VLDB Journal, **4**, 567-602 (1995), Stanley Y.W. Su, Editor

Sleepers and Workaholics: Caching Strategies in Mobile Environments (Extended Version)

Daniel Barbará and Tomasz Imieliński

Received August 30, 1994; revised version received, February 27, 1995; accepted March 28, 1995.

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 wireless communication channels. Palmtop-based units will often be disconnected for prolonged periods of time, due to battery power saving measures; palmtops also will frequently relocate between different cells, and will 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 individualization 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 clients' disconnection times on their performance. We study ways to improve further the efficiency of the invalidation techniques described. We also describe how our techniques can be implemented over different network environments.

Key Words. Wireless, caching, data management, information services.

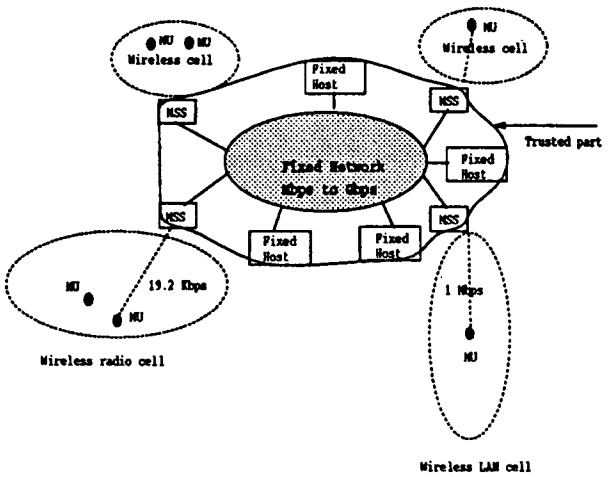
1. Introduction

In the mobile, wireless computing environment of the future, massive numbers of low-powered palmtop machines will query databases over wireless communication channels. Palmtop-based units often will be disconnected for prolonged periods of time, due to battery power saving measures; they also will 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 it enables

Daniel Barbará, Ph.D., is Senior Scientist, Matsushita Information Technology Laboratory, 2 Research Way, 3rd Floor, Princeton, NJ 08540, *daniel@mitl.research.panasonic.com,* and Tomasz Imieliński, Ph.D., is Associate Professor, Rutgers University, Department of Computer Science, New Brunswick, NJ 08903, *imielins@cs.rutgers.edu.*

Figure 1. Mobile environment



MU: Mobile unit (can be either dumb terminals or walkstations)
MSS: Mobile support station (has a wireless interface)
Fixed Host (no wireless interface)

unrestricted mobility for users. Mobility and portability will create an entire new class of applications and, possibly, new massive markets combining personal computing and consumer electronics.

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

In this article, we assume that the MUs can cache a portion of the database. They can do this on a disk (if they are equipped with a disk drive), 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 that MUs carry only 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 (e.g., reading and sending e-mail, or querying

local databases) will be separated by substantial periods of disconnection. In general, we distinguish between *awake time,* when the unit is "on," and *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 is the high frequency of disconnection compared to failures.[2]

In this article, we investigate a scenario where mobile units query databases that are replicated on stationary servers connected by the fixed network. First, we assume that each stationary server corresponds to the local MSS. This server communicates with mobile clients over a wireless channel. Second, we assume that the database is being accessed by users who are mobile within a *wide area*, who move between different cells, and who frequently disconnect. Third, we assume that data are 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."

The following are examples of applications that can profit from the techniques we discuss in this article. 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 news updates involving business information (e.g., recent sales/profit figures, or stock market data). Assume that each of the users has defined a "filter" that selects the data items of interest. 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 mobile unit to wake up later and query again.

---

1. There is another, intermediate state when the unit is in "dozing mode," that is, when its CPU is working on the lower rate, and the unit can be awakened by a message from outside. For our purposes, this state corresponds 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 article. We may as well assume that there is just one remote server.

4. For example, a mobile salesman may query inventory of the merchandise he is selling to check availability and pricing information.

1.2 Example 2

Consider a server that administers navigational data containing traffic reports and other useful information for travelers. Assume that this information is kept in a pictorial form, that is, a map with icons that summarize traffic volumes and other useful information in each section of the map. The map is divided in sections by a grid. Each section is given a data identification number. At any particular moment, each user is interested in a set of data items that corresponds to the section in which he is currently located, plus the neighboring sections. These could be, for instance, a set of nine 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 while running an application program that periodically refreshes the display by updating the values of each of the data items in each section. 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 large with respect to the relative displacement of the user per second. Again, the user may switch the mobile unit off and on.

If the clients cache some of the items, we need to worry about strategies that invalidate these caches when the data are updated. Although caching mechanisms have been studied extensively (e.g., Nitzberg and Lo, 1991), no technique deals effectively with the characteristics of wireless environments: low bandwidth and frequent disconnection of mobile units. There are many ways of handling the invalidation. We view each invalidation strategy as an *obligation* that the server maintains towards its clients. That obligation serves as a contract between the server and its clients. The mere understanding of the contract gives clients a great deal of information on how to handle their caches. This notion is similar to the *agreement set* (Mummert et al., 1994), which determines for each data object whether the server and client copies are equal. Mummert et al. (1994) used the logic of authentication (Burrows et al., 1990) and the notion of "belief" to reason about the cache coherence provided by some protocols. It is possible to use such tools to reason about the protocols presented in this article as well.

For example, consider the following obligation: "the server will notify about changes to item x every day at 1pm." The clients that cache copies of x know that the consistency of the current copy is only guaranteed to be yesterday's value. If they need a stricter consistency, they will have to ignore the cached value and go directly to the server, paying the price of communication.

The server agrees to follow a well-understood policy on when to notify the clients about changes to the items, and how to perform that notification. The policy could be as unbinding as "the server does not notify at all about changes." The format of the notification may vary from individual messages to broadcast reports. Also, the notification could range from information from which the status of a particular cache could be determined (by running a known algorithm) to delivering information about a particular item.

Notice that the common understanding of an obligation gives the clients a great deal of a-priori information. For instance, if the server agrees to notify every L seconds in case an item has changed, the lack of notification tells the clients the cached value is still valid for the next L seconds. This fact can be used to implement strategies that save transmission costs.

Two types of obligations are commonly used by today's systems:

- The server sends invalidation messages to the clients every time an item changes its value. The Andrew File System (Satyanarayanan et al., 1985) and Coda (Satyanarayanan et al., 1990) are examples of client-server architectures that use 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 (Sandberg et al., 1985) is an example of a client-server architecture that takes this approach. Obviously, this option generates a lot of traffic in the network.

