Partitioning vs. Replication for Token-Based Commodity Distribution

Uğur Çetintemel[†‡]

Institute for Advanced Computer Studies,
Department of Computer Science,
University of Maryland,
College Park
ugur@cs.umd.edu

Banu Özden

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

Michael J. Franklin

Computer Science Division, EECS
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 main approaches for distributing such tokens — replication and partitioning. Token replication requires expensive distributed synchronization protocols to provide data consistency, and is subject to both high latency and blocking in case of network partitions. On the other hand, token partitioning allows many transactions to execute locally without any global synchronization, which results in low latency and immunity against network partitions.

In this paper, we examine the Data-Value Partitioning (DVP) approach to token-based commodity distribution. We propose novel DVP strategies that vary in the way they redistribute tokens among the servers of the system. Using a detailed simulation model and real Internet message traces, we investigate the performance of our DVP strategies by comparing them against a previously proposed scheme, Generalized Site Escrow (GSE), which is based on replication and escrow transactions. Our experiments demonstrate that, for the types of applications and environment we address, replication-based approaches are neither necessary nor desirable, as they inherently require quorum synchronization to maintain consistency. We show that DVP, primarily due to its ability to provide high server autonomy, performs favorably in all cases studied.

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 Eminem, economy class tickets for a particular flight, etc.). Second, the operations of interest on these items

[†] Part of this work was done while the author was working at Bell Laboratories, Lucent Technologies Inc.
[‡] The full address of the contact author is: Uğur Çetintemel, Department of Computer Science, University of Maryland, College Park, MD 20742, USA. Tel: (301) 405 2717. Fax: (301) 405 6707. E-mail: `ugur@cs.umd.edu`

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 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 novel 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 a previously proposed scheme, Generalized Site Escrow (GSE) [7], which is based on replication and escrow transactions [10]. GSE generalizes 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. Second, although the basic approaches, DVP and GSE, were both developed a number of years ago, this paper 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 token-based commodity distribution.

The remainder of the paper is organized as follows. In Section 2, we briefly describe the system model and the base GSE and DVP algorithms. In Section 3, we propose several token redistribution strategies for DVP. 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 offer concluding remarks in Section 8.

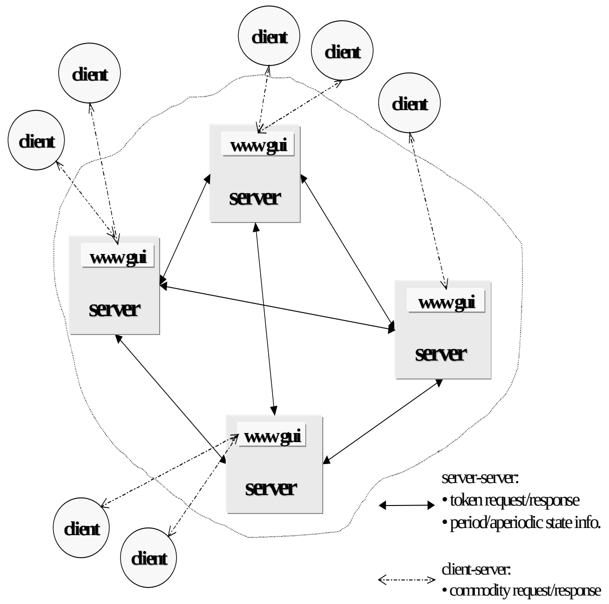

Figure 1: System model

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. For brevity, we leave aside many significant properties such as recovery and concurrency control, focusing mostly on the performance-related issues. Details of DVP can be found [11].

