A Variable Cache Consistency Protocol for Mobile Systems Using Time Locks

Abhinav Vora, Zahir Tari, and Peter Bertok

RMIT University
Melbourne, VIC 3001, Australia
{avora,zahirt,pbertok}@cs.rmit.edu.au

Abstract. By locally caching data, mobile hosts can operate while disconnected from the central server, however, consistency of data becomes more difficult to maintain. In this paper we propose a protocol that makes a distinction between two classes of consistency: *weak* and *strict*, and treats them differently. Strict consistency is used for data that needs to be consistent all the time, whereas weak consistency is for cases when stale data can be tolerated in the form of requiring only specific updates. Consistency is maintained by using strict read/write and permissive read/write time locks that enable data sharing for a fixed time period and support concurrency control. A Notification Protocol is also proposed, which enables propagating updates to clients and retrieving data from clients in a consistent manner. Performance tests have demonstrated that switching from strict to weak consistency can reduce the number of aborts (due to no access to a lock or conflicting operations) by almost half, even with high read-write sharing.

1 Introduction

With mobile computing devices becoming smaller and faster and networks supporting higher bandwidth and better reliability, mobile computing is becoming a major player in the computing arena and has a strong impact on how applications are designed. Handheld devices enable users to be active participants in distributed computing while on the move. Constraints and restrictions imposed by the mobile and often wireless environments include intermittent and weak connections, wide variations from high bandwidth, low latency through to low bandwidth, high latency to no connectivity at all [3,6].

Mobile clients can switch between connected and disconnected modes of operation in order to either reduce the cost of connection or overcome availability problems. Clients may also have to deal with weak connections that cannot be improved but still they may need to maintain continuous operation regardless of being connected or disconnected [6,7,10].

Disconnected operations are usually supported by caching information on mobile hosts. Lot of work on caching in mobile environments has been done apart from bearing many analogs in Database and File Systems. The Coda File System [7] differentiates between first-class (server copies) and second-class

J.-B. Stefani, I. Demeure, and D. Hagimont (Eds.): DAIS 2003, LNCS 2893, pp. 153–166, 2003.

(client cache) replicas and uses a optimistic concurrency control scheme (based on the premise of low write-sharing) to manage these replicas. In [4], a log-based approach for updating client caches in a relational database system is presented. Updates are centralised at the server; before accessing its cache, a client has to retrieve, from the server, log-records of updates since the cached copy of the data item was last updated.

Our aim here, is not to discuss all this work, but to briefly describe some of the recent and more advanced caching techniques for mobile environments (e.g. [9][8]).

- Pitoura and Bhargava [9] proposed different operations (e.g. *weak read* and *weak write*) in addition to the standard operations in a mobile database environment. Data located at strongly connected sites are grouped together to form clusters. Mutual consistency is required for copies at the same cluster, while different degrees of consistency are tolerated for copies at different clusters. Weak operations are supported within the same cluster, allowing operations on locally available data. Weak reads access inconsistent copies and weak writes make conditional updates. Strict (normal) read/write operations are also supported for strict consistency. This scheme, even though flexible and satisfies varying consistency requirements, is restrictive due to the strict database environment and transactional ACID properties. Furthermore, strict operations are not permitted while clients are in disconnected mode. Such restrictions are generally not necessary or less stringent in non-database environments.
- Lee et al. [8] proposed maintaining cache consistency by using invalidation/update reports. The server periodically broadcasts update or invalidation messages to clients, who update the content of the cache according to these messages. Update reports reflect the changing state of the database. A drawback of this method is that invalidation messages impose a high processing load on clients. Clients have to listen to all reports, even though there may be no changes to the data they cache or they may not even cache the data for which the report was issued or they are not interested in that specific update. Clients cannot disconnect to reduce the usage of the wireless link either, since they need to be connected and wait for the update reports.

