DCIM: Distributed Cache Invalidation Method for maintaining cache consistency in wireless mobile networks

Kassem Fawaz, and Hassan Artail, Senior *Member, IEEE*

Abstract— This paper proposes a client-based cache consistency scheme that is implemented on top of a previously proposed architecture for caching data items in MANETs that caches submitted queries in special nodes, called query directories (QDs), and uses them to locate the data (responses) that are stored in the nodes that requested them, called caching nodes (CNs). We have previously proposed a server-based consistency scheme, named SSUM, and in this paper, we introduce a totally client-based scheme that is termed DCIM which works by having the QD nodes monitor the TTL values of the cached data items and estimate update patterns on the server to adapt the TTL values accordingly. The QDs also monitor the request rates for data items and decide accordingly which items to prefetch. DCIM is analyzed to assess the delay and bandwidth gains (or costs) when compared to polling every time, and was also implemented and simulated it using ns2 along with two other client based schemes to assess its performance experimentally. The consistency ratio, delay, and overhead traffic are reported versus several variables, and DCIM shows to be superior when comprared to existing systems.

Index Terms— cache consistency, data caching, database caching, invalidation, MANET, TTL.

1 Introduction

MOBILE devices are the building blocks of mobile ad-hoc networks (MANETs). They are typically characterized by limited resources, high mobility, transient availability, and lack of direct access to the data source (server). In MANET environments, data caching is essential because it increases the ability of mobile devices to access desired data, and improves overall system performance [14] [25]. In a typical caching architecture, several mobile devices cache data that other devices frequently access or query. Data items are essentially an abstraction of application data that can be anything ranging from a database record, a webpage, an ftp file, etc.

The major issue that faces client cache management concerns the maintenance of data consistency between the cache client and the data source [2]. All cache consistency algorithms seek to increase the probability of serving from the cache data items that are identical to those on the server. However, achieving strong consistency, where cached items are identical to those on the server, requires costly communications with the server to validate (renew) cached items, considering the resource limited mobile devices and the wireless environments they operate in. Consequently there exist different consistency levels describing the degree to which the cached data is up to date. These levels, other than strong consistency, are weak consistency, delta consistency [4][5], probabilistic consistency [7][10], and probabilistic delta consistency [12].

With weak consistency, client queries might get served with inconsistent (stale) data items, while in delta consistency, cached data items are stale for up to a period of time denoted as *delta*. In probabilistic consistency, a data item is consistent with the source with a certain probability denoted as p. Finally, probabilistic delta consistency is a mix of the previous two approaches, where a certain cached item is at most *delta* units of time stale with a probability not less than p.

There are several mechanisms in the literature that approach the cache consistency issue in MANETs by attempting to optimize client server communication while trying to keep the data as fresh as possible. These mechanisms can be grouped into three main categories: push based, pull based, and hybrid approaches [5]. *Push-based* mechanisms are mostly server-based, where the server informs the caches about updates (pushes the updates) that have occurred to its source data. *Pull-based* approaches are client-based, where the client asks the server to update or validate its cached data. Finally, *hybrid* mechanisms combine push and pull methods, where either the server pushes data updates or the clients pull them from the server.

An example of pull-based approaches is the TTL-based algorithms, where a TTL value is stored alongside each data item d in the cache, and d is considered valid until T time units go by since the last cache update. Such algorithms are popular due to their simplicity, sufficiently good performance, and flexibility to assign TTL values to individual data items [13] [27]. Also, they are attractive in mobile environments [28], and are considered suitable in MANETs because of limited device energy and network bandwidth [25] [26], and frequent device disconnections [27]. Moreover, TTL algorithms are completely client based and require minimal server functionality. From this

- *K. Fawaz is with the ECE Department, American University of Beirut, Beirut, Lebanon. E-mail: kmf04@aub.edu.lb.*
- *H. Artail is with the ECE Department, American University of Beirut, Beirut, Lebanon. E-mail: hartail@aub.edu.lb.*

perspective, TTL based algorithms are more practical to deploy and are more scalable.

In this work, we propose a pull-based algorithm that implements adaptive TTL, piggybacking and prefetching, and provides weak to delta consistency guarantees. Cached data items are assigned adaptive TTL values that correspond to their update rates at the data source. Items with expired TTL values are grouped in validation requests to the data source to refresh them. Moreover, in certain situations non-expired items are also included in the validation requests. The data source sends the cache devices the actual items that have changed, or simply invalidates them, depending on their request rates. This approach, which we call Distributed Cache Invalidation Mechanism (DCIM), works on top of the COACS architecture we introduced in [1] and which provides a framework for cooperative data caching in mobile ad hoc networks. To our knowledge, this is the first work that offers a complete client side approach employing adaptive TTL that achieves superior performance in terms of availability, delay, and traffic.

In the rest of this paper, Section 2 discusses related work and reveals the contributions of the proposed system, which we elaborate in Section 3. Section 4 provides an analytical analysis of the system, whereas Section 5 presents the experimental results and discusses their significance. Section 6 finishes the paper with concluding remarks and suggestions for future works.

2 Related Work

Much work has been done in relation to cache consistency in MANETs. The proposed algorithms in these works cover push, pull, and hybrid approaches.

2.1 Push based approaches

The work on push-based mechanisms mainly uses invalidation reports (IR), where the server invalidates the cached items. The original IR approach was proposed in [2], but since then several algorithms have been proposed for MANETs. They include stateless schemes where the server stores no information about the client caches [2] [3] [15] [16] and stateful approaches where the server maintains the full state of its cached data, as in the case of the AS scheme [19]. Many optimizations and hybrid approaches were proposed to reduce traffic and latency, like SSUM [38] (more on it in Section 5), and the SACCS scheme in [17] where the server has partial knowledge about the mobile node caches, and flag bits are used both at the server and the mobile nodes to indicate data updates and availability. All of these mechanisms necessitate server side modifications and overhead processing. More crucially, they require the server to maintain some state information about the MANET, which is costly in terms of bandwidth consumption especially in highly dynamic MANET environments. DCIM, on the other hand, belongs to a different class of approaches, as it is a completely pull-based scheme. Hence we will focus our survey of previous work on pull-based schemes, although we will compare the performance of DCIM with that of our recently-proposed push-based approach, namely SSUM, in Section 5.

2.2 Pull based algorithms

Pull based approaches have also been proposed, and as discussed before, they fall in two main categories: client polling and Time to live (TTL).

2.2.1 Client polling

In client polling systems, such as those presented in [18] and [19], a cache validation request is initiated according to a schedule determined by the cache. There are variants of such systems (e.g., [18] and [7]]) that try to achieve strong consistency by validating each data item before being served to a query, in a fashion similar to the "If-modified-since" method of HTTP/1.1. In [18], each cache entry is validated when queried using a modified search algorithm inside the network, whereas in [7] the system is configured with a probability that controls the validation of the data item from the server or the neighbors when requested. Although client poll algorithms have relatively low bandwidth consumption, their access delay is high considering that each item needs to be validated upon each request. DCIM, on the other hand, attempts to provide valid items by adapting expiry intervals to update rates, and uses prefetching to reduce query delays.

2.2.2 TTL-based approaches

TTL-based approaches have been proposed for MANETs in several caching architectures [25],[26],[34],[35], [36], and [14]. The works in [25], [26], and [34] suggest the use of TTL to maintain cache consistency, but do not explain how the TTL calculation and modification are done. A simple consistency scheme was proposed in [35] and [36] based on TTL in a manner similar to the HTTP/1.1 max-age directive that is provided by the server, but no sufficient details are provided. Related to the above, we will show in Section 5 that approaches which rely on fixed TTL are very sensitive to the chosen TTL value and exhibit poor performance. In [14] a client prefetches items from nodes in the network based on a compiled index for request rates for every item, and maintains cache consistency with the data sources based on adaptive TTL calculated similar to the schemes of the Squid cache and the Alex file system (described later). This two-layer scheme introduces large overhead traffic, as two invalidation schemes work in parallel. Furthermore, the TTL calculations are seemingly inaccurate and are based on heuristics [10]. Finally, a poll-every-time/TTL mechanism is proposed in [42] for sensor networks, where three consistency modes are defined. The first is weak, where the item is served directly; the second is delta, where the item is served if it is at most d time units old, or else it is fetched; and the third is strong, where the item is always validated before serving. This approach is similar to the previous approaches in the sense that the expiry time is not well defined. Moreover, the strong mode has a high query delay, as was discussed in subsection 2.2.1.

In summary, it appears that there is no scheme to date that presents a well-founded method for adapting TTL

values at the client side. In reality, the above approaches provide shallow integration of TTL processing into the cache functionality, and none of them gives a complete TTL-based cache consistency scheme for MANETs. Additionally, they do not include mechanisms for reducing bandwidth consumption, which is crucial in MANET environments.

2.3 Hybrid Approaches

Hybrid approaches combining push and pull mechanisms were discussed in [12], [21], [22], [23], and [24]. The work in [12] provides pull functionality using TTL processing and push functionality by invalidating the TTL value after each update at the server. In [21], stale-tolerant items are served directly and their consistency is maintained using TTL. Other items are invalidated by the server if they have expired on the client side. Needless to say, the server has to store the TTL values of the cached items, which is not very practical. Push is implemented in [22] between servers and relay peers, while pull is employed between the caches and relay peers through TTL. The scheme in [23] generates and stores invalidation reports at the gateway node to the internet. In each generation interval, the client nodes pull for the new report, and are thus expected to hold the most-recent ones. In [24] a hybrid push pull algorithm based on prediction is provided, where the server pushes data when it predicts it will be requested soon, whereas the client prefetches data when it is most likely being updated. These algorithms, similar to the push algorithms, require the server to maintain state information and incur processing overhead.

