Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Data & Knowledge Engineering 45 (2003) 79–100

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Using reordering technique for mobile transaction management in broadcast environments

SungSuk Kim [a,*], SangKeun Lee [b,1], Chong-Sun Hwang [c,2]

[a] *Department of Computer Science and Engineering, Korea University, 1, 5-ka, Anam-dong, Sungbuk-ku, Seoul 136-701, South Korea*
[b] *3G Handsets Lab., LG Electronics Inc., 1042, Hoge-dong, Dongan-ku, Anyang-city, Kyungki-do 431-080, South Korea*
[c] *Department of Computer Science and Engineering, Korea University, 1, 5-ka, Anam-dong, Sungbuk-ku, Seoul 136-701, South Korea*

Received 17 May 2001; received in revised form 15 January 2002; accepted 18 June 2002

Abstract

As computer hardware and wireless network technologies are developed to a high degree, there are many research efforts which intend to utilize data broadcasting to a large population of mobile clients through wireless channels. In recent years, different models of data delivery have been explored, particularly the *periodic push* model where the server repetitively disseminates information without any explicit request. In this paper, we devise new transaction processing algorithms, *O-Post* for update transactions and *O-Pre* for read-only transactions, in broadcast environments. Basically, each client executes its transactions in an optimistic manner and does some consistency checks based on periodic invalidation reports. When any kind of conflicts is found, the conflict order is determined according to the notion of *reordering* and the remaining operations are executed to hold the decision. We also develop a cache algorithm to cope with frequent restarts due to the optimistic execution. Experimental results are given to show the benefits of the proposed algorithms.

Keywords: Wireless mobile computing; Data broadcast; Transaction management; Reordering; Local caching

* Corresponding author. Tel.: +82-2-924-0547; fax: +82-2-953-0771.
E-mail addresses: sskim@disys.korea.ac.kr, sugi91@hanmail.net (S.S. Kim), yalphy@lge.com, yalphy@yahoo.com (S.K. Lee), hwang@disys.korea.ac.kr (C.-S. Hwang).
[1] Tel.: +82-31-389-7383; fax: +82-31-389-7399.
[2] Tel.: +82-2-924-0547; fax: +82-2-953-0771.

0169-023X/03/$ - see front matter
doi:10.1016/S0169-023X(02)00155-6

1. Introduction

Wireless mobile computing is gaining more and more popularity because it can satisfy people's information needs at any time and anyplace. Rapid advances in computer hardware and wireless network technologies have also lead to the development of mobile computing. A portable computer can access data stored in the wired network via base station by wireless communications. In terms of service design and application development, we have to consider some essential characteristics special to the mobile computing as follows [2,7,17,24]:

- *Limited resources.* In a wireless communication, the transmission bandwidth is relatively small. A portable computer has low computing power, small storage space, and small display screen. Moreover, a battery-driven device has limited power.
- *Asymmetric communication.* From the power consumption point of view, sending data is more costly than receiving data for a portable computer.
- *Frequent disconnections.* Due to the interference of various noises or the exhaustion of battery power, a wireless connection will be frequently interrupted.
- *User's mobility.* The mobile computing environment can be regarded as a large heterogeneous environment. Users may move from one cell to another while accessing data. Service handoff, which enables continuous data access in different cells, should be transparently provided.

Briefly, there are two modes for users to access data through wireless channels. One is *broadcast*, which enables users to retrieve data by just listening to a certain channel. The other is *on-demand*, in which users send requests to get data of interest. Data broadcast is characterized by an inherent asymmetry in the communications: the bandwidth in the downstream direction (server-to-client) is much greater than in upstream direction (client-to-server). Thus, it allows users to retrieve data simultaneously with a cost independent of the number of users. Moreover, by broadcasting data, the servers avoid interrupts caused by requests. Generating efficient broadcast program is one of major research topic, which may also be combined with other purpose such as cache management or location-dependent data access [24].

These features cause some new problems and challenges on the mobile information systems [7]; traditional problems such as data format, cache management, transaction processing and so on, need to be discussed again with special emphasis on efficient resource usage and the properties of asymmetric communication.

To support transaction service in traditional distributed systems, many algorithms have been proposed and implemented, among which locking is very popular because of its comparatively stable performance under various environment settings [5]. However, the scheme may not adapt itself well to mobile environments since it originally requires a lot of message exchanges and can not cope with frequent disconnections efficiently. Thus, new transaction processing algorithms are strongly needed and are our main interest in this paper.

The proposed algorithms in this paper, *O-Post* for update transactions and *O-Pre* for read-only transactions, make good use of server's broadcast information. That is, to decrease the number of costly upstream messages and to maintain the consistency of mobile transactions, a server continuously disseminates both the values of all data objects and useful control information. Basically, each client executes its transactions in an optimistic manner and also does some consistency

checks *partially* or *fully* based on the periodic control information. In addition, cache data is maintained to reduce transaction span and to support fast re-execution after abortion. The effects of various parameters on the overall performance are also carefully studied through experiments.

The main contributions of our work are twofold:

(1) Because our algorithms decrease the number of aborted transactions and support fast re-execution, mobile computers can make good use of their limited resources.
(2) The server's overhead can be reduced considerably, since the most part of the responsibility for executing transactions is transferred to clients.

1.1. Related works

Broadcast-based data dissemination is likely to be a major mode of information transfer in mobile computing and wireless environments due to its (almost) unlimited scalability. In the near future, thus, they seem to be used to run sophisticated applications such as transactions. Many research efforts, however, have been limited to the development of mobile cache management algorithms [2,6,9]; that is, given the limited amount of bandwidth available for clients to communicate with the server in wireless environments, achieving cache coherency efficiently has been a challenging research issue.

In the work [15], the authors proposed new validation schemes, called *COREV* and R^2*COREV*, to remove the infinite delay and the cascading delays in validation process. To determine the validation order, two timestamp values—data-transfer completion time and local completion time—are assigned to transactions. In [4], the server disseminates some control information, called *Certification Report*, periodically. Clients execute their transactions optimistically and do some consistency checks partially based on the information, thus reducing the number of messages to the server.

Recently, there are some works to support transactions effectively in *pure push* environment where the server broadcasts data objects without any explicit request. In [22], two protocols, *F-Matrix* and *R-Matrix*, are proposed for mobile read-only transactions. Although *F-Matrix* shows good performance, it suffers from high overhead in terms of expensive computation and high bandwidth requirement for additional control information for consistency check. Furthermore, the two protocols adopt *update consistency* as a correctness criterion for transaction processing, which is weaker than serializability. In [20,21], a number of broadcast methods have been introduced to guarantee correctness of read-only transactions. In particular, the multiversion broadcast approach pushes a number of older versions for each data object, along with the recent version, to improve the possibility of transaction commitment. However, this approach increases the size of one broadcast cycle considerably and accordingly transaction response time. Moreover, the serialization order is fixed at the beginning of the read-only transaction, which is too restrictive and lacks flexibility. For *Serialization-Graph Testing* method, both mobile clients and the server have to maintain a copy of serialization graph for conflict checking. It incurs high overhead to maintain the serialization graph. In the work [14], each data object is associated with a timestamp being determined by a committed transaction which updated the data object finally. The algorithm offers both autonomy among mobile clients and the flexible adjustment of serialization order of mobile read-only transactions. However, the cost for managing timestamp may

be too high in the mobile environments. Authors in [18] propose a new concurrency control algorithm that solves the inconsistency problem that may be observed by the client during the broadcast. They use *Broadcast and Updated Cycles* for the purpose of conflict checking. However, the overhead to maintain and to transmit the control information depends on the size of database, which may not be small.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the system model and the notion of reordering. Section 3 describes the proposed reordering-based algorithms. Local caching technique is then described in Section 4. Related issues with mobile transaction processing are considered in Section 5. Section 6 presents a set of experiment results. Finally, Section 7 concludes the paper.