In this paper we describe a flexible cache consistency protocol for mobile distributed systems. First, we propose a notification protocol that caters for various consistency requirements. For cached data, clients can specify consistency requirements at different granularity (e.g. attribute level, object level), time intervals between update propagation and predicates (which trigger the updates) for the data they are interested in. This protocol is responsible for propagating and maintaining client specific information.

Data consistency is maintained through the use of shared time-locks, namely "*strict read/write lock*" and "*permissive read/write lock*". All these locks have time limits, after which they expire. Strict and permissive read locks are shareable between clients for the same data item, whereas strict write locks and permissive write locks (called ownership server locks) are exclusive write locks (that

can only be held by one client). The difference between ownership write locks and strict write locks is the way the propagation of writes is performed. Ownership server locks have writes immediately propagated to the server, effectively implementing write-through caching. Strict write locks have writes applied locally to data and updates are flushed to the server at commit time.

Operation efficiency, or its counterpart inefficiency measured by the number of operations rejected as they would lead to inconsistent data, indicates a noticeably better performance of the weak consistency protocol over the strict consistency protocol in every case. With increasing read-write sharing, the performance improvement is increasingly noticeable, e.g. in case of 3/4th overlapping read-write operations the weak protocol rejects only half as many operations as the strict protocol.

Section 2 describes the Notification Protocol, which defines the semantics of the client-server communication. The notions of strict and weak data consistency are defined along with the respective protocols are presented in Section 3. Section 4 discusses the results before concluding the paper in section 5.

2 Notification Protocol

This protocol has two types of messages: Notification Request and Update Notification message. After a short introduction, the two message types are described in this section.

A server manages data and access to that data on behalf of clients. Clients can cache replicas and work remotely on cached data. Multiple clients can cache the same data, hence the server needs to provide a concurrent access scheme to ensure the consistency of both, the data cached by the client and the data residing on the server. The concurrency control method implemented here is based on time locks that clients need to obtain from the server before they can work on locally cached data. In the proposed approach, different clients can have different consistency requirements for the same data item. At the same time any given client can have different consistency requirements for different data items that are cached locally. In addition to catering for the above consistency requirements with guarantees, the proposed protocol also has the following features.

- Clients can specify the granularity at which they would like to maintain consistency. E.g. a mobile funds transfer system that has cached a customer account object locally would like to maintain consistency at a attribute level (balance attribute) instead of the entire account object.
- Clients can specify the degree of consistency required for the cached data item. E.g. the mobile funds transfer system (from the above example) may wish to maintain strict consistency for the balance attribute whereas it might be happy with weak consistency for the address attribute.
- Clients can specify a predicate, whose false condition would trigger notification update messages. E.g. A client can request that updates be sent if only if the balance attribute of a the account object falls below $200.

- Clients can propagate updates to the server. E.g. when the mobile funds transfer system completes an operation, it will update the value balance attribute on the server.
- Clients can receive updates (that are relevant to them and they would like to know about) committed by other clients. E.g. if there are more than two clients caching the balance attribute of the same account object and if one of them updates it, then other clients that are holding a cached copy can receive the update that was committed on the server.

The proposed protocol keeps track of each client's requirements and provides the required consistency.

2.1 Notification Request

In case of a cache miss, i.e. when the requested data item is not available locally, the client retrieves the requested data from the server and puts it in the local cache. However, the client cannot start using the data until it acquires an appropriate lock on that data item. After a lock has been granted the client can analyse the data and send a Notification Request message indicating its consistency requirements for that data item to the server.

A Notification Request message can only be sent by a client holding a valid lock. A client can send multiple Notification Request messages over the duration it holds the locks, the last Notification Message received by the server overwrites any previous Notification Request messages. Between acquiring a lock and sending a Notification Message, other clients can modify or read the locked data item, if they have valid locks (e.g. in the case of shared read-write locks). This situation is dealt in the usual way as described in the Consistency protocol.

