Hybrid Cache Invalidation Schemes in Mobile Environments

Yuliang Bao Reda Alhajj Ken Barker
ADSA Lab & Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{bao, alhajj, barker}@cpsc.ucalgary.ca

Abstract

To make cached data consistent, the server periodically broadcasts an invalidation report to all mobile clients in its cell so that each can invalidate obsolete data items from its local cache. In this paper, we present two hybrid cache invalidation schemes: Hybrid cache invalidation with Simple Broadcasting (HSB), and Hybrid cache invalidation with Attribute Bit Sequence Broadcasting (HABSB). In these schemes the server is stateful as it stores the caching information of each mobile client in its cell to allow for long disconnections. However, the server periodically broadcasts an invalidation report similar to a stateless server. In HABSB, attribute bit sequences are added in each broadcasted invalidation report to maximize the cache-hit ratio and to reduce the data transmission of queried data items. The simulation results show that HSB and HABSB can significantly reduce the bandwidth requirement, and HABSB is very efficient in improving mobile caching.

Index Terms: cache consistency, cache invalidation report, mobile computing, wireless communication.

1. Introduction

A mobile computing environment has are two sets of entities: mobile clients and fixed hosts. Each fixed host, interchangeably called server or base station (BS) is equipped with a wireless interface to communicate with mobile clients through a wireless channel within a radio range called *a cell*. Each mobile client within a cell can roam in the cell or transfer through multiple cells. It can also connect to a BS through the wireless network in the current cell, or disconnect from the BS to save energy. We assume that the database is completely replicated in all servers and a mobile client can cache part of the database. Data is only updated in the servers by some transactions and every mobile client can only read some data items in the database by querying them.

Caching of frequently accessed data at the mobile client side is an effective technique to reduce interactions between the client and server, and hence improve data access performance and availability. However, a cache invalidation strategy is required to keep cached data in mobile clients' local memory consistent with those stored in the database server at the BS. When long and frequent disconnection occurs, it is difficult to enforce data consistency between mobile clients and the server.

There are two ways of managing the cache: having a stateful server, e.g., [13] or a stateless server, e.g., [1, 2]. A stateful server maintains the states of the mobile clients' caches, i.e., the server knows the data stored in each mobile client. Once a data item is changed on the server database and is cached by a mobile client, the conventional stateful server has to send an invalidation message to the client immediately. Hence, if a mobile client disconnects from the network, say to save battery power, it discards its whole cache since the client cannot communicate with the server and its cache is no longer valid. Moreover, mobile clients must inform the server when they enter or leave the cell.

A stateless server has no information about which mobile clients are currently in its cell and what contents are stored in their caches. Most research on stateless servers, e.g., [6, 7, 8, 12, 16] concentrate on the stateless server case because it has good scalability and communication efficiency. Data consistency is ensured by the server broadcasting an invalidation report (IR) to all mobile clients in its cell periodically. The IR contains all the identifiers (IDs) of updated data items and their update timestamps during a period of time. Rather than querying the server directly about the validity of cached copies, a client can monitor these invalidation reports over the wireless channel and use them to validate their local cache. To answer a query, the client must use the next IR and decide whether its cache is still valid. If there is a valid cached copy of the requested data item, the client returns it immediately. Otherwise, it sends a query request through the uplink channel. The IR-based solution is attractive because it reduces the traffic between the clients and server in the wireless channel,

0-7803-8577-2/04/$20.00 ©2004 IEEE

and scales to any number of clients monitoring IR. In general, a large IR can provide more information and is more effective for cache invalidation, but it consumes a large amount of broadcast bandwidth. Thus, the clients may need to spend more power listening to the IR as they cannot switch to power saving mode while listening.

The cell controlled by a server is scalable in terms of the number of mobile clients and the server does not have to inform each mobile client in its cell every time data changes individually (efficient). However, the stateless server does not have information about mobile clients in its cell; so more uplink and/or downlink (from the server to mobile clients) communication on the wireless channel is needed to enforce cache consistency.

In this paper, we propose two hybrid cache invalidation schemes: Hybrid cache invalidation with Simple Broadcasting (HSB), and Hybrid cache invalidation with Attribute Bit Sequence Broadcasting (HABSB). Both can minimize uplink communication, and hence optimize the overall performance of the system. In these schemes, the stateful server is employed to record the cached information of all mobile clients in the cell. When the server periodically (every L seconds) broadcasts the IR, which contains the update information of data items in the database within a period of time (say $w \times L$ seconds, where w is the number of IR intervals), it acts as a stateless server. The connecting mobile clients in the cell check the IR to invalidate their local caches. When a disconnected mobile client reconnects with a short disconnection time (less than $w \times L$ seconds), it can still use the newest IR to invalidate its local cache because this IR contains all the update information the client needs. However, if a mobile client disconnects for a very long time (longer than $w \times L$ seconds) and wants to reconnect to the server, the client cannot directly use the newest IR since it may not provide enough information for the client to invalidate its local cache, so the client only needs to send a short reconnect request message to the server. Since the server keeps the cache information of this client, the server simply returns an IR to the client to be used in invalidating its cache. Hence, these two hybrid cache invalidation schemes can significantly reduce the uplink request communication cost. Low uplink cost is quite important because the uplink bandwidth is much smaller than the downlink bandwidth. Moreover, uplink data transmission requires much higher power from mobile clients than downlink data reception [7]. Our hybrid cache invalidation schemes deal with considerably long disconnection cases; without increasing the uplink cost and the upcoming query processing cost.

