Sleepers and Workaholics: Caching Strategies in Mobile Environments

Daniel Barbará
Matsushita Information Technology Laboratory
2 Research Way, 3rd Floor
Princeton, N.J. 08540
daniel@mitl.research.panasonic.com

Tomasz Imieliński
Rutgers University
Department of Computer Science
New Brunswick, N.J. 08903 USA
imielins@cs.rutgers.edu

Abstract

In the mobile wireless computing environment of the future a large number of users equipped with low powered palmtop machines will query databases over the wireless communication channels. Palmtop based units will often be disconnected for prolonged periods of time due to the battery power saving measures; palmtops will also frequently relocate between different cells and connect to different data servers at different times. Caching of frequently accessed data items will be an important technique that will reduce contention on the narrow bandwidth wireless channel. However, cache invalidation strategies will be severely affected by the disconnection and mobility of the clients. The server may no longer know which clients are currently residing under its cell and which of them are currently on. We propose a taxonomy of different cache invalidation strategies and study the impact of client's disconnection times on their performance. We determine that for the units which are often disconnected (sleepers) the best cache invalidation strategy is based on signatures previously used for efficient file comparison. On the other hand, for units which are connected most of the time (workaholics), the best cache invalidation strategy is based on the periodic broadcast of changed data items.

1 Introduction

In the mobile wireless computing environment of the future [5] massive number of low powered palmtop machines will query databases over the wireless communication channels. Palmtop based units will often be disconnected for prolonged periods of time due to the battery power saving measures; they will also frequently relocate between different cells and connect to different data servers at different times.

The *mobile* or *nomadic* computing environment no longer requires users to maintain a fixed and universally known position in the network and enables unrestricted mobility of the users. Mobility and portability will create an entire new class of applications and possibly new massive markets combining personal computing and consumer electronics.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

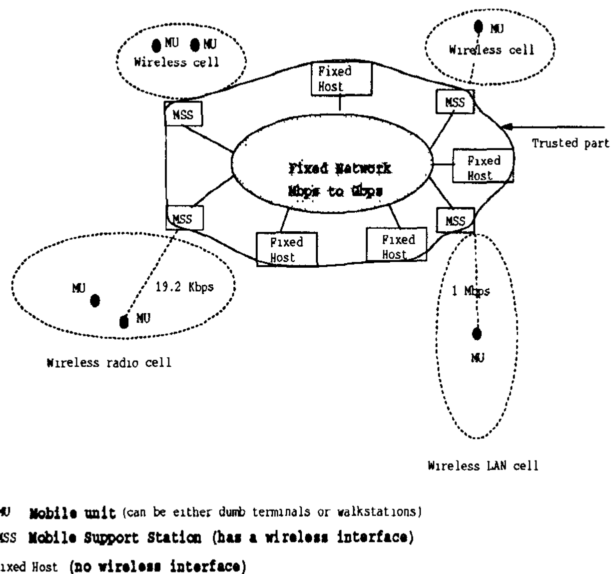

Figure 1: A mobile environment

Figure 1 displays the architecture of the general mobile environment. It consists of two distinct sets of entities: mobile units MUs and fixed hosts, as shown in Figure 1. Some of the fixed hosts, called MSS (Mobile Support Station), are augmented with a wireless interface to communicate with mobile units which are located within a radio coverage area called a *cell*. A cell could be a real cell as in cellular communication network or a wireless local area network which operates within the area of a building. In the former case the bandwidth will be severely limited (supporting data rates on the order of 10 to 20 kbits/s). In the latter, the bandwidth is much wider - up tp 10 Mb/sec. Fixed hosts will communicate over the fixed network, while mobile units will communicate with other hosts (mobile or fixed) via a wireless channel.

In this paper, we assume that the MUs can cache a portion of the database. They can do this in a disk (if

they are equipped with it), or any storage system that survives power disconnections, such as flash memories. We also assume that the data is updated in a stationary server and MUs only carry copies of the data for their own use.

Mobile computing will bring about a new *style* of computing. Due to battery power restrictions, the mobile units will be frequently disconnected (power off). Most likely, the short bursts of activity, like reading, sending e-mail, or querying the local databases will be separated by substantial periods of disconnection. In general, we will distinguish between the *awake time*, when the unit is "on", and a *sleep time* , when the unit is "off", and inaccessible to the rest of the network [1] For our purposes, the main difference between disconnection and failure will be the high frequency of disconnection compared to failures [2]

In this paper we investigate a scenario where mobile units query databases that are replicated on stationary servers connected by the fixed network. We will assume that each stationary server corresponds to the local MSS. This server communicates with mobile clients over a wireless channel. We assume also that the database is being accessed by users who are mobile within a *wide area*, move between different cells and frequently disconnect. We also assume that data is being updated at the servers and that the replicated copies are kept consistently[3].

Caching of frequently accessed data items will be an important technique to reduce contention on the narrow bandwidth wireless channel between a client and a server. However, cache invalidation strategies will be severely affected by the disconnection and mobility of the clients since a server may no longer know which clients are currently located under its cell and which of them are "on".

In the paper we propose a taxonomy of different cache invalidation methods. Further on, we analyze some of them and study the impact of disconnection time on their performance (which will be measured in overall query *throughput*). We show that caching is a general technique which may lead to overall throughput improvement by making it possible to answer queries locally without competing for the scarce wireless bandwidth.

[1] There is another, intermediate state when the unit is in the "dozing mode" when it's CPU is working on the lower rate and when the unit can be awaken by a message from outside. For our purposes this state will correspond to the awake state. In the truly disconnected mode the mobile unit will simply ignore incoming messages

[2] Another difference is due to the elective nature of disconnection - the user often knows when the disconnection will occur, so the mobile unit can prepare for it, as opposed to failures, which in general cannot be predicted.

[3] The replication of the database between many servers is actually not important for the considerations in this paper - we may as well assume that there is just one remote server

The following are examples of applications that can profit from the techniques we discuss in this paper. The scenario we envision is that of a massive number of users querying local databases over a wireless channel.