The Notification Request message is a tuple of the form

$$(X : T, \{a_1, a_2, \ldots, a_n\}, P \vee Q, \xi_T)$$

where, T is a type and X is an object of type T, a_1,a_2,...,a_n are attributes of X, Δ is the set of all possible attributes; that is $a_i \in \Delta \cup \{ANY\}$, Σ_X is the set of all operations over X, where $\{\text{ANY}\} \subset \Sigma_X$. P and Q are predicates, they are in the form of $\text{S}_1 \wedge \text{S}_2 \wedge ... \wedge \text{S}_n$, where $S_i \in \Sigma_X$ and ξ_T is a time interval specified in seconds.

$X : a_i$ is a pair that uniquely identifies an object and one of its data members. a_i can have the special value of ANY, which represents all data members of object X. P and Q are predicates, which when false would trigger the server to send an Update Notification message. P or Q can have the special value of ANY, which signifies any change to object X. ξ_T represents the maximum delay from the time an update of $X : a_i$ pair is registered until the relevant update notification message is sent. A value of zero signals an immediate update requirement and a value of T indicates that the client can put up with a T second notification delay. In addition to setting the degree of consistency, this field can also be used to improve performance by reducing communication costs,

as the server can combine several updates to clients into a single message. E.g. if there is a outstanding message for client C which needs to be delivered in 10 seconds, and another update is processed which requires a message to be delivered immediately, both of these updates can be sent together to C.

Servers maintains large amounts of data on behalf of clients. An client operation typically accesses only a small subset of the data and setting the granularity of concurrency control mechanisms appropriately, i.e. locking only the required part of the data, can avoid unnecessary blocking of other operations (by other clients) that want to access the non-required data. E.g. if a banking application locks all customer accounts at a branch, only one bank clerk can perform an online banking transaction at any time - which is clearly an unacceptable constraint in most cases. The proposed $X : a_i$ field solves this problem by enabling the client to specify the granularity of the object at which it would like to maintain consistency. A typical use is strict consistency for critical data members of an object, whereas for irrelevant or less important data it could choose not to be notified of modifications or have weak consistency.

When a client does not care/wish to be notified about any updates, it can send an appropriate Notification Request message for that *object:attribute* pair, which will result in the client never being notified of updates on that object. For example, a Mobile Electronic Funds Transfer system would not like to be notified of an address change of the account holder, it would only care for the current balance. Alternatively, when the client has weak consistency requirements, it could specify a Notification Request message of the type $(x : Account, value, ANY, 1000)$. E.g. a share price monitoring system would not care about price fluctuations of stocks that the user does not own or does not plan to watch, while during trading hours may need strict consistency for stock that the user owns, and relatively weaker consistency for stocks that the user has on his watch list.

2.2 Update Notification

Once an update is committed on the server, the update is matched against the predicate of all the Notification Request messages for the updated data item. The condition of update messages is met if the predicate condition of the Notification Request message is true. For every client that meets the condition of update messages, an Update Notification message is sent. This message is of the form:

$$(X : T, \{a_1, a_2, \ldots, a_n\}, \{c_1, c_2, \ldots, c_n\}, TS)$$

where T is a type, X is an object of type T, $\{a_1, a_2, \ldots, a_n\}$ is a set of attributes for which updates are sent (along with the old values), and $\{c_1, c_2, \ldots, c_n\}$ is the corresponding set of new values for the above attributes.

TS is the timestamp indicating when the update was accepted at the server. Clients can examine the TS value and check un-committed operations on the cached data to see whether there were any conflicting operations that need to be rolled-back by the client, such as local updates bearing time-stamps larger than the times-tamp of the update received from the server and using earlier values.

Consistency requirements determine the amount of messaging performed by the server, strict consistency requirements, i.e. immediate notification requests representing disproportionally higher load, because weaker consistency requirements allow grouping of several updates into one message, apart from filtering updates. The server can use a Dependency Table [1] for data and clients to determine which clients are affected by an update and who needs to be notified of the changes.

3 Consistency

This section briefly describes the concept of various time-locks along with their corresponding semantics. We then look at the issue of maintaining consistency. We describe the different consistency levels (i.e. strict and weak consistency) and their corresponding protocols.

