이동 컴퓨팅 환경에서 읽기-전용 트랜잭션을 지원하기 위한 비동기적 캐쉬 무효화 기법

김 일 도† · 남 성 헌††

요 약

이동 클라이언트/서버 데이터베이스 시스템에서 클라이언트 캐쉬의 상태 정보가 없이 비동기적 캐쉬 무효화 기법을 사용하여, 읽기-전용 트랜잭션을 지역에서 처리한다면 비동기적 무효화 리포트는 완료를 요청한 이동 트랜잭션의 대기시간에 대해 보장을 해주지 않는 문제가 발생한다. 이 문제를 해결하기 위해, 본 논문에서는 두 가지 종류의 메시지 사용을 제안한다. 트랜잭션의 처리 지연을 줄이기 위한 비동기적 무효화 메시지와 완료를 요청한 트랜잭션의 대기 시간을 보장해 주기 위한 안내 메시지이다. 비동기 무효화 리포트는 자신을 나타내기 위한 일련번호를 가지며, 안내 메시지는 가장 최근에 브로드캐스트 된 무효화 메시지의 일련번호를 가진다. 이동 클라이언트는 이 메시지들의 일련번호를 사용하여 자신의 캐쉬에 대한 유효여부를 점검함으로 이동 트랜잭션의 대기시간에 대해 보장해준다.

Asynchronous Cache Invalidation Strategy to Support Read-Only Transaction in Mobile Environments

Il-Do Kim† · Sung-Hun Nam††

ABSTRACT

In stateless server, if an asynchronous cache invalidation scheme attempts to support local processing of read-only transaction in mobile client /sever database systems, a critical problem may occur ; the asynchronous invalidation reports provide no guarantees of waiting time for mobile transactions requesting commit. To solve this problem, the server in our algorithm broadcasts two kind of messages, asynchronous invalidation report to reduce transaction latency and periodic ***guide message*** to avoid the uncertainty of waiting time for the next invalidation report. The asynchronous invalidation report has its own sequence number and the periodic ***guide message*** has the sequence number of the most recently broadcast asynchronous invalidation report. A mobile client checks its cache validity by using the sequence numbers of these messages.

키워드 : 이동 트랜잭션(Mobile Transaction), 비동기적 무효화 메시지(Asynchronous Invalidation Report), 안내 메시지(Guide Message), 캐쉬 일관성(Cache Consistency)

1. Introduction

Since the bandwidth of the wireless channels is very limited in a mobile computing environment, it is important to minimize mobile client's access to wireless channel, in order to reduce the contention on the narrow bandwidth. Local processing of read-only transactions and caching of frequently accessed data at mobile clients have been shown to be an useful techniques to minimize access to narrow channel. Local processing of read-only transactions means that mobile clients can commit or abort the read-only transactions without contacting server.

However, for caching to be effective, the cached data must be consistent with those data stored in the server. To our knowledge, most of cache invalidation schemes for mobile environments in the literature are stateless-based and adopt synchronous (periodic) manner by way of invalidation report broadcast. The primary concern of periodic invalidation report is to prevent loss of invalidation reports due to disconnection and mobility of mobile clients [2, 6, 7, 8].

An asynchronous Scheme (AS) [1] is presented as an alternative to the periodic invalidation reports approach. However, this scheme is stateful-based approach and broadcasts invalidation report whenever a data item is updated at the server. Although, it shows severa enhancements compared with periodic invalidation report approach, there are needed additional mechanisms to ensure cache consistency in error-prone mobile environment (especially, unrecognized loss of message of mobile clients caused by communication noises),

† 종신회원 : 해군사관학교 전산과학과 교수
†† 정 회 원 : 해군 해병대 제 6여단통신대장
논문접수 : 2002년 9월 17일, 심사완료 : 2003년 3월 13일

because it doesn't ask acknowledgement on broadcasting process. And, actually, the conventional recovery method by means of retransmissions and acknowledgements is not suitable for broadcast data in mobile environments [9].

To support local processing of read-only transaction, we have to ensure the consistency of client's read-only transaction when a data item is updated at the server. In [3], they presented an approach based on broadcast disk to support read-only transaction in mobile clients. In our previous work [4], we proposed an approach to support transaction semantics based on periodic invalidation report. However, all commit requests for locally completed transactions at mobile clients are passed to server.

In this paper, we assume stateless server to manage mobile clients caches, in the same way with [2] and consider additional disconnection state, such as loss of data caused by communication noises. Under these assumption and consideration, we suggest a local processing algorithm of read-only transaction based on asynchronous cache invalidation scheme. However, an asynchronous cache invalidation scheme to support local processing of read-only transaction may lead to a critical problem. That is, asynchronous invalidation no guarantees of waiting time for the next asynchronous invalidation report [2]. Thus, no guarantees can be given for the waiting time of transaction requesting commit. To solve this problem, the proposed algorithm adopts a new conceptual control message (***guide message***) to avoid the uncertainty of waiting time.

