혼합 데이타 전송에서 효율적인 트랜잭션 처리
(Efficient Transaction Processing in Hybrid Data Delivery)

이 상 근 †
(SangKeun Lee)

요 약 무선 정보 서비스에서 푸쉬-기반 브로드캐스팅은 데이타 항목의 개수가 작은 경우 많은 수의 클라이언트에게 정보를 확산시키는 매우 효과적인 기술이다. 그렇지만, 데이타베이스 용량이 큰 경우에는 풀-기반의 (클라이언트에서 서버로의) 역채널을 푸쉬-기반의 브로드캐스트와 결합한 이른바 혼합 데이타 전송이 유리할 수 있다. 본 논문은 순수 푸쉬-기반 데이타 브로드캐스트 환경에서 제시되었던 기선언-기반 트랜잭션 처리 기법을 혼합 데이타 전송에 적용하고, 시뮬레이션을 통해 그 성능을 분석한다. 시뮬레이션 결과를 통해, 기선언-기반 트랜잭션 처리 기법이 순수 푸쉬 데이타 전송뿐만 아니라 혼합 데이타 전송에서도 우수한 성능을 나타냄을 알 수 있다.
키워드 : 이동 컴퓨팅, 무선 정보 서비스, 혼합 데이타 전송, 캐싱, 트랜잭션 처리

Abstract Push-based broadcasting in wireless information services is a very effective technique to disseminate information to a massive number of clients when the number of data items is small. When the database is large, however, it may be beneficial to integrate a pull-based (client-to-server) backchannel with the push-based broadcast approach, resulting in a hybrid data delivery. In this paper, we analyze the performance behavior of a predeclaration-based transaction processing, which was originally devised for a push-based data broadcast, in the hybrid data delivery through an extensive simulation. Our results show that the use of predeclaration-based transaction processing can provide significant performance improvement not only in a pure push data delivery, but also in a hybrid data delivery.
Key words : Mobile Computing, Wireless Information Services, Hybrid Data Delivery, Caching, Transaction Processing

1. Introduction

With the advent of third generation wireless infrastructure and the rapid growth of wireless communication technology such as Bluetooth and IEEE 802.11, mobile computing becomes possible. People with battery powered mobile devices can access various kinds of services at any time any place. However, existing wireless services are limited by the constraints of mobile environments such as narrow bandwidth, frequent disconnections, and limitations of the battery technology. Thus, mechanisms to efficiently transmit information from the server to a massive number of clients have received considerable attention [1-3].

In recent years, broadcasting has been shown to be an effective data dissemination technique for wireless networks in many studies [1,2]. Particularly, there were a lot of research efforts in *periodic push* model where the server repetitively disseminates information without explicit request. One interest in the model is to deal with the problem of designing a broadcast schedule such that the average latency is minimized. The approach is to determine the broadcast frequency of each data item in accordance with users access frequency of the data, and then to distribute the broadcast slots of each data item as uniformly as possible. An approach to reduce the latency to a desirable level for each user is to make use of local storage. Caching frequently accessed data

· 본 연구는 고려대학교 특별연구비에 의하여 수행되었음
† 종신회원 : 고려대학교 컴퓨터학과 교수
yalphy@korea.ac.kr
논문접수 : 2003년 9월 22일
심사완료 : 2004년 2월 17일

items at the client side is an effective technique to improve performance in mobile computing systems. With caching, the data access latency is reduced since some data access requests can be satisfied from the local cache, thereby obviating the need for data transmission over the scarce wireless links.

In a periodic push model, however, average waiting time per data operation highly depends on the length of a broadcast cycle and different access patterns among clients may deteriorate the access time considerably [1]. For example, the number of data items in the database is large, the broadcast cycle may be long. Hence, clients have to wait for a long time before getting the required data. In this case, the clients are preferably willing to send a data request to the server explicitly through uplink channel to obtain optimal response time and to improve overall throughput [4]. We call such a broadcast-based data delivery supporting uplink channel as a *hybrid data delivery*. Our main concern in this paper is, as is the case with our early work [5], to handle the problem of preserving the consistency of mobile read-only transactions in a hybrid data delivery.