2. Preliminaries

2.1. System model

The adopted model for a mobile information system in this paper (Fig. 1) is similar to that specified in [1]. The mobile computing environment consists of two distinct sets of entities: a larger number of mobile clients (MCs) and relatively fewer, but more powerful fixed hosts (or data servers). The fixed hosts are connected through a wired network and may also be serving local terminals.

Data servers maintain data objects and process commit requests of update transactions. That is, update transactions which completed all data operations in mobile computers are delivered for validation to servers (the detailed algorithms related with transaction processing will be explained in Section 3). In addition, servers periodically broadcast data to a large mobile clients population. In [1], the authors devised new broadcast structure, named "broadcast disk", according to dif-

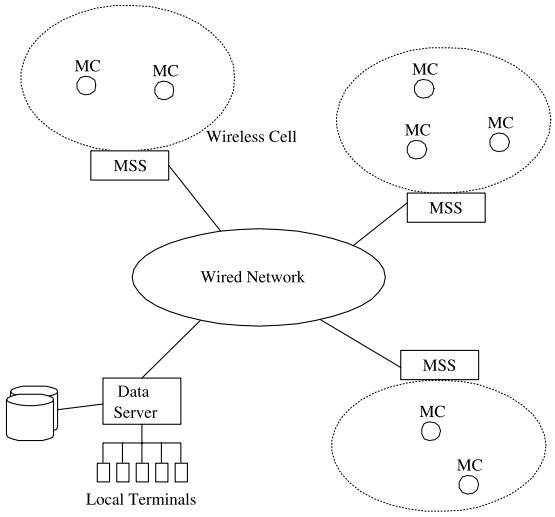


Fig. 1. Mobile computing system model.

ferent access frequency based on users request profile. By allocating more bandwidth to more frequently accessed data, they improved average waiting time. The authors in [10] proposed efficient indexing structure to indicate when a data object is shown and thus to reduce power consumption. Some other structures are presented for variable purposes in the recent works [17]. Generating efficient broadcast program, however, is beyond our main scope and we just assume that broadcast organization is flat [1]. Furthermore, as in the work [20,21], we assume that the values of data objects that are broadcast during each cycle correspond to the state of the database at the beginning of the cycle, i.e., the values produced by all transactions that have been committed by the beginning of the cycle.

The smallest logical unit of broadcast is called *bucket*. Buckets are the analog to blocks for disks. The first bucket of broadcast, BC_i ($i \geqslant 0$), has a serial number i, which indicates that servers will start to send $(i+1)$th periodic broadcast. Before broadcasting data object, servers will allocate some buckets for useful information, $BC_i.Cont$, and send them first. The buckets contain the following information, which were collected during the last cycle:

$$BC_i.Cont = \{Update, Commit, Abort\}$$

Update: the identifiers of data objects updated by committed transactions; *Commit*: the identifiers of committed transactions; *Abort*: the identifiers of aborted transactions.

Some of the fixed hosts, called mobile support stations (MSS), are equipped with wireless communication capabilities. Each MSS can communicate with MCs that are within its radio coverage area, called a wireless cell. It does a role of relaying messages between servers and clients; it gets over the service requests originated in its cell to servers, collects necessary information and broadcasts on behalf of servers.

An MC can connect to a server through wireless communication channel and views broadcast as a disk. Data operations of mobile transactions are executed in an optimistic manner based on data objects being broadcast. If it receives $BC_i.Cont$ while executing transactions, it first examines the correctness according to some criteria.

To simplify the system model, particularly, we do not consider the other roles of MSS except for communication. We also assume that there is only one data server; that is, we do not consider the issues related with the data distribution or replication.

2.2. The notion of reordering

Generally, *serializability* (SR) is adopted as a correctness criterion in transaction processing systems [5]. Although several algorithms have been developed, we take an optimistic concurrency control algorithm. This approach seems to be appropriate in mobile environments, because it needs a small number of messages for maintaining transactional consistency and it can make use of broadcasting facilities from the server. In particular, broadcasting both data value itself and some useful information ($BC.Cont$) can let mobile users check whether transactions execution is correct or not even though transactions are now in active state. If any kinds of conflict are found, the transactions have to be aborted. However, the more data conflicts occur, the more transactions are aborted. At that point, if we are able to determine the conflict orders in the order of the versions of accessed data value, rather than in execution order, some transactions can execute

their remaining operations according to determined conflict order. Therefore, we deploy the notion of *reordering* to reduce the number of aborted transactions.

In this paper, we classify the conflict types into *w–w*, *w–r*, and *r–w*. Here, *a–b* conflict means that the transaction which has been committed in the server executed *a* operation, a mobile transaction executed *b* operation, and *a* conflicts with *b*. If a mobile transaction finds conflicts from *BC.Cont*, it determines the conflict orders based on accessed data value not operation execution order. Throughout the next subsections, we will explain related issues in detail.

2.2.1. Post-reordering for update transactions

If *w–w* or *r–w* conflicts are found for an update transaction T_m, we will determine the conflict orders as *Post-Reordering*. Let us consider Fig. 2(a).

The fact that a transaction T_s which accessed data object *x*, *y* and was committed in the server will be shown to clients when they listen to $BC_i.Cont$ (in the detailed algorithm, the information related with data objects read by T_s does not need to be sent). During the same cycle, T_m executed two update operations, which involve conflicts with the operations of T_s. When *r–w* or *w–w* conflicts are found, we can determine that the operations of T_s were executed first and then the write operations of T_m are executed considering broadcast data. That is, if we consider the final states of the data objects, the result is the same as the decision made. Therefore, we can ignore *r–w* or *w–w* conflicts because the committed transactions in the server never depend on the data updated by T_m.

Definition 1. If a client finds *r–w* or *w–w* conflicts from $BC_i.Cont$, it determines that the committed transactions in the server were executed before the mobile transaction. The conflict order determination for those kinds of conflict is called *Post-Reordering*.

Although *Post-Reordering* is applied to current mobile update transactions, it has no effect on their correctness. Therefore, we can ignore the process of checking the occurrence of *r–w* or *w–w* conflicts. However, in case of *w–r* conflict, it may violate the transactional consistency among them. As a consequence, if *w–r* conflicts are found, the mobile transactions should be aborted. The server only needs to send the set of updated data objects.