2.1 System model

Our reference system model, illustrated in 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*, which denotes 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.

```
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

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.).

The basic idea underlying DVP is to split up the values of database items and store each of the constituent values (i.e., tokens) at different servers. Transactions are then executed locally at each server using the tokens locally available at that server. Only 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. The basic DVP algorithm is presented in Figure 2. We refer to the number of tokens available at s_i as tok_i. The number of tokens required to execute transaction t is $req(t)$, 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 accomplished. 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 the rest of the paper. In particular we avoid global strategies that require tight synchronization, and concentrate only on decentralized 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. 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, BASIC. 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:

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

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

3.2 Token count-based redistribution

The BASIC strategy makes *blind* redistribution decisions, since it does not maintain or utilize any information about the states of other servers. The SMART 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 $tstamp_j^i$, $j=1\ldots n$, and n is the number of servers.

SMART 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, $k \neq i$:

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

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

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 6.3.2 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_i 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 SHARE/PREFETCH 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, $j=1\ldots n$. 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.

SHARE/PRETECH 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:

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

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 as follows:

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

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

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 that is available at the two servers redistributing their tokens, it is quite robust in that it *in-*

crementally migrates the existing *global* token distribution in the system to match the *global* relative request rate distribution. Furthermore, we experimentally proved that, using this pair-wise mechanism, any 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). A similar pair-wise exchange mechanism has been proposed [2] in the context of Deno, an object replication system based on an epidemic weighted-voting protocol, in order to migrate existing weight distributions to target weight distributions.

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 SMART 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.

In this strategy, each commodity is assigned a primary server that is responsible for satisfying all the requests involving the tokens of a particular commodity when the system operates in the centralized mode. Each server continuously observes the global 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

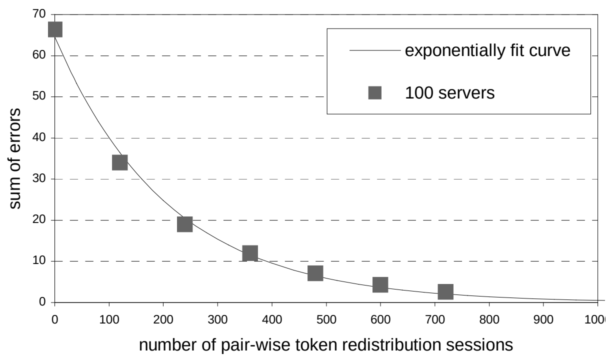

Figure 3: Incrementally converging to global target token distributions via pair-wise token exchanges

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.[1]

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.

In order to implement this scheme, GSE relies on: (1) *gossip messages*, which are periodic background

[1] An alternative model, which 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], enables client requests to be submitted directly to the corresponding primary. This latter model eliminates the (useless) initial request to a *non*-primary server.

messages that include the token allocations known by the sender server, in order 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.

A server s_i can perform token allocations as long as es_i, which s_i updates dynamically 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 will demonstrate, is relatively expensive to perform in wide-area and is the main performance bottleneck of replication-based approaches such as GSE.

Figure 4 depicts the basic GSE algorithm. Initially, only s_i is in the quorum of t (denoted by Q_t). Step 2 computes the escrow quantity es_i. The tight synchronization among the servers in Q_t enables s_i to use the escrow quantities of the other servers in Q_t. Since s_i is not up-to-date regarding the token allocations performed by the other servers in the system, it, thus, makes a conservative estimate regarding its escrow size. Server s_i makes this estimate by placing an upper bound on the escrow values assumed by non-quorum servers; thereby guaranteeing that simultaneous token allocations do not cause a lower-bound constraint violation. In order to make this estimate, s_i computes U_i, which denotes the set of token allocation transactions t_u such that t_u is known to s_i, and s_i is not sure that t_u is known to all the non-quorum servers. The sum

$$req(t) + \sum_{t_u \in U_i} req(t_u)$$

provides an upper bound on the number of token allocations that might be unknown to non-quorum servers. The inequality in Step 4 checks whether all but (at most) the last $es_i - req(t)$ allocations known to s_i are known to all the non-quorum servers. If this inequality evaluates to true, it means that es_i is insufficient; additional servers need to be added to Q_t. This is accomplished by making (remote) lock requests to involved servers regarding their views of *avail*, $avail_j$ for s_j). During this locking phase, the views of *avail* at the quorum servers are synchronized (i.e., the *avail* values at the quorum servers become equal). If es_i is still not sufficient and all servers are already in the quorum (Step 4), the transaction is aborted. If es_i is sufficient, the execution of t can proceed.

After all the necessary updates and logging are performed, t commits and all local and remote locks are released. It is important to emphasize that the performance of GSE heavily depends on how up-to-date each server is regarding the global token allocations performed and the cost of synchronization among servers