1.1 Example 1

Consider a large number of mobile users who are interested in the news updates involving business information, recent sales/profit figures, stock market data, etc. Assume that each of the users has defined a "filter" that selects the data items of interest to him/her. A user may switch his unit on to run an application program such as spreadsheet which queries these data items to perform some computations[4]. Subsequently, a user may switch off his/her mobile unit to wake up later and query again.

1.2 Example 2

Consider a server that administers navigational data containing traffic reports and other useful information for travelers. Assume this information is kept in a pictorial form: a map with icons that summarize traffic volumes in each section of the map and other useful information. The map is divided in sections by a grid. Each section is given a data identification number. Each user is interested at any particular moment in a set of data items that corresponds to the section in which he or she is currently located, plus the neighboring sections. These could be for instance, a set of 9 neighboring sections with the center section being the current location of the user. The mobile unit maintains a display of the data in these sections running an application program that periodically refreshes the display by asking the values of each of the data items involved (sections). There is a large degree of locality in these queries, since the users move relatively slowly. That is, the area covered by each section in the map is fairly big with respect to the relative displacement of the user per second. The user again, may switch the mobile unit off and on.

There are a number of possibilities to manage data in these scenarios:

- No caching in the MUs: each query goes directly to the server.
- Caching data in the MUs: here we have to worry about how to inform MUs that their cache items

[4] For example a mobile salesman may query inventory of the merchandise he is selling in order to check the availability and pricing information

have changed. Their are three possible ways of doing this:

- The server sends invalidation messages to the clients. The Andrew File System [11] is an example of a client-server architecture that uses this option. An invalidation message regarding a data item that just changed is directed to clients that are caching that particular item. To this end, the server has to locate appropriate clients. Since disconnected clients cannot be reached, each such client upon reconnection has to contact the server to obtain a new version of the cache. Hence disconnection automatically implies loosing a cache. The server in this case is **stateful** since it knows about the state of the clients' caches.
- The clients query the server to verify the validity of their caches. The Network File System [10] is an example of a client-server architecture that takes this approach. Obviously, this option generates a lot of traffic in the network.
- The server broadcasts a report (periodically or asynchronously) in which only the database items which have been updated are broadcasted. But then, since clients may have caches of different age, these reports have to be well defined by given a time window of reference for the updates or the update's timestamps, for instance. There is a number of possibilities for the composition of the reports which we shall describe in detail in the next section. The server in this case is **stateless** since it does not know about the state of the client's caches. (Or the clients themselves.)

Which method should we choose? The answer depends on many parameters, such as the intensity of updates and queries, and the sleep and awake patterns of the mobile units.

This paper is organized as follows. In Section 2 we state the problem. In Section 3 we describe some cache invalidation strategies. In Section 4 we present the models and analysis of the strategies. Section 5 present an asymptotic analysis based on the formulas derived in Section 4. Section 6 shows examples of different scenarios. Finally, in Section 7 we discuss future work and summarize the paper's conclusions.

2 Problem Statement

The database is a collection of n named data items. The data items can be numerical (stock data, temperature) as well as textual (news). Let us consider a set of data servers each covering a cell (see Figure 1). We assume that the database is fully replicated at each data server but each data server can only serve users which are currently located in its cell.

Assume we have a large number of MUs residing in a cell and issuing queries which are simple requests to read the most recent copy of an item. We assume that the database is updated only by the server[5]. The MUs exhibit a large degree of data locality, querying a particular subset of the database repeatedly. This subset is a hotspot for the MU.

Let us now examine the possible strategies for handling the read requests of the mobile users. The goal is to minimize the number of bits that are transmitted in the channel both ways: downlink (from the server to the MUs), and uplink (from the MUs to the server). Alternatively, the MUs may or may not cache data. If we choose to cache we may consider two ways of managing the caches: having an stateful server or having an stateless server. The stateful server knows which units currently reside in its cell. It also knows the states of their caches. If a particular data item changes and it is cached by a user U, then the server will send an invalidation message, or a refresh message (with the item value) to U. This is analogous to a standard caching strategy used for the client server architectures. To maintain the server state, the clients must inform the server when they come and go (i.e., enter its cell and leave it), they must also inform the server when they are about to disconnect and when they reestablish the connection. (This is, of course, subject to the assumption that the clients will have time to do so. This will not be the case if the disconnection is caused by a failure.)

The stateless server case offers a variety of algorithms. Here, the server has no information about which units are currently under its cell and what are the contents and "ages" of their caches (how long ago a particular unit cached a particular data item). We will distinguish here between synchronous and asynchronous cache invalidation methods. In asynchronous methods the server broadcasts an invalidation message for a given data item as soon as this item changes its value. A client who is currently is the connect mode can then invalidate the cached version of this item. A client who is disconnected looses its cache entirely. We may avoid this by piggybacking some extra information on asynchronous invalidation messages; for example we may include information about other data items and the timestamps of their most recent changes. Then instead of just plain invalidation message we get an *invalidation report*. In this case, the disconnected client may actually be able to save its cache if it can afford to wait for the first asynchronous invalidation report after

[5] This assumption is not really necessary but it considerably simplifies the further analysis

reconnection (provided that the report indicates that cached data items have not changed during the given client's disconnection period). Notice, however, that no guarantees can be given about when this report will be sent and hence no guarantees can be given for the waiting time.

Synchronous methods of cache invalidation are based on periodic broadcasting of ***invalidation reports.*** A client has to listen to the invalidation report first in order to conclude whether its cache is valid or not. (Notice that this adds some latency to query processing.) If the decision is negative the client has to issue a query to the server and refresh its cache. It may turn out that the invalidation report leads to a "false alarm," and in fact the cache was valid. However if, given an invalidation report, the MU concludes that the cache is valid, it must in fact be so. Hence, our schemes will only allow false alarm errors and will always correctly inform the client if his copy is invalid.

