Design and Evaluation of Redistribution Strategies for Wide-Area Commodity Distribution

Uğur Çetintemel[†]
University of Maryland, College Park
ugur@cs.umd.edu

Banu Özden
Bell Laboratories, Lucent Technologies
ozden@research.bell-labs.com

Michael J. Franklin
University of California, Berkeley
franklin@cs.berkeley.edu

Avi Silberschatz
Bell Laboratories, Lucent Technologies
avi@research.bell-labs.com

Abstract

The proliferation of e-commerce has enabled a new set of applications that allow globally distributed purchasing of commodities such as books, CDs, travel tickets, etc., over the Internet. These commodities can be represented on line by tokens, which can be distributed among servers to enhance the performance and availability of such applications. There are two fundamental approaches for distributing such tokens — partitioning and replication. Partitioning-based approaches eliminate the need for tight quorum synchronization required by replication-based approaches. The effectiveness of partitioning, however, relies on token redistribution *techniques that allow dynamic migration of tokens to where they are needed. We propose pair-wise token redistribution strategies to support applications that involve wide-area commodity distribution. Using a detailed simulation model and real Internet message traces, we investigate the performance of our redistribution strategies and a previously proposed replication-based scheme. Our results reveal that, for the types of applications and environment we address, partitioning-based approaches perform superior primarily due to their ability to provide higher server autonomy.*

1 Introduction

The proliferation of e-commerce and widespread access to the WWW have enabled a new set of applications that allow globally distributed purchasing of commodities and merchandise such as books, CDs, travel tickets, etc., over the Internet. Companies, such as Amazon.com, Expedia, etc., that support these types of applications are drastically increasing in number and scale. As these applications become more popular, centralized implementations will fall short of meeting their latency, scalability, and availability demands, thereby requiring distributed solutions. Conventional distributed implementations, however, are also not viable for these applications: they inherently require tight global synchronization, and, thus, break down in environments where the nature of the communications medium is failure-prone and unpredictable.

More effective distributed solutions can be realized by exploiting two important characteristics of these applications: First, they involve a set of commodity types with a limited inventory (e.g., the latest CD of Santana, economy class tickets for a particular flight, etc.). Second, the operations of interest on these items typically involve incremental updates (e.g., buying two economy tickets for a flight). It is, therefore, possible to achieve distribution by using *tokens* to represent the instances of commodities for sale. Previous work (e.g., [5, 7, 11]) exploited the notion of tokens to enable high-volume transaction processing for distributed resource allocation applications.

There are two fundamentally different approaches for distributing tokens — *replication* and *partitioning*. Token replication typically requires expensive distributed synchronization protocols to provide data consistency, and is subject to both high latency and blocking in case of network partitions or long delays in communications between groups of sites (which are indistinguishable from network partitions). Token partitioning allows many transactions to execute locally without any global synchronization, which results in low latency and immunity against network partitions. The effectiveness of token partitioning, however, relies on *token redistribution* techniques that allow dynamic migration of tokens to the servers where they are needed.

In this paper, we examine the Data-Value Partitioning (DVP) [11] approach to token-based commodity distribution. We propose pair-wise DVP strategies that vary in the way they redistribute tokens across the servers of the system. Using a detailed simulation model and real Internet message traces, we investigate the performance of our DVP redistribution strategies by comparing them against each other and a previously proposed scheme, Generalized Site Escrow (GSE) [7], which is based on replication and escrow transactions [10]. GSE general-

[†] Part of this work was done while the author was working at Bell Laboratories, Lucent Technologies Inc.

izes previous escrow algorithms for replicated databases, providing higher server autonomy and throughput.

The main contributions of the paper are twofold: First, we extend the previous work on DVP by proposing new token redistribution strategies and evaluating their performance under a range of workload scenarios using real wide-area message traces. Second, even though the basic approaches, DVP and GSE, were both developed a number of years ago, this work is the first to directly compare their performance. Thus, this paper provides valuable insight into the fundamental tradeoffs between partitioning and replication for the increasingly important problem of wide-area commodity distribution.

The remainder of the paper is organized as follows. In Section 2, we briefly describe the system model and the DVP algorithm. In Section 3, we propose several token redistribution strategies for DVP. We discuss the GSE approach in Section 4. We describe the experimental environment and methodology in Section 5 and present our experimental results in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Overview of token partitioning

In this section, we first define our system model. We then briefly give an overview of the basic DVP algorithm [11]. For brevity, we focus only on the performance-related issues.

