Wireless Personal Communications (2005) 34: 227–253
DOI: 10.1007/s11277-005-2845-2

Update Propagation Algorithms for Supporting Disconnected Write in Mobile Wireless Systems with Data Broadcasting Capability

ING-RAY CHEN[1], NGOC ANH PHAN[1] and I-LING YEN[2]

[1]*Department of Computer Science, Virginia Tech*
E-mail: irchen@vt.edu, nphan@vt.edu
[2]*Department of Computer Science, University of Texas, Dallas*
E-mail: ilyen@utdallas.edu

Abstract. We develop and analyze algorithms for propagating updates by mobile hosts in wireless client–server environments that support disconnected write operations, with the goal of minimizing the tuning time for update propagation to the server. These algorithms allow a mobile host to update cached data objects while disconnected and propagate the updates to the server upon reconnection for conflict resolutions. We investigate two algorithms applicable to mobile systems in which invalidation reports/data can be broadcast to mobile hosts periodically. We show that there exists an optimal broadcasting period under which the tuning time is minimized for update propagations. We perform a comparative analysis between these two update propagation algorithms that rely on broadcasting data and an algorithm that does not, and identify conditions under which an algorithm should be applied to reduce the total tuning time for update propagation by the mobile user to save the valuable battery power and avoid high communication cost. For real-time applications, we address the tradeoff between tuning time and access time with the goal to select the best update propagation algorithm that can minimize the tuning time while satisfying the imposed real-time deadline constraint. The analysis result is applicable to file/data objects that mobile users may need to modify while on the move.

Keywords: wireless mobile systems, data broadcasting, disconnected operations, performance analysis, mobile client–server systems

1. Introduction

A mobile host (MH) performing read/write operations to a remote server can voluntarily disconnect itself from the server to save its battery life and avoid high communication prices in wireless networks. Before disconnection, the MH can prefetch into its local cache frequently-used objects of various formats based on the MH's specification or the past access history [1]. During disconnection, the MH accesses these prefetched objects locally. When the MH is reconnected to the system, the updates to the cached objects can be reintegrated back to the server to resolve conflicts with updates performed by the server (from other sites). In the literature, the three phases of supporting disconnected operations are termed *hoarding*, *disconnection* and *reintegration* [2, 3], respectively.

Over the past few years, various algorithms have been proposed to support disconnected operations in wireless mobile environments in these three phases [4, 5, 6, 2, 7, 8]. However, most existing algorithms assume read-only operations during disconnection. To support object write/create operations during disconnection, the Caubweb project [8] proposes modifying contemporary web browers/servers. The basic idea is to support disconnected updates via a client proxy running on the MH side to cache staging updates while disconnected, and a script running on the server to accept requests from the client proxy upon reconnection. ARTour Web Express [4] and ROVER [9] both allow staging updates to be incrementally and asynchronously flushed back to the server to support intermittent or weak connections. The BAYOU system [10] proposes the notion of application-specific conflict resolution to allow application-specific merge algorithms to be applied when update conflicts are detected. Coda [7, 11] also provides mechanisms to resolve update conflicts upon reintegration to support disconnected operations on files.

None of these systems above addresses the issue of *when* a disconnected MH should be reconnected to the server. In our previous work [12], we developed a multiple web-page update propagation algorithm for supporting disconnected write operations in the context of web applications to address this issue. We identified *when* a disconnected MH should be reconnected to the server (after the prefetch phase) to propagate the web-page updates so that the system's performance in terms of the communication cost is optimized during the reintegration phase. The algorithm developed assumes that a MH communicates with its server *on-demand* via a service channel regardless of if the server may periodically broadcast information to all MHs regarding the validity of cached data objects. In this paper, we present a modified version of this web-page update propagation algorithm to deal with general file/data objects for supporting disconnected operations. Throughout the paper we will call it the "Query on Reconnection" or QOR algorithm for short.

In cases where the server can periodically broadcast invalidation reports containing information regarding objects having recently been updated using a broadcast channel, the QOR algorithm may incur unnecessary costs since the MH can read the invalidation report and/or changed objects from the broadcast channel to know whether its cached objects are still valid before propagating updates to the server.

The use of broadcast to disseminate data in wireless networks [13, 14, 15, 16] has received considerable attention due to the desirable scalability property to allow a large population of mobile users to concurrently read data broadcast by the server in the air. The general model is well known and there has been extensive analysis of broadcast channels and whether the server should broadcast only invalidation reports or additionally should also broadcast changed data objects [17, 18]. Most of the work concerns with the organization of broadcast data, i.e., when and how to broadcast particular data objects such that the access time for data retrieval (the response time for data access) from broadcasts is minimized. Furthermore, most work assumes read-only access by the MH with updates being performed at the server.

Datta et al. [13] proposed to dynamically alter the broadcast content depending on the patterns of client demand. In addition to the access time performance metric, they also proposed to use the tuning time (the duration of time a MH actively listens) to measure power conservation of a MH. To minimize the tuning time, Tan et al. [15] proposed two algorithms that exploit selective tuning, namely, Selective Dual-Report Cache Invalidation and Bit-Sequences with Bit Count. To facilitate the invalidation of cache contents, a server can periodically broadcast cache invalidation reports that contain information regarding data objects that have been updated in the last w broadcasting interval, with w being a design parameter. Kahol et al. [16] proposed an algorithm called Asynchronous Stateful (AS) that uses asynchronous invalidation messages and buffering of invalidation messages from the servers at the mobile host's Home Location Cache to support disconnection. Recently, prototypes based on data broadcasting to mobile devices have been developed and reported in the literature. Liebmann et al. [19] at Vienna University of Technology, Vienna, Austria, implemented a prototype based on J2EE that integrates push-based data dissemination with pull-based data requests where a back-end server broadcasts data requested to all edge devices instead of replying directly to the initiating devices. DaSilva et al. [20] at Virginia Tech developed a prototype based on UDP broadcasting to iPAQ devices for a stock-quote application to empirically analyze performance characteristics of *flat* versus *indexed* disk structure in broadcast-based database services and to demonstrate the scalability property.

While the issue of supporting disconnected write operations was not addressed by these papers, we draw upon the existing work in utilizing the same set of performance metrics (i.e., tuning time and access time with a different context – see Section 2.5) and designing update propagation algorithms based on the way broadcast data are organized and retrieved by a MH. Then we apply similar analysis methods adjusted to evaluate update propagation algorithms proposed for mobile wireless systems with data broadcasting capabilities. The novel aspect of our work is that we address the issue of how to support disconnected write operations by means of update propagations in the reintegration phase in systems with data broadcasting capability with the goal to minimize the tuning time. For real-time applications, we address the tradeoff between tuning time and access time with the goal to select the best update propagation algorithm that can minimize the tuning time while satisfying the imposed real-time deadline constraint.

Another group of work [21, 22] focused on transaction processing in mobile broadcast-based data dissemination systems. However, the bulk of research has been on the design of concurrency control algorithms for efficient processing of read-only transactions at the mobile clients. Lately, the work by Lee et al. [21] considered the design of algorithms to support update transactions performed by mobile clients. To some extent, the transaction processing algorithm presented in their work can be applied to support update propagations of a group of prefetched data items as a set of update transactions since an update transaction performed by a MH can be structured to contain only a single write operation on a data item. Their algorithm, however, deals mainly with the goal of efficient processing of online concurrent mobile transactions to detect early data conflicts and avoid transaction aborts and thus does not provide efficient supports for disconnected operations. Our work deals specifically with disconnected operations for data objects without transactional semantics, with the goal of minimizing the total time to propagate updates of all objects from the MH to the server so as to minimize the communication cost and save the MH's valuable battery power.

In this paper, we develop two update propagation algorithms applicable to systems with data broadcasting mechanisms to support disconnected operations. The first algorithm facilitates cache invalidation through the use of invalidation reports. We call it the "Invalidation Only" or INV algorithm. The second algorithm uses both invalidation reports and data broadcasting. We call it the "Invalidation Plus Broadcasting" or I + B algorithm. We compare these two algorithms with the QOR algorithm and for each algorithm we answer the questions of: (1) when to propagate and (2) how much *access time* and *tuning time* would be required to propagate a set of objects prefetched by a MH for supporting disconnected operations. The analysis result is useful for a MH to select the best update propagation algorithm to apply in order to minimize the tuning time when given a set of identified conditions. Here we note that even in the presence of data broadcasting, the MH does not necessarily have to read data from the broadcast if it discovers that the QOR algorithm will yield the best access or tuning time.

The rest of the paper is organized as follows. Section 2 states the system assumptions with a system model. Section 3 describes algorithms QOR, INV and I + B developed for supporting update propagations in mobile client–server environments, along with performance metrics used to assess the performance of these algorithms. Section 4 develops analytical models and shows numerical data to compare the performances of these three algorithms and identify conditions under which one algorithm performs the best among all. It also analyzes the tradeoff between tuning time and access time in these algorithms and illustrates how to apply the analysis results to select the best algorithm that can minimize the tuning time while satisfying a real-time deadline constraint. Finally, Section 5 concludes the paper and outlines possible future research areas deriving from this work.

2. System Model

2.1. Assumptions

We assume that a number of objects will be prefetched and stored in the MH's cache during the prefetching phase. Some of these objects are read-only, while others may be updated by the MH during the disconnection phase if needed to facilitate distributed authoring. We assume that a prefetching policy exists to determine which objects are to be prefetched, e.g., based on a prediction algorithm [6].

We assume that for each object, the MH also obtains from the server some update history information in terms of a general parameter, i.e., the update rate of that object by all users of the system. This information can be collected by the server readily by monitoring the update history of the object. For a cached object i, we denote this parameter as λ_i^w. In addition, we assume that the MH has some idea of how often it is going to perform updates on each cached object. For each cached object i, we call this parameter as λ_i. For read-only cached objects, $\lambda_i = 0$.