The validity of the client's copy is only guaranteed as of the last invalidation report. Formally this means the following. The server timestamps each report with the time at the initiation of the broadcast. If the last report was broadcast with timestamp T_i and a client determines that a particular item's cache is valid after listening to the report, this cache gets timestamped with the value T_i (marked as valid up to this time). If the client has to submit an uplink request because the cache is invalid, then the obtained copy has the timestamp equal to the timestamp of the request. The broadcast of the invalidation reports divides the time into intervals. Notice that the MU has to wait for the next invalidation report before answering a query (see Figure 2). The MU keeps a list of items queried during an interval and answers them after receiving the next report.

Notice that this way of operation has the following consequences:

- If two or more queries of the same item are posed in an interval, they will all be answered at the same time in the next interval.
- The answer to a query will reflect any updates to the item made during the interval in which the query was posed. (See Figure 2.) Notice that this is the case even if the query predates the update during the interval.

We can classify invalidation reports sent by stateless servers according to different criteria as follows:

- How the server sends the invalidation reports:
  - Asynchronous
    Here invalidation reports are broadcasted immediately after changes to data items occur. In particular, the report may just contain the name of

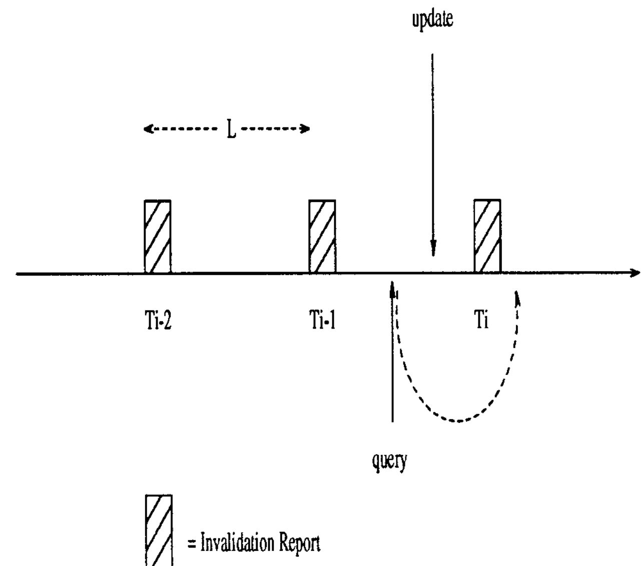

Figure 2: Invalidation Reports

    the changed data item. In general, it may have extra information about other data items such as timestamps of their most recent changes.
  - Synchronous
    When the invalidation reports are broadcasted periodically.

- What kind of information is sent in the invalidation reports:
  - State Based: reports that contain information about the values of the items in the database. For example, a state based report can give the values of the items that have changed since the last report.
  - History Based: reports that contain information about when the items values have changed. For example, a history based report can contain the identities of items that have changed during the last w seconds (where w is a parameter) and the timestamps of their last update.

- How the information is organized in the invalidation report:
  - Uncompressed: the reports contain information about individual items. For example, an uncompress report may contain the values of the items that have changed since the last report.
  - Compressed: the reports contain aggregate information about subsets of items. For example, a

compressed report may contain aggregate information about changes by using predicates such as "There was a change on departure time in one or more of the eastbound flights."

A wide range of methods can be proposed, in this paper we will concentrate on the synchronous caching methods for the stateless server. Asychronous methods do not provide any latency guarantees for the querying client; if the client queries a data item after the disconnection period then it either has to wait for the next invalidation report (with no time bound on the waiting time) or has to submit the query to the server (cache miss). In case of synchronous caching, there is a guaranteed latency due to the periodic nature of the synchronous broadcast. We will also demonstrate later that one of our synchronous broadcast methods (AT) is actually equivalent to the asynchronous broadcast.

In the next section we describe three of these methods.

3 Strategies

In this section we describe three strategies that use invalidation reports and a stateless server. In what follows, we assume that an invalidation report is broadcasted every L seconds and that D denotes the set of items in the database.

3.1 Broadcasting Timestamps

We call this strategy TS. In TS, the invalidation report is composed by the timestamps of the latest change for items that have had updates in the last w seconds. The MU listens to the report and updates the status of its cache. For each item cached, the MU either purges it from the cache (when the item is reported to have changed at a time larger than the timestamp stored in the cache), or updates the cache's timestamp to the timestamp of the report (if the item was not mentioned there). Notice that this is done for every item in the cache, regardless of whether there is a pending query to this item or not.

The server begins to broadcast the invalidation report periodically at times $T_i = iL$. The server keeps a list U_i defined as follows:

$$U_i = \{[j, t_j] | j \in D \text{ and } t_j \text{ is the timestamp of the last update of } j \text{ such that } T_i - w \leq t_j \leq T_i\} \quad (1)$$

Upon receiving the invalidation report, the MU compares the items in its cache $[j, t_j^c]$ (where $j \in D$ and t_j^c is the cache's timestamp for j) to decide whether to keep j in the cache or not. Also, the MU has a list $Q_i = \{j | j \text{ has been queried in the interval } [T_{i-1}, T_i]\}$. The MU also keeps a variable T_l that indicates the last time it received a report. If the difference between the current report timestamp and this variable is bigger than w, the entire cache is dropped. More formally, the MU runs the following algorithm:

```
if (Ti - Tl > w) { drop the entire cache }
else {
  for every item j in the MU cache {
     if there is a pair [j, tj] in Ui {
       if tj^c < tj {
         throw j out of the cache }
       else { tj^c = Ti }
     }
  }
}
for every item j ∈ Qi {
   if j is in the cache {
     use the cache's value to answer the query }
   else { go uplink with the query }}
Tl := Ti
}
```

Following the terminology of Section 2, TS uses synchronous, history based, and uncompressed reports.

