ACM Intl. Conf. on Mobile Computing and Networking (Mobicom'00), Aug. 2000

A Scalable Low-Latency Cache Invalidation Strategy for Mobile Environments

Guohong Cao
Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802
gcao@cse.psu.edu

ABSTRACT

Caching frequently accessed data items on the client side is an effective technique to improve performance in a mobile environment. Classical cache invalidation strategies are not suitable for mobile environments due to the disconnection and mobility of the mobile clients. One attractive cache invalidation technique is based on invalidation reports (IRs). However, the IR-based cache invalidation solution has two major drawbacks, which have not been addressed in previous research. First, there is a long query latency associated with this solution since a client cannot answer the query until the next IR interval. Second, when the server updates a hot data item, all clients have to query the server and get the data from the server separately, which wastes a large amount of bandwidth. In this paper, we propose an IR-based cache invalidation algorithm which can significantly reduce the query latency and efficiently utilize the broadcast bandwidth. Detailed simulation experiments are carried out to evaluate the proposed methodology. Compared to previous IR-based schemes, our scheme can significantly improve the throughput and reduce the query latency, the number of uplink request, and the broadcast bandwidth requirements.

1. INTRODUCTION

The falling cost of both communication and mobile terminals (laptop computers, personal digital assistants, hand-held computers, etc.) has made mobile computing commercially affordable to both business users and private consumers. In the near future, people with battery powered mobile terminals (MTs) can access various kinds of services over wireless networks at any time any place. However, due to limitations on battery technologies [4, 12], these MTs may be frequently disconnected (i.e., powered off) to conserve battery energy. Also, the wireless bandwidth is rather limited. Thus, mechanisms to efficiently transmit information from the server to the clients (running on MTs) have received considerable attention [1, 4, 7, 11, 14, 15].

Caching frequently accessed data on the client side is an effective technique to improve performance in a mobile environment. Average data access latency is reduced as several data access requests can be satisfied from the local cache thereby obviating the need for data transmission over the scarce wireless links. However, the disconnection and mobility of the clients make cache consistency a challenging problem. Effective cache invalidation strategies are required to ensure the consistency between the cached data at the clients and the original data stored at the server.

Depending on whether or not the server maintains the state of the clients' cache, there are two invalidation strategies: *stateful* server approach and *stateless* server approach. In the stateful server approach, the server maintains the information about which data are cached by which client. Once a data item is changed, the server sends invalidation messages to the clients that have copies of the particular data. The Andrew File Systems [10] is an example of this approach. However, in mobile environments, the server may not be able to contact the disconnected clients. Thus, a disconnection by a client automatically means that its cache is no longer valid. Moreover, if the client moves to another cell, it has to notify the server. This implies some restrictions on the freedom of the clients. In the stateless server approach, the server is not aware of the state of its clients' cache. The client needs to query the server to verify the validity of their caches before each use. The Network File System (NFS) [13] is an example that takes this approach. Obviously, in this option, the clients generate a large amount of traffic on the wireless channel, which not only wastes the scarce wireless bandwidth, but also consumes a lot of battery energy.

In [1], Barbara and Imielinski provided another stateless server option. In this approach, the server periodically broadcasts an *invalidation report (IR)* in which the changed data items are indicated. Rather than querying the server directly regarding the validation of cached copies, the clients can listen to these IRs over the wireless channel. In general, a large IR can provide more information and is more effective for cache invalidation, but a large IR occupies a large amount of broadcast bandwidth and the clients may need to spend more power on listening to the IR since they cannot switch to power save mode when listening to the IR. The broadcasting timestamp (TS) scheme [1] is a good example that limits the size of the IR by broadcasting the names and timestamps only for the data items updated during a window of w IR intervals (with w being a fixed parameter). However, any client who has been disconnected longer than w IR intervals cannot use the report, and it has to discard all cached items even though some of them may still be valid.

Many solutions are proposed to address the long disconnection problem. Wu *et al.* [16] modified the TS scheme to include cache validity checks after reconnection. In this way, the client is still able to keep most of its cache even after a long disconnection and save wireless bandwidth and battery energy. Jing *et al.* [9] proposed a *bit-sequence (BS)* scheme which uses a hierarchical structure of binary bit sequences with an associated set of timestamps to represent clients with different disconnection times. The BS structure contains the update information of the whole database, and it is good for clients with long disconnections to invalidate their cache. However, for disconnections that are barely longer than the window w, the use of BS report is quite wasteful since it needs to broadcast a long IR. Based on this observation, Hu and Lee [6] proposed a scheme which broadcasts a TS report or BS report based on the update and query rates/patterns and client disconnection time.

Although different approaches [1, 6, 9, 16] apply different techniques to construct the IR, these schemes maintain cache consistency by periodically broadcasting the IR. The IR-based solution is attractive because it can scale to any number of clients who listen to the IR. However, the IR-based solution has some drawbacks. For example, this approach has a long query latency since a client must listen to the next IR and use the report to conclude whether its cache is valid or not before answering a query. Hence, the average latency of answering a query is the sum of the actual query processing time and half of the IR interval. If the IR interval is long, the delay may not be able to satisfy the requirements of many clients. In most previous IR-based algorithms, when a client needs an invalid cache item, it requests the data from the server, and the server sends the data to the client. Although the approach works fine for some *cold* data items, which are not cached by many clients, it is not effective for *hot* data. For example, suppose a data item is frequently accessed (cached) by 100 clients, updating the data item once may generate 100 uplink (from the client to the server) requests and 100 downlink (from the server to the client) broadcasts. Obviously, it wastes a large amount of wireless bandwidth and battery energy.

In this paper, we will address the problems associated with the IR-based cache invalidation strategies. First, we propose techniques to reduce the query latency. With the proposed techniques, a small fraction of the essential information related to cache invalidation is replicated several times within an IR interval, and hence the client can answer a query without waiting until the next IR. Then, we propose techniques to efficiently utilize the broadcast bandwidth by intelligently broadcasting the data requested by clients. Clients can intelligently retrieve the data which will be accessed in the near future. As a result, most unnecessary unlink requests and downlink broadcasts can be avoided. Detailed simulation experiments are carried out to evaluate our proposed methodology. Compared to the previous IR-based algorithms, our algorithm can significantly improve the throughput (the number of queries served per IR interval) and reduce the query latency, the number of uplink request, and the broadcast bandwidth requirements.