The two types of obligations mentioned above are not appropriate for mobile, wireless systems. The first one requires that the clients register their presence, and that the server keep information about every cache. Besides, even if the client is not about to use a particular cache, it gets notified about its invalid status. This is a potential waste of bandwidth. In the second approach, the units are required to send a message every time they want to use their cache, which is wasteful of both the bandwidth and energy of the mobile unit.

We explore a different kind of obligation. In our techniques, the server broadcasts a report (periodically or asynchronously) in which only the updated database items are included. But, since clients may have caches of different ages, these reports have to be well defined (e.g., by a given time window of reference for the updates, or by the update's timestamps). There are a number of possibilities for the composition of the reports, which we 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 about 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.

In this article, we do not treat the case of MUs moving between cells. Therefore, all our algorithms deal with caching data within one cell only.

This article is organized as follows. In Section 2, we state the problem. In Section 3, we describe some obligations based on invalidation reports. In Section 4, we

present the models and analyses of the strategies. Section 5 presents an asymptotic analysis based on the formulas derived in Section 4. Section 6 shows examples of different scenarios. Sections 7 and 8 show ways of improving the effectiveness of the techniques presented. Section 7 focuses on reducing the size of the reports by relaxing the cache consistency. Section 8 presents some adaptive techniques to enhance the performance of the scheme. Section 9 shows the impact that different networking scenarios can have in the implementation of our techniques. In Section 10, we discuss future work and summarize the article.

2. Problem Statement

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

Assume we have a large number of MUs residing in a cell and issuing simple queries 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, repeatedly querying a particular subset of the database. 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 can manage the caches with either a stateful server or a 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 and leave its cell). 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. It has no information about which units are currently under its cell, or what are the contents and "ages" of their caches (how long ago a particular unit cached a particular data item). We

5. This assumption is not really necessary, but it considerably simplifies the further analysis.

distinguish 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 then can invalidate the cached version of this item. A client who is disconnected loses its cache entirely. We may avoid this by piggybacking 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 a plain invalidation message, we get an *invalidation report*. In this case, the disconnected client actually may be able to save its cache if it can afford to wait for the first asynchronous invalidation report after 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.

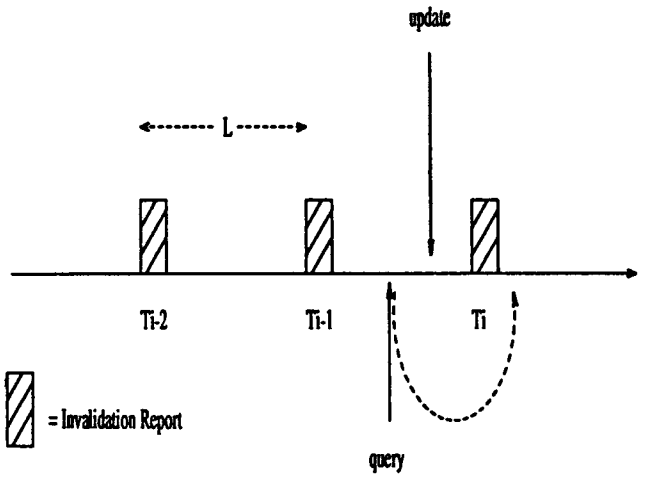
Synchronous methods of cache invalidation are based on periodic broadcasting of *invalidation reports*. A client must listen first to the invalidation report to conclude whether or not its cache is valid. (Notice that this adds some latency to query processing.) If the decision is negative, the client 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, given an invalidation report, if 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 (using the server's clock). 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 (Figure 2). (Waiting for the next invalidation report guarantees that all of the items are up-to-date as of the starting time of the report. This guarantee is much stronger than what the system would provide if the caches were read immediately.) 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 (Figure 2). Notice that this is the case even if the query predates the update during the interval.

Figure 2. Invalidation reports



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 contain only the name of 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 article, we concentrate on the synchronous caching methods for the stateless server. Asynchronous 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 it 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 to fulfill the obligations. 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 this case, the server agrees to the notify the clients about items that have changed in the last w seconds. Thus, the invalidation report is composed of the timestamps of the latest change for these items. 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 it 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.

The server begins to broadcast the invalidation report periodically at times $T_i = iL$. (Notice that w and L are unrelated, and the only constraint that applies between them is that $w \geq L$.) 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. Also, the MU has a list $Q_i = \{j \,|\, j \text{ has has been queried in the interval } [T_{i-1}, T_i]\}$. 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 (T_i − T_l > w) { drop the entire cache }
else {
    for every item j in the MU cache {
        if there is a pair [j, t_j] in U_i {
          if t_j^c < t_j {
            throw j out of the cache }
          else { t_j^c = T_i }
        }
    }
}
for every item j ∈ Q_i {
        if j is in the cache {
          use the cache's value to answer the query }
        else { go uplink with the query }}
T_l := T_i
    }
```

Notice that the timestamps in the cache need not be all the same. This is due to the fact that some of the items may have been requested uplink after being invalidated and, therefore, they received a different timestamp.

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 has the obligation to inform about the identifiers of the items that changed since the last invalidation report. An 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:

$$U_i = \{j \,|\, j \in D \text{ and the last update of } j \text{ occured at } t_j \text{ such that } T_{i-1} \leq t_j \leq T_i\} \tag{2}$$

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 (T_i − T_l > 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 them together in the periodic invalidation.[6] Also, in both cases, the client loses 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 has proven to be a useful practice for comparing two or more copies of a file that has a large number of pages (Fuchs et al., 1986; Madej, 1989; Barbará and Lipton, 1991; Rangarajan and Fussell, 1991). 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, 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., Fuchs et al., 1986; Madej, 1989). Other techniques (e.g., Barbará and Lipton, 1991; Rangarajan and Fussell, 1991) 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

6. Which actually may lead to saving in terms of total number of packets sent, due to better utilization of the space within packets.

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 changed easily to accommodate for this difference, as follows. The server periodically broadcasts the set of combined signatures. (The composition of the subsets of each combined signature is universally known and agreed on 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 combined uncached signatures are considered equal to the ones that are being broadcast in the current interval.

To make this article self-contained, and to take into account the difference between file comparison and our problem, we present a technique taken from Barbará and Lipton (1991) and an 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} (Fuchs et al., 1986; Rangarajan and Fussell, 1991). Notice that the server is agreeing to periodically report the value of the items. However, the server chooses to fulfill this obligation by aggregating the items into sets. This policy is also understood by the clients, along with the fact that the aggregation may cause the clients to falsely diagnose some of caches as invalid.

We describe a technique that is a variation of the techniques described in SUCC (Rangarajan and Fussell, 1991) and in SUSPECTS (Barbará and Fussell (1991). We assume that a MU contains n^* items in the cache and, of those, $f^* = \rho f$ items really need to be invalidated. That is, only a fraction ρ of the total number of items that need to be invalidated for all the MUs are present in the cache of a particular MU. (The actual value of ρ is irrelevant for our asymptotical analysis.) 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 given 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:

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

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

With this matrix in hand, the MU can run the following algorithm, where T is 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 becomes obvious when analyzing the probability of a false alarm.)