```
1.  Lock $avail_i$ and set $Q_t = \{s_i\}$.
2.  Set $es_i = \lfloor avail_i / n * |Q_t| \rfloor$.
3.  Compute $U_i$.
4.  If $es_i < req(t) + \sum_{t_u \in U_i} req(t_u)$
        If $|Q_t| < n$
            Lock $avail_j$ of a server $s_j \notin Q_t$ and add $s_j$ to $Q_t$
            Goto Step 2.
        Else abort; and for all $s_j \in Q_t$, unlock $avail_j$.
5.  Perform requisite updates
6.  Commit; and for $s_j \in Q_t$, unlock $avail_j$
```

Figure 4: The GSE algorithm implemented by s_i to execute transaction t

5 Experimental environment

5.1 Simulation model

In order to evaluate the performance of GSE and the DVP strategies, we implemented a detailed simulation model using CSIM [1]. The model consists of components that model a distributed set of servers, a population of clients making commodity requests, and communication latencies among servers.

5.1.1 Server module

The server module is responsible for executing client transactions using either DVP or GSE. It consists of:

- a *resource manager*, which schedules the CPU and disk (using a non-preemptive FIFO policy);
- a *buffer manager*, which handles transfer of data between the disk and buffer;
- 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, specified in terms of the number of instructions executed. No per-message-byte cost is modeled since we assume the use of short, fixed-sized control messages; and
- a *transaction manager*, which coordinates the execution of transactions.

We now describe the transaction manager in more detail. The transaction manager consists of several components. One component is the *lock manager*, which handles all locking associated with a given concurrency protocol. Deadlocks are handled using a timeout mechanism. The transaction manager also contains several algorithm-specific components. The *escrow manager* maintains all necessary information and makes all the quorum-based decisions when modeling the GSE algorithm. It also contains a *gossip manager* component that coordinates the sending of gossip messages to other servers. The other algorithm-specific component is the *DVP manager*, which is responsible for implementing the redistribution strategies that we described. The DVP manager also contains a *gossip manager* component that is responsible for disseminating state information via message exchange. Every message sent or received by a server is charged a fixed CPU cost. No per-message-byte cost is modeled since we assume the use of short, fixed-sized control messages.

5.1.2 Client transaction generator module

The *client transaction generator* models a population of clients submitting transactions to the system. The model generates transactions using exponential inter-arrival times and submits them to the network module for delivery to a server.

We model the transactions based on common characteristics of wide-area commodity distribution applications we discussed in Section 1. A client request involves a single commodity[2] 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 transactions we generate, therefore, specify a particular commodity and a value indicating the number of tokens requested of that commodity.

The commodity to be requested is selected uniformly among all the commodities in the database, and the number of tokens to request is chosen uniformly from a given interval. 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. Return transactions, if submitted, are issued by the client that previously submitted the corresponding allocation transaction.

5.1.3 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).

The traces were then formatted into lists whose entries are the latencies of each message exchange or an indication that the message was lost (approximately 2% of the messages were lost).

The server-server traces are used to drive the network component of the simulations as follows. At the beginning of a simulation run, one trace is randomly (uniformly) chosen for each pair of servers, and one entry is selected randomly (uniformly) on each trace as a starting point. Whenever a server sends a message to another server, the current latency value for the corresponding trace is read and the current entry is incremented. The message is delayed by this latency value and then inserted into the message queue of the destination server. If the entry indicates a message loss, then the message is re-sent after a timeout period of 200 ms (twice the average round-trip time between any pair of servers).