3.2 Amnesic Terminals

In this strategy, which we call Amnesic Terminals (AT), the server only broadcasts the identifiers of the items that changed since the last invalidation report. A MU that has been disconnected for a while, needs to start rebuilding its cache from scratch. As before, we assume that if the unit is awake, it listens constantly to reports and modifies its cache by dropping invalidated items.

As in TS, the server builds a list of items to be broadcast. However, the list in AT is defined as follows:

$$U_i = \{j | j \in D \text{ and the last update of } j \text{ occurred at } t_j \text{ such that } T_{i-1} \leq t_j \leq T_i\} \quad (2)$$

Upon receiving the invalidation report, the MU compares the items in its cache with those in the report. If a cached item is reported, then the MU drops it from the cache. Otherwise, it considers the cached item valid. Again, the MU has a list

$Q_i = \{j | j \text{ has been queried in the interval } [T_{i-1}, T_i]\}$.

Again, the MU keeps a variable T_l that indicates the timestamp of the last report received. If the difference between the current report timestamp and T_l is more than L, the entire cache is dropped. The algorithm for the MU is as follows:

```
if (Ti - Tl > L) { drop the entire cache }
```

```
else {
  for every item j in the MU cache {
      if j is in the report {
        throw j out of the cache }
  }
}
for every item j ∈ Q_i {
    if j is in the cache {
      use the cache's value to answer the query }
    else { go uplink with the query }}
T_l := T_i
}
```

Finally, we would like to compare the AT method to the asynchronous broadcast of invalidation messages for individual data items. Notice that in both cases the total number of messages downloaded by the server is identical, the AT simply groups then together in the periodic invalidation[6]. Also, in both cases, the client looses his cache entirely upon disconnection. Therefore AT is really equivalent to the asynchronous broadcast of invalidation reports and the results of analysis of AT will apply equally well to the asynchronous broadcast.

Following the terminology of Section 2, AT uses synchronous, history based, uncompressed reports.

3.3 Signatures

Signatures are checksums computed over the value of items. Sending combined signatures have proven to be a useful practice for comparing two or more copies of a file that has a large number of pages (see [2, 9, 4, 8] for examples of such techniques). The techniques compute a signature per page and a set of combined signatures that are the Exclusive OR of the individual checksums. Each combined signature, therefore, represents a subset of the pages. A node A sends its combined signatures to another node B, which can in turn can diagnose how many pages in its copy of the file are different from the A copy. Some of the techniques diagnose a fixed number of different pages by carefully selecting the subsets that compose the combined signatures (e.g., [4, 8]). Other techniques (e.g., [2, 9]) are probabilistic since they diagnose a page to be different with certain accuracy probability. In these techniques, the membership of a page in a subset is decided by random methods. Although most of the techniques (of both types) are designed to diagnose up to f different pages, most of them will render a superset of the differing pages when the actual number of differing pages is greater than f.

The file comparison problem differs from our problem in the sense that the MUs do not store the entire database in their caches and therefore cannot compute the entire set of combined signatures. However, all the techniques can be easily changed to accommodate for this difference, by doing the following. The server periodically broadcasts the set of combined signatures. (The composition of the subsets of each combined signature is universally known and agreed before any exchange of information takes place.) The MUs cache along with the individual items of interest, all the combined signatures of subsets that include items of interest for the MU. The not cached combined signatures are considered equal to the ones that are being broadcast in the current interval.

To make this paper self-contained and also to take into account the difference between file comparison and our problem, we present here a technique taken from [2] and the analysis of the probability of falsely diagnosing items. We call this technique SIG.

For each item i in the database, we can compute a signature $sig(i)$, based on the value of the item. If the signature has s bits, the probability of two different items having the same signature is 2^{-s}.

The signatures for a set of items can be combined into one by performing Exclusive OR of the individual signatures. If the individual signatures have s bits, the combined signature will also have s bits. If two combined signatures of the same subset are equal, the individual items are equal with a probability that depends on s. The probability that one or more of the items involved in the signature are different but have the same combined signature is approximately 2^{-s} ([4, 9]).

We describe a technique that is a variation of the techniques *SUCC* described in [9] and SUSPECTS in [2]). We assume that a MU contains n^* items in the cache, and of those, $f^* = kf$ items really need to be invalidated. There are m randomly chosen sets of items (*a priori*, before any exchange of signatures takes place) called $S_1, S_2, ..., S_m$. Each set is chosen so that an item i is in set S_j with probability $\frac{1}{f+1}$. The server computes the m combined signatures $sig_1, sig_2, ..., sig_m$ and broadcasts them. A MU k, caches signatures $sig_{i'_1}, sig_{i'_2}, ..., sig_{i'_k}$, of subsets that include items cached by the MU. The MU compares these signatures with the ones broadcast by the server, constructing a syndrome matrix as follows

$$\alpha_j = \begin{cases} 1 & \text{if } j = i_r, r \in \{1, ..., k\} \text{ and } sig_j \neq sig'_{i_r} \\ 0 & \text{otherwise} \end{cases}$$

Notice that the MU puts a 1 in α_j if j is one of the subsets whose signature is cached by it and the sent signature sig_j and the cached signature sig'_j do not match. In cases in which the two signatures do match or the MU does not cache this signature, the entry is filled with zero.

[6] Which actually may lead to saving in terms of total number of packets send due to better utilization of the space within packets

With this matrix in hand, the MU can run the following algorithm, where T the set of items whose cache is to be invalidated. The notation $i \in S_j$ means to test whether item i belongs to the subset whose combined signature is sig_j. The variable δ_f is a threshold chosen as $K(\frac{1}{1+f}(1-\frac{1}{e}))$. (The reason for this will become obvious when analyzing the probability of a false alarm.)