2.4. TTL in web caches

The several TTL algorithms proposed in MANETs are motivated by web caches research. These include the fixed TTL approach in [8] [27] and the adaptive TTL methods in [6], [20], [11], and [29]. Adaptive TTL provides higher consistency requirements along with lower traffic [6], and is calculated using different mechanisms [6],[20],[11],[31], and [32].

The first mechanism in [20] calculates TTL as a factor multiplied by the time difference between the query time of the item and its last update time. This factor determines how much the algorithm is optimistic or conservative. In the second mechanism, TTL is adapted as a factor multiplied by the last update interval. In dynamic systems, such approaches are inappropriate as they require user intervention to set the factors, and lack a sound analytical foundation [10]. In the third mechanism in [40] TTL is calculated as the difference between the query time and the k^{th} recent distinct update time at the server divided by a factor K, and the server relays to the cache the k most recent update times. Other mechanisms were proposed that take into consideration a complete update history at the server to predict future updates and assign TTL values accordingly [28]. The above approaches assume that the server stores the update history for each item, which does not make it an attractive solution. On the other hand, the approach in [33] computes TTL in a TCP-oriented fashion (additive increase multiplicative decrease) [30] to adapt to server updates. However, it is rather complex to tune, as it depends on six parameters, and moreover, our preliminary simulation results revealed that this algorithm gives poor predictions. Finally, the scheme in [10] computes TTL from an update risk that provides a probability for the staleness of cached documents. At the end, it is worth mentioning that piggybacking was proposed in the context of cache consistency to save traffic. In [9] the cache piggybacks a list of invalidated documents when communicating with the server, while in [39] the server piggybacks a list of updated documents when it communicates with the cache.

3 DCIM Architecture and Operations

This section describes the design of DCIM and the interactions between its different components.

3.1 System Model

The system consists of a MANET of wireless mobile nodes that are interested in certain data generated at a data source. The data source (server) is connected to the MANET via a gateway through a wired network. The data exchanged is abstracted by data items, as was mentioned in Section 1. The proposed DCIM system builds on top of COACS (Cooperative and Adaptive Caching System), which we introduced in [1] and did not include provisions for consistency. For completeness, a description of the COACS operations is provided in Appendix A. Briefly, the system has three types of nodes: query directories (QDs) that index the cached items, caching nodes (CNs) that hold the actual items, and requesting nodes (RNs). Although our recently-introduced SSUM [38] cache consistency scheme also builds on the COCAS architecture, it is a server-based approach, whereas DCIM is completely client-based, introduced to realize the benefits of this class of systems. In this regard, DCIM complements SSUM, which is why we contrast their performance in Section 5 to see how they compare.

3.2 Design Methodology

The goal of DCIM is to improve the efficiency of the cache updating process in a network of mobile devices which cache data retrieved from a central server without requiring the latter to maintain state information about the caches. It also aims to provide high consistency guarantees while maintaining high data availability and keeping bandwidth consumption under check.

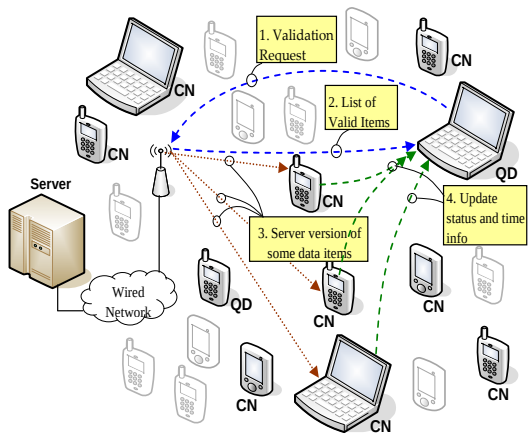

Fig. 1. Overview of DCIM basic design

The proposed system is pull-based, where the QDs monitor the TTL information and accordingly trigger the cache update and validate data items when necessary. DCIM is scalable by virtue of the QDs whose number can increase as the size of the network grows, and thus is more suitable to dynamic MANETs than a push-based alternative since the server does not need to be aware of CN disconnections. DCIM is also more suitable when data requests are database queries associated with multiple tables and attributes. In a push-based approach, the server would have to map a cached query to all of its data sources (table attributes) and execute this query proactively whenever any of the sources is updated. Moreover, DCIM adapts the TTL values to provide higher consistency levels by having each QD estimate the inter-update interval and try to predict the time for the next update and sets it as the item's expiry time. It also estimates the inter-request interval for each data item to predict its next request time, and then prefetches items that it predicts to be requested soon.

3.3 DCIM basic design

In DCIM, the caching system relies on opportunistic validation requests to infer the update patterns for the data items at the server, and uses this information to adapt the TTL values. These validation requests are essentially requests to the server to refresh a set of data items. The QD polls the server frequently to know about the update times of the items it indexes. It also piggybacks requests to refresh the items it indexes each time it has reason to contact the server, basically whenever an item it indexes expires. Nevertheless, to avoid unnecessary piggybacks to the server, the QD utilizes a two-phase approach. Specifically, at the end of each polling interval (T_{poll}), every QD issues validation requests for the items it indexes that have expired TTLs and have a high request rate. After a configurable number of polling intervals, denoted by N_{poll}, the QD issues a validation request for all the items it caches indexes for if at least one item has an expired TTL regardless of its request rate. We refer to the interval $N_{poll} \times T_{poll}$ as the piggyback interval, T_{pigg}. When the server receives a QD's request, it replies with a list of updated as well as non-updated items. The QD uses this information to adapt the TTL values to the server update rate for each item.

TABLE 1
PACKETS USED IN DCIM

Packet	Function	Description
CURP	Cache Update Request	Sent from QD to server to validate certain data items
SVRP	Server Validation Reply	Sent from server to QD to indicate which items are valid
SUDP	Server Update Data	Sent from server to CN. It includes updated data items and timestamps
URP	Update Received	Sent from CN to QD to inform QD that it holds an updated version of a data item.

Although in principle it achieves weak consistency, DCIM can attain delta consistency when at least one item has a TTL expired by the end of the piggybacking interval, thus causing a validation request to be issued periodically. Hence, the QD ensures that data items are at most one piggybacking interval stale. Figure 1 shows a scenario where a QD is sending a cache validation request to the server that is transmitting updates to concerned CNs and returning to the QD a list of valid items. The messages being sent from the CNs to the QD indicate notifications which inform the QD that the data was actually updated, thus serving as acknowledgments.

3.5 Detailed Design

In the remainder of this section, we describe the operations of DCIM in details, but first, we list the messages which we added in DCIM (see Table 1) to those already introduced in COACs. The reader is referred to [1] for a complete description of the basic COACS messages.

Figure 2 describes the basic interactions of DCIM through a scenario in which an RN is submitting a DRP for a query cached in the QD, but is in the waiting list at the moment because the corresponding item is being validated. Validation requests are issued by QD nodes using CURP messages that contain entries for items to be validated. Each entry consists of the query associated with this item, the timestamp of the item (last modification time), a "prefetch" bit (if set, instructs the server to send the actual item if it was updated), expired bit (indicates whether an item is expired or not), and the CN address that holds the item. Upon receiving a CURP message, the server identifies items that have changed and items that have not, and sends the QD in an SVRP the ids of items that did not change and those that changed but were not prefetched by the QD (does not have the prefetch bit set). It also sends the concerned CNs SUDP messages containing the actual items if they were prefetched by the QD and changed. A CN that receives such a message sends a URP message to the QD acknowledging the receipt of an update. Now the QD forwards to the CN the request that was in the waiting list using a DRP message, after which the CN sends the updated cached response to the RN via

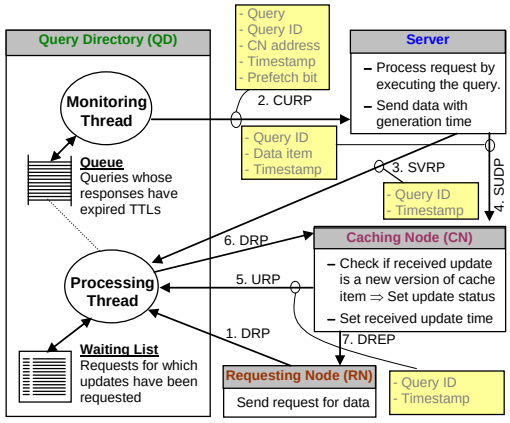

Fig. 2. Interactions between nodes in a DCIM system

a data reply (DREP) message.

3.5.1 TTL Adaptation

The QD in DCIM has a partial picture about the update patterns of each item at the server using the piggybacking mechanism. The QD stores the last update time of each item from the last validation request, and uses this information to predict the next update time. However, the QDs are after all mobile devices which have constraints in terms of power, processing, and storage capabilities, and obviously, sophisticated prediction schemes are slow and inadequate to use in this context. Alternatively, we use running average to estimate the inter-update interval, using timestamps of the items from the server's responses to issued validation requests. The QD can then calculate its own estimation for the inter-update interval at the server, and utilize it to calculate the TTL of the data item. The running average, also known as the exponentially weighted moving average, has the form: $IUI(t)=(1-a)\times IUI(t-1)+ a\times LUI$, where $IUI(t)$ represents the estimated inter-arrival time at time t and LUI represents the last inter-update interval. The QD only needs to store the estimated interval and the last updated time. In fact, this method has many properties that make it suitable for usage in this situation, mainly because of its simplicity and ease of computation, the minimum amount of data required, and the diminishing weights assigned to older data [37]. There are two parameters that control this estimator which are the initial value $IUI(0)$ and the value of a, whose value should be small, i.e. between 0.1 and 0.2, as to minimize the effect of random fluctuations, even if it means larger convergence times [29] (proven in appendix B). In the simulations we describe later, a was set to 0.125 and $IUI(0)$ to 0.