In a push-based broadcast environment, providing consistent data values to transactions has been identified as one of main issues in designing mechanisms [6-8], and several approaches to consistent and current data access despite updates in wireless data broadcast have been proposed in the literature[9-15]. The validation protocols with dynamic adjustment of serialization order of transactions are proposed in [10, 11]. BUC (Broadcasted and Updated Cycles) control information for each item in wireless data broadcast is deployed in [9]. A control information matrix and a serialization graph testing are used for concurrency checking in [14] and [13] respectively. A simple invalidation method is presented in [12], where an invalidation report is broadcasted at pre-specified points (e.g. at the beginning of each broadcast cycle) during the broadcast. To increase the number of read-only transactions that are successfully processed despite updates at the server, multiversion schemes are employed in [12]. There, old versions of data items are temporarily retained in a broadcast so that the number of aborted transactions could be reduced. The major problem, amongst others, with all the previous works are that the commit probability and/or response time of a wireless read-only transaction suffers a great deal when the size of transaction is large or data items are highly updated. This is attributed to an inherent property of a wireless data broadcast that data can only be accessed *strictly sequential.*

In our previous work [16, 17], a predeclaration-based query optimization was explored for efficient processing of wireless read-only transactions in a push-based broadcast. It is observed that, in a push-based data delivery, predeclaration in transaction processing has a novel property that each read-only transaction can be processed successfully with a *bounded* worst-case response time. This is because, a client retrieves data items in the order they are broadcasted, rather than the order they are requested. Here, clients are just tuning in broadcast channel and waiting for the data of interests. As mentioned before, however, it is sometimes necessary for clients to send messages to the server and a hybrid data delivery can be a good alternative model to deal with new requirements. Therefore, in this paper, we modify a predeclaration-based transaction processing [16] in order to apply it to the hybrid data delivery environment, and in turn, study its performance behavior. The proposed methods are particularly intended for applications like the online auction application, where the size of the broadcasted data is relatively small, but the number of clients is very large. Extensive experiments are provided and used to evaluate our methodology. Compared to other schemes, our solution improves the performance significantly.

The remainder of this paper is organized as follows. Section 2 describes our mobile computing system model. Section 3 introduces the proposed transaction processing algorithms. Section 4 studies the performance of the proposed algorithms. The conclusion of the paper is in Section 5.

2. System Model

In this section, we briefly describe the model of our mobile computing system. The system consists of a data server and a number of mobile clients connected to the server through a low bandwidth wireless network. A server maintains the consistency of a database and reflects refreshment by update transactions. The correctness criterion in transaction processing adopted in this paper is *serializability* [18]. The server also plays a role of servicing clients information demands. For efficiency, data items in the database are divided into *Push_Data* and *Pull_Data*. The server determines that data items in Push_Data are considered to be accessed more frequently than those in Pull_Data and thus it disseminates only Push_Data periodically and repetitively. Data in Pull_Data are serviced by broadcasting in an on-demand mode. In case of a pure-push broadcast, all data items are contained in Push_Data.

Each data item in Push_Data appears once during one broadcast cycle (*uniform broadcast* [1]). We assume that the content of the broadcasted at each cycle is guaranteed to be consistent. That is, the values of data items 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. Consistent with the rule, the server broadcasts Pull_Data at the end of a broadcast cycle for collected requests from clients during the last cycle. Besides, some useful information such as the set of data identifiers updated during the last cycle is delivered as a form of an invalidation report (*IR*) at the beginning of each cycle (and before every item in Push_Data is broadcasted).

Mobile clients do their jobs by utilizing their mobile terminals. When a data operation of a transaction is submitted, a way of acquiring data value is determined according to the data type. If the data item is an element of Push_Data, clients just tune in broadcast channel; in this case, they are passive listeners who make no request and such repetition allows the broadcast medium to be perceived as a special memory space. This makes broadcasting an attractive solution for large scale data dissemination. However, its limitation is that it can be accessed only sequentially and clients have to wait for the data of interest to appear on the channel. A direct consequence is that access latency depends on the volume of Push_Data, which has to be fairly small. If the data is not scheduled to be transferred through the channel (i.e. it is an element of Pull_Data), this is a standard client-server architecture where the data requests are explicitly made to the server. The average data access time depends on the aggregate workloads as well as the network load, but not highly on the size of Pull_Data.

It is evident that with too little broadcasting, the volume of requests at the server increase beyond its capacity, making service practically impossible [19]. However, in this paper, we do not touch the issues related with scheduling what data to disseminate. We just assume that Push_Data and Pull_Data are already determined and the access frequencies are not changed.