In order to model the communication latencies between servers and clients, we used a similar approach. We used client machines whose IP addresses were obtained from the web-server traces of a large telecommunications company. We chose 400 machines that were scattered over the Internet. We sent control messages from our four servers to these clients for a period of three days, and logged the latency values observed. The communication between the clients and the servers consists of a short client message that includes the transaction requested and a short response message from the server indicating the result of the transaction submitted. Thus, the sampled latencies can reasonably be used in modeling the client-server communication in our environment. The use of a client-server trace is similar to that described above for the server-server case.

The use of wide-area ICMP traces as described is a reasonable technique for our purposes, since (a) our

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

[2] We do not investigate transactions that involve multiple commodities in this work. Although it is straightforward to generalize the algorithms for that case, such a generalization is beyond the scope of this paper.

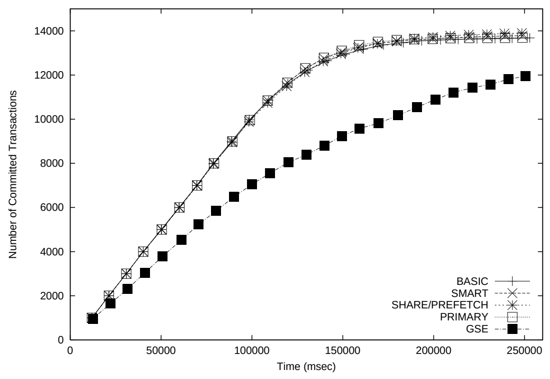

Figure 5: Number of committed transactions uniform, 10 servers, 100 trans/s

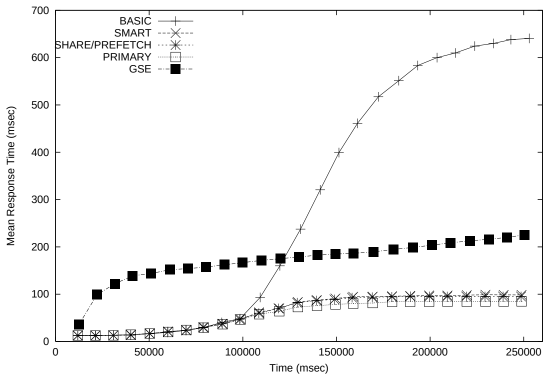

Figure 6: Mean response time uniform, 10 servers, 100 trans/s

model consists of servers communicating via message exchange over a wide-area network, and (b) the protocols we study only require the transfer of control messages but not any data messages. Table 1 shows the main simulation parameters and their default settings.

5.2 Methodology

All the results reported in the following section are the averaged results of ten independent simulation runs. For each set of runs, the same set of ten workloads (i.e., the same clients submitting the same requests at the same time) are used when comparing different approaches to ensure fairness. The only factor that results in different loads for the different approaches is the existence of return transactions. Since the scheduling of a return transaction is dependent on whether or not the corresponding allocation transaction commits, different approaches will typically see different return transactions. In particular, the higher the commit rate for an approach, the more return transactions that approach will observe.

Note that there are two reasons why a transaction may *not* commit in our environment. First, a transaction may timeout waiting for a local or remote lock and abort. Second, either there may not be sufficient tokens in the system to execute the transaction successfully or there may be sufficient tokens but the system may not be able to locate them in a timely manner.

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. Both modifications significantly improved the performance of GSE in our experiments.

The first modification addresses the *parallel* locking of the quorum servers. As suggested in [7], rather than requiring each server to construct a quorum sequentially, it is possible to estimate an initial quorum size and lock the quorum servers in parallel. If the quorum needs to be enlarged, a new set of non-quorum servers are added to the quorum and locked in parallel. The details about quorum size estimation can be found in [7].

The second modification involves the use of background messages. Recall that the performance of GSE depends heavily on the extent to which servers are up-to-date regarding the token allocations performed in the system. In the basic GSE approach, servers periodically broadcast their states to the other servers. In order to ensure that servers have the most up-to-date information possible, we modified the algorithm so that these background messages are sent whenever an update is committed. When a server receives such an update notification, it also notifies the other servers that it is aware of that update. While this approach is not likely to be practical for a real implementation, it improves the performance of GSE in our experiments because (1) the messages are sent in the background, not in the critical path of updates and (2) the messages are all short control messages. Thus, the messages have little or no negative impact on update performance, while providing the benefit of up-to-date information to all of the servers[3]. By implementing this aperiodic strategy, we give GSE an unfair advantage over DVP, which uses far fewer messages for disseminating state information, for the purposes of these experiments.

6.1 Basic performance

The graphs we present in this section demonstrate how the performance 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. Most of the measurements shown are *cumulative* measures. For example, the response time