$T = \emptyset$
for $j = 1$ *to* m *do*
  if $\alpha_j = 1$ *then*
    for $i = 1$ *to* n *do*
      if $i \in S_j$ *then*
        $count[i] := count[i] + 1$
for $i = 1$ *to* n *do*
  if $count[i] \geq m\delta_f$ *then*
    $T := T \bigcup i$

Essentially, an item is declared invalid if it belongs to "too many" unmatching signatures. (suspected of being out-of-date) Again, as in the previous two methods we assume that the MU, while awake, is constantly listening to the reported signatures and invalidating cached data items "on line".

Following the terminology of Section 2, SIG uses synchronous, state based, compressed reports.

4 Analysis

In this section we develop analytical models for the techniques presented in the last section. We begin by stating the assumptions of our model:

- There are n items in the database. We call the set of items D.
- The bandwidth of the wireless network is W.
- Each query that has to go uplink takes b_q bits. The answer takes b_a bits.
- Each timestamp takes b_T bits.
- Updates occur following an exponential distribution, at an update rate of μ per item.
- Each MU will repeatedly query a subset of D with a high degree of locality. This subset is thus a "hot spot" for the MU. Each item in the hot spot will be queried at the MU at the rate λ.
- In the caching strategies, the server broadcasts the invalidation report every L seconds.
- The MUs get disconnected and reconnected while they are in the cell (the user turns the machine on and off). We model this by assuming that in each interval an MU has a probability s of being disconnected and $1-s$ of being connected. We assume the behavior of the MU in each interval is independent on the behavior on the previous interval. Notice that this is a simplifying assumption, since in reality if a query has been issued in one interval, the MU will stay connected in the next in order to answer the query. We assume the query gets answered but independently the unit may decide to go to sleep.

We will use the following notation:

$$Prob[\text{ no queries in an interval } \mid \text{ unit is awake during the interval }] = e^{-\lambda L} \quad (3)$$

$$q_0 = Prob[\text{awake and no queries in an interval}] = (1-s)e^{-\lambda L} \quad (4)$$

$$p_0 = Prob[\text{no queries in an interval}] = s + q_0 \quad (5)$$

$$1 - p_0 = Prob[\text{one or more queries in an interval}] = (1-s)(1-e^{-\lambda L}) \quad (6)$$

$$u_0 = Prob[\text{ no updates during an interval}] = e^{-\mu L} \quad (7)$$

$$1 - u_0 = Prob[\text{ one or more updates during an interval}] = 1 - e^{-\mu L} \quad (8)$$

We now derive the basic equation that describes the throughput (number of queries that can be answered) for the stateless server case. First notice that the interval is always divided in two sections: the time taken to broadcast the report and the rest of the interval, which is used to send queries to the server and receive the answers. The total number of bits that can be transmitted during the interval is LW. We call the number of bits transmitted by the broadcast B_C. Therefore, the number of bits available for answering queries that were cache misses is $LW - B_C$. Now, if the total number of queries per interval handled by the system (throughput) is T, a fraction $T(1-h)$, where

h is the average hit ratio in a MU, corresponds to the queries that were not cache hits. Each one of this queries takes $(b_q + b_a)$ bits, so the traffic in bits due to queries that did not hit the caches is $T(1-h)(b_q+b_a)$. Since this amount has to be equal to $LW - B_C$, we have:

$$T = \frac{LW - B_C}{(b_s + b_a)(1-h)} \tag{9}$$

In order to "normalize" the throughput of each one of the techniques and to be able to fairly compare the effectiveness of each one of them, we define the effectiveness of an strategy as:

$$e = \frac{T}{T_{max}} \tag{10}$$

where T_{max} is the throughput given by an unattainable strategy in which the caches are invalidated instantaneously and without incurring any cost. In the rest of this section we will analyze T_{max} and the throughput and effectiveness of each of the strategies presented in the last section along with the ones obtained when caches are not used (all the queries are transmitted uplink).

4.1 Maximal Throughput

Consider a strategy in which the server knows exactly which units are in the cell and the contents of their caches. Assume also that everytime an update occurs, the server ***instantaneously*** sends an invalidation message to all the MUs that have the item in their cache. By this unattainable strategy we would get the maximum hit ratio (a miss would occur only when an update to the item has happened). Since there are no invalidation reports, B_C would be equal to 0. Then, the maximal throughput will be (using Equation 9):

$$T_{max} = \frac{LW}{(b_q + b_a)(1 - MHR)} \tag{11}$$

Where MHR is the maximal hit ratio, i.e., the hit ratio achieved by this strategy. To compute this, assume that we have a query occurring at some particular instant of time. The query will "hit" the cache if: a) the last query on this item occurred exactly τ seconds ago, and b) there has been no updates during the two queries.

The probability of the first event is simply the probability of the interarrival time being τ. That is $\lambda e^{-\lambda\tau}$. The probability of the second event is $e^{-\mu\tau}$. Therefore, MH can be computed as:

$$MHR = \int_0^\infty \lambda e^{-\lambda\tau} e^{-\mu\tau} d\tau \tag{12}$$

evaluating the integral,

$$MHR = \frac{\lambda}{\lambda + \mu} \tag{13}$$

4.2 No caching

Of course, when the MUs are not caching any data, there will not be any invalidation report ($B_C = 0$) and no intervals. However, we compute here the number of queries that can be processed during an interval of duration L, in order to compare this to the values obtained for the rest of the strategies. Since no caches are available, the hit ratio will be 0 and all the queries will go uplink for processing. Therefore, the throughput for the no-cache scenario is:

$$T_{nc} = \frac{LW}{(b_q + b_a)} \tag{14}$$

4.3 TS

We assume that the window w is a multiple of L. (This is a very natural choice of values for w, since queries get answered only after listening to the next invalidation report.) Thus $w = kL$. In order to compute B_C for this strategy, we need to calculate the number of items that have changed during the window w. We call this value n_c. This value can be computed as:

$$n_c = n(1 - e^{-\mu w}) \tag{15}$$

The total size of the report will be $n_c(log(n) + b_T)$. Therefore, the throughput is given by:

$$T_{TS} = \frac{LW - n_c(log(n) + b_T)}{(b_q + b_a)(1 - h_{ts})} \tag{16}$$

We need to compute The average hit ratio for TS, h_{ts}, is analyzed in [1]. The upper and lower bounds for it are:

$$\frac{(1-p_0)u_0}{1-p_0u_0} - \frac{(1-p_0)u_0^{k+1}s^k}{1-p_0u_0} - \frac{(1-p_0)u_0^{k+1}s^kq_0}{(1-p_0u_0)^2} < h_{ts} \tag{17}$$
$$< \frac{(1-p_0)u_0}{1-p_0u_0} - \frac{(1-p_0)u_0^{k+1}s^k}{1-q_0u_0}$$