3. Predeclaration-based Transaction Processing

Three predeclaration-based transaction processing methods in the work [16], *P* (Predeclaration), *PA* (Predeclaration with Autoprefetching) and PA^2 (PA/Asynchronous), need to be slightly modified here to work in the hybrid data delivery. The central idea is to employ predeclaration of readset in order to minimize the number of different broadcast cycles from which transactions read data. Two assumptions made in our proposed methods are listed below.

- Each data item in Push_Data, which is broadcasted periodically, has the following information as a minimum; < *a primary key, an offset to the start of the next broadcast, the value of data record* >. Here, an offset to the start of the next broadcast is necessary to guide a client to tune in at the beginning of the next broadcast at any point within a broadcast cycle, which will be further explained at Section 3.1 and 3.2.
- The information about the readset of a transaction is available at the beginning of transaction processing. We expect this can be easily done either by requiring a transaction to explicitly declare its readset or by using preprocessor on a

client, e.g. to identify all the items appearing on a transaction program before being submitted to the client system (note that additional reads may be included to the predeclared readset due to control statements such as IF-THEN-ELSE and SWITCH ones in a transaction program).

We now define the predeclared readset of a transaction *T*, denoted by *Pre_RS*, to be a set of data items that *T* reads *potentially*. For all methods, each client processes *T* in three phases: (1) *Preparation phase*: it gets *Pre_RS*, (2) *Acquisition phase*: it acquires data items belonging to *Pre_RS* from the periodic broadcast (for Push_Data), the server in an on-demand mode (for Pull_Data), or its local cache. During this phase, a client additionally maintains a set *Acquire(T)* of all data items that it has acquired so far, and (3) *Delivery phase*: it delivers data items to its transaction according to the order in which the transaction requires data.

3.1 Method *P*

Since the content of the broadcast at each cycle is guaranteed to be consistent, the execution of each read-only transaction is clearly serializable if a client can fetch all data items within a single broadcast cycle. Since, however, a transaction is expected to be started at some point within a broadcast cycle, its acquisition phase may therefore be across more than one broadcast cycle. To remedy this problem, in *P*, a client starts the acquisition phase synchronously, i.e. at the beginning of the next broadcast cycle. Since all data items for its transaction are already identified, the client is likely to complete the acquisition phase within a single broadcast cycle. Only when the pull-requested items are not served within the broadcast cycle, which is due to server saturation, the acquisition phase is restarted from scratch. More specifically, a client processes its transaction T_i according to Figure 1.

Theorem 1. *P generates serializable execution of read-only transactions if the server broadcasts only serializable data values in each broadcast cycle.*

Proof) It is straightforward from the fact that the data set read by each transaction is a subset of a single broadcast. □

```
1. On receiving Begin(T_i) {
        get Pre_RS(T_i)  by using preprocessor;
        send requests for item(s) belonging to Pull_Data;
        Acquire(T_i) = ∅;
        tune in at the beginning of the next broadcast cycle;
   }
2. While (Pre_RS(T_i) ≠ Acquire(T_i) ) {
        if (the current cycle ends) { /* beyond server capacity for pull requests
          Acquire(T_i) = ∅;
          restart this acquisition phase from scratch;
        } /* end of if
        for d_j in Pre_RS(T_i) {   /* for both push and pull data
           download d_j;
           put d_j into local storage;
           Acquire(T_i) ⇐ d_j;
        } /* end of for
   }
3. Deliver data items to T_i according to the order in which T_i requires,
   and then commit T_i.
```

Figure 1 Algorithm of method *P*

The main advantage of *P* is that it achieves a considerable reduction of transaction response time in an update-intensive environment without sacrificing serializability or currency of reads. In particular, each transaction is likely to be successfully committed within two broadcast cycles even in an extreme case where all data items in a database are updated during a broadcast cycle. This is because the acquisition phase can be completed within a broadcast cycle, if the server's queue is not congested. The disadvantage of *P* is that local storage is not fully utilized. In the following section, we devise two variants of *P* which utilize local storage as local caches.

3.2 Methods *PA* and PA^2

Clients can cache data items of interest locally to reduce access latency. Caching reduces the latency 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 times. In this section, clients use their available hard disks as local caches and caching technique is employed in the context of transaction processing. We therefore need to guarantee that transaction semantics should not be violated as a result of the creation and destruction of cached data based on the runtime demands of clients. In our work, transactional cache consistency can be easily maintained if a serializable broadcast is on the air in each broadcast cycle.