[3] We also experimented with increasing the frequency of the periodic messages of the basic GSE approach. The results showed, however, that for this environment, the aperiodic approach provides better performance with fewer messages than the periodic one.

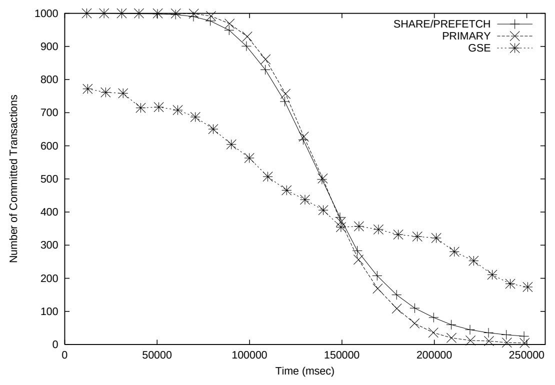

Figure 7: Number of committed trans. (instantaneous) uniform, 10 servers, 100 trans/s

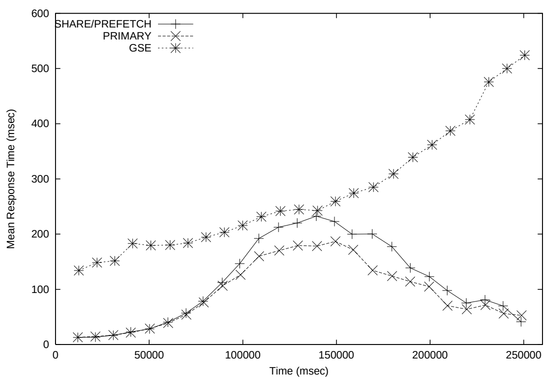

Figure 8: Mean response time (instantaneous) uniform, 10 servers, 100 trans/s

value that is plotted at time 10,000 ms is the mean of the response times for all transactions that have committed in the first 10 (simulated) seconds of the experiment. Whenever insightful, we also present *instantaneous* results that depict the performance of the approaches during a certain execution window. In all the experiments, we fix the number of servers at 10 and the mean inter-arrival time for client transactions at 10 ms. At this inter-arrival rate, all of the approaches demonstrate stable behavior.

6.1.1 Uniform workload

We now show the results of the algorithms under a uniform workload where the transaction requests are uniformly distributed across servers (i.e., θ=0).

Figure 5 presents *num-commits*, the number of committed transactions, by GSE and the DVP strategies. The figure reveals that the DVP strategies deliver a significantly higher *num-commits* than GSE does. 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 6 shows the complementary *resp-time*, mean response time, results. All the DVP strategies, except BASIC, 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 BASIC 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.

The GSE approach, as explained in Section 4, operates by forming a quorum of servers. It therefore requires tight synchronization among the quorum servers. Such synchronization turns out to be quite costly in a wide-area environment. The DVP approaches, however, do not require 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.

In order to gain more insight, we present the *instantaneous num-commits* results for GSE, PRIMARY, and SHARE/PREFETCH in Figure 7. Each point represents the number of committed transactions during a time window (of length 30,000 ms) that is centered at the time value on the x-axis. This figure, in contrast to Figure 5 that demonstrates the *cumulative* behavior of the system over time, presents the number of transactions committed during certain execution periods. Initially, *num-commits* for the DVP approaches are significantly larger than that of GSE. As *avail* decreases rapidly, *num-commits* also decreases. After about 150 seconds, GSE is able to commit more transactions than the DVP approaches. This is because, during that execution range, *avail* for the DVP approaches is much smaller than *avail* for GSE, making it (relatively) more difficult for the DVP approaches to commit transac-

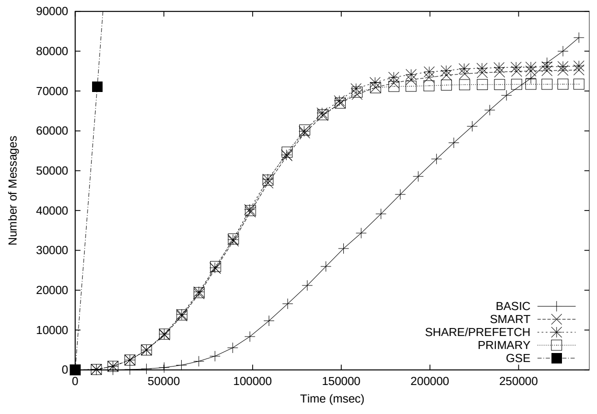

Figure 9: Number of messages sent uniform, 10 servers, 100 trans/s

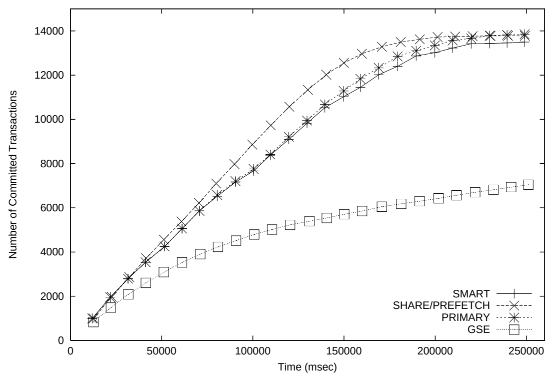

Figure 10: Number of committed transactions skewed, 10 servers, 100 trans/s

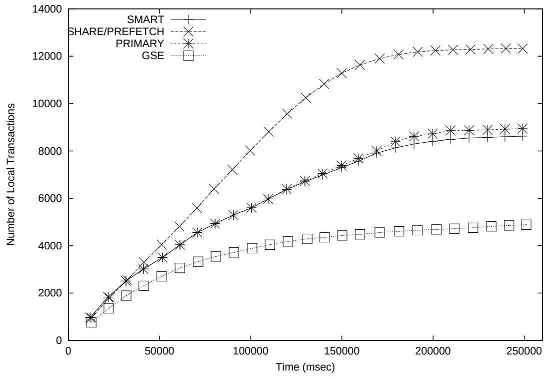

Figure 11: Number of locally executed transactions skewed, 10 servers, 100 trans/s

tions. Eventually, *num-commits* for the DVP approaches drop close to zero, as *avail* goes to zero. At the end of the execution range shown, *avail* is still high for GSE, so GSE can still continue to commit transactions.

Corresponding *instantaneous resp-time* results are shown in Figure 8. The figure demonstrates that both DVP strategies deliver better response times than GSE during all execution periods. Note the difference between the shapes of the *resp-time* curves of GSE and the DVP approaches. The *resp-time* of the DVP approaches initially increases as the number of locally available tokens decreases, reaching a peak. The *resp-time* values then begin to decrease as *num-commits* decrease and *avail* becomes so small that most of the transactions that commit are local ones (which have low response times). It can be seen that PRIMARY performs slightly better than SHARE/PREFETCH. The *resp-time* for GSE, however, increases gradually as the quorum sizes increase with decreasing *avail*.

A closer look at the raw results clearly demonstrates the fundamental drawbacks of GSE: (1) GSE cannot execute as many transactions locally as DVP approaches do, and (2) the size of the quorum that a server has to form also increases 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. The mean quorum size for GSE, which is the average number of quorum servers per transaction, is 1.1 in the initial 10 seconds, 2.1 in the period of [90,100] seconds, and 3.7 in the period of [190,200] seconds. Another problem of GSE is its high abort rate, which occurs mainly due to timeout in lock waits. Lock waits tend to become a serious problem in a wide-area system as the communication latencies are unpredictable.