```
T = ∅
for j = 1 to m do
    if α_j = 1 then
        for i = 1 to n do
          if i ∈ S_j then
            count[i] := count[i] + 1
for i = 1 to n do
    if count[i] ≥ mδ_f then
    T := T ∪ i
```

Essentially, an item is declared invalid if it belongs to "too many" unmatching signatures (i.e., it is 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 is 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 that the behavior of the MU in each interval is independent of the behavior of 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 to answer the query. We assume that the query gets answered but the unit may decide independently to go to sleep.

We use the following notation:

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

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

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

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

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

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

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 to 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 these 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:

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

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 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 every time 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: (1) the last query on this item occurred exactly τ seconds ago, and (2) there have been no updates during the two queries.

The probability of the first event is simply the probability of the interarrival time being τ (i.e., $\lambda e^{-\lambda\tau}$). The probability of the second event is $e^{-\mu\tau}$. Therefore, M, H, R 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 ($B_C = 0$) or any intervals. However, we compute the number of queries that can be processed during an interval of duration L 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, $w = kL$. 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:

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

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

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

The average hit ratio for TS, h_{ts} is analyzed in Appendix 1. The upper and lower bounds for it are:

$$\frac{(1-p_0)\,u_0}{1-p_0\,u_0} - \frac{(1-p_0)\,u_0^{k+1}s^k}{1-p_0\,u_0} - \frac{(1-p_0)\,u_0^{k+1}s^k\,q_0}{(1-p_0\,u_0)^2} < h_{ts} \tag{17}$$

$$< \frac{(1-p_0)\,u_0}{1-p_0\,u_0} - \frac{(1-p_0)\,u_0^{k+1}s^k}{1-p_0\,u_0}$$

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:

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

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

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

The hit ratio h_{at} is analyzed in Appendix 2. Its equation is:

$$h_{at} = \frac{(1 - p_0)\, u_0}{1 - q_0\, u_0} \tag{20}$$

4.5 SIG

We begin by 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).

Second, 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. An 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 of a valid cache being in a signature that does not match is

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

If we now define 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 K_p in the algorithm.)

By results that can be found in Barbará and Lipton (1991) and Rangarajan and Fussell (1991), based in the Chernoff inequality (Chernoff, 1952), 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 diagnosis is simply $p_{nf} = 1$ - p_f.

In reality, we want the probability of any of the valid caches in the MU being falsely diagnosed to be smaller than a certain threshold, δ. That is, $(n^* - f^*)p_{nf} \leq \delta$. 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 Appendix 3. The equation is:

$$h_{sig} = \frac{(1 - p_0)\,u_0 p_{nf}}{1 - p_0\,u_o} \tag{26}$$

5. Asymptotic Analysis

This section analyzes the throughput of the techniques presented in extreme cases. The first analysis we 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 $\rightarrow$ 0	s $\rightarrow$ 1
q_0	$e^{-\lambda L}$	0
p_0	$e^{-\lambda L}$	1
h_{ts}	$(1-e^{-\lambda L})e^{-\mu L}$	0
h_{at}	$(1-e^{-\lambda L})e^{-\mu L}$	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 $s \rightarrow 0$), 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 $s \longrightarrow 1$), a behavior that we call "sleepers," the hit ratios of all the techniques 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 $1 - p_0 u_0$, 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 especially true for small values of u_0 (i.e., for low update rates).

We now show 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	$u_0 \longrightarrow 1$
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 diagnosis. 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 article. Each scenario corresponds to a set of values for the parameters involved.

Scenario 1. The first scenario corresponds to the following set of values:

λ	10^{-1} query/s
μ	10^{-4} updates/s
L	10 seconds
n	10^3
b_T	512
W	10,000 b/s
k	100
f	10
g	16

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.

Scenario 2. In the next scenario, we increase the size of the database and the bandwidth. The parameters are as follows (notice that we have also decreased the window size for TS):

λ	10^{-1} query/s
μ	10^{-4} updates/s
L	10 seconds
n	10^6
b_T	512
W	10^6 b/s
k	10
f	10
g	16

Figure 4 shows the results for this set of parameters. They are similar to those for scenario 1. The reduced window size ($k = 10$) makes TS stay competitive with the rest of the techniques (otherwise, the size of the report would be too large).

Scenario 3. For the third scenario, we have the following set of parameters:

Fig. 3. Effectiveness, Scenario 1

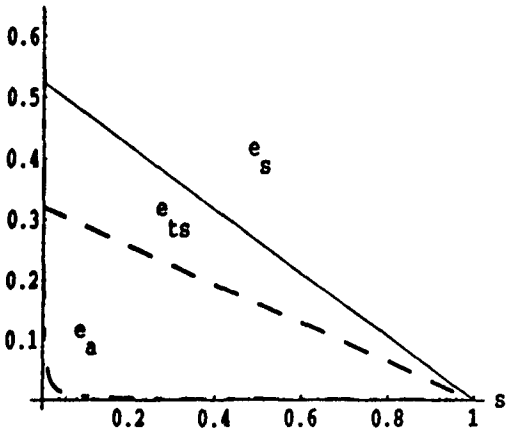