At the beginning of each broadcast cycle, a client tunes in and reads the invalidation report. For any data item d_i in its local cache, if indicated as updated one, the client marks d_i as "invalid" and gets d_i again from the current broadcast and puts it into local cache. Cache management in our scheme is therefore an invalidation combined with a form of autoprefetching [20]. Invalidated data items remain in cache to be autoprefetched later. In particular, at the next appearance of the invalidated data item in the broadcast, the client fetches its new value and replaces the old one.

There are two choices on when to start the acquisition phase. One is a synchronous approach where, as is the case with *P*, a client fetches data items from the beginning of the next broadcast cycle. We call this method *PA*. Similar to method *P*, when the pull-requested items are not served within the broadcast cycle, which is due to server saturation, the acquisition phase is restarted from scratch. More specifically, *PA* works according to Figure 2.

```
1. On receiving $Begin(T_i)$ {
        get $Pre\_RS(T_i)$ by using preprocessor;
        send requests for item(s) belonging to Pull__Data;
        $Acquire(T_i) = \emptyset$;
        tune in at the beginning of the next broadcast cycle;
  }
2. Fetch an invalidation report;
  For every item $d_i$ in local cache {
        if (indicated as updated one) {mark $d_i$ as "invalid"; }
  } /* end of for
  For every "valid" item $d_i$ in local cache {
        if ($d_i \in Pre\_RS(T_i)$) { $Acquire(T_i) \Leftarrow d_i$; }
  } /* end of for
  While ($Pre\_RS(T_i) \neq Acquire(T_i)$ ) {
        if (the current cycle ends) { /* beyond server capacity for pull requests
          $Acquire(T_i) = \emptyset$;
          restart this acquisition phase from scratch;
        } /* end of if
        for $d_j$ in $Pre\_RS(T_i) - Acquire(T_i)$ {   /* for both push and pull data
           download $d_j$;
           put $d_j$ into local storage;
           $Acquire(T_i) \Leftarrow d_j$;
        } /* end of for
  }
3. Deliver data items to $T_i$ according to the order in which $T_i$ requires,
  and then commit $T_i$.
```

Figure 2 Algorithm of method *PA*

Theorem 2. *PA generates serializable execution of read-only transactions if, in each broadcast cycle, the server broadcasts an invalidation report which is followed by serializable data values.*

Proof) It is straightforward from the fact that the data set read by each transaction is a subset of a single broadcast. □

Method *PA* remedies the problem with method *P* by utilizing a client storage as local cache, while still supporting transactions efficiently in an update-intensive database. Its synchronous approach, however, may incur unnecessary response time latency to short transactions in a rarely updated database. For example, if most of data items reside in local cache and all missed items can be retrieved from the current push-based broadcast, then a transaction would be completed within a single broadcast cycle in which it is initiated.

To get over the disadvantage of method *PA*, a client can take an asynchronous way, i.e. it fetches data items immediately without waiting for the next broadcast cycle. Unlike synchronous approaches, the acquisition phase may span across two different broadcasts in this case. This method is referred to as PA^2. Interestingly, in case the pull-requested items are not served within the latter broadcast cycle, the acquisition phase is restarted from scratch at the beginning of the next broadcast cycle, resulting in *PA* processing. The algorithm for method PA^2 is shown in Figure 3.

```
1. On receiving $Begin(T_i)$ {
        get $Pre\_RS(T_i)$ by using preprocessor;
        send requests for item(s) belonging to Pull__Data;
        $Acquire(T_i) = \emptyset$;
  }
2. For every "valid" item $d_i$ in local cache {
        if ($d_i \in Pre\_RS(T_i)$) { $Acquire(T_i) \Leftarrow d_i$; }
  } /* end of for
  While ($Pre\_RS(T_i) \neq Acquire(T_i)$ ) {
        if (the latter cycle ends) { /* beyond server capacity for pull requests
          $Acquire(T_i) = \emptyset$;
          restart this acquisition phase from scratch
                    at the beginning of the next broadcast cycle;
        } /* end of if
        for $d_j$ in $Pre\_RS(T_i) - Acquire(T_i)$ {   /* for both push and pull data
           download $d_j$;
           put $d_j$ into local storage;
           $Acquire(T_i) \Leftarrow d_j$;
           if (it is time to receive an invalidation report) {
                 tune in and fetch an invalidation report;
                 for every item $d_j$ in local cache {
                     mark $d_j$ as "invalid";
                     $Acquire(T_i) = Acquire(T_i) - (d_j)$
                 } /* end of for
           } /* end of if
        } /* end of for
  }
3. Deliver data items to $T_i$ according to the order in which $T_i$ requires,
  and then commit $T_i$.
```

