Autonomous Transaction Processing Using Data Dependency in Mobile Environments*

IlYoung Chung,[1†] Bharat Bhargava,[1] Malika Mahoui,[2] and Leszek Lilien[1]
[1] *Department of Computer Sciences*
and Center for Education and Research in Information Assurance and Security (CERIAS)
Purdue University, West Lafayette, IN 47907, USA
iy.chung@samsung.com, {bb, llilien}@cs.purdue.edu
[2] *Department of Computer and Information Science*
University of Pennsylvania, Philadelphia, PA 19104, USA
mmahoui@cis.upenn.edu

Abstract

Transaction processing in mobile database systems faces new challenges to accommodate the limitations of mobile environments, such as frequent disconnections and low bandwidth. We propose a transaction processing protocol that increases the autonomy of clients, based on the dependency relation among updated data items. Lists of dependents, sent by the server to the mobile clients along with requested data items, are used by each client to build partial serialization graphs. Utilizing the graphs, mobile clients can autonomously verify serializability of locally executed read-only transactions. This information can also help mobile clients in early detection of the necessity to abort update transactions. Simulations for various data access patterns initiated by mobile clients provide insights on performance of the proposed protocol. Performance is heavily dependent on the depth of the dependency information for each data item.

1. Introduction

Transaction processing has faced new challenges to accommodate the limitations of a mobile computing environment, including low bandwidth and more message losses. To meet them, mobile hosts should be more autonomous in the management of mobile transactions [12]. Fortunately, more powerful mobile devices with moderate storage capacities allow for caching of frequently accessed data on a mobile client and managing data locally [6, 7]. A caching strategy combined with an invalidation strategy ensures that locally stored data are consistent with data stored at the server.

We propose a new transaction processing protocol that provides more local autonomy to mobile clients. Specifically, it allows mobile clients to locally commit read-only transactions (relieving the server from the burden of committing all transactions), and to earlier abort transactions that can not be committed (which reduces the number of transaction aborts).

Several algorithms have been proposed to support concurrency control and commitment of transaction processing in mobile environments [1, 2, 8, 9, 10, 11, 14, 15]. Different local commit strategies for read-only transactions have been proposed in the push-based data dissemination environments, in which data items are broadcast to mobile clients without any explicit request [8, 11, 14]. In the push-based data delivery it is difficult to predict accurately the needs of mobile clients, which results in sending irrelevant data. This in turn worsens use of channel bandwidth and delays delivery of needed data data. We rely on pull-based request-response data delivery with caching.

Due to space limitations, proofs of correctness for our protocol are available only in [16].

2 System Model

Mobile hosts retain their network connections through the support of specialized stationary hosts with wireless communications abilities, which are called *mobility support stations (MSS)*. At any given instance, a mobile host may directly communicate only with the MSS responsible for the

*This research is supported by CERIAS and NSF grants CCR-9901712 and CCR-0001788.

†Currently with Samsung, Seoul, South Korea.

IEEE COMPUTER SOCIETY

cell in which it moves.

Mobile Client Caching Shared data are stored and controlled by a number of database servers executing on stationary hosts. Mobile clients have a limited storage capacity for cached data [12]. Caching frequently accessed data at mobile clients is, despite their limited storage capacity, an effective approach for reducing contention on the narrow bandwidths of wireless channels [6, 7, 12, 13].

Caching does not transfer ownership of data to mobile clients. Servers are the owners ultimately responsible for ensuring the correctness of transaction executions. As a result, some client autonomy is sacrificed. Increasing this autonomy is an objective for our protocol.

Mobile Transaction Processing Each MSS has a coordinator which retrieves transaction operations from mobile clients and monitors their execution in database servers within the fixed networks. Transaction operations are submitted by a mobile client to the coordinator in its MSS, which in turn sends them to the distributed database servers within the fixed network for execution.

In mobile *transaction processing (TP)* systems, on one extreme all data can be placed in and managed by the stationary network [7, 12], and on the other extreme mobile hosts can store and manage all data locally. We adopt an approach between these extremes by storing data locally, but treating them as a cache rather than as a primary copy. A transaction operation that accesses a data item stored in the cache can be processed without interaction with the database server.

In response to a client's message requesting for certification of a transaction at the end of its execution, the server checks whether the transaction satisfies the correctness criteria. It is a *transaction-oriented* or an *optimistic* approach [2, 8, 11, 14] in contrast to the pessimistic or operation-oriented approach [9].