3.1 Locks

In traditional and database systems [11,2], a *lock* is an operation that, when applied on a data item, ensures and guarantees continous access to the data item. Locks are used to provide concurrent access in a consistent manner to data when there are several clients accessing the same data simultaneously.

Typically, a lock is assigned to a client until it gives it up. All other lock requests for the same data item cannot be granted until the client holding the lock gives it up. This can be counter-productive for concurrency, since a client could get a lock for a data item and hold it indefinitely or the client could go off-line and be unable to give up a lock. This is a serious issue in mobile systems due to the unreliability of the mobile link. Clients face the prospect of frequent disconnection, coupled with the fact that typically mobile devices have limited battery life, a lock could be held indefinitely, thus effecting the concurrency of the system.

The idea or time-locks has been motivated from *leases* as presented in [5]. A lease gives its holder specific rights to perform some operation for a fixed duration of time. We extend a lease to different types (some being shareable) with varied durations. Time-locks are used as a means to force clients to give-up locks within a preassigned time and as a means of concurrent read-write sharing. The notion of locks being valid for a certain time period is important and required in mobile systems, so that, if clients that are holding locks get disconnected for long periods, the locks can be reclaimed after they expire and assigned to other clients. Also, clients would be aware of the validity of a lock at any given time without having to remain connected. They can now invalidate cached data when a lock expires, thus avoiding operation on stale data that might result in a conflicting operation.

Proposed Locks. We briefly describe the proposed locks, which were presented in [13], followed by a description of lock-time durations. The four types of time-locks are:

(A) **Strict Read Locks** (SRLs)
An SRL is a variation of a traditional read lock. Multiple SRLs can exist at any one time for the same data item. Before performing a write operation, the client needs to upgrade the SRL to a strict write lock (SWL). By obtaining an SRL, a client is guaranteed that no other clients will modify the data item.

(B) **Strict Write Locks** (SWL)
SWL is a variation of a traditional write lock. Only one client can hold an SWL on a data item at a time. Once a client gives up an SWL, or when it expires, the server sends an Update Notification (UN) message to all clients who have a cached a copy of the given data item.

(C) **Permissive Read Locks** (PRL)
PRL is a shared read lock that can be held by multiple clients. On gaining a PRL, all clients get concurrent read access to the data item (i.e. they can read from the local cache). Clients are notified of any updates on the data item while it holds a PRL for it.

(D) **Ownership Server Locks** (OSL)
An OSL is a shared write lock. Multiple clients can have a PRL on data item X while a client has a OSL on the same item x. The client having the OSL employs write-through caching by writing any update directly to the server. The server notifies the other clients holding PRLs on x, of the changes made to x.

Locks can be either upgraded or renewed upon the expiry of their validation period. Table 1 summarises the compatibility for the locks. Based on these lock properties we can split the locks into the following two pair-types: strict pair (SRL and SWL), and permissive pair (PRL and OSL).

Based on these lock properties we can split the locks into the following two pair-types: strict pair (**SRL** and **SWL**), and permissive pair (**PRL** and **OSL**).

Table 1. Lock Compatibility Matrix.

Current	PRL	SRL	SWL	OSL
none	Y	Y	Y	Y
PRL	Y	Y	Y	Y
SRL	Y	Y	N	N
SWL	Y	N	N	N
OSL	Y	N	N	N

Lock Periods. The validity of the time-locks introduced in section 3.1 expires after the lapse of the time period they were granted for. Clients can request for a lock to be allocated for a certain period, however, a server cannot always permit a lock to be allocated for the duration requested by a client, since it has to serve other clients and maintain a high level of concurrency. Hence, the server

moderates the client requested time period and allocates the lock for a duration it determines to be ideal.

