# Modeling Of Concurrent Web Sessions With Bounded Inconsistency In Shared Data

CIMS Technical Report: TR2005-868

Alexander Totok and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY, USA
{totok,vijayk}@cs.nyu.edu

**Abstract**

Client interactions with modern web-accessible network services are typically organized into *sessions* involving multiple requests that read and write *shared* application data. Therefore when executed concurrently, web sessions may invalidate each other's data. Depending on the nature of the business represented by the service, allowing the session with invalid data to progress might lead to financial penalties for the service provider, while blocking the session's progress and deferring its execution (e.g., by relaying its handling to the customer service) will most probably result in user dissatisfaction. A compromise would be to tolerate some *bounded* data inconsistency, which would allow most of the sessions to progress, while limiting the potential financial loss incurred by the service. In order to quantitatively reason about these tradeoffs, 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.

This paper develops such analytical models of concurrent web sessions with bounded inconsistency in shared data for three popular concurrency control algorithms. We illustrate our models using the sample *buyer* scenario from the TPC-W e-Commerce benchmark, and validate them by showing their close correspondence to measured results of concurrent session execution in both a simulated and a real web server environment. Our models take as input parameters of service usage, which can be obtained through profiling of incoming client requests. We augment our web application server environment with a profiling and automated decision making infrastructure which is shown to successfully choose, based on the specified performance metric, the best concurrency control algorithm in real time in response to changing service usage patterns.

## Keywords

Web-based network services, e-Commerce, client sessions with bounded inconsistency in shared data, concurrency control, analytic modeling, queueing theory, dynamic adaptation, middleware

## 1 Introduction

The Internet currently provides access to a variety of services such as e-mail, banking, on-line shopping, and entertainment. Typical interaction of users with such services is organized into *sessions*, a sequence of related requests, which together achieve a higher level user goal. An example of such interaction is an on-line shopping scenario for an e-Commerce web site, which involves multiple requests that (1) search for particular products, (2) retrieve information about a specific item (e.g., quantity and price), (3) add it to the shopping cart, (4) initiate the check-out process, and (5) finally commit the order.

In scenarios of this kind, session requests can both *read* and *write* application data *shared* among several users of the service. Thus, execution of concurrent client sessions may affect each other by changing the shared application state. In our example, the client's decision to commit the order (buying an item in step 5) is based on the information

presented in step 2. Thus, if the quantity or price of the item has changed (as a result of concurrent client activities), it might be undesirable to allow the client to commit the order (step 5) based on incorrect information. At this point the service provider needs to make a decision of whether to proceed with the execution of the request: 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 become out of stock, or selling it at a lower price), while blocking the session's execution might result in user dissatisfaction and can lead to a drop in user loyalty. In the latter case the session execution is *deferred*, and handling of the case is relayed to the customer service or awaits the intervention of system administrators, based on the nature of the business represented by the service.

A compromise would be to tolerate some *bounded degree* of *shared data inconsistency* [20, 29], 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 by the service. The current dominating approach in web-based shopping systems is to satisfy the client at all costs and never defer its session (which correponds to tolerating $q = \infty$), but one could envision scenarios where imposing some limits on the 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 on-line shopping, examples of the systems where such tradeoffs might prove beneficial, are on-line trading systems and auctions.

To enforce that the chosen degree of data consistency is preserved, the service can rely on different *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 [10] and for advanced transaction models [7]. However, these algorithms need to be modified to be able to enforce session data consistency constraints, because of substantial differences between classical transactions and web sessions. Web sessions are long-running user-driven activities of interactive nature, with inter-request times much higher than request execution times. Therefore, requests waiting for shared resources can not be blocked forever, as clients, expecially human ones, are typically willing to tolerate only small response delays. Web sessions also do not have well defined boundaries and it is quite problematic to *rollback* a web session, by "undoing" or "compensating" its effects – one can not "undisplay" a web page in the user's browser. In some of these properties, web sessions resemble advanced transaction models such as Sagas [9], conversational transactions [28], and cooperative transactions [14].

In this paper we consider three concurrency control algorithms for web sessions – *Optimistic Validation*, *Locking*, and *Pessimistic Admission Control*. The algorithms work by *rejecting* the requests of the sessions for which they can not provide data consistency guarantees (so these sessions become *deferred*). However, they utilize different strategies in doing so, which leads to different number of deferred sessions, not known to the service provider in advance. In order to meaningfully trade off 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 (as opposed to the percentage of deferred sessions), for 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 each of the three discussed concurrency control algorithms. We present our models in the context of the sample *buyer* scenario for the well-known TPC-W e-Commerce benchmark application [26]. We compare the results of our analytical models with the results of concurrent web session execution in a simulated, and in a real web application server environment. The three sets of results closely match each other, validating the models.

Besides allowing one to quantitatively reason about tradeoffs between the benefits of limiting tolerable session data inconsistency and the drawbacks of necessarily deferring some sessions to enforce this data consistency, the models also permit comparison between concurrency control algorithms, with regards to the chosen metric of interest. In particular, since the proposed models use as input 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 its server environment (e.g., middleware platform), that would choose an appropriate concurrency control algorithm in real time, in response to changing service usage patterns.

To test this claim we implement such an infrastructure as a part of the J2EE application server JBoss [13] and deploy the TPC-W application. 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 a request profiling module. Our experiments show that the infrastructure is always able to pick-up the best algorithm, so during a test run with the dynamic adaptation in place, the infrastructure achieves a higher value of the metric of interest as compared to a

Table 1: Main TPC-W service requests.

| Page | Functionality | Page | Functionality |
|---|---|---|---|
| Home | Entry point to the application | Search | Performs search for specific items and presents them |
| New products | Presents items recently appeared in the store, for a specific category | Best Seller | Presents the list of items frequently purchased lately, for a specific category |
| Item Details | Presents the item details, including *available quantity* and *price* | Add To Cart | Adds item to the shopping cart, and displays its contents |
| Cart | Presents contents of the shopping cart | Register | Prompts user to authenticate or register itself |
| Buy Request | Authenticates or registers user, presents the *updated* view of the shopping cart, prompts user to submit address and credit cart information | Buy Confirm | Commits the order: records the order info in the database, decrements the available quantities for the items purchased, presents confirmation page |

scenario where the concurrency control algorithm is fixed.

The rest of the paper is organized as follows. Section 2 provides the required background and presents assumptions used throughout the paper. Section 3 describes the analytical models. Section 4 compares our models' results with the results of concurrent web session execution in a simulated, and in a real web server environment, showing close correspondence, and thereby validating the analytical models. Section 5 presents our middleware infrastructure for session data consistency enforcement. In Section 6 we discuss related work, and we conclude in Section 7.

# 2 Background

In this section we discuss session representation and modeling, present the concurrency control algorithms, and state assumptions made in this study.

## 2.1 TPC-W Application

We present our analytical models in the context of the TPC-W transactional web e-Commerce benchmark application [26]. TPC-W specifies the application data structure and the functionality of an on-line store that sells books, however it neither provides implementation, nor limits implementation to any specific technology. The TPC-W specification describes in detail $14$ different web invocations that constitute the web site functionality, and how they change the application data stored in a database. Table 1 describes the most important service requests that make up the TPC-W buyer scenario.

## 2.2 Session Model

A user session consists of service requests, executed in a specific order and frequency and usually corresponds to a service usage pattern, which reflects typical client behaviour. In this study we adopt the *Customer Behavior Model Graph* (CBMG) [18] approach to specify the behavior of an average user session. CBMG is a state transition graph, where states denote results of service requests (web pages), and transitions denote possible service invocations. Transitions in CBMG are governed by probabilities $p_{i,j}$ of moving from state $i$ to state $j$ ($\sum_j p_{i,j} = 1$). In our model we also allow a finite number of finite-domain attributes for each state of the CBMG. These attributes can be used to represent session state, e.g., events like signing-in and signing-out of an e-Commerce web site, or the number of items put into the shopping cart. The set of state transitions and probabilities can in turn depend on the values of these attributes. Since the set of attributes and their values is finite, each extended CBMG may be reduced to an equivalent classical CBMG, by duplicating states for each possible combination of attribute values.