The main advantages of our strategy are that our algorithm achieves great improvements of transaction latency and it minimizes the number of aborted transactions compared with the algorithm that uses periodic invalidation report. The key design issues include, server broadcasting manner, mobile client's access to cached data and transactional consistency check on accessed data. With respect to the broadcasting manner, the server immediately broadcasts an invalidation report right after commit an update transaction at the server to reduce the transaction latency and to decrease the number of aborted transactions, and the asynchronous invalidation report has its own sequence number to prevent loss of message caused by communication noises. In addition to the asynchronous broadcasting, the server periodically broadcasts ***guide message*** that brings just the sequence number of the most recently broadcast invalidation report. This ***guide message*** will have very small fixed size. The sequence number of both messages is used to check cache validity of mobile clients. Through the cache validity check, each mobile client can ensure whether they are lost some broadcasting messages.

The remainder of the paper is organized as follows. Section 2 shows the design principles for supporting transactional cache consistency while retaining the broadcast- based cache invalidation scheme. Section 3 supports the serializable execution of read-only transaction on mobile client. Section 4 presents experiments and results, and finally we state our concluding remarks in section 5.

2. Basic Design Principles

In this section, we show the basic design principles of our approach to maintain cache consistency and to support read-only transaction based on asynchronous cache invalidation scheme and periodic ***guide message*** in mobile client/server database systems.

2.1 Server Broadcasting Manner

This is one of key aspects of our design issues in mobile client/server database systems. The server immediately broadcasts an invalidation report right after commit of an update transaction. And it is tagged with a sequence number. Each mobile client can check its own cache validity by using these sequence numbers. Whenever a mobile client receives an invalidation report, it checks if the sequence number of the invalidation report is sequential or not with that of the lastly received invalidation report. If so, it can be assumed that the cache of a mobile client is valid, then it executes invalidation process on its cache using the data identifiers in the invalidation report. If not, the mobile client has to refresh the cache. If a mobile client is on reconnection state, it waits for the next invalidation report and compares the sequence number of the lastly received invalidation report before disconnection and that of the first received invalidation report on reconnection state. If they are sequential, it can be assumed that the cache is valid, then executes the invalidation process on the cache.

However, the asynchronous cache invalidation scheme has its own shortcoming when it used to support local processing of read-only transaction by using the invalidation report. There are no guarantees when the invalidation report will be sent and hence no guarantees can be given for the waiting time. If the invalidation report is used to process locally read-only transaction, mobile clients sometimes have to wait the invalidation report that can not be expected when to

arrive. Hence, the server broadcast ***guide message*** to avoid uncertainty of waiting time. It is periodically broadcast by the server and has the sequence number of the most recently broadcast invalidation report. Whenever a mobile client receives ***guide message***, it checks the sequence number of the message. If the number is identical with the lastly received invalidation report, the mobile client's cache can be considered valid.

Then, we will explain how we use the asynchronous invalidation report and periodic ***guide message*** for transactional consistency check on mobile client in the following subsection.

2.2 A Mobile Client's Access to Cached Data and Consistency Check on Accessed Data

These are key aspects of our design issues in a mobile client/sever database system. In the periodic broadcasting approaches [2] to cache invalidation scheme, any read operation of cached data item in a broadcasting interval is deferred ; it is answered in the next broadcast interval even though the data item is in the mobile client cache. From the serializability point of view, this solution implies that each transaction issuing any read operation of cached data item in an interval $[ts_{i-1}, ts_i]$ should be serialized after all the committed transactions up to the timestamp ts_i. Although this solution provides a mobile client with the highest currency or freshness of data item with a bounded latency, i.e., a mobile client always reads the most recently committed data items, it notably loses the availability benefit of client caching and greatly raises transaction latency if the interval is long, especially if the data item at the local cache is valid. In our algorithm, in contrast to the previous deferred-access scheme, any access to cached data item is answered using the cached data immediately ; a mobile client does not wait until the next *IR* is arrived. The immediate-access scheme has the advantage of high availability of client caching. In the immediate-access scheme, although stale data (i.e., it is older than the data item's latest committed value) may be accessed by a mobile client, it will be detected by the mobile client through the consistency check on accessed data.

With respect to consistency check on accessed data, in our algorithm, a mobile client performs it on the basis of invalidation reports broadcast by the server. This scheme is based on the following property on invalidation reports.

Property on Invalidation Reports. A mobile client cache is transactionally consistent immediately after the mobile client processes an invalidation report. The above property is based on the assumption that the server is always keeping a consistent database state by allowing only serializable execution and keeping invalidation reports transactionally consistent. From the property on invalidation reports, we can derive the following **[Lemma 1]** upon which a mobile client's consistency check on accessed data of read-only transaction is based.

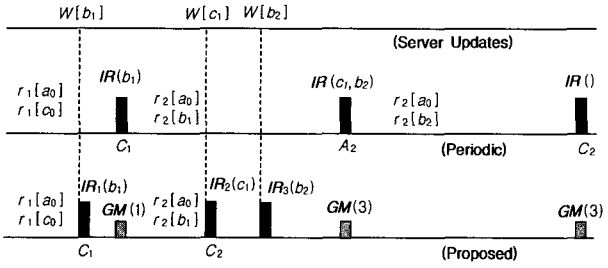
[Lemma 1] Suppose that a read-only transaction T_1 is executing on a mobile client. If no data item accessed by the transaction is invalidated after processing an invalidation report (*IR*) broadcast by the server at the time ts_i, $IR(ts_i)$, the execution involving T_1 is serializable up to ts_i.

