Distributed and Parallel Databases, 9, 5–38, 2001

Time Constrained Push Strategies in Client-Server Databases

VINAY KANITKAR AND ALEX DELIS
Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11201 USA

Received September, 1998; Revised March, 1999; Accepted April 22, 1999

Abstract. The widespread availability of networked environments and the arrival of high-speed networks have rekindled interest in the area of automatic data refresh/update mechanisms. In many application areas, the updated information has a limited period of usefulness. Therefore, the development of systems and protocols that can handle such update tasks within predefined deadlines is required. In this paper, we propose and evaluate two real-time update-propagation mechanisms in a client-server environment. The fundamental difference in these two time-constrained techniques, Client–Push and Server-Push, is in the location where the push-transactions are generated. In both these techniques, and in contrast to conventional methods, we propose the transport of the scripts of updating transactions in order to make client-cached data current. This avoids unnecessary shipments of data over the network. Instead, messages are used to maintain the consistency of cached data. In addition, the propagation of update transaction scripts to client sites is neither periodic nor mandatory, but is instead based on client-specific criteria. These criteria depend on the content of the database objects being updated. We carry out a comprehensive experimental evaluation of the suggested methods as we examine the following aspects: (a) time constrained push scheduling issues, (b) effects of various workloads on real-time push-transaction completion rates (efficiency), and (c) overheads imposed by push-transactions on the regular transaction processing. Our experiments show that Client–Push outperforms Server-Push only for a small number of clients. The opposite is true once the load is increased by attaching a large number of sites per server. The efficiency of the update push protocols is, as expected, dependent on the load on the system as well as the percentage of updates to the database. Surprisingly, the percentage of successfully completed real-time push-transactions is not affected very much by the strategy used to schedule them.

Keywords: push-transactions, client-server databases, update propagation, real-time scheduling

1. Introduction

Most contemporary database and corporate information systems have been built on a base architecture of inter-networked sites [2, 4]. The Client-Server Database (CSD) paradigm, in particular, has become very common in the realization of such systems. The server hosts the database and clients, with the assistance of communication networks, query and update it. In the past, such clients acted as little more than user-interface devices. Now, with the decreasing cost of powerful workstations, clients can also be utilized to perform database processing [5, 40, 43]. Through the use of local memory and disk buffers, frequently accessed data are cached at the clients and subsequent user operations on this data are off-loaded from the server to the clients [7]. This localized processing of transactions results in faster response times but requires more elaborate schemes for concurrency control [5, 40, 43] and crash recovery [11, 25].

An important measure of performance in the client-server model has been the response time in obtaining data from the server and the swift availability of updated data. There has been considerable research in this area with regard to improving server response times without compromising the requirements of data consistency and concurrency control [28, 43]. The increasing availability of dedicated network bandwidth gives us the opportunity to focus on automatic data update/refresh mechanisms. The process of automatically updating cached data or information is called *update-push*, and computing systems that implement such techniques are called *push-servers*. Until very recently, limited processing power at the server and constraints on network capacities have made push-server implementations somewhat limited in scope. Logically, current push mechanisms [8, 30, 35] operate as follows:

- Logical predicates specify the data that client machines want to receive automatic updates to.
- At specific time-intervals, or when the data changes, the updated data is automatically sent (pushed) to the client where it is displayed in place of, or in addition to, the old data.
- All updates to the data occur only at the server.

However, in reality, most push techniques execute in the form of a "programmed" pull. This means that the client site is set to contact the server at periodic intervals and download updates to the data [30]. The major disadvantage of this method is that the frequency with which automatic updates are propagated does not depend on the content of the data, but only on the pre-specified refresh interval. Limiting updates to the server simplifies the push protocol but does not reflect the contemporary mode of operation of client-server database systems, where the data can be updated at the clients as well. In order to address the issue of automatic update propagation in this environment, we propose the implementation of content-based push-servers.

In the discussion of our push techniques, we refer to updates occurring in the underlying CSD as *regular* updates so as to distinguish them from the pushed update transactions (or *push-transactions*). If a regular update causes the value of a data object to breach pre-specified conditions (such as range, lower or upper limits, inequalities, etc.) then the update in question needs to be "pushed" to the interested parties (i.e., clients) in the network. The effect of this *push-transaction* is to bring the data cached at all such sites up to date. Hence, the push mechanism is data-triggered rather than monotonic (i.e., simply based on elapsed time intervals). Another important feature of our proposal is that the effects of regular updates are propagated by shipping those transactions (as scripts) to interested clients where they are re-executed on locally cached copies of the data. Figure 1 shows the three types of transactions in the system: regular updates, queries and push-transactions. Regular updates and queries are part of the underlying client-server database while push-transactions belong to the automatic data refresh mechanism. It is worth noting that at any given time, networked sites can execute all three types of transactions simultaneously.

We describe two different mechanisms to push data updates to interested clients, namely: Server–Push and Client–Push. The basic difference in these two mechanisms is the point at which the push-transactions are initiated. In Server–Push, a regular update at a client is first shipped to the server where the effect of the update is evaluated against the set of client

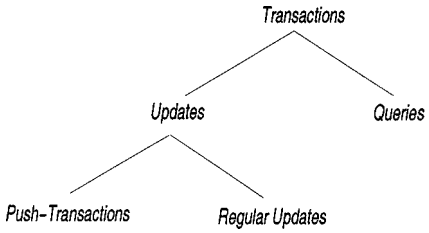

Figure 1. Classification of transactions.

push criteria. If the push criteria for some clients are satisfied then the updating transaction (i.e., the script) is pushed to those clients, thus avoiding unnecessary data movement. In Client–Push, the set of push-criteria attached to an object is initially shipped along with the object itself to a client that has issued an exclusive request. A regular update on this cached object causes the push-criteria to be evaluated at the client itself. The resulting push-transactions are sent to the corresponding clients directly bypassing the server. We would like to emphasize that the term Server–Push should not be confused with the conventional concept of a push-server [30]. A push-server merely indicates push-type of operations instigated anywhere in the network (i.e., server or client).

In our model, clients define their data sets of interest, and provide logical predicates (*push-predicates*) that specify the conditions to be satisfied for automatic updates to their cached copies of that data. Client predicates are in the form of forward-chaining ECA (Event-Condition-Action) rules, which are similar to production rules [14, 32, 42]. We consider events to be updates on the database. Once an event occurs, push-predicates that act as ECA conditions are evaluated. When a client's condition evaluates to true, the action taken is to ship the update to this client. We also impose time constraints on the transmission and execution of push-transactions [29, 31]. This is because in many applications like stock/futures market systems and geographical positioning systems, the usefulness of the data gets drastically reduced as it becomes older [35]. Thus, the deadlines corresponding to each push-criterion are used to impose a *period of usefulness* on the update. We create an extended form of the ECA template by incorporating such time constraints:

$$ON\langle EVENT\rangle\ IF\langle CONDITION\rangle\ THEN\langle ACTION\rangle\ WITHIN\langle DEADLINE\rangle$$

The requirement that updates to the database be communicated to interested users within pre-specified deadlines at all times may be unreasonable given the network infrastructure in today's corporate environments. However, with ever increasing network speeds and bandwidths, such user expectations can be met in the near future [3]. The algorithms used to prioritize time-constrained database transactions have a significant impact on the percentage of transactions that complete successfully [1, 19]. Therefore, in this paper, we consider three policies that have been widely used for scheduling real-time tasks. These policies, along with two additional criteria that we have developed, are described later in the paper. It should be noted that there are no deadline constraints on regular updates or queries. Deadlines are imposed only on push-transactions (i.e., scripts) as our emphasis is on the timely communication of data updates to interested sites within a network.

In order to evaluate the proposed protocols, we first developed detailed simulation packages for a data-shipping client-server database framework. Using this initial testbed, we performed a comprehensive study of the two push protocols under varying workloads. Simulation results indicate that although the Client–Push strategy is better for a small number of clients, Server–Push is clearly more scalable as the number of clients increases. To fully verify our simulation results, we implemented two operational prototypes running on a network of workstations (NOW). The outcome of our prototype experimentation shows very similar trends to those derived through simulation and provides conclusive evidence of the above results.

The paper is organized as follows: in the next section, we describe the data-shipping client-server database model. Section 3 describes the two update-push protocols while Section 4 outlines the simulation and prototype experiments and presents our experimental results. We discuss related work in Section 5. Conclusions can be found in Section 6.

2. The client-server database framework

This section describes the database processing model and locking scheme in a data-shipping client-server database (CSD) [5, 28]. The push-transactions and their supporting mechanisms described in this paper are superimposed on this architecture. Here, the database resides at the server and user transactions are initiated at the clients (figure 2). Client computing systems are expected to have significant processing and storage capabilities. A local area network is assumed to provide transport support for both query/update requests and

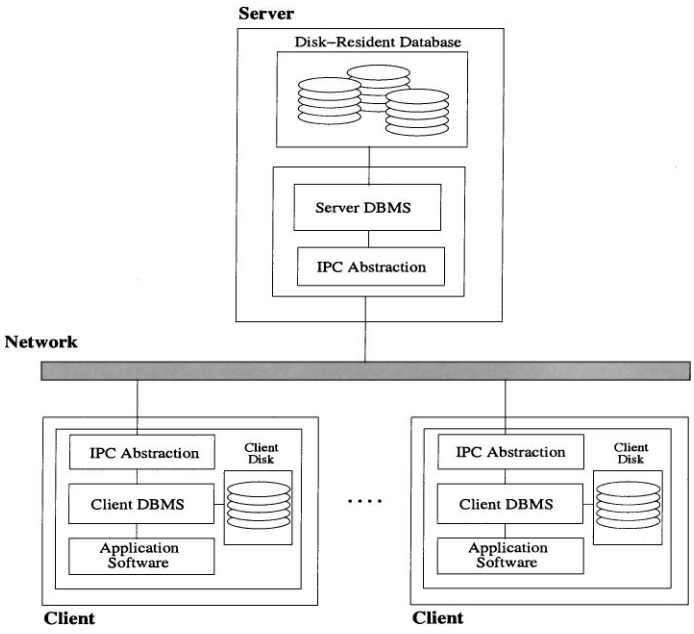

Figure 2. Used client-server framework.

data/result shipping. When a client transaction needs to access a database object, it first checks if the object is cached locally (with the required lock). If it isn't, the server is requested to grant the necessary lock and ship the object over to the client. Once all the requisite data becomes available, the transaction is locally executed at the client. Hence, all the transaction processing (TP) is performed at the clients while the server performs only low-level database functionalities on behalf of the requesting clients. Once data objects have been fetched from the server, they are cached at the client along with the acquired locks even after the transaction that requested them is over. This is called inter-transaction caching and it allows future requests on the cached data to be satisfied locally. Available disk and memory buffers at the clients are utilized in order to perform such caching. The use of data-shipping in conjunction with inter-transaction caching offers the following advantages: (i) off-loading of database processing from the server to the clients, (ii) shorter transaction response times as the data is stored closer to its point of use, and (iii) reduction of network usage and contention when transactions demonstrate spatial and/or temporal locality of database accesses.