We assume that the server is located somewhere in the fixed network and is not moved during a session. A MH communicates with the server via an intelligent server gateway located on the fixed network, e.g., it can be just the base station. Further, the communication time on the fixed network is negligibly small compared with that on the wireless link. This assumption is justified for future high-speed wired networks. We assume that the differencing technique is used between the server gateway and the MH to propagate updates. That is, instead of transmitting the whole object from/to the client/server gateway, only the differences between two versions of the object are transmitted to save the communication cost.

We do not address the issue of channel reservation [23] in this paper. We assume that the system has a number of service channels available and that whenever a MH requests a service channel for update propagation, the system is able to allocate one. When the broadcasting mechanism is available, we assume there exists a relatively high-bandwidth broadcast channel (compared with service channels) available for the server to broadcast invalidation reports and/or data to all MHs. The QOR algorithm will use the service channel allocated to it upon reconnection for update propagation, while the INV and I + B algorithms will read data from the broadcast channel in addition to using service channels allocated to them for update propagation. All other resource requirements are the same. We assume each MH can voluntarily disconnect and reconnect at will. When given identified network and workload conditions, each MH has freedom to select the best update propagation algorithm in order to save its battery power and communication cost.

While it is possible that optimization algorithms based on caching, transcoding and differencing may be used by the server gateway to minimize the volume of data sent over the wireless network, we will assume that two general cost parameters suffice for our analysis.

For the service channel, T_{diff} is the average one-way communication cost of transmitting the differences along a service channel and T_{ack} is the average one-way communication cost of transmitting a simple request, reply or acknowledgement along a service channel. These parameters can be estimated by knowing more specific parameter values of the wireless network under consideration. Let s_r be the average size of a simple acknowledgement/reply. Let s_o be the average size of an object. Let p_m be the average fraction of any object being modified. Let B_d be the bandwidth of the broadcast channel used by the server to broadcast data and invalidation reports. Let B be the bandwidth of the service channel used by the MH to propagate updates to the server with $B_d \geq B$. Assume that the communication time in the fixed network is relatively small compared with that in the wireless network. Then, T_{diff} and T_{ack} can be estimated as

$$T_{\text{diff}} = \frac{p_m s_o}{B} \tag{1}$$

$$T_{\text{ack}} = \frac{s_r}{B} \tag{2}$$

From the MH's perspective, T_{diff} accounts for the time for transmitting the update request that carries the version number of the original cached object and the differences between the latest version and the original prefetched version; T_{ack} accounts for the time for transmitting a request from the MH to the server. From the server's perspective, T_{diff} accounts for the time for transmitting the updates (differences of the rejected items) from the server to the MH, and T_{ack} accounts for the time for transmitting a reply from the server to the MH.

2.2. Query on Reconnection Algorithm

In this algorithm we assume that when the MH reconnects to the server, it enters into the reintegration phase during which it propagates all staging updates performed during the disconnection phase in a *batch* mode to the server to resolve update conflicts. For each modified object, the MH submits the differences between the original object prefetched from the server and the latest version that it modifies, as well as the version number associated with the original version, to the server. The server checks to see if the object has been modified during the MH's disconnection period by comparing the version number of the object it currently keeps with the version number submitted to it by the MH. If they are the same, the server will accept the update request and modifies the object accordingly based on the differences received from the MH; otherwise, the update request of the object is rejected.

If an update request is rejected, we assume that the MH will stay connected and will issue a request to write lock the object to prevent others from accessing the object. After the differences relative to the MH's original version before updating are sent to the MH, the MH will regenerate a new version. Then, the MH will apply an application-specific merge algorithm such as the UNIX diff3 program to resolve the update differences. After the update is done, the MH will propagate the changes to the server by means of differencing again and will then release the write lock. It is assumed that the MH will stay on-line performing the merge operation and update propagation in this algorithm, i.e., the case in which the MH locks the page and then goes away will not occur. If the MH is forced to be disconnected because of environment changes (e.g., due to roaming), we assume that the lock will be broken by the server after a timeout period. This can be implemented by the server by attaching a timestamp to the lock and releasing the lock after a timeout period expires. The MH upon reconnection will discover that it does not own the lock any more and will retry again by repeating part of the update propagation algorithm.

2.3. Invalidation Only Algorithm

In this algorithm, the server periodically broadcasts an invalidation report. Each item of the invalidation report consists of an ID and a timestamp. Let s_i be the size of an item in the invalidation report. Let N_{db} be the size of the database. The MH under this algorithm is disconnected from the server and is reconnected to the server only at the beginning of each broadcasting period to read the invalidation report before propagating updates. By comparing the id of the item in the invalidation report and the id of the item in the cache, it knows what items have been updated during the last invalidation period. The MH then propagates the updates of those items that are still valid to the server. For the items that have been invalidated, the MH sends a service request to the server. Upon receiving the request, the server will send the differences in reply. The MH after receiving the differences is then disconnected during which merge operations are performed in the background. This update propagation process then is repeated in subsequent broadcast intervals until all items in the cache are propagated. Let T_{rep} be the time to read a data item in an invalidation report. Then,

$$T_{\text{rep}} = \frac{s_i}{B_d} \tag{3}$$

2.4. Invalidation Plus Broadcast Algorithm

In this algorithm the server periodically broadcasts an invalidation report followed by new values of data items that have been updated in the last broadcast interval. The invalidation report consists of entries of updated items each with an ID, a timestamp, and a pointer that points to the new value. The MH under this algorithm again is disconnected from the server and is reconnected to the server only at the beginning of each broadcasting period to read the invalidation report, and if any data items yet propagated by the MH have been invalidated, new values of invalid cached data items. Specifically, by reading the invalidation report, the MH knows what cached items have been updated in the last invalidation interval. It then uses the pointers to retrieve the new values of those data items. For those data items that have been updated by the MH but are still valid, the MH propagates them to the server. For those data items that have been invalidated, the MH retrieves the new values from the broadcasting data. Then the MH is disconnected again during which new values are merged with updated data items and write operations are allowed to be performed on the remaining data items. The update propagation process then repeats in subsequent broadcasting intervals until all items are updated and propagated. Let s_b be the size of an item in the invalidation report. Due to extra information, the size of s_b is larger than the size of s_i in the INV algorithm. The time to read an object in an invalidation report is estimated as:

$$T_{\text{rep}} = \frac{s_b}{B_d} \tag{4}$$

Let T_{read} be the time to read a data item from broadcast. Then,

$$T_{\text{read}} = \frac{s_o}{B_d} \tag{5}$$

2.5. Performance Metric

The objective of the paper is to design and analyze update propagation algorithms for supporting write operations in mobile client–server environments and to identify the condition under which one algorithm can perform the best among all in terms of minimizing the total time to propagate updates made to the cached data items by the MH to the server. As the objective of update propagation algorithms is to minimize the communication cost and to save the battery power of the mobile device, the primary metric of interest is the total *tuning time* to propagate all updates from the MH to the server based on versioning and data broadcasting mechanisms supported by the server. Specifically, the tuning time is the update propagation time while the MH occupies a service channel. It represents the battery consumption time of the MH and, less the processing time at the MH, also reflects the communication cost. Another metric we are looking at is the total *elapsed time* taken for propagating updates. The elapsed time is different from the tuning time in that it includes the time duration in which the MH is disconnected from the server before all updates are propagated. For real-time applications with a deadline to propagate updates from the MH to the server, the elapsed time is an important metric.

The main goal of the update propagation algorithms developed in the paper is to minimize the tuning time, thereby saving the wireless bandwidth and power consumption of the MH. In the QOR algorithm, this is achieved by finding the longest disconnection period such that the total communication cost for propagating updates at the reconnection time is minimized. In the INV and I + B algorithms, this is achieved by having the MH disconnect and reconnect in cycles to reload new values and propagate updates of cached data items such that the total tuning time is minimized. For real-time applications with a deadline by which updates must be propagated, the goal is to minimize the tuning time while satisfying the real-time requirement such that the total elapsed time taken for update propagation is less than the deadline.

Here we should emphasize that we intentionally exclude (from the communication time metric above) certain costs that will always incur irrespective of when the MH is reconnected to the server, e.g., the connection set-up time, the server processing time, etc. since adding such cost terms to the cost objective function doesn't affect the outcome of the analysis. Also, while it is possible that the MH can employ heuristics to possibly make more intelligent decisions about when it should reconnect to the server to adapt to resource changes (e.g., wireless bandwidth and server load changes), we will not consider this possibility in the paper.

2.6. Parameter

Table 1 gives the notation of parameters considered in the paper. The parameter list consists of three groups: (a) wireless mobile system parameters (B and B_d), (b) application parameters (λ_i, λ_i^w, L, L_b, N_{db}, N, s_o, s_r, s_i, s_b, p_m, and D_m) and (c) computable parameters (T_{diff}, T_{ack}, T_{rep}, T_{read}, r_i, p_i, and C). In particular, we note that all application parameters except L and N (determined by the MH at runtime) can be supplied through statistical means from the server to the MH prior to disconnection so as to allow the MH to compute C under QOR, INV and I + B to determine the best algorithm for update propagation of disconnected writes.