The main difference between HSB and HABSB is that HSB uses the well known Broadcasting Timestamp strategy [1, 11] to send IRs, while HABSB modifies every IR by inserting in it the attribute bit sequence of each updated data. This significantly reduces the transmission of query results from the server to the querying client.

The rest of the paper is organized as follows. Section 2 reviews the previous cache invalidation schemes in mobile environment. Section 3 proposes the hybrid cache invalidation schemes. Section 4 describes the simulation model. Section 5 describes the simulation results. Section 6 includes a summary and the conclusions.

2. Related Work

As described in the literature, there are two types of cache consistency maintenance approaches for wireless mobile environments: stateless and stateful. Some approaches are briefly covered next; three of them compared with our schemes proposed in this paper and hence are covered in more details.

Cache consistency issues for mobile environments were first addressed by Barbara and Imielinski [1, 2]. Subsequent research efforts, e.g., [4, 8] were basically devoted to designing efficient algorithms to reduce IR overhead and improve uplink cost. For instance, Jing *et al* [8] proposed a scheme called Bit-Sequence with a compressed IR which consists of a set of binary bit sequences each with a corresponding timestamp. This algorithm is particularly attractive for mobile clients of frequent disconnections. However, the model requires the broadcast of a larger number of IR messages than the algorithms described in [1, 2]. In order to reduce the broadcast of IR messages, Hu *et al* [7] introduced several adaptive cache invalidation schemes to broadcast different IRs based on update frequency, mobile clients access and sleep-wakeup patterns. Wu *et al* [16] proposed an uplink validation check scheme that can deal with long sleep-wakeup patterns. Cai *et al* [14] introduced a view-based cache consistency algorithm with incremental techniques. Cao [4, 5] keeps the invalidated data objects in a mobile client cache. The mobile client can update a data object if it is received from the broadcast channel. This approach increases the broadcast channel utilization. However, keeping invalid data objects in a mobile client cache wastes precious cache memory. Also, whenever mobile client cache content is changed, the mobile client must piggyback the change to the server, thus consuming battery power and uplink bandwidth. Xu *et al* [17] considered data inconsistency caused by client movements and proposed three location-dependent cache invalidation schemes. Yue *et al* [19] employ an absolute validity interval for each data object. However, they failed to reduce the access delay introduced by periodic broadcast cycles.

Very few stateful cache consistency maintenance algorithms have been proposed for wireless mobile computing environments. For instance, Kahol *et al* [9] proposed an asynchronous stateful algorithm to maintain cache consistency, where a mobile station (MS) records all retrieved data objects for each mobile client. After a mobile client wakes up, it first retrieves a data object based on the mobile client cache content record and sleep-wakeup time; the MS sends an IR to that particular mobile client. Whenever an MS receives an update from the original server for each recorded data object, it immediately broadcasts that data object's IR to mobile clients. The advantage of this scheme is that the MS avoids unnecessary IR broadcast to mobile clients. Moreover, mobile clients can deal with any sleep-wakeup pattern without losing valid data objects. However, in order to maintain each mobile client cache state, the MS must record all cached data objects for each mobile client. Hence, a mobile client can only download data objects which it requested through the uplink. This makes the broadcast channel utility inefficient and sensitive to the number of mobile clients.

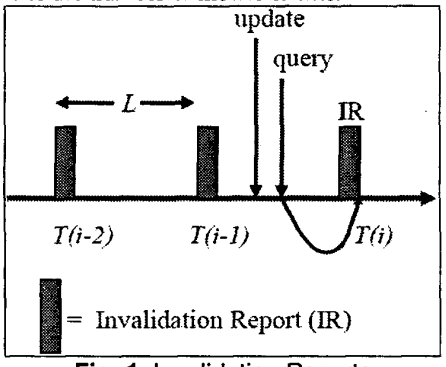

Fig. 1. Invalidation Reports

2.1 Broadcasting Timestamp Strategy

Barbara *et al* [1, 2] use repetitive broadcast of IR to notify mobile clients of the recent updates at the server. They proposed three stateless algorithms that suffer from the following drawbacks: 1) do not scale well for large databases and/or fast updating data systems because of the increased number of IR messages; 2) the average access latency is always longer than half of the broadcast period, simply because all requests can be answered only after the next IR; and 3) all cache entries are deleted when the time for a mobile client being disconnected from its MS (the sleep time) is longer than $w \times L$, thus leads to unnecessary bandwidth consumption, particularly if the data object is still valid.

Timestamp (*TS*) strategy is one of their proposals where the server broadcasts every L seconds an IR composed of the timestamps of the latest change for items that had been updated during the last $w \times L$ seconds. The client listens to reports and updates the status of its cache. For each item cached, regardless whether there is a related pending query, the client either discards it from the cache (when the item is reported to have changed at a time larger than the timestamp stored in the cache), or updates the cache's timestamp of the item to the timestamp of the report (if the item was not in IR).

As shown in Figure 1, the server broadcasts IR periodically at each time step $T_i = i \times L$, and keeps a list: $IR_i = \{[j, t_j] \mid j \in D\}$, where t_j is the timestamp of the last update of item j such that $T_i - w \times L \le t_j \le T_i$, and D denotes the set of data item IDs in the database.