Since in the optimistic approach a mobile client executes a transaction autonomously until all operations are completed, this approach reduces the communications overhead. Most of the existing TP techniques increase local autonomy in this way.

To reduce the burden placed by the certification process on the server and on the limited upstream communication bandwidth, some portions of transaction executions can be transferred to mobile clients .

Asynchronous Broadcasting In the transaction-oriented certification approach, the server broadcasts the list of updated data items along with the results of the certification process. Most of such earlier optimistic schemes adopted periodic broadcasts for the messages [2, 13]. The server broadcasts the list of data items that should be invalidated in the caches and the list of transactions that were committed in the last period.

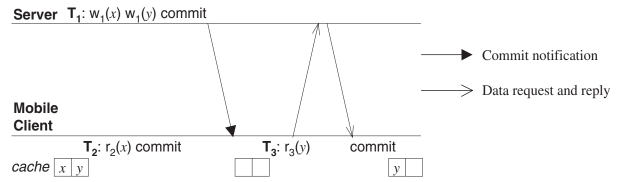

Figure 1. Conflict relation between transactions - an example.

This approach, called *synchronous broadcasting*, has significant weaknesses due to its periodic nature [5]: a high abort rate, degraded throughput (due to transaction blocking), and unnecessary periodic communications overhead when there are few or no updates.

We adopt the *asynchronous broadcast* approach, in which the control messages are broadcast by the server immediately after a commit decision is made by it. The asynchronous approach can eliminate or mitigate the above weaknesses [5].

3 Data Dependency Information

We assume that there is a central server that holds and manages all the data. Each data item in the system is tagged with a timestamp that uniquely identifies the state of the data. The timestamp associated with a data item is increased by the server when a transaction which updates the data item is committed. All data items updated by a transaction are given the same timestamp. If a data item x is updated by T_i, the timestamp of x is referred as $ts(x)$, and has the same value as $ts(T_i)$.

Def. 1 *Let T_i and T_j be two transactions. We say that T_i (directly) conflicts with T_j or that T_i (directly) precedes T_j, iff there exists a data item x such that either:*
1) T_i has performed a read/write operation on data item x with ts(x) = t1, and T_j performs a write operation on x with ts(x) = t2, and t1 $<$ t2, or
2) T_i has performed a write operation on data item x with ts(x) = t1, and T_j performs a read operation on x with ts(x) = t2, and t1 $\leq$ t2.

If T_i directly precedes T_j, then T_i precedes T_j in the serialization order. This is represented by an edge $T_i \rightarrow T_j$ in the serialization graph. Let's see a simple example execution of transactions in Figure 1. In this paper $r_i(x)$ and $w_i(x)$ denote a read and a write operation, respectively, performed by transaction T_i on data item x.

Since transaction T_2 reads a copy of x that was stored in the cache, its timestamp is lower than that of the data item x in the server, which was updated by transaction T_1. As

0-7695-1910-5/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

a result, T_2 precedes T_1 in the serialization order. On the other hand, T_3 follows T_1 because T_3 has read data item y with the timestamp equal to that of the server's original copy of y, which means that T_3 read the value of y written by T_1. As a result, $T_2 \rightarrow T_1 \rightarrow T_3$.

There can also exist indirect conflict relations between transactions, which can be defined as follows.

Def. 2 *T_i w-precedes T_j if there exist $w-1$ transactions T_1, T_2,..., T_{w-1}, such that $T_i \rightarrow T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_{w-1} \rightarrow T_j$.*

By default, T_i 1-precedes T_j simply means that T_i directly precedes T_j, which corresponds to the direct serialization order.

The serialization graph can be produced with transactions initiated by multiple mobile clients, and we can attain serializable execution by ensuring that this graph always remains acyclic [3, 4].

Serialization graph information is delivered (piggybacked on data delivery messages) to each mobile client. It is incrementally updated at mobile clients each time a data object is received from the server. With this information, mobile clients can decide either to commit or abort a *read-only* transaction autonomously, without submitting it to the server. For *update* transactions, a mobile client cannot detect all cycles caused by such transactions without submitting them to the server. However, using conflict information received from the server, it can detect early the necessity to abort most conflicting update transactions, thus saving upstream bandwidth and energy consumption.