[Proof] Let us assume that the execution involving T_1 is not serializable up to ts_i. This implies that T_1 has been involved in a cycle at serialization graph [10] at some point between its beginning and ts_i. Obviously, if no data item accessed by T_1 is invalidated after processing $IR(ts_i)$, there does not exist any edge from T_1 to another committed transaction T_2 due to $r_{T1}(x) - w\_{T2}(x)$ conflict at serialization graph. So, the serialization graph involving T_1 is acyclic and this is a contradiction to the fact there is a cycle at serialization graph.

A mobile client checks consistency on accessed data as an integral part of a cache invalidation process ; while a mobile client is being connected, it listens to every invalidation report broadcast by the server. At any time, if any data item read by the active transaction is invalidated and dropped from its cache, then the mobile client aborts the active transaction. Otherwise, it goes on the transaction and commits it if its commit request has been issued. However, in case of the ***guide message***, there is no invalidation process and transactional consistency check. Thus, if the mobile client ensures its cache consistency, just keeps the transaction processing and commits it if its commit request has been issued.

(Figure 1) explains the roles of invalidation report (*IR*), ***guide message*** (*GM*). And it shows how proposed algorithm improves the transaction latency and reduces the number of aborted transactions. *GM* has same meaning with empty *IR*. However, it has the sequence number of the most recently broadcast *IR* to help mobile client's cache validity check. In the periodic approach, although a_0 and c_0 is valid

data, T_1 is delayed to first *IR* and then committed. However, the proposed algorithm can commit T_1 earlier by asynchronous IR_1. T_2 is delayed until the second *IR* is arrived and aborted with periodic approach, and then the re-execution is committed at IR_3. However, the proposed algorithm can commit T_2 very early by asynchronous IR_2 without aborting. The second *GM* has the sequence number ③ that is the one of the third asynchronous invalidation report. The third *GM* has the same sequence number with the second *GM* because there is no *IR* between the second and third *GM*. When a mobile client is on reconnection (after disconnection) and receives third *GM*, its cache is valid if the sequence number of lastly received *IR* before the disconnection is ③ ; otherwise the cache may inconsistent with server database, then the mobile client has to refresh its cache.



(Figure 1) Transaction processing using *IR* and *GM*

3. Proposed Algorithm

We assume that update transactions are made and processed only at server and mobile clients only execute read-only transactions.

3.1 The Server's Algorithm

In basic design principles, the sever broadcast the *IR* that only contains updated data items of currently committed transaction. In frequent disconnection environments of wireless network, this way may hardly degrade the cache efficiency because mobile clients through out the entire cache whenever it lose some broadcasting *IR*. Thus, the *IR* is piggybacked with some extra information. The *IR* consists of data identifiers and their sequence numbers of data items have been updated during the last w seconds. The all updated data items in a committed transaction have same sequence number, and the sequence number of currently committed update transaction is the sequence number of *IR*.

To construct *IR* and *GM*, and to ensure the cache consistency of each mobile client, the server keeps the following information.

$[n]$ = { n is the sequence number of invalidation report that the most recently broadcast to mobile clients}

$[U(T_i)]$ = { $[j, v_j]$ j is a data item that will be updating by committing transaction T_i at server, and v_j is a new version that will be attached to j}

$[List(IR_n)]$ = { $[n, j, t_{IRi}]$ n is sequence number of the *IR*, j is a data item that broadcast by IR_n, and t_{IRn} is a time stamp of the IR_n}

With the above information, the server makes a *IR* whenever commit an update transaction. The *IR* consists of a representative sequence number n+1 and the identifiers js of the data items updated by transaction T_i, and the extra information piggybacked to original IR_{n+1}. The extra information gets from *List* (IR_n), and consists of the identifiers of data items and their sequence numbers having larger t_{IRn} than $ct - w$ (ct is current time). The resulting algorithm is as follows :

① When commit a T_i {
if (it is update transaction) broadcast IR_{n+1} ; }
② If it is time to broadcast *GM* {broadcast *GM*[n] ; }
③ If it is asked for $r_{T(j)}$ by a mobile client {
send the message containing (j's value, v_j) ; }

In server algorithm, we assume the transmitted data item always consistent with database and the server's transmitting order of messages and mobile client's receiving order is same.

3.2 A Mobile Client's Algorithm

The *IR*'s piggybacking mechanism gracefully keeps the cache efficiency under frequent disconnection mobile environments, But, still we have to consider another problem. A mobile client lost a invalidation report, then it received a periodic *GM* and invalidated the entire cache. Right after the entire cache invalidation, if the mobile client receives *IR* and it has enough information to maintain consistency, it also degrades the cache efficiency. Thus, the mobile client intelligently executes the cache invalidation process. Although the sequence number of periodic *GM* is not identical with that of the lastly received *IR*, the mobile client skips the cache invalidation process if the time interval (nt) between the arrival times of lastly received invalidation report and currently received periodic *GM* is smaller than a half of w, and if the mean arrival time (mt) of *IR* is not over

a half of w. It means the largest sum of the unrecognized or recognized disconnection period and the waiting time for the next IR is not over a w. If the waiting time is over a half of w, the mobile client invalidates the entire cache. Of course, it is the worst case.