Figure 3 Algorithm of method PA^2

Theorem 3. *PA^2 generates serializable execution of read-only transactions if, in each broadcast cycle,* *the server broadcasts an invalidation report which is followed by serializable data values.*

Proof) Let *broadcastcycle*$_i$ be the broadcast cycle in which some transaction T_l completes its acquisition phase and DS_i be the serializable database state that corresponds to the broadcast cycle *broadcastcycle*$_i$. We show that the values read by T_l correspond to the database state DS_i by using a contradiction. Let us assume that the value of data item d1 read by T_l differs from the value of d_l at DS_i. Then, an invalidation report should have been broadcasted at the beginning of *broadcastcycle*$_i$ and thus d_l should have been invalidated. □

4. Performance Analysis

4.1 Simulation Model

In this section, we aim at studying the performance behavior of our predeclaration-based transaction processing through an extensive simulation. Figure 4 shows the simulation model, where a single client is considered. This is partially because the performance of a single client read-only transaction is independent of the presence of other clients transactions in terms of conflicts. In a hybrid data delivery, however, the queue congestion at the serve side can occur due to heavy uplink requests from clients. In the experiment, however, we do not consider the congestion problem for the sake of simplicity (further, the server congestion problem is not specific to ours).

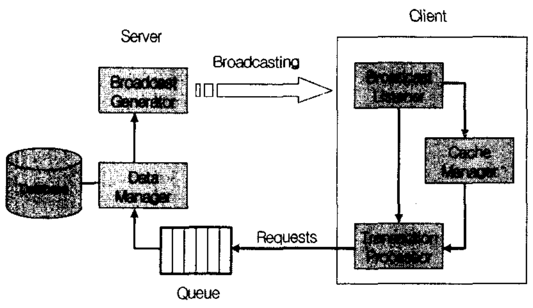


Figure 4 Simulation Model

The server first broadcasts an invalidation report which is followed by all data items in Push_Data, and then broadcasts requested data items among Pull_Data (of course, in case of pure-push broadcast model, all data items are contained in Push_Data). Broadcast model for Push_Data is *uniform*; that is, the server broadcasts each data item just once on a single wireless channel during one cycle. All data items in a cycle are in a consistent state. In the experiments, the access probabilities follow a *zipf* distribution with a parameter *theta* to model the non-uniform access; the first data is accessed the most frequently, and the last data is accessed the least frequently. *UpdateRate* is the number of updated data items during the time when all *NumberOfData* are broadcasted. The distribution of updates follows a *zipf* distribution with a parameter *theta* to model the non-uniform updates. There is a queue for storing uplink messages at the server. When a transaction needs a certain data in Pull_Data, the data request is delivered to the server and enqueued. The server serves those requests in a FIFO mode.

A mobile transaction issues *NumberOfOp* operations belonging to the range of *AccessRange*. In particular, predeclaration-based transactions are set to issue $\frac{3}{2} \times$*NumberOfOp* operations to account for additional reads due to control statements in the experiment. In the range, access probabilities also follow a *zipf* distribution. To model the disagreement between the access pattern of a transaction and the update pattern in the server, the first data item in *AccessRange* starts at $(Offset+1)^{th}$ data. Each client can maintain local cache which can hold up to *CacheSize* data items. The cache replacement policy is LRU in conjunction with auto-prefetching and the cache data consistency is maintained by monitoring an invalidation report, e.g. [2]. In the experiments, the time unit is set to the time which is needed for the server to disseminate one data item, 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.