3.5.2 Server operations

As this approach is basically client-based, the processing at the server is minimal. When the server receives the CURP message from the QD, it checks if all items have been changed by comparing their timestamps (Last modified time) with those included in the request. Items that have not changed are considered valid, and their ids are included in the SVRP response to the QD. On the other hand, items that have changed are treated in two ways: Expired items (those having the expiry bit set in the QD validation request) as well as non-expired ones but having the prefetch bit set are updated by sending SUDP packets (which contain timestamps of the associated data items) directly to the CNs, which is possible since their addresses were included in the request. On the other hand, the server informs the QD about items whose expiry and prefetch bits are not set (i.e., will not be requested soon), using an SVRP message. This is summarized in the flow diagram of Figure 3.

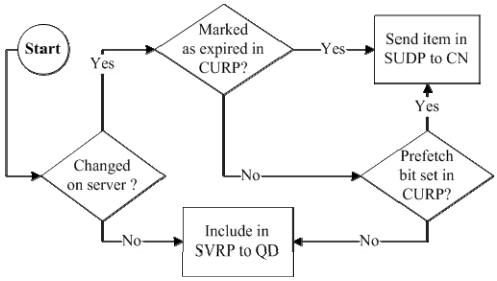

Fig. 3. Decision flow at the server

3.5.3 QD Processing

DCIM exploits the role of the QDs which store the cached queries plus their IDs, and the addresses of the CNs. A QD maintains two tables to manage the consistency of the cache in the CN nodes: the *Cache Information Table* whose data is common to all queries that are locally cached (Table 2), and the *Query Information Table* that stores query-specific data (Table 3). As shown, the QD maintains the weighted average of inter-request interval (IRI) for each data item it references (in a manner similar to the computation of the inter-update interval). The process that runs on the QD includes two threads: a monitoring thread and a processing thread.

Monitoring Thread:

The monitoring thread checks for expired data items, issues validation requests, and requests updates for data items. It performs these operations in two functions:

Inner Loop Function: After each sleep period of T_{poll}, the QD iterates over the entries corresponding to the cached data items it holds indexes for, checking each item's TTL value. If an item has an expired TTL, the QD sets its expiry bit and its state to *INVALID*. It also sets its "prefetch" bit if its average inter-request interval is lower than the piggyback period ($N_{poll}\times T_{poll}$), meaning that in this case the item will be requested with high probability by one or more RN nodes in the next piggybacking interval. The QD then sets the state field to *TEMP_INVALID* to indicate that a validation request for the item is in progress. Normally, nodes that request invalidated data items will have to wait till the server updates the CNs with new versions upon the request of the QD. At the end of the inner loop function, the QD prepares a CURP, and includes in it the validation requests for items that have expired and whose prefetch bits are set.

Outer loop function: When the monitoring thread completes N_{poll} iterations (corresponding to the piggyback interval defined above), it checks if at least one item has expired. If so, it issues a validation request for the whole

TABLE 2
ELEMENTS OF THE GENERAL CACHE INFORMATION TABLE

Parameter	Description
QD_{ID}	Identifier of this QD
WL	Query processing waiting list

TABLE 3
ELEMENTS OF THE QUERY-SPECIFIC CACHE INFORMATION TABLE

Parameter	Description
q_{ID}	Identifier of the locally cached query
CN_{ID}	Identifier of the CN that caches the response of this query
Time-stamp	The last modified time of the data item
TTL	Time to live value associated with this query
IRI	The estimated inter-request interval for this query
IUI	The estimated inter-update interval for this query
State	Indicates if the item is expired or was issued for validation

collection of cached items stored at the QD. In this request, similar to the request issued in the inner loop function, a prefetch bit indicates whether the item is expected to be requested soon or not, as was described above. If it is set, the server sends the actual item, else, it just invalidates the item. Hence, the Outer Loop Function allows the QD to piggyback validation requests for all items when there is a need to contact the server.

Note that the inner loop function issues validation requests *only* for expired items having high request rates, and updates them if necessary. Expired items with low request rates have to wait for at most $N_{poll} \times T_{poll}$ to be validated, while those with high request rates wait for at most T_{poll}. This mechanism reduces access delays by prefetching items with high request rates. In this regard, we note that in delay intolerant networks, the "prefetch" bit can be set for each item regardless of its request rate, assuming it was requested at least once in the past. This way, all items will be prefetched and the hit rate will be forced to be 100% or very close to it (when accounting for items that are requested while being validated), thus reducing response time considerably. Figure 4 summarizes the operations of the inner loop and outer loop functions.

Processing Thread:

This thread handles data requests from RN nodes and replies from the server (i.e., URP and SVRP packets) in response to CURP messages sent by the QD, and computes the TTL value.

Processing Data Request (DRP) Messages: The QD checks the state of the requested item in the DRP, and if it is *INVALID*, it issues an update request directly to the server, converts its state to *TEMP_INVALID*, and places the query on a waiting list. In the meanwhile, if the QD gets a DRP for the same item before the server replies, it also puts it on the waiting list. In all other cases, the query is processed by sending it to the CN that holds the result, in case of a hit, or to the nearest unchecked QD or the server, in case of a miss (regular COACS operations).

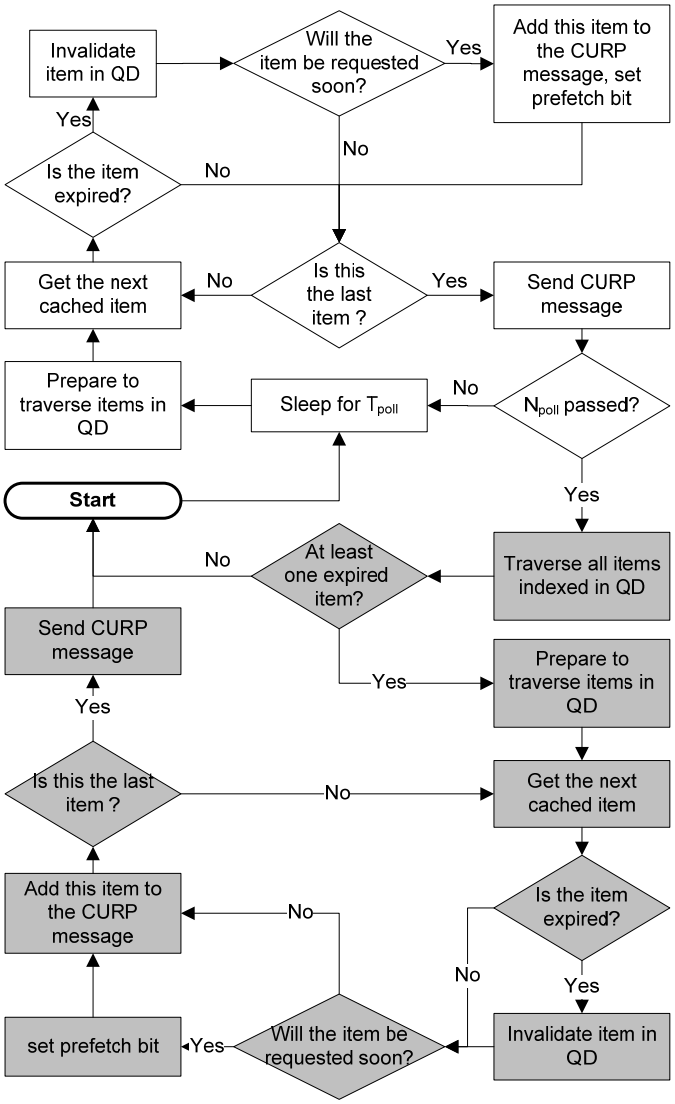

Fig. 4. Flow diagram illustrating the operations of the Inner Loop and Outer Loop (shaded part) functions

Processing SVRP and Update Reply (URP) Messages: If a URP packet was received, it must be for an item that has changed at the server. The QD calculates its TTL as explained below, and if the URP makes reference to items that have requests placed in the waiting list, those items are forwarded to the corresponding requesting nodes. On the other hand, the SVRP is sent from the server in response to a CURP packet, and it is expected to only contain the ids of the items that did not change on the sever, and those of the items that changed but were marked as unexpired and had the prefetch bit not set in the CURP (illustrated in Figure 3). The QD updates the TTL of all elements whose ids are contained in the SVRP. It helps to reiterate here that there are items which were specified in the CURP packet but not sent as part of the SVRP because the actual updated data items were sent directly to the CNs, which in turn are expected to send acknowledgements in URP packets to the QD for better reliability. If the QD does not receive an expected acknowledgement, it assumes that the CN is disconnected and will delete all associated queries, as per the design of COACS [1]. With

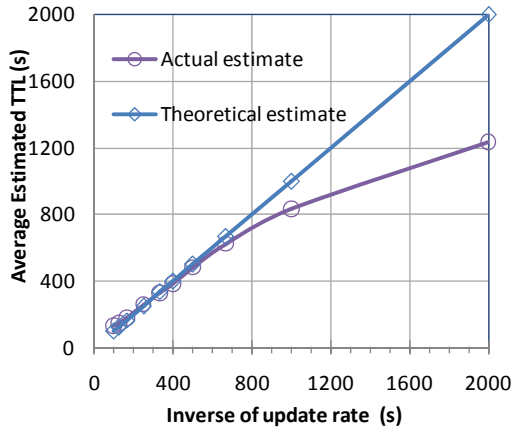

Fig. 5. Average TTL versus inverse of inter-update interval.