Table 1. Parameter list

B	Bandwidth of a service channel
B_d	Bandwidth of the broadcast channel
λ_i	Update rate for an object i by the MH
λ_i^w	Update rate for an object i by the world (including by the MH)
L	Length of the disconnection period by the MH
L_b	Length of the broadcast interval by the application
N_{db}	Size of the database
N	Number of items prefetched by the MH
s_o	Average size of a cached object
s_r	Average size of an acknowledgement or a reply
s_i	Average size of an item in the invalidation report for Invalidation only algorithm
s_b	Average size of an item in the invalidation report for Invalidation plus Broadcast algorithm
p_m	Fraction of an object being modified by the MH
D_m	Average time to execute a merge algorithm to resolve update conflicts
T_{diff}	Average one-way wireless communication time to propagate differences of two versions of an object via a service channel
T_{ack}	Average one-way wireless communication time to send an Ack/reply via a service channel
T_{rep}	Average time to read a data item from the invalidation report
T_{read}	Average time to read a broadcast data item
r_i	Probability that object i is updated by the server within a time period of L
p_i	Probability that object i is updated by the MH within a time period of L
C	Total cost (tuning time) for propagating updates to the server upon reconnection

3. Algorithms and Cost Analysis

In this section, We model and analyze the QOR, INV and I + B algorithms. We first derive expressions for r_i and p_i as defined in Table 1. Suppose that the length of the disconnection period (for the QOR algorithm) or the broadcast interval (for the INV and I + B algorithms) is L. Upon reconnection, the MH will attempt to propagate the update of a modified object. The update request will be denied, however, if the server has modified the object during L. Thus, r_i, the probability that an update request for object i is rejected by the server upon reconnection after the MH has been disconnected for a period of L, is given by

$$r_i = 1 - e^{-(\lambda_i^w - \lambda_i)L} \tag{6}$$

where we assume that updates to object i arrive at the system as a Poisson process, with rates λ_i and λ_i^w by the MH and by the world, respectively.[1]

Since we are interested in estimating the cost of propagating updates to the server after reconnection, we like to know the probability that an object has been modified by the MH at the end of the disconnection phase. Suppose that the length of the disconnection phase (for the QOR algorithm) or the broadcasting interval is L (for the INV and I + B algorithms) again.

[1] Note that we don't necessarily have to use the Poisson arrival assumption to model the server/client update processes. It is used only to facilitate the estimation of r_i and p_i; other statistical means based on history or user profile information can be used to estimate these two probability parameters.

Then, p_i, the probability that object i is updated by the MH within a period of L, is simply given by

$$p_i = \text{Prob}\{\text{updatetime} \leq L\} = 1 - e^{-\lambda_i L} \tag{7}$$

3.1. Query on Reconnection (QOR) Algorithm

In the QOR algorithm, we consider the case in which the MH prefetches a set of objects before disconnection, say, based on a prediction algorithm. All updates occurring before L are staged and later propagated back to the server at time L relative to the beginning of the disconnection phase. When the MH reconnects at time L, the server and the MH execute the QOR algorithm to propagate updates in a *batch* mode in order to shorten the reconnection time. The MH stays online until all data items are updated and propagated. Thus, the tuning time is the reconnection time. The design goal of the QOR algorithm is to shorten the tuning time.

Figure 1 illustrates the steps taken to execute the algorithm:

1. The MH sends to the server a bit vector A (carrying 0 or 1 values) indicating which cached objects have been updated by the MH, and also a value vector B (carring integer values) indicating the original version numbers associated with the cached objects prefetched into the MH before disconnection. This is done by sending a single message from the MH to the server. The time needed for this step is T_{ack}. Note that the server does not need to keep track of which objects are cached at the MH since it only needs to examine vector A to know what objects are cached at the MH (for those marked with bit 1), and vector B to know their version numbers.
2. The server decides accepting or rejecting updates based on vectors A and B received from the MH, and the current version numbers associated with the requested objects stored in

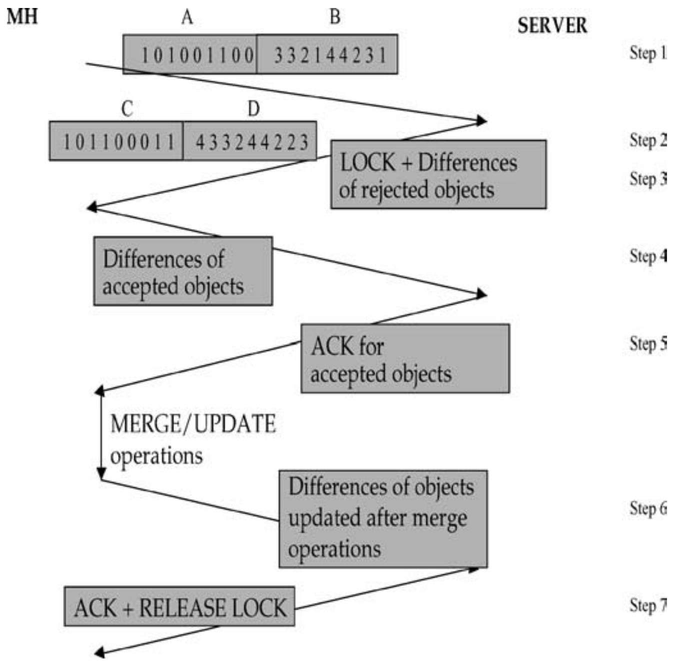

Figure 1. Steps taken by the query on reconnection algorithm.

the server. It then sends a bit vector C indicating which objects can be accepted (for which the value is 0) and which objects are to be rejected (for which the value is 1). Those objects not updated by the MH have their corresponding values also marked with 1 in vector C. The server also sends a separate version number vector D indicating the current version numbers of the objects stored in the server. All the above information are embedded in a single message sent from the server to the MH. The time for this step is also T_{ack}.

3. For those objects accepted by the server, the server sets a write lock on them and will commit to the updates as soon as it receives updates from the MH. For those objects not accepted by the server, the server also locks those objects to prevent them from being updated by other users in the system. Simultaneously, the server sends to the MH the differences of those objects rejected by the server (because the server has updated those objects) and also those objects that have not been updated by the MH but have been updated by the server. The time for this step is

$$\sum_{i=1}^{N} r_i T_{\text{diff}}$$

because on average $\sum_{i=1}^{N} r_i$ objects (out of N) are updated by the server during [0, L], for which the server must send the differences to the MH so that the MH can perform forced updates on the newest version.

4. When the MH receives vectors C and D, it knows immediately which objects are accepted by the server. Those accepted objects (i.e., those updated by the MH but not updated by the server during [0, L]) are then sent to the server by means of differencing. The time for this step is

$$\sum_{i=1}^{N} p_i (1 - r_i) T_{\text{diff}}$$

because on average $\sum_{i=1}^{N} p_i(1 - r_i)$ objects are updated by the MH but not updated by the server (thus are accepted by the server) during [0, L]. The server commits to the updates immediately when it receives the differences.

5. The server sends an acknowledgement via the service channel to the MH after it receives and processes the update propagation for those accepted objects. The time for this step is T_{ack}.
6. The MH then performs forced updates for those objects updated by the server, including objects updated by both the MH and the server, and objects not updated by the MH but updated by the server. Further, the MH also performs forced updates for those objects not updated by either the MH or the server due to the forced update policy[2] applying to non-updated objects at time L. For the former case, the MH will receive differences from the server and a merge operation will be applied to resolve conflicts. For the latter case, the MH will not receive differences from the server and a forced update will be

[2] Many applications necessitate force updates when the MH is reconnected to the server, i.e., the MH must update and propagate its updates to the server at an appropriate reconnection time so as not to miss the real-time deadline requirements. This real-time requirement calls for a special handling protocol for those objects that have not been updated by the MH at the time of reconnection and also mandates that the MH be reconnected to the server at an appropriate time prior to the real-time deadline to account for the time needed to propagate updates.

applied directly to the prefetched version. With the help of a conflict resolution tool/editor, we assume that the time for the MH to inspect the differences (for the former case) and adopt/edit changes is D_m on average for all cases. Thus, the total time for this step is given by

$$\sum_{i=1}^{N} r_i(D_m + T_{\text{diff}}) + \sum_{i=1}^{N}(1 - r_i)(1 - p_i)(D_m + T_{\text{diff}})$$

where the first term accounts for the time taken to generate and propagate updates for the former case (i.e., for those objects updated by the server during [0, L]) based on the versions received from the server; the 2nd term accounts for the time taken to generate and propagate updates for the latter case (i.e., for those objects not updated by either the MH or the server during [0, L]).

7. The server sends an acknowledgement to the MH after it receives and processes the update propagation for those objects being forcibly updated by the MH in the previous step. All locks are also released in this step. The time for this step is T_{ack}.

The total reconnection time to propagate updates of all cached objects is therefore equal to the sum of all the individual times in steps (1) to (7) above, i.e.,

$$\begin{aligned} C_N^{QOR} &= 3T_{\text{ack}} + T_{\text{ack}} + \sum_{i=1}^{N}[r_i T_{\text{diff}} + p_i(1 - r_i)T_{\text{diff}} + r_i(D_m + T_{\text{diff}}) \\ &\quad +(1 - r_i)(1 - p_i)(D_m + T_{\text{diff}})]. \end{aligned} \tag{8}$$

3.2. Invalidation Only (INV) Algorithm

Here we also consider the case in which the MH prefetches a set of cached objects before disconnection. However, in this case the server periodically broadcasts an invalidation report to the MH. The MH needs to tune in periodically to read the invalidation report on a *cycle by cycle* basis until all updated cached objects in the MH are propagated. The INV algorithm requires an update propagation procedure (described below) to be executed at the beginning of each broadcast cycle. The MH disconnects right after the procedure is executed and then reconnects to the server at the beginning of the next broadcast interval if necessary. Initially (in the first iteration), the number of cached items to be updated and propagated by the MH is the number of prefetched items N. In the second iteration, the same procedure is repeated, except that the MH only propagates the remaining items that have not been propagated in the first iteration, and so on. Let N_j be the remaining number of objects yet to be propagated by the MH to the server in iteration j and let m be the total number of iterations after which all N items eventually will be updated and propagated.

