Mobile Networks and Applications 7, 291–303, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

On Improving the Performance of Cache Invalidation in Mobile Environments

GUOHONG CAO
Department of Computer Science & Engineering, The Pennsylvania State University, University Park, PA 16802, USA

Abstract. Many cache management schemes designed for mobile environments are based on invalidation reports (IRs). However, IR-based approach suffers from long query latency and it cannot efficiently utilize the broadcast bandwidth. In this paper, we propose techniques to address these problems. First, by replicating a small fraction of the essential information related to cache invalidation, the query latency can be reduced. Then, we propose techniques to efficiently utilize the broadcast bandwidth based on counters associated with each data item. Novel techniques are designed to maintain the accuracy of the counter in case of server failures, client failures, and disconnections. Extensive simulations are provided and used to evaluate the proposed methodology. Compared to previous IR-based algorithms, the proposed solution can significantly reduce the query latency, improve the bandwidth utilization, and effectively deal with disconnections and failures.

Keywords: invalidation report, query latency, counter, failure recovery, mobile computing

1. Introduction

With the explosive growth of wireless techniques and mobile devices such as laptops, personal digital assistants, people with battery powered mobile devices wish to access various kinds of services at any time any place. However, existing wireless services are limited by the constraints of mobile environments such as narrow bandwidth, frequent disconnections, and limitations of the battery technology [6]. Thus, mechanisms to efficiently transmit information from the server to a massive number of clients (running on mobile devices) have received considerable attention [2,3,7,9,13,18].

Caching frequently accessed data items 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, classical cache invalidation strategies may not be suitable for mobile environments due to frequent disconnections [6,15] and high mobility of mobile clients. It is difficult for the server to send invalidation messages directly to the clients because they often disconnect to conserve battery power and are frequently on the move. For the clients, querying data servers through wireless links for cache invalidation is much slower than wired links because of the latency of the wireless links. Also, conventional client/server interactions cannot scale to massive numbers of clients due to the narrow bandwidth of the wireless links.

Barbara and Imielinski [2] provide a solution which is suitable for mobile environments. 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, and use them to validate their local cache. 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 major drawbacks. First, there is a long query latency associated with this scheme 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. Second, even though some data items are not cached by any client, the server still includes them in the IR, thereby wasting a significant amount of bandwidth. Third, even though many clients cache the same updated data item, all of them have to query the server and get the data from the server separately. Although the approach works fine for some *cold* data items, which are not cached by many clients, it is not effective for *hot* data items. 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.

We [3] addressed the first problem with a UIR-based approach. In this approach, a small fraction of the essential information (called updated invalidation report (UIR)) 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. However, if there is a cache miss, the client still needs to wait for the data to be delivered. Thus, both issues (query delay and bandwidth utilization) are related to the cache hit ratio. In this paper, we address the problems associated with the IR-based cache invalidation strategies. We use the UIR approach to reduce the query latency, and we propose techniques to improve the cache hit ratio. Instead of passively waiting, clients intelligently prefetch the data that are most likely used in the future. To help clients prefetch the right data and efficiently utilize the broadcast bandwidth, *counters* are used. Novel techniques are designed to maintain the accuracy of the counter in case of server failures, client failures, and disconnections. Extensive simulations are provided and used to evaluate the proposed methodology. Compared to previous IR-based algorithms, the proposed solution can significantly reduce the query latency, improve the bandwidth utilization, and effectively deal with disconnections and failures.

The rest of the paper is organized as follows. Section 2 develops the necessary background. In section 3, we propose techniques to efficiently utilize the wireless bandwidth and maintain the accuracy of the counter in case of server failures, client failures, and disconnections. Section 4 evaluates the performance of our algorithm. Section 5 concludes the paper.

2. Preliminaries

When cache techniques are used, data consistency issues must be addressed. The notion of data consistency is, of course, application dependent. In database systems, data consistency is traditionally tied to the notion of transaction serializability. In practice, however, few applications demand or even want full serializability, and more efforts have gone into defining weaker forms of correctness [12]. In this paper, we use the *latest value* consistency model[1] [1,3,11,14], which is widely used in dissemination-based information systems. In the latest value consistency model, clients must always access the most recent value of a data item. This level of consistency is what would arise naturally if the clients do not perform caching and the server broadcasts only the most recent values of items. When client caching is allowed, techniques should be applied to maintain the latest value consistency.

2.1. The IR-based cache invalidation model

In the IR-based cache invalidation strategy, the server periodically broadcasts invalidation reports (IRs), which indicates the updated data items. Note that only the server can update the data. To ensure cache consistency, every client, if active, listens to the IRs and uses these IRs to invalidate its cache accordingly. To answer a query, the client listens to the next IR and uses it to decide whether its cache is still 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 through the uplink. The server keeps track of the recently updated information and broadcasts an IR every L second. 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 listening to the IR since they cannot switch to power save mode while listening. In the following, we look at two IR-based algorithms.

[1] The Coda file system [16] does not follow the latest value consistency model. It supports a much weaker consistency model to improve performance. However, some conflicts may require manu-configuration and some updated work may be discarded.

2.1.1. The broadcasting timestamp (TS) scheme

The TS scheme was proposed by Barbara and Imielinski [2]. In this scheme, the server broadcasts an IR every L seconds. The IR consists of the current timestamp T_i and a list of tuples (d_x, t_x) such that $t_x > (T_i - w \cdot L)$, 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.

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 to save energy, and hence the client running the MT 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 its disconnection time is shorter than $w \cdot L$. However, if the client disconnects longer than $w \cdot L$, 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 items in its cache, discarding the entire cache may consume a large amount of wireless bandwidth in future queries.

2.1.2. The bit sequences (BS) scheme

In the BS scheme [10], the IR consists of a sequence of bits. Each bit represents a data item in the database. Setting the bit to 1 means that the data item has been updated. The update time of each data item is also included in the IR. To reduce the length of the IR, some grouping methods are used to make one bit coarsely represent several data items. Instead of including one update timestamp for each data item, the BS scheme uses one timestamp to represent a group of data items in a hierarchical manner. Let IR be $\{[B_0, TS(B_0)], \ldots, [B_k, TS(B_k)]\}$ where $B_i = 1$ means that half of the data items from 0 to 2^i at time $TS(B_i)$ have been updated. The clients use the bit sequences and the time-stamps to decide what data items in their local cache should be invalidated. The scheme is very flexible (no invalidation window size is needed) and it can be used to deal with the long disconnection problem by carefully arranging the bit sequence. However, since the IR represents the data of the entire database (half of the recently updated data items in the database if more than half data items have been updated since the initial time), broadcasting the IR may consume a large amount of downlink bandwidth.

Many solutions [7,10,17] are proposed to address the long disconnection problem, and Hu et al. [7] have a good survey of these schemes. Although different approaches [2,10] apply different techniques to construct the IR to address the long disconnection problem, these schemes maintain cache consistency by periodically broadcasting the IR. The IR-based solution is attractive because it can scale to any number of MTs who listen to the IR. However, this solution has long query latency, since the client can only answer the query after it receives the next IR to ensure cache consistency. Hence, the average latency of answering a query is the sum of the actual query processing time and half of the IR interval.