In this paper, we consider a sample TPC-W *buyer* session, whose structure is described by the CBMG in Fig.1 (left figure). Each session 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 a (number of) item(s) in the

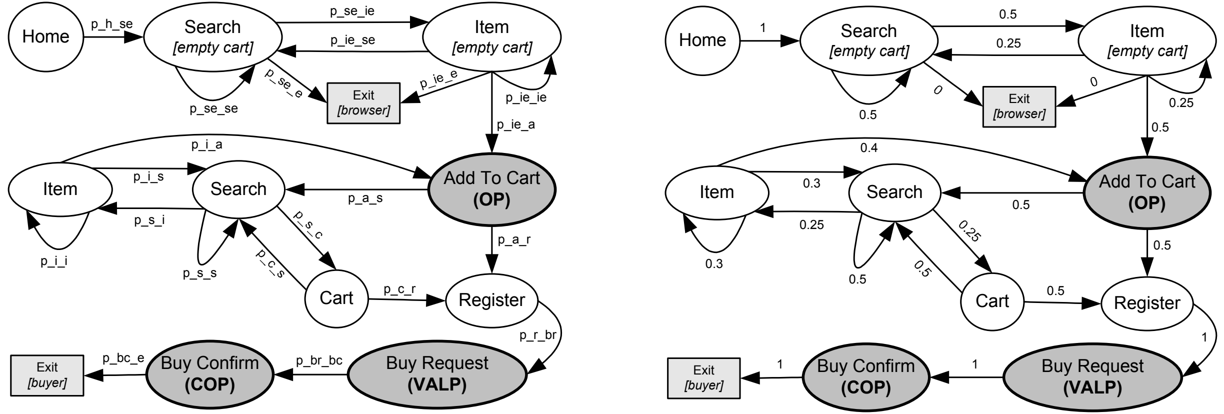


Figure 1: CBMG of the TPC-W buyer session (left) and an example used in the validation experiments (right).

shopping cart and completing the purchase (*buyer* session). Note that, for simplicity, we do not differentiate between the Search, New Products, and Best Sellers web requests and they are represented as one state. We have one (boolean-valued) CBMG state attribute for the Search and Item states, which denotes the presence of items in the shopping cart. To stress essential buyer activities in this sample scenario, we assume that once a user puts an item into the shopping cart, he never abandons the session and eventually commits the order. Each Item request carries an additional parameter – the `itemId` of the item to be displayed. We assume that there are $N$ items in the store, and that the $i$-th item is picked with probability $p_i$. The Add To Cart request chooses the same item that was picked in the preceding Item request, and it puts it in the shopping cart with quantity $1$.

## 2.3 Session Data Consistency Constraints

Information about the business-critical shared data that the service provider wants to cover by data consistency constraints can not be automatically extracted from the application structure or code – it needs to be identified by the service provider. To this end, we propose a flexible model for specifying web session data consistency constraints – the `OP-COP-VALP` model. The model is illustrated by the specification of the following data consistency constraint for the TPC-W application:

---

> For each session, the quantity of an item (with id $i$) seen in the Buy Request state which presents an updated view of the shopping cart, can differ by no more than $q_i$ units from the value seen by the Add To Cart request which inserted the item into the shopping cart. $q_i$ may be different for different items in the store.

---

**OP-COP-VALP model.** Potential shared data conflicts are identified by specifying pairs of *conflicting* service requests (operations): OPERATION (`OP` in short) and CONFLICTING OPERATION (`COP` in short). The relation is not symmetric apriori and means that `COP` *invalidates* `OP`, that is, the `COP` request changes some data, that was accessed or updated during the execution of an `OP` request by another session. One may also associate `correlation Id`(s) (`corr.Id` in short) with both the `OP` and the `COP` requests. `corr.Id`(s) is a (set of) value(s) that can be extracted from the parameters or the return value of the request. The `OP` and `COP` requests are considered *conflicting* if they come from different sessions (`COP` after `OP`) and their `corr.Id`(s) match (have non-empty intersection as sets). For our TPC-W application, `OP` is the Add To Cart request, with `corr.Id` being its `itemId` parameter; `COP` is the Buy Confirm request, with `corr.Ids` being the set of `itemIds` of the items in the shopping cart.

There are two ways to specify the data inconsistency that can be tolerated:

1. *Invalidation Distance*: one specifies the number of `COP` requests from other sessions that have to happen after the `OP` request, for this `OP` to become invalid. The intuition is that each `COP` changes the data to a certain (fixed) degree, so data inconsistency can be measured in terms of the number of `COP` requests.

2. *Numerical Distance*: one associates a numerical value (NUM_VAL) with the OP request, as a function of the request parameters and the return value; and defines how the (correlated) conflicting COP request changes this value. This makes possible specifying *absolute* or *relative* tolerable discrepancy for NUM_VAL. For the TPC-W application, NUM_VAL of the OP (Add To Cart) request is the available quantity of the item. The COP (Buy Confirm) request changes the correlated NUM_VAL value by decrementing it by the quantity of the item in the purchase (itemIds and quantities of the items in the purchase are all parameters of the Buy Confirm request). Thus, for our example constraints, $q$ is the relative inconsistency of NUM_VAL, that the service wishes to tolerate.

Web sessions are user-driven and open-ended. There is no global *commit* for a web session, at which point the service logic could make sure that OPs of the session have not been invalidated, and consider the session ended. Sometimes data consistency constraints for the session need only be satisfied if the session reaches a certain point, for example, in the buyer scenario one would want an item's price and quantity not to change substantially only if the user finally buys the item. Such events, when the service logic should *validate* a session's OPs, should be specified: in our model this is done by VALIDATION POINTs (VALP in short). VALP is a service request with a reference to a set of previously defined OPs (if necessary, correlated through corr.Ids), that it *covers*. The logic is that identified OPs need to be kept *valid* only for the time duration between the OP and VALP requests. For the TPC-W application, Buy Request corresponds to a VALP which covers all OPs of the session.

Abstracting application-specific data conflicts into the OP-COP-VALP model allows the application to delegate the responsibility for enforcing desired data consistency constraints to the underlying server environment (e.g., middleware). Generic middleware mechanisms could enforce data consistency constraints working only at the level of the abstract OP-COP-VALP model, with mapping of requests to OPs, COPs and VALPs, and other information specified by the service provider. This separation is consistent with the middleware paradigm of offloading functionality from the application code to the underlying server environment, and additionally permits *dynamic adaptation* of concurrency control policies to changes in parameters of service usage, in order to maximize the specified metric.

## 2.4 Concurrency Control Algorithms

Concurrency control techniques in transaction processing theory can be classified into two camps: *locking* techniques and *validation* techniques. The spirit of the first is to *lock* shared resources, preventing concurrent processes from accessing a locked entity until a certain safe point of execution is reached (e.g., transaction *commit*). The approach in the second camp is to let concurrent processes execute in parallel, accessing shared resources, and to *validate* execution in the end, hoping that conflicts either did not happen or cancelled each other out. Although these mechanisms are not directly applicable to web sessions, one can come up with similar concurrency control algorithms for web sessions with data consistency constraints. The algorithms determine whether to allow execution of a request (with all possible effects on shared application state) or to reject it. Once a request from the session has been rejected, the whole session is deemed *deferred*, with no additional requests coming from that session. This paper works with the following three natural algorithms, which are based on the OP-COP-VALP model for specification of data consistency constraints:

- **Optimistic Validation**: admit all OP and COP requests; when a VALP request arrives, validate the OPs that it *covers* – and admit or reject the VALP request accordingly. This technique resembles *backward validation* of classical transactions.

- **No-Waiting Locking** (**Locking**): this technique is applicable if every COP request in a session is preceded by a correlated OP request (think of OP and COP as READ and WRITE of the same data item). Assign a *logical lock* to each value of the corr.Id, and make the OP request obtain this lock when admitted and release the lock after the completion of the COP request. If OP can not obtain the lock it is immediately rejected (hence the name of the algorithm). Note that this technique has somewhat different semantics from the classical "no-waiting" locking – if the request is rejected the session is not restarted.

- **Pessimistic Admission Control**: admit OPs and VALPs; when a COP arrives, admit it only if it would not potentially invalidate OPs of other concurrent sessions. This technique resembles *forward validation* of classical transactions.

Note that these web session concurrency control algorithms build on top of serialization support of the underlying database and do not substitute conventional transactions – if the service logic requires it, the ACID properties of individual OP, COP, and VALP requests are guaranteed by the underlying middleware transaction service.

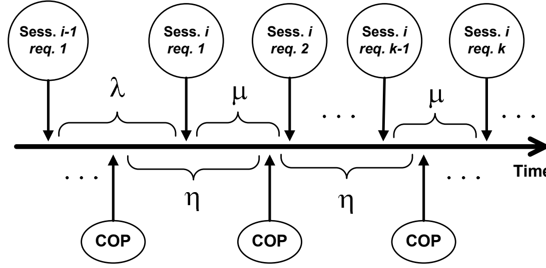


Figure 2: Analytical model.

### 2.5 Metrics of Interest

As client interaction with the service is organized in sessions and a client is satisfied only if its session successfully completes (i.e., it is not deferred), the measure of success of a particular concurrency control algorithm should be viewed in light of *how many sessions have completed successfully*. Therefore, we consider **percentage of successful sessions** as the main service performance metric throughout the study.

Another metric that we consider is the **percentage of requests belonging to successful sessions**, or simply **percentage of successful requests**, as a measure of what portion of system resources did good for clients, and what portion was wasted serving requests of deferred sessions. As we will see, the two metrics are not the same. While the first metric can be viewed as a *business* or *client satisfaction* metric, the second one is clearly a *system* metric.

Different concurrency control algorithms defer unsuccessful sessions at different stages of session execution, so the actual load on the service (e.g., request rate), produced by different algorithms is different. Therefore, another system metric we look at is the **effective request rate** seen by the service, measured in number of requests served per unit time (we count rejected requests too, because they also consume system resources).

## 3 Analytical Models

In this section we present three analytical models, one for each concurrency control algorithm (section 2.4). The models compute the three chosen metrics of interest (Section 2.5), based on parameters of service usage. The basic assumptions of the models are the following.