4.4 AT

The size of the report in AT becomes $n_L log(n)$, where n_L is the expected number of items that have changed since the last broadcast. This value can be computed similarly to Equation 15:

$$n_L = n(1 - e^{-\mu L}) \tag{18}$$

Again, the throughput can be found by computing from Equation 9 as follows:

$$T_{AT} = \frac{LW - n_L log(n)}{(b_q + b_a)(1 - h_{at})} \tag{19}$$

The hit ratio h_{at} is analyzed in [1]. Its equation is:

$$h_{at} = \frac{(1-p_0)u_0}{1-q_0u_0} \tag{20}$$

4.5 SIG

We begin analyzing the probability of false alarm. The probability of falsely diagnosing items in the cache can be studied from two points of view. First there is the probability of not diagnosing an item as invalid when in reality its cache is outdated. This, according to our earlier remarks can be bounded by 2^{-g} and can be made arbitrarily small by increasing g (at the expense of more bits transmitted of course).

Secondly, there is the probability of diagnosing a cache as invalid, when it is not. To compute this probability, consider first the probability of a valid cache being in a differing signature. For this to happen, the following must be true:

1. The item must belong to the set in the signature. This happens with probability $\frac{1}{1+f}$.
2. Some item whose cache has to be invalidated must be in the set and the signature must be different, the probability of which is $(1-(1-\frac{1}{1+f})^f)(1-2^{-g})$, (Notice that even though one or many of such items might not be in the cache of the particular MU, they might be in a subset that contains an item cached by the MU.) This expression can be approximated by $1-\frac{1}{e}$.

So, the probability p of an valid cache being in a signature that does not match is

$$p = \frac{1}{1+f}(1-\frac{1}{e}) \tag{21}$$

If we define now a binomial variable X with parameters m and p, we can compute the probability of this variable to exceed the threshold. This probability is $p_f = Prob[X \geq m\delta_f] = Prob[X \geq Kmp]$. (This explains the reason why the threshold δ_f was chosen to be Kp in the algorithm.)

By results that can be found in [2, 9], based in the Chernoff inequality ([3]), we can state that

$$p_f = Prob[X \geq Kmp] \leq exp(-(K-1)^2 m\frac{p}{3}) \tag{22}$$

with $1 \leq K \leq 2$.

Now the probability of not having a false diagnose is simply $p_{nf} = 1 - p_f$.

Now, in reality, we want that the probability of any of the valid caches in the MU being falsely diagnosed to be smaller than a certain threshold, δ. That is $(n^* - f^*)p_{nf} \leq \delta$. In order to make this true, we have to send m combined signatures such that:

$$m \geq \frac{3(ln(\frac{1}{\delta} + ln(n^* - f^*))}{p(K-1)^2} \tag{23}$$

Making $K = 2$ and noticing that $\frac{e}{e-1} < 2$ and $n > n^* - f^*$, Equation 23 can be made true if the following holds:

$$m \geq 6(f+1)(ln(\frac{1}{\delta}) + ln(n)) \tag{24}$$

The throughput using this number of signatures is given by:

$$T_{sig} = \frac{LW - 6g(f+1)(ln(\frac{1}{\delta}) + ln(n))}{(b_q + b_a)(1-h_{sig})} \tag{25}$$

The hit ratio for SIG is analyzed in [1]. The equation is:

$$h_{sig} = \frac{(1-p_0)u_0p_{nf}}{1-p_0u_0} \tag{26}$$

5 Asymptotic Analysis

This section analyzes the throughput of the techniques presented in extreme cases.

The first analysis we want to present shows the behavior as the probability of sleeping s tends to 0 and 1. The following table summarizes the limit values for hit ratios and probabilities.

parameter	$s \to 0$	$s \to 1$
q_0	$e^{-\lambda L}$	0
p_0	$e^{-\lambda L}$	1
h_{ts}	$\frac{(1-e^{-\lambda L})e^{-\mu L}}{(1-e^{-\lambda L}e^{-\mu L})}$	0
h_{at}	$\frac{(1-e^{-\lambda L})e^{-\mu L}}{(1-e^{-\lambda L}e^{-\mu L})}$	0
h_{sig}	$\frac{(1-e^{-\lambda L})e^{-\mu L}}{(1-e^{-\lambda L}e^{-\mu L})}p_{nf}$	0

As the MUs sleep less and less (i.e., as $s \to 0$), a behavior that we call "workaholic," the hit ratios for all the techniques presented approach the same value, with SIG lagging behind by the factor p_{nf}. In this case, the best throughput will be exhibited by AT, since its report will be the shortest one.

When the MUs sleep a lot (i.e., as $s \to 1$), a behavior that we call "sleepers," the hit ratios of all the technique approach 0. With large values of s, the no-caching scenario eventually will win. However, it is also important to notice that h_{at} (Equation 20), goes to 0 faster than its counterparts h_{ts} (Equation 18) and h_{sig} (Equation 26). The reason for this is that the denominator of h_{at} contains the term $1 - q_0u_0$ which tends to 1 as s approaches 0, while the other two contain

the term $1 - p_0 q_0$ which approaches $1 - u_0$. This is specially true for small values of u_0, i.e., for low update rates.