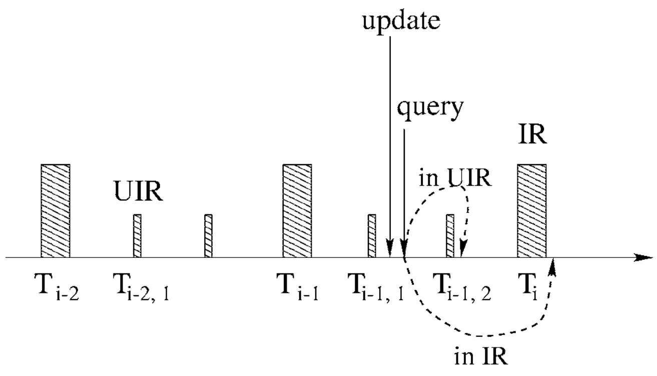


Figure 1. Reducing the query latency by replicating UIRs.

2.2. The UIR-based cache invalidation

In order to reduce the query latency, Cao [3] proposed to replicate the IRs m times; that is, the IR is repeated every $(1/m)$th of the IR interval. As a result, a client only needs to wait at most $(1/m)$th of the IR interval before answering a query. Hence, latency can be reduced to $(1/m)$th of the latency in the previous schemes (when query processing time is not considered).

Since the IR contains a large amount of update history information, replicating the complete IR m times may consume a large amount of broadcast bandwidth. In order to save the broadcast bandwidth, after one IR, $m - 1$ *updated invalidation reports* (UIRs) are inserted within an IR interval. Each UIR only contains the data items that have been updated after the last IR was broadcasted. In this way, the size of the UIR becomes much smaller compared to that of the IR. As long as the client downloads the most recent IR, it can use the UIR to verify its own cache. The idea of the proposed technique can be further explained by figure 1. In figure 1, $T_{i,k}$ represents the time of the kth UIR after the ith IR. When a client receives a query between $T_{i-1,1}$ and $T_{i-1,2}$, it cannot answer the query until T_i in the IR-based approach, but it can answer the query at $T_{i-1,2}$ in the UIR-based approach. Hence, the UIR-based approach can reduce the query latency in case of a cache hit. However, if there is a cache miss, the client still needs to fetch data from the server, which increases the query latency. Next, we propose a cache management algorithm to improve the cache hit ratio and the bandwidth utilization.

3. The counter-based cache invalidation algorithm

In this section, we present our counter-based cache invalidation algorithm to reduce the query latency and improve the bandwidth utilization.

3.1. Efficiently utilize the broadcast bandwidth by prefetching

In most previous IR-based algorithms, updating a hot data item may generate many unnecessary uplink requests and downlink broadcasts, and waste a large amount of wireless bandwidth. We address the problem by asking the clients to prefetch data that may be used in the near future. For example, if a client observes that the server is broadcasting a data item which is an invalid entry[2] of its local cache, it is better to download the data; otherwise, the client may have to send another request to the server, and the server will have to broadcast the data again in the future.

Using broadcast list to save energy. There may be a large number of hot data items. Broadcasting all of them may require a large amount of bandwidth. In our scheme, the server only broadcasts the hot data items which have been updated in the last IR interval. Note that the hot data item is only broadcasted once after the update even though the hot data *id* may appear w times in the IRs.

To save power, clients may only wake up during the IR broadcast time, and then how to prefetch data becomes an issue. As a solution, after broadcasting the IR, the server first broadcasts the *id* list (L_{bcast}) of the data items whose real data will be broadcasted next, and then broadcasts the real data of the data items in the *id* list. Each client should listen to the IR if it is not disconnected. At the end of the IR, the client downloads the *id* list L_{bcast} and finds out when the interested data will come, and wakes up at that time to download the data. With this approach, power can be saved since clients stay in the doze mode most of the time; bandwidth can be saved since the server may only need to broadcast the updated data once.

Relying on counters to identify the hot data. The effectiveness of the prefetch depends on how hot the broadcasted data is. Let us assume a data item is frequently accessed (cached) by n clients. If the server broadcasts the data after it is updated, prefetching the data may save uplink and downlink bandwidth up to a factor of n. Thus, it is very important to identify which data should be included in the *id* list. We address the problem with a *counter-based* scheme. In our scheme, a counter is maintained for each data item. The counter associated with a data item is increased by one when a client requests the data from the server. A client may replace its local cache data by some new data, and the server should decrease the counter associated with the discarded (replaced) cache item. To achieve this, clients need to notify the server which data items have been discarded from their cache. To save bandwidth, clients piggyback the discarded data item *id*s when they send new data requests to the server. When the server receives a request, it decreases the counters associated with the data items that the client will discard, and increases the counters associated with the requested data items.

The counter scheme has two benefits. First, based on the counters, the server can find out which data items are hot and broadcast their updates to the clients. For example, when a data item has a counter larger than a system parameter cnt_α, it is a hot data item and should be broadcasted. Second, in previous IR-based cache invalidation algorithms, the server

[2] We assume cache locality exists. When cache locality does not exist, other techniques should be used to improve the effectiveness of the prefetch.

includes all updated data *id*s to the IR even though some of them are not cached by any client. In the counter-based scheme, when the counter associated with a data item becomes 0, no client is caching the data, and hence the server does not add it to the IR even though the data is updated during the last IR interval. In this way, broadcast bandwidth can be saved.

3.2. Dealing with the problems of client failures and disconnections

Due to client failures and disconnections, the counters that are associated with the data cached by the failed MT, may become larger than expected. Moreover, all counters may grow larger than 0 or even larger than cnt_α after some amount of time and fail to represent the number of accessing clients. Some simple solutions may not work. For example, we can increase cnt_α based on the counter values so that there are not too many data items which have counters larger than cnt_α. If the traffic pattern does not change too much; i.e., hot data is always hot and cold data is always cold, this is a good solution. However, if the traffic pattern changes frequently, this solution may not be able to identify the hot data items. In the following, we provide two approaches to deal with the counter accuracy problem: the stateful server approach and the stateless server approach. Due to the disadvantages associated with the stateless server approach, we only describe it in the appendix.

The stateful server approach. In this approach, the server maintains a *cached item register* (CIR) for each client. The CIR maintains the cached data item *id*s of the client. When the server receives a data request from a client, it updates the relevant counters and the corresponding CIR. When a client sends a request to the server, it signs a lease with the server. The lease specifies a *lease renew time* before which the client should renew the lease with the server. If the client does not renew the lease after the lease renew time, the server assumes that the client has failed. Hence, the server decreases the relevant counters and removes the associated CIR. Note that the lease is automatically renewed if the client sends a new request to the server before the last lease expires.

When a client reconnects after a disconnection time longer than $w \cdot L$, it sends a request with the last received IR timestamp (before disconnection) to the server. Based on the timestamp and the CIR of the client, the server replies with the valid data *id*s if the number of valid *id*s is smaller than the number of invalid data *id*s, or vice versa. During a handoff, the client needs to notify the new base station or server about its old base station, and the new base station will be responsible to transfer the CIR information from the old base station.