The rest of the paper is organized as follows. Section 2 presents the IR-based cache invalidation model. In Section 3, we propose techniques to reduce the query latency and improve the wireless bandwidth utilization. Section 4 evaluates the performance of IR-based cache invalidation algorithms. Section 5 concludes the paper.

2. THE IR-BASED CACHE INVALIDATION MODEL

In a mobile computing system, the geographical area is divided into small regions, called cells [5]. Each cell has a *base station* (BS) and a number of *mobile terminals* (MTs). Inter-cell and intra-cell communications are managed by the BSs. The MTs communicate with the BS by wireless links. An MT can move within a cell or between cells while retaining its network connection. An MT can either connect to a BS through a wireless communication channel or disconnect from the BS by operating in the *doze* (power save) mode.

There are a set of database servers; each covers one or more cells. We assume that the database is updated only by the server. The database is a collection of N data items: $d_1, d_2, \cdots, d_n$, and is fully replicated at each server. A data item is the basic unit for update and query. MTs only issue simple requests to read the most recent copy of a data item. There may be one or more processes running on an MT. These processes are referred to as clients (we use the terms MT and client interchangeably). In order to serve a request sent from a client, the BS needs to communicate with the database server to retrieve the data items. The BS may also use cache techniques. Since the communications between the database servers and the BSs are through wired links, we assume traditional techniques [10, 13] can be used to maintain cache consistency. Since the communication between the BS and the database server is transparent to the clients (i.e., from the client point of view, the BS is the same as the database server), we use the terms BS and server interchangeably.

Frequently accessed data items are cached on the client side. To ensure cache consistency, the server broadcasts invalidation reports (IRs) every L seconds. The IR consists of the current timestamp T_i and a list of tuples (d_x, t_x) such that $t_x > (T_i - w * L)$, where d_x is the data item id, t_x is the most recent update timestamp of d_x, and w is the invalidation broadcast window size. In other words, IR contains the update history of the past w broadcast intervals. Every client, if active, listens to the IRs and invalidates its cache accordingly. To answer a query, the client listens to the next IR and uses it to decide whether its cache is valid or not. If there is a valid cached copy of the requested data item, the client returns the item immediately. Otherwise, it sends a query request to the server through the uplink. As shown in Figure 1, when a client receives a query between T_{i-1} and T_i, it can only answer the query after it receives the next IR at T_i. Hence, the average latency of answering a query is the sum of the actual query processing time and half of the IR interval.

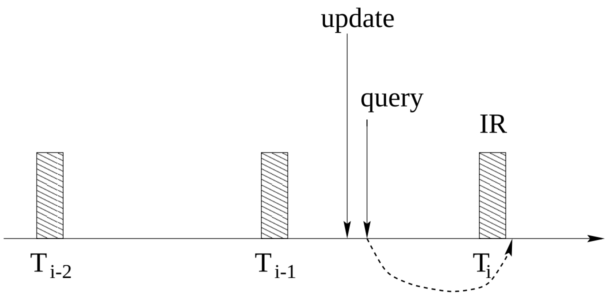

Figure 1: The IR-based cache invalidation model

In order to save energy, an MT may power off most of the time and only turn on during the IR broadcast time. Moreover, an MT may be in the power off mode for a long time and it may miss some IRs. Since the IR includes the history of the past w broadcast intervals, the client can still validate its cache as long as the disconnection time is shorter than $w * L$. However, if the client disconnects longer than $w * L$, it has to discard the entire cached data items since it has no way to tell which parts of the cache are valid. Since the client may need to access some data items in its cache, discarding the entire cache may consume a large amount of wireless bandwidth in future queries. As discussed earlier, algorithms such as the BS algorithm [9] are proposed to deal with the long disconnection problem.

3. THE PROPOSED CACHE INVALIDATION ALGORITHM

In this section, we present our IR-based algorithm. Different from previous IR-based algorithms [1, 6, 9, 16], which concentrate on solving the long disconnection problem, our algorithm concentrates on reducing the query latency and efficiently utilize the broadcast bandwidth. The proposed IR-based algorithm is independent of the cache invalidation strategies, and it can be based on any previous IR-based algorithm [1, 6, 9, 16] to deal with the long disconnection problem. To simplify the presentation, we use the TS scheme as our base algorithm, and show how the TS scheme can be modified to a new scheme which has low query latency and high throughput. Certainly, if the base algorithm is changed to the BS algorithm, the proposed algorithm will be able to tolerate long disconnections.

3.1 Reducing The Query Latency

We use a technique similar to the $(1, m)$ indexing [2, 7, 8] to reduce the query latency. The $(1, m)$ indexing was proposed to reduce the access latency during data broadcasting. In this scheme, a complete data index for the broadcast is repeated every $(\frac{1}{m})^{th}$ of the broadcast. In other words, the entire index occurs m times during a broadcast, where m is a parameter. In this way, a client only needs to wait at most $(\frac{1}{m})^{th}$ of the broadcast interval before getting the data index information. In the proposed algorithm, the index is replaced by the IR, i.e., the IR is replicated m times, and a client only needs to wait at most $(\frac{1}{m})^{th}$ of the IR interval before answering a query. Hence, the query latency can be reduced to $(\frac{1}{m})^{th}$ of the latency in the previous schemes when the query processing time is not considered.

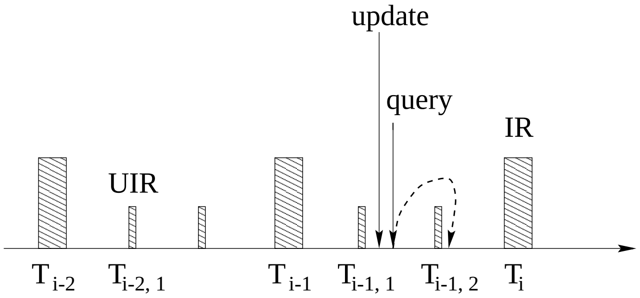

Figure 2: Reducing the query latency by replicating UIRs

Removing the redundant information in the IR: In order to support long disconnections, the IR contains many update history information. For example, in the TS strategy, the IR contains the update history of the past w broadcast intervals. In the BS strategy, the IR contains the update information of the whole database. Replicating the complete IR m times may consume a large amount of broadcast bandwidth. In order to save the broadcast bandwidth, we introduce the concept of *updated invalidation report* (UIR), which contains the data items that have been updated after the last IR has been broadcasted. More formally, the UIR consists of the previous IR timestamp and a list of (d_x, t_x) such that $t_x > T_i$, where T_i is the timestamp of the last IR. Instead of replicating the complete IR, the server inserts $(m - 1)$ UIRs into each IR interval. Since the UIR does not have the update history information, it saves a large amount of broadcast bandwidth.