2.1 System model

Our reference system model (Figure 1), consists of a set of servers and clients that communicate via message exchange over a wide-area network. We assume, for simplicity of exposition, that the system stores a single commodity with a *limited* number of *indistinguishable* instances and we represent each such instance with a single token.

Through a web-based interface, clients submit transactions that *allocate* tokens or *return* (i.e., deallocate) tokens that have previously been allocated. Therefore, the types of transactions that we model involve incremental updates to a data item, *avail*, denoting the number of tokens globally available in the system. We also assume, without loss of generality, that there is a lower-bound constraint on *avail*: the system must ensure that *avail* does not become negative at any time (e.g., tickets for a particular show should not be oversold). Therefore, *all* token return transactions can potentially commit (as they cannot violate the lower-bound constraint), whereas only *some* of the token allocation transactions can commit.

2.2 The DVP approach to token partitioning

DVP [11] is a non-traditional approach for representing and distributing data. It essentially applies to data items that can be *partitioned* into smaller pieces such that the pieces can also be regarded as instances of the original data item. The same operators that apply to the original item should also apply to the pieces (e.g., increment, decrement, set to zero, etc.).

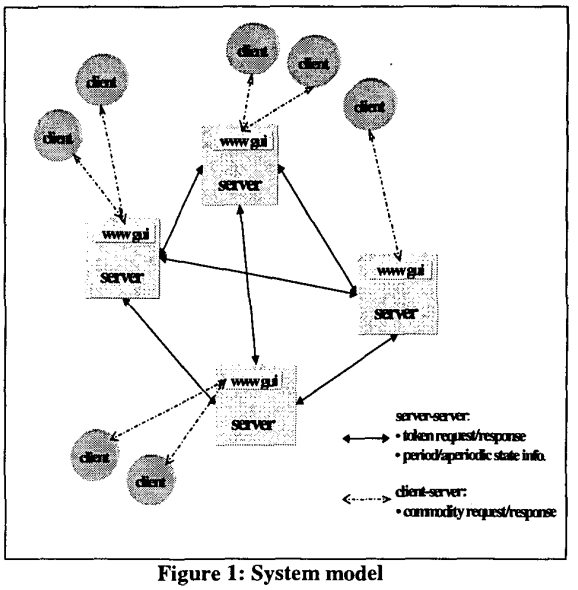

Figure 1: System model

The basic idea underlying DVP is to split up the values of database items and store each of the constituent values as tokens at different servers. Transactions are then executed locally at each server using the tokens locally available at that server. In the event that the number of tokens locally available is insufficient to execute the transaction, the server makes requests to other servers only to *borrow* some their tokens. If responses from other servers fail to arrive for any reason within a specified timeout period, the transaction is aborted. Tokens are locked before being accessed, however, only at the server where they reside, i.e., no lock requests are made to other servers. We refer to the number of tokens available at s_i as tok_i, and the number of tokens required to execute transaction t as $req(t)$, which is a negative value if t is a return transaction.

DVP is a fully *decentralized* scheme that does not require global synchronization. Due to its non-blocking behavior, it is immune to network partitions, and is thus particularly well-suited for environments with unpredictable and failure-prone communications (e.g., many servers on the Internet) due to the high server autonomy it enables.

3 Token redistribution strategies

Token partitioning enables servers to execute transactions locally as long as they have sufficient tokens. When a server cannot execute a transaction locally due to insufficient number of tokens, it must be able to locate tokens available at other servers and acquire them in a timely fashion in order to continue transaction execution. The performance of DVP relies upon how effectively this *token redistribution* among servers is accom-

Borrower server s_b:

1. Lock tok_b.
2. If $tok_b < req(t)$
   borrow_tokens().
   /* **borrow_tokens()** gathers tokens from other sites and updates tok_b as it receives tokens. */
3. Wait until $tok_b \geq req(t)$ or timeout.
4. If timeout
5. Abort; unlock tok_b; and exit.
6. Set $tok_b = tok_b - req(t)$.
7. Commit; and unlock tok_b.

Lender server s_l:

1. Lock tok_l.
2. Calculate the number of tokens, $resp(s_l, s_b)$, $0 \leq resp(s_l, s_b) \leq tok_l$, to be lent to s_b;
   Send $resp(s_l, s_b)$ tokens to s_b.
   Set $tok_l = tok_l - resp(s_l, s_b)$;
3. Unlock tok_l.

Figure 2: The Generic DVP algorithm

plished. The main questions that a token redistribution strategy needs to answer are:

1. when to request tokens,
2. which servers to request tokens from, and
3. how many tokens to request.