In order to describe serialization graph information delivered to mobile clients, we define the notion of order dependency between data items.

Def. 3 *Let x and y be two data items. We define the w-order dependency relation $\ll_{w,t}$ between data items x and y as follows:*
1) for w = 0: $x \ll_{0,t} y$, iff there exists a transaction T_i which performs write operations on both x and y such that ts(T_i)=t
2) for $w \geq 1$: $x \ll_{w,t} y$, iff there exist at least two transactions T_i and T_j, such that:
- T_i has performed a write operation on item x,
- T_j has performed a write operation on item y,
- T_i w-precedes and conflicts with T_j, and
- ts(T_j)=t

Lemma 1 *Suppose that read-only transaction T_k at a mobile client accesses data item y, which was updated by a committed transaction T_j with the timestamp ts(T_j)= t. T_k is involved in a cycle with T_j only if T_k accesses any data item x such that $x \ll_{w,t} y$ and $ts(x) < t$.*

Now we define the list of dependents for a data item.

Def. 4 *Let y be a data item and let t be the timestamp of a transaction that updates y. We define the list of dependents of y with timestamp t for window size w, as follows:*
$Dependent_{w,t}(y)$ =
$\{x \mid (x \ll_{i,t} y)$, where i is the lowest value within [0,w] $\}$
(w is the window size for the i-order dependencies used to define $Dependent_{w,t}(y)$).

The dependency list $Dependent_{w,t}(y)$ includes all data items that were updated by transactions that at most w-preceded the transaction which updated y. In other words, $Dependent_{w,t}(y)$ ignores all i-order dependencies for $i > w$.

In our protocol, to detect any cycle related to a read-only transaction being executed in a mobile client, the dependency list $Dependent_{w,t}(y)$ is piggybacked on the data item y, transferred in response to the request from the mobile client. $Dependent_{w,t}(y)$ includes every data item x which satisfies $x \ll_{w,t} y$ complete with its timestamp $ts(x)$. Hence, by Lemma 1, a mobile client can detect if any cycle is produced by a read-only transaction.

4 The Proposed Protocol

The protocol is given below. One note: Upon receiving data item x with control information ($Dependent_{w,t}(x)$), the client checks whether the transaction T accessing x generates a cycle in G_T^*, the serialization graph composed of G^* and T, in which G^* is the serialization graph for already committed update transactions.

Information maintained at the server This information includes:
1. $invalidation_list$: It includes the list of all data items updated by a transaction, and is attached to the $commit_notification$ message broadcast by the server when the transaction is committed.
2. $Dependent_{w,t}(y)$: $\{x \mid (x \ll_{i,t} y)$, where i is the highest value within [0,w] $\}$

Information created/maintained at mobile clients This information includes:
1. $cache$: A cache of a client includes data items which have already been used by the client for execution of a local transaction. Each data item x in the cache, $cache.x$, has its timestamp, $cache.x.ts$, which was delivered with x from the server. We use $server.x.ts$ to denote the timestamp of a data item x in the server.
2. $read_set$: This is the list of data items that are read by a transaction with their timestamps. The timestamp of an item x in $read_set$ is referred to as $read_set.x.ts$.
3. $write_set$: This is the list of data items which are written by a transaction with their timestamps. The timestamp of an item x in $write_set$ is referred to as $write_set.x.ts$.

IEEE COMPUTER SOCIETY

4. $commit_request$: A mobile client sends this message to the server when an update transaction T_i is completed. This message includes the id of T_i, its $read_set$ and $write_set$.

Algorithm at mobile clients We now present our algorithm for mobile clients.

MA. Transaction execution state
1. Initially each transaction is marked as *read-only*.
2. When a read-only transaction requests its first write operation, its state is changed to *update* transaction.