In this subsection, a mobile client's algorithm for achieving serializability of transactions is shown. Each mobile client keeps m and *ReadSet* defined as follows :

[m] = { m is a sequence number of the most recently receivedIR }

[*ReadSet*] = { [j] j is a data item that has been read by a active transaction }

A mobile client answers a read operation $r_{T(j)}$ either by answering the cached data immediately if a requested data item is in its cache or by going uplink to the server with the operation if a requested data item is not in its cache. On receiving a commit request $commit_T$ for a transaction T, a mobile client wait next broadcasting message (IR or GM). On reconnection state(after disconnection), a mobile client wait next broadcasting message (IR or GM). On receiving IR_n, a mobile client checks the sequence number n of IR is sequential or not with the m of the lastly received IR before disconnection. If so, the mobile client executes invalidation process on its cache. If not, it compares the m+1 and the smallest $n-i$ of IR, if m+1 is equal or larger than $n-i$, it executes invalidation process on its cache by using the data items in IR that have larger sequence numbers than m. If m+1 smaller than $n-i$, it invalidates the entire cache. After invalidation process, if there is active transaction, the mobile client checks the transactional consistency on accessed data item. If any data item read by the active transaction is invalidated and dropped from its cache, then the mobile client aborts the active transaction. Otherwise, it goes on the transaction and commits it if its commit request has been issued. On receiving periodic GM, there is no invalidation process and transactional consistency check. Thus, if the sequence number of GM is same with that of the lastly received IR, the mobile client just keeps the transaction processing and commits it if its commit request has been issued. Otherwise, it checks if the nt and the mt is smaller than a half of w. If so, it waits the next IR ; otherwise, it invalidates the entire cache and abort active transaction. The algorithmic description of mobile client's activity is as follows :

```
1. On receiving r_T(j) {
    if (j is in the cache) {
      answer by using the value in its cache ; }
    else {
      load j into its cache from server
      and answer by using the value ; }}
2. On receiving commit_T for T {
    wait next broadcasting message (IR or GM) ; }
3. On reconnection state (after disconnection) {
    wait next broadcasting message (IR or GM) ; }
4. On receiving the invalidation report IR_n {
    if (n > m+1 and m+1 < the smallest n-i of IR)
      { invalidate the entire cache ; }
    else {
      for ( every item j in its cache ) {
        if ( there is a data item j in IR ) {invalidate j ; }}
          if ( any data item was invalidated at its cache ){
            if ( there is an active T such that its ReadSet
              contains any dropped data item ) { abort T ; }
            else if (T's commit request has been issued)
              {commit T ; }
    }}}
5. On receiving guide message GM(n) {
    if (n > m) {
      if (nt and mt < a half of w) {wait next IR ; }
      else {
        { invalidate the entire cache ; }
        if ( there is active T){abort T ; }
        }}
        else if ( there is active T and it's commit request has been
              issued )
         { commit T ; }
}
```

With respect to disconnection, a mobile client aborts its transaction at following two cases ; First, when the $m+1 <$ the smallest $n-i$ of the currently received IR. Second, when the $m < n$ of the currently received GM and it can not get IR in a half of w after receiving the GM. Namely, although a transaction is processed during disconnection, the mobile client can commit the transaction on reconnection state regardless of the term of disconnection if the mobile client has received all IR broadcast by server.

The following **[Theorem 1]** indicates that proposed algorithm based on asynchronous IR and periodic GM generates serializable execution of read-only transactions at mobile clients.

[Theorem 1] Proposed algorithm generates serializable execution of read-only transactions.

[Proof] Since each read-only transaction that is intended to commit should access the most recently committed values of data items in proposed algorithm, each mobile transaction that will be committed cannot have any outgoing edges at serialization graph at commit point. This fact is obvious from **[Lemma 1]**. It means that serialization graph containing committed read-only transaction of mobile clients and

update transaction of server is acyclic, so the execution is serializable.

3.3 Additional Algorithm to Maintain Cache Efficiency

Here we basically throw out the entire cache when the mobile client can not ensure its cache validity from cache validity check. Our method is similar with that of [2]. However, throwing out the entire cache after a long desconnection period essentially takes away the benefits of caching, especially if the update rate is low and most of the cached objects are still valid. To improve cache efficiency, some papers prefer that mobile client sends a query message to server to get a list of data items that updated during its disconnection period [1, 5, 7].

Thus, in proposed algorithm, instead of throwing out the entire cache, mobile clients simply send a *cache refresh request* (*CR*) to refresh its cache if the number of lost *IR* is not over a certain number ; the number of lost *IR* gets from comparison of the sequence numbers of the lastly received *IR* before disconnection and currently received broadcasting message (*IR* or *GM*). And the *CR* brings the sequence number of lastly received *IR*. To service *CR* of mobile clients, server maintains the log of the sequence numbers of *IR*s and identifiers of updated data items for certain period. When sever receives the *CR* from mobile clients, it answers with a list of data identifiers that having larger sequence number than that in the mobile client's *CR*. When a mobile client receives the answer from server, it executes invalidation process on its cache by using this list. Then, it checks consistency on accessed data of active transaction.