It is possible to construct a cost function with a set of constraints and solve it to find the *optimal* token redistribution. Unfortunately, not only it is difficult to construct a realistic cost function due to the dynamic, distributed nature of the system, and the fact that tokens are perishable resources (i.e., they cease to exist after being used), but also it is long known that even simpler formulations of the problem are NP-hard [4, 5]. Since optimal solutions are impractical, we focus on heuristics-based solutions. In particular we avoid global strategies that require tight synchronization, and concentrate only on decentralized, pair-wise strategies that make progress using only *pair-wise* synchronization. Note that previous work on DVP [11] focused on the basic features of partitioning and ignored token redistribution issues.

In the rest of the section, we describe several token redistribution strategies for the basic DVP algorithm (Figure 2). We refer to the number of tokens requested by s_b from s_l as $req(s_b, s_l)$, the number of tokens returned by s_l to s_b as $resp(s_l, s_b)$.

3.1 Random redistribution

We begin by describing our baseline strategy, `random`. In this strategy, the borrower contacts a lender server, which the borrower picks *randomly*, and requests the exact number of tokens needed:

$$req(s_b, s_l) = req(t) - tok_b$$

If s_b does not receive the entire amount it needs, it then randomly chooses another lender server and requests the remaining amount. The lender computes the return amount as:

$$resp(s_l, s_b) = \begin{cases} req(s_b, s_l), & \text{if } req(s_b, s_l) \leq tok_l; \\ tok_l, & \text{otherwise.} \end{cases}$$

The messages exchanged among servers are minimal, and only include token requests and responses.

3.2 Token count-based redistribution

The `random` strategy makes *blind* redistribution decisions, since it does not maintain or utilize any information about the states of other servers. The `count-based` strategy attempts to make more intelligent decisions by incorporating knowledge of the *token counts* at other servers into this decision process.

The state maintained by a server s_i is basically a *token table* (*tt*) that stores an estimate of the number of tokens available at all servers; i.e., $tt[j]=(tok_j^i, tstamp_j^i)$, where tok_j^i is the token count at s_j at logical time $tstamp_j^i$, $j=1...n$, and n is the number of servers.

The `count-based` strategy makes more intelligent redistribution decisions at the expense of maintaining and disseminating token count information. Servers disseminate token count information using *piggybacking* and *broadcasting*. Each server, when sending a message to another server, also incorporates its token table into the message. In addition to this piggybacking, servers broadcast their states to other servers at the following critical points:

1. when the number of tokens available at a server becomes less than a certain threshold value — so that servers with fewer tokens can be differentiated from the ones with many more tokens;
2. when a server runs out of tokens — so that other servers do not make token requests to this server anymore, and;
3. when tokens are returned at a server that previously had run out of tokens — so that other servers may resume requesting tokens from this server.

A server s_i updates its token table when s_i commits a transaction t:

$$tok_i^i = tok_i^i - req(t) \text{ and } tstamp_i^i = tstamp_i^i + 1$$

and, when s_i receives a token table from server s_k, $k \neq i$:

$$tok_j^i = tok_j^k \text{ and } tstamp_j^i = tstamp_j^k, \quad \text{if } tstamp_j^k > tstamp_j^i$$

$\forall j = 1...n, j \neq i$.

A borrower server, s_b, consults its state table when choosing a lender server. It (rank) orders the servers according to their token counts. The lender servers are then chosen based on their ranks: at each step, a *minimal* set of lenders, L, that together have a sufficient number of tokens is chosen as lenders. Formally, $L \in S - \{s_b\}$ is chosen such that:

$$\sum_{l \in L} tok_l^b \geq req(t) - tok_b, \text{ and}$$

$$\neg \exists L' \; s.t. \; L' \subseteq S - \{s_b\}, \sum_{l \in L'} tok_l^b \geq req(t) - tok_b, \; and \; |L'| < |L|$$

where S is the set of all servers (see Section 6.2 for the other lender selection schemes). The borrower then contacts the lenders in *parallel*, without waiting for replies. The borrower makes a pessimistic assumption about the availability of tokens at lender servers and requests $req(t)$-tok_b tokens from each lender. If the estimated token count of a server is zero, then that server is not contacted at all. The redistribution at a lender proceeds similar to that in the basic case.

3.3 Token demand-based redistribution

In the previous strategies, token redistribution occurs as the result of a token request, which is initiated only when the borrower has insufficient tokens to execute a transaction successfully. The `demand-based` strategy, on the other hand, continually redistributes tokens across servers based on the token request rates at each server. Such a *demand-based* redistribution is likely to be beneficial especially when there is a skew in server workloads.