Upon receiving IR, the client compares the items in its cache $\{[j, t_j^c] \mid$ where $j \in D$ and t_j^c is the cache's timestamp for $j\}$ to decide if j should be kept in the cache. The client also has a list: $Q_i = \{j \mid j$ has been queried in the interval $[T_{i-1}, T_i]\}$

The client keeps a variable T_l that indicates the last time it received a report. If the difference between the current report timestamp and T_l is larger than $w \times L$, then the entire cache is dropped. Practically, the client runs Algorithm 1, given next.

Algorithm 1: Timestamp (TS) Client Cache Invalidation
Input: T_i, T_l, w, L, IR_i, t_j^c, t_j, Q_i
Output: query results or the queries for cache missed data

```
If (T_i - T_l > w×L)  Drop the entire cache;
Else {
    For every item j in the client cache {
        If (there is a pair [j, t_j] in IR_i)
            If (t_j^c < t_j)   Throw j out of the cache;
            Else               t_j^c := T_i;
    }
}
For every item j∈ Q_i {
    If (j is in the cache)
        Use the cache's value to answer the query;
    Else  Go uplink with the query;
}
T_l := T_i;
```

To save energy, a client may stay in the dozing mode most of the time and become activate only during the IR broadcast time. Moreover, a client may be in the power off mode for a long time to save energy, and hence the client may miss IRs. Since IR includes the history of the past $w \times L$ seconds, the client can still invalidate its cache as long as its disconnection time is shorter than $w \times L$. However, if the client disconnects for longer than $w \times L$, it must discard all cached data items since it has no way to tell which parts of the cache are valid. Unfortunately, if the client needs to access some items in its cache, discarding the entire cache may consume a large amount of wireless bandwidth in future queries.

2.2 Simple Checking Caching Scheme

In the *TS* strategy described in Section 2.1, if a mobile client disconnects longer than $w \times L$ seconds and wants to reconnect to the server, then its entire cache must be dropped, even though its cache may still contain many valid data items. This will require substantial uplink and downlink communication for subsequent query processing. Wu *et al* [16] propose a caching scheme, called the Simple Checking Scheme, to deal with this problem. To retain as many valid data items as possible in the cache, the algorithm validates each data item.

For the Simple Checking Scheme, the server broadcasts an IR periodically (same as in the *TS* strategy). All connected mobile clients in the cell will receive the IR and use it to invalidate their local caches. When a mobile client reconnects after a long disconnection, instead of dropping its entire cache, the client simply sends its cache timestamp and the IDs of all cached data items to the server for invalidation. The server then checks the validity of each cached data item and sends an invalidation report back to the mobile client. After receiving the report from the server, the client then invalidates its cache and continuously processes queries. The drawback of this scheme is that it requires substantial uplink communication for validity checking for large cache size.

2.3 Two Phase Cache Invalidation and One Phase Cache Invalidation

To check the validity of a mobile client's local cache after a very long disconnection, Kang *et al* [10] proposed two new cache validation schemes that may efficiently conserve the bandwidth both for cache validation and query processing afterwards. The first scheme is the Two Phase Cache Validation (2PCV), in which a mobile client checks the validity of its cache with the server in two phases. In the first phase, the mobile client sends to the server the IDs of the groups in its cache with the timestamp (T_l) of the most recent invalidation report it has received. Groups are used to partition the database according to some rules so each data object is mapped to a unique group. The server then replies to the mobile client sending the obsolete group IDs. In the second phase, the mobile client returns IDs for the server of cached data objects belonging to the obsolete groups in the first phase. The server then replies with the obsolete data object IDs. After this phase, the mobile client retains as many valid data objects as possible.

The second scheme is the One Phase Cache Validation (1PCV), in which a mobile client checks the validity of its cache with the server in one phase, i.e., the mobile client sends IDs to the server of the groups represented in its cache with the timestamp (T_l) of the most recent IR it has received. The server then replies to the mobile client with all obsolete object IDs, not the obsolete group IDs.

The disadvantage of 2PCV is that the mobile host must communicate with the server in two phases; consuming bandwidth in both phases. The advantage of 1PCV is that it downgrades the granularity of the identifiers transmitted between the server and the mobile hosts; this saves bandwidth. The disadvantage of 1PCV is that the server does not know what data are needed by the mobile host, so it may send the mobile host some irrelevant information that wastes downlink bandwidth.

3. Hybrid Cache Invalidation Schemes

In this section, we describe the proposed two cache invalidation schemes.

3.1 Hybrid cache invalidation with Simple Broadcasting (HSB)

This scheme is proposed to cope with the long disconnection problem. To serve the purpose, we apply a stateful BS instead of stateless BS. In this scheme, the stateful server keeps all the information of each mobile client in its cell, including clients who have disconnected for a very long time, but subsequently try to reconnect to the server. Meanwhile, the stateful server broadcasts periodically an IR that has the same structure as described earlier [1, 11]. In this situation, the stateful server acts as a stateless server because the server does not know if any mobile client in its cell has disconnected. The server assumes that every mobile client in its cell is in the connect mode, so it uses IR to invalidate the obsolete data items on the server side. Thus, for each data item k in the database, the stateful server maintains in the client list: its update time T_k and a list of client IDs that cached item k as well as the validity flags of k cached by those clients. When a mobile client (say j) newly cached an item k, j will be added in the client list associated with k and the validity flag of k cached by j is set to 0 (valid). If k is updated before client j is removed from the client list associated with k, then the validity flag of k cached by j will be set to 1 (invalid). Thus, the server has invalid cache information for each mobile client in its cell. Therefore, each time a mobile client (say i) disconnects for a long time (longer than $w \times L$ seconds) and wants to reconnect to the server, it sends a small "reconnect" request message and its cache timestamp TS_i (the timestamp of its latest IR received before its disconnection) to the server. Since the server has invalid

cache information for client i, it can reply directly to client i with an IR that contains all invalid data item IDs in client i's local cache. Also, client i can use the report to discard all obsolete data items in its cache. Therefore, unlike simple checking, 2PCV and 1PCV, HSB minimizes communication cost between a client and the server to enforce the validity checking after long disconnection. Moreover, in this scheme, the stateful server also stores the timestamp of every IR and the request time for each mobile client in its cell.