4. Performance

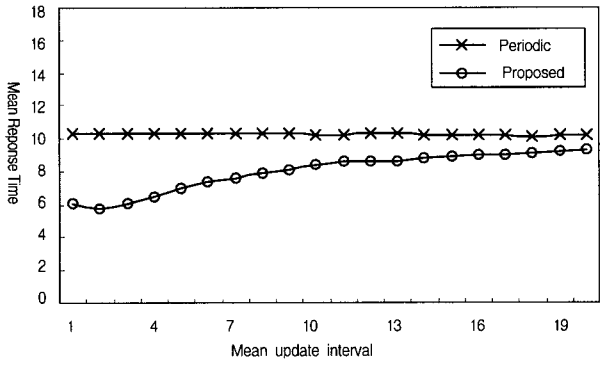
In this section, we describe the simulation model and present the results of experiments. In order to evaluate the performance of the proposed algorithm, we compare the proposed algorithm to the algorithms in the literature. However, in case of the AS [1], although it adopts asynchronous cache invalidation scheme, it can not be compared with the proposed algorithm since it adopts stateful server and does not work under unrecognized disconnection. Thus, we compare the proposed algorithm to the algorithm based on synchronous cache invalidation scheme in terms of the number of uplink messages and aborted transactions, and mean response time. The synchronous cache invalidation scheme is similar with TS of [2] that the server broadcast periodically the invalidation report. This algorithm also adopts immediate access scheme and checks the transactional consistency on accessed data as an integral part of a cache invalidation process in the same way with our proposed algorithm. In this simulation, we call this algorithm as periodic algorithm to make easy the comparison with the proposed algorithm. The basic scenario corresponds to the parameters setting in <Table 1>.

〈Table 1〉 Simulation Parameter Settings

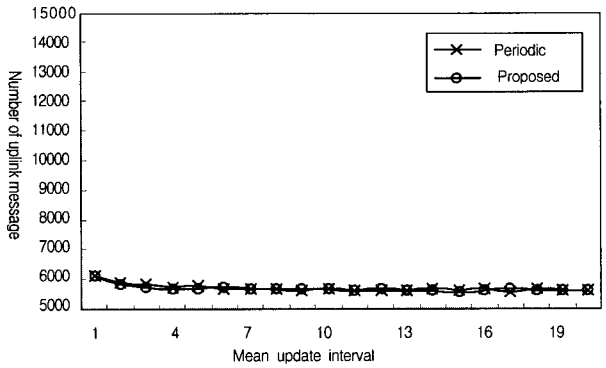
Parameter	Value
DababaseSize	1000 data items
DataItemSize	1200 bytes
DownlinkBW	1000 kbps
UplinkBW	10 kbps
TrSize	10 operations
ClientNum	1~100 clients
CacheSize	300 data items
TrNum	100 transactions of each client
Period	10 seconds
broadcastingWindowSize	1, 3 period
MeanUpdateInterval	1~20 seconds
ReadRequest	64 bytes
GuideMessage	64 bytes
CacheRefreshRequest	64 bytes
HotRegion	300 data items of database
ColdRegion	The rest of database except *HotRegion*
HotAccess	80%
ColdAccess	20%
HotUpdateAccess	70%
ColdUpdateAccess	30%
DisconnectionRate	0~90%

The server database consists of 1000 data items, and each data item is 1200bytes in size. The bandwidth of downlink is 1000kbps, and uplink is 10kbps. A transaction is consists of 10 operations. The number of clients is changed 1 to 100. And each client has cache of 300 data items and executes 100 transactions. The period of the periodic *IR* and the *GM* of the proposed approach is 10seconds. The broadcasting window size of invalidation report is set to 1 or 3 period. We examine the number of aborts and uplink messages, and the mean response time, where change the mean update interval 1 to 20 seconds at server. The read request message, the *GM* and the *CR* are 64 bytes in size. The size of asynchronous *IR* is $(NIR * bS) + (log(N) * k)$ bits where the *NIR* is the number of sequence numbers in the report, the *bS* is the bits of each sequence number, the $log(N)$ is the bits of each name of data item (assuming that N is the maximum number of data items in database and that each name is coded by $log(N)$ binary bits), and k is the number

of data items in the report. Mobile clients have common 300 data items (cache) hot region of the database to which 80% of its accesses are directed, while the remaining accesses are directed to the rest of the database. Server has same hot region of database with mobile clients to which 70% of its update accesses are directed, while the remaining update accesses are directed to the rest of the database. Disconnection period of each mobile client is decided by *DisconnectionRate*. The basic disconnection period is 10 seconds. Right after a mobile client commit a transaction, it is decided by using *DisconnectionRate* that the mobile client to sleep or not during 10seconds from current time. If it is decided to sleep, the decision process is repeated on next period and it is repeated till the mobile client is decided to be awaken during some period. The total time to be decidedto sleep is disconnection period of the mobile client. For example, right after a mobile client commit a transaction, if the 1st, 2nd and 3rd period are decided to sleep and 4th period is decided to be awaken, the total disconnection time is 30seconds. Thus, if the higher *DisconnectionRate* is given, the more frequent and longer disconnection period is made.