Lemma 1. *In case of r–w or w–w conflict on update transaction, conflict order decision as Post-Reordering does not violate serializability.*

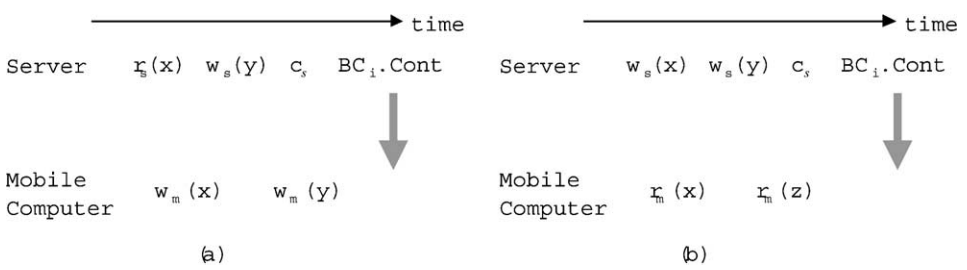


Fig. 2. Examples for reordering. (a) Post-reordering, (b) pre-reordering.

Proof. Let us assume that *r–w* or *w–w* conflicts between T_m and T_{si} ($i \geqslant 1$) have occurred after a mobile computer received $BC_j.Cont$. If *w–r* conflicts are found, T_m is aborted. Obviously, *r–w* or *w–w* conflicts are concerned with write operations, $w_m(x)$, of T_m. Since T_{si} has been committed and they would never depend on the result of write operations of T_m, we can determine the serialization order as $T_{si} \rightarrow T_m$. After that, the occurrence of *r–w* or *w–w* conflicts, if any, also has the same ordering. Therefore conflict order decision as Post-Reordering does not violate SR. □

2.2.2. Pre-Reordering for read-only transactions

Note that the unconditional abort due to *w–r* conflict in *Post-Reordering* may cost too much if most of transactions are read-only such as stock trading, weather information or traffic updates, and so on. Therefore, whenever *w–r* conflicts occur for read-only transactions, we determine conflict order as *Pre-Reordering*. Let us consider Fig. 2(b).

In the figure, a mobile transaction T_m reads a data object x. However, the data object x is also updated by the committed transactions T_{si} ($i \geqslant 1$) during the same or later cycle. When T_m listens to $BC_i.Cont$ from the next cycle, it will find the occurrence of *w–r* conflict. Considering the accessed data values, we can determine the serialization order as $T_m \rightarrow T_{si}$. Like this, instead of unconditionally aborting T_m, we determine that the read operation was executed before write operations. After that, however, we have to execute the remaining operations in a pessimistic manner in order not to violate the decision made. That is, the transaction should be aborted if any of the remaining operations may violate the pre-reordering decision.

Definition 2. If *w–r* conflicts occur, we determine that read operations of a mobile transaction happen before the write operations of committed transactions in the server. The conflict order determination for *w–r* conflict is called *Pre-Reordering*.

Lemma 2. *In case of w–r conflict on read-only transaction, conflict order decision as Pre-Rerodering does not violate serializability.*

Proof. *w–r* conflicts are related with one or more committed update transactions, T_{si} ($i \geqslant 1$), during the last cycle. The inapplicability of *Pre-Reordering* means that T_m has read the data objects which were updated by T_{si}. However, broadcast program is generated periodically and the updated data values will be shown only when next cycle broadcast starts. As a result, although *w–r* conflicts are found, the serialization order can be determined as *pre-reordering*. □

3. Proposed scheduling algorithms

3.1. Optimistic algorithm based on post-reordering: O-Post

3.1.1. Mobile computer's algorithm

Operations of T_m are executed optimistically. When a client listens to $BC_i.Cont$, it only checks the occurrence of *w–r* conflicts; if found, it aborts T_m; otherwise, it continues to execute the

```
◇ when receiving a r(x) (or w(x, new_value))
  - execute it from broadcast data
  - ReadSet (or WriteSet) ← x (or (x, new_value)).
◇ when receiving a commit operation
  - send the commit request with MobileData and then wait.
◇ when receiving BC_i.Cont
  if (T_m is active) /* check for the occurrence of w-r conflicts */
     S ← ReadSet ∩ BC_i.Cont.Update.
     if (S ≠ null) abort T_m;
     else BC_id ← i.
  else /* when it waits for the result of commit operation */
     if (BC_i.Cont.Commit ∋ T_m.T_id) commit T_m;
     else if (BC_i.Cont.Abort ∋ T_m.T_id) abort T_m;
     else wait for BC_{i+1}.Cont
```

Fig. 3. *O-Post* algorithm for mobile computers.

remaining operations. After all data operations are executed, it sends a commit request to the server with the following *MobileData*, and waits for the result:

$$MobileData = \{T_{id}, ReadSet, WriteSet, BC\_id\}$$

T_{id}: transaction identifier; ReadSet: the set of data objects which T_m has read; WriteSet: the set of data objects and its new value which T_m has updated; *BC_id*: The identifier of broadcast which T_m has received recently.

By listening to $BC_i.Cont$ from the next cycle, a mobile client can obtain the result of its commit request. *BC_id* should be maintained to prepare against the difference between the cycle that a transaction sends the commit request and the cycle the server starts to validate for it. That is, for example, due to the network delay or wait in the server queue, there may exist some $BC_i.Cont$ which a client has not considered since it sent a commit request to the server. Fig. 3 summarizes a mobile computers algorithm.

3.1.2. Server's algorithm

The roles of the server are as follow:

- processing commit requests for update transactions
- collecting the necessary information for the next *BC.Cont*
- broadcasting both control information and all data objects

When a commit request arrives, the server only checks the occurrence of *w–r* conflicts between T_m and the transactions which were committed but does not be shown to T_m. That is, the server has to consider all the transactions, T_{si} ($i \geqslant 0$), which were committed after *MobileData.BC_id*. The result will be added on the next *BC.Cont*.

Theorem 1. *Every history H generated by O-Post algorithm is serializable.*

Proof. T_m is only compared with the committed transactions T_{si} ($i \geqslant 0$). If *w–r* conflicts are occurred, T_m is aborted. If *r–w* or *w–w* conflicts are found, the conflict order is determined as *Post-Reordering* ($T_{si} \rightarrow T_m$). Therefore, the history *H* generated by *O-Post* algorithm has no cycle in the serialization graph; the history is always serializable. □

3.2. Optimistic algorithm based on Pre-Reordering: O-Pre

3.2.1. Mobile computer's algorithm

When *w–r* conflicts are occurred between a read-only transaction T_m and T_{si} ($i \geqslant 1$), we can determine that read operations are executed before update operations ($T_m \rightarrow T_{si}$, *pre-reordering*).

Once T_m is *Pre-Reordered*, the remaining operations are executed in a pessimistic manner. First of all, the client has to maintain $BC_i.Cont.Update$ into *UpdateList* for the remaining read operations. When a read operation $r(x)$ among the remaining operations is submitted, the client checks whether data x is an element of *UpdateList*; if so, T_m is aborted. This is to prevent the incorrect result which may be generated if the read operation is executed.