Each server maintains a simple token request rate value, rr_j, for each server s_j, j=1...n. This value indicates the number of tokens requested from a particular server during a certain period of time. Each server updates its own request rate value and disseminate it along with its token table using piggybacking and broadcasting as described before.

The `demand-based` strategy employs two key techniques that utilize request rate values of the servers:

1. *Token sharing* refers to the redistribution of tokens based on the token request rates as observed by the involved servers. The borrower server s_b sends its token request rate, rr_b, along with its token request. The lender server s_l computes the number of tokens that it should share with s_b as:

$$share(s_l, s_b) = \left\lfloor c \cdot tok_l \cdot \frac{rr_b}{rr_b + rr_l} \right\rfloor$$

aiming to achieve a balanced token redistribution according to relative request rates (where c is the sharing constant). Server s_l then returns:

$$resp(s_l, s_b) = \begin{cases} share(s_l, s_b), & \text{if } share(s_l, s_b) \geq req(s_b, s_l); \\ req(s_l, s_b), & \text{if } share(s_l, s_b) < req(s_b, s_l) \text{ and} \\ & \quad tok_l \geq req(s_b, s_l); \\ tok_l, & \text{otherwise.} \end{cases}$$

2. *Token prefetching* refers to the periodic, request rate-based redistribution of tokens in the background. Prefetching can potentially eliminate the need to search for tokens *as part of* transaction execution, thereby reducing response time and increasing availability. Periodically, each server contacts every other server in some prefixed order (in the background). When s_i contacts s_j, the tokens available at s_i and s_j are redistributed among the two servers based on their relative request rates as follows:

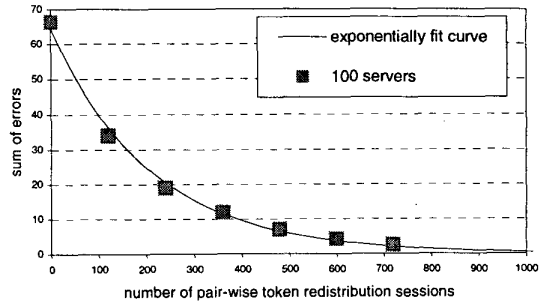

Figure 3: Incrementally converging to global target token distributions using pair-wise token exchanges

$$tok_i' = \left\lfloor (tok_i + tok_j) \cdot \frac{rr_i}{rr_i + rr_j} \right\rfloor$$

$$tok_j' = (tok_i + tok_j) - tok_i'$$

where tok_i' and tok_j' are the respective token counts at s_i and s_j *after* redistribution.

Although the token redistribution mechanism we described operates in a pair-wise fashion and uses *only* the information available locally at the two servers in contact (i.e., tok_i, tok_j, rr_i, and rr_j), it is quite robust in that it *incrementally* migrates the existing token distribution in the system to match the relative request rate distribution. In fact, we experimentally showed that, using this pair-wise mechanism, an existing token distribution converges to any desired global distribution *exponentially* fast. Figure 3 demonstrates this exponential convergence by plotting the number of pair-wise token exchanges (between randomly selected servers) versus the sum of errors between the global target token distribution and the existing token distribution (averaged over 1000 randomly selected target and initial token distributions for a system with 100 servers).

3.4 Primary-based, hybrid redistribution

Our preliminary experiments revealed that, as the number of tokens in the system decreases, it becomes increasingly harder to locate and gather the necessary number of tokens in a distributed fashion. Borrower servers typically make several unsuccessful attempts until they are able to gather the tokens they need. Even though there might be sufficient tokens globally available in the system, identifying the servers that have the tokens can be very costly, especially if the system consists of many servers. In such cases, transaction execution costs can become so high that the performance benefits of using a distributed system may be overshadowed.

The `primary` strategy attempts to add the benefits of using a centralized model for situations where only a small number of tokens remain in the system. The system operates in regular, decentralized mode (using the

count-based strategy) as long as the number of tokens available is above a fixed threshold value, but switches to a *centralized* mode of operation when the total number of tokens drops below this threshold. Each commodity is assigned a primary server that is responsible for satisfying all the requests involving the tokens of that commodity when the system operates in the centralized mode. Each server continuously observes the number of available tokens for the commodities for which it serves as the primary.