The structure of the client list associated with any data item k at the server is:

```
struct Client_List {
        int clientID;
        int validity_flag; // 0 and 1 represent valid cache and
                           // invalid cache of item k by the client
        Client_List *prev;
        Client_List *next;
}
```

On the mobile client side, each client i's cache maintains the timestamp (TS_i) of its cache and all cached data items. This scheme functions as follows: when a new mobile client i moves into the cell, it sends a "register" request to server S to establish communication. After S receives this request, it creates and stores a tuple (i, $ReqT_i$), where i is the client ID and $ReqT_i$ is the timestamp of the "register" request of client i. Then, S sends an acknowledgement and TS of the current IR to client i, which is thereby connected to server S and can send queries to S. Client i can also receive an IR and use it to invalidate its local cache. The whole process is reflected in two algorithms: Algorithm 2 runs on the server side and Algorithm 3 runs on mobile clients' side.

Algorithm 2: HSB (Stateful Server side)
Input: client lists, queries, $ReqT_i$, w, L
Output: IRs, and query results

1. Generate "register" request tuple record
Receive a "register" request from mobile client i;
Send Acknowledgement and TS of current IR to i;
Save (i, $ReqT_i$) tuple in memory;

2. Broadcast
At broadcast timestamp T_i, generate a new IR_i:
$IR_i = \{[j, t_j] \mid j \in D\}$;
Broadcast IR_i in its cell in $i \times L$ seconds;

3. Receive a query from mobile client i
Process and send the query result to client i;
For each data item k in the query result {
  If (there is a $(i,0)_k$ or $(i,1)_k$ record in the client list associated with k)
    Remove the located record from the client list;
  Insert $(i,0)_k$ at the head of the client list;
}

4. Receive *reconnect* request and TS_i from mobile client i
Set IR_i to empty;
Update $ReqT_i$ in (i, $ReqT_i$) tuple to the current time;
For each data item k that contains the ID of client i in its client list {
  if (flag of client i == 1) {
    // invalid caching of k by client i
    if (update timestamp of $k > TS_i$)
      Add the ID of k to the invalidation report;
    Remove client i from the client list associated with k;
  }
}
Send the invalidation report to client i;

Algorithm 3: HSB (at each mobile client i side)
Input: IR for reconnecting, IR, w, L
Output: the query results

1. Enter the cell
Send a "register" request message to the server;
Wait until receiving the Acknowledgement and TS of current IR from the server;

2. Receive IR when client i is in connect mode
For each data item k in client i's cache {
  If (update TS of k in $IR > TS_i$)
    Discard k from cache;
}
TS_i = current TS of the newest IR;

3. Reconnect to the server
If (disconnect time $\leq w \times L$)
  Use the newly received IR to invalidate its local cache; // same as step (2) above
Else {
  Send "reconnect" request message and TS_i to the server;
  Wait until receiving IR;
  Use the received IR to discard the invalid data items in its cache;
}

4. Generate a query
Wait until the next IR arrives;
For each data item k queried by client i {
  If (k is in the local cache)
    Add k to the query result list;
  Else
    Send ID of k to the server;
}
Wait until receiving missed data items rev_data;
Add rev_data in the query result list;
Provide the query result list to the user;
While (the cache is full before inserting rev_data)
  Discard the oldest cached data item from cache;
Insert rev_data into cache;

In our scheme, if mobile client i wants to leave the cell, it can send a "unregister" message to the server. The server will remove all information about client i. If client

i is leaving for a short period, and will return; client i does not need to notify the server because it will be treated as any disconnected mobile client in the cell.

If the server storage is nearly full, the information for some clients needs to be removed. Our scheme uses the following purge strategy: The server checks all (client ID, Request Timestamp) tuples and finds the client with the smallest (i.e., oldest) Request Timestamp, say client i. The server then tries to communicate with client i. If client i does not respond in a certain time period, the information of client i will be removed from the server storage. However, if a response is received, the server communicates to the client with the next smallest Request Timestamp. This process continues until a client with no response is found and removed from the server.

3.2 Hybrid cache invalidation with Attribute Bit Sequence Broadcasting (HABSB)

The Hybrid cache invalidation with Bit Sequence Broadcasting (HABSB) scheme is proposed to avoid discarding the whole tuple of an obsolete data item stored in a mobile client's cache and to cope with the long disconnection problem, and hence to improve the caching effectiveness. Similar to the Broadcasting Timestamp Strategy [1, 2], the server broadcasts IR periodically, say every L seconds. So, the connected mobile clients in the cell can use IR to validate their local caches. However, unlike the Broadcasting Timestamp Strategy, IR of our scheme contains the timestamps of the latest change for data items that have updates in the last $w \times L$ seconds, their IDs, and the attribute bit sequences of those updated data items. Thus, each bit in the attribute bit sequence corresponds to a specific attribute. Each '1' bit in the sequence represents the corresponding attribute which has been updated within a certain period of time (i.e. $[T_i - w \times L, T_i]$). Conversely, each '0' bit indicates that the corresponding attribute has not been updated during that period. The structure of the i^{th} IR in our scheme is captured by: $IR_i = \{[j, t_j, S_j] \mid j \in D\}$, where t_j is the timestamp of the last update of item j such that $T_i - w \times L \leq t_j \leq T_i$ and S_j is the attribute bit sequence of item j, and D denotes the set of data item IDs in the database.