The idea of the proposed technique can be further explained by Figure 2. In Figure 2, $T_{i,k}$ represents the time of the k^{th} UIR after the i^{th} IR. When a client receives a query between $T_{i-1,1}$ and $T_{i-1,2}$, it can answer the query at $T_{i-1,2}$ instead of T_i. Thus, to answer a query, the client only needs to wait for the next UIR or IR, whichever arrives earlier. Based on the received UIR or IR, the client checks whether its cache is still valid or not. If there is a valid cached copy of the requested data item, the client returns the data item immediately. Otherwise, it sends a query request to the server through the uplink. Since the contents of the UIR depend on the previous IR, each client has to receive the previous IR in order to use the UIR to validate its local cache. In other words, if a client missed the last IR when it receives an UIR (by comparing the timestamp associated with the UIR and the timestamp of the last received IR), it cannot answer any query until it receives the next IR, and use the received IR to validate its cache.

Reducing the timestamp overhead of the UIR: Since only data items that have been updated after the last IR are included in the UIR, the timestamps (associated with the data *id*s) can be removed to save bandwidth. Thus, at interval time $T_{i,k}$, $UIR_{i,k}$ can be constructed as follows:

$$UIR_{i,k} = \{d_x \mid (d_x \in D) \wedge (T_{i,0} < t_x \leq T_{i,k})\} \quad (0 < k < m)$$

Due to the use of UIR to reduce query latency, things are complicated. For example, a client may request an updated data item d_x during one UIR interval $T_{i,j}$. Just after the client gets the current version of the data from the server, it queries the same data after a short time. When the next UIR $T_{i,k} (k > j)$ arrives, the client finds that d_x is included in the UIR, and it knows that d_x has been changed after the previous IR. Since there is no timestamp associated with d_x, the client does not know whether the data has been updated after $T_{i,j}$, and hence it has to request the data from the server again. However, the data may not have been modified since last update; in other words, the client already has the current version of the data. One solution to deal with this kind of *false alarm* is as follows: whenever an updated data has been queried during the last IR interval, the timestamp of this data item is broadcasted with the data item *id*. Thus, in UIR, some data items are broadcasted with timestamp, others are not. However, we do not want to apply this solution because of its complexity and extra overhead. Also, the chance of false alarms is very rare, especially when we apply the techniques presented next.

3.2 Efficiently Utilize The Broadcast Bandwidth

As explained before, in most previous IR-based algorithms, updating a hot data may generate many unnecessary uplink requests and downlink broadcasts, which wastes a large amount of wireless bandwidth and battery energy. Since it is very difficult (if it is not impossible) for a stateless server to find out which data is the hot data and which one is the cold data, we propose to use the following approach to efficiently utilize the broadcast bandwidth. When the server receives a data request, it does not reply the request immediately. Instead, it saves these data *id*s in a list called L_{bcast}. After broadcasting the next IR, the server broadcasts the *id* list of the data items (L_{bcast}) that have been requested during the last IR interval. Then, it broadcasts the data items whose *id*s are in the *id* list L_{bcast}. Each client should always listen to the IR if it is not disconnected. At the end of the IR, the client downloads the *id* list L_{bcast}. For each data *id* in L_{bcast}, the client checks whether it has requested the server for the data or the data becomes an invalid cache entry due to server update[1]. If any of the two conditions is satisfied, it is better for the client to download the current version of the data item since the data will be broadcasted. If the client does not download the data, it may have to send another request to the server, and the server has to broadcast the data again in the near future.

The advantage of the stateless server approach depends on how hot the requested data is. Let us assume that a data item is frequently accessed (cached) by n clients. If the server broadcasts the data after it receives a request from one of these clients, the saved uplink and downlink bandwidth can be up to a factor of n when the data is updated. However, this approach may have two negative effects: i) if the client does not need the data in the future, downloading the data may waste some battery power. ii) if there is a cache miss, the client cannot get the requested data from the server until the next IR, which increases the query latency (but still shorter than the previous IR-based algorithms). However, considering the cache locality and the saved uplink and downlink bandwidth, we believe (and the simulation results also verify) that the benefits should outweigh the disadvantages. Note that the proposed scheme does not waste any bandwidth, since the server only broadcasts the data when the data has been requested by some clients. If the downloaded updated version is accessed in future queries, these queries will have low latency since the queries can be served from the local cache.

One important reason for the server not to serve requests until the next IR interval is due to energy consumption. In our scheme, a client can go to sleep most of the time, and only wakes up during the IR and L_{blist} broadcast time. Based on L_{blist}, it checks whether there is any interested data that will be broadcasted. If not, it can go to sleep and only wakes up at the next IR. If so, it can go to sleep and only wakes up at that particular data broadcast time. For most of the server initiated cache invalidation schemes, the server needs to send the updated data to the clients immediately after the update, and the clients keep awake to get the updated data. Here we tradeoff some delay for more battery energy. Due to the use of UIR, the delay tradeoff is not that significant; most of the time (cache hit), the delay can be reduced by a factor of m, where $(m - 1)$ is the number of replicated UIRs within one IR interval. Even in the worst case (for cache miss), our scheme has the same query delay as the previous IR-based schemes, where the clients cannot serve the query until the next IR. To satisfy time constraint applications, we may apply *priority requests* as follows: when the server receives a priority request, it serves the request immediately instead of waiting until the next IR interval. Since the server serves most of the data requests in the next IR interval, the probability of false alarms (defined in Section 3.1) is very low. For simplicity, we do not implement priority requests in the proposed algorithm, and there is no false alarm.

[1] The client may have a large probability to access the invalidated cache entry in the near future considering cache locality.

3.3 The Algorithm

This subsection presents the formal description of the algorithm which includes a server algorithm and a client algorithm. As explained before, the algorithm is based on the TS scheme. Since the proposed techniques do not depend on a particular algorithm, the proposed algorithm can also be modified by enhancing the capability of dealing with long disconnections.