The number of items in the database being updated by the server within a broadcast cycle L_b, regardless of the number of iterations that have been performed, is given by

$$N_L = \sum_{i=1}^{N_{db}} r_i(L_b) \tag{9}$$

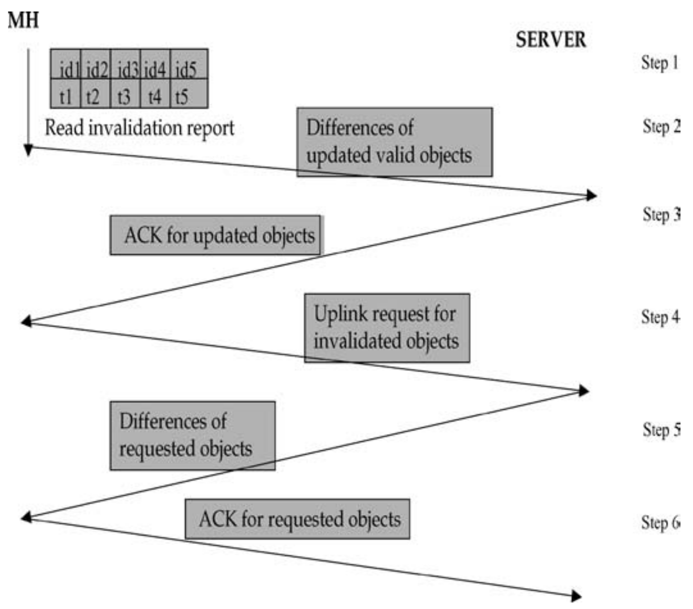
MH
SERVER
id1 id2 id3 id4 id5
t1 t2 t3 t4 t5
Read invalidation report
Differences of updated valid objects
ACK for updated objects
Uplink request for invalidated objects
Differences of requested objects
ACK for requested objects
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6

Figure 2. Steps taken by the INV algorithm during a broadcast interval.

The number of cached data items among N_j objects stored in the MH that have been updated by the server during the last broadcast interval in iteration j is given by

$$N_{MH,j} = \sum_{i=1}^{N_j} r_i(L_b) \tag{10}$$

Therefore, the number of objects updated by the MH, but not by the server in iteration j is given by

$$K_j = \max\left(0, \sum_{i=1}^{N_j} p_i(L_b) - \sum_{i=1}^{N_j} r_i(L_b)\right) \tag{11}$$

Figure 2 illustrates the steps taken to execute the update propagation procedure during a broadcast interval under the INV algorithm:

1. At the beginning of the broadcast interval, the MH tunes in to read the invalidation report. The time for this step is $T_{\text{rep}}N_L$ where T_{rep} is the time to read an entry broadcast in the invalidation report as given in Equation 3, and N_L is the number of items updated by the server in the last broadcast period as given in Equation 9. The MH must read all items in the validation report to know if any of its cached data items is invalidated, although only a subset of items are cached in the MH.
2. The MH compares the ids and timestamps in the invalidation report with the ids and timestamps of its cached data items to determine what items have been invalidated. The MH then propagates only those items that have been updated and are still valid back to the server. The time for this step is $T_{\text{diff}}K_j$ where T_{diff} is the time to send the differences of an object from the MH to the server via the service channel and K_j is the number of data items updated by the MH in the last broadcast interval in iteration j, as given in Equation 11. Note that in cases multiple MHs update the same item concurrently during a broadcast period, the server will only accept one update request and reject the others.

3. Upon receiving the propagation, the server sends an acknowledgement to the MH for those objects accepted for update. The time for this step is T_{ack} where T_{ack} is the time for the server to send an acknowledge to the MH via the service channel.
4. For the items that have been invalidated or rejected, the MH sends a request to the server. The time for this step is T_{ack} where T_{ack} is the time for the MH to send a query to the server via the service channel.
5. Upon receiving the request from the MH, the server sends back a reply in the form of differences relative to the cached items stored in the MH to the MH. The time for this step is $T_{\text{diff}}N_{MH,j}$ where T_{diff} is the time to transmit differences of a data item from the server to the MH via the service channel and $N_{MH,j}$ is the number of cached data items among N_j items stored in the MH that have been updated by the server during the last broadcast period in iteration j, as given in Equation 10.
6. The MH sends an acknowledgement to the server. The time for this step is T_{ack}.

The total tuning time under the INV algorithm to propagate updates of all cached objects is equal to the sum of communication times required for all steps in all m iterations, viz.,

$$C_N^{INV} = \sum_{j=1}^{m}(T_{\text{rep}}N_L + T_{\text{diff}}K_j + T_{\text{diff}}N_{MH,j} + 3T_{\text{ack}}) \tag{12}$$

The value of m can be estimated by defining a Bernoulli random variable X_i for item i such that X_i is 1 if item i is updated by the MH but not by the server in L_b and 0 otherwise. Then X_i is a Bernoulli random variable with probability $p_i(1-r_i)$. Therefore, the number of broadcast periods required until item i is updated can be modeled by a Geometric random variable with parameter $p_i(1-r_i)$ having independent Bernoulli trials. The expected number of periods required for item i to be updated and propagated by the MH is therefore $1/[p_i(1-r_i)]$. Since m is the number of periods after which all N items are updated and propagated by the MH, the value of m is given by the maximum of $1/[p_i(1-r_i)]$ for for i, $1 \le i \le N$, i.e.,

$$m = \text{MAX}_{i=1}^{N}\frac{1}{p_i(1-r_i)} \tag{13}$$

3.3. Invalidation Plus Broadcast (I + B) Algorithm

Here we also deal with the case in which the MH prefetches a set of cached objects before disconnection. The differences between I + B and INV is that the server periodically broadcasts not only the invalidation report, but also the data items that were updated in the last broadcast interval. The MH needs to tune in periodically to read the invalidation report. Then it uses the pointers to tune in to read the data items.

Figure 3 illustrates the steps taken to execute the update propagation procedure during a broadcast interval under the I + B algorithm:

1. At the beginning of the broadcast interval, the MH tunes in to read the invalidation report. The time for this step is $T_{\text{rep}}N_L$, where T_{rep} is the time to read the invalidation report as given in Equation 4 and N_L is the number of items updated by the server in the last broadcast period as given in Equation 9.
2. Based on the ids and the timestamps, the MH determines the items that have been invalidated. The MH then tunes in to read the data items from the broadcast. The time for this step is $T_{\text{read}}N_{MH,j}$ where T_{read} is the time to read the new value of a data item broadcasted as

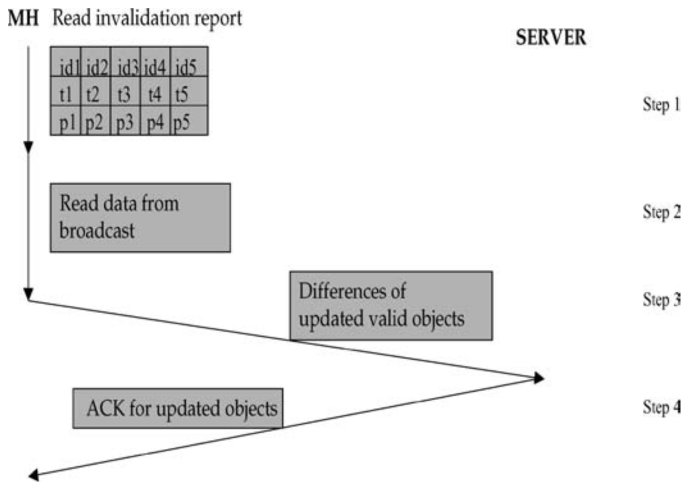

Figure 3. Steps taken by the I + B algorithm during a broadcast interval.

given in Equation 5, and $N_{MH,j}$ is the number of cached data items among N_j items stored in the MH that have been updated by the server during the last broadcast period, as given in Equation 10. This step is different from that in the INV algorithm in that the new data items are directly retrieved from the data stream broadcast by the server. The difference technique is not used here, however, because MHs have different copies of the same data item and there are no common differences that can be applied to all MHs. Thus, in the I + B algorithm, the MH must read an item's new value in its entirety from the broadcast data.

3. The MH then propagates the items that it has updated and that are still valid to the server. The time for this step is $T_{\text{diff}}K_j$, as in step 2 of the INV algorithm.
4. The server sends an acknowledgement to the MH. The time for this step is T_{ack}, as in step 3 of the INV algorithm.

As in the INV algorithm, the above process is repeated until all cached items meant to be updated by the MH are propagated. The MH disconnects after the above processing steps have been executed and reconnects to the server at the beginning of the next broadcast interval. Again, let N_j be the remaining number of items to be propagated by the MH to the server in iteration j and let m be the total number of iterations after which all N items eventually are updated and propagated as estimated by Equation 13. Then, the total tuning time under the I+B algorithm to propagate updates of all cached objects is given by:

$$C_N^{I+B} = \sum_{j=1}^{m} (T_{\text{rep}}N_L + T_{\text{read}}N_{MH,j} + T_{\text{diff}}K_j + T_{\text{ack}}) \tag{14}$$

4. Performance Analysis

In this section, we present numerical data obtained from applying Equations 8, 12 and 14 to compare performances of the QOR, INV and I + B algorithms. We compute the tuning time under QOR, C_N^{QOR}, as a function of the disconnection time period L through Equation 8. The best disconnection time period L_{opt} under QOR for the MH to reconnect to the server for update propagation is determined by computing C_N^{QOR} over a range of L values and identifying the one that yields the smallest tuning time. The tuning time under INV is computed by Equation 12 as a function of the broadcast interval L_b, involving an iterative computational procedure with the number of iterations m required (as determined by Equation 13) depending on the number of objects updated by the MH but not updated by the server in iteration j as determined by Equation 11. Lastly, the tuning times under I + B is computed by Equation 14 as a function of the broadcast interval L_b, by applying a similar iterative computational procedure.

4.1. Parameterization and Baseline Setting

We classify model parameters into two sets. The first set contains those parameters that more or less assume a constant value regardless of the application, namely, s_i, s_b, and s_r. The second set contains those parameters that are changeable reflecting the characteristics of the environment or application in question, namely, N_{db}, N, s_o, p_m, D_m, B, B_d and λ/λ^w.

We first consider a baseline system reflecting a possible setting of the model parameters in the second set for a real application. Then, recognizing that those parameters in the second set will vary depending on the characteristics of a real application, we vary the values of some of the parameters in the second set, such as the world update rate vs. mobile update rate, the number of objects in the database, the bandwidth, etc. in Section 4.4 to analyze the sensitivity of the results with respect to these key parameters.