If the number of tokens drops below a particular *callback threshold value*, the corresponding primary initiates the switch to the centralized mode by broadcasting *callback* messages. Each server, having received a callback message for a commodity, sends all the tokens of that commodity to the primary. After the callback, the primary executes all requests involving that commodity locally. When a *non*-primary receives a token request after a callback, it simply forwards the request to the primary, which executes the request and returns the result back to the client. An alternative model, which we do not discuss here, assumes the existence of a set of *forwarding* agents, such as cluster DNSs that translate logical names into the IP-addresses of one of the servers [3], and enables client requests to be submitted directly to the corresponding primary.

If the number of tokens later increases above the callback threshold, the system switches back to its decentralized operation: the primary redistributes the tokens it has to other servers (uniformly or depending on its knowledge of the request rates at servers).

4 Algorithms for token replication

We now briefly describe the *Generalized Site Escrow* (GSE) scheme [7], an efficient replication scheme based on escrow transactions [10], as the representative replication-based approach to distributed token maintenance.

In GSE, the number of tokens available in the system, referred to as *avail*, is *replicated* at all servers. The escrow quantity at s_i, which represents the number of tokens that s_i can dispense without contacting other servers, is computed as:

$$es_i = \frac{avail_i}{n}$$

where $avail_i$ is s_i's view of *avail*, and n is the number of servers. Each server periodically broadcasts the token allocations it performed to limit the extent to which views of *avail* is out-of-date. The escrow quantity as each server, therefore, dynamically decreases as tokens are allocated. Each server estimates the escrow quantities at other servers, which are then used to replenish its own escrow quantity and allocate tokens without contacting other servers, if possible.

GSE relies on: (1) *gossip messages*, which are periodic background messages that include the token allocations known by the sender server, to limit the global token allocations unknown to a server, and; (2) *quorum locking* to limit the number of token allocations that can be performed concurrently at the quorum servers.

Parameter	*Setting*	*Parameter*	*Setting*
Number of servers	10	Database size	200 (commodities)
Local lock timeout	200 (ms)	Number of tokens	200
Remote lock timeout	500 (ms)	Number of clients	400
CPU speed	2000 (ms)	Transaction inter-arrival time	exp(10.0) (msec)
Disk latency	0.0097 (s)	Update transaction probability	1.0
Disk transfer rate	40 (MB/s)	Transaction size	1 (commodities)
Buffer hit ratio	0.95	Tokens requested per transaction	uniform(1,5) (tokens)
Item read cost	1000 (instr.)	Return transaction probability	0.1
Item write cost	2000 (instr.)	Workload skew (θ)	0.0 or 1.0
Message cost	10000 (instr.)	Prefetch period	100 (s)
Control message size	64 (bytes)	Callback threshold	20 (tokens)
Commodity item size	512 (bytes)	Sharing constant	1.0

Table 1: Primary experimental parameters and default settings

A server s_i can perform token allocations as long as es_i, which s_i updates dynamically as it allocates tokens and receives gossip messages, is large enough. In case es_i is not sufficient, s_i needs to contact other servers and form a synchronous quorum of servers such that the combined escrow quantities of the quorum servers are enough to execute the transaction. Forming a quorum of servers involves *remotely* locking the *avail* values at the quorum servers by sending *lock/unlock* messages. Such *remote* locking involving multiple servers, as our results demonstrate, is relatively expensive to perform in wide-area.

5 Experimental environment

Using CSIM [1], we implemented a detailed simulation model [2]. The model consists of components that model a distributed set of servers, a population of clients making commodity requests, and communication latencies among servers. We provide only a high-level description here due to space limitations

Server module. The server module consists of (1) a *resource manager*, which schedules the CPU and disk (using a non-preemptive FIFO policy); (2) a *buffer manager*, which handles the data transfer between the disk and buffer; (3) a *communication manager*, which handles the passing of messages to and from the network module using a queue to implement ordered message retrieval and processing. Every message sent or received by the server is charged a fixed CPU cost; and (5) a *transaction manager*, which coordinates the execution of transactions according to a specified protocol. It handles all locking associated with a given concurrency protocol. Deadlocks are handled using timeouts.