The server coordinates concurrency-control in the NOW [5, 28] with the help of a global lock table. There are two kinds of locks, *shared* (read) and *exclusive* (read-write), and the locking scheme works as follows:

- If a client has requested a shared lock on an object, the server grants the lock only if no other client has an exclusive Lock on that object. Similarly, if a client requests an exclusive lock on an object, then this lock is granted only if no other client has any type of lock on that object. When a lock is granted to a client, a copy of the object is shipped over to the requesting client, if necessary.
- If an object is out of reach due to a conflicting request, then a callback mechanism is in effect. The goal of the callback mechanism is to recall the object in discussion as soon as possible. The clients currently holding the lock(s) are notified by the server to return both lock(s) and object(s) as soon as their operation is complete. Only when the object(s) has been returned does the server grant the lock to an outstanding request(s) and send the object to the appropriate client(s).

This locking scheme is a variant of strict *two phase locking* (2PL) [27, 28] for distributed environments. As all objects that are accessed by a regular client transaction have to be appropriately locked by the client, it is guaranteed that regular updates to the database are serializable. The server is also responsible for detecting and resolving deadlocks. This is achieved by maintaining a wait-for graph whose nodes represent clients and objects, and edges correspond to granted and outstanding requests. A resolution heuristic that could be used here is that a client is preempted only if it participates in a cycle, has acquired a lock and is currently blocked.

Since objects are cached in an inter-transaction manner, locks on cached objects are released only when the server requests the client to do so (callback). For example, a read-only (SL) cached lock is only released when the server indicates that it has been requested an exclusive lock on that object by another client. If a client needs to create space in its short and long term memory, objects can be returned to the server.

In the following section, we discuss push-transactions and their supporting mechanisms. Such push-transactions work simultaneously with conventional transaction processing.

3. Automatic update propagation

This section describes the push protocols in detail and also discusses the policies used to schedule time constrained push-transactions.

3.1. Terminology

In this subsection, we describe the two basic elements of our push protocols, namely, push-predicates and push-transactions.

Client push-predicates: Clients specify the data sets of their interest by providing logical predicates. These are called *push-predicates*. The *complete* set of client push-predicates (also termed base of predicates) is stored at the server at all times. A client's push-predicates denote not only the data that is to be automatically updated, but also specify the conditions under which updates should be propagated. In addition, a deadline is associated with every push-predicate which indicates a preferred time interval within which an update should be visible at the client.

The conditions specified in push-predicates are made up of conjunctions of boolean atoms that designate thresholds for database object values. For example, an user at a client computer (Client ID 34) may be interested in tracking the share price of XYZ Inc. To receive automatic updates of this information, the user can specify a push-predicate to the server of the database system. A template for such a push-predicate is given in figure 3. It specifies that the stock price of XYZ Inc. is to be pushed to Client 34 when the price drops below $8.50. Client 34 should receive this update within five minutes of the condition being satisfied.

Push predicates are only evaluated after the completion of a regular update, and not after a push-transaction finishes. Therefore, it is not possible for a push-transaction to trigger new ones.

Push-transactions: When a regular update to a database object causes violation of client push-predicate criteria, a push-transaction is shipped to each of the interested sites. A push-transaction specifies the operations to be performed on client cached objects and consists of the script of the original update. But, it is worth noting that push-transactions are different from regular updates. Regular updates have to work within the constraints of

```
PUSHPREDICATE example;
CLIENT ID: 34;
CONDITION: (( GetObject( "XYZ Inc." ).StockPrice < 8.5 );
ACTION: Push Transaction to update Stock Price;
DEADLINE: 5 mins;
```

Figure 3. A push predicate template.

the distributed two-phase locking protocol [5, 28]. In contrast, push-transactions need not acquire global exclusive locks on the cached data that they modify. Instead, the correct execution of a push-transaction is ensured by verifying that the objects accessed by it are in the same state as prior to the execution of the corresponding regular update. The version number of each object is used as an indicator of its state.

Once a push-transaction (i.e., the script) is received, the client processes it in order to bring its own copy of the involved object(s) to the current state. With this form of localized transaction processing, the amount of data that needs to be transported over the network can be reduced considerably. Push-transactions are scheduled with a higher priority than regular ones as they have to ideally complete within given time constraints. The manner in which transactions (regular and push) are handled by client transaction schedulers is described later in this section.

3.2. Types of push-transactions

There are several forms that push-transactions can take depending on the type of database object and the updating action to be performed. The two basic types are absolute-update and relative-update. An absolute-update can be pushed to clients at any time without risking global data consistency. An example of this is to set the stock price of XYZ Inc. to $56 per share at all sites that are interested in this information. In contrast, a relative-update can only be shipped if the client's copy of the object is data-consistent with the latest-copy, just before the update in question. An example relative-update would be to increase the stock price of XYZ Inc. by $0.53.

Since clients' push-criteria for the same data object can be different, it is possible that several versions of the data object are cached at different clients. Therefore, in order to manage relative-updates, it is necessary to maintain version information about the objects cached at the clients. In addition, a transaction precedence list (TPL) and the corresponding transaction scripts also need to be stored. Using this information, it can be determined which push-transactions (scripts) affect the copies of an object cached at different clients. Finally, the scripts have to be shipped to these clients so as to produce consistent database object values. The server undertakes the management of the TPLs.

Before a relative-update is pushed to a client, the objects' versions at that client are checked and multiple push-transactions are shipped together, if necessary. However, when many versions of an object are proliferated throughout the system, the overhead involved in maintaining TPLs for each object can become very large. In order to keep these overheads at a manageable level, we propose that the system maintain only a fixed number of transactions in the TPL for each object. When a client's version of the object falls farther behind than is possible to update using TPL push-transactions, the latest copy of the object (actual data) is shipped to that client. Depending on which site handles this processing, two different push strategies are proposed later in this section.

Consider the push-predicates shown in figures 4 and 5. These predicates can be part of a supply chain management system. The first predicate indicates that $Client_1$ is interested in being informed within five minutes when the inventory level of *Widgets* falls below 500 units. On the other hand, $Client_2$ needs to know whenever more than 200

```
PUSHPREDICATE One;
CLIENT ID: 1;
CONDITION: ( GetObject( "Widgets" ).Stock < 500 );
ACTION: Push Transaction to update Stock Quantity;
DEADLINE: 5 mins;
```

Figure 4. Push predicate for client 1.

```
PUSHPREDICATE Two;
CLIENT ID: 2;
CONDITION: ( GetObject( "Widgets" ).LastAllocationSize > 200 );
ACTION: Push Transaction to update Stock Quantity;
DEADLINE: 3 mins;
```

Figure 5. Push predicate for client 2.

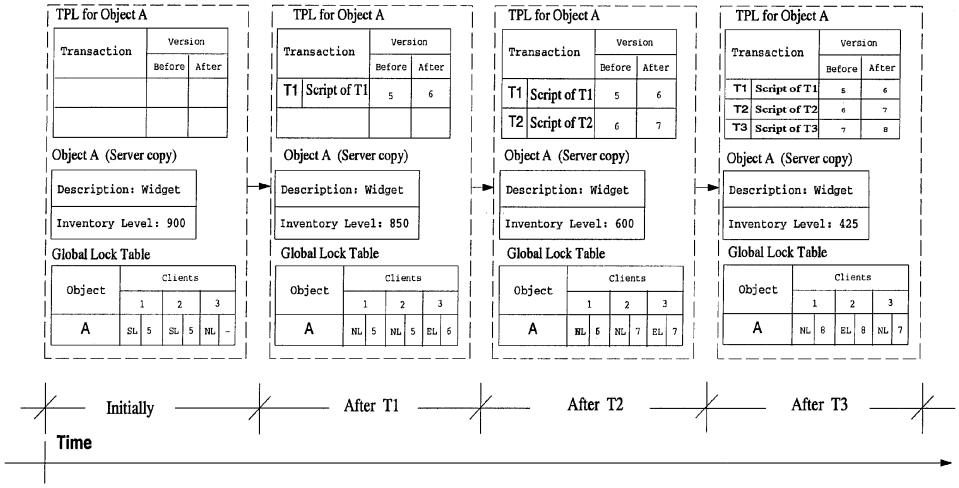

Figure 6. Supporting data structures for push-transaction handling.

Widgets are allotted to an assembly line within three minutes after the change occurs (figure 5).

Let the original quantity of *Widgets* in the inventory be 900 units. $Object_A$ contains both the quantity of *Widgets* as well as any additional required information. This object is cached—in a shared mode—at $Client_1$ and $Client_2$. This state of affairs is depicted in figures 6 and 7 as the initial state (shown as "*Initially*"). At this time, $Client_3$ does not maintain a cached copy of $Object_A$. This information is accurately reflected in the server's global lock table which also maintains the version numbers of $Object_A$ cached at each client.

Consider the following transactions:

T_1: Allocate 50 *Widgets* from the inventory to Assembly Line 1.
T_2: Allocate 250 *Widgets* from the inventory to Assembly Line 7.
T_3: Allocate 175 *Widgets* from the inventory to Assembly Line 2.

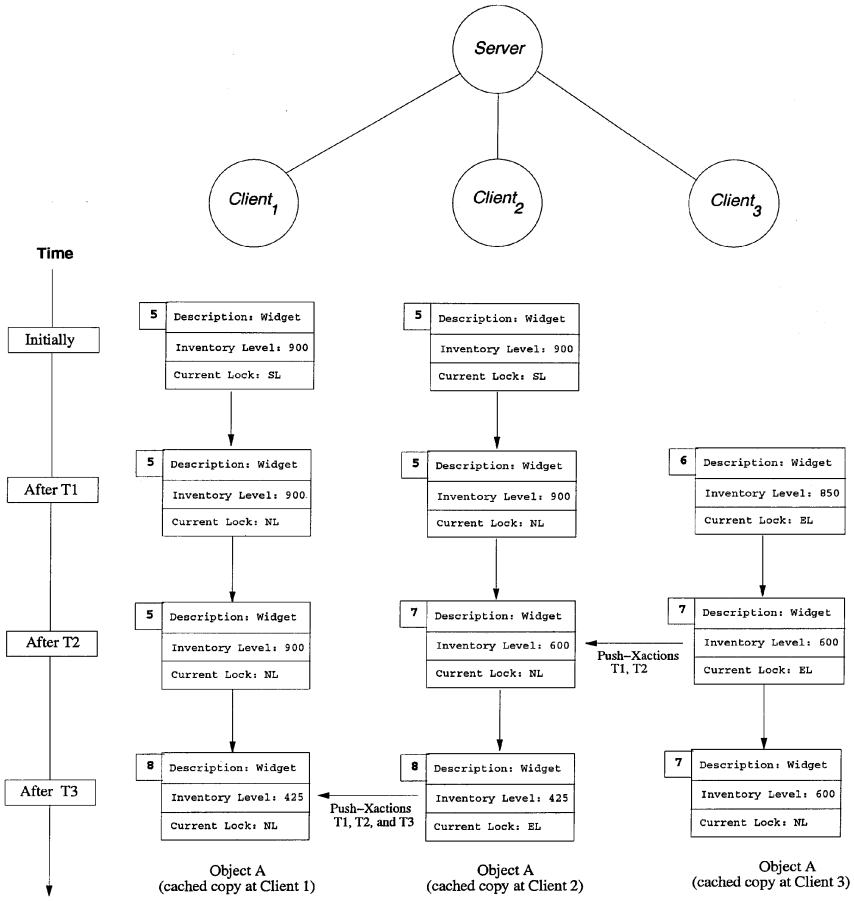

Figure 7. Evolution of values of a client cached object.