Our baseline setting considers a server with a small database size of $N_{db} = 100$ items for which the mobile user prefetches $N = 10$ items before disconnection. The database size is an important parameter for INV and I + B. If the database size is too large, then the tuning time overhead for them to read the invalidation report (plus data in I + B) would be too high particularly if the update rate of data items is high. Later we will analyze the effect of a larger database size. Each data item is assumed to be 1024 bytes in length ($s_o = 1024$), corresponding to a disk sector size for fast disk read/write. For the INV algorithm, the size of each `<id,timestamp>` entry is 8 bytes ($s_i = 8$) in the invalidation report, while for the I + B algorithm, the size of each `<id, timestamp, pointer>` entry is 12 bytes ($s_b = 12$) in the invalidation report since a pointer field is required to point to its updated value in the broadcast data. The size of a reply/ack is 8 bytes ($s_r = 8$). We also consider on average a fraction of 0.3 of each data item would be updated by the mobile user ($p_m = 0.3$). The broadcast channel is considered to be substantially larger in capacity than the service channel. We consider the case in which $B = 9.6$ kbps and $B_d = 56$ kbps. For the QOR algorithm, the mobile user will stay online until all items are updated and propagated after reconnection. We assume that the merging/update time $D_m = 10$ s. Finally, we consider the case in which the ratio of the world update rate to the mobile user update rate for each data item is $\lambda/\lambda^w = 0.55$, with the absolute value of λ^w varying in the range of 1–10 updates/h. This setting reflects an application for which the MH's update rate is 55% of the overall world update rate.

4.2. Tuning Time Components

We first break up the total tuning time into its constituent components to gain some insight into which component dominates the tuning time. For the QOR algorithm, the total tuning time is equal to the update propagation time after reconnection because the MH stays online once it is reconnected to the server until all its updated data items are propagated.

For the INV algorithm, the tuning time consists of three components: (a) reading invalidation reports; (b) update propagation (for transmitting differences of updated yet valid items from the MH to the server); (c) on-demand data retrieval (including the time for the MH to send requests via the service channel to the server for retrieving new values of updated data items in the last broadcast interval and the time for the server to send differences to the MH for the new values requested). For the I + B algorithm the tuning time consists of three components: (a) reading invalidation reports; (b) update propagation; (c) reading broadcast data.

Figures 4 and 5 show the tuning time components for the INV and I + B algorithms, respectively. Here, we vary the value of the broadcast interval L_b from 50 to 3600 s to see its effect on the tuning time components. The X-coordinate is the broadcast interval in seconds and the Y-coordinate is the tuning time. Figures 4 and 5 show that the "update propagation" component is insensitive to the broadcast interval L_b because the total time required to propagate N data items from the MH to the server via the service channel by means of differencing remains constant regardless of the magnitude of the broadcast interval and the number of iterations required to eventually propagate all N items. The "reading invalidation reports" component decreases as the broadcast interval increases, because the process of propagating updates takes several iterations to complete so the amortized cost for reading invalidation reports decreases when the broadcast interval increases. For the I + B algorithm (in Figure 5), the time to read broadcast data increases as the broadcast interval increases because more data items will be updated by the server when the interval is large and the probability that cached objects stored at the MH will be invalidated by the server increases as the interval

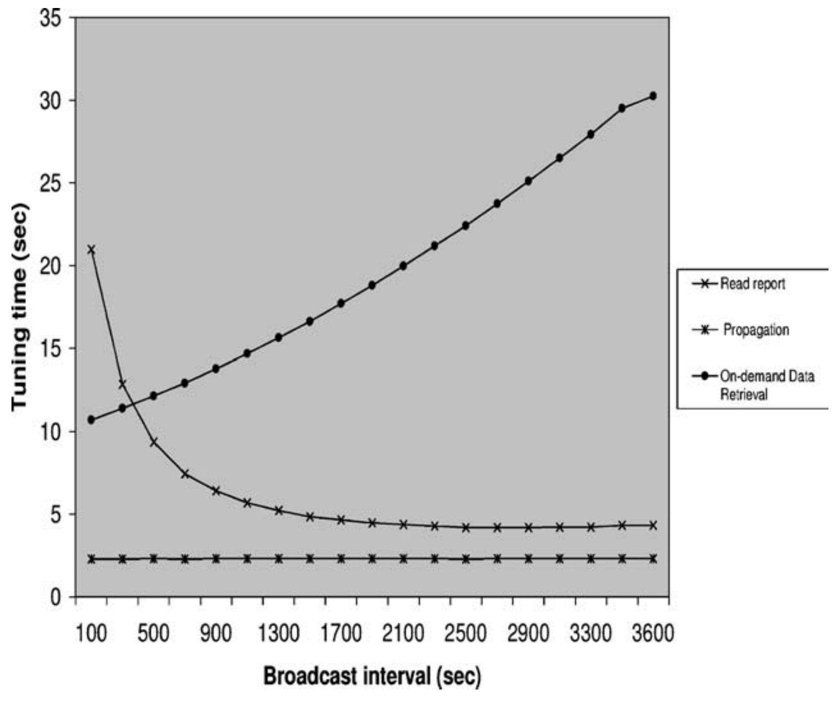

Figure 4. Three components of the tuning time for the INV algorithm.

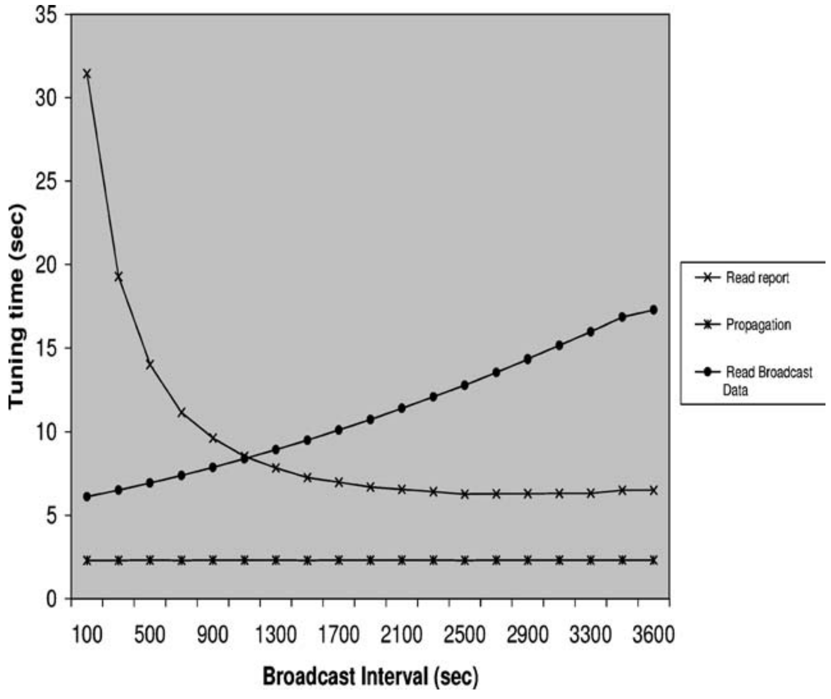

Figure 5. Three components of the tuning time for the I + B algorithm.

increases, in which case the MH must spend more time to read new values from the broadcast data.

4.3. Tuning Time Comparison in the Baseline System

In Figure 6 we compare QOR, INV and I + B operating under the baseline setting of parameter values given earlier. Again the X-coordinate is the broadcast interval in seconds and the Y-coordinate is the total tuning time. Since the QOR algorithm is insensitive to the variation of the broadcasting interval, we only show its tuning time (i.e., update propagation time) at the optimizing reconnection time point such that the reconnection time interval is minimized.

The optimizing reconnection time point under QOR can be obtained by varying the value of L based on Equation 8 and identifying the best L value that minimizes the tuning time. Figure 7 shows an example of this optimal reconnection time for minimizing the tuning time under QOR as a function of λ/λ^w. We observe that as the MH's update rate λ varies from $0.2\lambda^w$ (the top curve) to λ^w (the bottom curve), the curves get deeper. This means that when the MH's update rate is closer to the world update rate, the MH can save more tuning time by propagating updates at the optimal disconnection time point L. At the extreme case in which λ equals λ^w such that cached data objects are updated only by the MH, the optimal disconnection period equals infinity. In this special case, the MH can connect to the system at a time point (say 3600 as shown in Figure 7) at which the tuning time is very near to the optimal value so that the elapsed time is not excessively long. We also observe that the optimal disconnection point shifts toward right as we go from the top curve to the bottom curve. This indicates that the optimal reconnection time point for minimizing the tuning time is shorter when the MH's update rate is far below

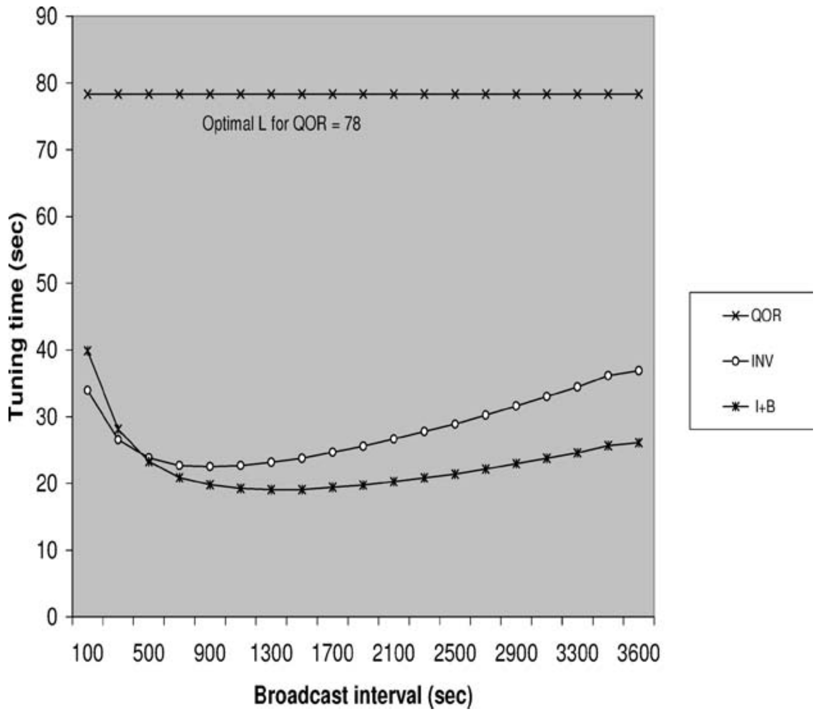

Figure 6. Comparing the tuning time of QOR, INV and I + B algorithms.