For *O-Pre* algorithm, each client should maintain the following information:

$$MobileData = \{T_{id}, ReadSet, UpdateList\}$$

T_{id}: transaction identifier; *ReadSet*: the set of data objects which T_m has read; *UpdateList*: data objects contained in $BC_i.Cont.Update$.

Note that *UpdateList* will be used only after the transaction has been pre-reordered. Fig. 4 summarizes *O-Pre* algorithm for mobile read-only transactions.

As shown in the algorithm, once T_m is pre-reordered, the check for *w–r* conflicts is not needed. To explain the reason, let's assume that another *w–r* conflicts occurred after being pre-reordered. However, the decision for conflicts is similar with that of the previous conflicts, which does not violate pre-reordered conflict orders. Of course, if the pre-reordered T_m tries to access data objects which are elements of *UpdateList*, it is aborted. Therefore, the conflict check can be omitted.

```
◇ when receiving a r(x)
  if no w-r conflict occurred yet
    - execute it from broadcast data; ReadSet ← x
  else /* if the transaction has already been pre-reordered */
    if (x ∈ UpdateList) abort T_m
    else    execute it; ReadSet ← x
◇ when receiving a commit operation
  - commit unconditionally
◇ when receiving BC_i.Cont
  if no w-r conflict occurred
      S ← ReadSet ∩ BC_i.Update
      if (S ≠ NULL) /* pre-reordering */
        UpdateList ← BC_i.Cont.Update
  else UpdateList ← UpdateList ∪ BC_i.Cont.Update
```

Fig. 4. *O-Pre* algorithm for mobile computers.

Read-only transactions can commit without contacting with the server. This is because the pre-reordering does not violate *SR* (Lemma 2) and the remaining operations are executed to preserve the pre-reordering (Theorem 2).

3.2.2. Server's algorithm

Unlike *O-Post* algorithm, the server only needs to broadcast the control information and data objects.

Theorem 2. *Every history H generated by O-Pre algorithm is serializable.*

Proof. If *w–r* conflict is found, the serialization order is determined as $T_m \rightarrow T_{si}$ ($i \geqslant 1$) according to *pre-reordering*. After that, let's assume that another edge, $T_{sj} \rightarrow T_m$ is added into history H. At the server, all update transactions are committed in order of validation. Therefore, the new edge is added only after T_{sj} is committed and T_m has read the data objects which T_{sj} has updated. However, the client can detect such conflicts by executing the remaining operations in a pessimistic manner and then, abort T_m. As a consequence, the history H generated by *O-Pre* algorithm has no cycle in the serialization graph; the history is always serializable. □

3.3. Discussion

Firstly, the proposed algorithms, *O-Post* and *O-Pre*, are very simple. The things that clients have to do to execute mobile transactions are to tune up broadcast channel and to maintain cache data, which do not need complex algorithms. Since the algorithms are mainly based on optimistic execution, they need a small number of message exchanges; in case of an update transaction, only one message from a client to the server is needed when all data operations are completed and the validation request is delivered. In case of a read-only transaction, there is no message exchange between a client and the server. This property is very desirable in mobile computing systems since scalability can be improved considerably when broadcasting mechanism is adopted. The server is only in charge of both validating update transactions and broadcasting database contents periodically. And mobile clients also make good use of poor battery power and low bandwidth.

Data currency may be an important attribute for some applications. In this paper, however, the currency of broadcast data is older than that of database in the server since new values are continuously generated by committed transactions while the server disseminates data items. And the overall throughput in an optimistic execution may be deteriorated under some environments in spite of the notion of reordering.

4. Local caching technique

Client caching reduces not only the latency but also the span of transactions, since transactions find data of interest in their local cache and thus need to access the broadcast channel for a smaller number of cycles; if local cache is not supported, the average data access time on a single item is a half cycle, which increases response time excessively according to the volume of the data being broadcast. In this section, we discuss the related issues to mobile transactional cache management.

4.1. Cache management

Disseminating both the values of data objects and additional control information (*BC.Cont*) can be useful in maintaining consistency of cache data in broadcast environment. Namely, *BC.Cont* which the server broadcasts at the beginning of each cycle is used as an "*invalidation report*" for cache data; invalidated data can be auto-prefetched with recent value passed through wireless channel without imposing any additional load on shared resources [2,3]. The basic assumption for cache management is that updates in the server can not be shown until next cycle and therefore all data on the air during a cycle are in consistent state. For cache replacement policy we simply accept least recently used (LRU) in this paper and don't touch on it in detail.

If cache data does not stale after *BC.Cont* is examined at the beginning of each cycle, it has the same value as that broadcast during the current cycle. Thus, when a client tries to access a data in cache, he or she can just do it with no check except only one case. The case is that a read-only transaction has already found *w–r* conflicts. In this case, the transaction has to check whether the accessed data in cache is an element of *UpdateList*. If so, the transaction has to be aborted; if not, it just executes the operation by using cache data. The reason is also to hold the notion of pre-reordering and to maintain serializability. The remaining parts are the same. Therefore, maintaining and accessing mobile cache are very simple and cost low.

4.2. Supporting fast re-execution

Caching improves the response time for mobile transactions. However, our algorithms are basically based on the optimistic execution, which causes frequent restarts due to abortion, resulting in much worse response time under high conflicts environments. Therefore, we devise cache management algorithm to cope with such cases.

At first, local cache data is divided into the following two parts: *N_Cache* (Normal cache) and *T_Cache* (Transaction cache). *N_Cache* is general purpose cache and the consistency is maintained as explained before. *T_Cache* only maintains the data for active transactions. That is, after a data operation, $o(x)$, is executed, the data x is stored into *T_Cache*. The data in *T_Cache* is never removed until the transaction commits, and the recent values are always auto-prefetched if they become stale. However, it is not reasonable to assume that the transaction will always execute the same data operations after abortion. Consider, for example, the following query from a client:

if $(x > 0)$ then read y; else read z;

If the value of data x is positive, only y is maintained in the cache. If the transaction is aborted and restarted later and the value of x is changed to negative one, it needs the value of data z not y, thus has to wait for the data to appear on the wireless channel. To cope with the problem, a client possibly maintains all data of the query into the *T_Cache* regardless of its actual execution. Of course, *T_Cache* rarely incurs additional overhead to mobile computer in pure-push broadcast environments. Although a transaction is aborted, *T_Cache* maintains almost all data objects that the transaction needs at restart time and thus it comes to nearly the same state as the time before abortion within at most one cycle. As a consequence, based on the notion of "reordering", a large

number of transactions can continue their remaining operations and come to commit although conflicts are found. And by using *T_Cache*, we can cope with a weak point such as frequent aborts in optimistic transaction processing under high conflict environments. Performance improvements from both transaction processing algorithms and cache are shown in Section 6.

5. Related issues

5.1. Disconnections

One key aspect of the mobile computing systems is the ability to deal with disconnections. Client disconnections are very common when data are delivered through wireless channel. There are many research efforts to cope with this problem [6,11,19], many of which focus on maintaining the consistency of mobile cache data. To support arbitrary disconnection pattern as opposed to broadcasting invalidation report, new cache scheme, called *AS*, is proposed in [12].