We have developed a simple model for determining lock-time periods, considering that a complete model would be very complex and involve parameters like application requirements, past data usage patterns, time of day and popularity of the data. We propose using previous lock times for a data item in determining the maximum time period a client can attain a lock. The actual lock times (opposed to the time a lock was initially granted) for the last n client requests is stored and used to calculate future lock periods. Using the actual lock times of previous operations in determining the duration of locks can be used reasonably accurately to predict the time periods for future requests. Formally, we calculate the average lock duration for a particular data item by

$$t_d^{average_n} = \frac{\sum_{i=1}^{n} t_d^{actual_i}}{n}$$

where $t_d^{average_n}$ is the average lock-time (for last n client requests) for data item d. $t_d^{actual_i}$ is the actual lock duration for the last i^{th} client request.

To calculate $t_d^{average_n}$ we used a value of 10 for n, i.e. the lock times for future requests for data item d would be the average of the actual lock time for the last 10 operations. The general usage of a system is determined by the time of the day, it would be expected that a system would be relatively idle at off-peak (night) times as compared to peak (day) times. If we were to choose a high value of n, it could result in an average time encompassing the entire day or more (depending on system workload and data contention). Locks can be granted for a longer period of time during off-peak times due to the fact that it would not result in decreased concurrency (due to low number of concurrent users), while the converse is true for peak times. Hence, taking an average that spans different times of the day (i.e. different usage patterns) is not an appropriate policy to determine lock times, rather taking an average time from the last n operations in the same usage pattern would result in a more accurate prediction. The value of n is based on the system usage and the contention for data and it needs to be fine-tuned for accurate times.

3.2 Data Consistency

With the existence of multiple copies of data the task of maintaining consistency amongst them is important. It is critical for replicas to be up-to-date with the primary copy (which resides on the server) as well as with other replicas. Consistency between these replicas can be maintained by using a scheme, where each update sent to the server is propagated by the server to all other replicas. Efficiency is an important consideration with cached data. The amount of messages that needs to be sent is proportional to the number of replicas that exist for a data item at any time. For example in a scenario with a high number of write requests, the Update Notification conditions, i.e. the predicates and time delay parameter, have to be selected carefully, otherwise weak consistency may turn

out to be very inefficient, as not all clients would be interested in each update immediately. A few clients might be interested in certain data items only and would like to be notified only if that value changes. Others might want to be notified only if the new value of the data item satisfies a certain predicate (e.g. Notify if x is greater than 10) or might not care about changes in certain data items at all.

The task of maintaining consistency for cached remote data can be split into two tasks: (1) *Maintaining consistency between replicas* and (2) *Moderating concurrent access and managing concurrent updates of data at the server.*

The first task is to ensure that the replicas are consistent with each other and with the primary copy at the server. For example, if an object x is stored at server s and cached at clients c_1, c_2 and c_3 and c_1 was to modify and propagate the updated value of x to s, maintaining consistency would entail s to propagate or inform the other clients, c_2 and c_3 of the new value of $'x'$. The clients c_2 and c_3 could have different consistency requirements and it may or may not be necessary for the server to notify clients with weak consistency requirements immediately. At the same time, clients with strict consistency requirements would need to be notified immediately or have their copy invalidated from the cache.

The second task pertains to managing concurrent updates at the server and ensuring consistency of objects with regards to concurrent operations. It involves ordering the different client operations and determining the final value of the data item. For example, if the server receives an update request for an object x from both c_1 and c_2 at the same time; the server should determine which request is to be satisfied first and whether the second operation should be allowed to proceed or should it be rejected.

The task of maintaining consistency is different for strict consistency requirements and for weak consistency requirements, and we treat them separately. However, the task of moderating concurrent access is the same for both weak and strict consistency requirements, as global consistency of the data needs to be maintained.

If two or more requests for a lock arrive at the server at the same time and only one of the requests can be granted the lock, a decision needs to be made based on which a request came first (First Come First Served – FCFS), how long is the lock period requested, what is the frequency of disconnection of the client, or a combination of these factors. The assignment of locks can lead to deadlocks or clients may starve [2]. A FCFS approach does not guarantee that all clients will be served and it does not stop a client from renewing locks and making other processes starve. Such situations, however, can be dealt with by applying already known techniques mentioned e.g. in [2,11].