3.3.1 *The Algorithm at The Server*

The server is responsible to construct the IR and UIR at predefined time interval. It is possible that an IR or UIR time interval reaches while the server is still broadcasting a packet. We use a scheme similar to the beacon broadcast in IEEE 802.11 [3], where the server defers the IR or UIR broadcast until it finishes the current packet transmission. However, the next IR or UIR should be broadcasted at its original scheduled time. The formal description of the algorithm at the server is as follows.

Notations:

- L, w, d_x, t_x: defined before.
- D: the set of data items.
- m: $(m-1)$ is the number of replicated *UIR*s within one IR interval.
- $T_{i,k}$: represents the time of the k^{th} UIR after the i^{th} IR.
- L_{data}: an *id* list of the data items that a client requested from the server.
- L_{bcast}: an *id* list of the data items that the server received in the last IR interval. Initialized to be empty.

```
(A) At interval time $T_i$, construct $IR_i$ as follows:
    $IR_i = \{< d_x, t_x >| (d_x \in D) \wedge (T_i - L*w < t_x \leq T_i)\}$;
    Broadcast $IR_i$ and $L_{bcast}$;
    for each $d_x \in L_{bcast}$ do
        Broadcast data item $d_x$;
        Execute Step B if the UIR interval reaches.
    $L_{bcast} = \emptyset$;
(B) At interval time $T_{i,k}$, construct $UIR_{i,k}$ as follows:
    $UIR_{i,k} = \{d_x \mid (d_x \in D) \wedge (T_{i,0} < t_x \leq T_{i,k})\}$
                                        $(0 < k < m-1)$
(C) Receives a $request(L_{data})$ from client $C_j$:
    $L_{bcast} = L_{bcast} \cup L_{data}$.
```

3.3.2 *The Algorithm at The Client*

The client validates its cache based on the received IR or UIR. If the client missed the previous IR, it has to wait for the next IR. In the algorithm, we assume that the client only sends a new query after its previous query has been served. In each query, the client may need to access multiple data items, and then it may send a request to the server to ask for multiple data items. The formal description of the algorithm at the client is as follows.

Notations:

- $Q_i = \{d_x \mid d_x$ has been queried before $T_i\}$.
- $Q_{i,k} = \{d_x \mid d_x$ has been queried in the interval $[T_{i,k-1}, T_{i,k}]\}$.
- t_x^c: the timestamp of cached data item d_x.
- T_l: the timestamp of the last received IR.
- L_{data}: an *id* list of the data items that a client requested from the server. Initialized to be empty.

```
(A) When a client $C_j$ receives $IR_i$ and $L_{bcast}$:
    if $T_l < (T_i - L*w)$
    then drop the entire cache or go uplink to verify the
        cache (or use other techniques to deal with long
        disconnection);
    for each data item $< d_x, t_x^c >$ in the cache do
        if $((d_x, t_x) \in IR_i) \wedge (t_j^c < t_x)$
        then invalidate $d_x$;
    for each $d_x \in L_{bcast}$ do
        if $(d_x \in L_{data})$
        then download $d_x$ into local cache;
            Use $d_x$ to answer the previous query;
        if $d_x$ is an invalid cache item
        then download $d_x$ into local cache;
    $T_l = T_i$; if $(Q_i \neq \emptyset)$ then query $(Q_i)$.
(B) When a client receives a $UIR_{i,k}$:
    if missed $IR_i$ then break;
                    /* wait for the next IR */
    for each data item $< d_x, t_x^c >$ in the cache do
        if $(d_x \in UIR_{i,k})$
        then invalidate $d_x$;
    if $(Q_{i,k} \neq \emptyset)$ then query $(Q_{i,k})$.
(C) Procedure query(Q)
    $L_{data} = \emptyset$;
    for each $d_x \in Q$ do
        if $d_x$ is a valid entry in the cache
        then use the cache's value to answer the query;
        else $L_{data} = L_{data} \cup d_x$;
    send $request(L_{data})$ to the server.
```

3.4 An Enhancement

Some implementation techniques can be used to further improve the performance. Since the timestamp has a very large overhead, we can associate one timestamp with those data items that have been updated in the same IR interval. More formally, IR can be changed as follows (Notations are defined in the algorithm):

$$IR_i = \{< T_k, D >| (0 < i - w < k \leq i) \\ \wedge D = \{d_x \mid T_{k-1} < t_x \leq T_k\}\}$$

The entry $< T_k, D >$ is used to represent those data items that have been updated in the $(k-1)^{th}$ IR interval. As a result, at each new IR interval, the server only needs to add a new constructed entry to the IR and remove the oldest IR (which is out of the broadcast window) from the IR. The client only needs to strip out those entries that it has not received after the last IR interval (saved in T_l), and restore the IR to the format defined in the algorithm. As a result, the client can save power since it only needs to receive part of the IR. For example, suppose the window size is w. If the client did not miss the report IR_{i-1}, it only needs to download the last $< T_i, D >$ instead of from $< T_i, D >$ to $< T_{i-w}, D >$, which may reduce the download time (power on time) by a factor of w. The following describes how the client strips out the necessary IR information.

```
$IR' = \emptyset$;
for $k = (l+1)$ to $i$ do
    for each $< T_k, D > \in IR_i$ do
        for each $d_x \in D$ do
            $IR' = IR' \cup < d_x, T_k >$;
$IR_i = IR'$.
```

4. PERFORMANCE EVALUATION

Most existing IR-based cache invalidation algorithms [1, 6, 9, 16] are proposed to deal with the long disconnection problem. Since they are based on the TS algorithm [1], without considering the long disconnection problem, these algorithms have similar performance to the TS algorithm. Since our major concern is not to deal with the long disconnection problem, we only compare the performance of the TS algorithm and the proposed algorithm.

4.1 The Simulation Model and System Parameters

In order to evaluate the efficiency of various invalidation algorithms, we develop a simulation model which is similar to that employed in [6, 9]. It consists of a single server that serves multiple clients. The database can only be updated by the server whereas the queries are made on the client side. There are 1000 data items in the database, which are divided into two subsets: the *hot* data subset and the *cold* data subset. The hot data subset includes data items from 1 to 50 (out of 1000 items) and the cold data subset includes the remaining data items of the database. Clients have a large probability (80%) to access the data in the hot set and a low probability (20%) to access the data in the cold set. The server uses 32 bits to represent a timestamp and a data *id*. Including message header overhead, each data item has 1024 bytes.