For performance evaluation in a pure push data delivery, methods *P*, *PA*, *PA*^{2}, are compared with *Invalidation-Only (IO)* method, *Multiversion with Invalidation (MI)* method in [12, 13], and *O-Pre*

Table 1 Parameter Description

Parameter	Value	Meaning
NumberOfData	10,000	the number of data items
NumberOfOp	varied	the number of read operations in a transaction
Push_Data	2,000	the size of Push_Data
Pull_Bandwidth	1,000	the bandwidth allocated for data requests
UpdateRate	varied	the number of updated data items during a cycle
theta(θ)	0.90	*zipf* distribution parameter
CacheSize	200	local cache size
AccessRange	whole	average access range for mobile transaction
Offset	50	disagreement between access patterns
ReadTime	1	execution time for read operation time unit
IRCheckTime	3	the time for checking when receive an *IR*
msgTransferTime	50	the time needed to send a data request
RestartTime	10	the time between abort and restart

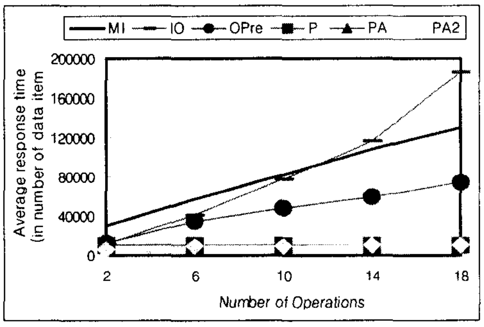
method in [10]. This is because these methods adopt serializability as a correctness criterion for transaction processing and the system model is very similar to ours. In a hybrid data delivery, ours and *O-Pre* [5] are compared in terms of response time. *IO* method is a pure optimistic algorithm, and a client does consistency checks based on a periodic invalidation report. Whenever any conflict is found, the transaction has to be aborted and restarts after *RestartTime*. With *MI* method, depending on individual update frequency, a single cycle consists of 1 to 4 version(s) per each data item. Thus, the length of one cycle in *MI* is much longer than those of *O-Pre, IO* and our methods.

4.2 Experimental Results

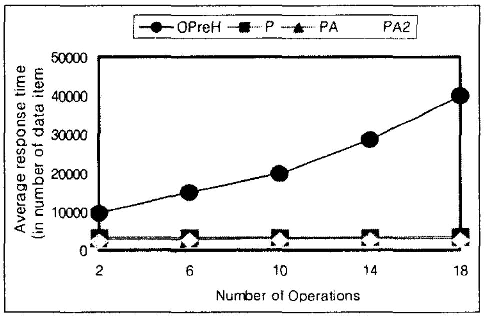
The characteristics of our proposed methods are explored quantitatively using the simulation model, and evaluated by showing improvement of average response time of transactions.

4.2.1 Effect of Number of Operations

Figure 5 shows the performance behavior as the number of operations issued by a transaction is increased in the pure push data delivery and the hybrid data delivery, respectively. We see that the performance behavior in Figure 5(a) is consistent with the analysis result in [16]. For long transactions (the number of issued operations are greater than 10 in our experiment), the response time of *IO* is increased rapidly. This is because a large value *NumberOfOp* decreases the probability of a transaction's commitment. As a result, a transaction



(a) Pure Push Data Delivery



(b) Hybrid Data Delivery

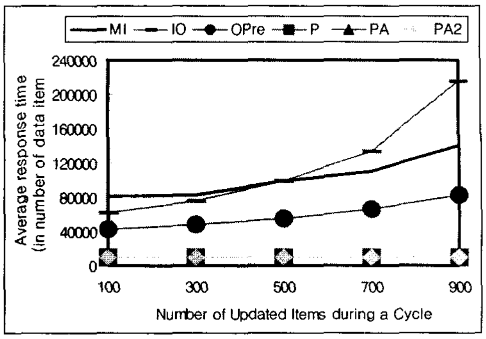
Figure 5 Effect of Number of Operations

suffers from many restarts until it commits. *MI* avoids this problem by making a client access old versions on each broadcast, thereby increasing the chance of a transaction's commitment. We can observe that the performance of *MI* is less sensitive to the number of items than *IO*. However, the increased size of broadcast affects the response time negatively. This explains why *MI* is inferior to *IO* for small operations. Method *O-Pre* shows fairly good performance, compared to *MI* and *IO*. With our *P*, *PA*, and *PA*^{2} methods, as a transaction can access data items in Push_Data in the order they are broadcasted, resulting in a stable performance. For example, when *NumberOfOp* is 14, ours yield the response time reduced by a factor of 10 on *MI* and *IO* methods. Among ours, *PA*^{2} exhibits only a marginal performance improvement over *PA*, which in turn shows a marginal improvement over *P*.