DVP approaches, on the other hand, can execute transactions locally most of the time, achieving higher commit rates and lower response times by avoiding latencies and delays typically encountered during inter-server communication.

Figure 9 presents *num-msgs*, the number of messages sent, for all of the approaches studied here. Note that this metric includes all messages exchanged during the operation of the system. GSE sends a large number of messages compared to the DVP strategies. Even with this large volume of gossiping among servers, GSE cannot perform comparably to DVP. BASIC initially sends fewer messages than the other DVP strategies, as it does not disseminate state information. As *avail* decreases, however, BASIC begins to suffer from its lack of information about the states of other servers, having to choose the servers to ask for tokens randomly. The other DVP strategies make smarter decisions about which servers to contact (or not to contact) and succeed in limiting their *num-msgs*. Beyond a certain point, PRIMARY uses slightly fewer messages than the other DVP approaches because once the system switches to centralized mode, all the tokens are handled by a single server, avoiding the need to contact multiple servers to gather tokens.

6.1.2 Skewed workload

In this section, we study the case where servers receive transactions according to a highly-skewed Zipf distribution (i.e., θ =1), as opposed to the uniform distribution case studied before.

We present the *num-commits* achieved by GSE and the DVP strategies under the skewed workload in Figure 10 (we drop BASIC from presentation in the rest of the paper, as it is 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) the SHARE/PREFETCH strategy achieves the highest *num-commits*.

For all the approaches, a few *popular* servers perform most of the token allocations due to the skew in the workload. In GSE, although the escrow sizes vary dynamically, the popular servers exhaust the tokens in

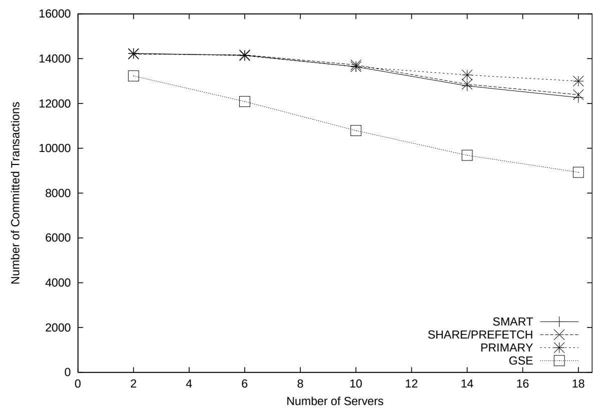

Figure 12: Number of committed transactions uniform, varied servers, 100 trans/s

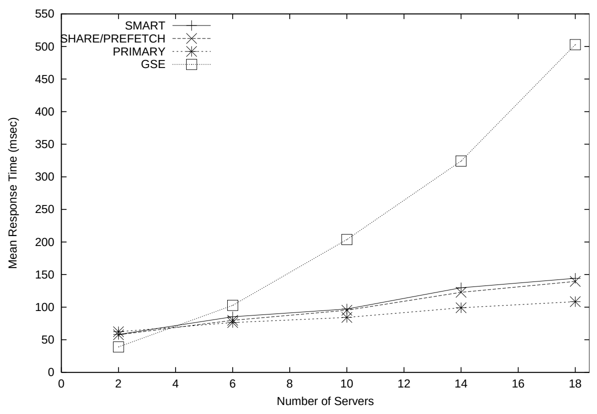

Figure 13: Mean response time uniform, varied servers, 100 trans/s

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 SHARE/PREFETCH achieves the highest *num-commits*. This superior performance is due to its ability to redistribute tokens dynamically across servers via sharing and prefetching based on request rates; i.e., by continuously transferring tokens from the less popular servers to more popular ones. SMART and PRIMARY provide virtually equivalent performance. Figure 11, which shows the number of transactions that are executed locally without contacting other servers, further justifies this behavior. In terms of response time SHARE/PREFETCH also outperforms the other approaches for most of the experiment (not shown).

The results of the experiments presented so far reveal that: (1) the proposed DVP strategies significantly outperform GSE under various workloads; (2) SMART, SHARE/PREFETCH, and PRIMARY outperform BASIC mainly because they exploit state information; (3) when there is a balanced workload, SHARE/PREFETCH and PRIMARY demonstrate the best performance, while PRIMARY demonstrates somewhat better response times; and (4) under skewed workloads, SHARE/PREFETCH compares favorably to other schemes as it adaptively redistributes tokens based on the loads on servers.

6.2 Scalability

6.2.1 Varying number of servers

We now investigate how the approaches perform as we scale the number of servers in the system. All the results presented here are the mean values of the relevant metrics at 200 seconds. The choice of 200 seconds is not arbitrary. In fact, at around 200 seconds, the results for the commit rate and response time related metrics tend to stabilize for all approaches. As before, we fix the transaction interarrival time at 10 ms. We present only the *num-commits* and *resp-time* results under the uniform workload.

Figure 12 shows the *num-commits* when varying the number of servers. All approaches presented are negatively affected as the system size is increased. The reasons for this result are twofold. First, the workload consists of update transactions only. Second, at the request rate for which we present the results, the performance of the system is not CPU or I/O bound, but is largely synchronization bound. The well-known benefits of using a distributed system to distribute the workload, thus, are not apparent here.