Our algorithms depend greatly on invalidation report (IR) both to maintain cache data consistent and to execute transactions locally. If a disconnection occurs, a client cannot listen to this information. Thus, it is possible to discard all cache data and temporary results for active transactions although some portion of them are still valid. As a consequence, one of the solutions to disconnections is to send more IRs during several cycles. That is, the server maintains control information for the previous N cycles ($BC_i.Cont =$ the array $[1 \ldots N]$ of $\{Update, Commit, Abort\}$); N turns out the tolerable cycle from disconnections. After reconnection, a client first checks how long it has been disconnected. If a disconnection is within a tolerable range, a client checks all control information which it did not listen to.

Of course, when long disconnections occur frequently, the size of invalidation reports should increase. In this case, *Bit-Sequence* method [11] is applicable to our algorithms.

5.2. Broadcast organization

When the server disseminates data objects to clients, the structure of broadcast data has a special feature according to the primary purpose. For example, the authors in [1] proposed Broadcast Disk approach to reduce the expected delay for a single data request in asymmetric environments. In contrast, the proposed indexing algorithms in [8,10,23] lead to significant improvement in terms of energy consumption while retaining a low access time. In [17], an adaptive access method based on the distributed indexing scheme is developed for the error-prone mobile environment. Like those examples, there are various kinds of broadcast structures, which affect on the mobile transaction processing algorithms. The authors in [16] try to combine broadcast and on-demand data delivery to build a highly scalable information system with limited bandwidth.

Although we consider a flat disk approach in this paper, the server can alternatively generate a multi-disk broadcast program based on a given access probability [1]. In this environment, however, to support transactional service, the following respects should be further considered:

- *Currency interval*: All data objects in one currency interval are in consistent state. If one major cycle consists of several minor cycles, how do we determine the proper interval and how many times are invalidation reports delivered during one major cycle?
- *Allowing explicit request*: There are some advantages if a client is allowed to request data objects through a wireless channel, which may change the server's broadcast program and client algorithms for processing transactions and maintaining cache consistency [13].

If several currency intervals consist of one major cycle, the server may need to disseminate as many control information as the number of currency intervals, since data accesses during some currency intervals may interrupt the consistency of transactions. On the other hand, if clients are allowed to request data objects to the server explicitly, they can access the recent values which are scheduled to be shown at the next cycle, which may also be a problem. To deal with this case, one of solutions is that a client sends the data request with the current timestamp of broadcast. If the server knows the timestamp, it will be able to reply the proper data value.

6. Performance analysis

6.1. Simulation model

In this section, we aim at studying the performance of our algorithms by way of simulation (Fig. 5). The simulation model is similar to that presented in [1].

The server continuously and repetitively broadcasts both all data objects (1 to *NumberOfData*) and control information. For simplicity, we assume that broadcast model is flat; that is, the server broadcasts each data object just once on a single wireless channel during one cycle. As we mentioned in Section 2, all data objects in a cycle are in a consistent state and control information is first delivered at the beginning of broadcast. In the experiments, the update probabilities follow a *zipf* distribution with a parameter *theta* to model the non-uniform access; the first data is updated the most frequently, and the last data is updated the least frequently. We assume that *UpdateRate* data objects are updated during one cycle and default value is 100. There is one queue for storing uplink messages in the server. When an update transaction completes all its data operations, the commit request is delivered to the server and enqueued. The server validates

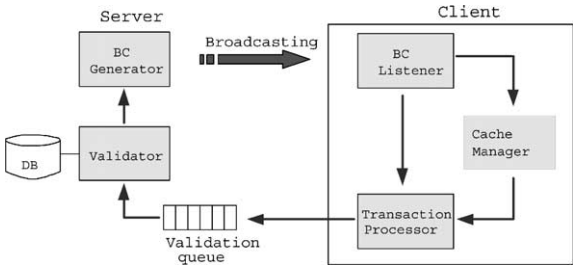


Fig. 5. Simulation model.

Table 1
Parameter description

Parameter	Value	Meaning
NumberOfData	1000	Number of data objects
UpdateRate	Varied	Number of updated data objects during a cycle
Theta (θ)	Varied	*zipf* distribution parameter
NumClient	Varied	The number of mobile clients
CacheSize	100	The size of local cache
AccessRange	400	Average access range for mobile transaction
Offset	0, 50	Disagreement between access patterns
ReadTime	1	Execution time for read operation—time unit
WriteTime	3	Execution time for write operation
BCCheckTime	3	The time for checking when receive *BC.Cont*
msgTransferTime	100	The time needed to send a commit request
ValidationTime	10	Time for validation in the server
RestartTime	10	Time between abort and restart

requests in FIFO mode. Except for a validation request, there is no message from clients to the server in broadcast environments. For simplicity, the issues related with disconnections are not considered.

Mobile transactions access data objects within the range of *AccessRange* which are the subset of the broadcast data objects ($AccessRange \subset [1 \ldots NumberOfData]$). In the range, access probabilities also follow a *zipf* distribution. In case of update transactions, the number of read operations is four times of write operations on average. To model the disagreement between the access pattern of a transaction and update pattern in the server, the first data object in *AccessRange* starts at *offset*. Each client maintains local cache which can hold up to *CacheSize* data objects. The cache replacement policy is LRU in conjunction with auto-prefetching: when the cache is full, invalidated data objects from *BC.Cont* are first replaced; if cache does not contain those data, the least recently used data object is replaced. After invalidated, stale cache data is auto-prefetched from broadcast.

In the experiments, the time unit is set to the time which is needed for the server to disseminate one data object, and we also assume that the time unit is the same as that which is required to execute one read operation in the client. Table 1 summarizes the parameters and the default values.

With the parameter setting, we have implemented our simulator by using Java programming language. In the experiments, we compare our algorithms with others introduced in [20,21]. Specifically, in case of update transactions, *Invalidation-Only* (IO) method is chosen for a comparison counterpart. In case of read-only transactions, both IO and *Multiversion with Invalidation* (MI) methods are considered. This is because these methods accept serializability as a correctness criterion for transaction processing, and the system model is very similar to ours.

IO method is an optimistic algorithm and a client does consistency checks based on periodic control messages. Note that IO method does not consider update transactions. However, we choose it by changing the method a little; namely, a client executes update transactions optimistically. When all operations are completed, a client sends a commit request to the server, which validates the request identically with *O-Post* algorithm. With MI method, by disseminating multi-

versions per each data item, the following two problems can be addressed. One is to handle disconnection and the other is to maintain transactional consistency. We assume disconnection-free environment in the paper. To explain the latter case, let us consider the situation where mobile clients execute their transactions based on broadcast data. If clients can find proper versions on broadcast channel although data items have been updated continuously in server side, the clients can continue their remaining operations, resulting in the decreased number of transaction abortion. For the purpose, the authors in [20,21] devise MI method. In our work, it is assumed that from 1 to 4 version(s) consist of one cycle and thus, the length of a single cycle in MI is longer than that in *O-Pre* or IO methods.

As stated before, the basic cache maintenance algorithm in IO and MI methods is LRU with autoprefetching. That is, the proposed cache algorithm in this paper is applied to only the proposed methods—*O-Post* and *O-Pre*.