1. User sessions adhere to a CBMG, which is assumed known.
2. New sessions arrive as a *Poisson process* [15] with arrival rate $\lambda$.
3. Session inter-request times are independent with mean $1/\mu$, that is, requests from a session form a random process with the event arrival rate $\mu$. When we state this explicitly, we assume a specific distribution of session inter-request times. We also discuss the affect of specific distributions of session inter-request times in Section 4.
4. A request is served immediately and is either admitted and processed by the service, or rejected, which in turn terminates that session. Request processing time, including serialization delays in the underlying database, is assumed to be negligible compared to the average session inter-request time. In Section 4 we discuss the motivation behind this assumption.

The models, although somewhat different, rely on the following three key modeling techniques, used in other modeling studies as well [22, 1, 23, 24]:

1. **Approximating independence assumptions.** Execution of multiple concurrent web sessions is a compound random process, comprised of multiple inter-dependent finite-living random processes representing each session, which are in turn spawned by the Poisson process of new session arrivals. The inter-dependence is complicated further by the presence of session data consistency constraints. To simplify analysis, we assume that certain events are *independent* and approximatable as a Poisson process. The main such assumption is that `COP`

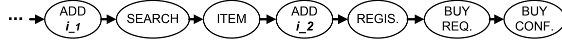


Figure 3: Example of a TPC-W buyer session.

requests form a Poisson process (with arrival rate $\eta$) which is independent of the Poisson process of incoming new sessions (see Fig. 2). Some of these assumptions are justified by the memorylessness property of Poisson process, in other cases they are not precisely correct, because requests originate from slightly correlated processes. However, as our validation results in Section 4 show, these assumptions prove a good approximation.

2. **Session enumeration technique.** In order to compute some probabilistic parameters, when it is difficult to do so purely analytically, we use the computational approach of *session enumeration*. We compute the value of the desired parameter for a session that has a particular known structure (sequence of requests). The final value of the parameter is the summation (over all possible sessions) of values obtained for individual sessions, weighted by the probability of the session having a specific structure. The number of possible session structures is, of course, infinite, but in the adopted CBMG session model, the probability of a session having length greater than $L$ decreases exponentially. For reasonably structured sessions with probabilities of transitions reflecting real-life service usage (see Section 4 for a discussion of this topic), it is sufficient to count all sessions of length less than 2 to 3 times the average session length (i.e., involving on the order of $30 - 40$ requests), to cover, say, 96-97% of all sessions, probability-wise. This makes it computationally feasible to implement the technique, which runs over the "majority" of session structures, and expect the running time of the modeling algorithm to be in the order of minutes, not hours. The enumeration algorithm also computes the probability space covered and specifically *adjusts* the computed value to *account for* sessions not enumerated.

3. **Fixed point iteration over an unknown value.** Due to the complex inter-dependent nature of concurrent session execution, it often happens that in order to compute a certain parameter $P$ through the session enumeration technique, we have to know the value of some other parameter, say $R$, which in turn depends on $P$. To break this loop, we assume some value for $P$, use the computation technique to find $R$ (and so $P$ as well), and iterate the procedure until convergence to a fixed point.

Although we illustrate our models using the CBMG and associated data consistency constraints of our sample TPC-W buyer scenario (Fig. 1), we note that the approach itself is general enough to be tailored to other CBMGs and associated session data consistency constraints.

Recall that our TPC-W sessions are divided between *browser* and *buyer* sessions. A browser session becomes a buyer session when it moves from the Item [empty cart] state to the Add To Cart state (Fig. 1). The probability that a session eventually makes this transition ($P_{\text{buy}}$) is easily computable from the state transition probabilities $p_{i,j}$ (this apparatus was developed in the CBMG model [18]). This gives us the rate of incoming buyer sessions $\lambda_{\text{buy}} = \lambda P_{\text{buy}}$. The rest of the section will concentrate on the buyer sessions. To clarify the presentation, we use small letters to denote probability values that are given by the model, e.g., $p_{i,j}$, and capital letters to denote values that we introduce and that need to be computed, e.g., $P_{\text{buy}}$.

In our TPC-W buyer session, `OP`s are Add To Cart requests, each adding one item to the shopping cart with quantity 1. Recall that we have $N$ items in the store, and probability of picking the $i$-th item is $p_i$. The `corr.Id` associated with each `OP` request is the `itemId` of the item put into the shopping cart. If a session has $K$ `OP` requests (i.e., $K$ items are put into the shopping cart, counting their quantities), we denote their `corr.Id`s (i.e., `itemId`s) as $i_1, i_2, \ldots, i_K$. Each `OP` request has an associated `NUM_VAL` value – the available quantity of the item at the moment of the request. Each individual `COP` with the same `corr.Id` $i$ decreases this value by 1. The session is successful, if `NUM_VAL` decreases by no more than $q_i$, between the `OP` (Add To Cart) and the `VALP` (Buy Request) requests. A Buy Confirm request, if admitted, decrements the available quantities of items that were purchased. We view the Buy Confirm request as *the set of unit decrements*, as many of them for each `itemId` as was the quantity of the item in the purchase. With this notation, an admitted Buy Confirm request produces the set of individual `COP`s ($K$ of them in total) with `corr.Id`s matching those of the `OP` requests in the session – $i_1, i_2, \ldots, i_K$. Throughout the rest of the section we will refer to the session in Fig. 3, as an example of a specific session structure. Note, that this session has two `OP`s with `corr.Id`s $i_1$ and $i_2$.

In all three analytical models, the first two percentage metrics – probability of session success ($P$) and percentage of requests belonging to successful sessions (*REQ*) – are computed by the session enumeration method, in which we actually enumerate not only session structures, but also all possible assignments of `corr.Ids` to `OPs`:

$$P = \sum_{\substack{\text{all sessions and} \\ \text{corr.Id assignments}}} P_{\text{sess}} \cdot P_{\text{succ}} \tag{1}$$

$$\mathit{REQ} = \frac{\sum_{\substack{\text{all sessions and} \\ \text{corr.Id assignments}}} P_{\text{sess}} \cdot P_{\text{succ}} \cdot L_{\text{succ}}}{L_{\text{av}}} \tag{2}$$

$P_{\text{sess}}$, the probability of a session having a particular sequence of requests $k_1, k_2, \ldots, k_L$ and `corr.Ids` $i_1, i_2, \ldots, i_K$ assigned to its `OPs` is given by the formula:

$$P_{\text{sess}} = \prod_{j=1}^{L-1} p_{k_j, k_{j+1}} \cdot \prod_{j=1}^{K} p_{i_j} \tag{3}$$

$P_{\text{succ}}$ is the probability of a particular session completing successfully. $L_{\text{succ}}$ is the number of requests in a particular session, when it is successful. $L_{\text{av}}$ is the average number of requests in a session. The third metric – request rate (*RATE*) – is given by the formula:

$$\mathit{RATE} = \lambda \cdot L_{\text{av}} \tag{4}$$

## 3.1 Optimistic Validation

The Optimistic Validation algorithm works by validating `VALP` requests (a single Buy Request in our case). The analytical model is built by assuming that we know the value of $\eta$ – the arrival rate of `COPs`. Using the value of $\eta$ we compute the probability of the Buy Request validation for a particular session structure. Using the session enumeration technique we compute the two percentage metrics (formulae (1) and (2)), along with $\eta$. Fixed point iteration over unknown $\eta$ completes the process. In developing this model for Optimistic Validation we assume that session inter-request times are exponentially distributed (with parameter $\mu$).

If a session is validated, its Buy Confirm request produces a set of $K$ `COPs`. So the expression for $\eta$, used by the session enumeration technique, is:

$$\eta = \lambda_{\text{buy}} \sum_{\substack{\text{all sessions and} \\ \text{corr.Id assignments}}} P_{\text{sess}} \cdot P_{\text{succ}} \cdot K \tag{5}$$

where the value of $P_{\text{sess}}$ is given by (3). To compute $P_{succ}$ – the probability of validating a session with a specific structure and a set of `corr.Ids` – we look at how many *distinct* `corr.Ids` are in the session (i.e., distinct items are in the cart), based on the known values of $i_1, i_2, \ldots, i_K$. For each distinct `corr.Id` $i$, all `OPs` with this `corr.Id` are validated, if the corresponding `NUM_VAL` value (i.e., available quantity of the item) decreases by no more than $q_i$ between the first `OP` (i.e., Add To Cart) and the `VALP` (i.e., Buy Request) requests. We assume, that validations of `OPs` with distinct `corr.Ids` are independent, so

$$P_{\text{succ}} = \prod_{\text{distinct } i \in \{i_1, i_2, \ldots, i_K\}} P_{\text{valid:corr.Id}=i},$$

where $P_{\text{valid:corr.Id}=i}$ – the probability of validating `OP` with `corr.Id` $i$, and with a specific distance between `OP` and `VALP`, which is inferred from the session structure. For example, in the session shown in Fig. 3 if the first item put into the cart has `itemId` 1 and the second has `itemId` 4, then the probability of session validation is the product of two validation probabilities: the first one – for `OP` with `corr.Id` 1 and a distance between `OP` and `VALP` of 5 requests, and the second – for `OP` with `corr.Id` 4 and a distance between `OP` and `VALP` of 2 requests.