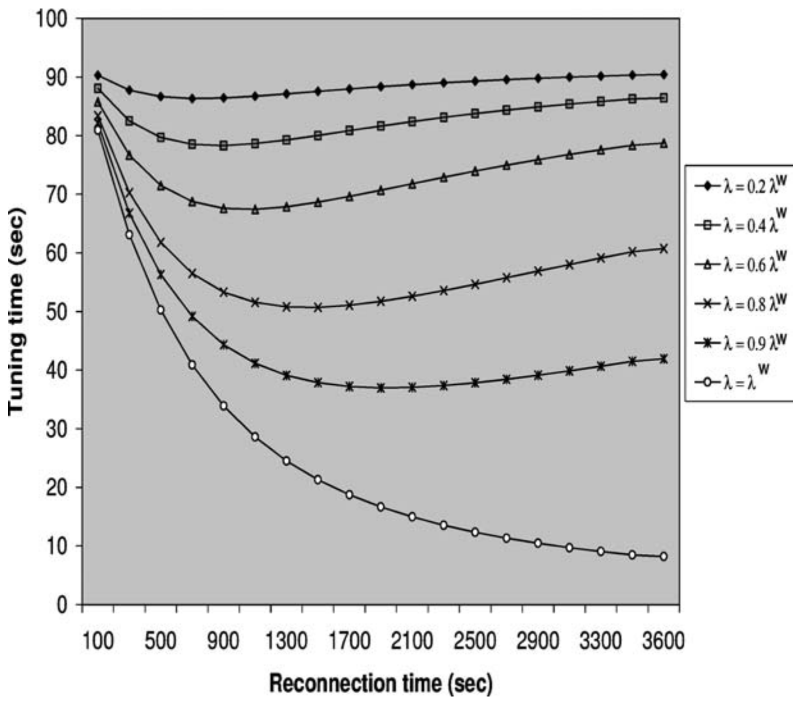

Figure 7. Optimal reconnection time point under QOR at various λ/λ^w ratios.

the world update rate, and it is longer when the MH's update rate is close to the world update rate.

For the INV and I + B algorithms, we observe from Figure 6 that there exists a broadcast interval L_b at which the tuning time is minimized. This is due to the fact that (a) when L_b is small, the tuning time for reading invalidation reports (for both the INV and I + B algorithms) is excessive because of the high broadcasting frequency; (b) when L_b is large, most cached data items stored in the MH have been updated in the last broadcast interval. For the INV algorithm, large L_b results in an increase of the "on-demand data retrieval" tuning time component. For the I + B algorithm, it results in an increase of the "reading broadcast data" tuning time component. Consequently, there exists an intermediate L_b value under which the tuning time is minimized.

Under the set of parameter values given, we observe that both the INV and I + B algorithms perform better than the QOR algorithm. We also observe there is a crossover point in the broadcast interval over which I + B performs better than INV. The crossover point occurs because when L_b is small, there is a high probability that the cached data items updated by the MH have not been updated by the server. In this case the INV algorithm will perform better than the I + B algorithm as there won't be too many on-demand data retrieval requests sent to the server using the relatively low-bandwidth service channel, i.e., the "on-demand data retrieval" tuning time for the INV algorithm is less than the "reading broadcast data" tuning time for the I + B algorithm. When L_b is sufficiently large, however, many cached items stored in the MH with a high probability may be updated by the server, so the MH needs to send more on-demand requests to the server to retrieve updated data. In this case the I + B algorithm is favored over the INV algorithm since in I + B, updated data are broadcast directly by the server from which the MH can read directly by using the pointers embedded in the invalidation report. This results in the existence of a crossover point in the broadcast interval after which the I + B algorithm performs better than the INV algorithm.

4.4. Sensitivity Analysis

In the following subsections, we analyze the sensitivity of results with respect to key system parameters. Under the baseline setting, we see that INV and I + B perform better than QOR. However, this observation is not universally true. Below we identify and analyze important model parameters that impact the performance of update propagation algorithms in wireless mobile system with broadcasting capability.

4.4.1. *Effect of Broadcast Channel Bandwidth on Tuning Time*

In Figure 8, we lower the broadcast channel bandwidth while keeping all other parameters with the same values. We observe that when the broadcast channel bandwidth is low at 19.2 kbps (compared with the service channel bandwidth fixed at 9.6 kbps), the QOR algorithm can perform better than both the INV and I + B algorithms, especially when the broadcast interval L_b is either sufficiently small or large. This is attributed to the fact that the tuning time for reading the invalidation report (in INV and I + B) and broadcast data (in I + B) increases as the broadcast channel bandwidth decreases. When L_b is sufficiently small, the tuning time of both the INV and I + B algorithms is high because of the high frequency of reading the invalidation reports and data, so the QOR algorithm performs the best. As L_b increases, the

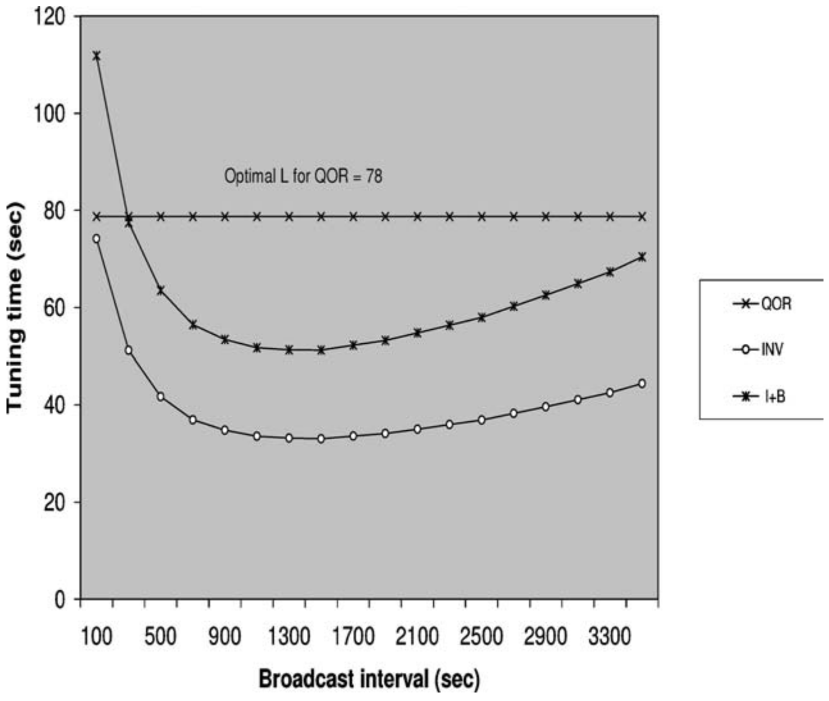

Figure 8. Tuning time at low broadcast bandwidth (19.2 kbps).

tuning time reaches its minimum for either the INV or I + B algorithm, at which point the QOR algorithm is worse than either the INV or I + B algorithm. When L_b is sufficiently large, the tuning time of both the INV and I + B algorithms becomes high again, at which point the QOR algorithm performs the best again. Figure 6 (in which $B_d = 56$ kbps) and Figure 8 (in which $B_d = 19.2$ kbps) together show that the broadcast channel bandwidth can greatly affect the relative performance level of the QOR algorithm compared with the INV and I + B algorithms. Specifically, the broadcast channel bandwidth can determine if QOR should be used instead of INV or I + B to propagate updates by the MH even if the system is broadcasting-enabled.

4.4.2. *Effect of λ/λ^w Ratio on Tuning Time*

Figure 9 compares the performances of the QOR, INV and I + B algorithms when the λ/λ^w ratio per data item is increased from 0.55 (as shown in Figure 6) to 0.7, while keeping all other parameters the same. Here we observe both the INV and I + B algorithms will benefit more from the increase of the λ/λ^w ratio, because there are fewer update conflicts between the MH and server and thus fewer cycles will be required for the MH to propagate all N objects to the server. This largely reduces the tuning time to read the invalidation reports (for both INV and I + B) and broadcast data (for I + B).