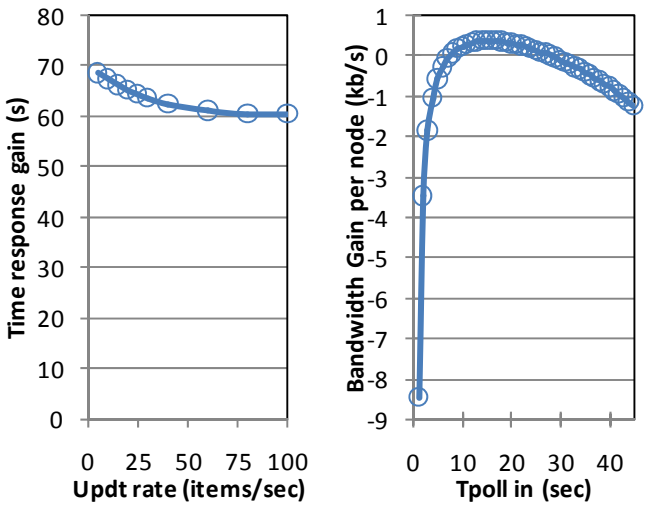

Fig. 6. Average Analytic performance measures: time response gain (left) and bandwidth gain (right)

regard to this last procedure, the performance of the system could be improved, especially in dynamic MANET environments, through incorporating into the design a replication scheme, similar to the one in [49], to replicate data on both the QDs and CNs and hence, reduce the overhead associated with node disconnections.

TTL calculation: In DCIM, the exact TTL calculation performed by the QD depends on whether the item was expired at the server or not, which is information contained in the SVRP and URP messages. The TTL value is calculated as per the steps below:

- If the item has changed on the server, the SVRP would contain the last updated time (denoted by LU_{new}) given the item had the prefetch bit not set, whereas the URP would contain the same value if this bit was set. In both cases, TTL is set to $(1\text{-}a)\times IUI + a \times (LU_{new}\text{-}Timestamp)$.
- If the item did not change on the server and the TTL did not expire on the QD, the TTL will not be modified. This case occurs because of the piggybacking procedure described before.
- If the item expired on the QD, but did not change on the server, the QD increases the TTL value by considering the current time as the update time, without changing the timestamp value it stores. The TTL value will be set to $(1\text{-}a)\times IUI + a \times (CurrentTime - Timestamp)$.

In some cases, the actual inter-update interval at the server could increase while the estimated inter-update interval may not have updated yet. This causes the last calculated inter-update interval when the item was last changed to become shorter than the time elapsed since the past update. This gives rise to a next expiry time occurring in the past. Should this situation occur, the QD reacts by setting the next expiry time to the estimated inter-update interval added to the current time (the time the item was validated when its timestamp did not change, or changed but the change was too old). This is done by setting TTL to $CurrentTime - Timestamp + IUI$. This situation stays in effect until the item gets a new timestamp (changes on the server).

For illustration purposes, a sample plot for the TTL value versus the update rate of a Poisson update process is depicted in Figure 5. This plot corresponds to an actual simulation run for 2000 seconds. It shows that at very low update rates (less than 1 update per 1000 sec) the estimated TTL does not adapt well. However, in actuality, time goes beyond the 2000 sec considered for this simulation time, meaning that more item updates will occur on the server during the longer time interval. It follows that the actual TTL will not diverge to the same extent as shown in Figure 5.

4 ANALYSIS

In this section we analyze DCIM to assess the bandwidth gain over a given time period and the query response time gain as compared to the poll-every-time (PET) consistency scheme. We define the bandwidth gain as the difference between the amounts of PET traffic and DCIM traffic, over a defined period of time. Similarly, the query response time gain is the difference between the times it takes to get the answer of the query (measured from the time of issuing the query) in the two schemes. The results are in agreement with the results shown in section 5.

Requests for data within the ad hoc network and arrival of data updates at the server are assumed to be random homogenous Poisson processes, and thus the inter-arrival times are represented by exponential random variables, as was suggested in [13] and [47]. We use λ_R to denote the rate of requests and λ_U for the rate of updates, and suppose that each query or data item can have its own rate. The PDFs of the inter-arrival times are therefore:

$$P_R(t) = \lambda_R e^{-\lambda_R t}, P_U(t) = \lambda_U e^{-\lambda_U t} \quad (1)$$

To estimate the response time and traffic gains, we borrow concepts from our previous work in [1] related to the average number of hops required in the various situations in the calculations:

- H_C is the average number of hops between the corner of the topology and a randomly selected node. It is used when a packet is sent between the server and a

node in system.

- H_R is the expected number of hops between any two randomly selected nodes.
- H_D is the expected number of hops to reach the QD containing the reference to the requested data, in the case of a hit.
- T_{in} is the transmission delay between two neighboring nodes (i.e., one hop delay), while T_{out} is the round trip time between the MANET and the server.
- S_D is the size of the data packet and S_R is the size of the request.

In what follows, we list the time response and bandwidth gains for DCIM when compared with the poll every time scheme. The details of the derivation are found in Appendices B and C.

4.1 Response Time Gain

In Appendix C we prove that the response time gain is:

$$T_{RTT}-P_{SC1}\times T_{RTT}-(1-P_{SC1})\times T_{MAN} \quad (2)$$

$$T_{RTT}=T_{out}+T_{in}(2H_C+H_D) \quad (3)$$

$$T_{MAN}=T_{in}(2H_R+H_D) \quad (4)$$

$$P_{SC1} = \lambda_U \times T_{RTT} \times e^{-\lambda_U \times T_{RTT}-1} \quad (5)$$

This measure is plotted in the left graph of Figure 6, where values consistent with the corresponding average values in the simulations were used: H_D=5, H_R=5.21, H_C=5.21, T_{in}=5ms, T_{out}=70ms, and λ_U=1/500. As implied from the expression above, the gain mainly depends on the update rate which causes it to decrease slightly when it increases. This agrees with Figure 8 in the experimental section below, where at low update rates, the difference in delay is around 70 ms, and decreases to be around 60 ms for large update rates.

4.2 Bandwidth Gain

The expression for the bandwidth gain (*GB*) is derived in Appendix D and is:

$$R_{tot}\times(P_{poll}\times B_{pi}+(1-P_{poll})\times B_{po})-M\times(B_{Rpoll}+B_{pollnc}+B_{pollc}) \\ -B_{Rpigg}-B_{piggnc}-B_{piggc} \quad (6)$$

where the the bandwidth usage of PET is described as:

$$R_{tot}\times(P_{poll}\times B_{pi}+(1-P_{poll})\times B_{po}) \quad (7)$$

and the is the bandwidth usage of DCIM is given by:

$$M\times(B_{Rpoll}+B_{pollnc}+B_{pollc})-B_{Rpigg}-B_{piggnc}-B_{piggc} \quad (8)$$

In the expressions above, $B_{po}=S_R(H_D+H_C)+S_DH_C+S_RH_R$, $B_{pi}=S_R(H_D+2H_C)$,$R_{tot} = \lambda_R x T_{pigg}$, $T_R = 1/\lambda_R$, and T_{pigg} is the piggybacking interval. T_{poll} is the polling interval, M is the number of polling intervals, $P_{poll} = e^{-\lambda_U \times T_R}$, $B_{pollc}=l\times(S_DH_C+S_RH_R)$, $B_{pollnc}=(K-l)\times S_RH_C$, $B_{Rpoll}=K\times S_RH_C$, $B_{piggc}= m\times S_D H_C$, $B_{piggnc}= (N-m)\times S_R H_C$, $B_{Rpigg}=N\times S_RH_C$, and N is the number of items. The expressions for l, K, and m, are found in Appendix D.

The bandwidth gain is plotted in the right graph of figure 6, where in addition to the same hop count values as those utilized above, the following values are used: $\lambda_R=\lambda_U$=1/500, N=4000, M=20, S_R=0.5KB, and S_D=10KB. It is worth noting, that N represents the number of cached items (requested at least once before), rather than the total number of items; this matches the experimental results since not all items will be requested within the simulation time. In effect, the traffic resulting from large piggybacking intervals is lower than that of small piggybacking interval. Also, the traffic demands for DCIM decrease exponentially for small polling intervals in both the analytical and experimental results shown in figure 11.

TABLE 4
SUMMARY OF THE DEFAULT SIMULATION PARAMETERS

Simulation Parameter	Default Value	Simulation Parameter	Default Value
Simulation time	2000 sec	Size of data item	10 KB
Network size	400×400 m^2	Number of data items updated/sec	20
Wireless bandwidth	2 Mb/s	Delay at the data source	40 ms
Node trans. range	100 m	Node request period	10 sec
Number of nodes	100	Node request pattern	Zipf (θ=1)
mobility model	RWP	Node caching capacity	200 KB
Node speed (v)	2 (m/s)	Cache Replacement	LRU
Node pause time	30 sec	Polling interval	2 sec
Total number of data items	10,000	N_{poll}	20

5 EXPERIMENTAL RESULTS

DCIM was implemented using ns2 [39], and a new database class was developed that mimics the server process in storing and updating data items and in processing the validation requests. Moreover, timers in ns2 were utilized to implement the monitoring thread: the timer sleeps for the polling interval duration and then wakes up to run the inner-loop function. According to the design, after N_{poll} runs of the inner-loop, (the piggybacking interval) the outer-loop is invoked. Ns2 is a single threaded simulator, but it is nevertheless capable of controlling the operations of the timers autonomously, thus acting similar to a multithreaded application.

Two additional schemes were implemented for comparison. The first is the poll-every-time mechanism (considered in Section 4), where each time an item is requested, it is validated. The second is the fixed-TTL mechanism, where all items have the same expiry interval. The TTL value is calculated by adding to the current time the expiry interval, and when a TTL value expires, the item is flagged as such, and is fetched from the server whenever it is requested.