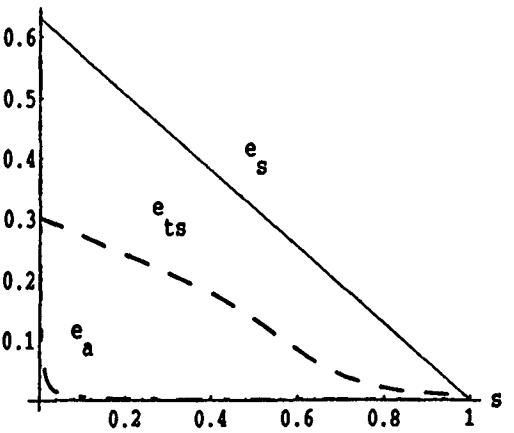


Fig. 4. Effectiveness, Scenario 2



λ	10^{-1} query/s
μ	10^{-1} updates/s
L	10 seconds
n	10^3
b_T	512
W	10,000 b/s
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 5 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 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.

Scenario 4 is similar to Scenario 3, but with a bigger database and higher bandwidth. Here is the set of parameters:

λ	10^{-1} query/s
μ	10^{-1} updates/s
L	10 seconds
n	10^6
b_T	512
W	10^6 b/s
k	10
f	200
g	16

The results for Scenario 4 are reported in Figure 6. We see that the effectiveness of AT is considerably reduced from the one obtained in Scenario 3. The reason for this is that the throughput for AT does not increase at the same rate that T_m does, thereby reducing the effectiveness. The strategy SIG, on the other hand, becomes more competitive for this scenario, being the choice for almost all the range of s values. As in Scenario 3, TS is not included because the size of the report exceeds L, rendering the technique unusable.

The fifth scenario is based on the following parameters:

λ	10^{-1} query/s
μ	0
L	10 seconds
n	10^3
b_T	512
W	10,000 b/s
k	100
f	1
g	16

This time, we vary the update rate from $\mu = 10^{-4}$ to $\mu = 210^{-4}$, and plot the results in Figure 7. 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.

Finally, Scenario 6 is based on the following parameters:

Fig. 5. Effectiveness, Scenario 3

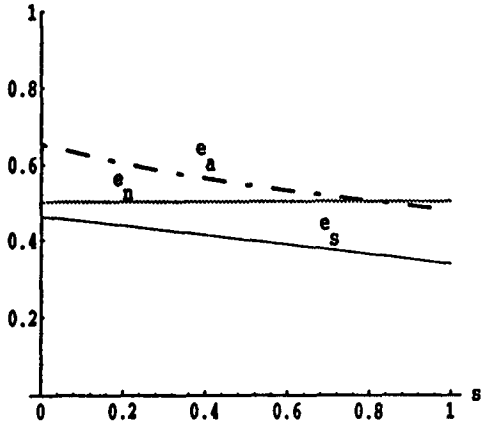
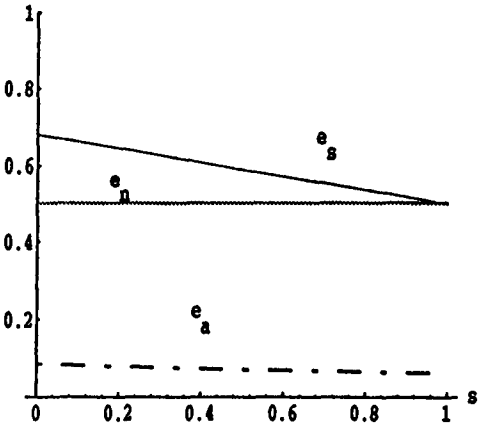


Fig. 6. Effectiveness, Scenario 4



λ	10^{-1} query/s
μ	0
L	10 seconds
n	10^6
b_T	512
W	10^6 b/s
k	10
f	10
g	16

The results, reported in Figure 8, are similar to those obtained in Scenario 5. Strategies AT and SIG are practically indistinguishable. Strategy TS degrades rapidly as the update rate increases.

7. Relaxing the Consistency of the Caches

If the applications supported by the system allow it, we could relax the consistency of the caches, thereby opening the door for shorter invalidation reports. For instance, if the MUs are caching stock prices, it may be perfectly acceptable to use values that are not completely up to date, as long as they are within 0.5% of the true prices. This can be accomplished by considering the cached values as quasi-copies of the values in the server (Alonso et al., 1990). A quasi-copy is a cached value that is allowed to deviate from the central value in a controlled way. Quasi-copies represent another form of obligation from the server. As long as the clients understand what the server is providing, the system can take advantage of this concept to reduce the size of the reports.

Fig. 7. Effectiveness, Scenario 5

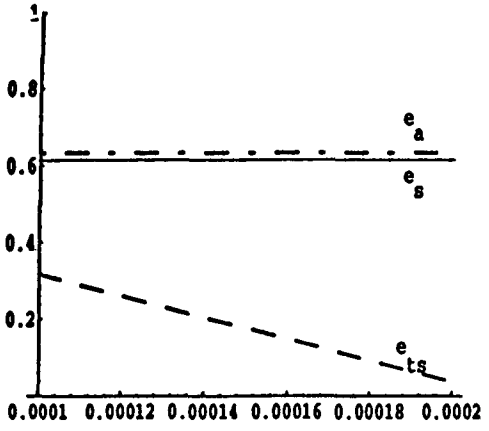
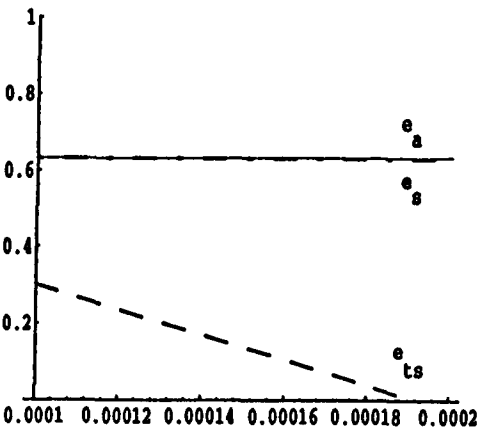


Fig. 8. Effectiveness, Scenario 6



One feature of a quasi-copy is the *coherency condition* attached to it. Once an object is kept cached as some unit, its coherency condition specifies the allowable deviations from the central copy. We are particularly interested in two conditions (taken from Alonso et al., 1990). The first one is the *delay condition*, which establishes how much time an image may lag behind the true value. Formally, for an object x, an allowable delay of α is given by the condition:

$$\bigvee \text{times } t \geq 0 \exists k \text{ such that } 0 \geq k \geq \alpha \text{ and } x'(t) = x(t-k) \qquad (27)$$

Where $x'(t)$ is the value cached at a unit at time t and $x(t)$ is the value at the server at t.

This coherency condition can be easily enforced by the MUs by simply dropping their cache of x every α seconds. On the other hand, if the MU drops x and the value of it has not changed at all, the unit may be querying the server unnecessarily the next time it needs x. So, a better way to weave this concept into our broadcasting techniques is the following.

For every item x in the database, the server keeps a vector $obligation_{list}(x)$ associated with it. The structure $obligation_{list}$ is built as a queue. The values stored in $obligation_{list}$ will be used to check whether the item should be included in the next report, in case it changes. For simplicity assume that $\alpha = jL$, that is, α is a multiple of the latency L. Then if x is reported at interval i, the value i is pushed into $obligation_{list}(x)$. If an MU queries the server for x at a time t, just before interval p, the value p is pushed into $obligation_{list}(x)$. Now, when it comes the time to build the report, besides the conditions expressed in the techniques described earlier, the server checks if the next interval is equal to $l + j$, where l is the first element in the queue $obligation_{list}(x)$. If so, x can be considered for reporting in case it also satisfies the normal conditions (e.g., in AT, the item has changed since the last report), otherwise, it need not be considered.

In the MU that is caching x, a timestamp $ts(x)$ marks the age of the cache. The cache is kept until:

- The value of x is invalidated by the report, or
- The cache is α seconds old. In this case, the unit waits for the next report. If x is there, it drops the cache, otherwise it keeps it and makes $ts(x)$ equal to the time of the current report.

The technique described above is bound to reduce the number of times x is reported.

A second coherency condition that is of interest to us is the *arithmetic condition* (Alonso et al., 1990). In it, if the value of the object is numeric, the deviations are limited to the difference between values of the object and its central copy. Formally,

$$\bigvee \text{ times } t \geq 0 \mid x'(t) - x(t) \mid \leq \epsilon \tag{28}$$