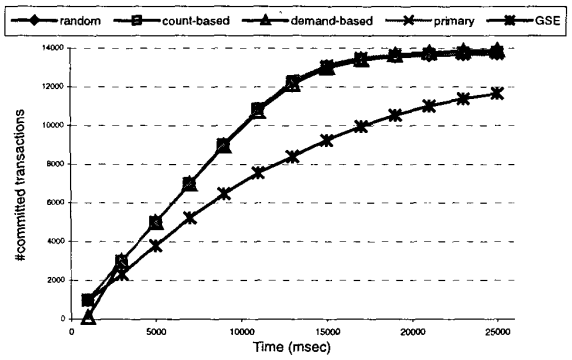

Figure 4: Number of committed transactions uniform, 10 servers, 100 trans/s

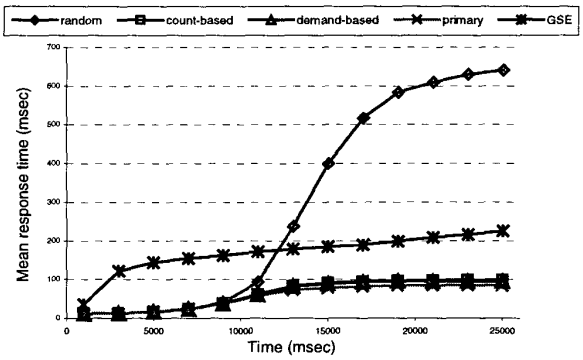

Figure 5: Mean response time uniform, 10 servers, 100 trans/s

Client transaction generator module. The *client transaction generator* models a population of clients submitting requests to the system. A client request involves a single commodity with a specific number of tokens (e.g., buying two tickets for a particular show). Infrequently, the tokens obtained from the system are returned (e.g., the return of a book, cancellation of a ticket). The commodity to be requested is selected uniformly randomly, and the number of tokens to request is chosen uniformly from a given range. Each successfully executed transaction potentially has a *return* transaction that returns the tokens obtained by the original transaction. A return transaction is submitted to the system with a given probability. Transactions are initiated using exponential inter-arrival times.

Network module. The *network* module models Internet communication latencies among the servers. Rather than using a synthetic model to generate communication latencies and inject certain failure modes (e.g., message losses, network partitions, etc.), we sampled messaging latencies over the Internet. For a period of three days, we continuously collected traces of pair-wise ICMP (i.e., ping) message exchange among four servers, which are located at College Park (Maryland), Murray Hill (New Jersey), Lexington (Kentucky) and Santa Barbara (California). These traces are then used to model messaging latencies among the servers (a description of the traces and how they are used can be found in [2]).

The use of wide-area ICMP traces as described is a reasonable technique for our purposes, since (1) our model consists of servers communicating via message exchange over a wide-area network, and (2) the protocols we study require the transfer of short control messages only (but not any data messages). Table 1 shows the main simulation parameters and their default settings. Many of the parameter settings are fixed based on numerous preliminary sensitivity experiments, which we discuss in detail in [2].

6 Performance experiments and results

This section present the results of our performance experiments comparing our DVP strategies and the (extended) GSE approach. We first discuss two modifications to the basic GSE approach that significantly improved its performance in our experiments. Our first modification, which was suggested in [7], involves the construction of a quorum *in parallel* rather than sequentially. Our second modification requires a server to send gossip messages to all others as it learns of a token allocation. All the results reported below are the averaged results of ten independent runs.

6.1 Basic performance

The graphs we present in this section demonstrate how the performance of the system changes over time. As to be expected, the performance of all the approaches becomes worse as the number of tokens globally available in the system decreases. In all the experiments, we fix the number of servers at 10 and the mean inter-arrival time for client transactions at 10 ms.

Uniform workload. Figure 4 presents *num-commits*, the number of committed transactions, by GSE and the DVP strategies under a workload where the transaction requests are uniformly distributed across servers; i.e., $\theta = 0$. The figure reveals that the DVP strategies deliver significantly higher *num-commits* than GSE. For all the DVP strategies, *num-commits* initially increases rapidly. With each committed allocation transaction, however, *avail*, the number of tokens globally available, decreases. *num-commits*, then, settles down as very few transactions can commit due to a lack of tokens in the system. The differences among different DVP approaches here are not significant. Figure 5 shows the complementary *resp-time*, mean response time, results. All the DVP strategies, except `random`, outperform GSE throughout the entire execution range. These DVP variants manage to keep their *resp-time* quite low, whereas the performance of GSE deteriorates quickly after several hundred transactions are executed. The *resp-time* of `random` drastically increases as *avail* decreases since it selects the lender servers randomly and cannot tell whether a server has any tokens or not. The other DVP strategies do not suffer from the same problem, achieving much better *resp-time* values.