Below, we present a possible scenario for the execution of these transactions where T_1 and T_2 are initiated at $Client_3$, and T_3 is executed at $Client_2$.

1. T_1 *at* $Client_3$: Since T_1 is a regular update, it is imperative that $Client_3$ obtains an exclusive lock on $Object_A$. Therefore, $Client_3$ requests an exclusive lock on $Object_A$ from the server. Now the server issues callbacks to $Client_1$ and $Client_2$ requesting them both to release their shared locks on the object. Once both clients have released their locks, the server grants an exclusive lock to $Client_3$ and ships the object over to it.

   Note that although $Client_1$ and $Client_2$ have released their locks on the object, they are not required to dispense with the cached copies of the object. At either sites, the "outdated" cached object is updated by a regular transaction that uses the object or when the clients' push-criteria for that object are met and a push-transaction is propagated to it.

   Once T_1 completes, the inventory level of *Widgets* is 850. At this point, push-predicates for $Client_1$ and $Client_2$ are evaluated. Since neither push-criteria are satisfied, there is

no update to be propagated. T_1 is added to the TPL for $Object_A$. Note that the TPL also stores the "before" and "after" version numbers of the object.

This new state is depicted in figures 6 and 7 as state "*After T1*."

2. T_2 *at* $Client_3$: Since $Client_3$ already has an exclusive lock on the object, it can run T_2 immediately. This modifies the value of available *Widgets* to 600. The copy of the object at the server is also updated. Transaction T_2 is added to the TPL for $Object_A$.

   After T_2 completes, the second push-criterion (figure 5) is met and a push-transaction is shipped to $Client_2$. Since the copy of $Object_A$ at $Client_2$ is two versions behind, the push-transaction consists of the scripts of the two updates that have taken place at $Client_3$, namely T_1 and T_2. This is done with the assistance of the server's TPL structure. When this push-transaction reaches $Client_2$, it modifies the copy of $Object_A$ without having to acquire an exclusive lock. The total effect of these two updates is shown in figures 6 and 7 as the state labeled "*After T2*."

3. T_3 *at* $Client_2$: As $Client_2$ has to acquire an exclusive lock on $Object_A$, it dispatches a request to the server. The server recalls the object from $Client_3$, and then ships the latest copy of the object (actual data) to $Client_2$.

   Once $Client_2$ has received the latest version of the object it allows T_3 to execute. The execution of T_3 triggers the push criterion that is pertinent to $Client_1$ (i.e., figure 4). Now a push-transaction has to be shipped to $Client_1$ to update its inventory of *Widgets* to 425 units. It is evident that $Client_1$ is "left" behind three versions/regular transactions so this push-transaction is made up of the scripts of all three updates T_1, T_2 and T_3. This state is depicted in figures 6 and 7 as state "*After T3*." At $Client_3$, the value of the *Inventory Level* remains at 600 units as $Client_3$ has no pertinent push-predicate.

In general, push-transactions may be handled in two distinct ways. They can be forwarded to interested clients with the assistance of of the server which maintains the complete push-predicate set. Alternatively, if client sites maintain their own push-predicate subsets, then push-transactions can bypass the server and propagate the necessary scripts directly to affected clients. These two schemes are termed Server–Push and Client–Push and we elaborate on them later in the section.

3.3. Scheduling push-transactions

A soft-type deadline is associated with each push-transaction and indicates the time within which the update must reach a particular client. A push-transaction is said to have successfully completed only if the update has successfully reached the requesting client within its specified time constraint. In time constrained systems, the priority assignment scheme used to schedule tasks often plays an important part in deciding the efficiency of the system. This efficiency is measured in terms of the percentage of transactions completed within their deadlines. In this paper, a push-transaction that misses its deadline is still executed with a lower priority. This is done so that concerned NOW sites ultimately "see" modifications made by others in a graceful manner.

Several methods have been used to assign priorities to task with time constraints [1, 9, 23]. The most elementary ones are:

- *First-Come First-Serve* (FCFS): This policy schedules tasks to be processed in the order in which they arrive. Since the deadline is not used, FCFS will schedule a newly arrived task with an earlier deadline after an older task that has a later deadline. Therefore, FCFS serves only as a baseline for comparison with other scheduling disciplines.
- *Earliest Deadline* (ED): The task with the earliest deadline is assigned the highest priority. As this policy does not take the expected processing time for tasks into consideration, it has the weakness that it can schedule tasks that have missed their deadline or are certain to.
- *Least Slack First* (LS): For each task, a slack time $S = d - (t + E)$, is defined. Here, t is the current time, and E and d are the estimated processing time and deadline for the task respectively. The slack time, S, is an estimate of how long a task can be delayed without missing its deadline. The task with the least available slack time is scheduled first.

An important assumption that we make when using the LS scheduling discipline is that the expected processing time of all tasks is known. The availability of the expected processing time for a task also allows us to determine the feasibility of the task meeting its deadline. In this manner, tasks that are expected to miss their deadlines are designated as *Tardy Transactions*, and are executed at a lower (default) priority. In ED scheduling, we adjust the deadline of a push-transaction by an estimate of the time required to ship the push-transaction to its execution point. Similarly, in LS scheduling, we include this transmission-time estimate as part of the expected processing time E when calculating the slack for the transaction. This is an adaptation of the virtual deadline assignment technique proposed in [22].

In the context of push-transaction scheduling, we take advantage of additional available information to enhance the quality of the schedules generated by the discussed priority-assignment algorithms. This information can be used at run-time and consists of:

- *Client Count* (CC): This denotes the number of clients that are interested in receiving a particular update. The use of this information allows the scheduler to break ties among push-transactions by choosing the transaction which affects a greater number of clients first.
- *De-scheduling of Redundant Push-Transaction Shipments* (DRT): Using available information about the push-transactions' operation and data object accesses, we attempt to de-schedule push-transactions whose effect has been affected by later push-transactions. This technique is useful for eliminating absolute and relative-updates that are succeeded by an absolute-update on the same data. For example, consider a database object O_9, and transactions T_1 and T_2 that update O_9 one after the other. Now, if the push-transactions corresponding to T_1 and T_2 are scheduled to be shipped to a particular client then, ideally, it is necessary to ship only the push-transaction for T_2 to that client. If T_1 has not yet been pushed to the client then it can be dropped from the push schedule.

  Such de-scheduling can be done at the site that is performing the update propagation. Once push-transactions have been received by interested NOW sites, it is again necessary to detect and eliminate redundant push-transactions. It is also important to ensure that the execution of push-transactions does not cause corruption of data at the client sites. This is done by the transaction schedulers at the clients.

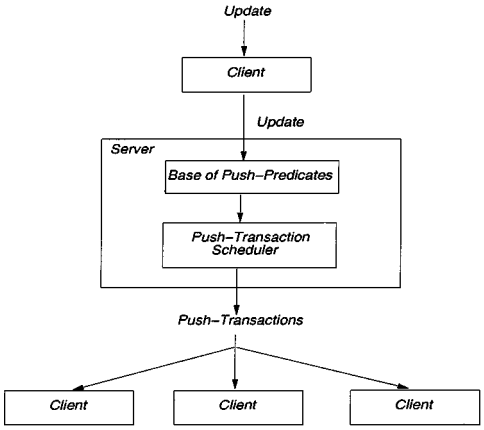

Figure 8. Server–push.

3.4. Description of the push protocols

3.4.1. Server–push. In this mechanism, all regular updates at the clients are immediately sent to the server. The complete set of client push-predicates is stored only at the server. Once a client update reaches the server, the base of push-predicates is evaluated to determine whether this change to the database state has triggered any push-transactions. If a regular update satisfies client-specified criteria then the server is required to push the update to such clients in the form of push-transactions. These push-transactions are transmitted to the requesting clients where they are executed to bring the clients' copies of the database objects up-to-date. The Server–Push mechanism is depicted in figure 8. At any given time, the server may have several push-transactions that are to be shipped to various sets of clients. These push-transactions are scheduled according to policies that try to minimize the number of missed deadlines in a soft time constrained fashion. The latter means that even if a push-transaction misses its deadline, it will ultimately be scheduled for execution using the default priority.

The outline of the Server–Push protocol is as follows:

0. Regular Update Transaction T commits at Client C.
1. Copies of the objects that have been updated by transaction T are shipped to the server, along with the transaction script for T.
2. For each object that T has updated

   2.1 The server evaluates the clients' push-predicates associated with that object.

   2.2 IF (the push-criteria for Client D are satisfied) THEN the server looks up in its version table as to check which version of the object is currently cached at Client D.

      2.2.1 IF (the object at Client D is one version behind the latest) THEN update transaction T is to be shipped to Client D.

2.2.2 ELSE_IF (the object at Client D is more than one version behind the latest but it can be updated using transaction scripts stored in the TPL) THEN such transactions from the TPL and transaction T are scheduled to be shipped from the Server to Client D.
2.2.3 ELSE_IF ((the object is not cached at Client D at all) or (the object is so many versions behind that it cannot be updated using transactions stored in the TPL)) THEN the latest copy of the object is transferred from the Server to Client D.
ENDIF

ENDIF

3. All the scheduled push-transactions are shipped to the appropriate clients. Transaction T is added to the TPL corresponding to that object.
4. Once these push-transactions arrive at their respective clients, they are handled by the clients' transaction scheduler and scheduled for execution according to the scheduling policy in use.

As an example, consider a regular transaction at $Client_{23}$ that reads objects O_3 and O_{11}, and updates O_{91}. In order to be able to execute this transaction, $Client_{23}$ has to first acquire an exclusive lock on O_{91}, and shared locks on objects O_3 and O_{11}. Once these locks are secured, the transaction can go ahead and modify O_{91}. After the transaction commits, its script is immediately shipped to the server along with the new version of object O_{91}. The server examines push-predicates associated with O_{91} to examine whether any push-criteria for any clients are met. If so, appropriate transaction scripts are shipped to those clients. At the clients, the push-transactions received from the server are scheduled at a higher priority than local regular transactions. The deadline associated with each push-transaction is used to assign it a priority among the other scheduled push-transactions.