MB. Cache update
1. Whenever the mobile client receives a $commit_notification$ message, it removes the cached copies of data items that are identified by the $invalidation_list$.
2. When a transaction T_i requests data item x, it receives its value, its timestamp, and the list $Dependent_{w,t}(x)$. For each data item z in the cache, z is removed from the cache if its timestamp $cache.z.ts$ satisfies one of the following conditions:
- $z \in Dependent_{w,t}(x)$ and $cache.z.ts < Dependent_{w,t}(x).z.ts$, or
- $z \notin Dependent_{w,t}(x)$ and $cache.z.ts < t_min$, where t_min =
$\{minDependent_{w,t}(x).y.ts \mid y \in Dependent_{w,t}(x) \}$

MC. Read-only transaction processing
1. If x is in the cache, read it and return.
2. If x is not in the cache, request x from the server.
3. Let A = $read_set \cap Dependent_{w,t}(x)$ and B = $read_set – Dependent_{w,t}(x)$.
4. For each $y \in$ A, if $read_set.y.ts \geq Dependent_{w,t}(x).y.ts$, go to 5; else abort and restart the transaction.
5. If for each $y \in$ B, $read_set.y.ts \geq t_min$, read data item; else abort and restart the transaction.
6. When a *read-only* transaction reaches the end of the execution, commit it locally.

MD. Update transaction processing
1. If x is in the cache, perform write on x and return.
2. If x is not in the cache, request x from the server.
3. Let A = $read_set \cap Dependent_{w,t}(x)$ and C = $write_set \cap Dependent_{w,t}(x)$.
4. For each $y \in$ A, if $read_set.y.ts \geq Dependent_{w,t}(x).y.ts$, go to 5; else abort and restart the transaction.
5. For each $y \in$ C, if $write_set.y.ts \geq Dependent_{w,t}(x).y.ts$, perform operation on data item; else abort and restart the transaction.
6. When an *update* transaction reaches the end of its execution, send a *commit_request* to the server including the id of the transaction, its *read_set* and *write_set*.
7. When a mobile client receives *commit_notification* (*abort_notification*) for a transaction, it commits (aborts) the transaction.

Algorithm at the server The server performs the following algorithm when it receives *commit_request* from a mobile client.
1. For each data item $x \in$ *read_set*, if $read_set.x.ts$=$server.x.ts$, go to 2; else send *abort_notification* to the mobile client.
2. For each data item $y \in$ *write_set*, if $write_set.y.ts$=$server.y.ts$, go to 3; else send *abort_notification* to the mobile client.
3. Insert the identification of the transaction into the *commit_notification* message.
4. Install the values of data items included in the *write_set* in the server database and increment the timestamp.
5. Build the *invalidation_list* by inserting into it each data item from the *write_set* along with its newly computed timestamp. Add this list to the *commit_notification* message.
6. Broadcast the *commit_notification* message.

Protocol correctness Proof of correctness of the server algorithm is trivial. To prove the correctness of the mobile client protocol, we needed only to prove [16]:

Theorem 1 *A read-only transaction committed by a mobile client does not introduce any cycles in the serialization graph.*

5 Performance Study

We very briefly present the simulation model and then present the results of the experiments evaluating performance of our protocol.

Simulation Model Our simulation model consists of the database model, which captures the characteristics of the database, the transaction model, which captures the data object reference behavior, and the system model, which captures the characteristics of the system's hardware and software. The system model consists of one component modeling the database server, multiple components modeling the mobile clients (their number is a parameter for the model), and a wireless network component. Due to space limitations, full details of the simulation model are given in [16].

In the experiments we assumed a database with 400 objects, cache capacity of 40 objects, and mobile client disconnection probability equal 0.2. We simulated three different types of workloads. Workload W1 has a per-client private "hot" region and a shared "cold" region. This workload models an environment where users access and update their own private data, while retrieving some of the shared data items. Workload W2 has a relatively high degree of locality per mobile client and a moderate amount of sharing

IEEE COMPUTER SOCIETY

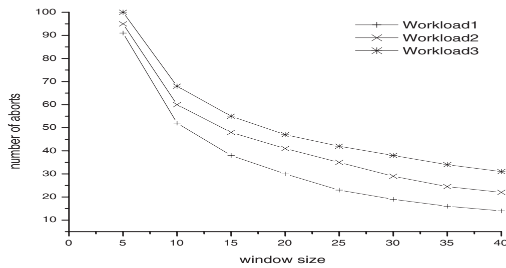

Figure 2. Number of aborts vs. window size.