We will show now the behavior as u_0 approaches 1. This happens for very small values of μL, that is, when the updates are infrequent. The next table shows the behavior of the hit ratios.

parameter	$u_0 \rightarrow 1$
h_{ts}	$1 - s^k - s^k q_0 < h_{ts} < 1 - s^k + \frac{s^{k+1}}{1-q_0}$
h_{at}	$1 - \frac{s}{1-q_0}$
h_{sig}	p_{nf}

From this table we can deduce that the hit ratio of TS (approximately $1 - s^k$) will be better than the one for AT, especially as the number of queries decreases (q_0 approaches 0). Since the size of the report is also proportional to the number of items that changed, TS is likely to be a winner over AT in this scenario. As for SIG, the hit ratio exhibits a constant behavior in this case, equal to the probability of not having a false diagnose. This is slightly better than the behavior of h_{TS}, but it is likely to be offset by the effect of a larger invalidation report.

It is worth pointing out that for update intensive scenarios (u_0 approaching 0), all the hit ratios will approach 0. Therefore, at high rates of updating, the no caching strategy will be a winner.

The analysis performed in this section shows the following conclusions:

- For "workaholics" the strategy AT will be the winner in throughput.
- For "sleepers" both TS and SIG may outperform AT. At some point for large values of s (heavy sleepers), no-caching will be the best choice. The strategy TS will outperform AT when the update rate is small.

6 Some Examples

In this section we show some scenarios for the techniques presented in this paper. Each scenario corresponds to a set of values for the parameters involved.

The first scenario corresponds to the following set of values:

λ	10^{-1} query/sec.
μ	10^{-4} updates/sec.
L	10 sec.
n	10^3
b_T	512
W	10000 b/sec
k	100
f	10
g	16

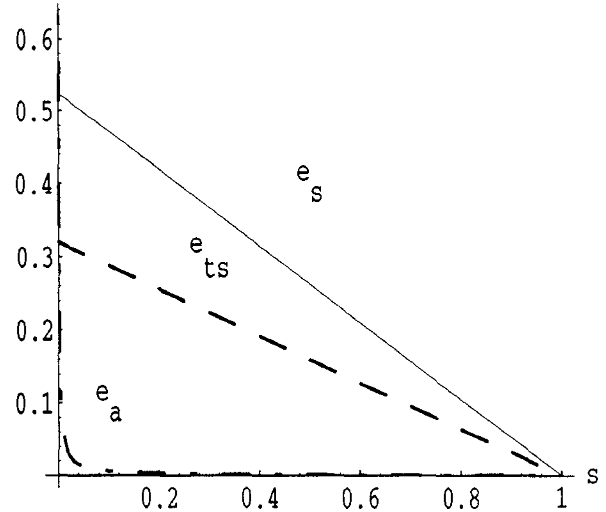

Figure 3: Effectiveness for Scenario 1

This set of parameters corresponds to an scenario of infrequent updates ($u_0 = 0.999$). Figure 3 presents the effectiveness of the techniques as s varies from 0 to 1.

As we can see, SIG (solid line) behaves better than the other two techniques during the entire range of s. The effectiveness of AT (e_a) goes rapidly to 0 as s grows. TS exhibits an intermediate effectiveness. It is worth pointing out that for this scenario, the value of e_n the effectiveness of the no-caching strategy remains very close to 0 for the entire interval.

For the second scenario, we have the following set of parameters:

λ	10^{-1} query/sec.
μ	10^{-1} updates/sec.
L	10 sec.
n	10^3
b_T	512
W	10000 b/sec
k	10
f	20
g	16

This scenario is update intensive (the rate of updates equals the rate of queries). We have increased f to reflect the need to respond to many more changes in SIG.

Figure 4 shows the behavior of the techniques as s goes from 0 to 1. TS is not included in this plot, since the size of the report for this scenario would exceed L, rendering the technique unusable.

We can see that AT dominates SIG for the entire

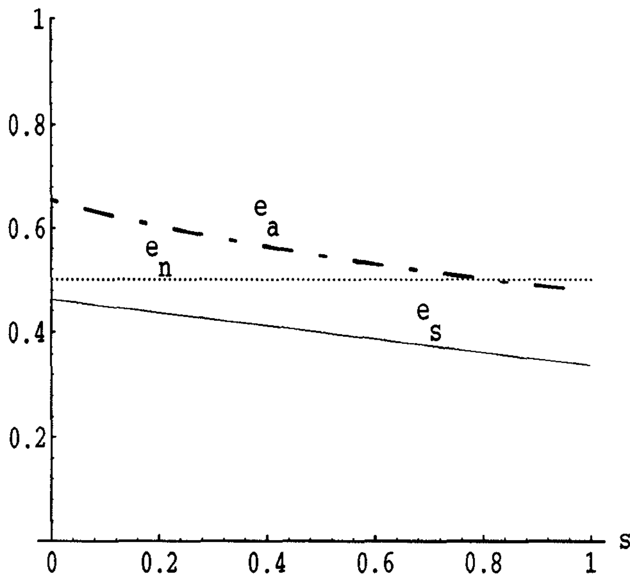

Figure 4: Effectiveness for Scenario 2

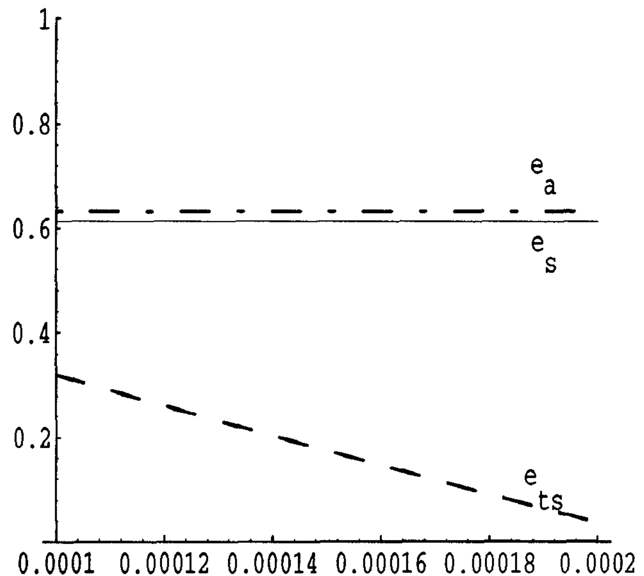

Figure 5: Effectiveness for Scenario 3

range. However, at some point ($s = 0.8$) the no-caching strategy becomes more advantageous. It is worth noticing that the values of efficiency remain relatively high even for $s = 1$. This is due to the relatively low value of T_{max} achieved in this scenario. In other words, due to the high update rate, even the maximum throughput achievable is low. It is encouraging that AT can achieve up to 40% of the maximum throughput in this disadvantageous situation.