As the evaluation of the base of predicates is done in a centralized manner (at the server only), only the server database system needs to be adapted significantly to perform such updates in a traditional client-server environment. The client database system needs minor modifications to accommodate the receipt and scheduling of push-transactions. The overheads introduced by this protocol on the processing of regular database transactions are: (i) the evaluation of the push predicates following each regular update and the shipment of the push-transactions, (ii) the immediate shipment of updated database objects from the clients to the server (this is unlike conventional data-shipping CSDs where updated objects are returned to the server only when they are explicitly called back), and (iii) execution of push-transactions at the clients. This protocol for pushing updated data to clients is expected to be beneficial when the server is better placed to deliver the updates than the originating client. This is the case when the network topology resembles a star or when the server has faster access to the network than the clients.

3.4.2. Client–push. In this technique, the client pushes modifications to concerned clients directly—as simple point-to-point transfers. When a client is granted an exclusive lock on an object, the set of push-predicates associated with that object are shipped over to the

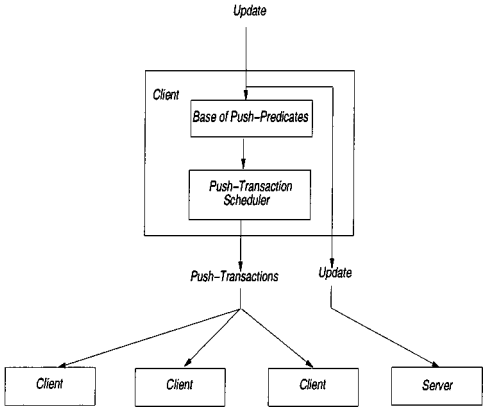

Figure 9. Client–push.

client. In addition, the client also maintains the TPL for that object. The complete set of push-predicates cached at a client forms its local base of predicates. After a regular update modifies an object, the local base of predicates is evaluated to determine which clients require the update to be communicated to them. The update is shipped to the interested set of clients directly. These pushes are assigned priorities according to the used scheduling algorithm and are executed in that order. The server is treated on a par with the other clients and an update-push is scheduled to the server as well (figure 9).

The algorithm for Client–Push is:

0. Regular Update Transaction T commits at Client C.
1. The push-predicates, object version lists and TPLs for the objects exclusively locked by Client C are available locally. Therefore, for each object that T has updated:

   1.1 Client C evaluates the push-predicates associated with that object.
   1.2 IF (the push-criteria for Client D are satisfied) THEN the client looks up in its version table as to which version of the object is currently cached at Client D.

      1.2.1 IF (the object at Client D is one version behind the latest) THEN update transaction T is scheduled to be pushed to Client D.
      1.2.2 ELSE_IF (the object at Client D is more than one version behind the latest but it can be updated using transaction scripts stored in the TPL) THEN such transactions from the TPL and transaction T are scheduled to be shipped from this Client to Client D.
      1.2.3 ELSE_IF ((the object is not cached at Client D at all) or (the object is so many versions behind that it cannot be updated using transaction scripts stored in the TPL)) THEN the latest copy of the object is transferred from this Client to Client D.
      ENDIF

   ENDIF

3. All scheduled push-transactions are shipped to the appropriate clients and to the server. Transaction T is added to the TPL corresponding to that object.
4. Once these push-transactions arrive at their respective clients, they are handled by the clients' transaction scheduler and scheduled for execution according to the scheduling policy in use. The server also executes these updates on its copies of the data objects.

Consider, once again, the example of a transaction at $Client_{23}$ that reads objects O_3 and O_{11}, and updates O_{91}. Since this is a regular transaction, the client acquires shared locks on O_3 and O_{11}, and an exclusive lock on O_{91}. The transaction can proceed only after all these locks have been acquired. Once the transaction commits, the push-predicates associated with O_{91} in the client's predicate-base are evaluated. If there are any clients that are interested in receiving this update, then push-transactions are pushed to those clients directly. The update is also shipped to the server to modify the copy of O_{91} stored there. Even after the transaction has committed, $Client_{23}$ retains the exclusive lock on O_{91}, the shared locks on O_3 and O_{11}, and also keeps the push predicates associated with O_{91} in its own base of predicates. Now, if a future transaction requests an exclusive lock on the object, the request can be satisfied locally.

The advantage offered by this technique is that the push load is off-loaded from the server giving it more processing power to serve non-push data object requests. Contrary to the Server–Push scheme, adapting a traditional CSD to provide such update management techniques, requires considerable modification to the client database system. Such modification includes incorporation of a push-transaction scheduler and an evaluation mechanism for the base of push-predicates. Here, the server database calls for few modifications. The overheads introduced by the Client–Push protocol are: (i) shipping the push-predicates and clients' object version information associated with each exclusively locked object, (ii) evaluating the push-predicates after each update and shipping the push-transactions to interested clients, and (iii) re-executing the updates on the clients' copies of the updated data.

3.5. Transaction handling at the clients

The transaction scheduler at each client is responsible for ensuring that the execution of regular transactions and push-transactions is performed correctly. This is an important task because an incorrect order of execution or an improper interleaving of regular updates and push-transactions can corrupt the contents of client-cached data objects. When two transactions access disjoint sets of objects, their execution can be performed either in serial or in parallel without any special considerations. Otherwise, a client's transaction scheduler is required to handle three possible situations that can arise due to conflicting object accesses, namely: (i) when two regular transactions access the same data in a conflicting manner, (ii) when a push-transaction updates the data that is being used by a regular transaction, and (iii) when the data access sets of two push-transactions conflict. The way in which each of these cases is handled is described below.

(i) *Two regular transactions access some database objects in a conflicting manner*: Regular transactions are required to lock the data that they access in the proper mode (shared or exclusive). In such cases, the order in which two transactions get to access a database object is determined by their success in acquiring the appropriate locks with the help of both the local and the server's lock managers.

Transactions cannot be granted conflicting locks on the same object. Conflicting locks requests are granted by the lock manager serially. This guarantees that all database accesses by regular transactions are serialized. Local lock managers also perform the necessary deadlock detection to make sure that locally cached objects are not locked indefinitely.

(ii) *Object accessed by a push-transaction conflict with those of a regular update*: The manner in which the execution of *write* conflicting push-transactions and regular updates is carried out depends on the order in which the two arrive at the local scheduler. If the push-transaction arrives first then the regular update transaction is blocked. Once the push-transaction has completed, the regular transaction can acquire its necessary locks from the local lock manager, if these objects/locks have been cached by the client. Otherwise, the objects/locks are requested from the server. Data objects whose versions at the client are older than those at the server will be refreshed when the latter grants the requested locks.

If the push-transaction is initiated after the regular transaction then the latest copies of the objects accessed by the regular update will have been fetched from the server (if they are not already present at the client). Therefore, it is necessary to de-schedule the push-transaction in order to prevent the contents of these objects from getting corrupted. De-scheduling the push-transaction means that some objects that it would have updated may remain out of date. Such objects can be identified by their version numbers. The most recent copies of these objects can be requested from the server.

(iii) *Two push-transactions access intersecting sets of database objects*: Sometimes the client's transaction scheduler is required to decide the order of execution of two (or more) push-transactions whose write-sets intersect. In such cases, the scheduler can decide which push-transaction should be executed first by examining the version numbers of the objects accessed by each transaction. Using these version numbers, the transaction scheduler can derive a *happened-before* relation among the push-transactions and execute them in that order.

4. Experimental evaluation of the models

To investigate the feasibility of our push-transaction strategies, we have constructed a simulation model of a data-shipping client-server database system using the CSIM simulation library [17]. We first evaluated the two push protocols in this setting. Then, in order to further establish our results, we developed distributed prototypes that allow us to examine the push mechanisms in a network of workstations (NOW). In this section, we describe the setups for the simulations and the prototypes, and discuss the results of our experimentation. The key objectives of our combined experimental effort were to investigate the following:

(i) the scalability of the two push strategies as the number of clients attached to the server, and hence the regular and push-transaction load, increases, and
(ii) the effect of the various scheduling policies on the performance of the two push strategies.

4.1. Baseline experimental parameters

In this subsection, we describe the parameters that are common to the simulations as well as the prototype experimentation. The database that we have used consisted of 10,000 objects. The database was resident at the server and the cache of each client was able to accommodate 10% of its size. The database was divided into 10 disjoint partitions and the frequency of accesses to each partition was determined according to a Normal or a Zipf distribution [18]. The actual probability distributions are shown in figures 10 and 11. There was no restriction on client accesses to any portion of the database. Therefore, for a system with N clients, there could be N-way contention for each database object. The network topology used is that of an Ethernet LAN and is depicted in figure 2.

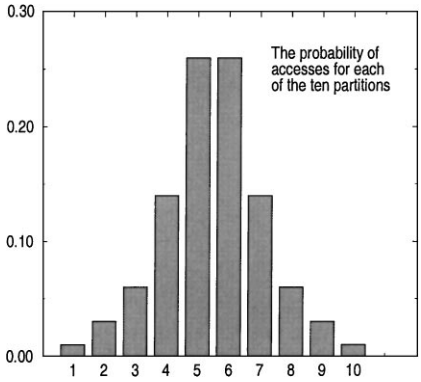

Figure 10. The database access pattern for each client according to a normal distribution.

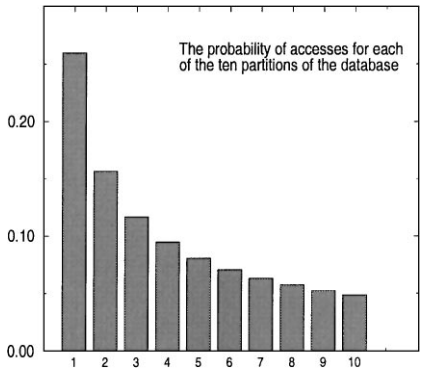

Figure 11. The database access pattern for each client according to a Zipf distribution ($\theta = 0.271$, see Appendix).

In the CSD, regular transaction arrivals at each client have been modeled as a Poisson processes. The load on the system is varied by changing the number of clients attached to the server and the percentage of regular update transactions. The CPU processing time for each transaction was modeled according to an exponential distribution. Each locally generated client transaction is either an update or a query. The number of objects requested by each transaction is exponentially distributed around a fixed mean (i.e., Table 1). Database object requests from client transactions are either shared or exclusive lock requests which are generated randomly according to the update percentage for the experiment. The server processes all requests for data objects/locks from the clients, and is also responsible for maintaining an up-to-date lock table and resolving serialization issues. Objects that have been fetched from the server are cached at the clients in an inter-transaction manner. The buffer and disk space available at the clients are used to maintain local data-space.

We work with three sets of experiments as Table 1 indicates. The intention of the first experiment is to investigate the behavior of the proposed strategies in an environment that receives sizable updates (i.e., 10% of all transactions are modifications). The second experiment examines the case where the database receives an extreme number of modifications (i.e., 20%). Finally, the third experiment uses the Zipf distribution for accessing objects. In this, a limited range of objects receives most of the traffic/work. The specific values of the parameters for each set of experiments are given in Table 1.

In all our experiments, each client was assumed to be interested in updates that occurred to approximately 5% of the objects in the database. This set of objects was selected randomly (Uniform distribution) at the beginning of each experiment. Since the deadline assignment plays an important role in the evaluation of a time constrained system, we describe our

Table 1. Parameters for each set of experiments (simulations and prototyping).