Comparing Figure 6 with Figure 9, we also see that a high λ/λ^w ratio especially favors the I + B algorithm over the INV algorithm since in Figure 9, the I + B algorithm is at least as good as the INV algorithm even if the broadcast interval L_b is small. This is because when λ (the MH update rate on a single item) is high compared with λ^w (the server update rate on a single item) and L_b is small, the MH under either INV or I + B can successfully propagate updated items without having update conflicts with the server with a high probability. As a

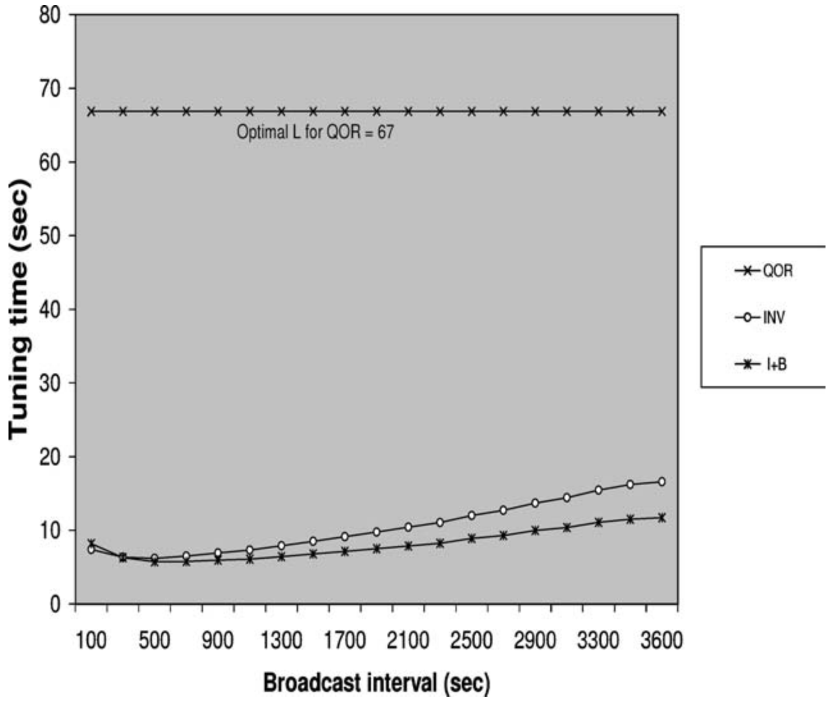

Figure 9. Tuning time at high λ/λ^w ratio.

result, the difference between the "on-demand data retrieval" cost in INV and the "reading broadcast data" cost in I + B is negligible. As L_b becomes larger, many cached items stored in the MH will be updated, so the INV algorithm will incur a higher tuning time to send on-demand data retrieval requests through the low-bandwidth service channel, resulting in its performance being worse than the I + B algorithm.

4.4.3. *Effect of the World Update Rates* λ^w

Figure 10 compares the performances of the QOR, INV and I + B algorithms when the absolute value of the world update rate per data item is increased from 1–10 updates/h (as shown in Figure 6) to 10–100 updates/h, while keeping all other parameters the same. Figure 10 correlates well with other results presented earlier in two ways. First, as the update rate increases, many cached items stored in the MH may be updated by the server, so the INV algorithm is compared unfavorably with the I + B algorithm because of the high tuning time to use the service channel to request for updated items from the server. Second, as the update rate increases, there is a large overhead to read invalidation reports for both INV and I + B and to read broadcast data for I + B, so the QOR algorithm may outperform these two broadcast-based update propagation algorithms. Figure 10 suggests that under the condition that the world update rate of data items is high, if L_b is small then use the I + B algorithm; otherwise, use the QOR algorithm to propagate updates.

4.4.4. *Effect of Database Size* N_{db}

In Figure 11, we change the database size from 100 (as shown in Figure 6) to 1000 data items. We observe that when the database size is sufficiently large, the QOR algorithm performs better than both the INV and I + B algorithms because of a high overhead of

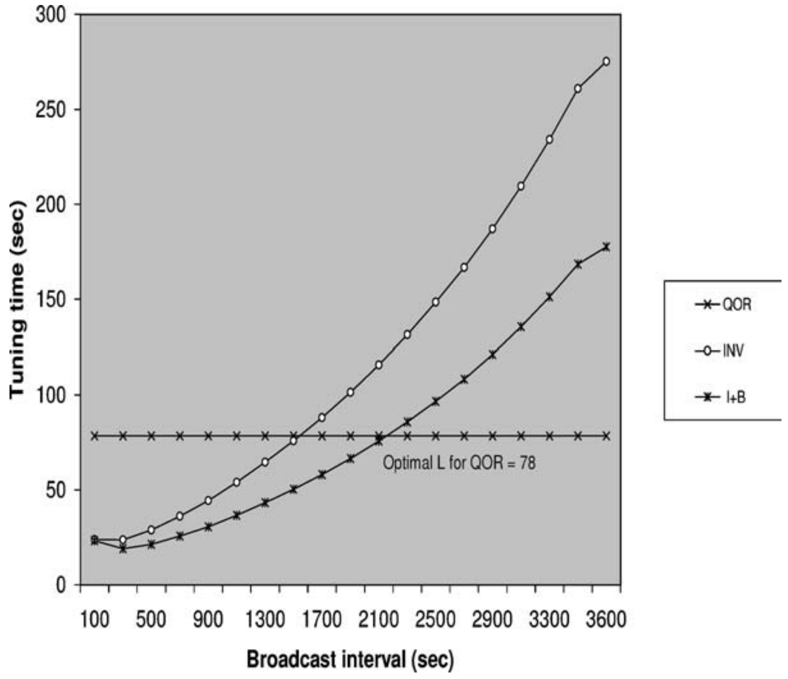

Figure 10. Tuning time at high world update rate.

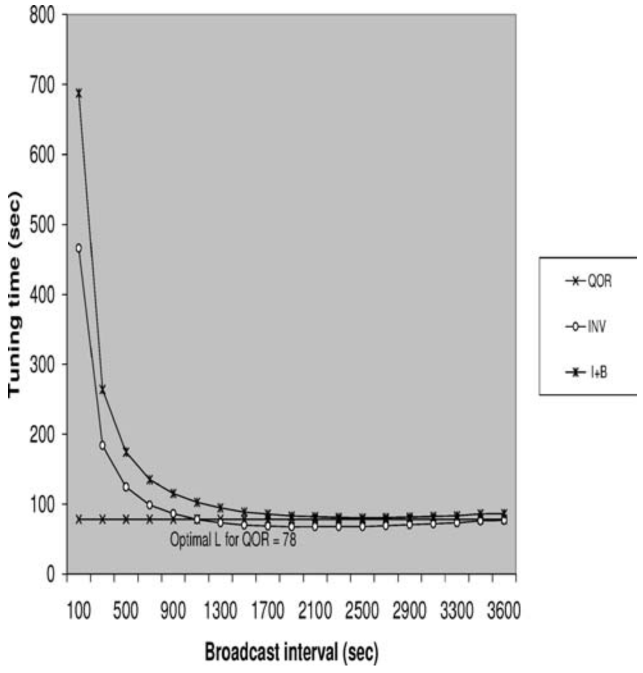

Figure 11. Tuning time at large database size.

reading the invalidation report and broadcast data for the latter two algorithms. We also observe that between INV and I + B, the INV algorithm in this case performs slightly better than the I + B algorithm when L_b is small due to smaller invalidation reports in size.

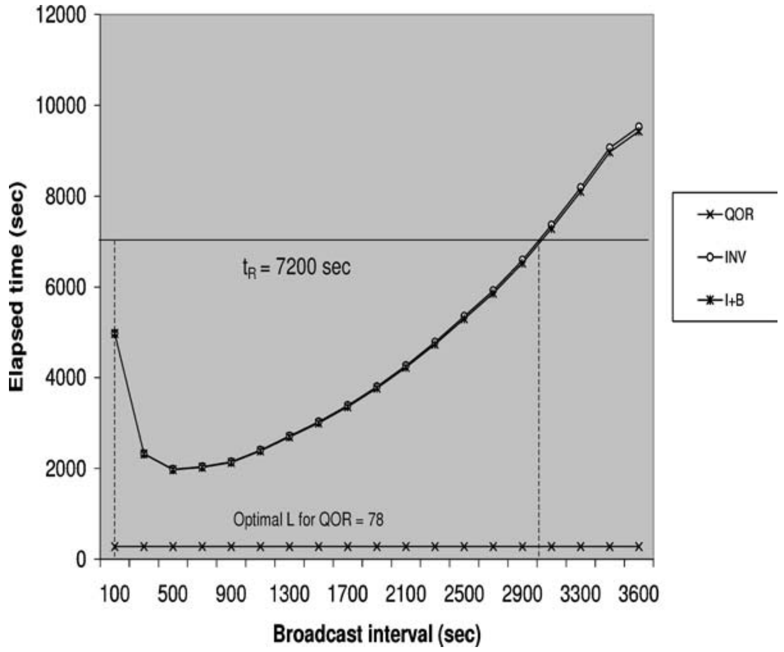

Figure 12. Comparing elapsed time of QOR, INV and I + B at high world update rate.

4.5. Elapsed Time Analysis for Real-Time Applications

For real-time applications, the actual elapsed time required for the MH to propagate updates to the server in the face of possible update conflicts may become a more important metric than the tuning time. Since both the INV and I + B algorithms require multiple cycles to propagate updates, the elapsed time depends on the length of the broadcast interval. Consequently, they may not be suited for real-time applications. For the QOR algorithm, the elapsed time is the sum of the reconnection time interval L and the time to stay online to propagate updates. For INV and I + B, the elapsed time is the total duration (including the waiting time for m broadcast cycles) to propagate all updates.

All the elapsed time vs. L_b graphs can be obtained in the same way as the tuning time vs. L_b graphs reported earlier. Here we illustrate the tradeoff between tuning time and elapsed time by Figure 12 in which we compare the elapsed time required by the QOR, INV and I + B algorithms to propagate updates in the baseline system with the world update rate in the range of 10–100 updates/h, corresponding to the tuning time vs. L_b graph shown in Figure 10.

We see that the QOR algorithm has the smallest elapsed time among all. The elapsed time of the INV algorithm is virtually the same as that of the I + B algorithm due to the fact that the same number of iterations is required by both algorithms to complete the update propagation from the MH to the server. In particular, Figure 12 shows that the elapsed time for the INV and I + B algorithms can become very large when L_b is sufficiently large. The reason is that many cached items stored in the MH would be updated by the server during the last broadcast period especially when the world update date is high, thus resulting in many iterations being required by the INV and I + B algorithms to eventually propagate all updates made by the MH and, consequently, resulting in a very high elapsed time.