Again, if a deviation of ϵ is tolerable for the applications we are running, we could simply modify the strategies of Section 3 to report an item, but only if it changes more than the prescribed limit. This will also reduce the number of times the item is reported.

8. Adaptive Invalidation Reports

We have proposed a number of strategies for stateless cache invalidation. In this section, we propose an extension of one of these strategies to dynamically adjust to the changing query, update, and wake-up ratios of the environment.

Our technique will be based on the broadcasting timestamps (TS) strategy. Recall that, in the TS strategy, the server sends the timestamps of the latest change for items which have had updates in the last w seconds. The MU listens to the "report" and updates its cache by either throwing out an item that has changed according to the report (its timestamp is larger than the timestamp stored in the cache) or updates its timestamp to the timestamp of the latest report (if the item was not mentioned there). When awake, the unit always listens to the reports and updates the cache, even when it is not actively querying them.

The hit ratio, denoted by h_{TS}, depends, of course, on the window size w. The larger the window, the more "immune" the unit is to the sleep time. Note, that the window size has nothing to do with the latency between the two consecutive invalidation reports.

There are a number of important shortcomings with the TS method. To illustrate them, let us consider a scenario where a particular data item never changes. In this way, this data item will not be included in any invalidation report broadcasted after the timestamp (assuming that the time starts from zero). Therefore, if a client cached this data item earlier and then went to sleep, loosing some of the invalidation reports, he will have to submit an uplink request upon the next query. If the item i is queried very often by clients who are disconnected most of the time

(sleepers) there will be a lot of unnecessary uplink traffic. In the environment in which sleepers prevail, that would be a highly undesirable feature. In fact, for an item like this, the hit ratio could be equal to one, if the clients are adequately informed. Therefore, it makes sense to keep an "infinite" window for the item like this, including the pair $< i, 0 >$ is each invalidation report.[7] Hence: *If an item is queried very often by units that sleep a lot, it makes sense to extend the window size for such an item.*

Consider another extreme case. Suppose that some item j changes so often that the hit ratio is equal to zero (i.e., the clients have to request the item uplink upon each query). In the situation like this it makes no sense to include j in the invalidation report. Hence, the window for such an item should be equal to zero.[8]

Therefore: *If the hit ratio for a given data item is low even for units that do not sleep at all, then the item should not be included in the report.*

In summary, we should modify TS to have window size dependent on the data item. To do that, the server has to use some feedback from the clients to modify the window size accordingly. In the rest of this section, we present two approaches that use different feedback information.

8.1 Method 1

In this method, the clients send some extra information along with the queries that are sent uplink. The server uses this information to figure out how much, if any, to change the window size for each item.

Let us define the maximal hit ratio for an item MHR(i) as the hit ratio that a client caching that item would achieve if the client did not sleep at all (never got disconnected). If MHR(i) is high, *and* the actual hit ratio (AHR(i)) is lower due to the sleep time, then we will increase the window size. We will also *decrease* the size of the window if the MHR is low or if the overall *query activity for this item is low.*

Both the maximal hit ratio and the current hit ratio for a given data item can be determined by the server if it knows about all cache hits of the client. But how does the server get this information if the clients are satisfying the cache hits locally? When the uplink query about the item i is requested, the clients will piggyback all the timestamps of requests about that were satisfied locally from the time of the previous uplink request about i. In this way, the server, knowing the update history and the full history of queries, can compute both MHR(i) and AHR(i) (the actual hit ratio). In fact, we assume that the server will make this computation only periodically. The whole time scale will be divided into evaluation periods,

7. 0 is the time at the beginning of the time scale.

8. One has to be careful, though, during the "transition" period when clients may falsely conclude from the absence of this item in the report that it is unchanged.

which are multiples of the invalidation report latencies L. Hence, the reevaluation of the server's strategy, which results in the changes of individual window's sizes, will happen only once per evaluation period.

Let us now discuss what happens during the server's strategy reevaluation. We assume that the evaluation period is kL.

If MHR(i) > AHR(i) then there is room to improve. This can be done by increasing the window size for the item i. But is it worth it? If we increase the size of the window, we increase the overall cumulative size of the invalidation reports (since the particular will stay longer in the report).

Let

- $q[i]$ be the total number of queries about i that the server received during the last evaluation period. Notice that this parameter is not the number of queries that went uplink, but the total number of queries posed by the clients. The server receives this as part of the information that is piggybacked with the queries.
- AHR(i, *new*) be the actual hit ratio for i during the last evaluation period.