The Server: The server broadcasts IRs (and UIRs in our algorithm) periodically to the clients. The server assigns the highest priority to the IR (or UIR) broadcasts, and equal priorities to the rest of the messages. This strategy ensures that the IRs (or UIRs) can always be broadcasted over the wireless channels with the broadcast interval specified by the parameter L (or $\frac{L}{m}$). All other messages are served on a FCFS (first-come-first-serve) basis. It is possible that an IR or UIR time interval reaches while the server is still in the middle of broadcasting a packet. We use a scheme similar to the beacon broadcast in IEEE 802.11 [3], where the server defers the IR or UIR broadcast until it finishes the current packet transmission. However, the next IR or UIR should be broadcasted at its originally scheduled time interval. To simplify the simulation, the IR interval L is set to be $20s$. The UIR is replicated 4 time ($m = 5$) within each IR interval.

The server generates a single stream of updates separated by an exponentially distributed update inter-arrival time. All updates are randomly distributed inside the hot data subset and the cold data subset, whereas 33.3% of the updates are applied to the hot data subset. In the experiment, we assume that the server processing time (not data transmission time) is negligible, and the broadcast bandwidth is fully utilized for broadcasting IRs (and UIRs) and serving clients' data requests.

The client: Each client generates a single stream of read-only queries. Each new query is generated following an exponentially distributed time. The client processes generated queries one by one. If the referenced data items are not cached on the client side, the item *ids* are sent to the server for fetching the data items. Once the requested data items arrive on the channel, the client brings them into its cache. Client cache management follows the LRU replacement policy, but there are some differences between the TS algorithm and our algorithm. In the TS algorithm, since the clients will not use the invalid cache items, the invalidated cache items are first replaced. If there is no invalid cache item, LRU is used to replace the oldest valid cache item. In our algorithm, if there are invalid data items, the client replaces the oldest invalid item. If there is no invalid cache item, the client replaces the oldest valid cache item. The difference is due to the fact that the client in our algorithm can download data from the broadcast channel.

As mentioned before, the major concern of this paper is to reduce the query latency and improve the bandwidth utilization. To simplify the presentation and simulation, we do not model disconnections in this paper. Since the proposed techniques are independent of any particular algorithms, the disconnection problem can be solved by applying previous techniques [1, 9, 16] to our algorithm. Most of the system parameters are listed in Table 1.

Number of clients	100
Database size	1000 items
Data item size	1024 bytes
Broadcast interval (L)	20 seconds
Broadcast bandwidth	10000 bits/s
Cache size	50 to 300 items
Mean query generate time (T_{query})	25s to 300s
Broadcast window (w)	10 intervals
UIR replicate times ($m-1$)	4 (5 - 1)
Hot data items	1 to 50
Cold data items	remainder of DB
Hot data access prob.	0.8
Mean update arrival time (T_{update})	1s to 10000s
Hot data update prob.	0.33

Table 1: Simulation parameters

4.2 Simulation Results

Experiments were run using different workloads and system settings. The performance analysis presented here is designed to compare the effects of different workload parameters such as mean update arrival time, mean query generate time, and system parameters such as cache size on the relative performance of the TS algorithm and our algorithm. The performance is measured by the query delay, the number of uplink requests per IR interval, and the throughput (the number of queries served per IR interval). Note that minimizing the number of uplink requests is a desirable goal as clients in a mobile environment have limited battery power and transmitting data requires a large amount of power.

Since the client caches are only partially full at the initial stage, the effectiveness of the invalidation algorithms may not be truly reflected. In order to get a better understanding of the true performance for each algorithm, we collect the result data only after the system becomes stable, which is defined as the time when the client caches are full. For each workload parameter (e.g., the mean update arrival time or the mean query generate time), the mean value of the measured data is obtained by collecting a large number of samples such that the confidence interval is reasonably small. In most cases, the 95% confidence interval for the measured data is less than 10% of the sample mean.

4.2.1 The Cache Hit Ratio

The performance metrics such as the query delay, the throughput, and the uplink cost have strong relation with the cache hit ratio. For example, if the cache hit ratio is high, the query delay can be reduced since the client can process most of the queries locally and does not need to request the data from the server. To help understand the simulation results, we first look at the cache hit ratio difference between the TS algorithm and our algorithm.

The left graph of Figure 3 shows the cache hit ratio as a function of the number of clients. As can be seen, the cache hit ratio of our algorithm increases as the number of clients increases, but the cache hit ratio of the TS algorithm does not change with the number of clients. When the number of clients in our algorithm drops to 1, the cache hit ratio of our algorithm is similar to the TS algorithm. In the TS algorithm, a client only downloads the data that it has requested from the server. However, in our algorithm, clients also download the data which may be accessed in the near future. Considering 100 clients, due to server update, one hot data item may be changed by the server, and the clients may have to send requests to the server and download the data from the server. In the TS algorithm, it may generate 100 cache misses if all clients need to access the updated data. In our algorithm, after a client sends a request to the server, other clients can download the data. In other words, after one cache miss, other clients may be able to access the data from their local cache. Certainly, this ideal situation may not always occur, especially when the cache size is small or the accessed data is cold. However, as long as some downloaded data items can be accessed in the future, the cache hit ratio of our algorithm will be increased. Due to cache locality, a client has a large chance to access the invalidated cache items in the near future, so downloading these data items in advance should be able to increase the cache hit ratio. As the number of clients decreases, clients have less opportunity to download data requested by others, and hence the cache hit ratio decreases. This explains why our algorithm has similar cache hit ratio when the number of clients drops to 1. The right side of Figure 3 shows the cache hit ratio under different cache sizes when the number of clients is 100. Based on the above explanation, it is easy to see that the cache hit ratio of our algorithm is always higher than that of the TS algorithm for one particular cache size (cache size is 50 items, 100 items, or 300 items),

From Figure 3 (the right figure), we can see that the cache hit ratio grows as the cache size increases. However, the growing trend is different between the TS algorithm and our algorithm. For example, in the TS algorithm, when the update arrival time is $1s$, the cache hit ratio does not have any difference when the cache size changes from 50 data items to 300 data items. However, in our algorithm, under the same situation, the cache hit ratio increases from about 40% to 58%. In our algorithm, clients may need to download interested data for future use, so a large cache size may increase cache hit ratio. However, in the TS algorithm, clients do not download data items that are not addressed to them. When the server updates data frequently, increasing the cache size does not help. This explains why different cache size does not affect the cache hit ratio of the TS algorithm when $T_{update} = 1s$.