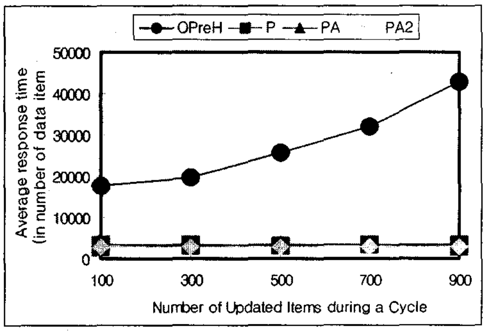
Turning to the hybrid data delivery, as expected, Figure 5(b) shows the superior performance to the pure push data delivery. For example, when *NumberOfOp* is set to 10, the average response time is reduced by a factor of 4 with the use of our methods. This result verifies the usefulness of a hybrid data delivery. With our *P*, *PA*, and *PA*^{2} methods, since a transaction can access data items in the order they are broadcasted, the average response time is almost independent of transaction size. Notably, we have observed that our methods performance in a hybrid data delivery is dominated by pulled data item(s), i.e. only a single pulled data is likely to make local cache, which favors frequently accessed data, useless in terms of transaction response time. Since items in Pull_Data is less frequently accessed, it is unlikely for a client to hold items in Pull_Data in local cache, thereby waiting a long time. This implies that *cost-based caching* may be beneficial in a hybrid data delivery, which is our future work. This explains why our three methods shows almost same response time; caching of frequently accessed data does not contribute to performance improvement. Compared to *O-PreH*, when *NumberOfOp* is 10, ours yield the response time reduced by a factor of 7 on *O-PreH*.

4.2.2 Effect of Update Rate

In this experiment, we consider the effect on the intensity of updates (*UpdateRate*) at the server. The parameter means the number of updated data items while the server disseminates all data items ([1...*NumberOfData*]). Therefore, in spite of the same value for *UpdateRate*, data items are less frequently updated in a hybrid data delivery since



(a) Pure Push Data Delivery



(b) Hybrid Data Delivery

Figure 6 Effect of Update Rate

the length of a cycle is very small.

Figure 6 shows the effect of update rate on the performance of various methods when *NumberOfOp* is set to 10. First, let us consider a pure push data delivery. Again, we see that the performance behavior in Figure 6(a) is consistent with the analysis result in [16]. A higher update rate means a higher conflict ratio, and also a higher probability of cache invalidation. This explains why the response time of *IO* deteriorates so rapidly. *MI* also degenerates as update rate increases. This is because a higher update rate leads to larger number of updated items in the database, resulting in a larger broadcast size. Unlike *IO*, however, with *MI*, a transaction can proceed and commit by reading appropriate old versions of items which are on the air. This difference of commitment probability is the main reason why *MI* beats *IO* for high update rate (in our experiment, when *UpdateRate* > 500). For a low update rate, there is a high probability that a transaction commit successfully even with *IO*. Thus, *IO* shows better response time than *MI* since the former retrieves each item more quickly than the latter. With *P*, *PA*, and PA^2, the response time is not affected by update rate significantly, and further, is almost identical. This is because, in most cases, a transaction can access data items within a single broadcast cycle, which contains consistent, serializable data values.

Next, consider the case in a hybrid data deliver. As expected, Figure 6(b) shows the superior performance to the pure push data delivery. Compared with our previous work, Figure 6(b) shows that ours are superior to *O-PreH* for all range of update rate. Since *O-PreH* is an optimistic algorithm, the performance gets deteriorated as the update rate is increased. Irrespective of update rate, since the performance in a hybrid data delivery is dominated by pulled data item(s), the use of local caching, which prioritizes frequently accessed data, is almost useless in terms of transaction response time. With *P*, *PA*, and PA^2, the response time is not affected by update rate significantly, since a transaction can access consistent, serializable data values within a single broadcast cycle in most cases.

5. Conclusion and Future Work

From the limited resource point of view, broadcast-based data delivery is especially suitable in mobile computing environments. If the number of data items to be broadcasted is considerable, it may result in poor utilization of valuable resources and low throughput.