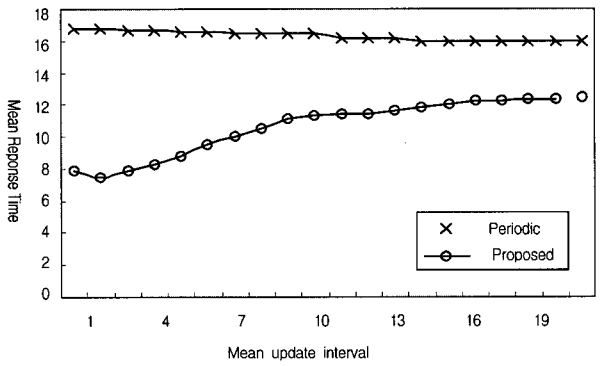
(a) Mean response time



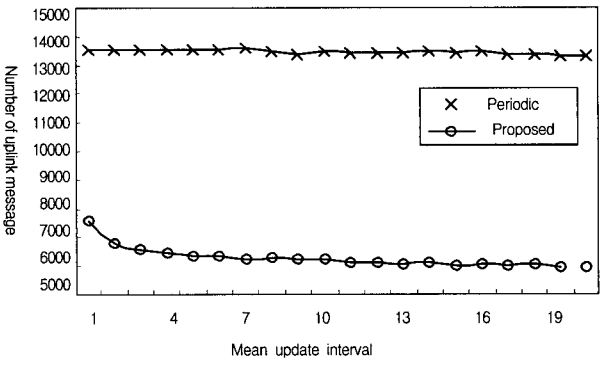
(b) Number of uplink messages

(Figure 2) Respones time/Uplink messages (*ClientNum* : 20, *DisconnectionRate* : 0%, *BroadcastingWindowSize* : 1)

(Figure 2). shows the influence of early commits on response time and the number of messages transmitted on uplink from mobile clients to server, where *BroadcastingWindowSize* is set to 1 period and *DisconnectionRate* is set to 0. In spite of the little difference of the number of uplink messages between the both algorithms, the periodic algorithm keeps higher response time all over *MeanUpdateInterval* and there is almost no change on response time. This is because, the main factor to decide response time is the $Period$. Any transaction requesting commit always has to wait next periodic invalidation report. On the contrary, the proposed algorithm shows much better response time when the *MeanUpdateInterval* is small, and the difference of response time between the two algorithms gradually gets closer as the *MeanUpdateInterval* increases. This is because, the proposed algorithm gets much more early commit chances when the *MeanUpdateInterval* is small, however, these chances gradually decrease as the *MeanUpdateInterval* gets larger. If there are no updates, the response time of the both algorithms will be equal. This is because, a commit request transaction of the proposed algorithm is only committed by *GM* that has the same $Period$ with the periodic *IR*.



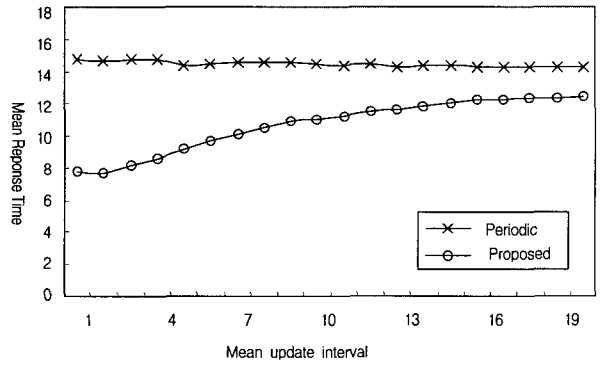
(a) Mean response time



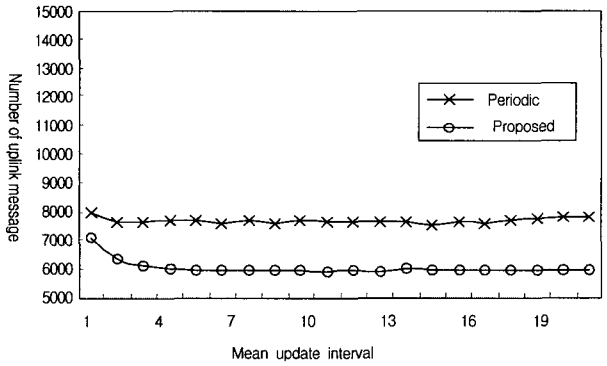
(b) Number of uplink messages

(Figure 3) Response time/Uplink messages (*ClientNum* : 20, *Disconnection* : 30%, *BroadcastingWindowSize* : 1)

(Figure 3) shows the influence of *DisconnectionRate* on the response time and on the number of uplink messages. Here we raise the *DisconnectionRate* to 30%, but other parameters have same values with those of (Figure 2). In periodic algorithm, the number of uplink messages dramatically increases compared with that of (Figure 2). And, as the result of it, the mean response time also increase. It shows the periodic algorithm is highly sensitive to *DisconnectionRate*. This is because, the mobile clients throw out entire cache when the disconnection period is over the *BroadcastingWindowSize*. It causes a lot of read requests from mobile clients that executing transaction. However, the proposed algorithm shows lower sensitivity to *DisconnectionRate* compared with the periodic algorithm.