For applications with a real time deadline t_R by which updates of cached items must be propagated back to the server, we would like to minimize the tuning time while satisfying the real-time constraint. Figure 12 also illustrates a case in which t_R is 7200 s (or 2 h) for which QOR under the optimizing L, as well as INV or I + B under a L_b period in the range of (100, 3000) s, can satisfy the imposed real-time constraint. We make use of the corresponding tuning time data for the system in the same setting shown in Figure 10 to select one algorithm that would minimize the tuning time while satisfying the real-time constraint as follows. If L_b is in the range of (100, 2100) and the server broadcasts changed data objects in addition to invalidation reports, then the MH would choose I + B over INV and QOR because in this range I + B provides the smallest tuning time among all (see Figure 10) while satisfying the real-time constraint (see Figure 12); if the server only broadcasts invalidation reports, then the MH would choose INV over QOR. If L_b is in the range of (2100, 3000) then the MH would choose QOR over INV and I + B because in this range QOR provides the smallest tuning time among all. Finally, if L_b is not in the range of (100, 3000) the only choice is QOR since it is the only algorithm among all that can satisfy the real-time constraint t_R.

5. Conclusions and Future Work

In this paper, we developed three update propagation algorithms, namely, QOR, INV and I + B, for supporting disconnected operations in mobile wireless systems with data broadcasting capability. We analyzed their performances in terms of the tuning time needed to propagate updates, including the time needed to detect and resolve conflict if it happens.

Our analysis results showed that for the QOR there exists an optimal disconnection time point, and for INV and I + B there exists a broadcast interval under which the tuning time of the MH to propagate all updated items to the server is minimized. We observed that the I + B algorithm can perform better than the QOR and INV algorithms over a wide range of parameter values, although the INV algorithm can perform better than the I + B algorithm when the broadcast interval is small and the broadcast channel bandwidth is small. In particular, when the broadcast channel bandwidth is small, the INV consistently outperforms the I + B algorithm due to the use of differencing techniques. The QOR algorithm is observed to be worse than the INV and I + B algorithms except when the database size is large as there is no overhead for reading large invalidation reports. Nevertheless, there exist conditions under which the QOR algorithm performs better than the INV and I + B algorithms, especially when the update rate is high, or the broadcast channel bandwidth is low. The crossover point depends on the the values of model parameters identified in the paper. Overall, there is no single algorithm that is always the best among all in terms of minimizing the tuning time. The analysis performed in the paper allows the MH to select the best update propagation algorithm to apply to minimize the tuning time when given a set of parameter values by the server characterizing the operating conditions.

Finally, we note that for real-time applications where the elapsed time metric is considered important, the QOR algorithm in general is observed to be the best algorithm since it does not propagate updates in cycles as in the INV and I + B algorithms. Thus, it is able to complete the propagation of updates within the smallest possible elapsed-time period. We also observe that there exists a tradeoff between tuning time and elapsed time in INV and I + B. By making use of the tuning time vs. L_b and the elapsed time vs. L_b analysis results for QOR, INV and I + B, one could select the best algorithm that can minimize the tuning time while satisfying a real-time deadline constraint.

There are some possible future research directions extended from this work, including (a) extending the analysis to the case where the invalidation report may contain information regarding recently updated data items in the last w report intervals; (b) analyzing the total tuning time in situations where a MH may not be able to get a service channel when it reconnects to the server; (c) aggressively determining the best server broadcast interval applying to a large number of MHs for supporting disconnected write operations such that the cumulative tuning times from all MHs to propagate their updates is minimized; and (d) applying similar modeling methods to analyze performances of update propagation algorithms for supporting disconnected *transactions* in mobile wireless system with or without data broadcasting.

References

1. S. Saha, M. Jamtgaard and J. Villasenor, "Bringing the Wireless Internet to Mobile Devices", *IEEE Computer*, Vol. 34, No. 6, pp. 54–58, June 2001.
2. J. Jing, A.S. Helal and A. Elmagarmid, "Client–server Computing in Mobile Environments", *ACM Computing Survey*, Vol. 31, No. 2, pp. 117–157, June 1999.
3. E. Pitoura and G. Samaras, *Data Management for Mobile Computing*, Kluwer Academic Publishers, 1998.
4. H. Chang, et al., "Web Browsing in a Wireless Environment: Disconnected and Asynchronous Operation in ARTour Web Express", *3rd ACM/IEEE Conference Mobile Computing and Networking (MobiCom'97)*, Budapest, Hungary, pp. 260–269, Sept. 1997.
5. R. Floyd, R. Housel and C. Tait, "Mobile Web Access Using eNetwork Web Express", *IEEE Personal Communications*, Vol. 5, No. 5, pp. 47–52, Oct. 1998.
6. Z. Jiang and L. Kleinrock, "Web Prefetching in a Mobile Environment", *IEEE Personal Communications*, Vol. 5, No. 5, pp. 25–34, Oct. 1998.
7. J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System", *ACM Trans. Computer Systems*, Vol. 10, No. 1, pp. 3–25, Feb. 1992.
8. M.S. Mazer and C.L. Brooks "Writing the Web While Disconnected", *IEEE Personal Communications*, Vol. 5, No. 5, pp. 35–41, Oct. 1998.
9. A.D. Joseph, J.A. Tauber and M.F. Kaashoek, "Mobile Computing with the Rover Tool-Kit", *IEEE Transactions on Computers*, Vol. 46, No. 3, pp. 337–352, 1997.
10. D.B. Terry, et al., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System", *ACM SIGOPS Operating Systems Review*, Vol. 29, No. 5, pp. 172–182, Dec. 1995.
11. M. Satyanarayanan, "The Evolution of Coda", *ACM Transactions on Computer Systems*, Vol. 20, No. 2, pp. 85–124, May 2002.
12. I.R. Chen, N.A. Phan and I.L. Yen, "Algorithms for Supporting Disconnected Write Operations for Wireless Web Access in Mobile Client–Server Environments", *IEEE Transactions on Mobile Computing*, Vol. 1, No. 1, pp. 46–58, 2002.
13. A. Datta, A. Celik and V. Kumar, "Broadcast Protocols to Support Efficient Retrieval from Databases by Mobile Users", *ACM Transactions on Database Systems*, Vol. 24, No. 1, pp. 1–79, March 1999.
14. T. Imielinski, S. Vishwanathan and B.R. Badrinath, "Data on Air: Organization and Access", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 3, pp. 353–372, 1997.
15. K.L. Tan, J. Cai and B.C. OOi, "An Evaluation of Cache Invalidation Strategies in Wireless Environments", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, No. 8, pp. 789–807, Aug. 2001.
16. A. Kahol, S. Khurana, S.K.S. Gupta and P.K. Srimani, "A Strategy to Manage Cache Consistency in a Disconnected Distributed Environment", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, No. 7, pp. 686–700, July 2001.
17. S. Acharya, R. Alonso, M. Franklin and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments", *1995 ACM SIGMOD International Conference on Management of Data*, New York, pp. 199–210, 1995.
18. T. Imielinski, S. Vishwanathan and B.R. Badrinath, "Energy Efficient Indexing on Air", *1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, pp. 25–36, May 1994.
19. E. Liebmann and S. Dustdar, "Adaptive Data Dissemination and Caching for Edge Service Architectures Built with the J2EE", *2004 ACM symposium on Applied Computing*, Nicosia, Cyprus, pp. 1717–1724, 2004.

20. L.A. DaSilva, S.F. Midkiff and I.R. Chen, “A Hands-On Course on Wireless and Mobile Systems Design”, *2nd IEEE Annual Conference on Pervasive Computing and Communications – Workshops (PerEd’04)*, Orlando, FL, pp. 241–246, March 2004.
21. V.C.S. Lee, K.W. Lam, S.H. Son and E.Y.M. Chan, “On Transaction Processing with Partial Validation and Timestamp Ordering in Mobile Broadcast Environments”, *IEEE Transactions on Computers*, Vol. 51, No. 10, pp. 1196–1211, 2002.
22. E. Pitoura and P.K. Chrysanthis, “Multiversion Data Broadcast”, *IEEE Trans. on Computers*, Vol. 51, No. 10, pp. 1224–1230, 2002.
23. J. Hoe, J. Yang, S. Papavassiliou, “Integration of Pricing with Call Admission Control to Meet QoS Requirement in Cellular Networks”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 9, pp. 898–910, Sept. 2002.
24. G. Cao, “On Improving the Performance of Cache invalidation in Mobile Environments”, *ACM/Kluwer Mobile Networks and Applications*, Vol. 7, pp. 291–303, 2002.

Ing-Ray Chen received the BS degree from the National Taiwan University, Taipei, Taiwan, and the MS and PhD degrees in computer science from the University of Houston. He is currently an associate professor in the Department of Computer Science at Virginia Tech. His research interests include mobile computing, pervasive computing, multimedia, distributed systems, real-time intelligent systems, and reliability and performance analysis. Dr. Chen has served on the program committee of numerous conferences, including as program chair for 29th IEEE Annual International Computer Software and Application Conference in 2005, 14th IEEE International Conference on Tools with Artificial Intelligence in 2002, and 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology in 2000. Dr. Chen currently serves as an Associate Editor for *IEEE Transactions on Knowledge and Data Engineering*, *The Computer Journal*, and *International Journal on Artificial Intelligence Tools*. He is a member of the IEEE/CS and ACM.

Ngoc Anh Phan received her Bachelor of Science degree from Moscow Technical University of Communication and Computer Science in 1997, and a Master of Science degree in Computer

Science from Virginia Polytechnic Institute and State University (Virginia Tech) in 1999. She is currently a Ph.D student at Virginia Tech and a Senior Software Engineer at America Online Inc. Her research interests include wireless communications, data management, sensor networks, fault tolerance, and mobile computing.

I-Ling Yen received her BS degree from Tsing-Hua University, Taiwan, and her MS and PhD degrees in Computer Science from the University of Houston. She is currently an Associate Professor of Computer Science at the University of Texas at Dallas. Dr. Yen's research interests are in distributed systems, fault-tolerant computing, self-stabilization algorithms, and security. She has served as program co-chair for the 1997 IEEE High Assurance Systems Engineering Workshop, the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering Technology, and the 1999 Annual IEEE International Conference on Computer Software and Applications Conference. Dr. Yen is a member of the IEEE/CS.