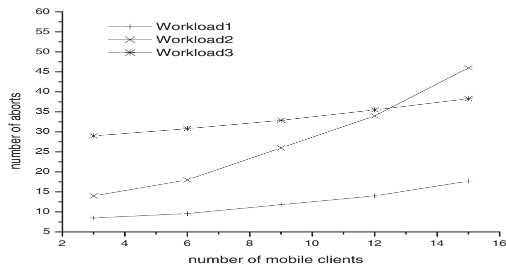

Figure 3. Number of aborts vs. number of mobile clients.

and data contention. Workload W3 is a low-locality and moderate-write-probability workload (models cases when exploiting dependency information is not expected to pay off significantly).

Results and Discussion We have used as our performance metrics the *number of aborts*, the *average waiting time*, and *system throughput*.

Number of aborts - results and discussion Figures 2 and 3 show the average number of aborts that occur before the protocol is able to commit a total of fifty transactions. Figure 2 displays results of varying the dependency window size. When it is small, most of the transactions must restart several times before they are finally committed. For small window sizes, most read operations on cached copies cannot be locally verified by mobile clients. As a result, there is a possibility that a transaction that accessed consistent data and that does not produce a cycle is aborted.

As the window size grows, most of the accesses to cached copies can be verified using the dependency information, so the number of aborts is reduced. When the window is larger, aborts of transactions are due mainly to conflicts between transactions or inconsistencies in cached data caused by lost messages. Under Workload 1, in which most of transactions at mobile clients access their own private data, the protocol shows better performance as the window grows. Under this workload, the probability that $Dependent_{w,t}(x)$ contains dependency information for already read data items is very high. As a result, the mobile client can determine whether an active transaction might produce a cycle in the serialization graph or not. On the other hand, under Workload 3, more transactions are aborted compared with Workload 1. All mobile clients show a uniform access pattern to data items under this workload. As a result, $Dependent_{w,t}(x)$ for a requested data item x has dependency information for many data items from the entire database. This makes the actual length covered by the list smaller.

In case of Workload 1, the number of aborted transactions is not reduced rapidly when the window size gets larger than approximately 30. This is because the number of hot-bound data items for each mobile client is limited. When the window size exceeds a certain threshold value, $Dependent_{w,t}(x)$ may include dependency information for most hot-bound data items for each mobile client. However, under Workload 3, since $Dependent_{w,t}(x)$ can not contain timestamps of all data items that show conflicts with x, some transactions can not be verified using this information. In the experiments, the protocol shows intermediate performance under Workload 2, since each mobile client has its hot-bound data items which are shared with some other mobile clients.

We also examined aborts for the cases when the number of mobile clients gradually increases from 3 to 15. As shown in Figure 3, the number of aborts increases slightly under Workload 1, due to the increased rate of conflicts on cold-bound data items. Accesses to hot-bound data items do not cause conflicts among transactions, since each mobile client has its own private hot-bound data set. Thus, under Workload 1, the number of aborts is less dependent upon the population of mobile clients in a cell. Under Workload 3, the number of aborts also increases slowly as the number of mobile clients gets larger. In this workload, each mobile client shows uniform access pattern for all data items in the database. Hence, most of data items are requested from the server, not from the cache. As a result, the increased population of mobile clients does not significantly change the probability that a transaction can be committed. However, under Workload 2, as the number of clients increases, more cold-bound data items are updated, which can invalidate more cached copies of hot-bound data items. This increases the probability that the timestamp of an accessed data item x in the cache ($cache.x.ts$) is lower than $Dependent_{w,t}(y).x.ts$. Thus, for Workload 2 more transactions are aborted with the increasing number of mobile clients.

Average waiting time - results and discussion In terms of waiting time, we measured the delay for a transaction until it can decide to either commit or abort. As shown in Fig-

IEEE COMPUTER SOCIETY

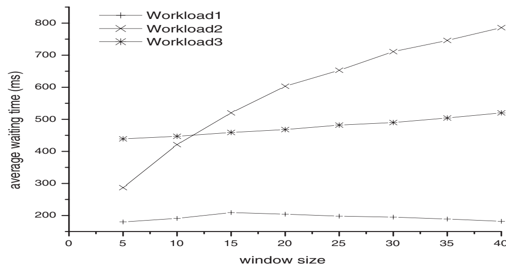

Figure 4. Average waiting time vs. window size.

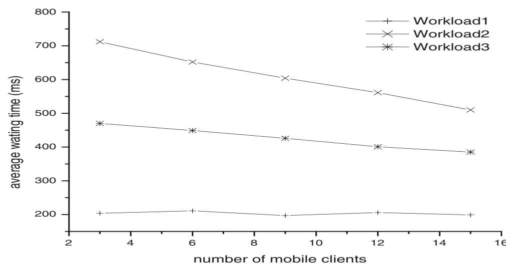

Figure 5. Average waiting time vs. number of mobile clients.