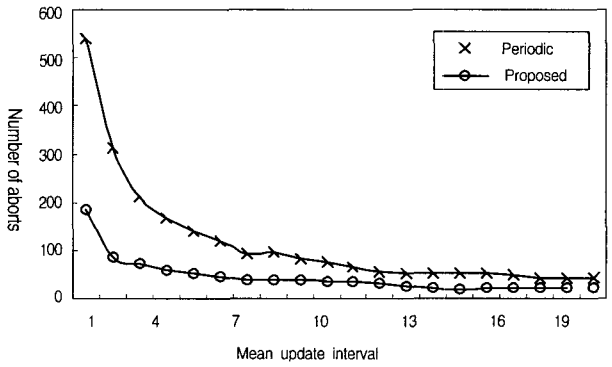
(a) Mean response time



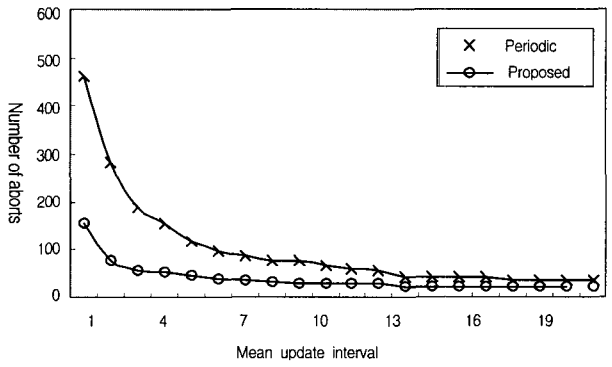
(b) Number of uplink messages

(Figure 4) Response time/Uplink messages (*ClientNum* : 20, Disconnection : 30%, BroadcastingWindowSize : 3)

(Figure 4) shows the influence of *Broadcasting WindowSize* on the response time and on the number of uplink messages under high *DisconnectionRate*. Here we raise the *BroadcastingWindowSize* to 3 periods, but other parameters have same values with those of (Figure 3). In periodic algorithm, the larger *BroadcastingWindowSize* fairly reduces the read requests of mobile clients compared with that of (Figure 3) and it also reduces the response time. However, it steel keeps higher response time and larger number of uplink messages than proposed algorithm. And proposed algorithm shows comparatively lower sensitivity to *BroadcastingWindowSize* than the periodic algorithm. It means that the number of *CacheRefreshRequest* on reconnection state of mobile clients has little effect on the number of uplink messages. Actually, in the simulation results, the highest number of *CacheRefreshRequest* was 606 under the environment of (Figure 3) and Here was 312. these are very small portion of the total number of uplink messages.



(a) Broadcsating windowsize : 1

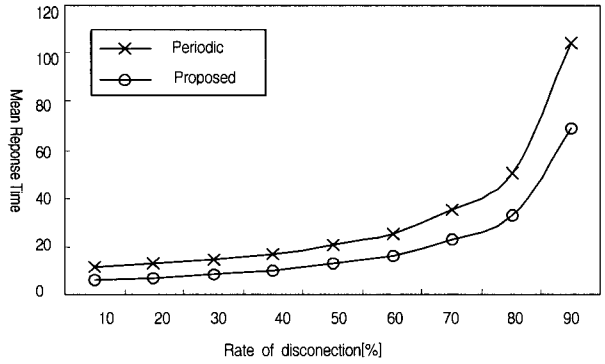


(b) Broadcsating windowsize : 3

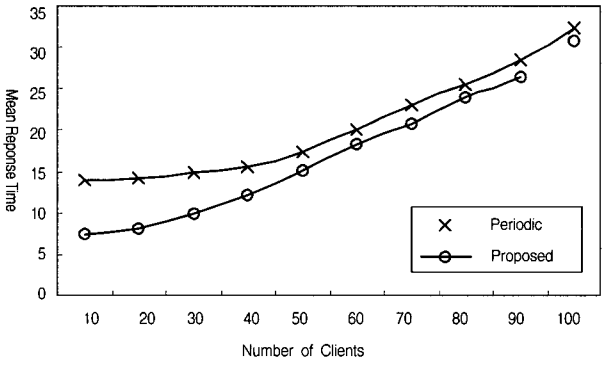
(Figure 5) The number of aborts (*ClientNum* : *20*, *Disconnection* : *30%*)

(Figure 5) shows each number of aborted transactions in both environments of (Figure 3), (Figure 4). the number of aborts of both algorithms decrease as the *MeanUpdateInterval* increases and our algorithm keeps the lower number of aborts all over *MeanUpdateInterval*. Especially, the difference between both algorithms gets bigger as the *Mean Update Interval* gets smaller. The most difference occurs when the *MeanUpdateInterval* is 1 second. As already explained in (Figure 1), the deferred broadcasting of periodic algorithm is main reason for the high abort rate : The more updates are occurred within a period, the more transactions

are aborted with the periodic algorithm. On the contrary, the higher update frequency in the proposed algorithm makes more early commit chances than the periodic algorithm. Thus, the difference is much larger when the *Mean Up date-Interval* is smaller than the *Period.*



(a) DisconnectionRate : 10~90%



(b) ClietNum : 10~100

(Figure 6) *Response time ((a) clientNum : 20, (b) DisconnectionRate : 30%, (a)(b) BroadcastingWindowSize : 3)*

(Figure 6) shows each response time in both case that the *DisconnectionRate* is changing from 10 to 90 and the *ClientNum* is changing from 10 to 100. In any case, the proposed algorithm shows better response time. However, when the number of clients is over certain level, the response time of the both algorithms gets closer. This is because, the concurrent transaction execution of multiple clients causes a lot of cache requests and it makes worse the wireless network bottleneck. And the network bottleneck mostly occurs on uplink that has relatively narrower bandwidth than downlink. The network bottleneck of uplink delays the arrival time of cache requests at server, and the delay increases the transaction processing time. If the network bottleneck is over a certain degree, the main factor to decide the response time of transaction is the message delay of uplink. Thus, the response time of both algorithms gets closer.