Figure 2 shows the structure of the counter-based cache invalidation. The server maintains a CIR for each client. Based on the counter value and the update history, the server constructs and broadcasts IRs periodically. After the IR, the server broadcasts the hot data which has been updated during the last IR interval. Based on the received IR, the clients

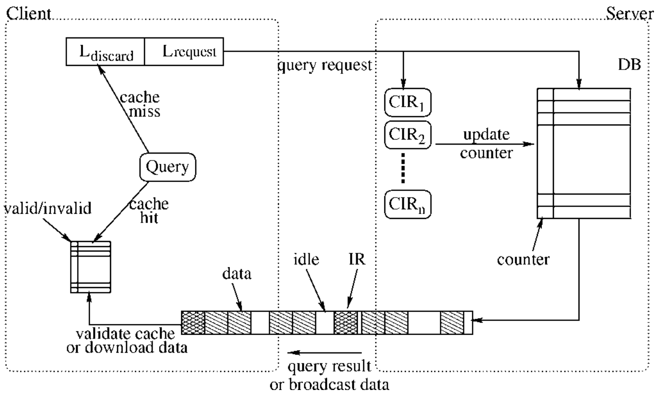


Figure 2. The counter-based cache invalidation.

validate their local cache. In case that the client cannot serve the query from its local cache, it has to send a request to the server, which will send back the requested data. Figures 3 and 4 show the formal descriptions of the server algorithm and the client algorithm, respectively.

3.3. Handling server failures

When the server fails, the CIRs may be lost, and the counter values may not be accurate. As one solution, the CIR and the counter information can be stored on stable storage. By using some checkpointing and recovery schemes [4,5], the recovery time can be reduced at the cost of increased overhead on normal operations. As another option, the server can reconstruct its state by polling its clients. Since the server may lose the information about the clients, and then it may not be able to poll all the clients. One possible solution is to broadcast a *recovery* message to notify clients to report their cache information. However, this approach has two problems. First, since clients need to access the reverse control channel to get the permission to use the uplink channel, it may result in lots of collisions if they respond to the *recovery* message at the same time. Second, due to disconnections, there is no guarantee that all clients will respond the *recovery* and then the server may not be able to collect all the necessary information about the clients. As a result, some items may be cached by the clients, but the server may not know and will not add the data item *id*s to the IR even though the data have been updated.

Due to the problems or overhead associated with the above solutions, we propose to apply a stepwise approach to address the server failure problem. After the server recovers from the crash, it broadcasts a *recovery* message with the timestamp of the recovery. Then, it resets all the counters associated with the data items. During the recovery process, it includes all the data item *id*s in its IR or UIR regardless of the associated counter value. The recovery process terminates after w IR intervals. When the recovery process terminates, all processes should be able to learn about the *recovery* process. If a client disconnects longer than w IR interval, it sends a request to the server during the reconnection, and the server can notify the client about the *recovery* process. For each client, on receiving the broadcasted *recovery* message, it changes its status to

Notations:

- L, w, d_x, t_x: defined before.
- D: the set of data items.
- cnt_x: the counter associated with data d_x.
- cnt_α: the counter threshold. When $cnt_x > cnt_\alpha$, B_x will be set to 1, and the data item will be broadcasted after the IR.
- $lease_j$: the time when the lease of CIR_j expires.
- T_α: the lease time between the server and the client. For simplicity, we use one lease time for all clients.
- L_{bcast}: an *id* list that the server will broadcast after the IR.
  L_{discard}: a list of data items that will be replaced from the cache.
  L_{request}: a list of new data items that the client requested from the server.

```
(A) At interval time $T_i$, construct $IR_i$ as follows:
    $IR_i = \{\langle d_x, t_x \rangle \mid (d_x \in D) \wedge (cnt_x > 0) \wedge (T_i - L \cdot w < t_x \leqslant T_i)\}$;
    $L_{\text{bcast}} = \{d_x \mid (T_{i-1} < t_x \leqslant T_i) \wedge (cnt_x > cnt_\alpha)\}$;
    Broadcast $IR_i$ and $L_{\text{bcast}}$; $L_{\text{update}} = \emptyset$;
    **for** each $d_x \in L_{\text{bcast}}$ **do**
        broadcast data item $d_x$;
        Execute step B if the *UIR* interval reaches.
    **for** each $CIR_j$ /* remove timeout clients */
        **if** $T_i > lease_j$
        **then for** each item $d_x \in CIR_j$ **do** $cnt_x --$; remove $CIR_j$.

(B) At interval time $T_{i,k}$, construct $UIR_{i,k}$ as follows:
    $UIR_{i,k} = \{d_x \mid (d_x \in D) \wedge (cnt_x > 0) \wedge (T_{i,0} \leqslant t_x < T_{i,k})\}\ (0 < k < m)$;
    **for** each $d_x \in UIR_i$ **do**
        **if** $d_x \in L_{\text{update}}$ **then** $UIR_{i,k} = UIR_{i,k} - d_x$; $UIR_{i,k} = UIR_{i,k} \cup \langle d_x, t_x \rangle$.

(C) Receives a *request*($L_{\text{discard}}, L_{\text{request}}$) from client $C_j$:
    $lease_j = T_\alpha +$ current time;
    $CIR_j = CIR_j \cup L_{\text{request}} - L_{\text{discard}}$;
    **for** each $d_x \in L_{\text{request}}$ **do**
        $cnt_x ++$; send data to $C_j$; $L_{\text{update}} = L_{\text{update}} \cup d_x$;
    **for** each $d_x \in L_{\text{discard}}$ **do** $cnt_x --$;

(D) Receives a *reconnect*($T_l$) from $C_j$:
    $L_{\text{valid}} = L_{\text{invalid}} = 0$;
    **for** each $d_x \in CIR_j$
        **if** $(t_x > T_l)$ **then** $L_{\text{invalid}} = L_{\text{invalid}} \cup d_x$ **else** $L_{\text{valid}} = L_{\text{valid}} \cup d_x$;
    **if** $(|L_{\text{invalid}}| > |L_{\text{valid}}|)$
    **then** send *valid*($L_{\text{valid}}$) to $C_j$
    **else** send *invalid*($L_{\text{invalid}}$) to $C_j$.
```

Figure 3. The algorithm at the server.

recovery_begin. At this stage, when it sends a data request to the server in case of a cache miss, it appends the local cache status and changes its status to *recovery_finish*. With these cache status information, the server can rebuild the CIR of the client. Since clients responds to the *recovery* message at different time, the number of collisions on the reserve control channel can be significantly reduced. After w IR intervals, the server finishes the recovery process, and switches back to the original algorithm, where it only adds the data *id*s whose counter is larger than to the *IR* or *UIR*.

4. Performance evaluation

In this section, we evaluate the performance of the proposed algorithm, the TS algorithm [2], and the BS algorithm [10] by extensive simulations.