The simulation area was set to 400×400m^2, populated with 100 nodes that were randomly distributed. Propagation was according to the two-ray model, and the node's bitrate was set to 2 Mbps. Mobility was based on the random waypoint model (RWP), with a maximum speed of 2

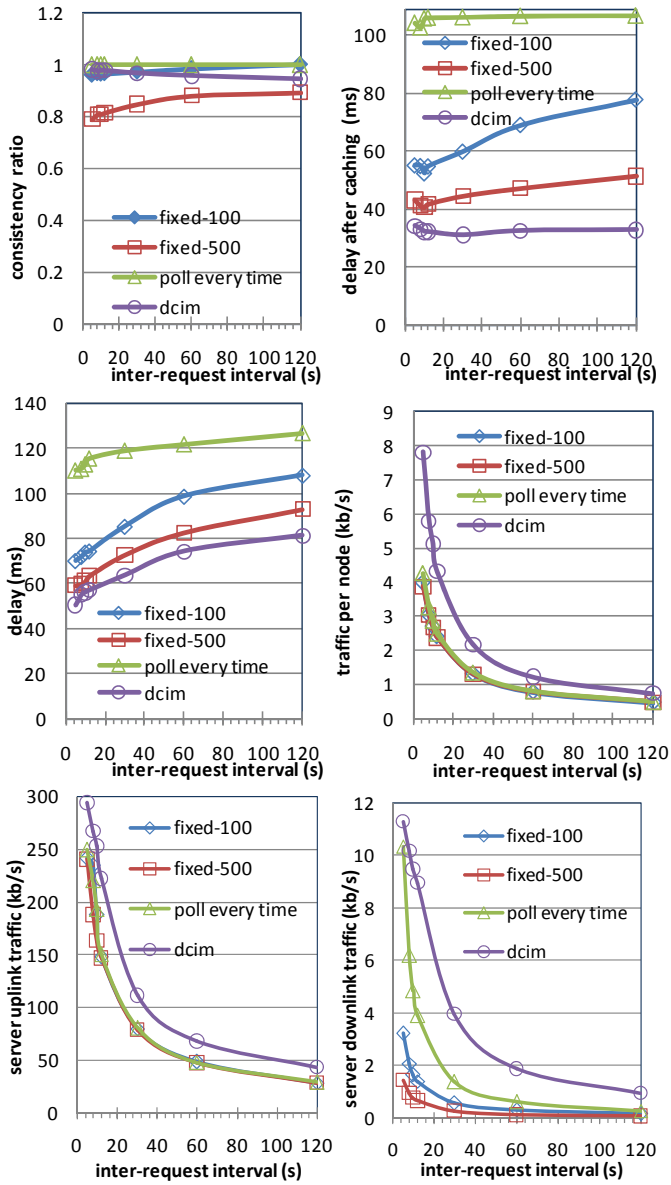

Fig. 7. Performance measures versus inter-request times

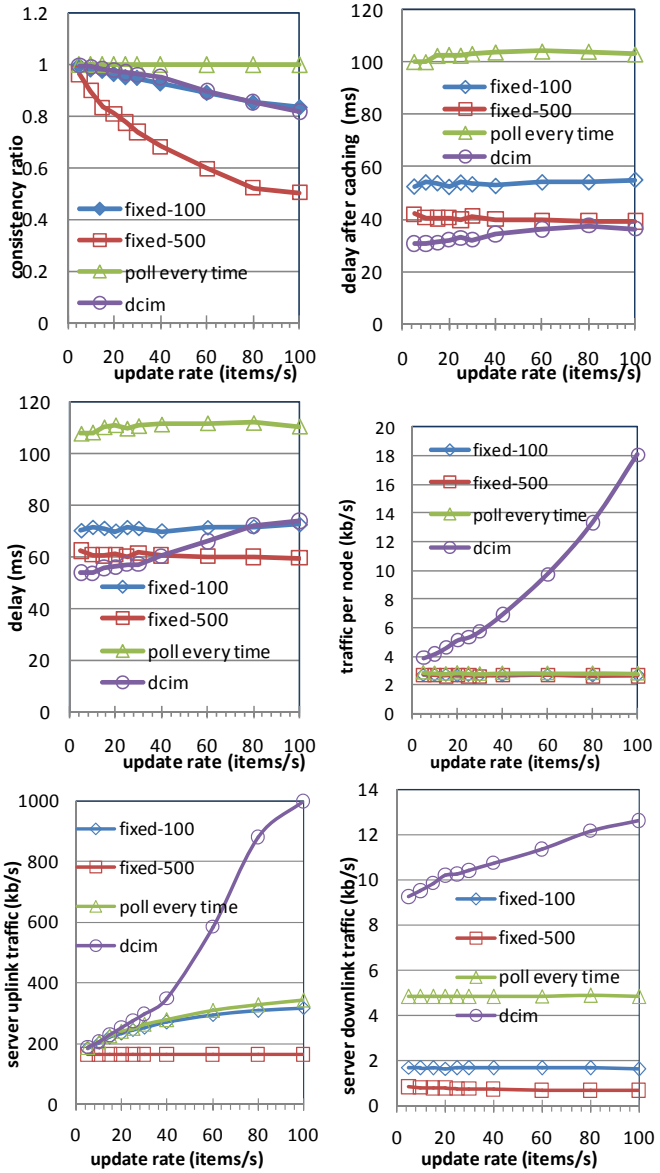

Fig. 8. Performance measures versus update rates

m/s. The server node was connected to the MANET via a gateway and a wired link whose propagation delay was simulated at 40ms, thus resulting in a server access delay of 80ms. The server has 10,000 items which are updated according to a Poisson random process at an average rate of about 20 items per sec. In the default scenario, each node issues a data request every 10 seconds according to a Zipf access pattern, frequently used to model non-uniform distributions [43]. In Zipf law, an item ranked i $(1 \le i \le nq)$ is accessed with probability: $1/\left(i^{\theta}\sum_{k=1}^{n_q} 1/k^{\theta}\right)$, where θ ranges between 0 (uniform distribution) and 1 (strict Zipf distribution). The default value of the Zipf parameter θ was set to 1. In the default scenario, there are 7 QDs, and the capacity for each of the CNs (Caching Nodes) is 200 Kb. The simulation parameters are summarized in Table 4.

The reported results are from 5 experiments that involve varying the request rate, the update rate, the zipf parameter, the maximum velocity, and the polling interval. The results are the 1) consistency ratio (with the data source), 2) query delay (regardless of the source of the results), 3) cached data query delay, 4) uplink traffic, 5) downlink traffic, and 6) average overhead traffic.

5.1 Varying the request rate

In this experiment, the inter-request interval was varied between 5s and 120s. The results are plotted in the graphs of Figure 7, where it is evident that the poll every time scheme provides the highest consistency ratio (top left graph), since the requested items are validated for each request. This causes the items to be always fresh, except in certain cases when they change just after being validated. However, when using fixed TTL, the caches might serve stale items (as in the case of TTL=500s), but this possibility decreases when the TTL is less than the update