The size of each IR will increase because of the added bit sequences; but this can significantly increase the cache hit ratio and reduce the size of transmission of up-to-date values of obsolete cached data items from the server to mobile clients when processing queries. We now provide a more detailed description.

After receiving the newest IR, the connected mobile clients in the cell can use it to assign all updated data item attributes (i.e., bit '1's in the attribute bit sequences of updated data items in IR) to "invalid" and assign the validity flags of these updated data items in cache to "invalid", instead of discarding whole tuples as in HSB. These newly updated data items will most likely be queried in the near future. However, only part of the attributes of these updated data items will be changed, so it is a good idea to keep the unchanged attributes of these updated data items in the client's local cache to increase cache hit ratio. Thus, if such an invalid cached data item (say k) is queried, then the mobile client sends the ID and attribute bit sequence of k to the server, where '1' bits in the sequence represent corresponding invalid attributes of k and '0' represent valid attributes of k.

After receiving the query request, the server responds with only the new values of "invalid" attributes of k. The client just updates these attributes, assigns the bit flags of them to "valid" and sets the validity flag of k in the cache to "valid". Hence, the downlink communication cost is greatly decreased when the server answers queries because no unnecessary attributes will be transmitted. This scheme performs better than the HSB scheme when the cache size is relatively large (≥2000 data items) and only a small part (e.g. 20%) of any updated data item is updated each time (See Sections 4 and 5). Finally, Algorithm 4 runs on the server side; Algorithm 5 runs on mobile clients' side; and the structure of each client i cache is represented as follows:

```
Struct Client_Cache {
    int dataID;
    int inv_flag;  // 0: valid caching; 1: invalid caching
    {set of bit flags in the attribute bit sequence of dataID};
    Client_Cache *prev;
    Client_Cache *next;
  }
```

Algorithm 4: HABSB (Stateful Server side)
Input: the client lists, queries, $ReqT_i$, w, L
Output: the invalidation reports, and query results

1. **Generate "register" request tuple record**
   Same as Step 1 in Algorithm 2;
2. **Broadcast**
   At broadcast timestamp T_i, generate a new IR:
   $IR_i = \{[j, t_j, S_j] \mid j \in D\}$
   where S_j is the attribute bit sequence of item j;
   Broadcast IR_i in its cell in $i \times L$ seconds;
3. **Receive a query from mobile client i**
   For each data item k in the query {
   If (there is an attribute bit sequence of k)
   Send only up-to-date values of bit "1" attributes of k to client i;
   Else Send the whole tuple of k to client i;
   If (there is a $(i,0)_k$ or $(i,1)_k$ record in the client list associated with k)
   Remove $(i,0)_k$ or $(i,1)_k$ record from the client list;
   Insert $(i,0)_k$ into head of the client list associated with k;
   }

4. **Receive "reconnect" request and TS_i from client i**
Same as Step (4) in Algorithm 2;

Algorithm 5: HABSB (Mobile client i side)
Input: IR for reconnecting, IR, w, L
Output: the query results

1. **Enter the cell**
Send "register" request message to the server;
Wait until receiving the Acknowledgement and timestamp of the current IR from the server;
2. **Receive IR when client i is in connect mode**
For each data item k in client i cache {
If (update timestamp of k in IR > TS_i) {
// invalid caching of k
Set the validity flag of cached k to 1;
Assign bit flags of all updated attributes of k (i.e. '1' bits in the attribute bit sequence of k in IR) to "invalid";
}
}
TS_i = current timestamp of the newest IR;
3. **Reconnect to the server**
If (disconnect time $\leq w \times L$)
Use the newly received IR to invalidate its local cache;
// same as Step 2 above;
Else {
Send "reconnect" request message and TS_i to the server;
Use the received IR to discard the invalid data items in its cache;
}
4. **Generate a query**
Wait until the next IR arrives;
For each data item k queried by client i {
If (k is in the local cache) { // k is valid in cache
If (validity flag of cached k is 0)
Add k to the query result list;
Else
Sends the ID and attribute bit sequence of k to the server;
}
Else Send ID of k to the server;
}
Add received data items rev_data into the query result list;
Provide the query result list to the user;
While (the cache is full before inserting rev_data)
Discard the oldest invalid cached data item;
Insert rev_data into cache, reset their validity flags to 0, and assign all bits in their attribute bit sequences to 0;

4. Simulation Model

We use the communication bandwidth consumption of a set of mobile clients in the cell of a single database server to analyze the performance of different schemes. The bandwidth consumption includes the uplink cost of checking cache validity during reconnection, querying missed data items, and the downlink cost of broadcasting IR, sending IR upon "reconnect" request, and sending query result. The bandwidth consumption is accumulated as the average of the communication cost over 20,000 queries. The bandwidth consumption for each query is quite small when the mobile client frequently disconnects for a long time, so we accumulate the bandwidth consumption for 1000 queries. Finally, the following assumptions made in our model are typical for such an environment [7, 16].