4.1. The simulation model and system parameters

In order to evaluate the performance of various cache invalidation algorithms, we develop a simulation model which is similar to that used in [3,10]. It consists of a single server that

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.

```
(A) When $C_j$ receives $IR_i$ and $L_{bcast}$:
    if $lease_j$ is about to expire
    then send dummy data to the server to extend its lease;
    if $T_l < (T_i - L \cdot w)$ then send $reconnect(T_l)$ to the server;
    for each data item $\langle d_x, t_x^c \rangle$ in the cache
        if $(d_x \in IR_i) \wedge (t_x^c < t_x)$
        then invalidate $d_x$;
        else $t_x^c = T_i$;
    for each $d_x \in L_{bcast}$ do
        if $d_x$ is an invalid cache entry
        then download $d_x$ into local cache;
    $T_l = T_i$; if $(Q_i \neq \emptyset)$ then query$(Q_i)$.

(B) Receives a $UIR_{i,k}$:
    if missed $IR_i$ then break;          /* wait for the next IR */
    for each data item $\langle d_x, t_x^c \rangle$ in the cache
        if $(d_x \in UIR_{i,k}) \vee ((\langle d_x, t_x \rangle \in UIR_{i,k}) \wedge (t_x^c < t_x))$
        then invalidate $d_x$;
    if $(Q_{i,k} \neq \emptyset)$ then query$(Q_{i,k})$.

(C) Procedure query$(Q)$
    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 $L_{request} = L_{request} \cup d_x$;
    send $request(L_{discard}, L_{request})$ to the server; $L_{discard} = \emptyset$;
    Use the received data to answer the query;
    Add the received data item to local cache, if there is a need to replace one data item $d_x$ out of cache,
    $L_{discard} = L_{discard} + d_x$.
```

Figure 4. The algorithm at the client.

serves multiple clients. The database can only be updated by the server while the queries are made on the client side. The data items in the database 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 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. In the IR or UIR, 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 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 [8], 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 20 s. The UIR is replicated four times ($m = 5$) 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, while 33.3% of the updates are applied to the hot data subset. In the experiment, we assume

Table 1
Simulation parameters.

Number of clients	100
Database size (D)	1000–90000 items
Data item size	1024 bytes
Broadcast interval (L)	20 s
Broadcast bandwidth	10000 bits/s
Cache size (c)	50 to 300 items
Mean think time (T_t)	0–300 s
Broadcast window (w)	10 intervals
UIR replicate times ($m - 1$)	4 ($= 5 - 1$)
Hot data items	1–50
Cold data items	remainder of DB
Hot data access probability (p_h)	0.8
Mean update arrival time (T_u)	1–10000 s
Hot data update probability	0.33
Mean disconnect time (T_d)	0–400 s
Client disconnect probability (P_d)	0.1

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 separated from the completion of the previous query by either an exponentially distributed think time or an exponentially distributed disconnection time. A client can only enter disconnection mode when the outstanding query has been served.

The client processes generated queries one by one. If the referenced data items are not cached on the client side, the data *id*s 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 (or BS) 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 clients in our algorithm can download data from the broadcast channel.

To simplify the simulation model, we fix the counter threshold cnt_α to be 10 and we do not consider client failures except section 4.2.8. With $cnt_\alpha = 10$, most of the updated hot data will be broadcasted since there are 100 clients in the cell and they have a large probability (80%) to access the hot data. Most of the system parameters are listed in table 1.

4.2. Simulation results

4.2.1. The cache hit ratio

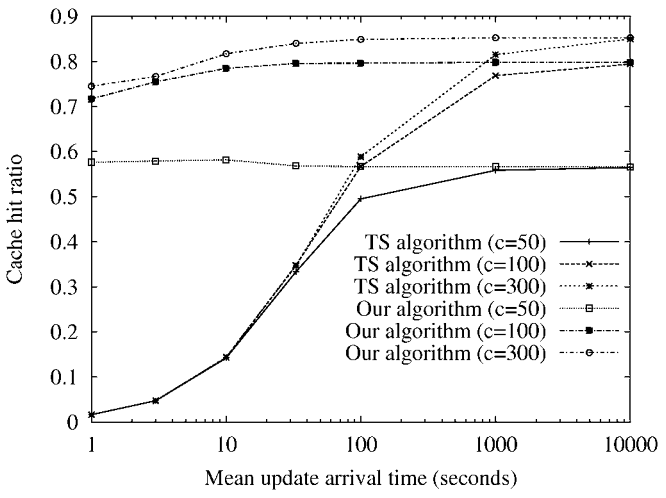
The performance metrics such as the query delay and the throughput have strong relations with the cache hit ratio. For example, if the cache hit ratio is high, the query delay can be reduced since the client can process most of the queries locally and does not need to request the data from the server.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

As shown in figure 5, without considering client disconnections, for one particular cache size (cache size is 50 items, 100 items, or 300 items), the cache hit ratio of our algorithm is always higher than that of the TS algorithm. In the TS algorithm, a client only downloads the data that it has requested from the server. In our algorithm, clients also download the updated hot data items that are broadcasted by the server. 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. This explains why our algorithm has higher cache hit ratio than the TS algorithm.

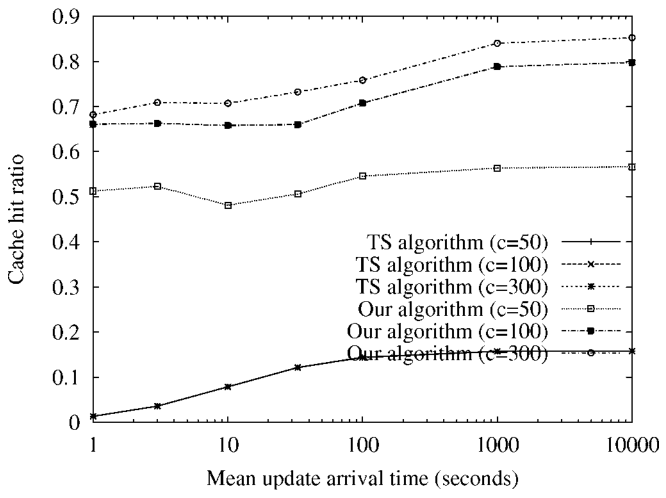
From figure 5, without considering client disconnections, we can see that the cache hit ratio grows as the cache size increases. However, the growing trend is different between the TS algorithm and our algorithm. For example, in the TS algorithm, when the mean update time is 1 s, the cache hit ratio does not have any difference when the cache size changes from 50 data items to 300 data items. In our algorithm, under the same situation, the cache hit ratio increases from 57% to 82% (figure 5(a)). In our algorithm, clients may need to download interested data for future use, so a large cache size may increase the cache hit ratio. But, 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 $T_u = 1$ s.

As shown in figure 5(a), the cache hit ratio drops as the mean update time decreases. However, the cache hit ratio of the TS algorithm drops much faster than our algorithm. When the update arrival time is 10000 s, without disconnection, both algorithms have similar cache hit ratio for one particular cache size. With $c = 300$, as the mean update time reaches 1 s, the cache hit ratio of our algorithm still keeps around 80%, whereas the cache hit ratio of the TS algorithm drops to near 0. In the TS algorithm, when the mean update time is very high (e.g., 10000 s), most of the cache misses are due to cold data accesses; when the mean update time is very low (e.g., 1 s), besides the cold data cache misses, most of the cached hot data may be updated by the server and resulting in a large amount of cache misses. In our algorithm, the updated hot data are most likely to be broadcasted and downloaded by the clients. When the cache size is large enough to save the hot data, accessing hot data should not result in cache misses. This explains why the cache hit ratio does not change too much when the mean update arrival time changes.

Generally speaking, client disconnections reduce the cache hit radio. However, both schemes react differently to the client disconnections. In the TS algorithm, if a client disconnects longer than $w \cdot L$, it invalidates all cached data although some of them may be valid. As a result, the client will have many cache misses before it fills up the cache again.