In this paper, we have analyzed the performance of predeclaration-based transaction processing both in a pure push data delivery and in a hybrid data delivery. Mobile transactions are able to efficiently retrieve most data items in the order they are broadcasted, rather than in the order they are requested. Further, the role of maintaining transactional consistency is fully delivered to clients. In particular, both the introduced notion of predeclaration and the use of explicit data requests from clients improve the response time greatly.

Much future work remains. We plan to study the impact of server congestion due to heavy uplink requests in a hybrid data delivery. We also plan to analyze the optimal value for both the volume of Push_Data and the bandwidth of pulled data requests. Further, we would like to analyze the effectiveness of cost-based caching mechanism in a hybrid data delivery.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communication environments. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp.199-210, 1995.

[2] D. Barbara and T. Imielinski. Sleepers and workaholics: Caching in mobile environments. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp.1-12, 1994.

[3] T. Imielinski, S. Viswanathan, and B. Badrinath. Data on air: Organization and access. *IEEE Transactions on Knowledge and Data Engineering*, Vol.9, No.3, pp.353-372, 1997.

[4] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp.183-194, 1997.

[5] S. Kim, S. Lee, C.-S. Hwang, and S. Jung. O-preh: Optimistic Transaction processing algorithm based on pre-reordering in hybrid broadcast environments. In *Proceedings of the 10th International Conference on Information and Knowledge Management*, pp.553-555, 2001.

[6] D. Barbara. Mobile computing and databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, Vol.11, No.1, pp.108-117, 1999.

[7] T. Imielinski and R. Badrinath. Wireless mobile computing: Challenges in data management. *Communications of the ACM*, Vol.37, No.10, pp.18-28, 1994.

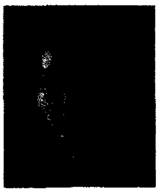
[8] K.-L. Tan and B. C. Ooi. *Data Dissemination in Wireless Computing Environments*. Kluwer Academic Publishers, 2000.

[9] A. Al-Mogren and M. H. Dunham. Buc, a simple yet efficient concurrency control technique for mobile data broadcast environment. In *Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, pp.564-569, 2001.

[10] S. Kim, S. Lee, and C.-S. Hwang. Using reordering technique for mobile transaction management in broadcast environments. *Data and Knowledge Engineering*, Vol.45, No.1, pp.79-100, 2003.

[11] E. Mok, H. V. Leong, and A. Si. Transaction

processing in an asymmetric mobile environment. In *Proceedings of the 1st International Conference on Mobile Data Access*, pp.71-81, 1999.
[12] E. Pitoura and P. Chrysanthis. Exploiting versions for handling updates in broadcast disks. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pp.114-125, 1999.
[13] E. Pitoura and P. Chrysanthis. Scalable processing of read-only transactions in broadcast push. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pp.432-439, 1999.
[14] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham. Efficient concurrency control for broadcast environments. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp.85-96, 1999.
[15] R. Srinivasa and S. H. Son:. Quasi-consistency and caching with broadcast disks. In *Proceedings of the 2nd International Conference on Mobile Data Management*, pp.133-144, 2001.
[16] S. Lee, C.-S. Hwang, and M. Kitsuregawa. Using predeclaration for efficient read-only transaction processing in wireless data broadcast. *IEEE Transactions on Knowledge and Data Engineering*, Vol.15, No.6, pp.1579-1583, 2003.
[17] S. Lee, M. Kitsuregawa, and C.-S. Hwang. Efficient processing of wireless read-only transactions in data broadcast. In *Proc. of the 12th International Workshop on Research Issues on Data Engineering*, pp.101-111, 2002.
[18] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Massachusetts, 1987.
[19] K. Stathatos, N. Roussopoulos, and J. Baras. Adaptive data broadcast in hybrid networks. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pp.326-335, 1997.
[20] S. Acharya, M. Franklin, and S. Zdonik. Disseminating updates on broadcast disks. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pp.354-365, 1996.



이 상 근
1994년 2월 고려대학교 전산과학과(현, 컴퓨터학과) 학사 졸업. 1996년 2월 고려대학교 대학원 전산과학과(현, 컴퓨터학과) 석사 졸업. 1999년 8월 고려대학교 대학원 전산과학과(현, 컴퓨터학과) 박사 졸업. 2000년 4월~2001년 3월 동경대학교 생산기술연구소 특별연구원. 2001년 4월~2003년 2월 LG전자정보통신 단말연구소 선임연구원. 2003년 3월~현재 고려대학교 컴퓨터학과 조교수