- AHR(i, *old*) be the actual hit ratio for i during the period preceding the last evaluation period.
- Report(i, *new*) be the number of times i was mentioned in the invalidation reports during the last evaluation period. This value, of course, will be an integer in the interval $[0,k]$.
- Report(i, *old*) be the number of times i was reported in the invalidation reports within the previous evaluation period. Again, this value is an integer in the interval $[0,k]$.

We now define

$$\begin{aligned} Gain\,(i) = {} & ((1 - \mathrm{AHR}\,(i,new)) - \mathrm{AHR}\,(i,old)))\,q\,[i]\,b_q - \\ & (\mathrm{Report}\,(i,new) - \mathrm{Report}\,(i,old))(log\,(n) + b_T) \end{aligned} \tag{29}$$

rearranging this equation, we have:

$$\begin{aligned} Gain\,(i) = {} & (\mathrm{AHR}\,(i,old) - \mathrm{AHR}\,(i,new))\,q\,[i]\,b_q - \\ & (\mathrm{Report}\,(i,new) - \mathrm{Report}\,(i,old))(log\,(n) + b_T) \end{aligned} \tag{30}$$

where n is a number of items in the database, b_q is the size in bits of the query that is sent uplink (on the average) and b_T is the size of the timestamp.

Gain (i) represents the overall difference in terms of the number of bits being send uplink and downlink, as a result of increasing the actual hit ratio at the expense of the larger invalidation report. If the gain is positive (above a certain threshold ϵ), then we increase the size of the window, otherwise we will decrease the size of the window.

Windows are modified according to the following formula, where ϵ is a small integer

$$w\,(new, i\,) = \begin{cases} w\,(old, i) + \epsilon & \text{if } Gain\,(i) \geq 0 \\ w\,(old, i) - \epsilon & \text{otherwise} \end{cases} \tag{31}$$

where $w(new, i)$ is a new window size, while $w(old, i)$ is the old window size.

We still have to answer what happens at the first evaluation period, when AHR(i, old) is undefined. We always start with the same window size $w_0(i)$ for all items. In the first reevaluation, we *increase* the size of the window for a given data item if the MHR(i) is larger then AHR(i); otherwise, we decrease the size of the window.

One can easily see that, in the case of the never or rarely changing data item, its window will increase steadily if the query rate is high, and the units sleep a lot. Also, in the reverse situation, if there so many queries and the maximal hit ratio is small, the window will eventually shrink to zero.

Notice that, given the history of prior query requests that have been satisfied locally (cache hits), those that had to go uplink (cache misses), and the history of updates, the server can determine a posteriori the optimal window size $w(i)$ for the item i. This size will minimize the sum of all invalidation report entries about the item i, plus the total size of the uplink requests that would be submitted if a given window w would be applied. We do not do this here, being aware of the dangers of data overfitting, and preferring the incremental dynamic approach.

8.2 Method 2

In this method, the clients do not send extra information with the queries. The server use a much coarser measure to determine if the window size needs to be changed.

Let $Q[i, new]$ be the number of queries that have been *asked* (i.e., sent uplink) by the clients during the last evaluation period and, consequently, $Q[i,old]$ be the number of queries in the previous interval. Let Report(i,new) and Report(i,old) be as before.

Then we define the gain in this method as:

$$Gain(i) = (Q[i, new] - Q[i,old])\, q[i]\, b_q - [\text{Report}(i,new) - \text{Report}(i,old)][log(n) + b_T] \quad (32)$$

And, as in Method 1, modify the window size using Equation 31.

Notice that the difference in the traffic uplink in this method is measured as the actual difference, but does not reflect the change in hit ratios. In other words, if a sudden, bursty activity over an item occurs, this method will wrongfully diagnose the need to change the window size for the item. In return for this coarser behavior, the method is less costly since it does not require the clients to piggyback any information with the queries or the server to compute the hit ratios.

9. Role of Different Network Environments

So far, we have discussed cache invalidation methods quite abstractly, without referring to any particular network environment. It is clear that, to be able to use

invalidation reports, the network has to support broadcasting mode. Additionally, it should be possible for the MSS to precisely control the timing to download the invalidation reports. In this section, we discuss how realistic this last assumption is, and what we can do in networks in which this is not the case.

In environments like Ethernet, which use a CSMA/CD type of a protocol, it is not easy to control the time in which the broadcast will be performed. Similarly, in CDPD (Cellular Digital Packet Data),[9] which is beginning to be deployed, there are no guarantees about the timing for any frame of digital information, since voice channels carry a higher priority than data. In such cases, we can resort to the use of the multicast mode as a way of "addressing" the invalidation reports. In this mode, the receivers are tuned to responding to a particular address. The client and the MSS simply agree on the multicast group address of invalidation reports. The MSS periodically downloads the invalidation report under the predefined multicast address, and the wireless client tunes to that address. The CPU of the MU can be in a doze mode, and needs to be awakened only when a message to that particular address arrives (i.e., the invalidation report). In this way, we avoid having to be active all the time, waiting for the report to be sent. Precise timing and synchronization are not important any more.