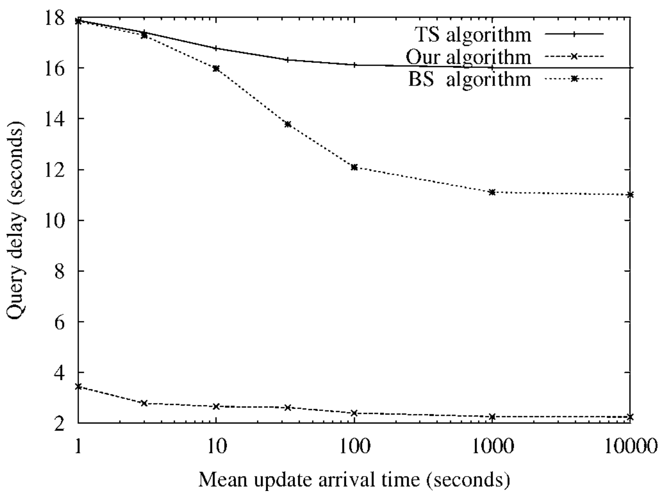


(a)

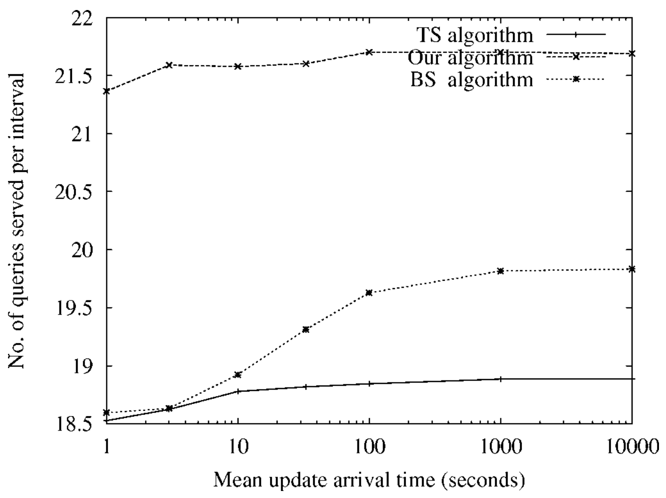


(b)

Figure 5. The cache hit ratio under different cache sizes ($T_t = 100$ s, $D = 1000$ items). There is no client disconnection in (a), while the client disconnection probability is 0.1 ($T_d = 400$ s) in (b).



(a)



(b)

Figure 6. The query delay and throughput as a function of the mean update arrival time ($T_t = 50$ s, $T_d = 400$ s, $D = 1000$ items, $c = 100$ items).

When the mean update arrival time is small, most of the cache misses are due to the server update. When the mean update arrival time increases, most of the cache misses are due to the client disconnections. This explains why the cache hit ratio of the TS algorithm in figure 5(a) is much less than that in figure 5(b), especially when the mean update arrival time is greater than 100 s. In our algorithm, if a client disconnects longer than $w \cdot L$, it sends a request to the server, and the server notifies the client which cache items are valid based on the CIR and the update history. As a result, the clients do not need to discard all cached data, and hence the cache hit ratio of our algorithm does not drop too much when considering client disconnections.

4.2.2. The effects of the mean update arrival time

As shown in figure 6, our algorithm outperforms the IR algorithm and the BS algorithm in terms of query delay and throughput. In our algorithm, in case of a cache hit, a client only needs to wait for the next UIR to serve queries, and hence the query delay is about $\frac{1}{2} \cdot \frac{20}{5} = 2$ s. In case of a cache miss, the client needs to send a request to the server and wait for the reply. From figure 5, the cache hit ratio of our algorithm does not have too much difference between $T_u = 10000$ s and $T_u = 1$ s. However, the query delay increases from about 2 s when $T_u = 10000$ s to almost 4 s when $T_u = 1$ s. This can be explained as follows. When the mean update arrival time is very small (e.g., 1 s), a large amount of hot data may be updated by the server, and the server may broadcast these data. Since broadcasting these data occupies the broadcast channel for some time, clients may experience long delay during cache misses.

In the BS algorithm and the TS algorithm, clients have to wait until the next IR before answering any query. Thus, the query delay is always longer than 10 s. During a cache miss,

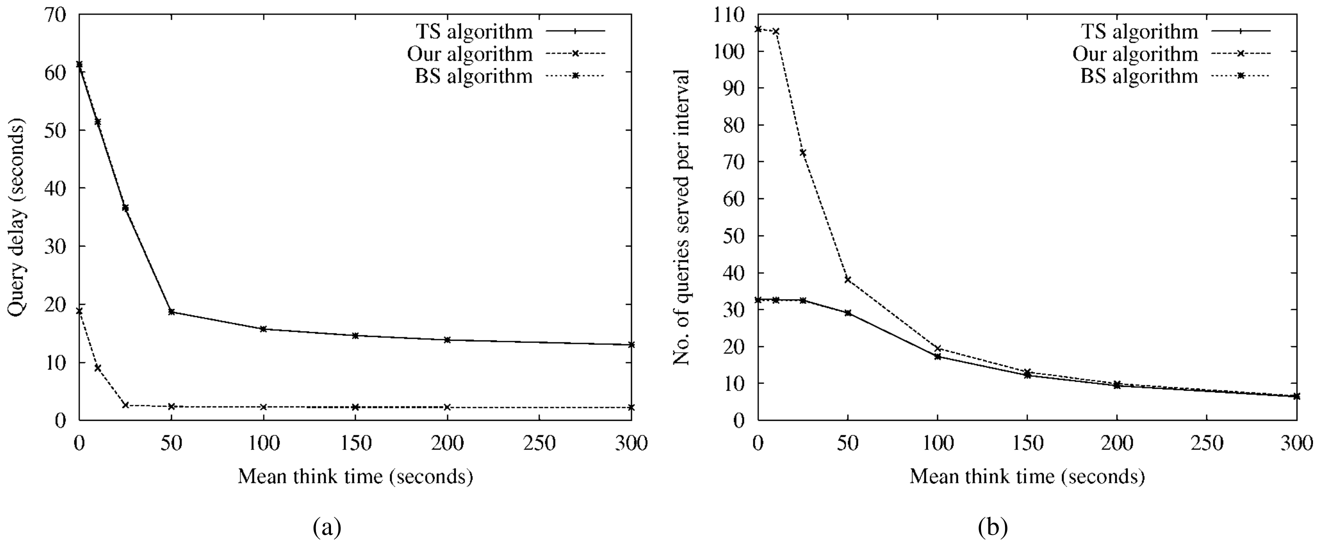


Figure 7. The query delay and throughput as a function of the mean think time ($T_u = 10$ s, $D = 1000$ items, $c = 100$ items). There is no client disconnection.

the clients have to request data from the server and the server may receive many requests. Due to the bandwidth limitation, these requests may suffer from long delays. When the mean update arrival time is very low (e.g., 1 s), the cache miss rate of the BS algorithm and the TS algorithm are all very high, and hence both algorithms have long query delay. As the mean update arrival time increases, the BS algorithm and the TS algorithm react differently. In the TS algorithm, when a client disconnects longer than $w \cdot L$, it discards the whole cache, and hence, the TS algorithm still has low cache hit ratio. This explains why the delay of the TS algorithm does not drop after the mean update arrival time is greater than 100 s. However, in the BS algorithm, the IR has enough update information for the client to invalidate its cache even though the disconnection time is larger than $w \cdot L$. As a result, the query delay drops as the mean update arrival time increases.