As shown in Figure 3 (the right figure), the cache hit ratio drops as the update arrival time decreases. However, the cache hit ratio of the TS algorithm drops much faster than our algorithm. When the update arrival time is $10000s$, both algorithms have similar cache hit ratio for one particular cache size. With $cache = 300\ items$, as the update arrival time reaches $1s$, the cache hit ratio of our algorithm still keeps around 58% whereas the cache hit ratio of the TS algorithm drops to near 0. This can be explained as follows. When the update arrival time is very low (e.g., $1s$), most of the cache misses are due to hot data access; when the update arrival time is very high (e.g., $10000s$), most of the cache misses are due to cold data access. Since our algorithm is very effective to improve cache performance when accessing hot data, the cache hit ratio of our algorithm can be significantly improved when the update arrival time is low. However, as the mean update arrival time drops further ($T_{update} < 1s$), the cache hit ratio of our algorithm drops much faster than before. At this time, the hot data changes so fast that the downloaded hot data may be updated before the client can use it, and hence failing to improve the cache hit ratio. Note that when the update arrival time is very high, the cache performance depends on the LRU policy, and it is very difficult to further improve the cache hit ratio except increasing the cache size.

4.2.2 The Query Delay

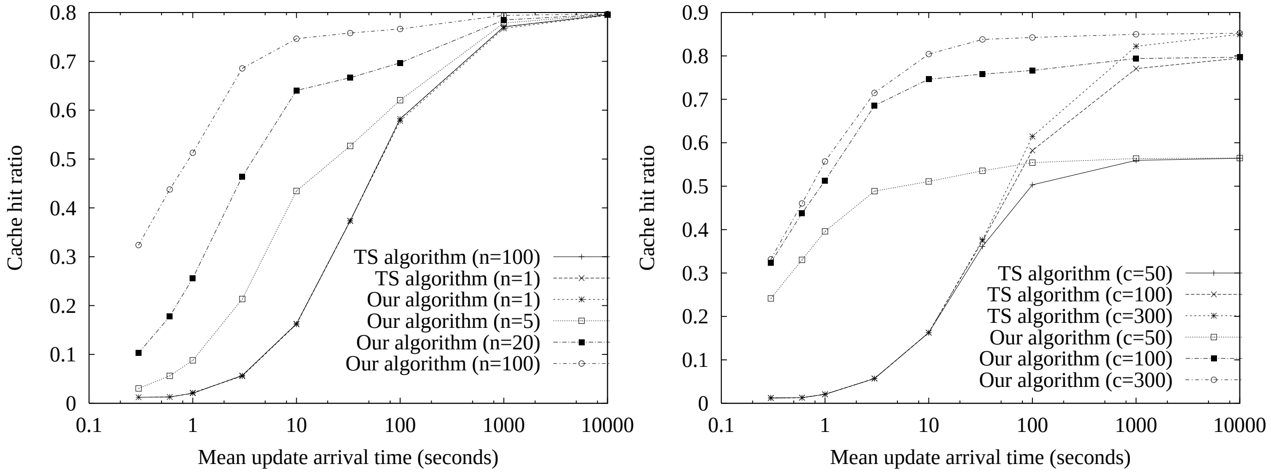

Figure 3: A comparison of the cache hit ratio ($T_{query} = 100s$). The left figure shows the cache hit ratio as a function of the number of clients when the cache size is 100 items. The right figure shows the cache hit ratio under different cache sizes when the number of clients is 100.

We measure the query delay as a function of the mean query generate time and the mean update arrival time. As shown in Figure 4, our algorithm significantly outperforms the TS algorithm.

As explained before, each client generates queries according to the mean query generate time. The generated queries are served one by one. If the queried data is in the local cache, the client can serve the query locally; otherwise, the client has to request the data from the server. If the client cannot process the generated query due to waiting for the server reply, it queues the generated queries. Since the broadcast bandwidth is fixed, the server can only transmit a limited amount of data during one IR interval, and then it can only serve a maximum number (α) of queries during one IR interval. If the server receives more than α queries during one IR interval, some queries are delayed to the next IR interval. If the server receives more than α queries during each IR interval, many queries may not be served, and the query delay may be out of bound. The left figure of Figure 4 shows the query delay as a function of the mean query generate time with $T_{update} = 10s$ and $cache = 100\ items$. When the query generate time is lower than $70s$ (e.g., $60s$), the query delay of the TS algorithm becomes infinite long (cannot see from Figure 4). However, even when the query generate time reaches $30s$, the query delay of our algorithm is still less than $10s$. This is due to the fact that the cache ratio of our algorithm is still high (see Figure 3) and a large number of queries can be served locally. Thus, the number of requests sent to the server is still lower than α. Note that the query delay of our algorithm can also grow out of bound if the query generate drops further (e.g., $20s$).

In Section 3.1, we mentioned that the query delay can be reduced by a factor of m if the IRs are replicated m time during one IR interval. However, this is only true if the cache hit ratio can reach 100%. Since the cache hit ratio cannot be 100%, the query delay can never be reduced by a factor of m. In our algorithm, without considering the priority request, a client cannot answer the query until the next IR interval in case of cache miss. Therefore, during a cache miss, the TS algorithm and our algorithm have the same query delay. However, in case of cache hit, our algorithm can reduce the query delay by a factor of m. As shown on the right graph of Figure 4, as the mean update arrival time increases, the cache hit ratio increases and the query delay decreases. Since our algorithm has high cache hit ratio than the TS algorithm, the query delay of our algorithm is shorter than the TS algorithm. For example, with $T_{update} = 10000s$, our algorithm reduces the query delay by a factor of 3 compared to the TS algorithm. Although the cache ratio of the TS algorithm is more than doubled from $T_{update} = 10s$ to $T_{update} = 33s$, the query delay of the TS algorithm does not drop too much (from 18.7s to 16.1s). This can be explained by the fact that a client in the TS algorithm cannot answer a query until it receives the next IR, and hence the average query delay is at least half of the IR interval even when the client has a valid cache copy. Since the query generate time is exponentially distributed, multiple queries may arrive at a client during one IR interval. The client only serves the query one by one. In case of cache miss, due to queue effect, the query delay may be longer than the IR interval $20s$ (as shown in Figure 4).