	Experiment set 1		Experiment set 2		Experiment set 3	
Parameter Push protocol	Server push	Client push	Server Push	Client push	Server push	Client push
Database size (objects)	10,000	10,000	10,000	10,000	10,000	10,000
Server main memory size (objects)	2,500	2,500	2,500	2,500	2,500	2,500
Client memory cache size (objects)	100	100	100	100	100	100
Client disk cache size (objects)	900	900	900	900	900	900
Scheduling strategies	ED, LS, FCFS	ED, LS, FCFS	ED, LS, FCFS	ED, LS, FCFS	ED, LS, FCFS	ED, LS, FCFS
Access distribution	Normal	Normal	Normal	Normal	Zipf	Zipf
Percentage of regular updates	10%	10%	20%	20%	10%	10%
Average number of objects accessed by each transaction	30	30	30	30	30	30

deadline assignment model in detail. In a data-driven update protocol, an update to a data value has varying significance to different clients. With this in mind, we define a set of (client, probability, deadline) triplets for each database object. After an update to an object, the push scheduler evaluates the set of triplets for that object. For each triplet (C_i, P_i, D_i), a push-transaction is shipped to Client C_i with probability P_i. This push-transaction has to reach within D_i time units after the update transaction commits. Thus, D_i is the time interval for which Client C_i recognizes the value of receiving the update. The probabilities in each triplet are used to emulate the triggering of push-transactions following the probable satisfaction of client criteria. In the prototype experiments, for both Server–Push and Client–Push, the cost of evaluating a push-predicate was estimated as an exponentially distributed delay with an average of 25 milliseconds. This time delay tries to incorporate the cost of fetching the push-predicate and the corresponding object (from the disk or main memory), and the cost of evaluating the contents of the object with respect to the predicate criterion.

For each triplet (C_i, P_i, D_i), the push probability (P_i) was generated uniformly distributed in the range [0, 1), while the deadline (D_i) was generated according to an exponential distribution which had the average transaction length as its average. Hence, for most triplets, the deadlines are moderately tight (figure 12). As an example, let (12, 0.7, 4) be the triplet for $Object_{91}$ corresponding to $Client_{12}$. Since the push probability specified in this triplet is 0.7, this mean that 70% of the updates to $Object_{91}$ should be communicated to $Client_{12}$. The deadline for the propagation of each such update is 4 seconds.

The results that we present are the:

(i) percentage of update-pushes that reached the clients within their specified deadlines in the two systems,
(ii) effect of the three scheduling strategies on the efficiency of the two push strategies, and
(iii) average response times for non-push object requests from clients.

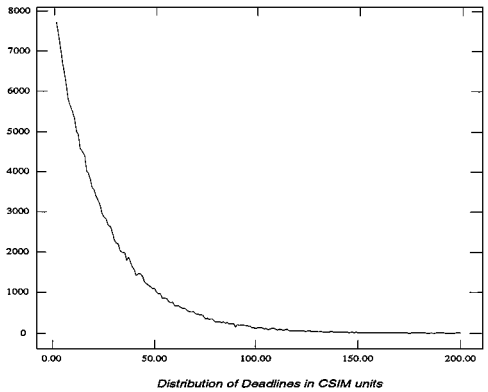

Figure 12. The distribution of deadlines with the number of push deadlines of each value on the Y-axis (for one run of each set).

4.2. Simulation experiments

4.2.1. Simulation-specific parameters. The processing time for every update transaction was modeled as an exponential distribution with a mean of 25 CSIM time units. The deadlines for push-transactions were also exponentially distributed with an average of 25 time units. Transactions were scheduled according to the ED, LS and FCFS scheduling disciplines. In addition to these scheduling strategies, we used the CC (Client Count) and DRT (De-scheduling of Redundant Transactions) criteria to enhance the quality of the schedules. It is important to note that for reasons of simplicity, our simulations have considered absolute-update transactions only. This is not the case in our prototype experimentation where relative-updates are allowed as well.

The clients and the server were simulated by CSIM processes. Each CSIM process has its own private data space. This allows us to simulate the operation of a NOW on a single workstation. Each simulation was run for 150,000 CSIM time units (which took about 9 hours for 100 clients). The network was also emulated by means of a CSIM process and all data transfers between the clients and the server were routed through it. Network delays in the transmission of data or requests were modeled as those in an Ethernet LAN [9].

4.2.2. Simulation results. In the first set of experiments, we compare the efficiency of the three scheduling algorithms for the Server–Push and Client–Push strategies under moderate update loads (10% of all accesses to the database are updates). Accesses to the database are according to a Normal distribution (figure 10). The plots of the percentage of transactions pushed and completed within their deadlines using the Server–Push strategy are shown in figure 13. It can be seen that for a small number of clients, the efficiency of no single scheduling strategy is consistently better than the other strategies. However, as the number of clients increases, Earliest Deadline scheduling performs marginally better than the FCFS or LS scheduling policies. The performance of the FCFS strategy is seen to be the worst with just over 50% of transactions successfully completed in the presence of 100 clients in the system. The efficiency of the Client–Push method for each of the scheduling algorithms can also be seen in figure 13. Here, for a small number of clients, there is a perceptible difference in the performance of the different scheduling strategies. However, as the number of clients increases, the increased contention for the network and database objects causes the three scheduling strategies to demonstrate the same levels of efficiency.

Comparing the performances of the two push techniques (figure 13), we see that the performance of the Client–Push method is considerably better than Server–Push for a small number of clients. This is because the contention for the network is low, and therefore the single-stage push-transactions are faster than the two-stage pushes. As the number of clients becomes larger, the increase in the total number of push-transactions is slightly super-linear. In Server–Push, the scheduler at the server (with a greater number of transactions to schedule) is able to generate a better push schedule than the individual schedulers at the clients in Client–Push, and is also able to identify and de-schedule a greater number of push-transactions. In both push strategies, the CC criterion is used to enhance the schedules generated, but it provides greater benefit in the Server–Push strategy. Both optimizations cause the Server–Push strategy to perform better. This is because the server can examine

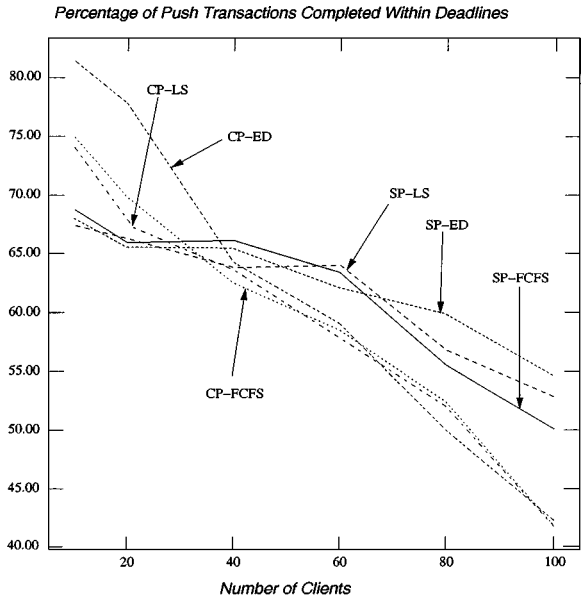

Figure 13. Percentage of push transactions completed within their deadlines (simulation experiment set 1, 10% updates, normal access distribution).

push-transactions from various sources and propagate the more beneficial ones first. The other important cause for the rapid deterioration in the performance of the Client–Push strategy is the increased contention for the network.

The second set of experiments compares the performance of the Server–Push and Client–Push strategies using the three scheduling strategies under heavy update loads (20% of the database accesses are updates with a Normal database access distribution). As the plots in figure 14 show, the higher percentage of updates causes the overall efficiency to be lower than what we see for 10% updates (Experiment Set 1). This is a direct result of the increased contention for exclusive locks on database objects as well as the resulting rise in the contention for the network. The performance of the three scheduling strategies is very close to each other for up to 80 clients. In the presence of 100 clients, we can see that the LS scheduling strategy performs the best, followed by the ED and FCFS scheduling strategies. The efficiency of the Client–Push method for heavy update loads is shown in figure 14. The results show a similar pattern as the Client–Push strategy in Experiment Set 1. The main difference in the two is the reduced efficiency caused by the increase in the percentage of updates. As seen earlier, the push-transaction scheduling algorithms do not significantly affect the performance of the Client–Push strategy. Comparing the Server–Push and Client–Push strategies, we see that although the Client–Push method performs better than the Server–Push method for a small number of clients, its performance degrades very rapidly as the number of clients in the system increases. Eventually, for a large number of clients, the Server–Push method exhibits better performance.

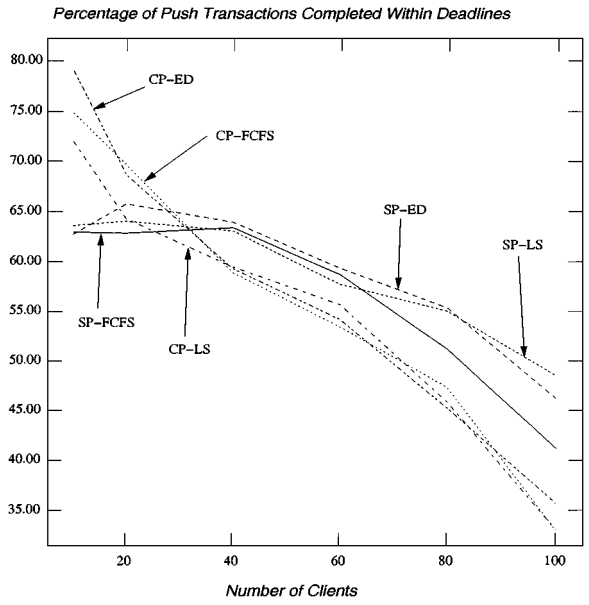

Figure 14. Percentage of push transactions completed within their deadlines (simulation experiment set 2, 20% updates, normal access distribution).

In the third set of experiments, we evaluated the efficiency of the two push strategies for a Zipf database access distribution. The percentage of updates was kept at 10%. The efficiency of the two methods are shown in figure 15. The performance of the two push strategies follows a similar pattern as that in the previous two sets of experiments. As the number of clients increases, the network contention in the Client–Push strategy becomes great enough to obviate the advantage over the two-phase operation of the Server–Push method.

In figure 16, we show the average response times achieved by the server in serving non-push object requests (Experiment Set 3). As the number of clients is small, the response times offered by the server when using the Client–Push method are close to those achieved using the Server–Push strategy. As the number of clients increases, the average object request time in the system is much higher when using the Client–Push technique. These higher response times mean that the throughput of the CSD using Client–Push is lower than that using Server–Push. The object response times seen in the CSD for Experiment Sets 1 and 2 are very similar to those shown in figure 16.