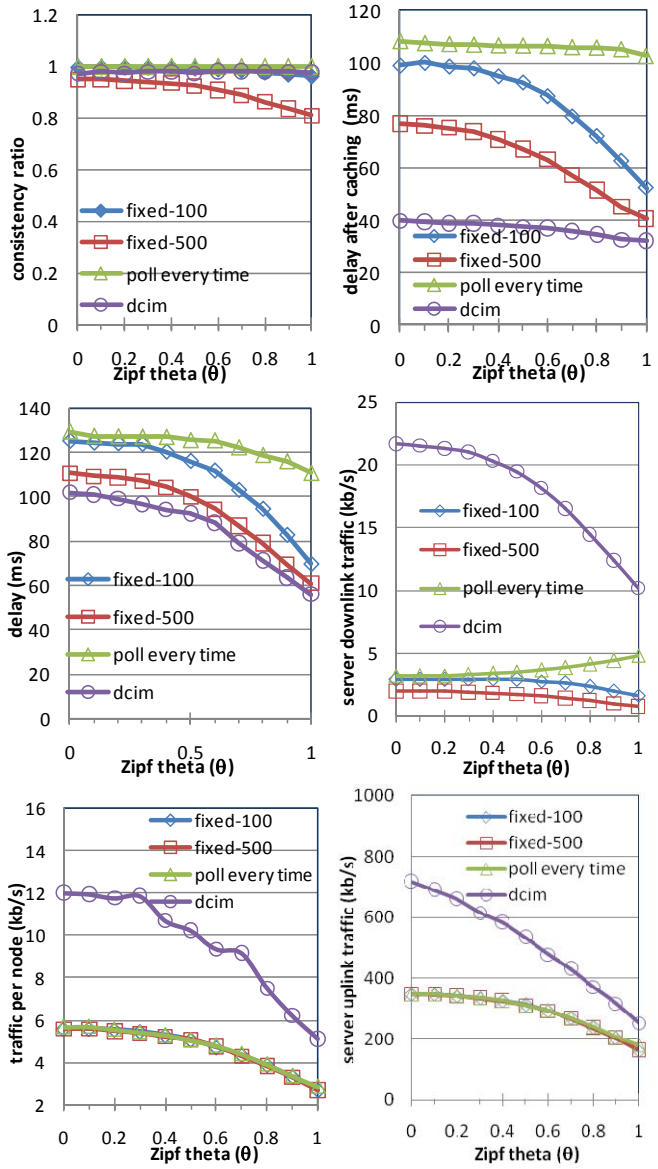

Fig. 9. Performance measures versus the zipf parameter

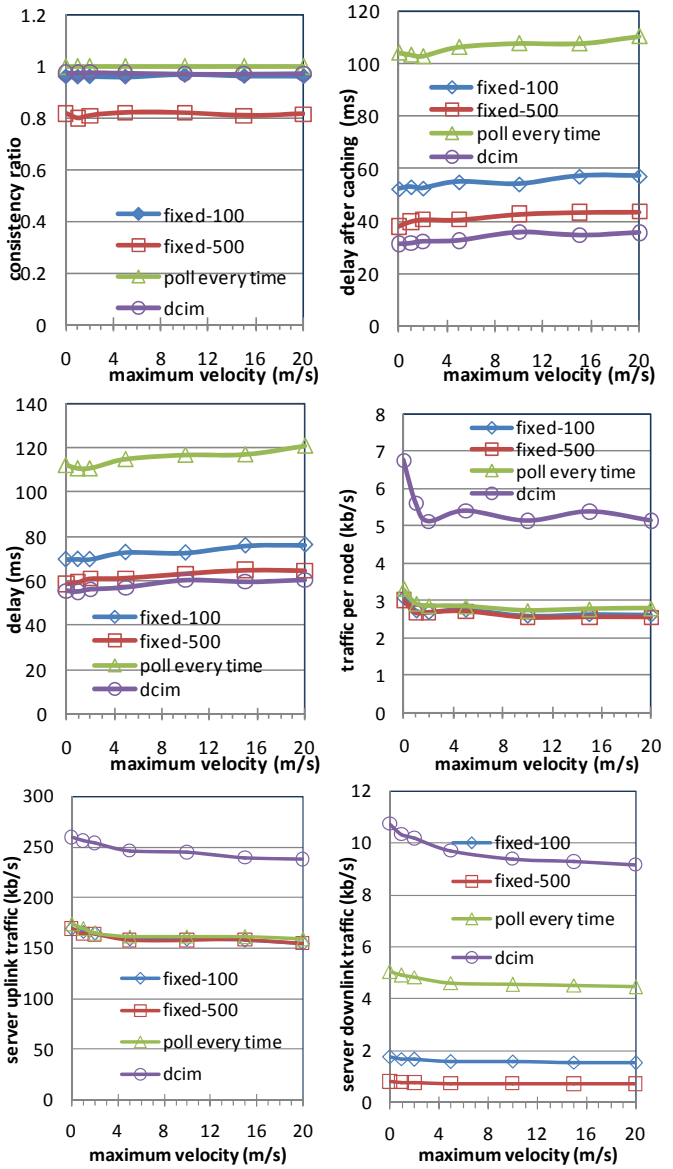

Fig. 10. Performance measures versus the node maximum velocity

interval (as in the case of TTL=100s). As a matter of fact, getting the right TTL value is a key issue in relation to the performance of client-based consistency approaches. DCIM is a better approach as it tries to get the appropriate TTL value through piggybacking, which helps in getting a high consistency ratio. Moreover, prefetching enables DCIM to provide a high hit ratio, and hence much lower delays than the other approaches. As also shown, the query delay gets smaller after the item is cached but increases by a small margin due to less prefetching as the request rate decreases. Finally, DCIM consumes more traffic on the server side due to prefetching, but is not far off from the other schemes. As for the node traffic, by piggybacking large amount of items, DCIM consumes more traffic when compared to other approaches. However, as the request rate decreases, prefetching does not happen that often, and this leads to lower traffic as shown in the graph. This is how DCIM adapts prefetching to the request rate of items.

5.2 Varying the update rate

The results for this scenario are shown in Figure 8. A TTL value of 100 seconds is less than the inter-update intervals in all of the scenarios simulated, and hence, it must provide the best consistency level. As shown, DCIM's consistency ratio coincides with that of TTL=100s, which is higher than that of TTL=500s. Of course, increasing the update rate in any TTL algorithm would decrease its consistency, but with a good TTL estimate, an acceptable consistency could be obtained (comparing TTL=500s and DCIM at 100 update/sec). Nevertheless, fixed TTL approaches have higher hit rates than poll every time, but less than DCIM, which uses prefetching. This implies that the delay after caching is the lowest in case of DCIM, and does not vary as the update rate changes since prefetching is altered by request rates.

The gains in delay and consistency, discussed above, are manifested in increased traffic as the update ratio in-

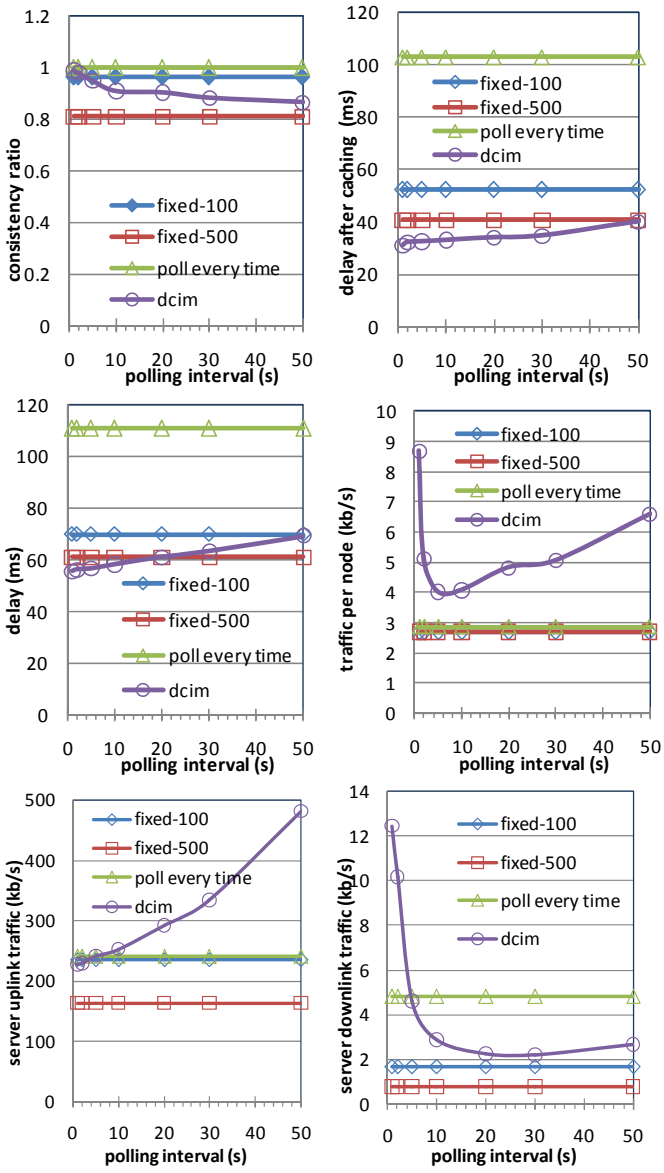

Fig. 11. Performance measures versus the polling interval

creases. However this traffic is not high at the server, and is very low in the MANET (less than 20 kbps, while the bandwidth is 2Mbps). The reason for the high traffic is the piggybacking of requests, which increases in frequency as update rates increase. Without this traffic though, the QDs cannot infer the update rate and cannot calculate reliable TTL values.

5.3 Varying the Zipf parameter

This scenario studies the effect of the popularity of certain data items on performance by varying the value of the zipf parameter θ. The results are shown in Figure 9.

In actuality, varying the θ value is analogous to varying the items' request rates. This scenario actually shows the prefetching adaptation to the request rates, which was explained in Section 3. As θ increases, the diversity of the requested items decreases, meaning that a smaller subset of the items is requested more. In case one item is updated at the server before the TTL expiration, more stale cached items will result. This is why the consistency ratio decreases as θ increases. However, DCIM maintains the TTL for all items regardless of their request rates, and this gives a constant consistency at 98%. The situation is reversed when considering hit ratios. For low θ values the hit ratio for fixed TTL is low since item requests are distributed across all items, which increases the probability of having expired items while the request interval is fixed. As θ increases, the same requests will be distributed over a smaller set which increases the probability of hits. It is evident that through prefetching, DCIM provides nearly constant hit rate, which results in lower delays as explained before. DCIM produces more traffic when compared to the other approaches, but this traffic decreases as θ increases since there is a smaller subset of items to validate. More items will have lower update rates, and will not be validated frequently.

5.4 Varying the maximum node velocity

The maximum node velocity is varied between 0 m/s and 20 m/s, and the results are shown in the graphs of Figure 10. Velocity changes show no special results, although there is a mild increase in the delay when velocity increases, which is considered normal. In fact, the use of a proactive routing protocol masks the delay by making the paths always available.

The graphs of Figure 10 show expected results as topology changes are irrelevant to a client-based consistency scheme. We note that the reported MANET traffic is the DCIM overhead traffic and does not include routing traffic which must have increased as a function of velocity.

5.5 Varying the polling interval

Here, the polling interval is varied between 1 and 50 seconds, while the fixed TTL values are kept constant, i.e. 100 and 500 seconds. The results are shown the graphs of Figure 11. The increase of the polling interval causes a decrease in both the consistency ratio and the hit ratio, and consequently, an increase in the delay which remains below that of fixed TTL. Moreover, the traffic in the uplink direction increases when the piggyback interval increases due to the decrease of hit rate. Finally, it is worthy to point out the decrease in the traffic in the downlink direction at the server and the sharp decrease in the traffic per node in the network. These results are expected since with increasing the polling interval, the validation requests originating from the inner loop function become more apart in time. However, when piggybacking intervals are very large, the QD predicts that items will be requested before the end of the piggybacking interval. This leads to more prefetching and consequently more traffic.