In our client model, the client generates new queries after an exponentially distributed think time or disconnection time. As the query delay drops, the next query may arrive earlier and be served earlier. Thus, the server can serve more queries within an IR interval. Since three algorithms have different query delays, they have different throughput. As shown in figure 6(b), our algorithm has the highest throughput, whereas the TS algorithm has the lowest, and the BS algorithm in the middle.

4.2.3. The effects of the mean think time

Figure 7 shows the effects of the mean think time on the performance of the TS algorithm, the BS algorithm, and our algorithm. Without considering client disconnections, the BS algorithm and the TS algorithm have similar performance when the database size is small. In our client model, each client generates queries according to the mean think time and the client disconnection 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. 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 will be delayed to the next IR interval. As shown in figure 7(a), the query delay of the TS (and BS) algorithm grows larger than the IR interval when the mean think time drops below 50 s.

Although the server can only serve a maximum number (α) of client requests during one IR interval, 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 and the BS algorithm, our algorithm can serve more queries locally, and the clients send less requests to the server. As a result, our algorithm has higher throughput than the BS algorithm and the TS algorithm when the mean think time is very low. For example, as shown in figure 7, when the query update time reduces from 25 s to 0, 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 about 32 whereas the throughput of our algorithm increases from about 72 to about 105. When the mean think time is high, the broadcast channel has enough bandwidth to serve the client requests, and hence three algorithms have similar throughput.

4.2.4. The effects of the database size

Figure 8 shows the effects of the database size on the performance of the TS algorithm, the BS algorithm, and our algorithm. As can be seen, the throughput and the query delay of the TS algorithm and our algorithm do not change too much as the database size changes. However, the performance of the BS algorithm is significantly affected by the database size. When the database has more than 50000 data items, the query delay of the BS algorithms grows significantly, and the

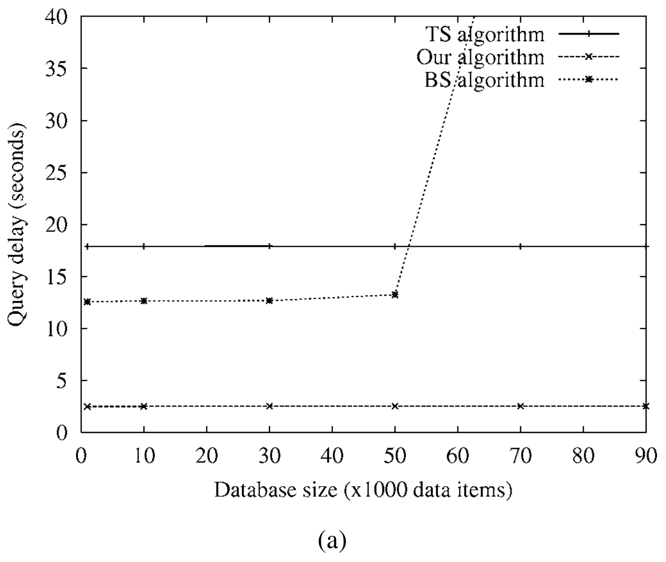


Figure 8. The effects of the database size ($T_u = 100$ s, $T_t = 25$ s, $T_d = 400$ s, $c = 100$ items).

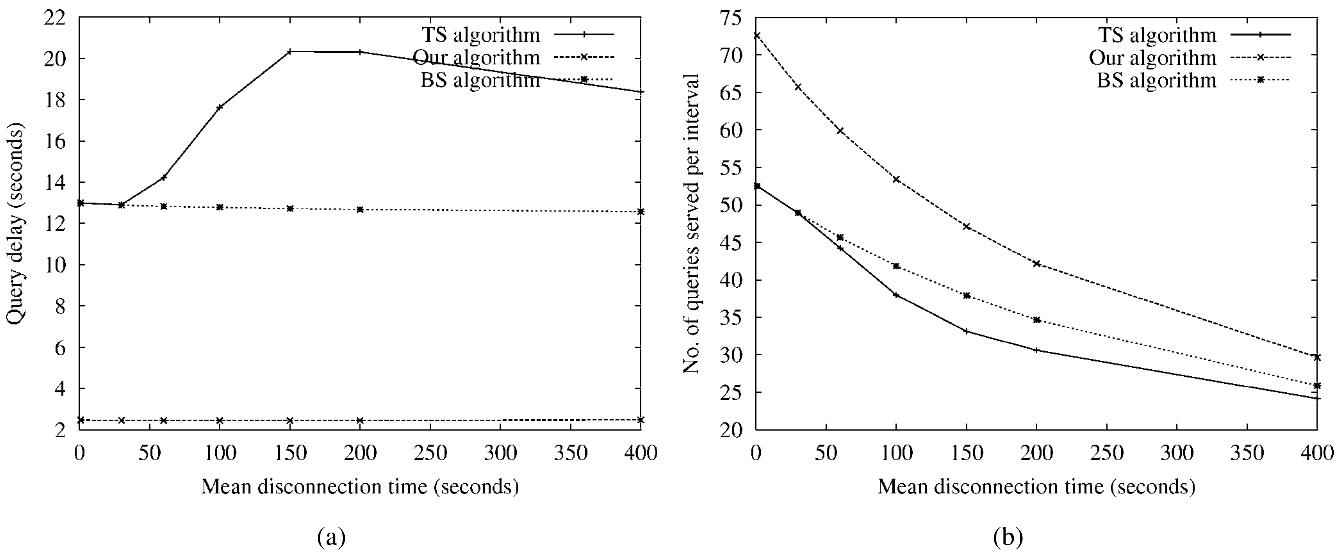


Figure 9. The effects of the client disconnection time ($T_u = 100$ s, $T_t = 25$ s, $D = 1000$ items, $c = 100$ items).

throughput drops to almost 0 as the database grows to 90000 data items. In the BS algorithm, the IR includes the update information of the whole database and the IR size grows as the database size increases. As the database size increases to 90000 data items, almost all of the downlink bandwidth are used to broadcast the IR, and hence, the query delay increases significantly and the throughput drops to almost 0. In the TS algorithm and our algorithm, the IR size is related to the number of updates in the previous w IR interval. In our model, the number of updates is not related to the database size, and hence the database size does not affect the performance of our algorithm and the IR algorithm.

4.2.5. The effects of the client disconnection time

Figure 9 shows the effects of the client disconnection time. In the BS algorithm and our algorithm, the client keeps the valid cache items even though it has been disconnected for a long time. Thus, the query delay of the BS algorithm and our algorithm do not change too much as the client disconnection time changes. Since a client generates less queries when the mean disconnection time increases, the throughput of both algorithms drops as the mean disconnection time increases.

In the TS algorithm, when the disconnection time is less then $w \cdot L$ seconds, clients can still validate their cache by using the IR since the IR includes the update history of the previous $w \cdot L$ seconds. However, if a client disconnects longer than $w \cdot L$ seconds, it has to discard the whole cache. Since the client disconnection time is exponentially distributed, the cache hit ratio of the TS algorithm drops as the disconnection time grows until $w \cdot L = 200$ s. Since the query delay increases as the cache hit ratio drops, the query delay of the TS algorithm grows to almost 20 s as the disconnection time increases to 200 s.