4.2.3 The Number of Queries Served Per IR Interval

As explained in the last subsection, due to the limited broadcast bandwidth, the server can only serve a maximum number (α) of client requests during one IR interval. However, the throughput (the number of queries served per IR interval) may be larger than α since some of the queries can be served by accessing the local cache. Since our algorithm has higher cache hit ratio than the TS algorithm, our algorithm can serve more queries locally, and the clients send less requests to the server. Thus, although α is the same for two algorithms, our algorithm has a higher throughput than the

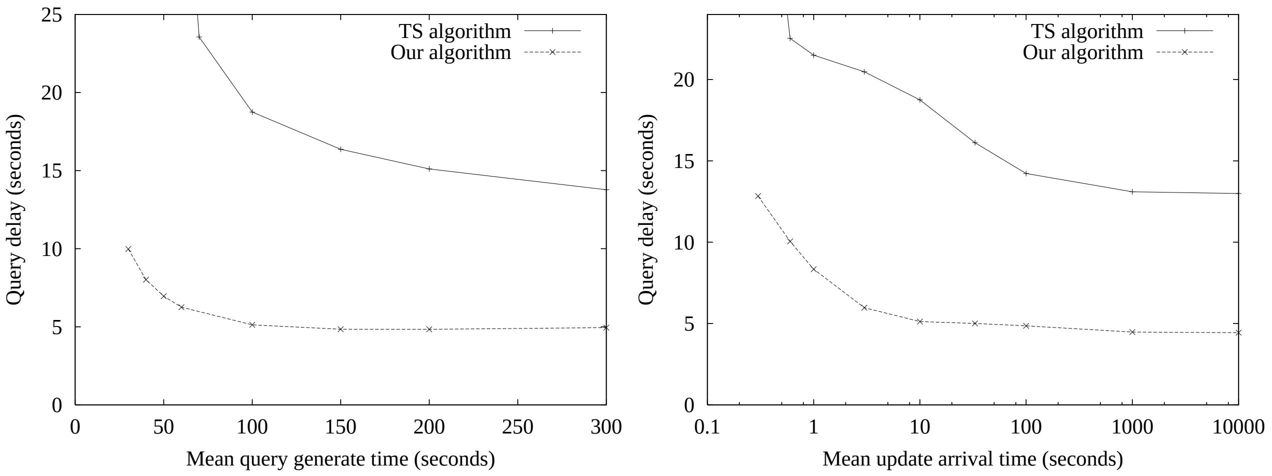

Figure 4: A comparison of the query delay. The left figure shows the query delay as a function of the mean query generate time ($T_{update} = 10s$, $cache = 100\ items$). The right figure shows the query delay as a function of the mean update arrival time ($T_{query} = 100s$, $cache = 100\ items$)

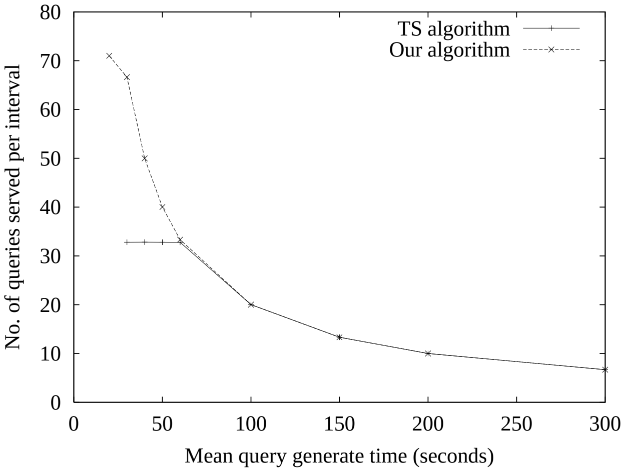

Figure 5: The number of queries served per IR interval ($T_{update} = 10s$, $cache = 100\ items$)

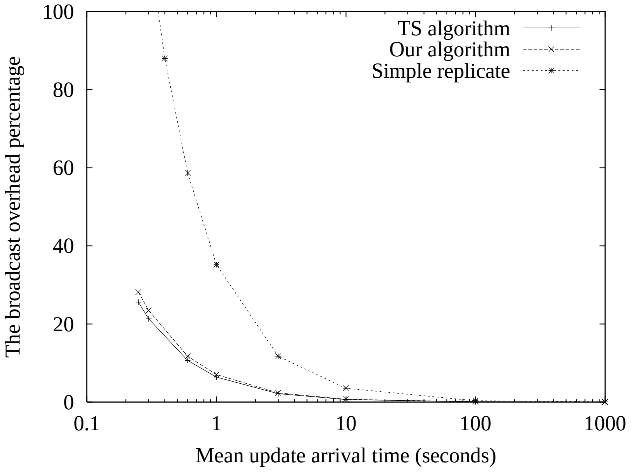

Figure 6: The broadcast overhead as a function of the update arrival time ($T_{query} = 100s$, $cache = 100\ items$)

TS algorithm. For example, as shown in Figure 5, when the query generate time reduces to $30s$, the number of requests in the TS algorithm is larger than α, and some queries cannot be served. As a result, the throughput of the TS algorithm remains at 32 whereas the throughput of our algorithm reaches 70. In the TS algorithm, since the broadcast channel has already been fully utilized when $T_{query} = 60s$, further reducing the query generate time does not increase the throughput. When the query generate time is low, the broadcast channel has enough bandwidth to serve client requests, and hence both algorithms can serve the same number of queries (although they have difference query latency).

4.2.4 The Broadcast Overhead

Let T_{ir} represent the average time that the server spends on broadcasting the IRs within one IR interval. Let T_{uir} represent the average time that the server uses to broadcast the UIRs within one IR interval (T_{uir} is 0 in the IR algorithm). The broadcast overhead percentage is $\frac{T_{ir}+T_{uir}}{L}$. Figure 6 compares the broadcast overhead of our algorithm to the IR algorithm and the *simple replicate algorithm*, which simply replicates the IR $m(m = 4)$ times within each IR interval (i.e., the broadcast interval (L) changes to $\frac{20}{4+1} = 4s$). As can be seen, the simple replicate approach has the highest broadcast overhead and the TS algorithm has the lowest broadcast overhead. Due to the use of UIR techniques (see Section 3.1), the broadcast overhead of our algorithm is slightly higher than the TS algorithm, but far lower than the simple replicate algorithm. For example, When $T_{update} = 0.35s$, in the simple replicate approach, the server cannot answer clients' queries since all available bandwidth are used to broadcast IRs. However, in our algorithm, the broadcast overhead is only about 20%.