5.6 Energy Consumption

DCIM is by design a client based approach, and moving all the processing to the client side might hinder the mobile devices' energy resources. To investigate this possibility, we conducted an experiment to assess the energy load DCIM imposes on the QD, in particular, as it is responsible for monitoring and maintaining the items in the cache. The experiment was conducted in two stages. The

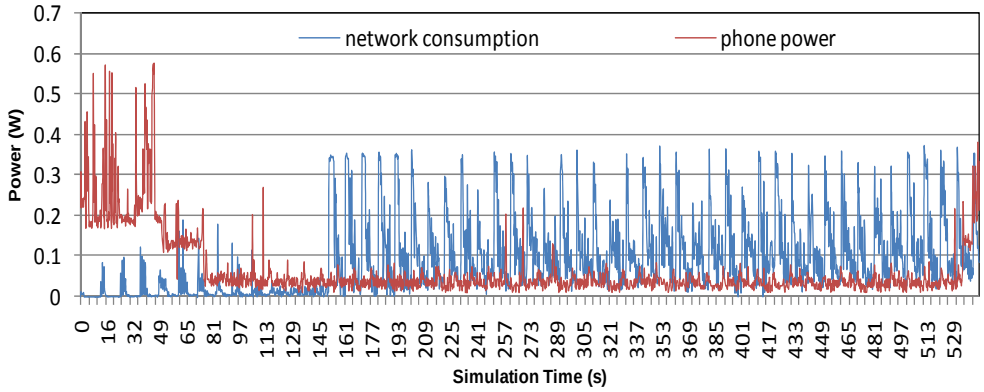

Fig. 12. Energy consumption at the QD

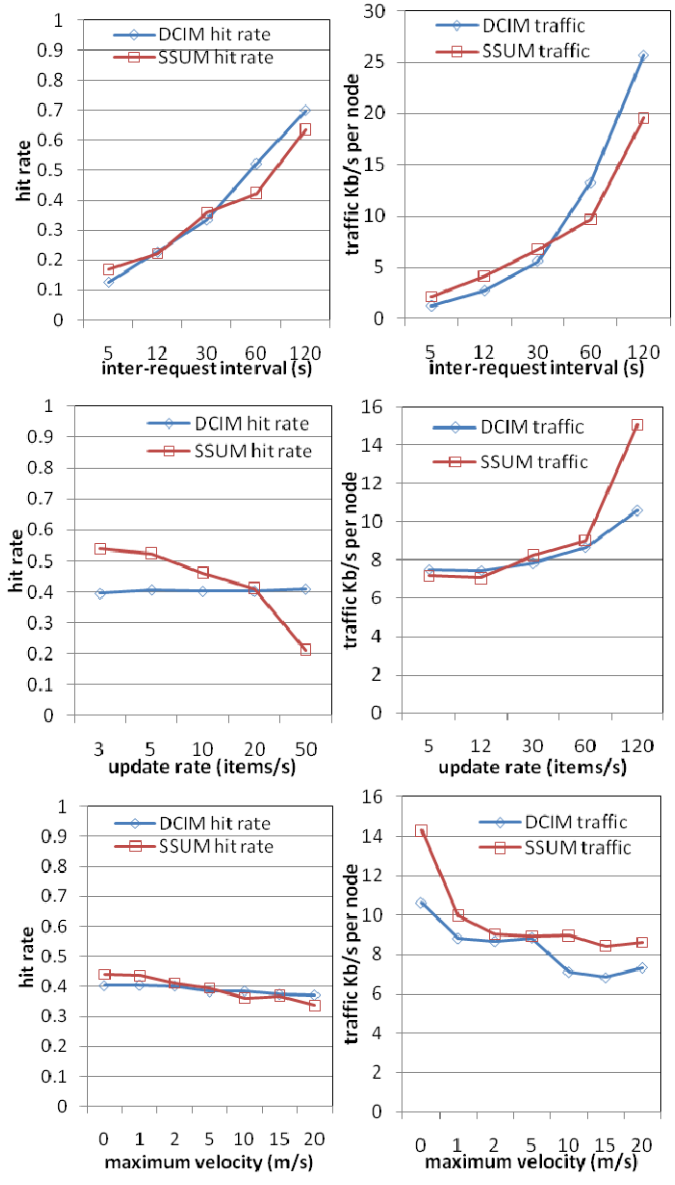

Fig. 13. Performance comparison between SSUM and DCIM

first had to do with studying the processing energy consumption of DCIM on a QD, while the second stage assessed the network energy consumption on the QD. For this purpose we implemented the QD functions using J2ME and installed the application on a NOKIA E71 device. We monitored the performance of the application using an energy profiler, a tool distributed by NOKIA, for 300 seconds. This tool measures the internal current in the device and reports real time power consumption. As shown in Figure 12, in the first 30 seconds, the application was initiating, so it does not count as a part of the QD operations. After that, it is obvious that processing consumes little energy (less than 0.068W on average).

In the second stage we monitored the energy consumed by network communications, using the ns2 energy model. In ns2, a node is given an initial energy and after each transmission or reception, this energy is diminished by the transmission/reception power multiplied by the corresponding delay. The results are also shown in Figure 12. The first 500 seconds are recorded from one of the simulations performed before, where the average transmission/reception power was 0.115W. The average power consumption from networking and processing was 0.182W. To understand the implications of this value, we consider the NOKIA E71 device's battery which has a capacity of 1548 mA.h and a voltage rating of 3.9V. With this rate of power consumption, the battery would last for about 33 hours of continuous use.

5.7 Comparison with SSUM

This section compares the performance of DCIM to that of SSUM [38] (mentioned in Section 3.1). In SSUM the server propagates item update information to the QDs, and stores information about each item cached in the network. For each item, the server computes a ratio of its update rate to its request rate. If this ratio exceeds a given threshold, the item is deleted from the server's state table, and no updates about it are propagated. However, if this ratio falls below another threshold, the caching node receives updates for the corresponding item. Hence, SSUM reduces traffic associated with unnecessary updates for items that are more updated than requested.

DCIM is an alternative approach that relies on the client side to implement the consistency mechanism. The server stores no information about the MANET or the history, except for the last update time. Also, DCIM adapts to the update and request rates differently, but like SSUM, it also tries to save traffic. The update rates are estimated on the client side, rather than being maintained by the serv-

TABLE 5
HIGH LEVEL COMPARISON BETWEEN DCIM AND THE OTHER CACHE CONSISTENCY APPROACHES

Property/metric	Poll Each Time	Fixed TTL
Type	Pull (client side)	Pull (client side)
Query Delay	High	Based on chosen TTL
Consistency Ratio	Highest	Depending on chosen TTL
Traffic at Server	low	Low to medium
Practicality	Impractical due to high delays + disconnections from server	Impractical: TTL value does not work for all items in all scenarios
MANET traffic	Low	Low
Scalability	scalable	Not scalable, TTL value assignment is not scalable
Property/metric	**SSUM**	**DCIM**
Type	Push (server side)	Pull (client side)
Query Delay	Low	Low
Consistency Ratio	High	High (depending on chosen polling interval)
Traffic at Server	Medium	Medium, < 10 Kb/s per node. (network bandwidth = 2 Mbps)
Practicality	Limited practicality: requires server maintaining state of MANET	Most practical, since it achieves low delay, high consistency ratio
MANET traffic	Medium	Medium
Scalability	Limited Scalability: Server maintains state of MANET	Scalable like COACS, and also because it operates on the client side.

er. SSUM achieves consistency with a delta equal to the communication time between the server and caching nodes. However as illustrated by the graphs of Figures 7 through 11, DCIM is able to provide consistency guarantees if the polling interval is not high, but at the expense of increased traffic which is actually comparable to what SSUM generates.

To confirm the above analysis, three scenarios from [38] were run to compare with DCIM using the same parameters. The area was set to 750×750m², the θ value used was 0.5, the request period was 20 sec, and the item size was varied between 1 and 10 KB. All other parameters kept the same values as before. We report the hit rate and node traffic versus the request interval, update rate, and node velocity. As seen in Figure 13, both approaches perform similarly in terms of hit rate and node traffic. The θ value of 0.5 means there is more variety in the requested items, and given there is a total of 10,000 items, the probability of requesting an element several times is low, which reflects on the hit rate values. As for the traffic, each approach has its share of traffic consumption. In SSUM, maintaining the server state, and pushing data items proactively constitute the traffic overhead. While in DCIM, validation requests and proactive fetching of items are responsible for its overhead traffic.

In Table 5, we conclude the experimental results presented in this section, by comparing DCIM to the presented pull based approaches and SSUM according to delay, consistency, traffic, scalability, and practicality.

6 CONCLUSION

In this work, we presented a pull based approach to insure the consistency of data items cached inside a MANET. This approach relies on estimating the inter update intervals of the data items to set their expiry time. It makes use of piggybacking to increase the estimation accuracy of the inter update interval and to reduce traffic, and also prefetches items with high request rates to reduce query delays. We compared this approach to two pull-based approaches, namely fixed TTL and client polling, in addition to the server-based approach SSUM. The evaluation showed that DCIM provides better performance than the first two schemes, but at the expense of more traffic (in the order of 10 Kbps), and a comparable bandwidth consumption with SSUM.

For future work, we will explore three directions to extend DCIM. First, we will explore more sophisticated TTL algorithms to replace the running average formula. Secondly, as we indicated in section 3, we will design a replica allocation scheme to increase data availability. It will include an update propagation method that insures high data consistency among the replicas with minimum traffic. Thirdly, DCIM assumes that all nodes are well behaved, as issues related to security were not considered. However, given the possibility of network intrusions, we will explore integrating appropriate security measures into the system functions. These functions include the QD election procedure, QD traversal, QD and CN information integrity, and TTL monitoring and calculation. The first three are typical in a MANET and can be mitigated through encryption and trust schemes [44] [45]. The last issue was not tackled before, except in the case of [46] which considers the manipulation of invalidation reports. Similarly, a change in the last update time in DCIM can trick the QD into deciding that the item was not changed on the server.