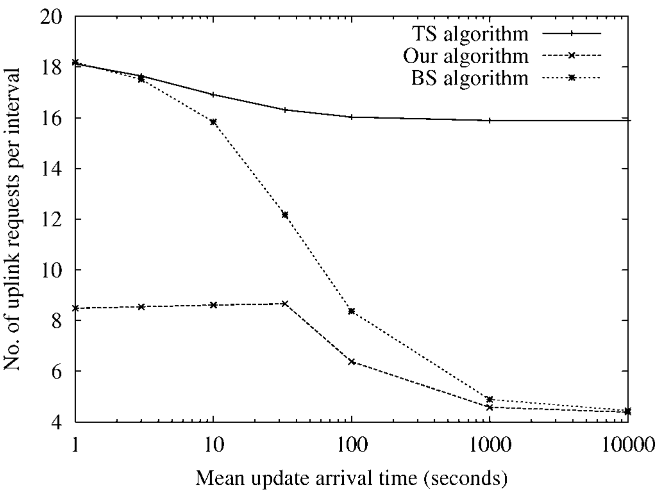


Figure 10. The number of uplink requests per IR interval ($T_t = 50$ s, $D = 1000$ items, $c = 100$ items).

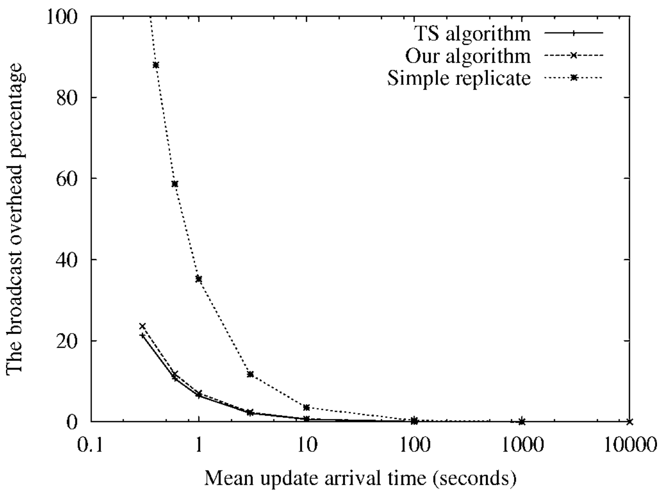


Figure 11. The broadcast overhead as a function of the update arrival time ($T_t = 100$ s, $c = 100$ items).

4.2.6. The number of uplink requests

As shown in figure 10, our algorithm has the lowest uplink cost, whereas the TS algorithm has the highest uplink cost and the BS algorithm is in the middle. This can be explained by the fact that three algorithms have different cache miss rates. Note that a client only sends a uplink request when a cache miss occurs. Generally speaking, as the mean update arrival time increases, the cache miss ratio drops and the number of uplink requests decreases. However, the decrease trend of three algorithms are different. In the TS algorithm, the number of uplink requests does not increase when the mean update arrival time grows above 100 s, which corresponds to the trend in figure 5. In the BS algorithm, since the client can obtain enough information form the IR to invalidate their caches even after a long disconnection, the cache miss ratio decreases as the mean update arrival time increases. This explains why the number of uplink requests in the BS algorithm continues to drop as the mean update arrival time increases. Note that the size of the uplink request in our algorithm may be large. However, compared to the header overhead and the delay resulted from competing for the reverse control channel, this overhead is very small.

4.2.7. The broadcast overhead

In our algorithm, the *id* list L_{bcast} can be implemented more efficiently. For example, since the *id* is long (32 bits), the most significant bit can be used to represent whether the real data of this item will be broadcasted. Only when the IR does not include the *id* of the data item in L_{bcast}, it is broadcasted in L_{bcast}. Certainly, the client needs to re-construct the L_{bcast} after it has received the IR. Let T_{uir} represent the average time that the server uses to broadcast the UIRs and the necessary L_{bcast} (except those already in the IR) within one IR interval (T_{uir} is 0 in the IR algorithm). Let T_{ir} represent the average time that the server spends on broadcasting the IRs within one IR interval. The broadcast overhead percentage is $(T_{\text{ir}} + T_{\text{uir}})/L$. Figure 11 compares the broadcast overhead of our algorithm to the IR algorithm and the *simple replicate algorithm*, which simply replicates the IR $4(m = 5)$ times within each IR interval (i.e., the broadcast interval L changes to $20/(4 + 1) = 4$ s). 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, the broadcast overhead of our algorithm is slightly higher than the TS algorithm, but far lower than the simple replicate algorithm. For example, when $T_u = 0.3$ s, 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%, which is similar to that of the TS algorithm.

4.2.8. The effects of client failures and the parameter cnt_α

In figure 12, n represents the number of client failures. When a client is (non-voluntarily) disconnected longer than the lease time, it is counted as a failure, and hence n can be very large. When a client fails, the counter value may not be accurate until the lease of the client expires. However, if the lease time[3] is relatively small compared to the failure time, this will not be a big issue. Since we evaluate the performance when the system becomes stable, failure itself does not have too much effect on the cache hit ratio. However, failure combined with cnt_α may significantly change the cache hit ratio.

In figure 12(a), $cnt_\alpha = 20$. The $n = 80$ approach, $n = 90$ approach have similar cache hit ratio to the TS approach, whereas the $n = 0$ approach, $n = 10$ approach, and the $n = 50$ approach have similar cache hit ratio. In the $n = 80$ approach and the $n = 90$ approach, most data items have counters smaller than cnt_α. As a result, updated data items will not be broadcasted and clients cannot prefetch data.

[3] In this paper, we do not evaluate the effect of lease time on the cache hit ratio in case of client failures. Since the effect strongly depends on the characteristics of the workload, we will further evaluate it when we get real workloads.

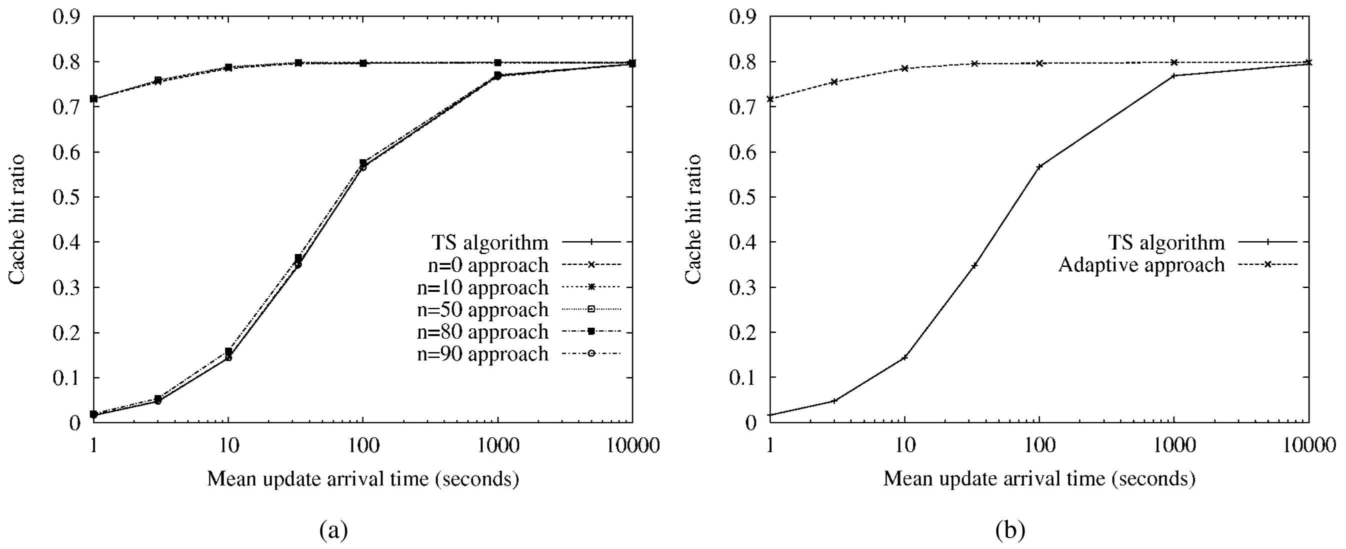


Figure 12. The effects of client failures and the parameter cnt_α ($T_u = 100$ s, $T_t = 100$ s, $D = 1000$ items, $c = 100$ items).

Figure 12(b) shows an approach in which cnt_α is adaptive to the amount of available bandwidth and the number of clients. In this approach, the updated data items in the previous IR interval are sorted based on the counter value and the server broadcasts as much updated items as possible. With this modification, the cache hit ratio of the proposed algorithm is much higher than the TS algorithm.

5. Conclusions

IR-based cache invalidation has received considerable attention due to its scalability. However, IR-based approach suffers from long query latency and cannot efficiently utilize the broadcast bandwidth. 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 uses counters to identify the hot data items and broadcasts the updated hot data items, whereas 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. We evaluated the performance of the proposed techniques and addressed various kinds of failures such as server failures, client failures, and disconnections. Simulation results showed that our algorithm can cut the query delay by a factor of 5, and increase the throughput by a factor of 3.5 compared to the TS algorithm and the BS algorithm.

Acknowledgements

We would like to thank the anonymous referees whose insightful comments helped us to improve the presentation of the paper. This work was supported in part by the National Science Foundation CAREER Award CCR-0092770.

Appendix. The stateless server approach

In this approach, the server does not have CIRs. To maintain counter accuracy, the server asks clients to report their cached data items, and hence recalculates the counter periodically. More specifically, if the server suspects that the counter of a data item is not accurate, it uses the following scheme to get the accurate value. When the suspected data item is updated, the server resets its counter to 0 and does not broadcast this data item. However, the server still adds the data item *id* to the IR for the next w IR windows. If a client is interested in the suspected data item, it needs to send a request to the server. The server sends out the data and increases the counter.

When a client sends a request to the server, it attaches information about the cached data items that will be replaced. For each replaced data item, the client attaches the timestamp of the last IR that it used to invalidate the data item. The server updates the counter for each received data item *id* based on the associated timestamp and the time when the server starts to check the counter accuracy of the data item. For example, if the server receives a replaced data item *id* whose associated (the last IR) timestamp is smaller than the time when the server starts to check the accuracy of the data item, the server will not decrease the counter of the data item. Otherwise, the server decreases the counter.

Although the stateless server approach does not need to maintain CIRs, it has several disadvantages. First, it is difficult for the server to decide when to start the process of checking counter accuracy; however, the decision may significantly affect the performance. Second, the server needs to maintain information about when it starts the process of checking counter accuracy, and the client needs to associate an extra timestamp for each data item *id* when it sends a request to the server. Third, during handoff or long disconnection, the client needs to notify the server about its cached data, which may consume a large amount of uplink bandwidth. Due to these disadvantages, we did not further evaluate the performance of the stateless server approach.

References

[1] S. Acharya, M. Franklin and S. Zdonik, Disseminating updates on broadcast disks, in: *Proc. 22nd VLDB Conf.* (September 1996).

[2] D. Barbara and T. Imielinski, Sleepers and workaholics: Caching strategies for mobile environments, in: *ACM SIGMOD* (1994) pp. 1–12.

[3] G. Cao, A scalable low-latency cache invalidation strategy for mobile environments, in: *ACM Int. Conf. on Mobile Computing and Networking (MobiCom)* (August 2000) pp. 200–209. An extended version has been accepted by IEEE Transactions on Knowledge and Data Engineering.

[4] G. Cao and M. Singhal, On coordinated checkpointing in distributed systems, IEEE Transactions on Parallel and Distributed Systems 9(12) (December 1998) 1213–1225.

[5] G. Cao and M. Singhal, Mutable checkpoints: A new checkpointing approach for mobile computing systems, IEEE Transactions on Parallel and Distributed Systems 12(2) (February 2001) 157–172.

[6] G.H. Forman and J. Zahorjan, The challenges of mobile computing, IEEE Computer 27(6) (April 1994) 38–47.

[7] Q. Hu and D. Lee, Cache algorithms based on adaptive invalidation reports for mobile environments, Cluster Computing (February 1998) 39–48.

[8] IEEE 802.11, Wireless LAN Media Access Control (MAC) and Physical Layer (PHY) specifications, 802.11 wireless standards (1999) `http://grouper.ieee.org/groups/802/11`

[9] T. Imielinski, S. Viswanathan and B. Badrinath, Data on air: Organization and access, IEEE Transactions on Knowledge and Data Engineering 9(3) (May/June 1997) 353–372.

[10] 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 (1997) 115–127.

[11] A. Kahol, S. Khurana, S. Gupta and P. Srimani, An efficient cache management scheme for mobile environment, in: *The 20th Int. Conf. on Distributed Computing Systems* (April 2000) pp. 530–537.

[12] H. Korth, The double life of the transaction abstraction: Fundamental principle and evolving system concept, in: *Proc. VLDB* (September 1995).

[13] W. Lee, Q. Hu and D. Lee, A study on channel allocation for data dissemination in mobile computing environments, Mobile Networks and Applications (1999) 117–129.

[14] G. Cao, Proactive power-aware cache management for mobile computing systems, IEEE Transactions on Computers (June 2002).

[15] R. Powers, Batteries for low power electronics, Proceedings of IEEE 83(4) (April 1995) 687–693.

[16] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel and D. Steere, Coda: A highly available file system for a distributed workstation environment, IEEE Transactions on Computers 39(4) (April 1990).

[17] K. Wu, P. Yu and M. Chen, Energy-efficient caching for wireless mobile computing, in: *The 20th Int. Conf. on Data Engineering* (February 1996) pp. 336–345.

[18] J. Yuen, E. Chan, K. Lam and H. Leung, Cache invalidation scheme for mobile computing systems with real-time data, in: *ACM SIGMOD Record* (December 2000).

Guohong Cao received the B.S. degree from Xian Jiaotong University, Xian, China. He received the M.S. degree and Ph.D. degree in computer science from the Ohio State University in 1997 and 1999, respectively. Since Fall 1999, he has been an Assistant Professor of computer science and engineering at Pennsylvania State University. His research interests include distributed fault-tolerant computing, mobile computing, and wireless networks. He was a recipient of the Presidential Fellowship at the Ohio State University. He is a recipient of the NSF CAREER award.
E-mail: gcao@cse.psu.edu
WWW: www.cse.psu.edu/˜gcao