Another problem relates to message latency. Let's assume that a request for a lock arrives at the server at time t and the previous write lock was released at time k. If the client server message latency is greater than $(t - k)/2$, the client can have inconsistent data as the Notification Protocol may not have managed to deliver the last Update Message to the client in question. To address this issue, the client request for a write lock needs to include a counter signifying the

number of writes seen by the client on the data item. The server compares this counter with the actual number of writes that have been applied to the data on the server. If these values are not the same, the write request has to be rejected and the client needs to send a new request. The new value of the data can be piggy-backed to the client with the reject message; or alternatively we could wait for the Notification Protocol to deliver the new value. If the server's and client's write counters contain the same value, the lock is granted.

Strict Consistency Protocol. Consistency requirements for a particular data item are termed as strict requirements when the client would like to get immediate notification about every update on the data item and would like all updates it makes to be final. Updates committed on the server are immediately propagated to all clients with strict consistency requirements. For example, a financial application making decisions based on the current balance of an account would immediately like to know of any changes in the balance. Similarly, if an application debits a sum against an account, the effect of this operation has to be final that cannot be rolled back.

A client wanting to maintain strict consistency would use strict locks for the purpose. In case of a cache miss (client request for an operation on data that is not in the client cache) the client request is relayed to the server. The server returns a replica of the data item, and the client has to reply by specifying the granularity and type of its consistency requirements via the Notification Protocol.

Strict Locks. A client wishing to read data, requests an SRL for it. Multiple clients can hold SRLs on the same data simultaneously. A client can request the SRL to be upgraded to a SWL when it wishes to make updates to the data. A request for a SWL is granted if there are no other write locks on the data item. All read locks are revoked when a SWL is granted to a client: the Notification Protocol sends an invalidation message to all clients having a read lock on that data item.

This scheme might seem a bit unfair for clients holding a SRL, since their lock is not guaranteed to stay for the duration it was granted. This scheme can be modified such that no SRLs are revoked, instead, requests for new SRLs are rejected and the request for a SWL is queued. Once all current SRLs expire, the request for the strict write lock can be completed, thus not starving clients that are only interested in reading.

To reduce unnecessary data transfer, when a client's SRL is invalidated, the client is not required to clear its cache, i.e. it can still keep the cached copy, but before reading or writing data it needs to obtain an appropriate lock from the server. When a client sends a requests for any lock to the server, the server checks and ensures that the client has valid data in its cache. If the cached data is invalid, the server piggy-backs the new value. A client c holding a SWL on a data item x can disconnect from the server, work with the locally cached copy, reconnect before the lock expires to either, renew the lock or send the updates to the server and release the lock. The protocol guarantees, for the duration of

the lock, that no other client would read or update x until the lock expires or c voluntarily releases it. When the SWL is released, the server propagates the updates by sending Update Notification messages to all clients holding a cached copy of x.

Weak Consistency Protocol. Weak consistency is used when a client does not need up-to-date data all the time. Update messages may be delayed or even omitted if the changes are considered to be insignificant; e.g. the client can inform the server that changes below a given threshold need not be communicated.

However, a client using weakly consistent data is susceptible to conflicts arising out of read-write or write-write data sharing, since it may have missed an update and read or worked with stale data. Clients employing weak consistency use the permissive pair of locks as defined above. In the following discussion, the use of permissive locks for the weak consistency protocol is given.

Permissive Locks. The permissive pair of locks allows a client to hold read locks on data items that are locked by other clients for writing. Consistency is maintained by forcing the owner of the write lock to make write-through updates directly to the server. The server then propagates updates to clients that hold a PRL on the data item.