4.3. Prototype experiments

From the simulation results, it was evident that the Server–Push strategy was a more scalable alternative than the Client–Push method for the handling of push-transactions with time constraints. In order to provide conclusive evidence of this observation, we developed prototype client-server database systems using Solaris 2.5 socket and thread libraries. One

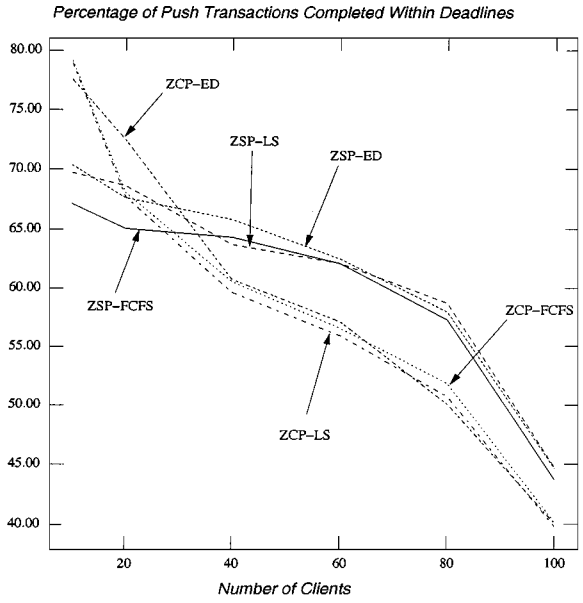

Figure 15. Percentage of push transactions completed within their deadlines (simulation experiment set 3, 10% updates, Zipf access distribution).

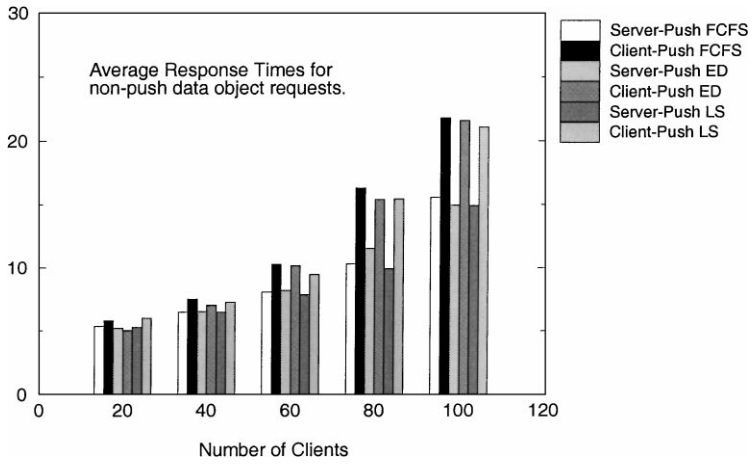

Figure 16. Average response times for non-push object requests for simulation experiment set 3 (averaged over all clients).

system used the Server–Push protocol while the other used the Client–Push method for pushing updates to clients. The test-bed for our prototype experiments was a system consisting of five Sun ULTRA-1 workstations which were connected by means of a 10 Mbps Ethernet LAN.

4.3.1. Prototype-specific parameters. The database consisted of 10,000 objects with the size of each object as 4,096 bytes. The management of an object database at the clients and the server was performed using the Paged File (PF) layer library [37]. A paged-file is a logical collection of equal-sized pages. The PF layer allows random access to any page from the paged-file and also provides a main-memory page buffer manager. In our experiments, we assume that one PF page contains exactly one database object.

Communication between the clients and the server was done using TCP sockets. The use of TCP ensured that the problems of loss of packets and out-of-order packet delivery were handled automatically. In both systems, the server has been designed to be connection-oriented, i.e., once a connection has been established between the server and the clients then that connection is maintained for the duration of the experiment. The overhead of establishing TCP connections is much higher than that of actual data transfer, and maintaining open socket connections is one way of avoiding it.

The server and the clients were implemented as multi-threaded programs. At the beginning of the experiment, the server establishes socket connections with each client and creates a unique thread to handle all future interactions with that client. Using one server-thread per client allows the greatest possible degree of concurrency in the handling of client data requests. The global lock table is used in conjunction with a wait-for graph to determine and resolve deadlocks in client requests [39]. Accesses to the lock table and other common data were serialized with the help of Solaris mutual exclusion primitives (mutex).

In a major improvement over our simulation experiments, our prototype systems maintain clients' object version information, transaction precedence lists (TPL), and the required transaction scripts so that object copies that are up to n versions behind may be brought up-to-date using stored push-transactions. This allows the system to manage relative-updates as well as absolute-updates. An object in a client's cache that is more than n versions behind the current copy is updated by just shipping the latest version of the object to the client. In our experiments, we chose the value of n as three but, in fact, any value can be used for n if the corresponding overhead of maintaining the precedence lists and transaction scripts can be tolerated. An important factor in the choice of n is the tightness of the push deadlines with respect to the execution times of push-transactions. When a client-cached object is many versions behind its latest version, it may be easier to meet the push deadline by shipping the latest copy of the object than by transferring and executing several push-transactions from the TPL.

Transactions arrive at each client with an inter-arrival mean time of 5 seconds (Poisson distribution). The average processing time for each transaction was 5 seconds, exponentially distributed. The deadlines for the push-transactions were exponentially distributed around the same average. A transaction (regular or push) is executed by creating a separate thread for it. The execution of transactions is performed as follows: if all the required data is available at the client then it is locked by the transaction and loaded into the client's memory buffers. Updated objects are marked as dirty so that they are eventually written back to the disk by the PF layer's buffer manager. Now, the transactions spends its prescribed CPU processing time calculating products of random numbers in a loop. Once this loop terminates, the transaction releases its lock on the data and commits. Here, too, we have used the ED, LS and FCFS scheduling strategies (in conjunction with CC and DRT) to prioritize the

push-transactions. For these experiments, we considered 30% of all push-transactions as relative-updates whereas the other 70% were absolute-updates.

4.3.2. Prototyping results. The percentage of push-transactions that completed within their deadlines for Experimental Set 1 (as described in Table 1) are shown in figure 17. It can be seen that the Client–Push strategy performs much better than the Server–Push method for a small number of clients. The low traffic contention on the network implies that the direct client-to-client push-transactions are much faster than the two-stage push-transactions through the server. However, once the number of clients increases the percentage of transactions that are successfully completed drops much more rapidly for Client–Push than for Server–Push. Shipping transactions to the server first, and allowing the latter to determine whether transactions should be propagated to other clients causes much less network contention than Client–Push. In addition, the server (Server–Push) is able to generate better push schedules than the individual clients (Client–Push). Since the server has a larger number of transactions to schedule, it can ensure that transactions that are to be shipped to a greater number of clients are given a higher priority. As seen earlier in the simulations, the algorithm used to schedule the push transactions does not have a significant effect on the percentage of deadlines that are met. The only significant observation that can be made in this regard is that in Server–Push, the FCFS scheduling strategy is clearly inferior to the ED and LS algorithms.

An important side-effect of the increased load on the server, and the higher network delays is an increase in the time that normal (non-push) transactions spend in waiting for

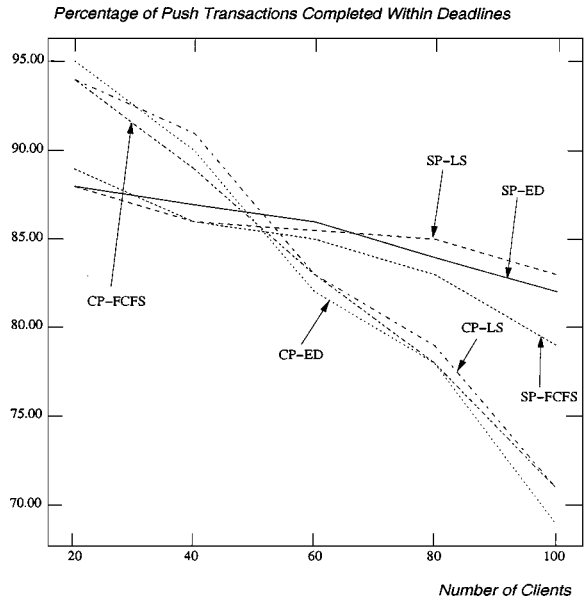

Figure 17. Percentage of push transactions completed within their deadlines (prototype experiment set 1, 10% updates, normal access distribution).

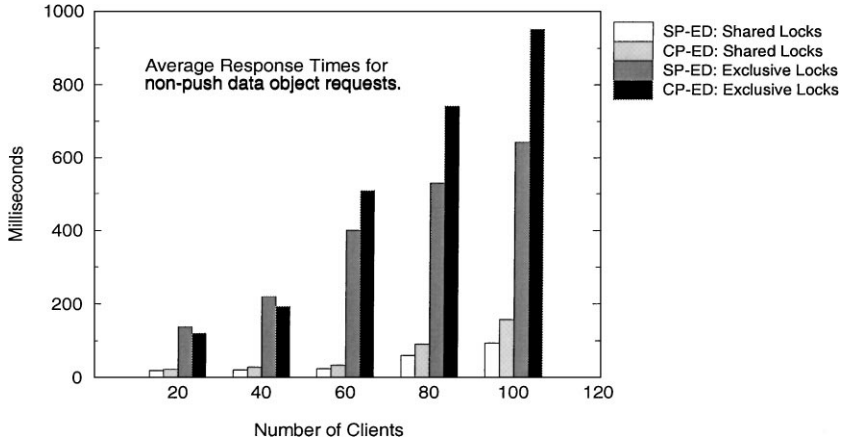

Figure 18. Average response times for non-push object requests for prototype experiment set 1 (averaged over all clients).

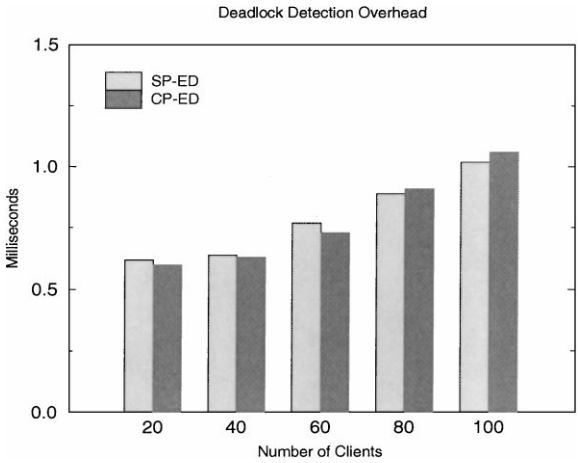

Figure 19. Average deadlock detection overhead for non-push object requests (prototype experiment set 1).

their requested data. The average times spent by client transactions while waiting for their requested data to arrive from the server are shown in figure 18. From this figure, we can see the average time required for an object request (shared or exclusive lock) to be satisfied by the server in both systems for increasing numbers of clients. The blocking times for client object requests increase drastically for the system using Client–Push. In contrast, in the system using Server–Push, the increase in these blocking times is much more gradual. The overhead caused by the deadlock detection algorithm is relatively low, even when there are one hundred clients in the system (figure 19).