REFERENCES

[1] H. Artail, H. Safa, K. Mershad, Z. Abou-Atme, N. Sulieman, "COACS: A Cooperative and adaptive caching system for MANETS", *IEEE TMC*, v.7, n.8, pp. 961-977, 2008.

[2] D. Barbara and T. Imielinski, "Sleepers and Workaholics: Caching Strategies for Mobile Environments," In *Proc. ACM SIGMOD*, pp. 1-12, May 1994.

[3] G. Cao, "A Scalable low-Latency Cache Invalidation Strategy for Mobile Environments," *IEEE TKDE*, v. 15, n. 5, pp. 1251-1265, 2003.

[4] D. Li, P. Cao, and M. Dahlin. WCIP: Web Cache Invalidation Protocol. IETF Internet Draft, March 2001, http://tools.ietf.org/html/draft-danli-wrec-wcip-01.

[5] Jiannong Cao; Zhang, Y.; Cao, G.; Li Xie; , "Data Consistency for Cooperative Caching in Mobile Environments," Computer , v.40, n.4, pp.60-66, 2007

[6] P. Cao, C. Liu, "Maintaining strong cache consistency in the World-Wide Web," *IEEE Trans. Computers*, v. 47, pp. 445–457, 1998.

[7] W. Li, E. Chan, D. Chen and S. Lu, "Maintaining probabilistic consistency for frequently offline devices in mobile ad hoc networks,"

in Proc. 29th IEEE Int'l Conf. Distributed Computing Systems, pp. 215-222, 2009.

[8] J. Jung, A.W. Berger, H. Balakrishnan, Modeling TTL-based internet caches, *in Proc. IEEE INFOCOM 2003*, San Francisco, CA, March 2003.

[9] B. Krishnamurthy, C. Wills, "Study of piggyback cache validation for proxy caches in the World Wide Web," In *Proc. USENIX*, Monterey, CA, December 1997.

[10] J. Lee, K. Whang, B. Lee, and J. Chang, "An update-risk based approach to TTL estimation in web caching," In Proc. WISE 2002, pp. 21-29, 2002.

[11] D. Wessels, Squid Internet object cache, August 1998, http://squid.nlanr.net/Squid/.

[12] Y. Huang, J. Cao, Z. Wang, B. Jin and Y. Feng, "Achieving flexible cache consistency for pervasive internet access," In Proc. PerCom, pp. 239-250, 2007.

[13] O. Bahat, A. Makowski, "Measuring consistency in TTL-based caches," Performance Evaluation, v. 62, pp. 439–455, 2005.

[14] M. Denko, J. Tian, "Cooperative Caching with Adaptive Prefetching in Mobile Ad Hoc Networks," In Proc. IEEE WiMob'2006, pp.38-44, June 2006.

[15] J. Jing, A. Elmagarmid, A. Helal, and R. Alonso, "Bit-Sequences: An Adaptive Cache Invalidation Method in Mobile Client/Server Environments," Mobile Networks and Applications, pp. 115-127, 1997.

[16] Q. Hu and D. Lee, "Cache algorithms based on adaptive invalidation reports for mobile environments," *Cluster Computing*, pp. 39-50, 1998.

[17] Z. Wang, S. Das, H. Che, and M. Kumar, "A scalable asynchronous cache consistency scheme (SACCS) for mobile environments," *IEEE TPDS*, v. 15, n. 11, pp. 983-995, 2004.

[18] S. Lim, W. C. Lee, G. Cao and C. R. Das, "Cache invalidation strategies for internet-based mobile ad hoc networks," Computer Comm., v. 30, pp. 1854-1869, 2007.

[19] Kahol, S. Khurana, S. Gupta, and P. Srimani, "A scheme to manage cache consistency in a distributed mobile wireless environment," *IEEE TPDS*, v. 12, n. 7, pp. 686-700, 2001.

[20] V. Cate, "Alex – A Global Filesystem," in *Proc. USENIX*, pp. 1-12, May 1992.

[21] Y. Li and G. Le, "A caching model for real-time databases in mobile ad-hoc networks," Database and Expert Systems Applications, pp. 186-196, Springer, Berlin, Heidelberg, 2005.

[22] J. Cao, Y. Zhang, L. Xie, and G. Cao, "Consistency of Cooperative Caching in Mobile Peer-to-Peer Systems over MANETs," In *Proc. 3rd Int'l workshop on mobile distributed computing*, v. 6, pp. 573-579, 2005.

[23] W. Li, E. Chan, Y. Wang, D. Chen, "Cache Invalidation Strategies for Mobile Ad Hoc Networks," In Proc. ICPP 2007, pp.57, Sept. 2007.

[24] Y. Huang, J. Cao, B. Jin, X. Tao and J. Lu, "Cooperative Cache Consistency Maintenance for Pervasive Internet Access," Wireless Comm. and Mobile Computing, v.10, pp. 436-450, 2009.

[25] L. Yin and G. Cao, "Supporting cooperative caching in ad hoc networks," IEEE TMC, v. 5, n. 1, pp. 77- 89, 2006.

[26] G. Cao; L. Yin; C. Das, "Cooperative cache-based data access in ad hoc networks," Computer, v. 37, n. 2, pp. 32-39, 2004.

[27] X. Tang, J. Xu, W-C. Lee, "Analysis of TTL-based consistency in unstructured peer-to-peer networks," IEEE TPDS, v. 19, n. 12, pp.1683-1694, 2008.

[28] L. Bright, A. Gal, and L. Raschid, "Adaptive pull-based policies for wide area data delivery," ACM Trans. Database Systems, v. 31, n. 2, pp. 631 – 671, 2006.

[29] Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz and K. Worrell, "A hierarchical internet object cache," In Proc. USENIX, pp. 13, 1996.

[30] V. Jacobson, "Congestion Avoidance and Control," ACM SIGCOMM Computer Comm. Review, v. 25, pp. 187, 1995.

[31] X. Chen and P. Mohapatra, "Lifetime behavior and its impact on web caching," in Proc. IEEE Workshop on Internet Applications, pp. 54–61, 1999.

[32] Y. Sit, F. Lau, C-L. Wang, "On the cooperation of Web clients and proxy caches," In Proc. 11th Int'l Conf. Parallel and Distributed Systems, pp. 264- 270, July 2005.

[33] Urgaonkar, A. Ninan, M. Raunak, P. Shenoy, K. Ramamritham, "Maintaining mutual consistency for cached web objects," in Proc. 21st Int'l Conf. Distributed Computing Systems, p. 371, 2001

[34] N. Chand, R. Joshi and M. Misra, "A zone co-operation approach for efficient caching in mobile ad hoc networks," Int'l J. of Comm. Systems, v. 19, pp. 1009-1028, 2006.

[35] Y. Du, S.K.S. Gupta, "COOP - A cooperative caching service in MANETs," In Proc. ICAS-ICNS, pp.58-58, Oct. 2005.

[36] Y. Du, S. K. S. Gupta and G. Varsamopoulos, "Improving on-demand data access efficiency in MANETs with cooperative caching," Ad Hoc Networks, v. 7, pp. 579-598, 2009.

[37] Holt, "Forecasting seasonals and trends by exponentially weighted moving averages," International Journal of Forecasting, v. 20, n. 1, pp. 5-10, 2004.

[38] K. Mershad and H. Artail, "SSUM: Smart Server Update Mechanism for Maintaining Cache Consistency in Mobile Environments," IEEE TMC, v. 9, n. 6, pp.778-795, 2010.

[39] B. Krishnamurthy, C.E. Wills, "Piggyback server invalidation for proxy cache coherency," *In Proc. 7th WWW Conf.*, Brisbane, Australia, April 1998.

[40] Y. Fang, Z. J. Haas, B. Liang and Y. B. Lin, "TTL prediction schemes and the effects of inter-update time distribution on wireless data access," Wireless Networks, v. 10, pp. 607-619, 2004.

[41] J. Shim, P. Scheuermann, R. Vingralek, "Proxy cache algorithms: design, implementation, and performance," IEEE TKDE, v.11, n. 4, pp.549-562, 1999.

[42] N. Dimokas, D. Katsaros and Y. Manolopoulos, "Cache consistency in Wireless Multimedia Sensor Networks," Ad Hoc Networks, v. 8, n. 2, pp. 214-240, 2010.

[43] G. Zipf, Human Behavior and the Principle of Least Effort. Addison-Wesley, 1949

[44] N. A. Boudriga and M. S. Obaidat. "Fault and intrusion tolerance in wireless ad hoc networks".In Proc. IEEE WCNC, v. 4, pp. 2281–2286, Washington, DC, 2005.

[45] P. Papadimitratos, Z. Haas. "Secure data transmission in mobile ad hoc networks", In Proc. ACM workshop on wireless security, pp. 41–50, New York, 2003.

[46] W. Zhang and G. Cao, "Defending against cache consistency attacks in wireless ad hoc networks," Ad Hoc Networks, vol. 6, pp. 363-379, 2008.

[47] H. Maalouf, and M. Gurcan, "Minimisation of the update response time in a distributed database system," Performance Evaluation, v. 50, n. 4, pp. 245-66, 2002.

[48] Mondal, S. K. Madria and M. Kitsuregawa, "An Economic Incentive Model for encouraging Peer Collaboration in Mobile-P2P networks with support for constraint queries," Peer-to-Peer Networking and Applications, vol. 2, pp. 230-251, 2009.

[49] T. Hara and S. Madria, "Dynamic Data Replication using Aperiodic Updates in Mobile Adhoc Networks," in Database Systems for Advanced Applications, 2004, pp. 111-136.