As discussed earlier, permissive locks are time locks and are allocated for a period determined by the server. Clients send a request for a Permissive Read Locks (PRL) from the server when they want to read a data item. Multiple clients can hold a PRL on the same data item at the same time. The server assures clients with a PRL that they would be informed of any changes to the data item for which they hold the lock. The maximum delay (assuming that the client is connected) in propagating updates is the message latency from the server to the client. Before the PRL expires, a client can either renew the PRL or upgrade it to an Ownership Server Lock (OSL) in order for the client to retain the PRL or make updates to it respectively. In the case that the client fails to do so, the lock expires.

Once a PRL is granted to a client, the server guarantees, for the duration of the PRL, that all updates on the locked data will be propagated to the client. The client can relax this by nominating a predicate that need to be satisfied in order to receive an update. The client can also specify the maximum delay (update latency) for the duration of the lock, i.e. updates need not to be sent immediately when they are received by the server. Before the PRL expires, the client can either renew it, upgrade it to an OSL (to make updates) or release it.

The OSL is a write lock that forces the holder to make write-through updates. This allows the server to immediately propagate changes to other clients holding a PRL on the updated item. A client with an OSL cannot make changes to data while disconnected, as consistency guarantees can not be accomplised, including propagating updates to PRL holders and imposing write-through updates.

Clients using the permissive pair of locks need to have good connections and must stay connected for the duration of the lock. If they fail to do so, there is a high probability of having conflicting operations due to missed updates. Clients

with less reliable connection or wanting to work offline should use the strict locks.

The two lock-pairs can inter-operate as shown in table 1. For example, two clients could have different consistency requirements for the same data item. One might want strict consistency while the other requires weak consistency. A client with weak read requirements can be easily accommodated with any type of lock (strict or weak), though OSL cannot exist in the presence of any form of strict locks since it would be impossible to provide strict guarantees while there are concurrent weak write locks on the same data item.

This scheme can be seen to give precedence to strict consistency requirements over weak consistency requirements. This can be justified since the chances of a weak operation completing are lower than a strict operation completing. Under such a condition, it would be beneficial (in terms of the system throughput) to aid in the completion and be sure about a commit for a strict operation rather than taking a chance with a weak operation that might have to abort.

4 Discussion

We have simulated the proposed approach and the results were presented in [12]. A comparision of the proposed approaches was made to two-phase locking (2PL) [11] and optimistic concurrency control [2]. The strict and weak protocols were compared to each other and the number of operations completed was measured. The throughput and the scalability of the systems were also simulated.

The proposed approach performs better than 2PL [11] in terms of higher data availability and higher resulting system throughput. 2PL requires that all locks be obtained before any commit, which prevents other clients from getting locks on data items on which locks are already held, even though the clients holding the locks may have finished processing the locked data item. In our approach, the use of permissive locks allows clients to share both read and write locks which results in higher concurrent processing of data. Even though this may lead to aborts if there is a read-write or write-write conflict, effective sharing can be achieved when there are less writers and more readers, since the updates are propagated to clients via the notification protocol.

The performance of the optimistic concurrency control scheme is marginally better than that of the strict consistency protocol. This can be attributed to the fact that a higher number of messages need to be exchanged for the strict consistency protocol.

In terms of the amount of data being transferred between clients and servers, the proposed approach broadcasts less and more relevant data to the clients than the methods that use broadcasting invalidation and update reports [8]. The Notification Protocol provides a method by which clients can elect to be notified only if an event occurs or if a certain class of conditions is true, this dispenses with updates that clients are not interested in. With broadcast reports, the onus of listening to updates is on the clients. This not only limits the clients' ability to work offline, but also results in an increase in the processing load of the client, as it ends up listening to updates it is not interested in, in order not to miss updates

that it is interested in. In our approach, this filtering is done on the server side, and hence it can be guaranteed that all the updates sent to the client are those it is interested in.

The weak consistency protocol works well in situations with high data contention. As shown in [12], the weak consistency protocol performs better by a factor or almost two and a half over the strict consistency protocol. However, the weak consistency protocol requires a high amount of message traffic between clients and servers. There is a linear increase in the number of messages that are exchanged for every write operation. This is due to the weak consistency protocol's use of write-through caching, which also allows it to sustain a higher share rate, and eventually higher operation completion rate. The strict consistency protocol requires a stable number of messages regardless of the number of updates that are made, since updates are only propagated to the server at the end. This also means that other clients cannot receive these updates on time and operations may need to be aborted by the server due to conflicting writes.