The third scenario is based on the following parameters:

λ	10^{-1} query/sec.
s	0
L	10 sec.
n	10^3
b_T	512
W	10000 b/sec
k	100
f	1
g	16

We vary this time the update rate from $\mu = 10^{-4}$ to $\mu = 210^{-4}$ and plot the results in Figure 5. This scenario corresponds to "workaholics" ($s = 0$) with a varying rate of updates. We see AT overperforming TS in the entire range. The TS technique degrades rapidly with the increase on the update rate. SIG, on the other hand behaves marginally worse than AT in the entire range of values.

7 Conclusions and Future Work

We have proposed three new cache invalidation methods suitable for the wireless environment with the high rate of client's disconnection. In all three strategies the server periodically broadcasts the *report* which reflects the changing database state. We have categorized the mobile units on the basis of the amount of time they spend in their sleep mode into sleepers, and workaholics. Different caching strategies turn out to be effective for different populations. Thus, signatures which are based on the data compression technique for file comparison are best for long sleepers, when the period of disconnection is long and difficult to predict. Broadcasting with timestamps proved to be advantageous for query intensive scenarios, that is, scenarios when the rate of queries is greater than the rate of updates, provided that the units are not workaholics. As the rate of updates increases, TS becomes less and less efficient. Finally, the AT method was best for workaholics, that is, units which rarely go to sleep and are awake most of the time.

The methods presented in this paper are by no means exhaustive. There is a number of possible improvements. Aggregate invalidation reports can be considered, with varying granularity of time (timestamps given on the per minute instead of, say, per second basis) and items (changes reported only per group of items). We plan to address these isles in a future paper.

Caching is only one example of the wide range of classical systems issues which are expected to be affected by the mobile computing environment [5]. In particular, we expect that data broadcasting over wireless medium

will be an effective way of disseminating information to a large number of users [6] since the cost of the wireless broadcast does not depend on the number of users. In this paper we made a step in this direction by demonstrating how broadcasting can be used for cache invalidation.

There is a number of possible improvements to the scheme presented in the paper: the performance of signatures can be improved by considering the weighted schemes where each data item would be weighted according to the relative frequency it is accessed in a given cell and according to how often it is updated. For example, the "hot spot" items can be individually broadcast while the rest of the database items would participate in the signatures. In this way the signature will vary from cell to cell depending on the local usage patterns. The database may not be fully replicated in all data servers either. In this case the cost of querying different data items may depend on the location of the user.

Finally, we like to point out that broadcast solutions require MUs to listen for reports that include items the MU may not be caching. This presents a problem if the user is paying for the listening time. However, there are ways to alleviate this problem. For instance, the server can broadcast indexes that will tell the unit when to listen to items of interest [7]. Moreover, we believe that there will be services for which the user will pay a flat fee (subscription rate) or no fee (services constrained to buildings) for their usage. In these cases, broadcast mechanisms have the potential of providing better throughput than other solutions.

8 Acknowledgments

We would like to thank K. Kaplan and J.L. Palacios for their help on developing the probabilistic models presented in this paper and S.Vishwanthan for a number of useful comments. Special thanks go to Hector Garcia-Molina for helpful suggestions concerning the general model of the paper and for carefully proofreading it.

References

[1] D. Barbará and T. Imieliński. Sleepers and Workaholics: Caching Strategies in Mobile Environments. Technical Report MITL-TR-58-93, MITL, June 1993.

[2] D. Barbará and R.J. Lipton. A Class of Randomized Strategies for Low-Cost Comparison of File Copies. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):160–170, 1991.

[3] H. Chernoff. A Meassure of Asymptotic Efficiency for Tests of a Hypotesis Based on the Sum of Observations. *Annals of Mathematical Statistics*, 23:493–509, 1952.

[4] W.K. Fuchs, K. Wu, and J. Abraham. Low-Cost Comparison and Diagnosis of Large Remotely Located Files. In *Proceedings of the Fifth Symposium on Reliability of Distributed Software and Database Systems*, January 1986.

[5] T. Imielinski and B.R. Badrinath. Querying in Highly Mobile and Distributed Environments. In *Proceedings of the Eighteen International Conference on Very Large Databases, Vancouver*, August 1992.

[6] T. Imielinski, B.R. Badrinath, and S. Viswanathan. Data Dissemination in Wireless and Mobile Environments. Technical Report 59, WINLAB, Rutgers University, June 1993.

[7] T. Imielinski, S. Viswanthan, and B.R. Badrinath. Indexing on Air. In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, Minnesota*, May 1994.

[8] T. Madej. An Application of Group Testing to the File Comparison Problem. In *Proceedings of the International Conference on Distributed Computing Systems*, June 1989.

[9] S. Rangarajan and D. Fussell. Rectifying Corrupted Files in Distributed File Systems. In *Proceedings of the International Conference on Distributed Computing Systems*, May 1991.

[10] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Desing and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX Summer Conference*, pages 119–130, June 1985.

[11] M. Satyanarayanan, J. H. Howard, D. N. Nichols, R. N Sidebotham, A. Z Spector, and M. J. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 35–50, December 1985.