The *num-commits* value for GSE drops drastically with increasing number of servers. We observe that *num-commits* for GSE always lies below those of the DVP strategies. The DVP strategies scale better; with PRIMARY being the one with the best scalability as it is quite robust for regions with small *avail* regardless of the number of servers in the system. SMART and SHARE/PREFETCH perform similarly to each other.

Figure 13, which presents the *resp-time* results, shows that GSE has the best response time for the case where there are two servers. Its performance, however, degrades and falls behind those of the DVP approaches very rapidly with increasing number of servers. In addition, its *num-commits* is below those of the DVP strategies for all system sizes shown (including the two-server case). This is not surprising because as the number of servers increases, synchronization among them becomes increasingly more difficult and expensive. The DVP results are consistent with the corresponding *num-commits* results.

6.2.2 Varying transaction rate

Next, we investigate the performance of the approaches as the transaction interarrival time is varied. Similar to the previous case, each point plotted is the mean value of the corresponding metric at 200 seconds. As before, we fix the number of servers at ten.

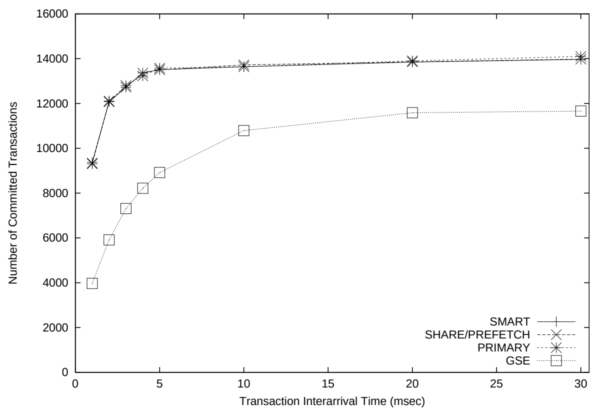

Figure 14: Number of committed transactions uniform, 10 servers, varied transaction rate

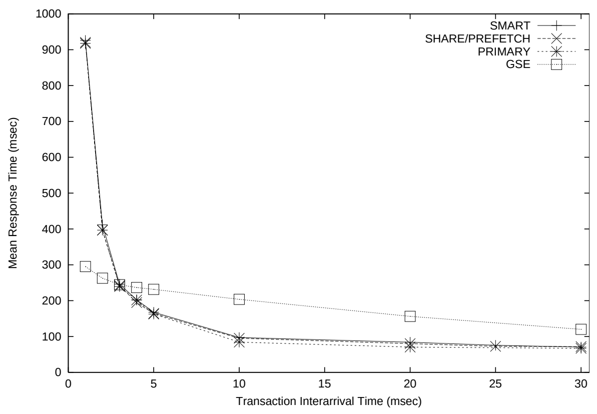

Figure 15: Mean response time uniform, 10 servers, varied transaction rate

We show the *num-commits* results for the GSE and the DVP strategies in Figure 14. It can be seen that for low transaction interarrival times (i.e., high transaction arrival rates), *num-commits* achieved by GSE and the DVP approaches deteriorates drastically. The system cannot sustain interarrival times lower than about two-three ms, below which the performance worsens substantially. For these and lower values, the state of the system changes very rapidly and the state information maintained at each server becomes outdated quickly. Another factor contributing to low *num-commits* is the increased lock contention at servers which results in a high number of transaction aborts. All systems recover as the interarrival times are increased, and, in fact, their performance does not improve significantly above an interarrival time of five ms. We observe similar results for all the DVP strategies.

Examining Figure 15, we observe (relatively) high *resp-time* values for the DVP strategies for interarrival times below three ms (for which the system is not stable). Although GSE seems to do a better job in limiting its *resp-time* values for interarrival times below three ms, its *num-commits* is much lower than those of the DVP strategies for all the interarrival values shown.

6.3 Other experiments

In the remainder of the section, we briefly discuss additional experiments that we conducted in order to explore potential improvements to the partitioning strategies discussed earlier.

6.3.1 Load balancing

Comparison of the results for the balanced and skewed workload cases reveal that the DVP approaches suffer, although not as much as GSE, from load imbalance. We, therefore, investigated the potential benefits of employing a simple *load balancing* mechanism to distribute the load evenly across servers. In this mechanism each client, rather than submitting the transaction to its *closest* server, *randomly* picks a server to submit the transaction[4]. Servers use the SHARE/PREFETCH strategy — or any other strategy for that matter — without any modifications. The advantage of such load balancing is that the workload seen by each server will (on average) be similar. In such a case, as we observed from the uniform workload results, the DVP approaches will demonstrate significantly better performance. The downside is that such a randomized selection eliminates the potential gains of submitting the transaction to the closest server. This restriction potentially results in increased *client-side* response times. Experimental results demonstrate that this is indeed the case: Our simple load balancing technique achieves better server-side response times than the *no*-load balancing strategy. In fact, it achieves results quite similar to those of SHARE/PREFETCH for the uniform workload case. Client-side response times, however, turn out to be similar, or slightly better for the *no*-load balancing case, demonstrating the aforementioned trade-off.