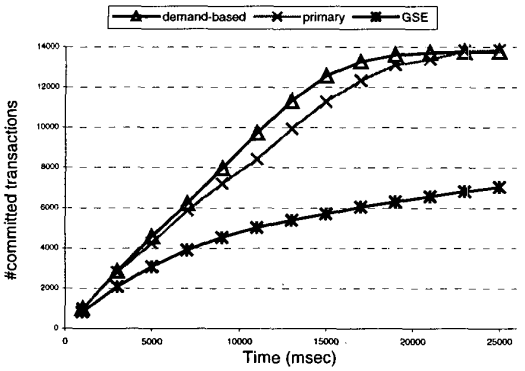

Figure 6: Number of committed transactions skewed, 10 servers, 100 trans/s

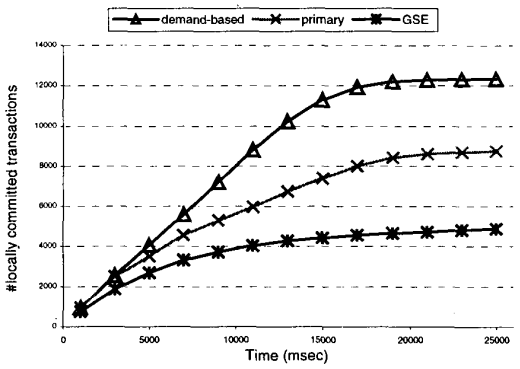

Figure 7: Number of locally executed transactions skewed, 10 servers, 100 trans/s

The GSE approach, as explained in Section 4, operates by forming a quorum of servers. It thereby requires tight quorum synchronization, which is quite costly in a wide-area environment. The DVP approaches do not require quorum synchronization: they can continue executing transactions locally as long as they have sufficient tokens at their disposal. Even in the case when a server runs out of sufficient tokens and has to make requests to other servers, it never has to communicate synchronously with multiple servers.

A close look at the raw results clearly demonstrates the fundamental drawbacks of GSE: (1) GSE cannot execute as many transactions locally as DVP approaches, and (2) quorum sizes increase as *avail* decreases. GSE, therefore, not only has to contact other servers most of the time, but also has to lock more and more servers as *avail* decreases, aggravating its inefficiency. Another problem of GSE is its high abort rate, which occurs mainly due to timeout in lock waits.

Skewed workload. Figure 6 presents the *num-commits* achieved by GSE and the DVP strategies under a skewed workload where servers receive transactions based on a highly-skewed Zipf distribution; i.e., θ =1. We drop `random` and `count-based` from presentation in the rest of the graphs, as they are consistently outperformed by the other DVP approaches. Comparison of these results with those for the balanced case immediately shows that (1) all approaches are negatively impacted by the high skew in the workload, (2) DVP approaches still significantly outperform GSE, and (3) `demand-based` achieves the highest *num-commits*.

For all the approaches, a few *popular* servers perform most of the token allocations due to the workload skew. In GSE, although the escrow sizes vary dynamically, the popular servers exhaust the tokens in their local escrows rapidly, having to form quorums most of the time. In DVP, the popular servers consume their local tokens quickly, and then have to contact the less popular servers in order to obtain tokens.

Comparing the individual DVP strategies, observe that `demand-based` achieves the highest *num-commits* due to its ability to redistribute tokens dynamically across servers via sharing and prefetching; i.e., by continuously transferring tokens from the less popular servers to more popular ones. The `count-based` and `primary` strategies provide virtually equivalent performance. Figure 7, which shows the number of transactions that are executed *locally* without contacting other servers, further justifies this behavior. In terms of response time (not shown), `demand-based` also outperforms the other approaches.

6.2 Other experiments

In the rest of the section, we briefly discuss additional experiments, which we only summarize here due to space limitations (complete results can be found in [2]).

Scalability and contention effects. We conducted experiments where we varied the number of servers and transaction rates. We observed that the DVP approaches scaled and performed better than GSE — `primary` demonstrated the best scalability since its performance is not affected much by the number of servers once the system switches to centralized mode.

Non-deterministic lender selection. The DVP strategies we presented are deterministic in that they use the estimated token counts at servers to choose the lender servers. This determinism may lead to a situation where, for a given period of time, most of the token requests are targeted to the same server — making that server a *hot-spot*. Even though the probability of hot-spots did not turn out to be significant (no more than .15 higher than the random selection case on average), we further studied two randomized schemes to eliminate them. The first scheme chooses the lenders randomly with equal probability (among those with a non-zero token count). The second scheme is based on the idea of *lottery scheduling* [13], where lenders are chosen with a probability proportional to their token counts. The results showed that the lottery-scheduling scheme performs somewhat better than the random scheme, achieving commit rates similar to those of the deterministic scheme, while suffering less than a 20% deterioration in response times on average.