From the above analysis and (Figure 1), we can come to a conclusion as following : If there occurs any update on server, a transaction of mobile client is in one of three states in relation to the updated data item. First, the transaction has same data item with the updated data item. Second, the transaction doesn't have same data item and it is an active transaction. Third, the transaction doesn't have same data item and its commit request has been issued. In first state, our algorithm can early abort the transaction. In second state, the updated data item doesn't have any influence on the transaction processing in the both algorithms. In third state, our algorithm can early commit the transaction. Thus, our algorithm having advantage in the first and third state can get better response time than the periodic algorithm. However, if there exists no update on the server, the response time of the both algorithms will be equal, because the transactions in our algorithm are also committed only by *GM* that has same *Period* with the periodic *IR.*

5. Conclusions

In this paper, we have presented a new algorithm to support local processing of read-only transaction based on asynchronous invalidation report and periodic *guide message* in mobile client/server database systems. Our algorithm adopted the asynchronous invalidation report to reduce transaction latency and to decrease the number of aborted transactions. And, it uses periodic *guide message* to avoid the uncertainty of waiting time which is unavoidable with the asynchronous invalidation report. To ensure mobile clients cache consistency, each broadcasting invalidation report has its own sequence number and the periodic *guide message* has the sequence number of the most recently broadcast invalidation report. To reduce cache requests, when a mobile client's disconnection period is over the *BroadcastingWindowSize*, instead of throwing out entire cache, the mobile client simply sends *Cache RefreshRequest* that brings the sequence number of the lastly received invalidation report. And the server answers with the list of data items that updated during the mobile client's disconnection period. The main contribution of our algorithm is to allow the local processing of read-only transaction based on asynchronous cache invalidation scheme by using the periodic *guide message.* And the simulation results present the much smaller number of read requests and aborted transactions, and it shows the much better response time.

References

[1] A. Kahol, S. Khurana, S. Gupta and P. Srimani, "An Efficient Cache Maintenance Scheme for Mobile Environment," Proc. of International conference on Distributed Computing Systems, April, 2000.

[2] D. Barbara and T. Imielinsky, "Sleepers and Workaholics : Caching strategies in Mobile Environments," Proc. of ACM SIGMOD International Conference on Management of data, 1994.

[3] E. Pitoura and P. K. Chrysanthis, "Scalable Processing of Read-Only Transactions in Broadcast Push," Proc. of international Conference on Distributed Computing Systems, 1999.

[4] SangKeun Lee, Chong-Sun Hwang, HeonChang Yu, "Supporting Transactional Cache Consistency in Mobile Database Systems," Proc. of the international Conference on Data Engineering, 1996.

[5] K. K. Wu, P. S. Yu and M. S. Chen, "Energy-efficient Caching for Wireless Mobile Computing," Proc. of the International Conference on Data Engineering, 1996.

[6] J. Jing, A. Elmagarmid, A. Helal and R. Alonso, "Bit sequences : an adaptive cache invalidation method in mobile client/sever environments," Proc. of Mobile Net works and Applications, 1997.

[7] Q. Hu and D. K. LEE, "Cache algorithms based on adaptive invalidation reports for mobile environments," Proc. of Cluster Computing, 1998.

[8] G. Y. Liu and G. Q. McGuire Jr., "A mobility-aware dynamic database caching scheme for wrieless mobile computing and communications," Proc. of Distributed and parallel Databases, 1996.

[9] Shou-Chin Lo, Arbee L. P. Chen, "An Adaptive Access Method for Broadcast Data under an Error-Prone Mobile Environment," IEEE Transaction on Knowledge and Data Engineering, July/August, 2000.

[10] P. A. Berstein, V. Hadzilacos and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley.

김 일 도

e-mail : dotcom1126@naver.com

1981년 해군사관학교 전자공학과(공학사)

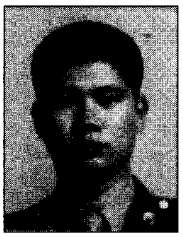
1987년 고려대학교 수학과(이학석사)

1992년 고려대학교 전산과학과(이학박사)

1988년 해군사관학교 전산과학과 조교수

1993년~현재 해군사관학교 전산과학과 부교수

관심분야 : 트랜잭션 처리, 데이터 모델링



남 성 헌

e-mail : shmam@disys.korea.ac.kr

1988년 해군사관학교 조선공학과(공학사)

1997년 고려대학교 전산과학과(이학석사)

2002년 고려대학교 컴퓨터학과(이학박사)

2002년~현재 해군 해병대 제 6여단통신대장

관심분야 : 캐쉬 일관성 유지, 이동 컴퓨팅, 분산 운영체제