6.2. Experimental results

Response time is measured from the time a transaction starts its execution to the time the transaction commits. If a transaction aborts, it has to wait during *RestartTime* and then restarts again. Thus, if it aborts frequently, the response time is longer and longer. In experiments for all methods, mobile computers make use of local cache when executing transactions.

We have made experiments to show the superiority of our algorithms by mainly examining response time. Since the response time suffers from the occurrence of abortion and restarts, the number of aborted transactions is also considered. Once a transaction is aborted, we have the following three choices to examine the effects of abortion on local cache data: (a) the same transaction is executed again, (b) some of data operations are changed, and (c) entirely new transaction is generated. In these experiments, we set up 5:4:1 ratio for the choices.

6.2.1. Response time of update transactions

To allow transactions to be executed by IO method for update transactions, the server disseminates the sets which contain data object identifiers updated and read by committed transactions during the last cycle.

Fig. 6 represents the effect of number of data operations per transaction on response time, when the number of clients is set to constant value ((a) 100 and (b) 300). As the number of operations

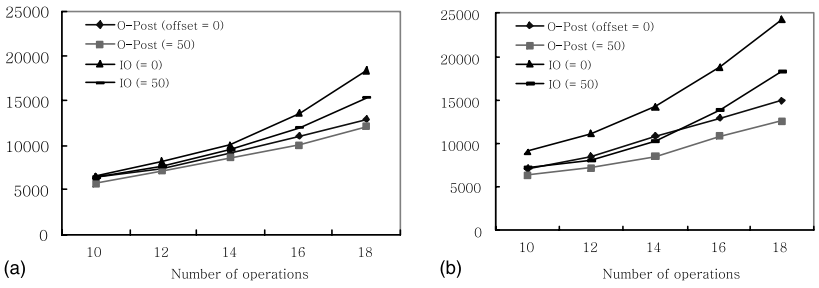


Fig. 6. Effect of number of operations: (a) clients = 100, (b) clients = 300.

increases, the transaction lifetime ranges over several broadcast cycles. This is because, the probability of abortion increases since both methods execute local transactions optimistically. IO method, which is a pure optimistic method, unconditionally aborts transactions whenever any kind of conflicts is found, resulting in high response time. In contrast, the response time in *O-Post* method is comparatively small, because it can reduce the number of aborted transactions by applying both the *post-reordering* technique and the proposed cache algorithm. When a transaction has a small number of operations (in this analysis, 10), the difference between the two methods is insignificant. For a large number of operations (in this analysis, 18), however, *O-Post* is superior to IO almost by a factor of 2.

The figure also contains the results when *Offset* parameter is set to 50. Recall that positive value of *Offset* means that hot data in the server side may not be accessed frequently any more in the client side. We can observe that, although the number of conflicts decreases and the two methods can reduce response time to a certain degree, the tendency of increase in response time is very similar.

Fig. 7 shows the performance behavior of the two methods as the number of clients varies. For this experiment, each transaction has 10 data operations and the number of write operations is set to 2 (Fig. 7(a)) or 4 (Fig. 7(b)). From the figure, we see that the probability of a transaction being aborted one or more times gets higher as the number of clients increases. As expected, in a general case, if a transaction executes more update operations, the response time gets worse.

The performance difference between the two methods, however, is not so big as our expectation in this experiment. This is due to the following reason: as the number of clients who execute update transactions increases, the data objects are more frequently updated in the server. This incurs a lot of conflicts in clients side, thus the overall response time is negatively affected. To cope with the case, some other techniques, such as timestamp and multi-version, could be combined with the optimistic algorithms.

The interesting thing from the results in Fig. 7 is that *O-Post* method with 4 write operations shows better response time than that with 2 write operations, which seems to be counter intuitive. In *O-Post* method, transactions have to be aborted only when *w–r* conflict is found. If there are the same number of operations per transaction and transactions execute more write operations under some condition, the probability of *w–r* conflicts occurrence decreases a little (just 3–7%) and thus we come to obtain the results shown in Fig. 7.

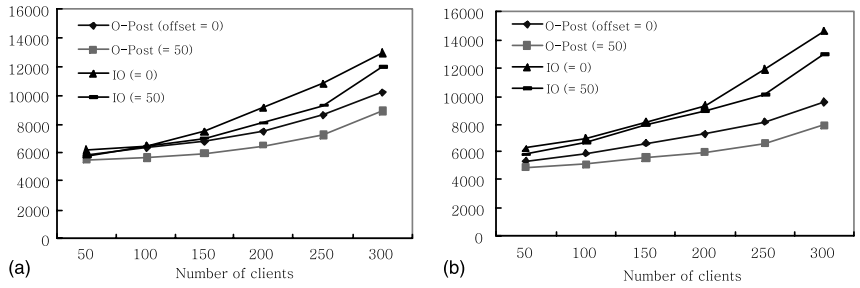


Fig. 7. Effect of number of clients: (a) write operations = 2, (b) write operations = 4.

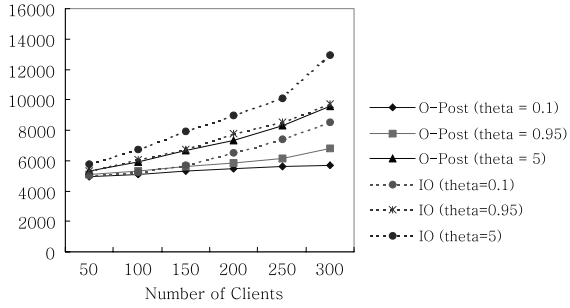


Fig. 8. Effect of theta value in zipf distribution: offset = 0.

Fig. 8 shows the effect of *theta* value in *zipf* distribution. As stated in the system model, the value determined affects the access pattern in the server. If it has a higher value (= 5), most accesses come together into a small portion of data (= hot data), which increases the probability of conflict occurrences. The results follow our expectation: response time suffers from high theta value and frequent updates in the server, which are found in both methods. However, transactions in *O-Post* method can decrease the probability of abortion particularly under low *theta* value. For example, a transaction restarts 1.7 times on average where there are 300 clients and *theta* value is 5, while most transactions commit without abortion under low *theta* value (= 0.1).

6.2.2. Response time of read-only transactions

When we make experiments for read-only transactions, we consider only one client instead of several clients in experiments for update transactions. This is because there is no conflict relationship among them. In case of read-only transactions, three methods are compared. Before we go into analyzing the results, we have to observe the properties of broadcast carefully. In *O-Pre* and IO methods, the server disseminates the value of each data object exactly once during a cycle. For MI method, however, from 1 to 4 version(s) per each data object consist of one cycle. Thus, the length of one cycle with MI method is longer than both *O-Pre* and IO methods. We expect that this difference will be one of dominating factors to the relative performance behavior.