7 Related work

O'Neil proposed Escrow transactions [10] to enable concurrent access to high traffic *aggregate fields* on which only a restricted class of commuting operations (such as incremental updates) are allowed. Escrow transactions execute a special *escrow* operation that attempts to *put in escrow* (i.e., reserve) some of the resources that it plans to acquire. All escrow operations that succeed are logged. Transactions consult this log before executing an escrow operation and see the total amount of resources that are escrowed by all uncommitted transactions. If the total quantity of *un*escrowed resources is sufficient, the transaction proceeds; otherwise it aborts. Haerder [6] extended the escrow transactional model for centralized database environments to DB-sharing systems where multiple DBMSs share the database at the disk level.

Kumar and Stonebraker [9] generalized the notion of escrow transactions for replicated data. Each server is assigned an escrow quantity that can be used to execute transactions locally. The escrow quantities at servers are updated by the use of a periodic global snapshot algorithm, which needs to be executed sufficiently frequently for servers to have an up-to-date view of the global state. On the other hand, frequent execution of such a costly algorithm may itself degrade performance. Both DVP and GSE employ mechanisms that eliminate the need for a global snapshot algorithm.

The work most closely related to our investigation of various redistribution strategies is [8], where Kumar discussed several *borrowing policies* for escrow transactions in a replicated environment. Kumar devised four borrowing policies that (1) select lender servers either randomly or according to a pre-specified order, and (2) that borrow either the exact amount they need or borrow an amount such that the final escrow quantities at the involved servers become equal. Unlike our strategies, however, Kumar's borrowing policies do not make use of the knowledge of the global state of the system to improve its effectiveness and adapt dynamically to workload.

Golubchik and Thomasian discussed demand-driven token allocation schemes in the context of a fractional data allocation method (FDA) [5]. One such scheme they describe enables token partitioning between the involved servers based on demand (as in our `demand-based` strategy). In [12], Thomasian further discussed FDA and proposed an abstract model for optimal *initial* allocation of tokens, which we do not address in this paper. It is worth noting that no previous work has explored the fundamental tension between replication and partitioning for token-based resource distribution, which is our main focus in this paper.

8 Conclusions

Token-based commodity distribution can meet the demands of a class of newly emerging Internet-based e-commerce applications. In this paper, we experimentally evaluated and compared two fundamentally different approaches to token distribution — partitioning and replication — using real Internet message traces. We proposed several pair-wise token redistribution strategies for the partitioning-based approach and experimentally evaluated them under different workloads.

Our experiments reveal a number of significant results for wide-area token-based commodity distribution. First, replication-based approaches are neither necessary nor desirable for the kinds of applications and environment we address in this study: partitioning-based approaches perform and scale better primarily due to their ability to provide higher server autonomy. Second, the use of information about the system state (e.g., token counts, token demand rates) turns out to be crucial for making redistribution decisions. Third, in the case of non-uniform workloads, demand-based redistribution yields notable performance improvements.

References

[1] *CSIM 18 Simulation Engine (C++ Version)*: Mesquite Software, Inc., Austin, TX 78759-5321, 1999.

[2] U. Cetintemel, B. Ozden, M. J. Franklin, and A. Silberschatz. Partitioning vs. replication for token-based commodity distribution. UMIACS-TR-2000-62, University of Maryland, 2000.

[3] M. Colajanni, P. S. Yu, and D. M. Dias. Analysis of task assignment policies in scalable distributed web-server systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(6):585-600, 1998.

[4] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287-313, 1982.

[5] L. Golubchik and A. Thomasian. Token allocation in distributed systems. In *Proc. Intl. Conf. on Distributed Computing Systems (ICDCS)*, Yokohama, 1992.

[6] T. Haerder. Handling hot spot data in DB-sharing systems. *Information Systems*, 13(2):155-166, 1988.

[7] N. Krishnakumar and A. J. Bernstein. High throughput escrow algorithms for replicated databases. In *Proc. Very Large Databases (VLDB)*, Vancouver, 1992.

[8] A. Kumar. An analysis of borrowing policies for escrow transactions in a replicated data environment. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, 1990.

[9] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated databases. In *Proc. Intl. Conf. on Management of Data)*, 1988.

[10] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405-430, 1986.

[11] N. Soparkar and A. Silberschatz. Data-value partitioning and virtual messages. In *Proc. Symposium on Principles of Database Systems*, Nashville, 1990.

[12] A. Thomasian. Fractional data allocation method for distributed database systems. In *Parallel and Distributed Information Systems (PDIS)*, Austin, 1994.

[13] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *Proc. Symp. on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, November 1994.