We assume, that the portion of `COPs` with a particular `corr.Id` is proportional to the number of `OPs` with the same `corr.Id`, because if the session is validated, every `OP` is eventually followed by the `COP` with the same `corr.Id`. `OP` has `corr.Id` $i$ with probability $p_i$. This means that the flow of `COPs` with a particular `corr.Id` $i$, if viewed as a Poisson process, has arrival rate $\eta_i = \eta \cdot p_i$. Thus, computation of $P_{\text{valid:corr.Id}=i}$ reduces to the following problem.

Given two Poisson processes, the first with arrival rate $\mu$ (session requests between `OP` and `VALP`), the other with arrival rate $\eta_i$ (the flow of `COP`s with specific `corr.Id` $i$), find the probability that $M$ requests from the first flow ($M$ being the distance between `OP` and `VALP`, known from the session structure) come earlier than $Q$ requests from the second flow ($Q$ being actually $q_i + 1$, where $q_i$ is the tolerable inconsistency, because the $q_i + 1$-st `COP` will invalidate `OP`). This probability (let's denote it $P_{\text{succ}}(\mu, M, \eta_i, Q)$) is exactly $P_{\text{valid:corr.Id}=i}$.

The probability that exactly $k$ requests arrive in a Poisson process with arrival rate $\mu$ in the interval $(0, t)$: $P(\mu, k, t) = \frac{(\mu t)^k}{k!} e^{-\mu t}$. The PDF of the random variable representing the time of the $M$-th request arrival is

$$pdf_{\mu,M}(t) = \lim_{\Delta t \to 0} \frac{P\{M^{\text{th}} \text{ req. arr. in } (t, t+\Delta t)\}}{\Delta t} = \lim_{\Delta t \to 0} \frac{P(\mu, M-1, t) \cdot P(\mu, 1, \Delta t)}{\Delta t} = \frac{(\mu t)^{M-1} \cdot \mu \cdot e^{-\mu t}}{(M-1)!}$$

$P_{\text{succ}}(\mu, M, \eta_i, Q)$ is obtained as the convolution of the PDF $pdf_{\mu,M}(t)$ of the time of $M$-th request arrival in the first process and the probability that by that time there will be less than $Q$ requests that would have arrived in the second Poisson process, $P(\eta_i, < Q, t)$. The latter is equal to $\sum_{k=0}^{Q-1} P(\eta_i, k, t) = \sum_{k=0}^{Q-1} \frac{(\eta_i t)^k}{k!} e^{-\eta_i t}$, and thus (we omit some details for brevity):

$$\begin{aligned}
P_{\text{succ}}(\mu, M, \eta_i, Q) &= \int_0^\infty P(\eta_i, < Q, t) \cdot pdf_{\mu,M}(t) dt = \int_0^\infty \left( \sum_{k=0}^{Q-1} \frac{(\eta_i t)^k}{k!} e^{-\eta_i t} \right) \frac{(\mu t)^{M-1} \cdot \mu \cdot e^{-\mu t}}{(M-1)!} dt = \dots \\
&= \sum_{k=0}^{Q-1} \frac{\eta_i^k \cdot \mu^M}{k! \cdot (M-1)! \cdot (\mu + \eta_i)^{k+M}} \int_0^\infty t^{k+M-1} \cdot e^{-t} dt = \\
&= \frac{\mu^M}{(M-1)! \cdot (\mu + \eta_i)^M} \sum_{k=0}^{Q-1} \frac{\eta_i^k}{k! \cdot (\mu + \eta_i)^k} \Gamma(k + M) \qquad (6)
\end{aligned}$$

where $\Gamma(z) = \int_0^\infty t^{z-1} \cdot e^{-t} dt$ is the Gamma function [2], defined for complex values $z$, and known for positive integer $k$: $\Gamma(k) = (k-1)!$. Substituting this into equation (6) gives the final expression for $P_{\text{succ}}(\mu, M, \eta_i, Q)$:

$$\frac{\mu^M}{(M-1)! \cdot (\mu + \eta_i)^M} \sum_{k=0}^{Q-1} \frac{(k + M - 1)! \cdot \eta_i^k}{k! \cdot (\mu + \eta_i)^k}$$

Finding $P$ and *REQ* is completed by the fixed point iteration process over unknown $\eta$. The value given by r.h.s. of (5), if viewed as a function of $\eta$ is a strictly decreasing function, because the greater the argument (the assumed value of $\eta$), the fewer the number of validated sessions, and, in turn, the less the value of the r.h.s. of (5). Finding the intersection of a strictly decreasing positive function with the function $y = x$ is straightforward.

To compute effective request rate (*RATE*) by formula (4), we need to know $L_{\text{av}}$. If sessions are allowed to progress till the end, then the average session length ($L_{\text{av}}^{\text{ideal}}$) can be easily computed from the CBMG state transition probabilities $p_{i,j}$. The presence of the concurrency control algorithm makes some sessions shorter, because they are rejected. Identifying the points in a session's structure when the session can be rejected and comparing its length with the length of the same session running in the absence of any concurrency control algorithms, shows how $L_{\text{av}}$ relates to $L_{\text{av}}^{\text{ideal}}$. In the case of Optimistic Validation method and the particular CBMG of the TPC-W session we consider, we conclude that every unsuccessful session is one request shorter than when it is successful, because the Buy Request is rejected and there is no final Buy Confirm request. Therefore, $L_{\text{av}} = L_{\text{av}}^{\text{ideal}} + P - 1$.

The complexity of the algorithm is linear in $q_i$, polynomial in $N$ and the number of states in the CBMG, and exponential in $L$ (the maximum length of sessions counted in the session enumeration technique). The latter parameter contributes the most to the complexity of the computation, but as we pointed out earlier, being on the order of several dozens for reasonably structured real-life sessions, it makes it feasible to use the algorithm.

### 3.2 Locking

Recall, that the Locking algorithm works by assigning $q_i + 1$ *logical locks* to each `corr.Id` value $i$, where $q_i$ is the tolerable `NUM_VAL` inconsistency. Each `OP` tries to obtain a lock associated with the `OP`'s `corr.Id`. If it does not succeed, the request is rejected, the session is considered aborted, and the locks held by the session are released. All locks are released after the `COP` request.

In the model for the Locking algorithm we assume that we know the values of $P_{\text{lock}}$ – the probability that `OP` (regardless of its `corr.Id`) succeeds in obtaining a lock and $T$ – the average time the lock is held for. $P_{\text{lock}}$ is then used to compute $\lambda_{\text{OP}}$ – the arrival rate of `OP`s. All three values are used to compute the probability of a particular session's success ($P_{\text{succ}}$), which is used by the session enumeration technique to compute all three metrics of interest (formulae (1), (2) and (4)), along with the values of $P_{\text{lock}}$ and $T$. Fixed point iteration over unknown $P_{\text{lock}}$ and $T$ completes the model.

A session is successful if it acquires the lock on every `OP` request, so

$$P_{\text{succ}} = \prod_{j=1}^{K} P_{\text{lock}:i_j} \tag{7}$$

where $P_{\text{lock}:i}$ is the probability of obtaining the lock for `corr.Id` $i$ (we assume that the probabilities of obtaining locks for different `corr.Id`s are independent).

Finding $P_{\text{lock}:i}$ is the cornerstone of the model. To achieve this, we need $\lambda_{\text{OP}}$ – the arrival rate of `OP` requests. Using the probabilities of state transitions $p_{i,j}$ it is easy to compute $P_{\text{ret}}$ – the probability that after visiting the Add To Cart state a session will return to it again (see [18]). In the Locking algorithm, the progress of a *buyer* session is conditional on it being admitted in every Add To Cart request, so the probability of returning to the Add To Cart state is equal to $P_{\text{lock}} P_{\text{ret}}$. In addition to the first `OP` request in each buyer session, which contributes an arrival rate portion of $\lambda_{\text{buy}}$ towards $\lambda_{\text{OP}}$, there is the flow of second `OP`s with arrival rate $\lambda_{\text{buy}} P_{\text{lock}} P_{\text{ret}}$, the flow of third `OP`s with arrival rate $\lambda_{\text{buy}} (P_{\text{lock}} P_{\text{ret}})^2$, and so on. Therefore,

$$\lambda_{\text{OP}} = \lambda_{\text{buy}} \sum_{k=0}^{\infty} (P_{\text{lock}} P_{\text{ret}})^k = \frac{\lambda_{\text{buy}}}{1 - P_{\text{lock}} P_{\text{ret}}} \tag{8}$$

The overall flow of `OP`s divides into $N$ subflows of requests with a particular `corr.Id` $i$, with arrival rates $\lambda_{\text{OP}} \cdot p_i$. For each `corr.Id` $i$, we consider `OP` requests as "customers", $q_i + 1$ locks as $q_i + 1$ "servers" and the time between an `OP` request and the corresponding `COP` request (during which the lock is held) in a session as "customer service time". Then the $q_i + 1$-lock algorithm of the Locking method introduces the virtual queueing system **M/G/q$_i$+1/q$_i$+1** [11], with the arrival rate of "customers" being $\lambda_{\text{OP}} \cdot p_i$. The number of "customers" in such a system in the steady state – random variable $\xi$ – depends only on the expected value of the distribution $G$ (which represents "customer service time"), i.e., only on the average time of holding a lock – $T$. It is possible to obtain the lock only if the corresponding queueing system is not full, i.e., there are fewer than $q_i + 1$ "customers" in the system, therefore,

$$P_{\text{lock}:i} = \text{pr}(\xi < q_i + 1) = 1 - \frac{\frac{(\lambda_{\text{OP}} \cdot p_i \cdot T)^{q_i+1}}{(q_i+1)!}}{\sum_{k=0}^{q_i+1} \frac{(\lambda_{\text{OP}} \cdot p_i \cdot T)^k}{k!}} \tag{9}$$

Imagine that we know the values of $P_{\text{lock}}$ and $T$. Equation (8) gives us the value of $\lambda_{\text{OP}}$. Then, in the session enumeration phase we compute the metrics of interest, using (7) and (9). The value of $P_{\text{lock}}$ is computed by observing that $P_{\text{lock}} = \sum_{i=1}^{N} p_i \cdot P_{\text{lock}:i}$. $T$ is also computed by the session enumeration technique:

$$T = \sum_{\substack{\text{all sessions and} \\ \text{corr.Id assignments}}} P_{\text{sess}} \cdot T_{\text{sess}},$$

where $T_{\text{sess}}$ is the average time a lock is held in a particular session. The value of $P_{\text{sess}}$, the probability of a session having a particular structure and a particular `corr.Id` assignments to its `OP`s, is given by (3). In the $M/G/c/c$ system, customer service time is counted only for the customers *admitted* to the system. The zero time of a customer discarded without serving due to the limited server capacity does not count towards average customer service time.

Therefore, in the computation of $T_{\text{sess}}$, we count only non-zero locking time periods, and among all $K+1$ possible lock acquisition outcomes we consider only the $K$ outcomes *that start with the first* `OP` *having obtained its lock*. For each outcome, we know the position of its `OP` – `COP` periods (when the locks are acquired and released) and their average duration. For example, if both `OP`s obtain locks in the example in Fig. 3, then we have two locking periods, the first lasting for $6/\mu$, on average, the second for $3/\mu$, with the average of $4.5/\mu$ for this outcome ($1/\mu$ is the average session inter-request time). If the second `OP` fails in obtaining the lock, we end up having only one locking period (between the first and the second `OP`s) lasting for $3/\mu$, on average. So for the session example in Fig. 3 we have: $T_{\text{sess}} = P_{\text{lock}:i_2}(4.5/\mu) + (1 - P_{\text{lock}:i_2})(3/\mu)$.

The average number of requests in a session ($L_{\text{av}}$), used to compute *REQ* in (2) and *RATE* in (4), is also computed using the session enumeration technique in a manner analogous to computing $T_{\text{sess}}$ – for each possible lock acquisition outcome we know the number of requests in the session. The Locking algorithm model is completed by fixed point iteration over the pair of unknown $\{P_{\text{lock}},T\}$. Specifically, we start by assigning $P_{\text{lock}}$ any value, say $0.5$, and $T$ – its lower bound, the average value of just one inter-request time period ($1/\mu$), and compute new values of $P_{\text{lock}}$ and $T$ by session enumeration. These new values serve as input for the next iteration, and the process repeats. Our experiments show that this process converges very quickly to the fixed point. The complexity of the whole algorithm is analogous to that of the Optimistic Validation algorithm.

### 3.3 Pessimistic Admission Control

This algorithm gives the worst performance with regards to the metrics of interest (we defer discussion of the reasons to Section 4), so models for it are irrelevant if one's goal is to maximize the metrics. We present it in the paper only for completeness, restricting our attention to only the strict consistency case – $q = 0$. Recall, that the Pessimistic Admission Control algorithm works by admitting the `COP` requests that are not going to potentially invalidate other sessions.

Unlike the first two models, the model for the Pessimistic Admission Control algorithm does not require a fixed point iteration. First, we compute $T$ – the average time between an `OP` and a `VALP` requests in a session. This value is used in the session enumeration to compute $P_{\text{succ}}$ for a particular session, to get the first two metrics – $P$ and *REQ* (formulae (1) and (2)). In the TPC-W buyer session, the number of requests does not depend on its success, because possible request rejection only happens in the last request of the session – the Buy Confirm request. Therefore, the average session length ($L_{\text{av}}$) is the same as in the absence of any concurrency control algorithms – $L_{\text{av}}^{\text{ideal}}$, which is computed from $p_{i,j}$ (see [18]). This observation completes the model by computing the *RATE* metric (equation (4)). The complexity of the whole algorithm is analogous to that of the Optimistic Validation algorithm.

We compute $T$, the average time between an `OP` and the `VALP` requests in a session, in a separate session enumeration pass. The value of $T$ for a particular session is immediately seen from the session's structure. For the session in Fig. 3, it is equal to $3.5/\mu$, because there are two `OP` – `VALP` periods, of 5 and 2 inter-request times, respectively. Note that $T$ depends solely on the structure of ECBMG and its state transition probabilities $p_{i,j}$.

In the TPC-W buyer session, the `COP` (i.e., Buy Confirm) request is admitted, if all individual `COP`s, comprising it, are admitted. In the strict consistency case ($q = 0$), if an individual `COP` performing a unit decrement of `NUM_VAL` for a particular `corr.Id` is admitted from a session, this implies that no concurrently active sessions involve that `corr.Id`. Therefore, an arbitrary number of additional `COP`s for the same `corr.Id` from the same session can also be admitted at the same time. Therefore,

$$P_{\text{succ}} = \prod_{\text{distinct } j \in \{i_1, i_2, \ldots, i_K\}} P_{\text{admit}:j}$$

where $P_{\text{admit}:j}$ is the probability that `COP` with `corr.Id` $j$ is admitted (we assume independence for different `corr.Id`s).

To compute $P_{\text{admit}:j}$, we need to know $\lambda_{\text{OP}}$ – the arrival rate of `OP` requests. Each session produces, on average, $V_{\text{add}}$ number of `OP` requests, where $V_{\text{add}}$ is the average number of visits to the Add To Cart state (easily computable from $p_{i,j}$ [18]). Therefore, $\lambda_{\text{OP}} = \lambda_{\text{buy}} V_{\text{add}}$. The arrival rate of `OP`s with `corr.Id` $i$ is $\lambda_{\text{OP}} \cdot p_i$. For each `corr.Id` $i$, we consider the following virtual queueing system: `OP` requests are "customers", the "customers" are "served" while the session is between the `OP` and the `VALP` requests. There are an infinite number of "servers" in the system, because all sessions are allowed to progress between an `OP` and the `VALP`. The queueing system described is **M/G/∞** [11] with the arrival rate of customers being $\lambda_{\text{OP}} \cdot p_i$. The number of customers in the system in the steady state is a random

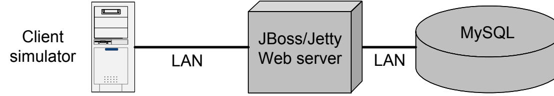


Figure 4: Web server test environment.

Table 2: Analyses of request logs of real shopping sites compared to our model parameters.

| **Study** | Arlitt et al. [3] | Cherkasova et al. [8] | Menascé at al. [18, 19] | Our model |
|---|---|---|---|---|
| Distr. of session length (req) | exponential, mean 8 | exponential, mean 36.5 | exponential for human users, 7 types of sessions: mean 6.9 – 14.77 | asymptotically exponential, mean 14.45 |
| Distr. of session inter-request times (s) | mean 40, charts resemble log-normal | don't say, in simulation use exponential, mean 5 | N/A | exponential (and log-normal), mean 10 |

variable (denoted by $\xi_i$ for `corr.Id` $i$). An individual `COP` with `corr.Id` $i$ is admitted if there are no concurrently active sessions that involve the same `corr.Id` $i$, i.e., $\xi_i = 0$. Therefore,

$$P_{\text{admit}:i} = \text{pr}(\xi_i = 0) = \frac{(\lambda_{\text{OP}} \cdot p_i \cdot T)^k}{k!} e^{-\lambda_{\text{OP}} \cdot p_i \cdot T}$$

where $T$ is the expected value of distribution $G$ (representing customer service time), i.e., the average time between an `OP` and a `VALP` requests in a session, which was computed earlier.

# 4 Model Validation

To validate our models we conducted the following two sets of experiments:

1. Execute concurrent TPC-W sessions with bounded data inconsistency in a simulation environment, implemented in Java and consisting of a virtual server with a logical rendering of TPC-W (with no actual database accesses) and a driver to simulate client load. This simulation represents an ideal rendering of the process, with resource contention limited only to Java synchronization, and request response times being effectively zero.

2. Run concurrent sessions against a real application deployed in a real web server environment. To accomplish this, we implemented the TPC-W benchmark as a J2EE component-based application [25] and deployed it on the open-source J2EE application server JBoss [13] (bundled with the Jetty HTTP/Web server) and MySQL database, each running on a dedicated 1GHz dual-processor Pentium III workstation (Fig. 4). We implemented the three concurrency control mechanisms as embedded middleware modules in the JBoss/Jetty application server, so that the TPC-W application code contained no logic enforcing session data consistency constraints. A separate workstation was used to produce client load – web sessions adhering to the TPC-W buyer CBMG (Fig. 1) and to the client load parameters – $\lambda$ and $\mu$. The maximum sustainable request rate of the web application server was approximately $40$ req/s, with the bottleneck being the JBoss server, not the database server (unlike in typical TPC-W usage, the purpose of the tests was not to stress the database, but rather to exercise data conflicts among different web sessions).

Each test run generated $4000$ sessions, with statistics gathered from the middle $75\%$ portion of the run time to cut off warm-up and cool-down regions. For each experiment, we measured the three metrics of interest and compared them with the results produced by the analytical models.

The parameter space of the sample TPC-W buyer CBMG with the specified data consistency constraints is very large – it consists of $p_{i,j}$ (state transition probabilities), $N$ (number of items in the on-line store), $p_i$ (probability of picking the $i$-th item), $\lambda$, $\mu$ (client load parameters), and $q_i$ (tolerable consistency). We conducted the experiments for several sets of values, varying parameters in all dimensions of this space. All of them showed that the analytical models

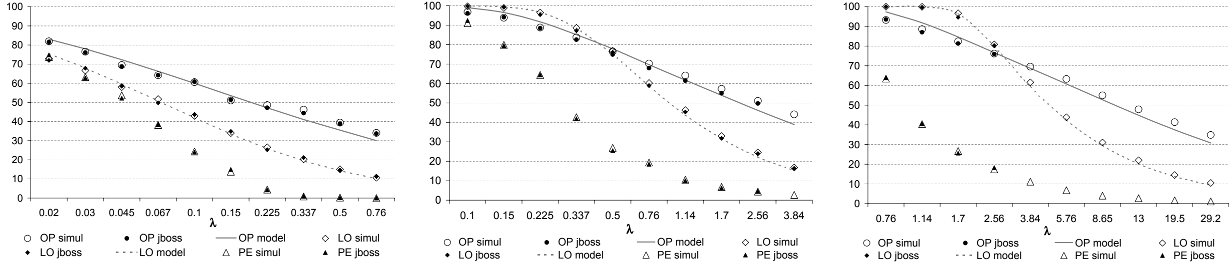


Figure 5: Percentage of successful sessions for $q = 0$ (left), $q = 6$ (middle), and $q = 30$ (right).

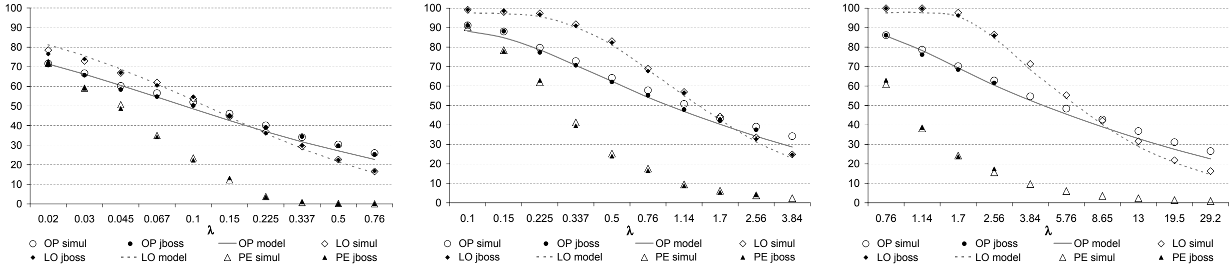


Figure 6: Percentage of requests belonging to successful sessions for $q = 0$ (left), $q = 6$ (middle), and $q = 30$ (right).

produce results closely corresponding to those of the simulation and the real web server environment experiments. Due to limited space here, we discuss the results of experiments for a "typical" on-line store workload, in which we fix all parameters except $\lambda$, which varies to present different client load, and $q_i$, for which we report on three sets of experiments: the strict consistency case ($q_i = 0$, for all items) and two relaxed consistency ones ($q_i = 6$ and $q_i = 30$).

To choose workload parameters representative of real-life usage, we studied several publicly available analyses of request logs of real shopping sites [3, 8, 18, 19]. Conclusions of these studies (shown in Table 2) differ (apparently because they were conducted at different times and for different web sites), for example in how session inter-request times are distributed. In both of our experiments (simulation and JBoss/Jetty/MySQL) we choose to use an exponential distribution, and later in the section we study how our results would differ if session inter-request times instead followed a log-normal distribution. The chosen values of CBMG state transition probabilities $p_{i,j}$ are shown in Fig. 1 (right figure) – note that the given CBMG has only buyer sessions. The other parameters are: $N = 5$;[1] $p_i = 0.2$ for $i = 1 \ldots 5$; $\mu = 0.1$, (it corresponds to an average session inter-request time of 10s). We have assumed for our models that request *processing* times are negligible compared to session *inter-request* times. This assumption is based on the fact that user *think* time is generally much higher (in order of tens of seconds) than users are willing to wait for request response (several seconds). In our JBoss/Jetty/MySQL tests, request response times were generally in the 20-100ms span, reaching 350ms under the maximum load (compare it to 10s of average session inter-request time).

Figures 5 and 6 compare the results of the two main metrics – the percentage of successful sessions and the percentage of requests belonging to successful sessions – for the three algorithms – Optimistic Validation (`OP`), Locking (`LO`), and Pessimistic Admission Control (`PE`), obtained in the simulation experiments (`simul`), in the JBoss/Jetty/MySQL web server tests (`jboss`), and by the analytical models (`model`). We do not include in the charts the analytical model results for Pessimistic Admission Control, because it is always outperformed by the other two algorithms. The charts are also missing the JBoss test results for $\lambda$ greater than 2.56, which we were unable to run due to limited server capacity.

The first observation is that the results of the models closely match both the simulation and the real web server environment results, which validates our proposed models. The models do sometimes have a little discrepancy with

[1] Having $N = 5$ does not mean that the store has only 5 items. A model's items may correspond to only those specific *hot-spot* items, for which the service provider wants to guarantee bounded data inconsistency.

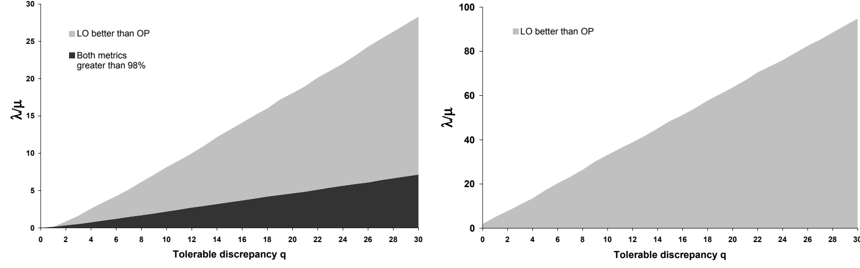


Figure 7: Algorithm performance comparison for the percentage of successful sessions (left) and the percentage of requests belonging to successful sessions (right).

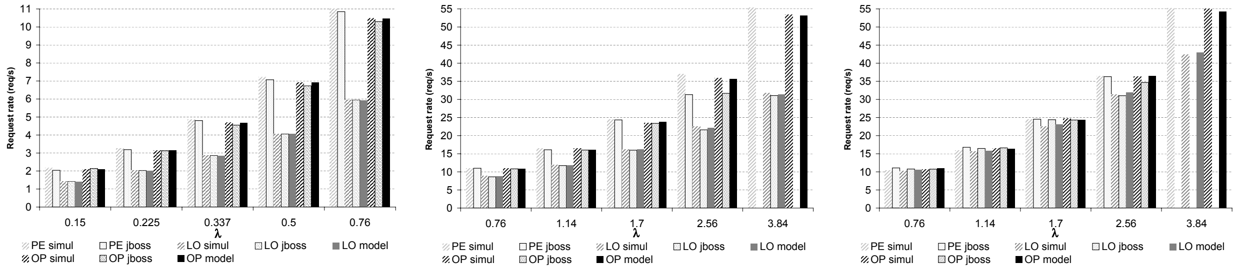


Figure 8: Request rate for $q = 0$ (left), $q = 6$ (middle), and $q = 30$ (right).

the experimental resuts, which tends to grow towards the ends of the $\lambda/\mu$ spectrum (note that $\lambda/\mu$ determines the "conflict rate" – the greater the value, the greater the number of concurrent sessions running, the more data conflicts they see, the less the values of the two percentage metrics of interest). However, it often happens that at the ends of the spectrum we have a clear algorithm winner, so discrepancy between the model and measurements does not hamper choosing the best concurrency control method. For example, with a large conflict rate (at the right end of the spectrum), Optimistic Validation always performs better than the other two algorithms.

**Pessimistic Admission Control.** This algorithm always performs worse than the other two, with respect to both percentage metrics. This happens, to our understanding, because of the "altruistic" nature of the method – sessions are rejected on `COP`s to give way to concurrent ones which otherwise would have been invalidated, but some of those sessions will also end up getting rejected, so some sessions are sacrificed in vain.

**Optimistic validation vs. Locking.** These two methods compete to achieve the best value for the metrics. Optimistic Validation's "selfish" approach seems to work better for higher rates of conflicts. The Locking algorithm is more "thoughtful" in that it works by rejecting sessions earlier (on `OP` requests), when it just sees the possibility of later conflicts. It may reject some sessions prematurely, but it lets other sessions run in a less competitive environment. And it seems to work, especially for higher values of $q_i$, where for lower rates of conflicts the Locking method outperforms its rival in both percentage metrics. The algorithm also works better for the percentage of successful requests metric, than it does for the percentage of successful sessions. The reason for this lies in the nature of the algorithm – it rejects unsuccessful sessions earlier in their lifetime, which makes them considerably shorter than successful ones. This, in turn, increases the portion of requests that belong to successful sessions.

To summarize the differences in performance of the Optimistic Validation and the Locking algorithms, we identify the regions where one algorithm works better than the other, according to the analytical models. Note that in the ideal setting, the "rate of conflicts" (and so – both of the metrics) depend only on the ratio of $\lambda$ and $\mu$. As in the previous experiments, we fixed the values of all the parameters except $\lambda, \mu$ and $q_i$. Gray areas in Fig. 7 show the regions where the Locking algorithm outperforms the Optimistic Validation, for the percentage of successful sessions (left chart), and the percentage of requests belonging to successful sessions (right chart). In both charts, the X axis plots the value of tolerable inconsistency $q_i$ (equal for all items), from 0 to 30, and the Y axis plots the ratio $\lambda/\mu$. We only considered

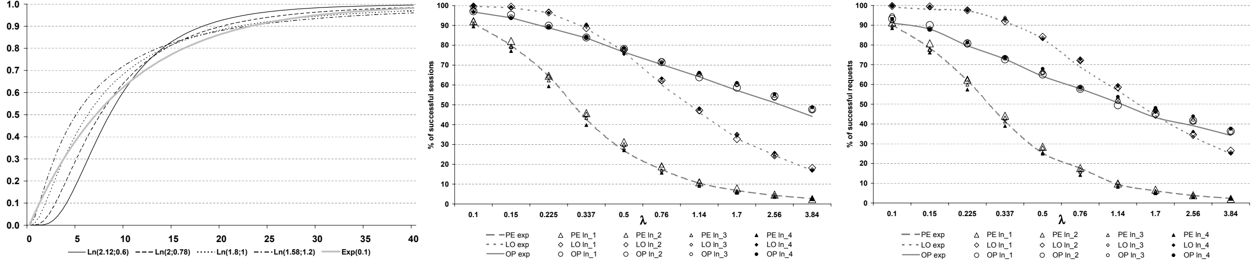


Figure 9: CDF of the exponential and log-normal distributions (left). Comparison of the two main metrics for the four log-normal and one exponential distribution of session inter-request times, for $q_i = 6$ (middle and right).

the cases where at least one of the metrics lies in the interval of $2\%$ – $98\%$. The dark gray area in the left chart shows the region where the metrics for both algorithms are greater than $98\%$.

**Effective request rate.** Figure 8 shows the results for the third metric of interest – effective request rate. Note that for greater values of $\lambda$, request rates of the web server experiments are a little bit lower than the predicted and the simulation ones. This happens because under normal load, request response times are in the order of $20$–$100$ms span, which is indeed negligible compared to the session inter-request times (10s on average). However, under higher load, response times become higher (and reach $300$–$350$ms for $\lambda = 2.56$). These increased response times start making a slightly noticeable contribution to the interval between *sending* the requests, which become higher, so the effective request rate decreases.

**Server capacity considerations.** Operating under higher user loads also reveals another major difference between the Optimistic Validation and the Locking algorithms – the Locking method produces lower request rates on the service. This happens because of the shorter sessions in the Locking algorithm, which stems from the algorithm's main policy – abort potential unsuccessful sessions earlier. This difference may become important if the service operates under server capacity limitations – the algorithm may become preferrable over the Optimistic Validation technique, as one producing lesser load on the service, or with request rates better matching prescribed quotas. For example, we generally were unable to conduct experiments with $\lambda$ being $3.84$ and higher, because the projected request rates surpassed the capability of our web application server environment ($\sim 40$ req/s).

**Log-normal distribution of session inter-request times**. For our experiments we used exponentially distributed session inter-request times, but the analysis in [3] shows that they actually might resemble more a log-normal distribution. To find out how the metrics of interest depend on the session inter-request times, we conducted additional simulations with four different log-normal distributions used as the session inter-request times: Ln(2.12;0.6), Ln(2.12;0.6), Ln(1.8;1), and Ln(1.58;1.2), chosen so that their mean values were 10s, matching that of the exponential dstribution Exp(0.1) used in the previous simulations ($E[\mathrm{Ln}(\mu;\sigma)] = e^{\mu+\sigma^2/2}$). Figure 9 shows Cumulative Distribution Functions (CDF) of the five distributions of interest (left chart), and the two main percentage metrics (middle and right charts), for the case of $q_i = 6$. As we can see, the metrics are quite insensitive to the actual distribution of session inter-request times (but rather depend on its mean value), as was also suggested by our analytical models; only the model for Optimistic Validation used a specific distribution of session inter-request times.

# 5 Middleware Infrastructure for Data Consistency Enforcement

The proposed models of concurrent web sessions use as input parameters specified by the service provider (the CBMG structure, $N$, $q_i$) and service usage parameters ($\lambda$, $\mu$, $p_i$, and the CBMG state transition probabilities). The latter could be obtained through *profiling* of incoming client requests, which makes possible the implementation of an *automated* decision making module as a part of the middleware platform hosting the service. This module would choose an appropriate concurrency control algorithm *in real time*, in response to changing service usage patterns. Note that the application code of the service then does not need to contain any logic to enforce session data consistency constraints – this task is performed completely by the underlying middleware.

Table 3: The client load of the dynamic adaptation experiments.

| **Phase** | Phase 1 | Phase 2 | Phase 3 | Phase 4 | Phase 5 |
|---|---|---|---|---|---|
| **Model parameters** | Buyer-1, $N = 5$, $p_i = 0.2$, $\lambda = 3$, $q_i = 10$ | Buyer-1, $N = 5$, $p_i = 0.2$, $\boldsymbol{\lambda = 1}$, $q_i = 10$ | **Buyer-2**, $N = 5$, $p_i = 0.2$, $\boldsymbol{\lambda = 0.5}$, $q_i = 10$ | Buyer-2, $N = 5$, $p_i = 0.2$, $\lambda = 0.5$, $\boldsymbol{q_i = 3}$ | Buyer-2, $N = 5$, $\boldsymbol{p_i = \{0.8, 0.1, 0.04, 0.03, 0.03\}}$, $\lambda = 0.5$, $q_i = 3$ |

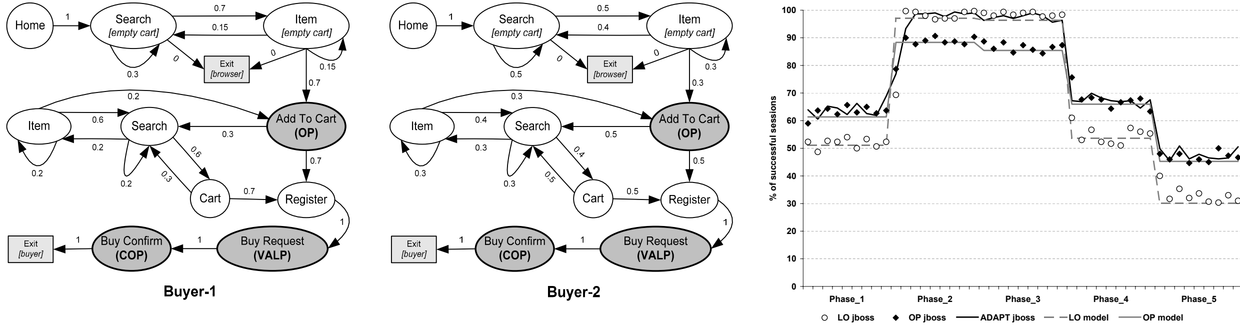


Figure 10: The buyer CBMGs and the results of the dynamic adaptation experiments.

We implemented one such decision making module as a set of pluggable services in the J2EE application server JBoss [13]. The infastructure consists of the following submodules:

- **Profiling Service** – profiles incoming web requests, keeping track of each session in progress; produces estimates on the service usage parameters (with confidence intervals) based on observed history of client requests.
- **Concurrency Control Service** – based on the predefined mapping of web requests to `OP`s, `COP`s and `VALP`s, performs actual concurrency control for web sessions according to the employed algorithm, by rejecting appropriate client requests, utilizing the ability of the middleware to *intercept* request execution at different levels (web, inter-component, database); the algorithms are tailored to be able to switch concurrency control methods on the fly and still enforce data consistency for older sessions that started before the switch.
- **Analytics Service** – computes the analytical models, based on the predefined structure of the CBMG and the set of given model parameters; this service is run on a separate machine because of its CPU intensive nature.
- **Decision Making Service** – main control module that orchestrates actions of the other infrastructure services. It periodically extracts the model parameters from the profiling service, computes the models using the analytics service and decides to switch to the better concurrency control algorithm if that shows metric benefits greater than a predefined threshold. Care is taken to avoid switching the algorithm due to abrupt fluctuation in service usage.

We conducted three experiments with our infrastructure. The first two fixed the concurrency control algorithm (Optimistic Validation and Locking), and the third tested automatic adaptation, with the objective of maximizing the percentage of successful sessions. All three experiments used the same client load and service-specific parameters, which consisted of the 5 phases shown in Table 3 (the two CBMGs used for the client load – "Buyer-1" and "Buyer-2" – are shown in Fig. 10). The value of $\mu$ is always $0.1$, as in all our experiments. Each phase generates 2700 sessions, and differs from the previous one in one or two parameters (highlighted in the table). Each phase is divided into 9 epochs with 300 sessions each, for which the results are shown on the right chart of Fig. 10.

Our experiments show that the infrastructure is always able to pick-up the best algorithm, so during the test run with the dynamic adaptation in place, the infrastructure achieves a higher percentage of successful sessions (75.6%) compared to the Locking (67.1%) and the Optimistic Validation (70.2%) tests, where the algorithms are fixed.

# 6 Discussion and Related Work

The notion of a web session, as a structural organization of client commmunication with Internet services, was first investigated in [8] and [16]. Since then several other studies have explored session characterization of Web workloads [18, 19, 3, 21] and proposed models for session-based workload simulation [16, 6, 18]. The work in [5, 17] acknowledged that *service usage patterns* affect the performance of Internet services. Our study follows this trend, relating session-oriented characterization of service usage to service performance and other metrics, and thereby guiding selection of server-side resource management mechanisms. To this end, we (1) propose models of concurrent execution of web sessions with bounded inconsistency in shared data, (2) show that the models reflect actual behavior of concurrent sessions, and (3) demonstrate that the models allow *dynamic adaptation* of the server-side concurrency control mechanisms to increase the chosen metric.

Some of the context of this study is influenced by previous research. Allowing a bounded discrepancy of $q_i$ is analogous to relaxation in epsilon-serializability [20]. The ideas of specifying conflicting operations and validation points in the `OP-COP-VALP` model relate to semantics-based concurrency control [4], while defining tolerable inconsistency through the invalidation and numerical distances is analogous to the Order and Numerical Errors in TACT [30].

Web sessions with bounded inconsistency in shared data can be viewed as long-running open-ended "transactions" with specific data consistency constraints, and resemble advanced transaction models such as Sagas [9], conversational transactions [28], and cooperative transactions [14]. However, unlike Sagas, web sessions don't have well defined boundaries and are not divided into sub-transactions. Conversational transactions are "chopped" into a chain of smaller transactions, each of which corresponds to recieving a message and sending a reply. Previous work on conversational transactions has primarily focused on providing mechanisms to durably store and efficiently recover the conversation context, rather than dealing with data conflicts; in the web sessions context, the former problem admits simple solution such as cookies (although it is not 100% failure resilient [28]), while the latter issue has not received as much attention. Like web sessions, cooperative transactions are also viewed as long-running, open-ended activities with a user-defined notion of correctness of execution. However, cooperative transactions retain much of the control with the clients, who explicitly manage shared resources and transaction isolation. In web sessions, clients are oblivious to the specification of correctness, which is provided by the server provider in the `OP-COP-VALP` model. The main difference of web sessions from all transaction models is the inability to *rollback* a web session, by "undoing" or "compensating" its effects.

Analytical modeling of concurrency control mechanisms is a well studied classical problem [22, 1, 23, 24]. However, while conducting this study, we were unable to find a model for concurrency control that would specifically cover the case of web sessions modeled with CBMGs. As rightfully pointed out in [1], "nearly every study is based on its own unique set of assumptions regarding database system resources, transaction behavior, and other such issues." Most previous models focused on modeling classical database transactions which enforce strict consistency. The most notable peculiarities of our model are that (1) our No-Waiting Locking algorithm substantially differs from the classical "no-waiting" locking algorithm, by having no restarts; (2) we don't model resource contention, assuming request processing times are negligible compared to session inter-request times; and (3) we allow flexible relaxed consistency on a per-item basis. In conventional database transactions, it is acknowledged that *pessimistic* (locking) concurrency control approaches are more suitable for higher rates of conflicts, while *optimistic* ones work well for lower conflict rates [1]. As we have seen, in the web sessions case, the situation is opposite. This performance difference observed in simulations, also steered us to creation of our own models. We believe that the difference in locking mechanisms is the main reason why optimistic and locking approaches have the opposite behavior in classical transactions and in the case of web sessions. Interestingly, there are some similiarities between our results and the results of previous studies in the area of *real-time* database transactions [12, 27]. Such transactions have associated *completion deadlines*, that they have to meet in order to be successful. Real-time database transactions share the commonality that optimistic approaches often outperform the locking ones.

# 7 Summary

This paper proposes analytical models to characterize concurrent execution of web sessions with bounded inconsistency in shared data. The models predict the performance of concurrency control algorithms by computing the chosen metrics of interest, based on parameters of service usage. We have illustrated our models using the sample buyer

scenario from the TPC-W e-Commerce benchmark, and have validated them by showing their close correspondence to measured results of concurrent session execution in both a simulated and a real web application server environment. We have also implemented a profiling and automated decision making middleware infrastructure which has been shown to successfully choose, for the specified performance metric, the best concurrency control algorithm in real time in response to changing service usage patterns.

## Acknowledgments

This research was sponsored by DARPA agreements N66001-00-1-8920 and N66001-01-1-8929; by NSF grants CAREER:CCR-9876128, CCR-9988176, and CCR-0312956; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

## References

[1] R. Agrawal, M. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.

[2] L.V. Ahlfors. *Complex Analysis*. McGraw–Hill, 1979.

[3] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology*, 1(1):44–69, 2001.

[4] B.R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.

[5] P. Barford, A. Bestravos, A. Bradley, and M. E. Crovella. Changes in web client access patterns: Characteristics and caching implications. In *World Wide Web. Special Issue on Characterization and Performance Evaluation*, 2:15–28, 1999.

[6] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS'98*, pages 151–160, 1998.

[7] N.S. Barghouti and G.E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[8] L. Cherkasova and P. Phaal. Session-based admission control: A mechanism for peak load management of commercial Web sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002.

[9] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts And Techniques*. Morgan Kaufmann Publishers, 1993.

[11] D. Gross and C.M. Harris. *Fundamentals of Queueing Theory*. John Wiley & Sons, 1974.

[12] J.R. Haritsa, M.J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th Symposium on Principles of Database Systems*, April 1990.

[13] JBoss Open-Source Java Application Server. `http://www.jboss.org`.

[14] G.E. Kaiser. Cooperative transactions for multiuser environments. In *Modern Database Systems*, pages 409–433, 1995.

[15] L. Kleinrock. *Queueing Systems*. John Wiley & Sons, 1975.

[16] D. Krishnamurthy and J. Rolia. Predicting the performance of an E-Commerce server: those mean percentiles. In *Proceedings of the ACM SIGMETRICS 1st Workshop on Internet Server Performance*, June 1998.

[17] D. Llambiri, A. Totok, and V. Karamcheti. Efficiently distributing component-based applications across wide-area environments. In *Proceedings of ICDCS*, May 2003.

[18] D. Menascé, V. Almeida, R. Fonseca, and M. Mendes. A methodology for workload characterization of e-commerce sites. In *Proceedings of the ACM Conference on Electronic Commerce*, November 1999.

[19] D. Menascé, V. Almeida, R. Riedi, Fl. Ribeiro, R. Fonseca, and W. Meira Jr. In search of invariants for e-business workloads. In *Proceedings of the ACM Conference on Electronic Commerce*, pages 56–65, 2000.

[20] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *Knowledge and Data Engineering*, 7(6):997–1007, 1995.

[21] W. Shi, R. Wright, E. Collins, and V. Karamcheti. Workload characterization of a personalized web site and it's implication on dynamic content caching. In *Proc. of the 7th Workshop on Web Caching and Content Distrib. (WCW'02)*, 2002.

[22] Y.C. Tay, R. Suri, and N. Goodman. A mean value performance model for locking in databases: The no-waiting case. *Journal of the ACM*, 32(3):618–651, 1985.

[23] A. Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Transactions on Database Systems*, 18(4):579–625, 1993.

[24] A. Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[25] TPC-W-NYU. A J2EE implementation of the TPC-W benchmark `http://cs1.cs.nyu.edu/~totok/professional/software/tpcw/tpcw.html`.

[26] Transaction Processing Performance Council. TPC-W Benchmark. `http://www.tpc.org/tpcw/default.asp`.

[27] O. Ulusoy and G.G. Belford. Real-time transaction scheduling in database systems. *Information Systems*, 18(8):559–580, 1993.

[28] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann Publishers, 2002.

[29] M.H. Wong and D. Agrawal. Tolerating bounded inconsistency for increasing concurrency in database systems. In *Proceedings of the 11th Symposium on Principles of Database Systems*, June 1992.

[30] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.