6.3.2 Non-deterministic selection of lenders

The DVP strategies presented are deterministic in that they use the estimated token counts at servers to choose the lender servers. This deterministic selection 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*. Our experiments showed that, due to the differences among the views at different servers, such situations do not occur very frequently: the probability is no more than 10% and 30% higher than the random selection case on average for the uniform and skewed cases, respectively.

In order to eliminate such situations completely, we studied two randomized schemes. The first scheme chooses the lenders randomly with equal probability (among those with a non-zero token count). The other

[4] Note that this is a very practical and easy-to-implement scheme.

scheme is based on the idea of *lottery scheduling* [13], where lenders are chosen with a probability proportional to their token counts. Experiments reveal that the lottery-scheduling scheme performs somewhat better than the completely random scheme, achieving commit rates similar to those of the deterministic scheme, while suffering less than a 20% deterioration in response time under both the uniform and skewed workloads (using SMART).

7 Related work

Most previous related work focused on exploiting application semantics to improve performance in certain classes of applications. O'Neil proposed Escrow transactions [10] to enable concurrent access to high traffic *aggregate fields* on which only a restricted class of operations (such as incremental updates) are allowed. This technique utilizes the commutativity property of such operations; i.e., two operations can run in any order and still produce the same final result provided that they do not violate upper/lower bound constraints on the involved data items. 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 extended the escrow transactional model for centralized database environments to DB-sharing systems [6] where multiple DBMSs share the database at the disk level. He proposed the use of a hierarchical escrow scheme that consists of a global escrow and local distributed escrows. He discussed a technique that enables the hierarchical scheme to behave like a purely global scheme for only a critical margin of aggregate values.

Kumar and Stonebraker generalized the notion of escrow transactions for replicated data [9]. Each server is assigned an escrow quantity that can be used to execute transactions locally. The escrow quantities at servers are readjusted by the use of a periodic global snapshot algorithm. This algorithm has 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 simple 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. Note that one of the policies, in which the lender server is chosen randomly and the exact amount needed is requested, is similar to our BASIC strategy. 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 SHARE 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 also proposed several pair-wise token redistribution strategies for the partitioning-based approach and evaluated them under different workloads.

Our experiments reveal a number of significant results for 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 turns out to be crucial for making token redistribution decisions. Third, when there is a balanced load on the servers, the use of complicated redistribution schemes does not merit their complexity; simple strategies can be as effective, provided that they utilize minimal state information. Finally, in the case of skewed workloads, however, the use of extra information about request rates may yield notable performance improvements.

In terms of future work, we are planning to investigate hierarchical server organizations, which can improve the performance significantly when the number of servers is large.

References

[1] *CSIM 18 Simulation Engine (C++ Version)*: Mesquite Software, Inc., 3925 W. Braker Lane, Austin, TX 78759-5321, 1999.

[2] U. Cetintemel and P. J. Keleher. Light-Weight Currency Management Mechanisms in Deno. In *Proc. 10th IEEE Workshop on Research Issues in Data Engineering (RIDE)*, San Diego, February 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)*, Los Angeles, 1990.

[9] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated databases. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, Chicago, 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 (PODS)*, Nashville, 1990.

[12] A. Thomasian. Fractional data allocation method for distributed database systems. In *Proc. Intl. Conf. on 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.