4.2.5 The Number of Uplink Requests

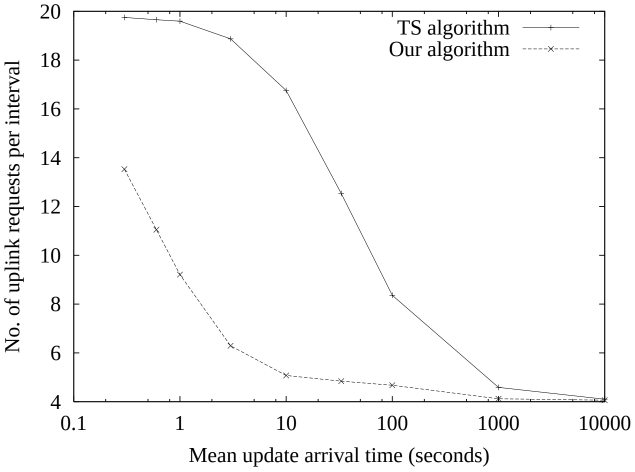

Figure 7: The number of uplink requests per IR interval ($T_{query} = 100s$, $cache = 100\ items$)

Figure 7 shows the uplink cost of both algorithms. Since our algorithm has lower cache miss rate than the TS algorithm and clients only send uplink requests when there are cache misses, our algorithm has lower uplink cost compared to the TS algorithm. It is interesting to find that both algorithms have similar uplink cost when the mean update arrival time is very high (e.g., $10000s$), but a significant difference when the mean update arrival time is $10s$. From Figure 3, we can find that both algorithms have similar cache miss ratio (1 - cache hit ratio) when $T_{update} = 10000s$, but a significant difference when $T_{update} = 10s$. As shown in Figure 7, our algorithm can cut the uplink cost by a factor of 3 (with $T_{update} = 10s$), and hence the clients can save a large amount of energy and bandwidth. When the update arrival time is smaller than $1s$, the uplink cost of the TS algorithm does not increase, but the uplink cost of our algorithm increases much faster than before. This can be explained by the fact that the cache hit ratio of the TS algorithm already drops to near 0 when $T_{update} = 1s$, but the cache ratio of our algorithms continues dropping when $T_{update} < 1s$.

5. CONCLUSIONS

IR-based cache invalidation have received considerable attention due to its scalability. However, most of the previous IR-based algorithms [1, 6, 9, 16] concentrate on dealing with the problem of long disconnections, and not much work has been done to address the drawbacks associated with the IR-based algorithms such as long query delay and low bandwidth utilization. In this paper, we proposed techniques to deal with these problems. In the proposed algorithm, a small fraction of the essential information related to cache invalidation is replicated several times within an IR interval, and hence a client can answer a query without waiting until the next IR. Moreover, the server can intelligently broadcast the data items requested by the clients, while the clients intelligently retrieve the data items which will be accessed in the near future. As a result, most unnecessary unlink requests and downlink broadcasts can be avoided. Simulation results showed that our algorithm can cut the query delay by a factor of 3, and double the throughput compared to the TS algorithm.

In this paper, we only perform simulation studies of the TS algorithm and the proposed algorithm. In order to get a better understanding of the algorithms, analytical model can be built. Based on the analytical model, we can systematically setup system parameters such as m, w, L, according to the system workload. Also, we did not evaluate the performance under client disconnections. As future work, we will extend our algorithm and combine it with previous techniques [1, 6, 9, 16] to deal with client disconnections.

6. REFERENCES

[1] D. Barbara and T.Imielinksi, “Sleepers and workaholics: Caching strategies for mobile environments,” *ACM SIGMOD*, pages 1–12, 1994.

[2] A. Datta, D. Vandermeer, A. Celik, and V. Kumar, “Broadcast Protocols to Support Efficient Retrieval from Databases by Mobile Users,” *ACM Transactions on Database Systems*, 24(1):1–79, March 1999.

[3] The editors of IEEE 802.11, “Wireless LAN Media Access Control (MAC) and Physical Layer (PHY) Specifications,” *802.11 Wireless Standards (http://grouper.ieee.org/groups/802/11)*, 1999.

[4] G. Forman and J. Zahorjan, “The Challenges of Mobile Computing,” *IEEE Computer*, 27(6), April 1994.

[5] D.J. Goodman, “Cellular Packet Communication,” *IEEE Trans. Communication*, 38(8):1272–1280, Aug. 1990.

[6] Q. Hu and D. Lee, “Cache Algorithms based on Adaptive Invalidation Reports for Mobile Environments,” *Cluster Computing*, pages 39–48, Feb. 1998.

[7] T. Imielinksi, S. Viswanathan, and B. Badrinath, “Data on Air: Organization and Access,” *IEEE Transactions on Knowledge and Data Engineering*, 9(3):353–372, May/June 1997.

[8] T. Imielinski, S. Viswanathan, and B. Badrinath, “Energy Efficient Indexing on Air,” *ACM SIGMOD’94*, pages 25–36, 1994.

[9] J. Jing, A. Elmagarmid, A. Helal, and R. Alonso, “Bit-Sequences: An adaptive Cache Invalidation Method in Mobile Client/Server Environments,” *Mobile Networks and applications*, pages 115–127, 1997.

[10] M. Kazar, “Synchronization and Caching Issues in the Andrew File System,” *USENIX Conf.*, pages 27–36, 1988.

[11] W. Lee, Q. Hu, and D. Lee, “A Study on Channel Allocation for Data Dissemination in Mobile Computing Environments,” *ACM/Baltzer Mobile Networks and Applications*, pages 117–129, 1999.

[12] R. Powers, “Batteries for Low Power Electronics,” *Proc. IEEE*, 83(4):687–693, April 1995.

[13] S. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, “Design and Implementation of the Sun Network File System,” *Proc. USENIX Summer Conf.*, pages 119–130, June 1985.

[14] K. Stathatos, N. Roussopoulos, and J. Baras, “Adaptive Data Broadcast in Hybrid Networks,” *Proc. of the 23rd VLDB Conf.*, 1997.

[15] N. Vaidya and S. Hameed “Scheduling DAta Broadcast in Asymmetric Communication Environments,” *ACM/Baltzer Wireless Networks (WINET)*, May 1999.

[16] K. Wu, P. Yu, and M. Chen, “Energy-efficient caching for wireless mobile computing,” *The 20th Intl. Conf. on Data Engineering*, pages 336–345, Feb. 1996.