The weak consistency protocol is ideal for clients with reliable connections that do not intend to work in disconnected mode. The strict consistency protocol is better for disconnected operations, since it guarantees that if a client holds a strict write lock, no other client can modify that data item. Of course this affects concurrency and could result in higher abort rate of operations by other clients. Starvation of clients requests is prevented by the use of time-locks.

5 Conclusion

In this paper we have demonstrated that the weak consistency protocol, with a linear increase in messages over the number of writes, works reasonably well compared to Two-phase locking, optimistic concurrency control and even the strict consistency protocol. The weak consistency protocol has been shown to be appropriate for clients with relatively strong network connection and who wish to work while connected, while the strict consistency protocol is more applicable for operation in disconnected mode.

The notification protocol presented is responsible for propagating updates to clients, and obtaining and maintaining client specific information such as time interval within which updates need to be propagated and predicates that would trigger an update message. This paper clearly demonstrated that it is possible to support disconnected operations with a reasonable degree of concurrency and low message overhead. A loose sharing method for data was shown which performed very well even under high read/read or read/write sharing environments. A real life example, implemented and run successfully, proved the viability of the method and highlighted its advantages. The proposed techniques harness a tremendous sharing potential and proved to be scalable over a large client base.

Acknowledgements

This work was supported by the Australian Research Council (ARC) Linkage-project grant no. LP0218853.

References

1. J. Challenger, A. Iyengar and P. Dantzig, "A Scalable System for Consistently Caching Dynamic Web Data", *Proc. of the 18th Annual Joint Conf. of the IEEE Computer and Communications Societies*, New York, 1999.
2. G. Coulouris, J. Dollimore and T. Kindberg, "Distributed Systems: Concepts and Design. (2nd Ed)", *Addison-Wesley*, 1994.
3. G. H. Forma and J. Zahorjan, "The Challenges of Mobile Computing", *IEEE Computer*, 27(6), 1994, pp. 38-47.
4. A. Delis and N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures", *In Proc. of the 18th Int. Conf. on Very Large Databases*, August 1992.
5. C. G. Gray and D. R. Cheriton. "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency", *Proc. of the 12th ACM Symposium on Operating Systems Principles*, 1989.
6. T. Imienlinski and B. R. Badrinath, "Wireless Mobile Computing: Challenges in Data Management", *Communication of the ACM*, 37(10), 1994.
7. J. J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System", *ACM Transactions Computer Systems*, 10(1), 1992, pp. 213-225.
8. S. Lee, C. Hwang and H. Yu, "Supporting Transactional Cache Consistency in Mobile Database Systems", *Proce. of MobiDE*, Seattle, 1999, pp. 6-13.
9. E. Pitoura, B. Bhargava, "Data Consistency in Intermittently Connected Distributed Systems", *IEEE Transaction on Knowledge and Data Engineering*, 11(6), 1998, pp. 896-915.
10. E. Pitoura and B. Bhargava, "Building Information Systems for Mobile Environments", *Proc. of 3rd Intl. Conf. on Information and Knowledge Management*, 1994, pp. 371-378.
11. R. Ramakrishnan and J. Gehrke, "Database Management Systems. (2nd Ed.)", *McGraw Hill Publication*, 1999.
12. A. Vora, Z. Tari and P. Bertok, "A Flexible Cache Consistency Protocol for Mobile Systems Using Shareable Read/Write Time Locks", Technical Report TR-02-4, RMIT University, July 2002.
13. A. Vora, Z. Tari and P. Bertok, "A Mobile Cache Consistency Protocol using Shareable Read/Write Time Locks", *Proc. of 9th Intl. Conf. on Parallel and Distributed Systems*, Taipei, Taiwan, December 2002.