Fig. 9 shows a variation of response time as transactions execute various number of read operations when *UpdateRate* is set to 100. From both (a) and (b), we observe that the response time of IO method is increasing rapidly. This is because a large number of operations cause

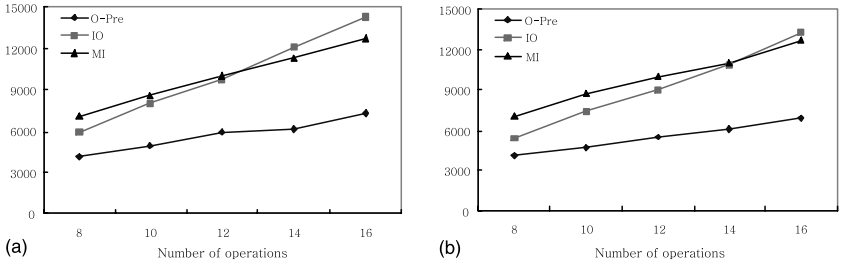


Fig. 9. Effect of number of operations: (a) offset = 0, (b) offset = 50.

frequent conflicts, and decrease the probability of a transaction's commitment. As a result, a transaction suffers from many restarts until it commits. Turning to MI method, it can avoid the problem with IO method by making a client access old versions, thereby increasing the chance of a transaction's commitment. Thus, the performance of MI shows a stable linear distribution. This means that transactions are rarely aborted. Actually, almost all transactions can commit without abortion in MI method. This accounts for the superiority of MI to IO for a moderate and large number of operations. However, some older versions per data object are delivered during a cycle in MI method, which increases the length of a cycle, resulting in poor response time. Therefore, in the case where a transaction has a small number of operations and the probability of conflict occurrence is low, MI is inferior to IO, on the contrary. As opposed, the response time of *O-Pre* is comparatively small and stable, since transactions can execute the remaining operations by applying *pre-reordering* technique although conflicts occurred. Furthermore, our cache management allows the aborted transactions, if any, to re-execute operations fast.

Before examining the effect of update rate, we briefly discuss the effectiveness of local caching with the three methods. For MI method, each cache entry is composed of $\langle dataidentifier, datavalue, version\rangle$. If an active transaction is invalidated, the remaining operations need to access proper versions and thus a local cache has to maintain multiple versions for each data. It naturally drops the effective size of cache and accordingly the cache hit ratio. For example, while *O-Pre* and IO methods shows 12–19% cache hit ratio, cache hit ratio in MI method is approximately 7–12%. Of course, if transactions are aborted one or more times, the cache scheme proposed in this paper shows higher cache hit rate (about 60% in our analysis).

From the first experiment for read-only transactions, we note that the number of aborted transactions in MI method is very small in spite of its poor response time. For example, no transaction is aborted among 1000 transactions where each transaction executes 8 operations and *UpdateRate* is 100. Under the same environments, 9 transactions are aborted in *O-Pre* and 26 transactions in IO. Even when each transaction executes 16 read operations, the number in MI method is only 3. However, 191 in *O-Pre* method and 394 in IO method. From these results, we conjecture that disseminating a lot of information to clients may be useful in some application area where correctness criterion, for example, can be weakened or long transaction has to be supported.

Fig. 10 shows the effect of *UpdateRate* parameter when each transaction executes 10 data operations. The overall behavior in response time is very similar to Fig. 9. Here again, a pure

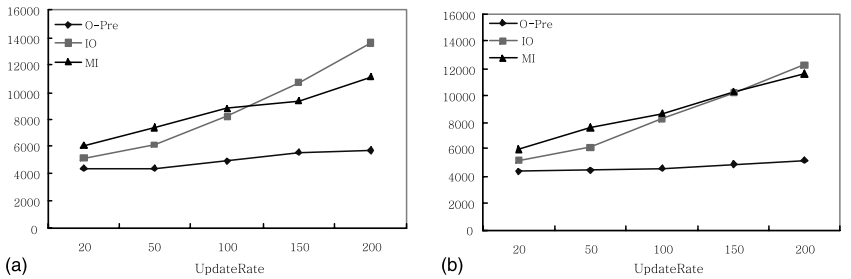


Fig. 10. Effect of update rate: (a) offset = 0, (b) offset = 50.

optimistic method, IO, suffers heavily from frequent update pattern. For example, when *UpdateRate* increases from 20 to 200, the response time becomes three times. The response time of the other two methods, however, does not get worse heavily than IO. Notice that the underlying reason is not same. For MI method, the number of aborted transactions is very small (in our analysis, only 2 transactions are aborted when *UpdateRate* is 200). For *O-Pre* method, although the number of aborted transactions is moderate (= 145), the length of one cycle is very small and fast re-execution after abortion is supported by an efficient local caching.

In addition to the results on average response time in Fig. 9 and Fig. 10, another point we need to address is the worst case response time. If a transaction is aborted several times until commitment, its response time increases highly, which is also unacceptable property in mobile computing environments. In experiments under some condition such as high update rate (= 200) and low *offset* value (= 0), a transaction is aborted 7 times in IO method and 4 times in *O-Pre* method at the worst case. At that case, the response time is 22361 and 8347, respectively. Considering the number of aborts for two methods at the worst case, the reason is also the same as the average case. In contrast, transaction abort merely affects response time in MI method. However, if a lot of data objects are updated in the server and the length of a broadcast cycle is longer due to older versions, response time also deteriorates.

Fig. 11 contains the results from two kinds of experiments. In Fig. 11(a), *theta* value is set to 0.95 and 5 where there are 300 clients and offset value is set to 0. Overall graph patterns are conformed with those of the other experiments. However, there is one point to note from the results of MI method. There are also a few number of conflicts where *UpdateRate* is 200 and *theta* is 0.95 but the number of old versions per data object increases, resulting in extending the length of a cycle. While *theta* is 5, a large portion of data updates are occurred in hot data and most data interesting to mobile transactions may be found in the tail of broadcast, which resulting in longer average waiting time.

Fig. 11(b) shows the effect of local cache size. The response times in all methods are improved but the patterns are not similar. In IO method, there is a limitation in response time improvement since transactions are aborted frequently due to conflicts occurrence. The improvement in *O-Pre* method is mainly from *T_Cache* and thus the range of improvement from cache size is also limited to a degree. In contrast, MI method is less benefited from the same size of cache than other two methods where since old versions are also maintained in the cache. However, as stated before,

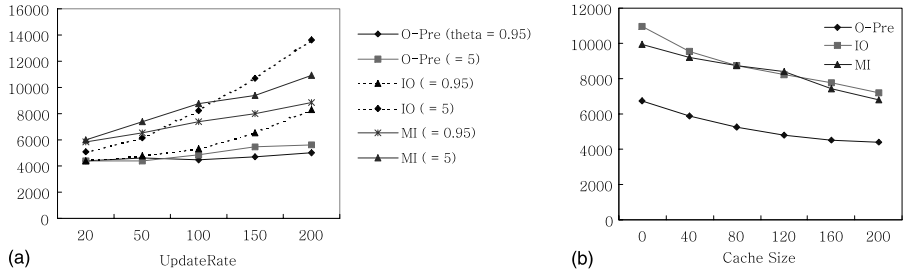


Fig. 11. Effect of (a) theta value in zipf distribution and (b) cache size: client = 100, offset = 0.

there are few aborts in MI method. Therefore, the response time is improved considerably as cache size grows larger and larger.

7. Conclusion

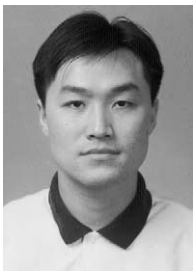
From the limited resource point of view, data broadcast is especially suitable in mobile computing environments. However, the volume of information required for clients to execute mobile transactions is considerable, which may need frequent message deliveries from clients to the server, resulting in poor utilization of resources.

In this paper, we have presented two transaction processing algorithms, *O-Post* and *O-Pre*, which are based on reordering technique. Since mobile transactions are executed optimistically and the role of maintaining transactional consistency is delivered to clients partially or fully, the overhead of the server can be taken off considerably and each mobile client can make good use of its limited resources. In particular, the introduced reordering technique, which is based on both periodic control information and the properties of broadcast data, decreases the number of aborted transactions. Further performance improvement could also be achieved by adopting an effective local cache algorithm, which allows aborted transactions to restart fast.

References

[1] S. Acharya, R. Alonso, M. Franklin, S. Zdonik, Broadcast disks: data management for asymmetric communications environments, in: Proc. of the ACM SIGMOD Conference on Management of Data, 1995, pp. 199–210.
[2] S. Acharya, Broadcast disks: dissemination-based data management for asymmetric communication environments, Ph.D. thesis, Brown University, 1998.
[3] S. Acharya, M. Franklin, S. Zdonik, Prefetching from a broadcast disk, in: Proc. of 12nd International Conference on Data Engineering, 1996, pp. 276–285.
[4] D. Barbara, Certification reports: supporting transactions in wireless systems, in: Proc. of the 17th IEEE International Conference on Distributed Computing Systems, 1997, pp. 466–473.
[5] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison Wesley, 1987.
[6] D. Barbara, T. Imielinski, Sleepers and workaholics: caching strategies in mobile environments, in: Proc. of the ACM SIGMOD Conference on Management of Data, 1994, pp. 1–12.
[7] M.H. Dunham, A. Helal, Mobile computing and databases: anything new? ACM SIGMOD Record 24 (4) (1995) 5–9.
[8] Q. Hu, W.-C. Lee, D.L. Lee, Indexing techniques for wireless data broadcast under data clustering and scheduling, in: Proc. of International Conference on Information and Knowledge Management, 1999, pp. 351–358.
[9] Q. Hu, D.L. Lee, Adaptive cache invalidation methods in mobile environments, in: Proc. of 6th IEEE International Symposium on High Performance Distributed Computing, 1997, pp. 264–273.
[10] T. Imielinski, S. Viswanathan, B.R. Badrinath, Energy efficient indexing on air, in: Proc. of the ACM SIGMOD International Conference on Management of Data, 1994, pp. 25–36.
[11] J. Jing, A. Elmagarmid, A.S. Helal, R. Alonso, Bit-Sequence: an adaptive cache invalidation method in mobile client/server environments, ACM/Baltzer Mobile Networks and Application 2 (3) (1997) 115–127.
[12] K.S. Khurana, S.K. Gupta, P.K. Srimani, An efficient cache maintenance scheme for mobile environment, in: Proc. of the 20th International Conference on Distributed Computing Systems, 2000, pp. 530–537.

[13] S.S. Kim, S.K. Lee, C.-S. Hwang, S.Y. Jung, O-PreH: optimistic transaction processing algorithm based on pre-reordering in hybrid broadcast environments, in: Proc. of 10th International Conference on Information and Knowledge Management, 2001, pp. 553–555.
[14] V. Lee, S. Son, K. Lam, On the performance of transaction processing in broadcast environments, in: Proc. of 1st International Conference on Mobile Data Access, 1999, pp. 61–70.
[15] Y. Lee, S. Moon, Commit-reordering validation scheme for transaction scheduling in client-server based teleputing systems: COREV, in: Proc. of the International Conference on Information and Knowledge Management, 1997, pp. 59–66.
[16] C.W. Lin, D.L. Lee, Adaptive data delivery in wireless communication environments, in: Proc. of the 20th International Conference on Distributed Computing Systems, 2000, pp. 444–452.
[17] S.C. Lo, A. Chen, An adaptive access method for broadcast data under an error-prone mobile environment, IEEE Transactions on Knowledge and Data Engineering 12 (4) (2000) 609–620.
[18] A. Al-Morgren, M.H. Dunham, BUC, a simple yet efficient concurrency control technique for mobile data broadcast environment, in: Proc. of the 12th International workshop on Database and Expert Systems Applications, 2001, pp. 564–569.
[19] S.H. Phatak, B.R. Badrinath, Conflict resolution and reconciliation in disconnected databases, in: Proc. of 10th International Workshop on Database and Expert Systems Applications, 1999, pp. 1–6.
[20] E. Pitoura, P. Chrysanthis, Exploiting versions for handling updates in broadcast disks, in: Proc. of the International Conference on Very Large Data Bases, 1999, pp. 114–125.
[21] E. Pitoura, P. Chrysanthis, Scalable processing of read-only transactions in broadcast push, in: Proc. of the 19th IEEE International Conference on Distributed Computing System, 1999, pp. 432–441.
[22] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, K. Ramamritham, Efficient concurrency control for broadcast environments, in: Proc. of the ACM SIGMOD Conference on Management of Data, 1999, pp. 85–96.
[23] N.H. Vaidya, S. Hameed, Scheduling data broadcast in asymmetric communication environments, Wireless Network 5 (3) (1999) 171–182.
[24] J. Xu, D.L. Lee, Querying location-dependent data in wireless cellular environments, in: WAP Forum/W3C Workshop on Position Dependent Information Services, 2000.



SungSuk Kim received his BS, MS degrees in computer science and engineering from Korea University, Seoul, South Korea, in 1997 and 1999, respectively. He is currently a Ph.D. candidate in computer science and engineering in Korea University, Seoul, South Korea. His current research interests include distributed database systems, mobile information systems and moving objects management. He is a member of ACM and IEEE Computer Society.

SangKeun Lee received his BSc., MSc. and Ph.D. degrees in Computer Science and Engineering from Korea University, Seoul, South Korea, in 1994, 1996 and 1999, respectively. From April 2000 to March 2001, he was a visiting postdoctoral fellow, supported by Japan Society for the Promotion of Science (JSPS), in Institute of Industrial Science, University of Tokyo, Japan. He is currently a research engineer in 3G Handsets Lab., LG Electronics Inc., South Korea. His research interests include mobile information access, data dissemination, Wireless Application Protocol (WAP), WWW applications and distributed heterogeneous database systems. He is a member of ACM and IEEE Computer Society.

Chong-Sun Hwang received his BS and MS degrees in mathematics from Korea University, Seoul, South Korea, in 1966 and 1970, respectively. He received his Ph.D. in computer science and statistics from University of Georgia in 1978. He was an assistant professor of Lander University in Greenwood, South Carolina, USA, and is currently a professor in Department of Computer Science and Engineering, Korea University, Seoul, South Korea. His research interests include mobile computing systems, parallel and distributed database systems and knowledge-based systems.