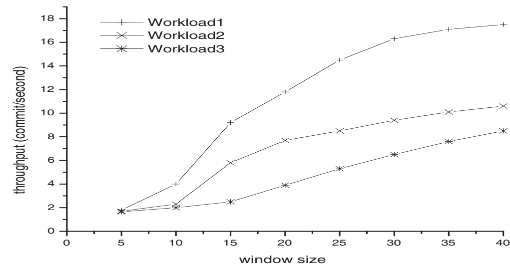

Figure 6. Throughput vs. window size.

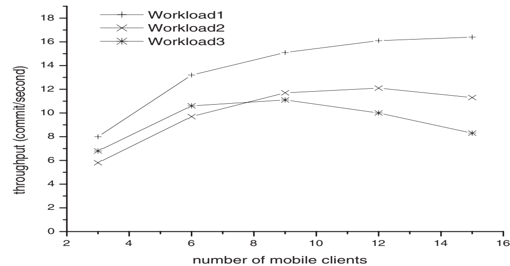

Figure 7. Throughput vs. number of mobile clients.

ure 4, the protocol shows different patterns of waiting time under each workload. For Workload 1, transactions have to wait almost uniformly, approximately for 200 ms. This relatively low waiting time occurs for two reasons. First, most of the transactions are read-only. As a result, mobile clients can verify them autonomously, thus avoiding exchange of messages with the server, which saves time. Second, transactions access most data items from caches, which is very fast compared with requesting data from the server. On the other hand, for Workload 2, it takes much longer to decide whether to commit or abort a transaction, because the ratio of update transactions is relatively high when compared to the other two workloads. The waiting time gets longer with the increasing window size. This can be explained using Figure 2. When the window size is small, most of the transactions are autonomously aborted earlier, using $Dependent_{w,t}(x)$. As the window gets larger, fewer such aborts occur. Hence, most of the update transactions complete execution, and are sent to the server to be verified. In case of Workload 3, the delays are longer when compared to delays for Workload 1, since most of data items accessed by transactions must be acquired from the server. The slight increase of the waiting time for this workload is mainly due to the reduced number of earlier aborts.

Figure 5 illustrates the impact of client population on the average waiting time. The waiting time for a transaction is relatively short under Workload 1, since we can expect that most of the hot-bound data items are maintained in each client's cache. In case of Workload 2, as the number of mobile clients gets larger, conflicts between transactions increase. Some of these aborts of update transactions can be detected by mobile clients without sending the *commit_request* message and waiting for the decision of the server. As a result, the average waiting time decreases with the increasing number of mobile clients. For the same reason, the waiting time for Workload 3 decreases slowly.

Throughput - results and discussion Figure 6 presents the total system throughput for each workload as the window size is gradually increased from 5 to 40. In this experiment, the protocol provides the best performance under Workload 1. The protocol has a lower throughput for Workload 2, and the poorest performance for Workload 3. Due to no per-client locality in Workload 3, there is a relatively high possibility of data conflicts, thus in this case caching is less beneficial than for the other two workloads. When the window size w for $Dependent_{w,t}(x)$ is very small, we cannot expect a satisfying performance for any workload, since most of the transactions have to be aborted when a mobile client receives $Dependent_{w,t}(x)$ attached to the requested data item x.

Figure 7 shows the throughput of the protocol for the

IEEE COMPUTER SOCIETY

population of mobile clients varying from 3 to 15. The high degree of per-client locality and low conflict ratios between accessed data items let the protocol commit more transaction for Workload 1 when compared with the other two workloads. For read-only transactions, the mobile clients can verify more transactions, since the dependency information delivered from the server may contain information for data items that are likely to be maintained in their own caches. For update transactions, more transaction can be committed since there might be fewer conflicts between them. The throughput of the protocol does not degrade for Workload 1, because the increased number of mobile clients does not cause more conflicts between transactions. However, for Workloads 2 and 3, throughput of the protocol degrades when the number of mobile clients exceeds approximately the value 10. This is due to the increased number of conflicts between transactions. For Workload 2, the increased number of write accesses to cold data items is the main factor which invalidates the cached data copies, causing more aborts. For Workload 3, since all mobile clients show the same access patterns over the entire database, the degree of conflicts is directly affected by the number of mobile clients.