Table 1. Simulation Parameters

Parameter	Definition	Setting
client_num	number of mobile clients	30
DB_size	database size	100,000 items
cache_size	cache size	5000 items
data_size	data item size	2048 bits
attr_size	attribute size	64 bits
ID_size	data ID size	64 bits
TS_size	timestamp size	64 bits
down_bw	network downlink bandwidth	10,000 bps
up_bw	network uplink bandwidth	10,000 bps
ctr_msg_size	control message size	64 bits
update_data_num	number of updated data items per transaction	10
query_data_num	number of data items referenced by a query	20
w	number of IR intervals	10
L	Broadcast interval	20 sec
update_arrtime	mean update arrival time	100 sec
query_arrtime	mean query arrival time	10 sec
discon_time	mean disconnect time	1000 sec
discon_prob	Disconnect probability after answering a query	0.1
hot_update_region	hot update region ratio	0.05
hot_update_prob	hot update probability	0.9
hot_query_region	hot query region ratio	0.01
hot_query_prob	hot query probability	0.9
updated_attr_ratio	Ratio of updated attributes over the whole attributes of a data item	0.2

- All data items in the database are updated in the server by independent transactions
- Each update transaction arrives in an exponential distributed interarrival time with mean update_arrtime.
- All queries are read-only and generated in each mobile client. Each query arrives in an exponential distributed

interarrival time interval with mean query_arrtime, and cannot be processed until the next IR arrives and the previous query has been answered.

- Each mobile client will disconnect in an exponential distributed time, denoted discon_time with a probability denoted *discon_prob* after it answers a query. If some queried data items are not cached in the mobile client, then the IDs of these data items will be sent to the server which replies with these data items.
- The message priority is: the update transaction has the highest priority, then broadcasting IR, followed by the IR upon "reconnect" request and transmitting queried data items, while all other messages are of lower priority.

Listed in Table 1 are the default parameters, definitions and setting used throughout the simulations described in this paper, unless otherwise specified. All simulations are performed on Linux operating system using the CSIM simulation package 19; refer to [11] for more details on this package.

5. Simulation Results

We conducted some experiments to demonstrate the applicability and effectiveness of the proposed caching schemes. For this purpose, we compared the proposed two schemes HSB and HABSB with the three schemes described in Section 2: namely, 1PCV, 2PCV and simple checking. Further, we employed different data access patterns, such as server update patterns and mobile client query patterns, similar to Hu and Lee (1997). For the database server, two database regions, the hot and cold update regions, are specified. The hot update probability denotes the probability that an update will address a data item in the server's hot update region. The hot and cold query regions in the database are defined for mobile clients querying data items. The hot query probability refers to the probability that a query will address a data item in the mobile client's hot query region. Data items in a specified region are randomly chosen.

To compare the performance of the five schemes, the bandwidth consumption is accumulated under different system parameters, including the number of mobile clients, database size, cache size, disconnect probability, etc. We used the following parameters in the simulation: 1) the number of mobile clients is client_num=30; 2) the database size is DB_size=100,000 data items; 3) the cache size is cache_size=5000 data items; 4) the size of a data item is data_size=2048 bits; 5) the size of an attribute in a tuple is attr_size=64 bits; and 6) the size a control message size is ctr_msg_size=64 bits. We also assume that the uplink bandwidth is the same as the downlink bandwidth; both take the value of 10,000 bits per second.

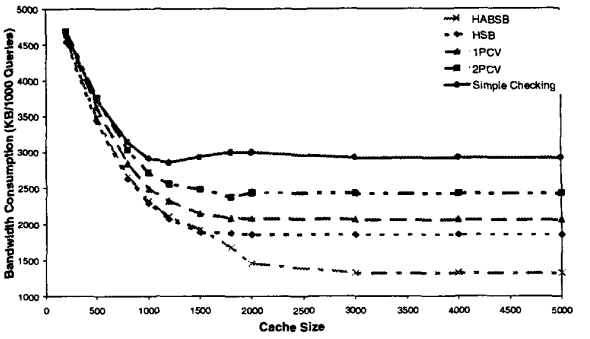

Fig. 2. Impact of cache size

In the first experiment, we examine the impact of cache size for each mobile client on total bandwidth consumption. Figure 2 shows the total bandwidth consumption per 1000 queries for the five schemes. Most queried data items are not cached when the cache size is small (<500 data items); so the server must send a large amount of queried data items to the mobile clients. This overhead reflects the effectiveness of the validity checking during reconnection. This is why the differences in these five schemes are relatively small for small cache size. However, as cache size increases, the differences become larger. From Figure 2, we see that the Simple Checking Scheme has the highest bandwidth consumption, while HSB and HABSB consume the lowest bandwidth. This is because the cache hit ratio will be higher when the cache size grows, so the influence of validity checking for reconnection on the total bandwidth consumption will increase. For fixed cache size, the larger the cost of validity checking, the bigger the total bandwidth consumption. The Simple Checking scheme has the highest cost for validity checking so it exhibits the highest bandwidth consumption. Both 1PCV and 2PCV have less communication cost for validity checking; so they are better than the Simple Checking Scheme. On the other hand, HSB and HABSB have the lowest cost for validity checking, so they use the lowest bandwidth. Furthermore, if the cache size is relatively large (>2000), HABSB has lower bandwidth consumption than HSB. The reason is that HABSB keeps most invalid (i.e., obsolete) data items in the local cache instead of using the IR for discarding whole tuples as invalid data items. This will greatly increase the cache hit ratio when processing queries. During query processing in HABSB, only the updated attributes of whole tuples of queried data items will be sent from the server to the mobile client when these data items are cached but obsolete. This will reduce the downlink cost for answering queries. The HSB discards all invalid cached items using IR, but this will increase data

transmission costs over the HABSB for later query processing.

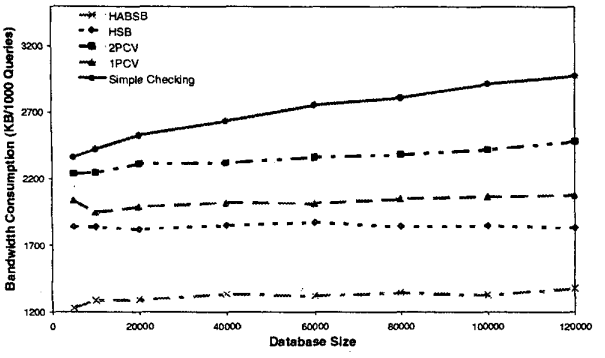

Fig. 3. Impact of database size

The second simulation is dedicated to investigate the impact of database size on total bandwidth consumption. Figure 3 shows that the total bandwidth consumption for Simple Checking Scheme goes up quickly as database size increases. The other four schemes are much less affected by database size because of the small uplink cost for validity checking during reconnection. HSB and HABSB consume less bandwidth than the other schemes because the uplink cost for validity checking is quite small (only a short "reconnect" request message is sent to the server). Furthermore, HABSB performs much better than HSB because HABSB stores more useful and valid information about the data in the local cache. Thus, HABSB minimizes the transmission of queried data.

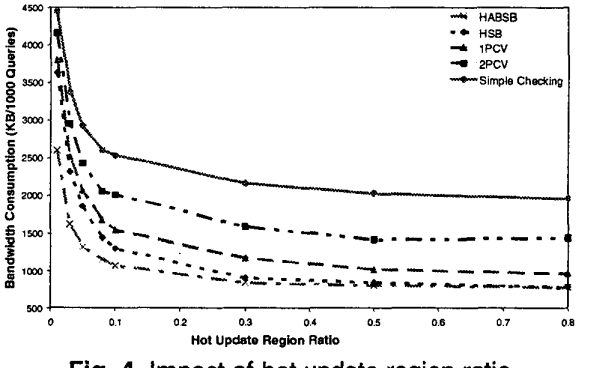

Fig. 4. Impact of hot update region ratio

In the third experiment, bandwidth consumption is tested for various hot update region ratios. The results are plotted in Figure 4, where it can be easily seen that bandwidth consumption decreases as the hot update region ratio increases for all the five schemes because the influence of caching goes down. HSB and HABSB have lower bandwidth consumption than the other schemes. When the ratio is low (e.g., < 40%), HABSB significantly outperforms HSB. This is because when the ratio decreases, more cached data items will be updated and thus be invalidated by the IR. The HABSB keeps the most valid attributes of these obsolete data items in cache, but the HSB simply discards them from cache. Hence HABSB will require less data transmission of queried data items than HSB. However, if the ratio increases, cache-missed data items increase, so caching has less effect on improving system performance because of the low cache hit ratio. The overhead of attribute bit sequences in the IR will make the difference between HABSB and HSB less significant.

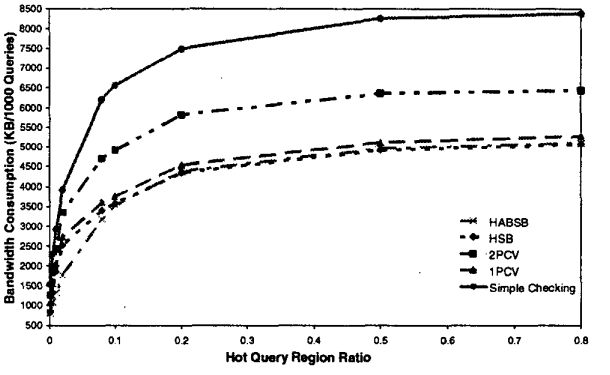

Fig. 5. Impact of hot query region ratio

In the fourth experiment, we tested the influence of hot query region ratio on the bandwidth consumption. Figure 5 shows that bandwidth consumption of all the five schemes grows rapidly for low ratio (from 0% to 20%) and then grows slowly after the ratio increases beyond 20%. This illustrates that caching has substantial effect on bandwidth consumption when the ratio of hot query region is low. Both HSB and HABSB are still the best ones. However, when the ratio is very low (< 10%), HABSB outperforms HSB because most queried data items are cached but may be discarded due to the update for HSB, while HABSB retains most of these data items in cache. Thus, HABSB requires less data transmission for later query processing than HSB does.

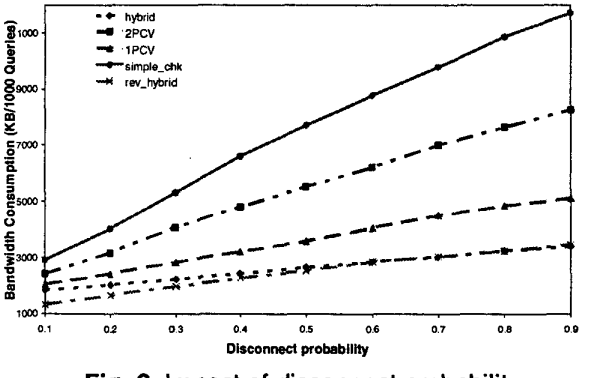

Fig. 6. Impact of disconnect probability

In the fifth experiment, bandwidth consumption is examined under various disconnection probabilities. The results are reported in Figure 6, where it can be easily seen that increasing disconnection probability will induce more bandwidth consumption for all the five schemes, but with different slopes. Both HABSB and HSB have lower slopes than the other three schemes because of the low cost of uplink communication for validity checking. Finally, HABSB is superior to HSB when the disconnection probability is less than 60%.

6. Summary and Conclusions

We proposed two hybrid cache invalidation schemes, called HSB and HABSB. These two schemes are energy-efficient caching schemes since they allow any mobile client in a server's cell to disconnect for long periods to save energy, while exploiting caching benefits.

Both HSB and HABSB employ a stateful server to cope with cache validity checking after long disconnection. When a mobile client wakes up after disconnecting longer than $w \times L$ seconds, it only needs to send the server a short "reconnect" request message. Since the server has the caching information of each mobile client in its cell, it can send back IR upon receiving the request. This can significantly reduce the uplink cost for validity checking, and hence reduce the overall bandwidth consumption. The conducted experiments demonstrated that HSB and HABSB outperform Simple Checking Scheme, 2PCV and 1PCV. The HSB and HABSB require the lowest bandwidth consumption, so they consume the smallest energy. Hence, they are the most energy efficient among the five schemes considered in the simulation. Moreover, in our HABSB scheme, the attribute bit sequences are added in IR to increase the cache hit ratio and reduce the data transmission of queried data items. Finally, when caching has large impact on system performance, HABSB is superior to HSB.

References

[1] D. Barbara and T. Imielinksi, "Sleeper and Workaholics: Caching Strategy in Mobile Environments," *Proc. of ACM SIGMOD*, pp.1-12, 1994.

[2] D. Barbara and T. Imielinski, "Sleepers and Workaholics: Caching Strategies in Mobile Environments," (extended version), *VLDB Journal*, Vol.4, pp.567-602, 1995.

[3] D. Barbara, "Mobile computing and databases – a survey," *IEEE Transactions on knowledge and Data Engineering*, Vol.11, No.1, pp.108-117, 1999.

[4] G. Cao, "A Scalable Low-latency Cache Invalidation Strategy for Mobile Environments," *Proc. of the International Conference on Mobile Computing and Networking*, pp.200-209, 2000.

[5] G. Cao, "On Improving the Performance of Cache Invalidation in Mobile Environments," *ACM/Kluwer Mobile Network and Applications*, Vol.7, No.4, pp.291-303, 2002.

[6] B.Y. Chan, A. Si and H.V. Leong, "A Framework for Cache Management for Mobile Databases: Design and Evaluation," *Distributed and Parallel Databases*, Vol.10, pp.23-57, 2001.

[7] Q. Hu and D.L. Lee, "Cache Algorithms Based on Adaptive Invalidation Reports for Mobile Environments," *Cluster Computing*, Vol.1, No.1, pp.39-50, 1998.

[8] J. Jing, A. Elmagarmid, A. Heal, and R. Alonso, "Bit-Sequences: An Adaptive Cache Invalidation Method in Mobile Client/Server Environments," *Mobile Networks and Applications*, Vol.2, No.2, pp.115-127, 1997.

[9] A. Kahol, et al, "A Strategy to Manage Cache Consistency in a Distributed Mobile Wireless Environment," *IEEE Transactions on Parallel and Distributed Systems*, Vol.12, No.7, pp.686-700, 2001.

[10] H. Kang and S. Lim, Bandwidth-Conserving Cache Validation Schemes in a Mobile Database System, *Proc. of MDM*, LNCS 1987, pp.121-130, 2001.

[11] H. Schwetman, "CSIM User Guide (version 19)," *Mesquite Software Inc.*, 1994.

[12] K.L. Tan, "Organization of invalidation reports for energy-efficient cache invalidation in mobile environments," *Mobile Networks and Applications*, Vol.6, pp.279-290, 2001.

[13] J. Yao and M. H. Dunham, "Caching management of mobile DBMS," *Integrated Computer-Aided Engineering*, Vol.8, pp.151-169, 2001.

[14] K. Tan, J. Cai and B. Ooi, "An Evaluation of Cache Invalidation Strategies in Wireless Environments," *IEEE Transactions on Parallel and Distributed Systems*, Vol.12, No.8, pp.889-897, 2001.

[15] Z. Wang, S. K. Das, H. Che and M. Kumar, "SACCS: Scalable Asynchronous Cache Consistency Scheme for Mobile Environments," *Proc. of ICDCS workshops: International workshop on Mobile and Wireless Networks*, Rhode Island, May 2003.

[16] K.L. Wu, P.S. Yu and M.S. Chen, "Energy-Efficient Caching for Wireless Mobile Computing," *Proc. of IEEE ICDE*, pp.336-345, 1996.

[17] J. Xu, X. Tang and D.L. Lee, "Performance Analysis of Location-Dependent Cache Invalidation Schemes for Mobile Environments," *IEEE Transactions on Knowledge and Data Engineering*, Vol.15, No.2, 2003.

[18] H. Yu, L. Breslau and S. Shenker, "A Scalable Web Cache Consistency Architecture," *Proc. of ACM SIGCOMM*, pp.163-174, 1999.

[19] J.C. Yuen, et al, "Cache Invalidation Scheme for Mobile Computing Systems with Real-time Data," *SIGMOD Record*, Dec. 2000.