The performance of the two push strategies for the second experimental set is shown in figure 20. Here, the percentage of update transactions is 20% of all transactions. The trends

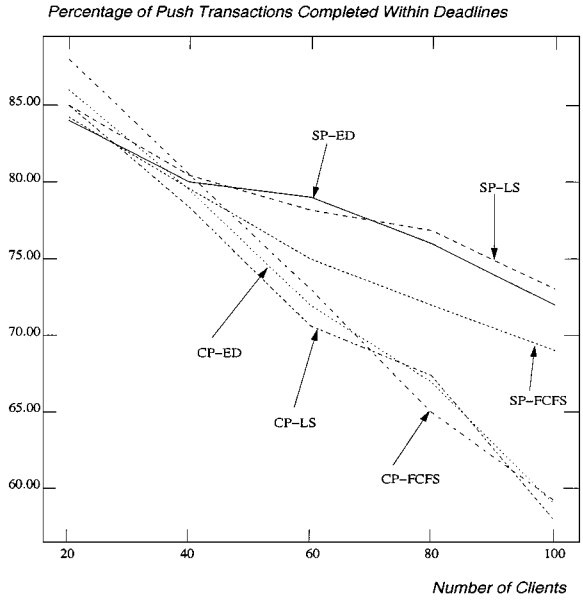

Figure 20. Percentage of push transactions completed within their deadlines (prototype experiment set 2, 20% updates, normal access distribution).

that can be seen here are very similar to those seen in figure 17. An important distinction is that the overall performance levels are much lower. This is expected since the increased rate of updates causes a corresponding increase in the number of push-transactions, and object transfers between clients and servers. Increased conflicting lock requests for data objects mean that clients are required to give up database objects, and re-fetch them later if necessary. As the curves show, the Client–Push method is slightly better for 20 clients, but once the number of clients increases to 40 and above, Server–Push offers a higher level of efficiency. The key factor that affects the performance of Client–Push is the increased network contention and latency. This is made worse by the TCP exponential back-off policy. Figure 21 shows the waiting time for clients' non-push object requests. The significantly greater network contention in Client–Push causes a corresponding increase in these waiting times.

In the final set of experiments, we used the Zipf access distribution to generate clients' accesses to the database. The percentage of update transactions is 10%. As the curves in figure 22 show, Server–Push offers a much more stable level of performance as compared to Client–Push. For 20 clients, the efficiency of Server–Push is 87% and when the number of clients is increased to 100, the efficiency drops to about 80%. In contrast, as the number of clients increases beyond 40, the efficiency of Client–Push deteriorates very rapidly and for 100 clients, only 69% of the push-transactions are able to meet their deadlines. The difference in transaction blocking times, even for shared locks, becomes considerably worse in Client–Push as the load increases.

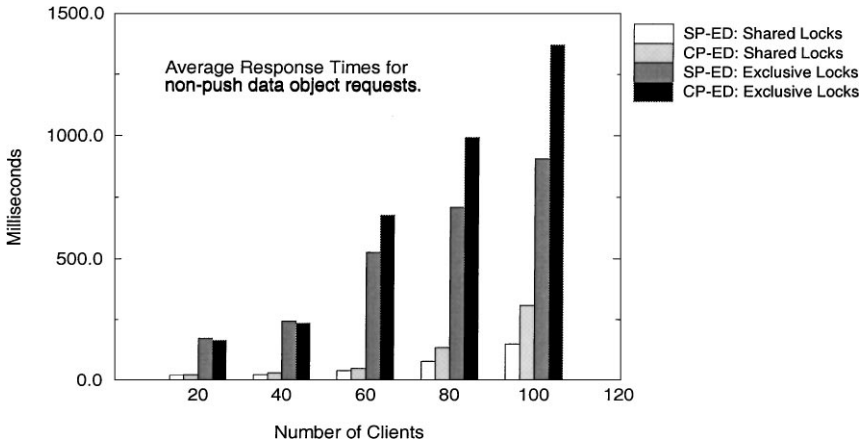

Figure 21. Average response times for non-push object requests for prototype experiment set 2 (averaged over all clients).

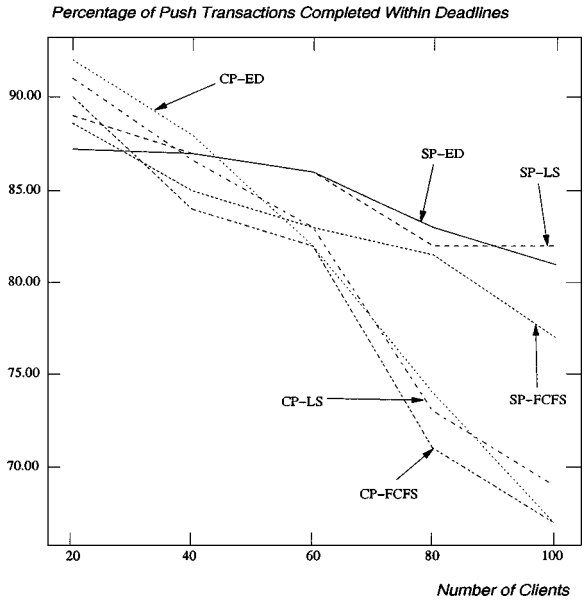

Figure 22. Percentage of push transactions completed within their deadlines (prototype experiment set 3, 10% updates, Zipf access distribution).

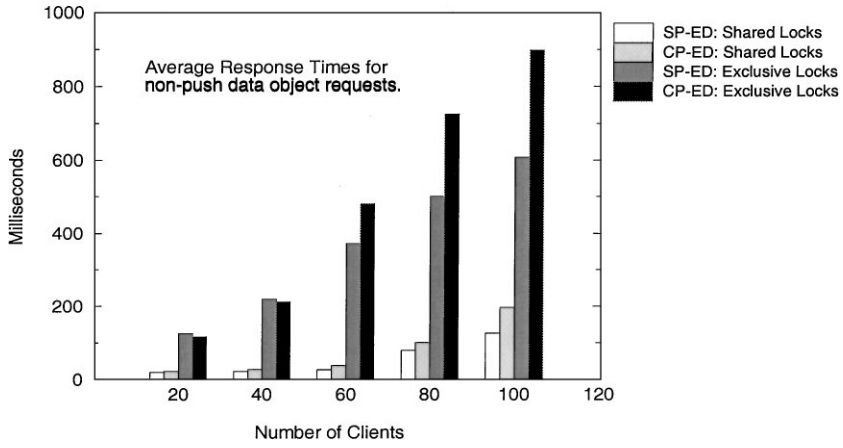

Figure 23. Average response times for non-push object requests for prototype experiment set 3 (averaged over all clients).

Comparing the results of the prototype experiments with those of the simulations, it can be seen that the results in both follow very similar trends. The major differences that can be seen in the two sets of results are:

- The performance level of Server–Push is more stable in the prototype experiments than in the simulations independent of the number of clients attached to the server.
- The percentage of push-transactions that complete successfully is higher in the prototypes than in the simulation experiments. We believe this is because we had over-estimated the disk access and network transmission times in our simulations to reflect worst-case scenarios.

5. Related work

The main question addressed in this paper could be summarized as follows: given a CSD configuration and an update to the database, what are the alternatives in propagating the results of this operation to *interested* clients within specific time constraints? Although, there has been some research into this specific area, there has been considerable interest in two closely related areas, viz, client-server information and database systems, and active rule-based systems.

In [2], Alonso et al. proposed the utilization of individual user's local storage capacity to cache data locally in an Information Retrieval System. This significantly improves the response time of user queries that can be satisfied by the cached data. The overhead incurred is in maintaining valid copies of the cached data at multiple user sites. This overhead is reduced by allowing copies of the data to diverge from each other in a controlled fashion (*quasi-copies*). Propagation of updates to the users' computers is scheduled at more convenient times, for example, when the system is lightly loaded. Delis and Roussopoulos [8]

examine the problem of managing server imposed updates that affect client cached data. They introduce five possible strategies and provide simulation results in support of the hypotheses. The strategies differ mainly in their approaches to server complexity and network bandwidth utilization. In the simplest case, updates are sent to clients only on demand while in more complicated cases, the server maintains a catalog of binding information which designates the specific areas of the database that each client has cached.

Huang et al. [16] propose Static and Dynamic Divergence Caching in order to reduce data transmissions between data servers and application points. These two techniques use the ideas of *tolerant reads*, and (ii) *automatic refresh*. When a client makes use of a tolerant read, it indicates that is willing to accept any of the *n* prior versions of an object. In automatic refresh, the server provides new data to interested client at constant or variable intervals. The type of data refresh designates the divergence caching policy used. Keller and Basu [21] introduce the concept of predicate-based client-side caching in client-server database architectures. Client queries are executed at the server and the results are used to fill the client cache. The contents of client caches are described by means of predicates. Only if a query is not locally computable then it (or a part of it) is sent to the server for execution. Otherwise, the query is executed on the locally cached data. The server also stores predicate descriptions of client caches so that clients can be notified when their cached data has been updated. Several methods are proposed for maintaining the currency of client-cached data: automatic refresh by the server, invalidation of cached data and predicates, or refresh upon demand. Predicate indexing [34] and predicate merging techniques are used to efficiently support examination of a client cache description in answering the cache completeness question.

In [26], a technique for propagating updates in highly replicated databases is proposed. Here, the key idea is to construct a minimum-cost spanning tree of the network interconnections between the hosts in the system. Updates are propagated along the edges of this spanning tree such that when a node *h* receives a message originating at node *o* it is passed on to all the children of *h* with respect to *o*. This procedure ensures that, in the event of no failures, the update is propagated to all the nodes in the system. The notion of continuous queries in append-only database systems is discussed in [38]. A continuous query is similar to a standard query except that once it has been issued, it runs repeatedly at discrete intervals until a pre-specified stopping condition is reached. In the case of append-only databases, these continuous queries can be run either on the entire database to provide a complete result or just on the newly added data records if an incremental result is desired. The design of an event-driven continual query system called OpenCQ is presented in [24]. Here, an active data monitoring mechanism is proposed that re-executes the continual queries whenever the user-specified events and data conditions occur. OpenCQ is a push-enabled system in that the delivery of updated information or query results is accomplished without user intervention. This is an important departure from conventional techniques in DBMSs and the World-Wide Web which have relied upon users to initiate data update requests.

Maintenance of push-predicates can be achieved with the help of an active database system. There is a large body of work in this area which covers many aspects of active databases. They include reaction-rules to updates [10, 13] as well as the usage of integrity constraints and triggers in prototypes [6, 12, 15, 36, 41].

In this paper, we evaluate two alternative automatic data update protocols, which differ from the above approaches in the following ways:

(i) updates to client cached data are delivered in the form of update transactions, and not as up-to-date copies of database objects,
(ii) the delivery of updated data is neither periodic nor does it depend only on the fact that the data has been updated. Instead, database clients are allowed to specify their own sensitivity to change in the contents (value) of the data,
(iii) there are no deadlines on the transactions that perform regular transaction processing.
(iv) the delivery of updated data is constrained by deadlines within which an update to the data must be pushed to an interested client.

