to centralized datasources. Asynchronous propagation of updates achieves scalability and guarantees that updaters are not penalized by blocking on write operations. The overall effect of applied design patterns and optimizations is two-fold. First and foremost, remote clients are almost completely insulated from wide-area effects. In the few cases when remote clients incur wide-area intercomponent calls, the communication overhead is as small as possible due to the remote façade design pattern. Second, both local and remote clients experience improved performance due to aggressive caching of stateful components. Further details about the design patterns and optimizations, their implementation, and the experiment results can be found in Llambiri et al. [2003].

9. CONCLUSION

In this article we looked at several performance and service management issues encountered in modern component-based Internet services. We showed that exposing and using detailed information about how clients use Internet services enables mechanisms that achieve the two interconnected goals: (1) providing improved QoS to the service clients, and (2) optimizing server resource utilization. To differentiate among different levels of service usage (service access) information, we introduced the notion of the service access attribute and identified four related groups of service access

attributes, which correspond to different levels of service usage information, ranging from high-level structure of client web sessions to low-level fine-grained information about utilization of server resources by different requests. We showed how the identified service usage information can be collected; for this we implemented a request profiling infrastructure in the JBoss Java EE application server. In the context of four representative service management problems: (1) maximizing reward brought by Internet services; (2) optimizing utilization of server resource pools; (3) providing session data integrity guarantees; and (4) enabling service distribution in wide-area environments, we showed how collected service usage information is used to improve service performance, to optimize server resource utilization or to achieve other problem-specific service management goals.

REFERENCES

AKKERMAN, A., TOTOK, A., AND KARAMCHETI, V. 2005. Infrastructure for automatic dynamic deployment of J2EE applications in distributed environments. In *Proceedings of the 3rd International Working Conference on Component Deployment (CD'05)*. Lecture Notes in Computer Science, vol. 3798, Springer, Berlin.

AKULA, V. AND MENASCÉ, D. 2007. Two-level workload characterization of online auctions. *Electron. Commerce Res. Appl. 6*, 2, 192–208.

BARNES, D. AND MOOKERJEE, V. 2009. Customer delay in e-commerce sites: Design and strategic implications. In *Business Computing, Handbooks in Information Systems*, vol. 3, G. Adomavicius and A. Gupta Eds., Emerald Group Publishing, Bradford, England, 74–85.

CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. 2002. Performance and scalability of EJB applications. *ACM SIGPLAN Not. 37*, 11, ACM, New York.

CHEN, M., KICIMAN, E., FRATKIN, E., BREWER, E., AND FOX, A. 2002. Pinpoint: Problem determination in large, dynamic, Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*. IEEE Computer Society, Los Alamitos, CA.

CHEN, X., MOHAPATRA, P., AND CHEN, H. 2001. An admission control scheme for predictable server response time for web accesses. In *Proceedings of the International World WideWeb Conference (WWW'01)*. ACM, New York.

CHERKASOVA, L. AND PHAAL, P. 2002. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Trans. Computers 51*, 6, 669–685.

EJB. 2011. Enterprise JavaBeans Technology. http://www.oracle.com/technetwork/java/index-jsp-140203.html.

ELNIKETY, S., NAHUM, E., TRACEY, J., AND ZWAENEPOEL, W. 2004. A method for transparent admission control and request scheduling in dynamic e-commerce web sites. In *Proceedings of the International World Wide Web Conference (WWW'04)*. ACM, New York.

FLEURY, M. AND REVERBEL, F. 2003. The JBoss extensible server. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*. Lecture Notes in Computer Science, vol. 2672, Springer, Berlin.

GAO, L., DAHLIN, M., NAYATE, A., ZHENG, J., AND IYENGAR, A. 2003. Application specific data replication for edge services. In *Proceedings of the International World Wide Web Conference (WWW'03)*. ACM, New York.

GAO, L., DAHLIN, M., ZHENG, J., ALVISI, L., AND IYENGAR, A. 2005. Dual-quorum replication for edge services. In *Proceedings of the 6th ACM/IFIP/USENIX International Middleware Conference*. Lecture Notes in Computer Science, vol. 3790, Springer, Berlin

GARCIA-MOLINA, H. 1983. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Datab. Syst. 8*, 2, 186–213.

GVU WWW USER SURVEYS. 2001. Georgia Institute of Technology. Graphics, Visualization and Usability (GVU) Research Center. http://www.gvu.gatech.edu/user_surveys/.

JAVA EE. 2011. Java Platform Enterprise. http://www.oracle.com/technetwork/java/javaee/.

JAVA EE WEB. 2011. Java EE web application technologies. http://www.oracle.com/technetwork/java/javaee/tech/webapps-138511.html.

JAVA PET STORE. 2006. Sample Java EE application. http://java.sun.com/developer/releases/petstore/.

JBOSS. 2011. JBoss Java application server. http://www.jboss.org.

JDBC. 2011. Java database connectivity technology. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html.

JETTY. 2011. HTTP server and servlet container. http://jetty.codehaus.org/jetty/.

KLEINROCK, L. 1975. *Queueing Systems.* Wiley, Hoboken, NJ.

LLAMBIRI, D., TOTOK, A., AND KARAMCHETI, V. 2003. Efficiently distributing component-based applications across wide-area environments. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*. IEEE, Los Alamitos, CA.

MARINESCU, F. 2002. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms.* Wiley, Hoboken, NJ.

MARRS, T. AND DAVIS, S. 2005. *JBoss at Work: A Practical Guide*. O'Reilly Media, Sebastopol, CA.

MENASCÉ, D., ALMEIDA, V., FONSECA, R., AND MENDES, M. 1999. A methodology for workload characterization of e-commerce sites. In *Proceedings of the 1st ACM Conference on Electronic Commerce (EC'99)*. ACM, New York.

MENASCÉ, D., ALMEIDA, V., RIEDI, R., RIBEIRO, F., FONSECA, R., AND MEIRA, W. 2000. In search of invariants for e-business workloads. In *Proceedings of the 2nd ACM Conference on Electronic Commerce (EC'00)*. ACM, New York.

MOSKALYUK, A. 2006. IT Facts: e-commerce research blog on ZDNet.com, Nov. 2006. http://blogs.zdnet.com/ITFacts/?p=12030.

MYSQL. 2011. MySQL Database. http://www.mysql.com/.

PECAUT, D., SILVERSTEIN, M., AND STANGER, P. 2000. Winning the online consumer: Insights into online consumer behavior, Boston Consulting Group. http://www.bcg.com.

ROUSSAS, G. 1997. *A Course in Mathematical Statistics*. Academic Press, Amsterdam.

SELVRIDGE, P., CHAPARRO, B., AND BENDER, G. 2001. The world wide wait: Effects of delays on user performance. *Int. J. Industrial Ergonomics 29*, 1, 15–20.

SHI, W., WRIGHT, R., COLLINS, E., AND KARAMCHETI, V. 2002. Workload characterization of a personalized Web site – and its implications for dynamic content caching. In *Proceedings of the 7th International Workshop on Web Caching and Content Distribution (WCW'02)*. IWCW, Boulder, CO.

SINGH, I., STEARNS, B., JOHNSON, M., AND THE ENTERPRISE TEAM. 2002. *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley, London.

TEDESCHI, B. 2005. Glitches in booking first class online. *The New York Times* (4/10/05), Travel Section, 6.

TOTOK, A. AND KARAMCHETI, V. 2007. Modeling of concurrent web sessions with bounded inconsistency in shared data. *J. Parall. Distrib. Comput. 67*, 7, 830–847.

TOTOK, A. AND KARAMCHETI, V. 2010a. Optimizing utilization of resource pools in web application servers. *Concurrency Comput: Pract. Exper. 22*, 18, 2421–2444.

TOTOK, A. AND KARAMCHETI, V. 2010b. RDRP: Reward-driven request prioritization for e-commerce web sites. *Electron. Commerce Res. Appl. 9*, 6, 549–561.

TPC-W. 2005. Transaction Processing Performance Council. Transactional web e-commerce benchmark. http://www.tpc.org/tpcw/.

TPC-W-NYU. 2006. A Java EE implementation of the TPC-W benchmark. http://www.cs.nyu.edu/totok/professional/software/tpcw/tpcw.html.

VANBOSKIRK, S., LI, C., AND PARR, J. 2001. Keeping customers loyal. Forrester Research, May. http://www.forrester.com.

WANG, M., CHAN, N., PAPADIMITRIOU, S., FALOUTSOS, C., AND MADHYASTHA, T. 2002. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*. IEEE, Los Alamitos, CA.

WONG, M. H. AND AGRAWAL, D. 1992. Tolerating bounded inconsistency for increasing concurrency in database systems. In *Proceedings of the 11th Symposium on Principles of Database Systems (PODS'92)*. ACM, New York.

Received July 2009; revised July 2010; accepted February 2011