In Multiple Access Schemes, which are based on the concept of reservation, such as PRMA protocol developed at WINLAB (Nanda et al., 1991) or the MACAW (Bharghavan et al., 1994) protocol at Xerox, it is possible to guarantee synchronization on the downlink channel, and to guarantee the precise timing of delivery. Then the MU has to be awakened by a timer just before the invalidation report is about to be transmitted. (Some clock synchronization algorithm should be run by the system to guarantee a maximum deviation of the MU's clock with respect to the server's clock.)

In summary, the concept of invalidation reports is largely orthogonal to the specific networking environment. It is just the concept of the *address* of the report that changes, depending on the underlying network. The address could be either a *timestamp* or a *multicast address.*

10. Conclusions and Future Work

We have proposed three new cache invalidation methods suitable for the wireless environment with its high rate of client's disconnection. In all three strategies, the server periodically broadcasts the report that reflects the changing database state. We have categorized the mobile units into sleepers and workaholics on the basis of the amount of time they spend in their sleep mode. Different caching strategies

9. The basic idea of CDPD is that empty voice channels on existing cellular networks are used to send data packets by using a protocol similar to CSMA/CD, called Digital Sensing Multiple Access with Collision Detection (DSMA/CD).

turn out to be effective for different populations. Thus, signatures that 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 where 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 that rarely go to sleep and are awake most of the time.

The methods presented in this article are by no means exhaustive. There are 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 issues in a future article.

Caching is only one example of the wide range of classical systems issues that are expected to be affected by the mobile computing environment (Imieliński and Badrinath, 1992). In particular, we expect that data broadcasted over a wireless medium will be an effective way of disseminating information to a large number of users (Imieliński et al., 1993), since the cost of the wireless broadcast does not depend on the number of users. In this article, we made a step in this direction by demonstrating how broadcasting can be used for cache invalidation.

There are a number of possible improvements to the scheme presented in this article: 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 broadcasted, 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.

We have presented two methods for dynamically adjusting the invalidation report based in the information that the server gets from the mobile stations. One can easily see that this method will improve the behavior of the caches in cases such as a rarely changing item or a rapidly changing item.

Other methods can benefit from the same type of dynamic adjusting. For instance, in SIG, a dynamic strategy could change the way the combined signatures are formed, according to the individual demands for the items.

Finally, we would 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. 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.

Acknowledgments

We would like to thank K. Kaplan and J.L. Palacios for their help on developing the probabilistic models presented in this article, 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 article, and for carefully proofreading it.

References

Alonso, R., Barbará, D., and Garcia-Molina, H. Data caching issues in an information retrieval system. *ATM Transactions on Database Systems,* 15:359-384, 1990.

Barbará, D. and Lipton, R.J. A class of randomized strategies for low-cost comparison of file copies. *IEEE Transactions on Parallel and Distributed Systems,* 2(2):160-170, 1991.

Bharghavan, V., Demers, A., Shenker, S., and Zhang, L. Macaw: A media access protocol for wireless LANs. *Proceedings of the ACM SIGCOMM,* London, 1994.

Burrows, M., Abadi, M., and Needham, R. A logic of authentication. *ACM Transactions on Computer Systems,* 8(1):18-36, 1990.

Chernoff, H. A measure of asymptotic efficiency for tests of a hypotesis based on the sum of observations. *Annals of Mathematical Statistics,* 23:493-509, 1952.

Fuchs, W.K., Wu, K., and Abraham, J. Low-cost comparison and diagnosis of large remotely located files. *Proceedings of the Fifth Symposium on Reliability of Distributed Systems,* 1986.

Imieliński, T. and Badrinath, B.R. Querying in highly mobile and distributed environments. *Proceedings of the Eighteenth International Conference on Very Large Databases,* Vancouver, B.C., 1992.

Imieliński, T., Badrinath, B.R., and Viswanathan, S. Data dissemination in wireless and mobile environments. Technical Report 59, WINLAB, Rutgers University, 1993.

Imieliński, T., Viswanathan, S. , and Badrinath, B.R. Indexing on air. *Proceedihgs of the ACM-SIGMOD International Conference on Management of Data,* Minneapolis, MN, 1994.

Madej, T. An application of group testing to the file comparison problem. *Proceedings of the International Conference on Distributed Computing Systems,* 1989.

Mummert, L., Wing, J.M., and Satyanarayanan, M. Using belief to reason about cache coherence. *Proceedings of the ACM Sigact-Sigops Symposium on the Principles of Distributed Computing,* 1994.

Nanda, S., Goodman, D., and Timor, U. Performance of PRMA: A packet voice protocol for cellular systems. *IEEE Transactions on Vehicular Technology,* 40(3) 1991.

Nitzberg, B. and Lo, V. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer,* 24(8) 1991.

Rangarajan, S. and Fussell, D. Rectifying corrupted files in distributed file systems. *Proceedings of the International Conference on Distributed Computing Systems,* 1991.

Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the Sun network filesystem. *Proceedings of the USENIX Summer Conference,* 1985.

Satyanarayanan, M., Howard, J.H., Nichols, D.N., Sidebotham, R.N., Spector, A.Z., and West, M.J. The ITC distributed file system: Principles and design. *Proceedings of the Tenth ACM Symposium on Operating Systems Principles,* 1985.

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

Figure 9. Scenario I

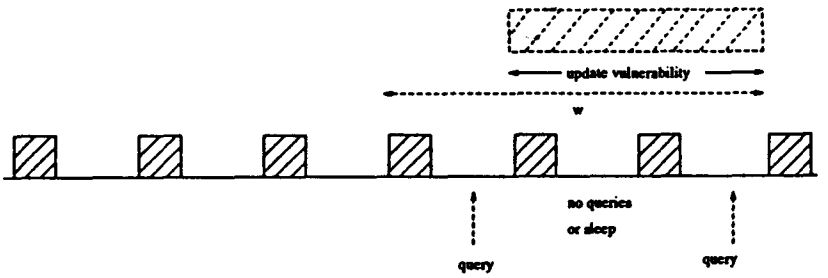


Figure 10. Scenario II

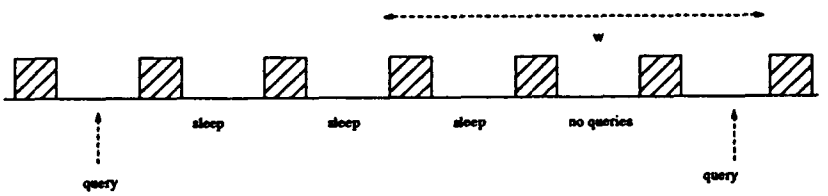
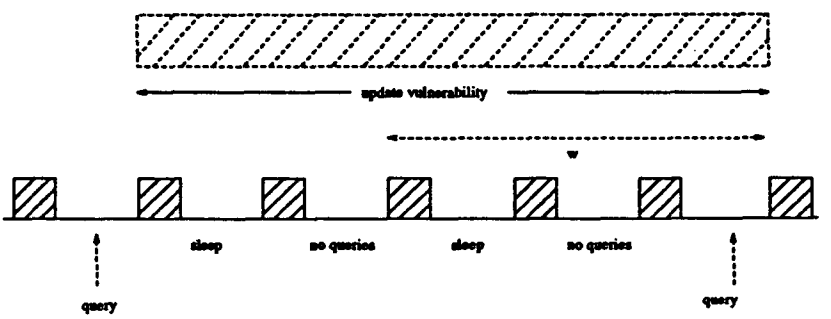


Figure 11. Scenario III



Appendix 1

To compute the hit ratio, we assume that a query has occurred at a particular instant in time, and we compute the conditional probability of the value in the cache being valid. To illustrate the issues, let us consider three scenarios in Figures 9, 10, and 11.

Figure 9 shows a scenario in which the last query occurred within the window w

of the current query. We will have a miss if an update occurred within the shaded region indicated in the Figure. Notice that the MU could have disconnected in the periods between the two queries with no effect to the hit probability.

Figure 10 shows a scenario in which the last query occurred in a period beyond k intervals before the current query. In this case, if the unit has gone to sleep for k periods, as shown in the Figure, the MU would have no way of knowing whether the cache is invalid, and would have to declare a miss (independent of whether an update occurred between the two queries).

Figure 11 shows another scenario in which the last query occurred in a period beyond k intervals before the current query. However, in this case, the MU was not disconnected for k or more periods. Therefore, the query will be a hit if there are no updates during the shaded region indicated in the figure.

Now we can divide the cases of study in two: when the number of periods between the two queries is less than k, and when it is larger than k. Let us begin with the first case.

Let the two queries occur at i intervals of each other, with $1 \leq i \leq k$. Then, all we need for the second query to be a hit is no updates during i periods. The probability of a hit in this case becomes $(1 - p_0)p_0^{i-1}u_0^i$. The first term is the probability of the first query; the second term the probability of having $(i - 1)$ periods of no queries in the intervals between the queries; and the last term the probability of no updates during i intervals.

For the second case, the queries occur at i intervals of each other, with $k < i$. To have a hit in this case, the unit cannot sleep k or more consecutive intervals, and we should not have updates during i intervals. Let p_{ki} be the probability of sleeping k or more consecutive intervals, given that the queries happen at i intervals of each other. This probability is difficult to compute, but we present expressions for the upper and lower bounds. An upper bound can be expressed as:

$$p_{ki} \leq s^k p_0^{i-1-k} + (i - 1 - k)q_0 s^k p_0^{1-2-k} \tag{33}$$

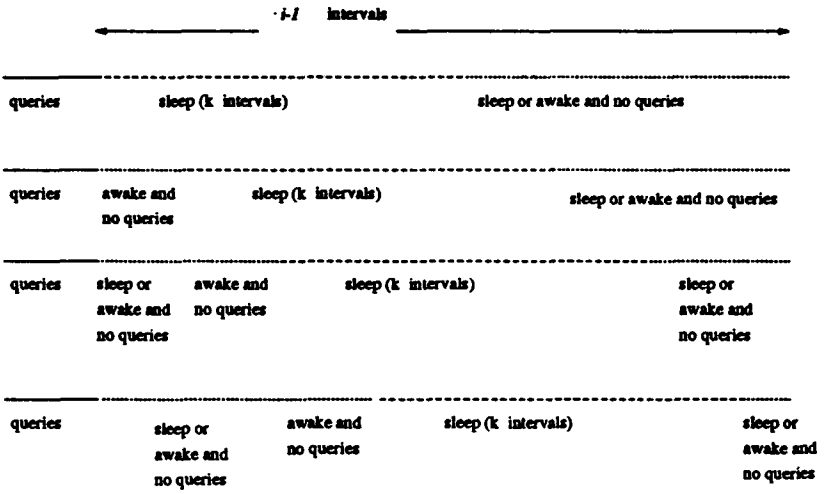
To see the justification for Equation 33, consider Figure 9, which shows the way in which the $i - 1$ intervals can contain a "sleeping" streak of k intervals or more.

Since the probability of having no queries (being asleep or posing no queries while awake) during $i - 1$ intervals is $p_0^{(i-1)}$, the probability of having no queries and no "sleeping" streak of k intervals during these $i - 1$ intervals is $p_0^{(i-1)} - p_i^k$. Multiplying this probability by u_0^i, we get the probability of a hit ratio for the second case.

Therefore,

$$h_{ts} > \sum_{i=1}^{k} ((1 - p_0)p_0^{i-1}u_0^i) + \sum_{i=k+1}^{\infty} (1 - p_0)(p_0^{(i-1)} - (s^k p_0^{i-1-k} + (i - 1 - k)q_0 s^k p_0^{i-2-k}))u_0^i \tag{34}$$

Figure 12. Sleeping streak



Manipulating Equation 34, it becomes:

$$h_{ts} > (1 - p_0)u_0 \sum_{i=0}^{k-1} (p_0u_0)^i + (1 - p_0)u_0 \sum_{i=k}^{\infty} (p_0u_0)^i - (1 - p_0)u_0s^k \sum_{j=0}^{\infty} ((p_0^j u_0^{j+k}) + \frac{q_0}{p_0} (jp_0^j u_0^{j+k})) \quad (35)$$

And,

$$h_{ts} > \frac{(1-p_0)u_0}{1-p_0u_0} - \frac{(1-p_0)u_0^{k+1}s^k}{1-p_0u_0} - \frac{(1-p_0)u_0^{k+1}s^kq_0}{(1-p_0u_0)^2} \quad (36)$$

For the other bound, it is enough to see that:

$$p_{ki} > (i - 1 - k)\ s^k q_0^{i-1-k} \quad (37)$$

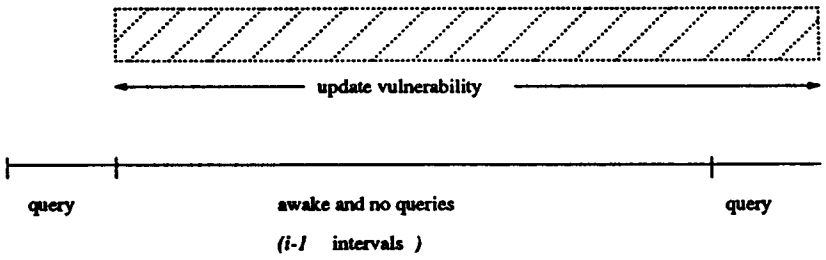
Therefore,

$$h_{ts} > \sum_{i=1}^{k} ((1 - p_0)p_0^{i-1}u_0^i) + \sum_{i=k+1}^{\infty} (1 - p_0)(p_0^{(i-1)} - (i - 1 - k)s^k q_0^{i-1-k})u_0^i \quad (38)$$

Manipulating this equation we have:

$$h_{ts} < \frac{1-p_0)u_0}{1-p_0u_0} - \frac{(1-p_0)u_0^{k+1}s^k}{1-q_0u_0} \quad (39)$$

Figure 13. Scenario for AT



Appendix 2

To compute the hit ratio for AT, it is enough to consider the event in the last query, which happened i intervals before the current one. Also, the unit should not have been disconnected at any of the $i - 1$ intervals between the queries (otherwise, the MU would have dropped its cache altogether). Also, there can be no updates in the last intervals (see Figure 10).

Thus, the equation for the hit ratio becomes:

$$h_{at} = (1 - p_0) \sum_{i=1}^{\infty} q_0^{i-1} u_0^i \tag{40}$$

or simply:

$$h_{at} = \frac{(1-p_0)\, u_0}{1-q_0\, u_0} \tag{41}$$

Appendix 3

The analysis is similar to the one made for AT. Here, to have a hit, the MU must not issue any queries between the last one and the current one, separated by $i - 1$ intervals. It does not matter whether the unit sleeps. Also, no updates must occur during the i intervals. Finally, the diagnosis must be correct. This last event happens with probability p_{nf} (Equation 22).

So, the equation becomes:

$$h_{sig} = (1 - p_0) \sum_{i=1}^{\infty} p_0^{i-1} u_0^i \tag{42}$$

or simply:

$$h_{sig} = \frac{(1-p_0)\, u_0\, p_{nf}}{1-p_0\, u_0} \tag{43}$$