6. Conclusions

With the proliferation of high capacity networks and very powerful workstations, the issue of automatic update propagation can now be examined in a new light. In this paper, we propose a data-triggered model for pushing data updates to interested sites in a client-server database framework. Clients provide a set of predicates that indicate the data they are interested in as well as the conditions that the cached data values have to comply with. We develop and evaluate two protocols for the automatic propagation of pertinent updates to client-cached data, namely *Server–Push* and *Client–Push*. The key difference between these two protocols is in the location where the client-predicates are evaluated and push-transactions generated. In Server–Push, data updated at the clients is immediately moved to the server where the set of client criteria is evaluated. If the conditions specified by a client are met, then the update is "pushed" to it. The latter is carried out in the form of a push-transaction (i.e., script). In Client–Push, the set of push-predicates for an object are shipped to the client that intends to update it. Once the object has been updated, the client itself examines the set of push-predicates and ships the push-transaction to interested clients.

These two protocols differ from conventional push techniques in several key aspects, namely:

- Our proposed push protocols are data-driven, i.e., updates are pushed to clients only when the updates cause the contents of the data to satisfy client-specified criteria. This is very different from most existing push techniques which propagate updates to client-cached data either at periodic intervals or if the data changes.
- Instead of refreshing client-cached data through the propagation of copies of the updated data, we suggest the use of updating transactions (*push-transactions*) that make cached data current by executing on them.
- To ensure timely communication of updates to interested clients, we require that push-transactions reach such clients within pre-specified time constraints. Hence, the measure of efficiency that we use is the percentage of updates that are pushed to their destinations within their deadlines.

As the first step in our evaluation, we have developed simulation packages of the two push techniques. Initial results obtained from the simulations indicate that the performance of the two push mechanisms is directly dependent on the number of clients in the system and the workload used. To further establish these results, we have developed prototype systems that ran on a cluster of workstations connected by means of a dedicated LAN. The results of our prototype experimentation closely matched those derived from the simulations. The main conclusions we derive from our experimental results are:

- The Client–Push update propagation technique offers better performance in the presence of a small number of clients. However, Server–Push is much more scalable push alternative. The degradation in the efficiency of Server–Push is very gradual as the number of clients is increased from 20 to 100. These observations were true regardless of the database access distribution and the percentage of updates in the transaction stream.
- In both push strategies, increased contention for database objects reduced the efficiency of the system. However, in the Client–Push strategy, contention for the network also played a significant role in reducing the efficiency of the system.
- As the number of clients increases, the average object response time for non-push requests becomes considerably larger when using the Client–Push strategy, than when using the Server–Push method.
- The time constrained scheduling policies used to prioritize the push-transactions had very little effect on the efficiencies of both systems. The only noticeable effect of these scheduling techniques was that for Server–Push, the FCFS algorithm was clearly inferior to the ED and LS algorithms.

We are presently investigating two areas in which to extend our study of the proposed strategies, namely: (i) the design of push protocols for network configurations other than an Ethernet LAN. In the presence of several separate paths for client-to-client communication it may be possible to devise techniques that offer better performance than Server–Push, and (ii) the use of *a priori* application and user information to detect and further eliminate redundant or unnecessary push-transactions.

Appendix

A.1. Zipf distribution with parameters N (number of ranges) and θ

For ranges 1 to N:

$$\text{Frequency of Range: } F_i = \frac{1}{i^{1-\theta}}$$

$$\text{Probability of Range: } P_i = \frac{F_i}{\sum_{j=1}^{N} F_j}$$

Acknowledgments

This work was supported in part by the National Science Foundation under Grant NSF IIS-9733642 and the Center for Advanced Technology in Telecommunications in Brooklyn, NY. We would like to thank the anonymous referees for their comments and suggestions that greatly helped us improve the presentation of the paper.

References

1. R.K. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: A performance evaluation," ACM—Transactions on Database Systems, vol. 17, no. 3, 1992.
2. R. Alonso, D. Barbara, and H. Garcia-Molina, "Data caching issues in an information retrieval system," ACM—Transactions on Database Systems, vol. 15, no. 3, pp. 359–384, 1990.
3. S. Banerjee and P.K. Chrysanthis, "Data sharing and recovery in gigabit-networked databases," in Proceedings of the Fourth International Conference on Computer Communications and Networks, Las Vegas, NV, September 1995.
4. A. Bouguettaya, D. Galligan, R. King, and J. Simmons, "Implementation of interoperability in large multidatabases," in Third International Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems, Vienna, Austria, April 1993.
5. M. Carey, M. Franklin, M. Livny, and E. Shekita, "Data caching tradeoffs in client-server DBMS architecture," in Proceedings of the 1991 ACM SIGMOD Conference, Denver, CO, May 1991.
6. U. Dayal, B. Blaustein, and A. Buchmann, "The HiPAC project: Combining active databases and timing constraints," ACM SIGMOD Record, vol. 17, no. 1, pp. 51–70, 1988.
7. A. Delis and N. Roussopoulos, "Performance comparison of three modern DBMS architectures," IEEE—Transactions on Software Engineering, vol. 19, no. 2, pp. 120–138, 1993.
8. A. Delis and N. Roussopoulos, "Techniques for update handling in the enhanced client-server DBMS," IEEE Transactions on Data and Knowledge Engineering, vol. 10, no. 3, pp. 458–476, 1998.
9. E. Drakopoulos and M. Merges, "Performance analysis of client-server storage systems," IEEE Transactions on Computers, vol. 41, no. 11, pp. 1442–1452, 1992.
10. K. Eswaran, "Specifications, implementations and interactions of a trigger subsystem in an integrated database system," Technical Report, IBM Research Laboratory, San Jose, CA, USA, 1976.
11. M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt, "Crash recovery in client-server EXODUS," in Proceedings of the ACM-SIGMOD Conference, San Diego, CA, June 1992.
12. E. Hanson, "The design and implementation of the ariel active database rule system," Technical Report WSU-CS-91-06, Wright State University, July 1991.
13. E. Hanson and J. Widom, "Rule processing in active database systems," in Advances in Databases and Artificial Intelligence, L. Delcambre and F. Petry (Eds.), JAI Press: Greenwich, CT, USA, 1992.
14. F. Hayes-Roth, "Rule based systems," Communications of the ACM, vol. 28, no. 9, pp. 921–935, 1985.
15. L. Howe, "Sybase data integrity for on-line applications," Technical Report, Sybase Inc., Emeryville, CA, USA, 1986.
16. Y. Huang, R. Sloan, and O. Wolfson, "Divergence caching in client-server architectures," in Proceedings of the 3rd International Conference on Parallel and Distributed Systems, 1994, pp. 131–139.
17. Mesquite Software Inc, "CSIM—The Simulation Engine," http://www.mesquite.com/tutor96.htm, 1997.
18. Y. Ioannidis, "Universality of serial histograms," in Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.
19. V. Kanitkar and A. Delis, "A case for real-time client-server databases," in Proceedings of the 2nd International Workshop on Real-Time Databases, Burlington, VT, USA, September 1997.
20. V. Kanitkar and A. Delis, "Real-time client-server push strategies: Specification and evaluation," in Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium, Denver, CO, USA, June 1998.
21. A. Keller and J. Basu, "A predicate-based caching scheme for client-server database architectures," The VLDB Journal, vol. 5, no. 1, pp. 35–47, 1996.

22. V. Lee, K. Lam, and S. Hung, "Virtual deadline assignment in distributed real-time database systems," in Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications, Tokyo, Japan, October 1995.
23. C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," ACM-Journal of Association of Computing Machinery, no. 20, pp. 46–61, 1973.
24. L. Liu, C. Pu, and W. Tang, "Continual queries for internet scale event-driven information delivery," IEEE, Transactions on Knowledge and Data Engineering: Special Issue on Web Technologies, vol. 11, no. 4, pp. 610–628, July/August 1999.
25. C. Mohan and I. Narang, "ARIES/CSA: A method for database recovery in client-server architectures," in Proceedings of the 1994 ACM SIGMOD Conference, Minneapolis, Minnesota, May 1994, pp. 55–66.
26. T. Ng, "Propagating updates in a highly replicated database," in Proceedings of the 6th International Conference on Data Engineering, Los Angeles, CA, USA, February 1990.
27. M. Ozsu and P. Valduriez, Principles of Distributed Databases, z/e, Prentice Hall, Upper Saddle River, NJ, January 1999.
28. E. Panagos and A. Biliris, "Synchronization and recovery in a client-server storage system," The VLDB Journal, vol. 6, no. 3, pp. 209–223, 1997.
29. M. Papazoglou, A. Delis, A. Bouguettaya, and M. Haghjoo, "Class-Library support for workflow environments and applications," IEEE Transactions on Computers, vol. 46, no. 6, pp. 673–686, 1997.
30. Ziff Davis Inc., "PC Magazine Online," The Intranet Channel. http://www8.zdnet.com/pcmag/features/push serv/_intro.htm, 1997.
31. K. Ramamritham, "Where do deadlines come from and where do they go?," International Journal of Database Management, vol. 7, no. 2, pp. 4–10, 1996.
32. L. Raschid, T. Sellis, and A. Delis, "A simulation-based study on the concurrent execution of rules in a database environment," Journal of Parallel and Distributed Computing, vol. 20, no. 1, pp. 20–42, 1994.
33. B. Salzberg and V.J. Tsotras, "A comparison of access methods for time-evolving data," ACM Computing Surveys, vol. 31, no. 2, pp. 158–221, June 1999.
34. T. Sellis and C. Lin, "A study of predicate indexing for DBMS implementations of production systems," Technical Report, University of Maryland, College Park, MD, February 1991.
35. D. Siegel, Database Services for the New York Stock Exchange (NYSE). Personal Communication, December 1996. Securities Industry Automation Corporation (SIAC), Brooklyn, NY.
36. M. Stonebraker and G. Kemnitz, "The POSTGRES next-generation database management system," Communications of the ACM, vol. 34, no. 10, pp. 78–92, 1991.
37. MiniRel Development Team. The MiniRel Relational DBMS. University of Wisconsin, Madison, 1989.
38. D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous queries over append-only databases," in ACM SIGMOD International Conference on the Management of Data, San Diego, CA, June 1992, pp. 321–330.
39. A. Thomasian, "Concurrency control: Methods, performance, and analysis," ACM Computing Surveys, vol. 30, no. 1, pp. 70–119, 1998.
40. Y. Wang and L. Rowe, "Cache consistency and concurrency control in a client/server DBMS architecture," in Proceedings of the 1991 ACM SIGMOD Conference, Denver, CO, May 1991.
41. J. Widom, "Deduction in the starburst production rule system," Technical Report, IBM Almaden Research Center, San Jose, CA, USA, May 1991. IBM Research Report RJ 8135.
42. J. Widom and S. Ceri (Eds.), Active Database Systems: Triggers and Rules For Advanced Database Processing, Morgan Kaufmann, 1996.
43. K. Wilkinson and M.A. Neimat, "Maintaining consistency of client-cached data," in Proceedings of the 16th International Conference on Very Large Data Bases, August 1990, pp. 122–133.