6 Conclusions

We proposed a protocol that increases the autonomy of clients in mobile database systems. We defined the dependency relation among updated data items. Lists of dependents send by the server to the mobile clients along with requested data items are used to build partial serialization graphs for each client. By receiving this dependency information, mobile clients can autonomously verify serializability of locally executed read-only transactions. This information can be used by mobile clients to detect early the necessity to abort update transactions.

We have conducted simulations for various access patterns initiated by mobile clients in order to examine the performance of the proposed protocol. The performance of the protocol is heavily dependent on the depth of the dependency information (window size) for each data item. We found that the protocol exploits data access locality. In such cases dependency information may include more related data items that mobile clients need to autonomously verify read-only transactions.

References

[1] S. Acharya, R. Alonso, M.J. Franklin and S.B. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communications Environments," in *Proc. ACM SIGMOD International Conf. on Management of Data*, pp.199–210, 1995.

[2] D. Barbara, "Certification Reports: Supporting Transactions in Wireless Systems," in *Proc. IEEE International Conf. on Distributed Computing Systems*, pp.466–473, 1997.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Massachusetts, 1987.

[4] B. Bhargava, "Concurrency Control in Database Systems," *IEEE Trans. on Knowledge and Data Engineering*, vol.11, no.1, pp.3–16, 1999.

[5] I. Chung, J. Ryu and C.-S. Hwang, "Efficient Cache Management Protocol Based on Data Locality in Mobile DBMSs," in *Current Issues in Databases and Information Systems, Proc. Conf. on Advances in Databases and Information Systems, Lecture Note in Computer Science*, vol.1884, pp.51–64, Springer, 2000.

[6] J. Jing, A. Elmagarmid, A. Helal and A. Alonso, "Bit Sequences: An Adaptive Cache Invalidation Method in Mobile Client/Server Environments," *Mobile Networks and Applications*, vol.2, no.2, pp.115–127, 1997.

[7] A. Kahol, S. Khurana, S.K. Gupta and P.K. Srimani, "An Efficient Cache Maintenance Scheme for Mobile Environment," in *Proc. International Conf. on Distributed Computing Systems*, pp.530–537, 2000.

[8] V.C.S. Lee and K.-W. Lam, "Optimistic Concurrency Control in Broadcast Environments: Looking Forward at the Server and Backward at the Clients," in *Proc. International Conf. on Mobile Data Access, Lecture Note in Computer Science*, vol.1748, pp.97–106, Springer, 1999.

[9] S.K. Madria and B. Bhargava, "A Transaction Model to Improve Data Availability in Mobile Computing," *Distributed and Parallel Databases*, vol.10, no.2. pp.127–160, 2001.

[10] E. Pitoura and B. Bhargava, "Data Consistency in Intermittently Connected Distributed Systems," *IEEE Trans. on Knowledge and Data Engineering*, vol.11, no.6, pp.896–915, 1999.

[11] E. Pitoura and P.K. Chrysanthis, "Exploiting Versions for Handling Updates in Broadcast Disks," in *Proc. International Conf. on Very Large Databases* pp.114–125, 1999.

[12] E. Pitoura and G. Samaras, *Data Management for Mobile Computing*, Kluwer, Boston, 1998.

[13] M. Satyanarayanan, "Mobile Information Access," *IEEE Personal Communications*, vol.3, no.1, pp.26–33, 1996.

[14] J. Shanmugasundaram, A. Nithrakashyap and R. Sivasankaran, "Efficient Concurrency Control for Broadcast Environments," in *Proc. ACM SIGMOD International Conf. on Management of Data*, pp.85–96, 1999.

[15] K. Stathatos, N. Roussopoulos and J.S. Baras, "Adaptive Data Broadcast in Hybrid Networks," in *Proc. International Conf. on Very Large Data Bases*, pp.326–335, 1997.

[16] I. Chung, B. Bhargava, M. Mahoui, and L. Lilien, "Autonomous Transaction Processing Using Data Dependency in Mobile Environments," Technical Report, Department of Computer Sciences, Purdue University, West Lafayette, IN, March 2003.

0-7695-1910-5/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY