Benefit-based Data Caching in Ad Hoc Networks

Bin Tang, Himanshu Gupta, Samir Das
Computer Science Department
Stony Brook University
Stony Brook, NY 11790
Email: {bintang,hgupta,samir}@cs.sunysb.edu

***Abstract*— Data caching can significantly improve the efficiency of information access in a wireless ad hoc network by reducing the access latency and bandwidth usage. However, designing efficient distributed caching algorithms is non-trivial when network nodes have limited memory. In this article, we consider the cache placement problem of minimizing total data access cost in ad hoc networks with multiple data items and nodes with limited memory capacity. The above optimization problem is known to be NP-hard. Defining *benefit* as the reduction in total access cost, we present a polynomial-time centralized approximation algorithm that provably delivers a solution whose benefit is at least one-fourth (one-half for uniform-size data items) of the optimal benefit. The approximation algorithm is amenable to localized distributed implementation, which is shown via simulations to perform close to the approximation algorithm. Our distributed algorithm naturally extends to networks with mobile nodes. We simulate our distributed algorithm using a network simulator (*ns2*), and demonstrate that it significantly outperforms another existing caching technique (by Yin and Cao [30]) in all important performance metrics. The performance differential is particularly large in more challenging scenarios, such as higher access frequency and smaller memory.**

I. Introduction

Ad hoc networks are multihop wireless networks of small computing devices with wireless interfaces. The computing devices could be conventional computers (e.g., PDA, laptop, or PC) or backbone routing platforms, or even embedded processors such as sensor nodes. The problem of optimal placement of caches to reduce overall cost of accessing data is motivated by the following two defining characteristics of ad hoc networks. Firstly, the ad hoc networks are multihop networks without a central base station. Thus, remote access of information typically occurs via multi-hop routing, which can greatly benefit from caching to reduce access latency. Secondly, the network is generally resource constrained in terms of channel bandwidth or battery power in the nodes. Caching helps in reducing communication, which results in savings in bandwidth as well as battery energy. The problem of cache placement is particularly challenging when each network node has limited memory to cache data items.

In this paper, our focus is on developing efficient caching techniques in ad hoc networks with memory limitations. Research into data storage, access, and dissemination techniques in ad hoc networks is not new. In particular, these mechanisms have been investigated in connection with sensor networking [15, 25], peer-to-peer networks [1, 17], mesh networks [18], world wide web [24], and even more general ad hoc networks [12, 30]. However, the presented approaches have so far been somewhat “ad hoc” and empirical without any strong analytical results. In contrast, the theory literature abounds in analytical studies into the optimality properties of caching and replica allocation problems (see, for example, [3]). However, distributed implementations of these techniques and their performances in complex network settings have not been investigated. Its even unclear whether these techniques are amenable to efficient distributed implementations. Our goal in this paper is to develop an approach that is both analytically tractable with a provable performance bound in a centralized setting, and is also amenable to a natural distributed implementation.

In our network model, there are multiple data items; each data item has a server, and a set of clients that wish to access the data item at a given frequency. Each node carefully chooses data items to cache in its limited memory to minimize the overall access cost. Essentially, in this article, we develop efficient strategies to select data items to cache at each node. In particular, we develop two algorithms – a centralized approximation algorithm which delivers a 4-approximation (2-approximation for uniform-size data items) solution, and a localized distributed algorithm which is based on the approximation algorithm and can handle mobility of nodes and dynamic traffic conditions. Using simulations, we show that the distributed algorithm performs very close to the approximation algorithm. Finally, we show through extensive experiments on *ns-2* [11] that our proposed distributed algorithm performs much better than prior approach over a broad range of parameter values. Ours is the first work to present a distributed implementation based on an approximation algorithm for the general problem of cache placement of multiple data items under memory constraint.

The rest of the paper is organized as follows. In Section II, we formally define the cache placement problem addressed in this paper, and present an overview of the related work. In Section III, we present our designed centralized approximation and distributed algorithms. Section IV presents simulation results. We end with concluding remarks in Section V.

II. Cache Placement Problem

In this section, we formally define the cache placement problem addressed in our article, and discuss related work.

A multi-hop ad hoc network can be represented as an undirected graph $G(V, E)$ where the set of vertices V represents the nodes in the network, and E is the set of weighted edges in the graph. Two network nodes that can communicate directly with each other are connected by an edge in the graph. The edge weight may represent a link metric such as loss rate, delay, or transmission power. For the cache placement problem addressed in this article, there are multiple data items and each data item is served by its server (a network node may act as a server for more than one data items). Each network node has limited memory and can cache multiple data items subject to its memory capacity limitation. The objective of our cache placement problem is to minimize the overall access cost. Below, we give a formal definition of the cache placement problem addressed in this article.

Problem Formulation. Given a general ad hoc network graph $G(V, E)$ with p data items $D_1, D_2, \ldots, D_p$, where a data item D_j is served by a server S_j. A network node may act as a server for multiple data items. We use m_i to represent the memory capacity of a network node i. *For clarity of presentation*, we assume uniform-size (occupying unit memory) data items for now. Our techniques easily generalize to non-uniform size data items, as discussed later. We use a_{ij} to denote the access frequency with which a network node i request the data item D_j, and d_{il} to denote the weighted distance between two network nodes i and l. The *cache placement problem* is to select a *set of sets* of cache nodes $M = \{M_1, M_2, \ldots, M_p\}$, where each network node in M_j stores a copy of D_j, to minimize the total access cost

$$\tau(G, M) = \sum_{i \in V} \sum_{j=1}^{p} a_{ij} \times min_{l \in (\{S_j\} \cup M_j)} d_{il},$$

under the memory capacity constraint that

$$|\{M_j | i \in M_j\}| \leq m_i \qquad \text{for all } i \in V,$$

which means each network node i appears in at most m_i sets of M. The cache placement problem is known to be NP-hard [3].

A. Related Work

Below, we categorize the prior work by number of data items and network topology.

Single Data Item in General Graphs. The general problem of determining optimal cache placements in an arbitrary network topology has similarity to two problems in graph theory viz. facility location problem and the k-median problem. Both the problems consider only a single facility type (data item) in the network. In the facility-location problem, setting up a cache at a node incurs a certain fixed cost, and the goal is to minimize the sum of total access cost and the setting-up costs of all caches, without any constraint. On the other hand, the k-median problem minimizes the total access cost under the number constraint, i.e., that at most k nodes can be selected as caches. Both problems are NP-hard, and a number of constant-factor approximation algorithms have been developed for each of the problems [9, 10, 16], under the assumption of triangular inequality of edge costs. Without the triangular inequality assumption, either problem is as hard as approximating the set cover [16, 21] and thus, cannot be approximated better than $O(\log |V|)$ unless $\mathbf{P} = \mathbf{NP}$. Here, $|V|$ is the size of the network. In other related work, Nuggehalli et al. [22] formulate the caching problem in ad hoc networks as a special case of the connected facility location [26].

Single Data Item in Tree Topology. Several papers in the literature circumvent the hardness of the facility-location and k-median problems by assuming that the network has a tree topology [4, 19, 20, 27, 28]. In particular, Tamir [27] and Vigneron et al. [28] design optimal dynamic programming polynomial algorithms for the k-median problem in undirected and directed trees respectively. In other works, Krishnan et al. [20] consider placement of k "transparent" caches, Kalpakis et al. [19] consider a cost model involving reads, writes, and storage, and Bhattacharya et al. [4] present a distributed algorithm for sensor networks to reduce the total power expended. All of the above works consider only a single data time in a tree network topology.[1]

Multiple Data Items. Hara [12] proposes three algorithms for cache placement of multiple data items in ad hoc networks. In the first approach, each node caches the items most frequently accessed by itself; the second approach eliminates replications among *neighboring* nodes introduced by the first approach; the third approach require one or more "central" nodes to gather neighborhood information and determine caching placements. The first two approaches are largely localized, and hence, would fare very badly when the percentage of client nodes in the network is low, or the access frequencies are uniform. For the third approach, it is hard to find stable nodes to act as "central nodes" in ad hoc networks because of frequent failures and movements. All the above approaches assume the knowledge of access frequencies. In extensions of the above work, [13] and [14] generalize the above approaches for push-based systems and updates respectively. In other related works, Xu et al. [29] discuss placement of "transparent" caches in tree networks.

Our work on cache placement problem is most closely related to the works by Yin and Cao [30] and Baev and Rajaraman [3]. Yin and Cao [30] design and evaluate three simple distributed caching techniques, viz., *CacheData* which caches the passing-by data item, *CachePath* which caches the path to the nearest cache of the passing-by data item, and *HybridCache* which caches the data item if its size is small enough, else caches the path to the data. They use LRU policy for cache replacement. To the best of our knowledge, [30] is the only work that presents a distributed cache placement algorithm in a multi-hop ad hoc network with memory constraint at each node. Thus, we use the algorithms in [30] as a comparison point for our study.

Baev and Rajaraman [3] design a 20.5-approximation algorithm for the cache placement problem with uniform-size data items. For the non-uniform size data items, they show that there is no polynomial-time approximation unless $\mathbf{P} = \mathbf{NP}$. They circumvent the non-approximability by increasing the given node memory capacities by the size of the largest data item, and generalize their 20.5-approximation algorithm.

[1][20] formulates the problem in general graphs, but designs algorithms for tree topologies with single server.

However, their approach (as noted by themselves) is not amenable to an efficient distributed implementation.

Our Work. In this article, we circumvent the non-approximability of the cache placement problem by choosing to maximize the benefit (*reduction* in total access cost) instead of minimizing the total access cost. In particular, we design a simple centralized algorithm that delivers a solution whose benefit is at least one-fourth (one-half for uniform-size data items) of the optimal benefit without using any more than the given memory capacities. To the best of our knowledge, ours and [3] are the only[2] works that present approximation algorithms for the general placement of cache placement for *multiple* data items in networks with *memory constraint*. However, as noted before, [3]'s approach is not amenable to an efficient distributed implementation, while our approximation algorithm yields a natural distributed implementation which is localized and shown (using ns2 simulations) to be efficient even in mobile and dynamic traffic conditions. Moreover, as stated in Theorem 2, our approximation result is an improvement over that of [3] when optimal access cost is at least $(1/40)^{th}$ of the total access cost without the caches. Finally, unlike [3], we do not make the assumption of the cost function satisfying the triangular inequality.

III. Cache Placement Algorithms

In this section, we first present our centralized approximation algorithm. Then, we design its localized distributed implementation that performs very close to the approximation algorithm in our simulations.

A. Centralized Greedy Algorithm (CGA)

The designed centralized algorithm is essentially a greedy approach, and we refer to it as CGA (Centralized Greedy Algorithm). CGA starts with all network nodes having all empty memory pages, and then, iteratively caches data items into memory pages maximizing the benefit in a greedy manner at each step. Thus, at each step, the algorithm picks a data item D_j to cache into an empty memory page r of a network node such that the benefit of caching D_j at r is the maximum among all possible choices of D_j and r at that step. The algorithm terminates when all memory pages have been cached with data items.

For formal analysis of CGA, we first define a set of variables A_{ijk}, where selection of a variable A_{ijk} indicates that the k^{th} memory page of node i has been selected for storage of data item D_j, and reformulate the cache placement problem in terms of selection of A_{ijk} variables. Recall that for simplicity we have assumed that each data item is of unit size, and occupies one memory page of a node.

Problem Formulation using A_{ijk}. Given a network graph $G(V, E)$, where each node $i \in V$ has a memory capacity of m_i pages, and p data items $D_1, \ldots, D_p$ in the network with the respective servers $S_1, \ldots, S_p$. Select a set Γ of *variables* A_{ijk}, where $i \in V$, $1 \leq j \leq p$, $1 \leq k \leq m_i$, and if $A_{ijk} \in \Gamma$ and $A_{ij'k} \in \Gamma$ then $j = j'$, such the total access cost $\tau(G, \Gamma)$ (as defined below) is minimized. Note that the memory constraint is subsumed in the restriction on Γ that if $A_{ijk} \in \Gamma$, then $A_{ij'k} \notin \Gamma$ for any $j' \neq j$. The total access cost $\tau(G, \Gamma)$ for a selected set of variables can be easily defined as:

$$\tau(G, \Gamma) = \sum_{j=1}^{p} \sum_{i \in V} a_{ij} \times min_{l \in (\{S_j\} \cup \{i' | A_{i'jk} \in \Gamma\})} d_{il}.$$

Note that the set of cache nodes M_j that store a particular data item D_j can be easily derived from the selected set of variables Γ.

Centralized Greedy Algorithm (CGA). CGA works by iteratively selecting a variable A_{ijk} that gives the highest "benefit" at that stage. The benefit of adding a variable A_{ijk} into an already selected set of variables Γ is the reduction in the total access cost if the data item D_j is cached into the empty k^{th} memory page of the network node i. The benefit of selecting a variable is formally defined below.

Definition 1: (Benefit of selecting A_{ijk}.) Let Γ denote the set of variables that have been already selected by the centralized greedy algorithm at some stage. The *benefit of a variable* A_{ijk} ($i \in V$, $j \leq p$, $k \leq m_i$) with respect to Γ is denoted as $\beta(A_{ijk}, \Gamma)$ and is defined as follows:

$$\beta(A_{ijk}, \Gamma) = \begin{cases} \text{Undefined} & \text{if } A_{ij'k} \in \Gamma,\ j' \neq j \\ 0 & \text{if } A_{ijk'} \in \Gamma \\ \tau(G, \Gamma) - \tau(G, \Gamma \cup \{A_{ijk}\}) & \text{otherwise} \end{cases}$$

where $\tau(G, \Gamma)$ is as defined before. The first condition of the above definition stipulates that if the k^{th} memory page of the node i is not empty (i.e., has already been selected to store another data item j' due to $A_{ij'k} \in \Gamma$), then the benefit $\beta(A_{ijk}, \Gamma)$ is undefined. The second condition specifies that the benefit of a variable A_{ijk} with respect to Γ is zero if the data item D_j has already been stored at some other memory page k' of the node i. □

Algorithm 1: Centralized Greedy Algorithm (CGA)

```
BEGIN
    $\Gamma = \emptyset$;
    while (there is a variable $A_{ijk}$ with defined benefit)
        Let $A_{ijk}$ be the variable with maximum $\beta(A_{ijk}, \Gamma)$.
        $\Gamma = \Gamma \cup \{A_{ijk}\}$;
    end while;
    RETURN $\Gamma$;
END.                                                  ◇
```

The total running time of CGA is $O(p^2|V|^3\overline{m})$, where $|V|$ is the size in the network, $\overline{m}$ is the average number of memory pages in a node, and p is the total number of data items. Note that the number of iterations in the above algorithm is bounded by $|V|\overline{m}$, and at each stage we need to compute at most pV benefit values where each benefit value computation may take $O(pV)$ time.

Theorem 1: CGA (Algorithm 1) delivers a solution whose total benefit is at least half of the optimal benefit.

Proof: Let L be the total number of iterations of CGA. Note that L is equal to the total number of memory pages in the

[2] [2] presents a competitive online algorithm, but uses polylog-factor bigger memory capacity at nodes compared to the optimal.

network. Let Γ_l be the set of variables selected at the end of l^{th} iteration, and let ζ_l be the variable added to the set Γ_{l-1} in the l^{th} iteration. Let ζ_l be a variable A_{ijk} signifying that in the l^{th} iteration CGA decided to store j^{th} data item in the k^{th} memory page of the i node. Without loss of generality, we can assume that the optimal solution also stores data items in all memory pages. Now, let λ_l be the variable $A_{ij'k}$ where j' is the data item stored by the optimal solution in the k^{th} memory page of node i. By the greedy choice of ζ_l, we have

$$\beta(\zeta_l, \Gamma_{l-1}) \geq \beta(\lambda_l, \Gamma_{l-1}), \qquad \forall l \leq L. \tag{1}$$

Let O be the optimal benefit,[3] and C be the benefit of the CGA solution. Note that[4]

$$C = \sum_{l=1}^{L} \beta(\zeta_l, \Gamma_{l-1}). \tag{2}$$

Now, consider a modified network G' wherein each node i has a memory capacity of $2m_i$. We construct a cache placement solution for G' by taking a union of data items selected by CGA and data items selected in an optimal solution for each node. More formally, for each variable $\lambda_l = A_{ij'k}$ as defined above, create a variable $\lambda'_l = A_{ij'k'}$ where $k' = m_i + k$. Obviously, the benefit O' of the set of variables $\{\zeta_1, \zeta_2, \ldots, \zeta_L, \lambda'_1, \lambda'_2, \ldots, \lambda'_L\}$ in G' is greater than or equal to the optimal benefit O in G. Now, to compute O', we add the variables in the order of $\zeta_1, \zeta_2, \ldots, \zeta_L, \lambda'_1, \lambda'_2, \ldots, \lambda'_L$ and add up the benefits of each newly added variable. Let $\Gamma'_l = \{\zeta_1, \zeta_2, \ldots, \zeta_L\} \cup \{\lambda_1, \lambda_2, \ldots, \lambda_l\}$, and recall that $\Gamma_l = \{\zeta_1, \zeta_2, \ldots, \zeta_l\}$. Now, we have

$$\begin{aligned} O &\leq O' = \sum_{l=1}^{L} \beta(\zeta_l, \Gamma_{l-1}) + \sum_{l=1}^{L} \beta(\lambda'_l, \Gamma'_{l-1}) \\ &= C + \sum_{l=1}^{L} \beta(\lambda'_l, \Gamma'_{l-1}) && \text{From (2)} \\ &\leq C + \sum_{l=1}^{L} \beta(\lambda_l, \Gamma_{l-1}) && \text{Since } \lambda_l = \lambda'_l,\ \Gamma_{l-1} \subseteq \Gamma'_{l-1} \\ &\leq 2C && \text{From (1) and (2)} \end{aligned}$$

■

The following theorem follows from the above theorem and the definition of benefit, and shows that our above result is an improvement of the 20.5-approximation result of [3] when the optimal access cost is at least $(1/40)^{th}$ of the total access cost without the caches.

Theorem 2: If the access cost without the caches is less than 40 times the optimal access cost using optimal cache placement, then the total access cost of the CGA solution is less than 20.5 times the optimal access cost. □

Non-uniform Size Data Items and Set-up Costs. To handle non-uniform size data items, CGA continues to select data items in the order of their benefit per unit size until each node's memory is *exceeded* by the last data item cached. At the end of the above process, the CGA picks the better of the following two feasible solutions: (i) Each node caches only its last data item, (ii) Each node caches all the selected data items except the last. For (i) to be feasible, we assume that size of the largest data item in the system is less than the memory capacity of any node. It can be shown that above process yields a solution whose benefit is at least 1/4 of the optimal benefit. Our techniques can also be easily generalized to incorporate set-up costs of placing a cache at a node, by extending the benefit function appropriately. For the rest of the article, we assume arbitrary size data items.

B. Distributed Greedy Algorithm (DGA)

In this subsection, we describe a localized distributed implementation of CGA. We refer to the designed distributed implementation as DGA (Distributed Greedy Algorithm). The advantage of DGA is that it adapts to dynamic traffic conditions, and can be easily implemented in an operational (possibly, mobile) network with low communication overheads. While we cannot prove any bound on the quality of the solution produced by DGA, we show through extensive simulations that the performance (in terms of the quality of the solution delivered) of the DGA is very close to that of the CGA. The DGA is formed of two important components – nearest-cache tables and localized caching policy – as described below.

Nearest-cache Tables. For each network node, we maintain the *nearest* node (including itself) that has a copy of the data item D_j for each data item D_j in the network. More specifically, each node i in the network maintains a *nearest-cache* table, where an entry in the nearest-cache table is of the form (D_j, N_j) where N_j is the closest node that has a copy of D_j. Note that if i is the server of D_j or has cached D_j, then N_j is i. In addition, if a node i has cached D_j, then it also maintains an entry (D_j, N_j^2), where N_j^2 is the *second-nearest cache*, i.e., the closest node (other than i itself) that has a copy of D_j. The second-nearest cache information is helpful when node i decides to remove the cached item D_j. Note that if i is the server of D_j, then N_j is i. The above information is in addition to any information (such as routing tables) maintained by the underlying routing protocol. The nearest-cache tables at network nodes in the network are maintained as follows in response to cache placement changes.

<u>Addition of a Cache.</u> When a node i caches a data item D_j, it broadcasts an `AddCache` message to all of its neighbors. The `AddCache` message contains the tuple (i, D_j) signifying the ID of the originating node and the ID of the newly cached data item. Consider a node l that receives the `AddCache` message (i, D_j). Let (D_j, N_j) be the nearest-cache table entry at node l signifying that N_j is the cache node currently closest to l that has the data item D_j. If $d_{li} < d_{lN_j}$,[5] then the nearest-cache table entry (D_j, N_j) is updated to (D_j, i), and the `AddCache` message is forwarded by l to all of its neighbors. Otherwise, the node l sends the `AddCache` message to the single node N_j (which could be itself) so that N_j can possibly

[3]Note that a solution with optimal benefit also has optimal access cost.

[4]Note that $O \neq \sum_{l=1}^{L} \beta(\lambda_l, \Gamma_{l-1})$. Also, in spite of (2), the benefit value C is actually independent of the order in which ζ_l are selected.

[5]The distance values are assumed to be available from the underlying routing protocol.

update information about its second-nearest cache. The above process maintains correctness of nearest-cache entries in a static network with bounded communication delays because of the following fact. Any node whose nearest-cache table entry *needs* to change in response to addition of a cache at node i is guaranteed to have a path $\mathcal{P}$ to i such that every intermediate node on $\mathcal{P}$ will need to change its nearest-cache table entry (and hence, forward the `AddCache` message). In addition, the second-nearest cache entries are also correctly maintained because of the following observation. If node i_1 has cached the data item D_j and the second-nearest cache to i_1 is i_2, then there exist two *neighboring* nodes i_3 and i_4 (not necessarily different from i_1 or i_2) on the shortest path from i_1 to i_2 such that the nearest-cache node of i_3 is i_1 and of i_4 is i_2.

Deletion of a Cache. To efficiently maintain the nearest-cache tables in response to deletion of a cache, we maintain a *cache list* C_j for each data item D_j at its server S_j. The cache list C_j contains the set of nodes (including S_j) that have cached D_j. To keep the cache list C_j up to date, the server S_j is informed whenever the data time D_j is cached at or removed from a node. Note that the cache list C_j is almost essential for the server S_j to efficiently update D_j at the cache nodes. Now, when a node i removes a data item D_j from its local cache, it first requests C_j from the server S_j. Then, the node i broadcasts a `DeleteCache` message with the information (i, D_j, C_j) to all of its neighbors. Consider a node l that receives the `DeleteCache` message and let (D_j, N_j) be its nearest-cache table entry. If $N_j = i$, then the node l updates its nearest-cache entry using C_j, and forwards the `DeleteCache` message to all its neighbors. Otherwise, the node l sends the `DeleteCache` message to the node N_j, based on the the argument similar to the case of `AddCache`. An alternate strategy that does not require cache lists is to assume that $C_j = \{S_j\}$; however, such a strategy will definitely lead to more access cost.

Integrated Cache-Routing Tables. Nearest-caching tables can be used in conjunction with any underlying routing protocol to reach the nearest cache node, as long as the distances to other nodes are maintained by the routing protocol (or available otherwise). If the underlying routing protocol maintains routing tables [23], then the nearest-cache tables can be integrated with the routing tables as follows. For a data item D_j, let H_j be the next node on the shortest path to N_j, the closest node storing D_j. Now, if we maintain a *cache-routing* table having entries of the form (D_j, H_j, δ_j) where δ_j is the distance to N_j, then there is no need for routing tables. However, note that maintaining cache-routing tables instead of nearest-cache tables *and* routing tables doesn't offer any clear advantage in terms of number of messages transmissions.

Mobile Networks. To handle mobility of nodes, we could maintain the integrated cache-routing tables in the similar vein as routing tables [23] are maintained in mobile ad hoc networks. Alternatively, we could have the servers periodically broadcast the latest cache lists. In our simulations, we adopted the latter strategy, since it precludes the need to broadcast `AddCache` and `DeleteCache` messages to some extent.

Localized Caching Policy. The caching policy of DGA is as follows. Each node computes benefit of data items based on its "local traffic" observed for a sufficiently long time. The *local traffic* of a node i includes its own local data requests, non-local data requests to data items cached at i, and the traffic that the node i is forwarding to other nodes in the network.

Local Benefit. We refer to the benefit computed based on node's local traffic as the *local benefit*. For each data item D_j *not* cached at node i, the node i calculates the local benefit gained by caching the item D_j, while for each data item D_j cached at node i, the node i computes the local benefit lost by removing the item. In particular, the local benefit B_{ij} of caching (or removing) D_j at node i is the reduction (or increase) in access cost given by

$$B_{ij} = t_{ij}\delta_j,$$

where t_{ij} is the access frequency observed by node i for the item D_j in its local traffic, and δ_j is the distance from i to N_j (or N_j^2) – the nearest-node other than i that has the copy of the data item D_j. Using the nearest-cache tables, each node can compute the local benefits of data items in a localized manner using only local information. Since the traffic changes dynamically (due to new cache placements), each node needs to continually recompute local benefits based on most recently observed local traffic.

Caching Policy. A node decides to cache the most beneficial (in terms of local benefit per unit size of data item) data items that can fit in its local memory. When the local cache memory of a node is full, the following cache replacement policy is used. Let $|D|$ denote the size of a data item (or a set of data items) D. If the local benefit of a newly available data item D_j is higher than the total local benefit of some set D of cached data items where $|D| > |D_j|$, then the set D is replaced by D_j. Since, adding or replacing a cache entails communication overhead (due to `AddCache` or `DeleteCache` messages), we employ a concept of *benefit threshold*. In particular, a data item is newly cached only if its local benefit is higher than the benefit threshold, and a data item replaces a set of cached data items only if the difference in their local benefits is greater than the benefit threshold.

Distributed Greedy Algorithm (DGA). The above components of nearest-cache table and cache replacement policy are combined to yield our Distributed Greedy Algorithm (DGA) for cache placement problem. In addition, the server uses the cache list to periodically update the caches in response to changes to the data at the server. The departure of DGA from CGA is primarily in its inability to gather information about all traffic (access frequencies). In addition, the inaccuracies and staleness of the nearest-cache table entries (due to message losses or arbitrary communication delays) may result in approximate local benefit values. Finally, in DGA, the placement of caches happens simultaneously at all nodes in a distributed manner, which is in contrast to the sequential manner in which the caches are selected by the CGA. However, DGA is able to cope with dynamically changing access frequencies and cache placements. As noted before, any changes in cache placements trigger updates in the nearest-cache table, which in turn affect the local benefit values. Below is a summarized description of the DGA.

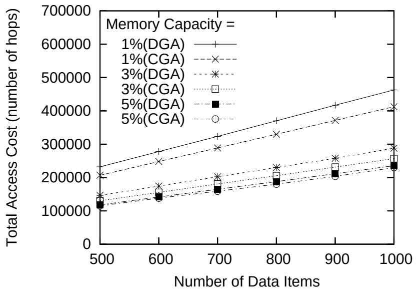
Algorithm 2: Distributed Greedy Algorithm (DGA)

```
**Setting**
  A network graph $G(V, E)$ with $p$ data items.
  Each node $i$ has a memory capacity of $m_i$ pages.
  Let $\Theta$ be the benefit threshold.
**Program of Node** $i$
**BEGIN**
  **When** a data item $D_j$ passes by:
    **if** local memory has available space and ($B_{ij} > \Theta$)
    **then** cache $D_j$
    **else** if there is a set $D$ of cached data items such that
      (local benefit of $D < B_{ij} - \Theta$) and ($|D| \geq |D_j|$),
      then replace $D$ with $D_j$.
  **When** a data item $D_j$ is added to local cache
    Notify the server of $D_j$.
    Broadcast an AddCache message containing $(i, D_j)$
  **When** a data item $D_j$ is deleted from local cache
    Get the cache list $C_j$ from the server of $D_j$
    Broadcast a DeleteCache message with $(i, D_j, C_j)$
  **On** receiving an AddCache message $(i', D_j)$
    **if** $i'$ is nearer than current nearest-cache for $D_j$,
    **then** update nearest-cache table entry and
      broadcast the AddCache message to neighbors
    **else** send the message to the nearest-cache of $i$
  **On** receiving a DeleteCache message $(i', D_j, C_j)$
    **if** $i'$ is the current nearest-cache for $D_j$
    **then** update the nearest-case of $D_j$ using $C_j$, and
      broadcast the DeleteCache message.
    **else** send the message to the nearest-cache of $i$

For **mobile networks**, instead of AddCache and
DeleteCache messages, for each data item, its server
periodically broadcasts (to the entire network) the latest
cache list.

**END.**                                                  ◊
```

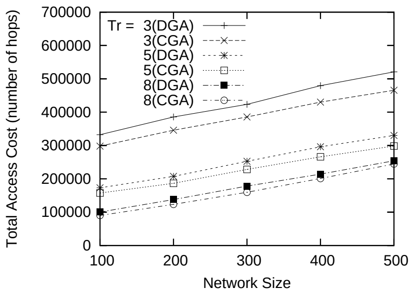
Performance Analysis. Note that the performance guarantee of CGA (i.e., proof of Theorem 1) holds even if the CGA were to consider the memory pages in some arbitrary order and select the most beneficial caches for each one of them. Now, based on the above observation, if we assume that local benefit is reflective of the accurate benefit (i.e., if the local traffic seen by a node i is the only traffic that is affected by caching a data item at node i), then DGA also yields a solution whose benefit is one-fourth of the optimal benefit. Our simulation results in Section IV-A show that DGA and CGA indeed perform very close.

IV. Performance Evaluation

We demonstrate through simulations the performance of our designed cache placement algorithms over randomly generated network topologies. We first compare the relative quality of the solutions returned by CGA and DGA. Then, we turn our attention to application level performance in complex network settings, and evaluate our designed DGA with respect to a naive distributed algorithm and the HybridCache algorithm [30] using the ns-2 simulator [11].



(a) Varying no. of data items and memory capacity. Here, number of nodes is 500 and transmission radius is 5.



(b) Varying network size and trans. radius (T_r). Here, the number of data items is 1000 and and each node's memory capacity is 20 units.

Fig. 1. Performance comparison of CGA and DGA. Here, each data item is unit size and number of clients (for each data item) is 250.

A. CGA vs. DGA

In this subsection, we evaluate the relative performance of CGA and DGA, by comparing the benefits of the solutions delivered.

For the purposes of implementing a centralized strategy, we use our own simulator for implementation and comparison of our designed algorithms. In our simulator, DGA is implemented as a dynamically evolving process wherein initially all the memory pages are free and the nearest-cache table entries point to the corresponding servers. This initialization of nearest-cache table entries results in traffic being directed to servers, which triggers caching of data items at nodes, which in turn causes changes in the nearest-cache tables and further changes in cache placements. The process continues until convergence. To provide a semblance of an asynchronous distributed protocol, our simulation model updates routing and nearest-cache table entries in an arbitrary order across nodes.

Simulation Parameters. In our cache placement problem, the relevant parameters are: (i) number of nodes in the network, (ii) transmission radius T_r (two nodes can directly transmit with each other iff they are within T_r distance from each other), (iii) number of data items, (iv) number of clients accessing each data item, (v) memory capacity on each node. The first two parameters are related to network topology, the next two parameters are application-dependent, and the last parameter is the problem constraint (property of the nodes). Here, we assume each data item to be of unit size (one memory page). Below, we present a set of plots wherein we vary some of the above parameters, while keeping the others constant.

Varying Number of Data Items and Memory Capacity. Figure 1(a) plots the access costs for CGA and DGA against the number of data items in the network for different local memory capacities. Here, the network size is 500 nodes in a 30 × 30 area.[6] We use a transmission radius (T_r) of 5 units. The memory capacity in each node is expressed as the percentage of the number of data items in the network. We vary the number of data items from 500 to 1000, and the memory capacity of each node from 1% to 5% of the number of data items. The number of clients accessing each data items is fixed at 50% of the number of nodes in the network.

We observe that the access cost increases with the number of data items as expected. Also, as expected, we see that CGA performs slightly better since it exploits global information, but DGA performs quite close to CGA. The performance difference between the algorithms decreases with increasing memory capacity, since with increasing memory capacity both the algorithms must converge to the same solution (access cost zero) as all client nodes will eventually be able to cache all the data items they wish to access. While this degenerate situation is not reached, the trend is indeed observed.

Varying Network Size and Transmission Radius. In the next plot (Figure 1(b)), we fix the number of data items in the network to 1000 and the memory capacity of each node to 2% of data items. As before, 50% of the network nodes act as clients for each of the data item. In this plot, we vary the network size from 100 nodes to 500 nodes and transmission radius (T_r) from 3 to 8. Essentially, Figure 1(b) shows the access cost as a function of network size and transmission radius for the two algorithms. Once again, as expected CGA slightly outperforms DGA, but DGA performs very close to CGA.

B. DGA vs. HybridCache

In this subsection, we compare DGA with the HybridCache approach proposed in [30] by simulating both approaches in *ns2* [11] (version 2.27). The *ns2* simulator contains models for common ad hoc network routing protocols, IEEE Standard 802.11 MAC layer protocol, and two-ray ground reflection propagation models [7]. The DSDV routing protocol [23] is used to provide routing services. For comparison, we also implemented a *Naive* approach, wherein each node caches any passing-by data item if there is free memory space and uses LRU (least recently used) policy for replacement of caches. We start with presenting the simulation setup, and then present the simulation results in the next subsection.

B.1 Simulation Setup

In this subsection, we briefly discuss the network set up, client query model, data access pattern model, and performance metrics used for our simulations.

Network Setup. We simulated our algorithms on a network of randomly placed 100 nodes in an area of $2000 \times 500\ m^2$. Note that the nominal radio range for two directly communicating nodes in the *ns2* simulator is about 250 meters. In our simulations, we assume 1000 data items of varying sizes, two randomly placed servers S_0 and S_1 where S_0 stores the data items with even IDs and S_1 stores the the data items with odd IDs. We choose the size of a data item randomly between 100 and 1500 bytes.[7]

Client Query Model. In our simulations, each network node is a client node. Each client node in the network sends out a single stream of read-only queries. Each query is essentially a request for a data item. The time interval between two consecutive queries is known as the *query generate time* and follows exponential distribution with mean value T_{query} which we vary from 3 to 40 seconds. We do not consider values of T_{query} less than 3 secs, since they result in a query success ratio of much less than 80 % for Naive and HybridCache approaches. Here, the *query success ratio* is defined as the percentage of the queries that receive the requested data item within the *query success timeout* period. In our simulations, we use a query success timeout of 40 seconds.

The above client query model is similar to the model used in previous studies [8, 30]. However, query generation process differs slightly from the one used in [30] in how the queries are generated. In [30], if the query response is not received within the query success timeout period, then the same query is sent repeatedly until it succeeds, while on success of a query, a new query is generated (as in our model) after some random interval.[8] Our querying model is better suited (due to exact periodicity of querying) for comparative performance evaluation of various caching strategies, while the querying model of [30] depicts a more realistic model of a typical application (due to repeated querying until success).

Cache Updates. In our simulations, we do not explicitly use any updates of the caches from the servers, since they will generate additional update traffic that will have similar affect on all the three techniques compared. This will also add new metrics to evaluate such as age and freshness of day, since the accessed data could be stale because of message losses and delays. In this article, we restrict ourselves to studying query delays and related metrics.

Data Access Models. For our simulations, we use the following two patterns for modeling data access frequencies at nodes.

1) Spatial pattern. In this pattern of data access, the data access frequencies at a node depends on its geographic location in the network area such that nodes that are closely located have similar data access frequencies. More specifically, we start with laying the given 1000 data items uniformly over the network area in a grid-like manner resulting in a virtual coordinate for each

[6]Since the complexity of CGA is a high-order polynomial, the running time is quite slow. Thus, we have not been able to evaluate the performance on very large networks.

[7]The maximum data size used in [30] is 10 KBytes, which is not a practical choice due to lack of MAC layer fragmentation/reassembly mechanism in the 2.27 version of *ns2* we used.

[8]In the original simulation code of HybridCache ([30]), the time interval between two queries is actually 4 seconds plus the query generate time (which follows exponential distribution with mean value T_{query}).

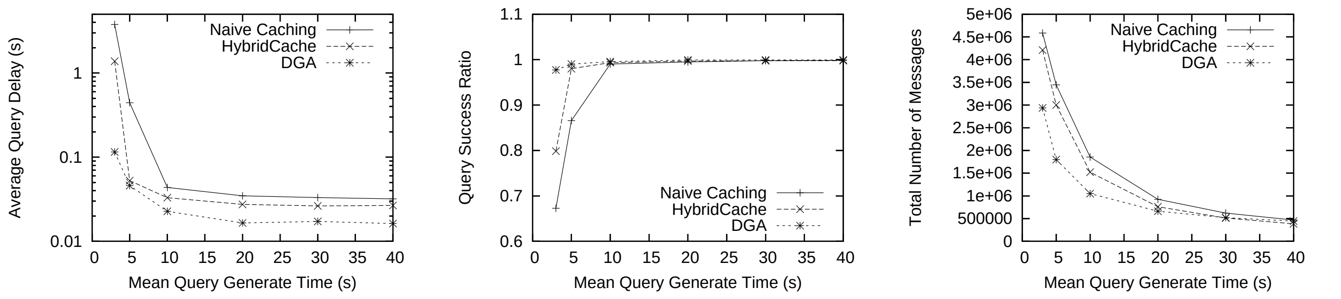


Fig. 2. Varying mean query generate time on spatial data access pattern. (a) Avg. Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

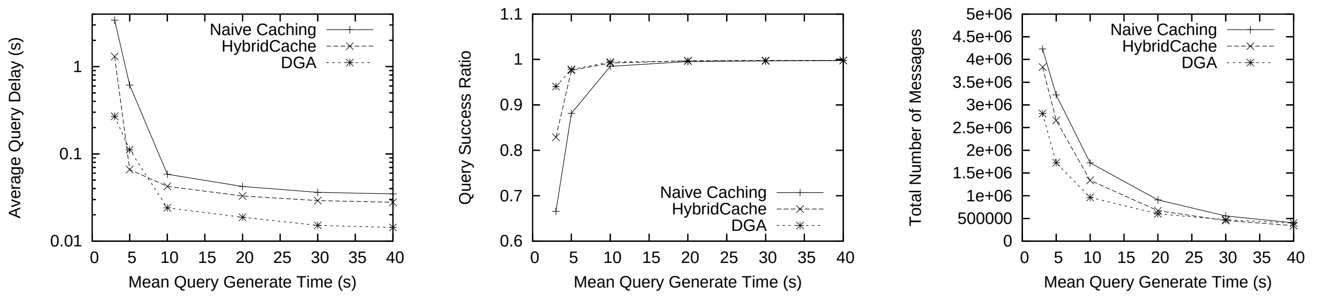


Fig. 3. Varying mean of query generate time on random data access pattern. (a) Avg. Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

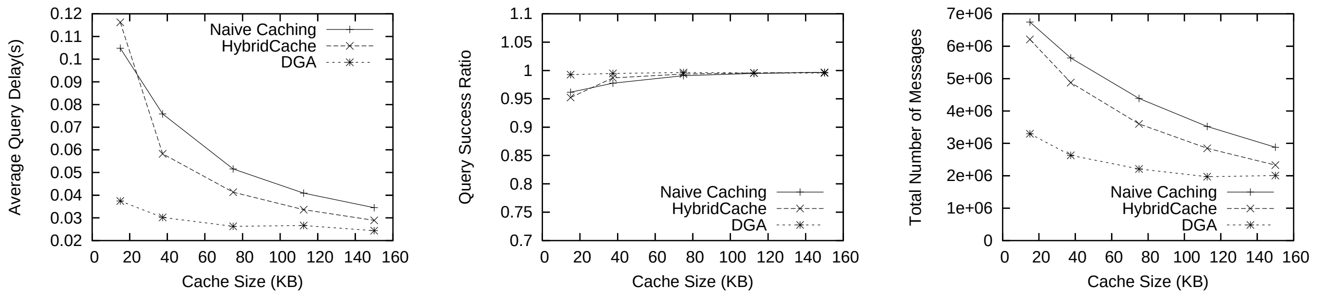


Fig. 4. Varying cache size on spatial data access pattern. Here, $T_{query} = 10$ secs. (a) Avg. Query Delay, (b) Query Success Ratio, (c) Total No. of Messages.

data item. Then, each network node accesses the 1000 data items in a *Zipf-like* distribution [5, 31], with the access frequencies of the data items ordered by the distance of the data item's virtual coordinates from the network node. More specifically, the probability of accessing (which can be mapped to access frequency) the $j^{th}(1 \leq j \leq 1000)$ closest data item is represented by $P_j = \frac{1}{j^\theta \sum_{h=1}^{1000} 1/h^\theta}$, where $0 \leq \theta \leq 1$. Here, we have assumed the number of data items to be 1000. When $\theta = 1$, the above distribution follows the strict Zipf distribution, while for $\theta = 0$, it follows the uniform distribution. As in [30], we choose θ to be 0.8 based on real web trace studies [5].

2) Random pattern. In this pattern of data access, each node uniformly accesses a predefined set of 200 data items chosen randomly from the given 1000 data items.

Performance Metrics. We measure three performance metrics for comparison of various caching strategies, viz., average query delay, total number of messages, and query success ratio. Query delay is defined as the time elapsed between query request and query response, and average query delay is the average of query delays over all queries. Total number of messages includes *all* message transmissions between neighboring nodes, including messages due to queries, maintenance of nearest-cache tables and cache-lists, and periodic broadcast of cache-lists in mobile networks. Messages to implement routing protocol are not counted, as they are the same in all three approaches compared. Query success ratio has been defined before. Each data point in our simulation results is an average over five different random network topologies, and to achieve stability in performance metrics, each of our experiments is run for sufficiently long time (20000 seconds for our experiments).

DGA Parameter Values. We now present a brief discussion on choice of values of benefit threshold and local traffic window size for DGA. For static networks, we compute local benefits based on the most recent 1000 queries. Since, the data access frequencies remains static in our experiment setting, computing local benefits based on as large a number of queries as possible is a good idea. However, we observed that most recent 1000 queries are sufficient to derive complete knowledge of local traffic. For mobile networks with spatial data access pattern, the access frequencies at a client node change with the node's location. Thus, we compute local benefits using only 50 recent queries.

Also, we chose a benefit threshold value of 0.008 when the cache size is default 75 KBytes (capable of storing 100 average sized data items), based on the typical benefit value of the 100^{th} most beneficial data item at a node. We use similar methodology for choosing benefit threshold values for other values of cache sizes.

B.2 Simulation Results

We now present simulation results comparing the three caching strategies, viz., Naive Approach, HybridCache approach of [30], and our DGA, under the random and spatial data access patterns (as defined above) and study the effect of various parameter values on the performance metrics.

Varying Mean Query Generate Time. In Figure 2, we vary the mean query generate time T_{query} in the spatial data access pattern while keeping the cache size as constant and all network nodes as client nodes. We choose the cache size to be big enough to fit about 100 average sized data items (i.e., 75 KBytes). We observe that our DGA outperforms the other two approaches in terms of all three performance metrics of query average delay, query success ratio, and total number of messages. In comparison with HybridCache strategy, our DGA has an average query delay of less than half for all parameter values, always has better query success ratio and lower message overhead. For the mean query generate time of 3 seconds, average query delay in all approaches is high, but our DGA outperforms HybridCache by a more than a factor of 10. Also, for very low mean query generate times, our DGA has a significantly better query success ratio. Figure 3 depicts similar observations for the random access data patterns, except that for mean query generate time of 5 second we have a slightly worse average query delay than that of HybridCache (but a significantly better query success ratio).

Varying Cache Memory Size. In Figure 4, we vary the local cache size of each node in the spatial data access pattern while keeping the mean query generate time T_{query} constant at 10 seconds. We vary the local cache size from 15 KBytes (capable of storing 20 data items of average size) to 150 KBytes. We observe in Figure 4 that our DGA outperforms the HybridCache approach consistently for all cache sizes and in terms of all three performance metrics. The difference in the average query delay is much more significant for lower cache size – which suggests that our DGA is very judicious in choice of data items to cache. Note that HybridCache performs even worse than the Naive Approach when each node's memory is 15 KB.

Mobile Networks. Till now, we have restricted our discussion and simulations to ad hoc networks with static nodes. Now, we present performance comparison of various caching strategies for mobile ad hoc networks, wherein the mobile nodes move based on the "random waypoint" movement model [6]. In the random waypoint movement model, initially nodes are placed randomly in the area. Each node selects a random destination and moves towards the destination with a speed selected randomly from (0 m/s, $v_{\max}$ m/s). After the node reaches its destination, it pauses for a period of time (chosen to be 300 seconds in our simulations as in [30]) and repeats the movement pattern.

In Figure 5, we compare various cache placement algorithms under the spatial data access pattern for varying mean query generate time, while keeping other parameters constant ($v_{\max} = 2$ m/s and local cache size = 75 KBytes). We observe that our DGA again outperforms HybridCache and Naive approaches in terms of query delay and query success ratio for all values of T_{query}. In particular, the average query delay of DGA is almost always better than HybridCache by up to a factor of 2, with the query success ratio of DGA being always better by a few percentages. Also, in terms of number of messages, our DGA performs only slightly worse than the other two algorithms.

In Figure 6, we compare various cache placement algorithms under the spatial data access pattern for increasing mobility, i.e. $v_{\max}$ values, while keeping other parameters constant ($T_{\text{query}} = 10$ seconds and local cache size = 75 KBytes). Again, we observe that our DGA outperforms HybridCache and Naive approaches for most mobilities. For very high mobilities ($v_{\max} \geq 15$ m/s), our DGA performs slightly worse than HybridCache in terms of average query delay, but has a *significantly* better query success ratio (78 % versus 95 %). Note that a significantly better query success ratio is more desirable than a slightly better average query delay. Again, in terms of number of messages, our DGA performs slightly worse than the other two algorithms.

Summary of Simulation Results. Our simulation results can be summarized as follows. Both the HybridCache and DGA approaches outperform the Naive approach in terms of all three performance metrics, viz., average query delay, query success ratio, and total number of messages. Our designed DGA almost always outperforms the Hybrid approach in terms of *all* performance metrics for a wide range of parameters of mean query generate time, local cache size, and mobility speed. In particular, for frequent queries or smaller cache size, the DGA approach has a significantly better average query delay and query success ratio. For very high mobility speeds, the DGA approach has a slight worse average query delay, but significantly better query success ratio. Much of the success of DGA comes from the optimized placement of caches. This not only reduces query delay, but also message transmissions, which in turn leads to less congestion and hence fewer lost messages due to collisions or buffer overflows at the network interfaces. This, in turn provides a better success ratio. This "snowballing" effect is very apparent in challenging cases such as frequent queries and small cache sizes.

V. Conclusions

We have developed a paradigm of data caching techniques to support effective data access in ad hoc networks. In particular, we have considered memory capacity constraint of the network nodes, and developed efficient algorithms to determine near-optimal cache placements to maximize reduction in overall access cost. Reduction in access cost leads to communication cost savings and hence, better bandwidth usage and energy savings. Our later simulation experience with *ns2* also shows that better bandwidth usage also in turn leads to less message losses and thus, better query success ratio.

The novel contribution in our work is the development of a 4-approximation centralized algorithm, which is naturally amenable to a localized distributed implementation. The distributed implementation uses only local knowledge of traffic. However, our simulations over a wide range of network and application parameters show that the performance of the

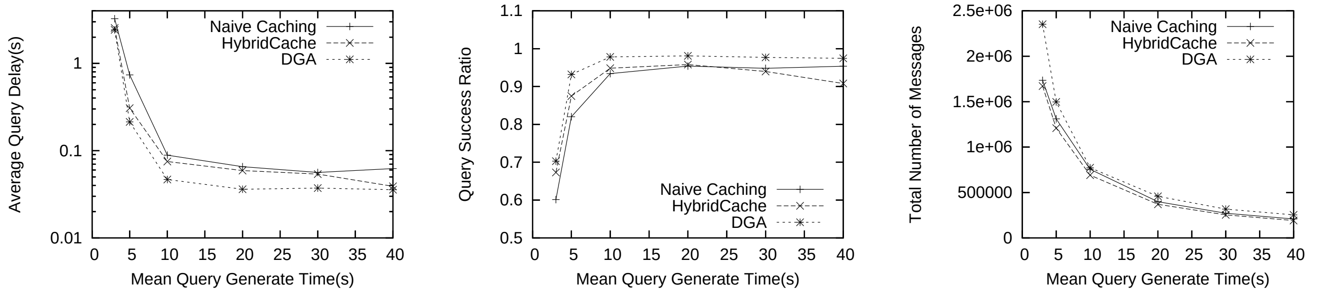


Fig. 5. Varying mean query generate time in spatial data access pattern with $v_{\max}$= 2 m/s. (a) Avg. Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

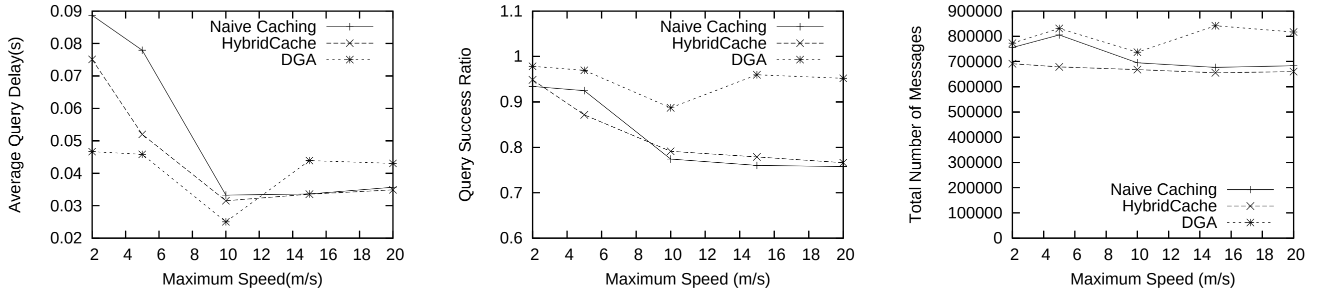


Fig. 6. Varying $v_{\max}$ in spatial data access pattern. Here, $T_{query} = 10$ seconds. (a) Avg. Query Delay, (b) Query Success Ratio, (c) Total Number of Messages.

two algorithms is quite close. We note that ours is the first work that presents a distributed implementation based on an approximation algorithm for the problem of cache placement of multiple data items under memory constraint.

We further compare our distributed algorithm with a competitive algorithm (HybridCache) presented in literature that has a similar goal. This comparison uses the *ns2* simulator with a complete wireless networking protocol stack including dynamic routing. We consider a broad range of application parameters and both stationary and mobile networks. These evaluations show that our algorithm significantly outperforms HybridCache, particularly in more challenging scenarios, such as higher query frequency and smaller memory.

REFERENCES

[1] A. Aazami, S. Ghandeharizadeh, and T. Helmi. Near optimal number of replicas for continuous media in ad-hoc networks of wireless devices. In *Intl. Workshop on Multimedia Information Systems*, 2004.

[2] B. Awerbuch, Y. Bartal, and A. Fiat. Heat & dump: Competitive distributed paging. In *FOCS*, 1993.

[3] I. Baev and R. Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *SODA*, 2001.

[4] S. Bhattacharya, H. Kim, S. Prabh, and T. Abdelzaher. Energy-conserving data placement and asynchronous multicast in wireless sensor networks. In *MobiSys*, 2003.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.

[6] J. Broch, D. Maltz, D. Johnson, Y. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *MOBICOM*, 1998.

[7] J. Broch, D. A. Maltz, D. B. Johnson, Y-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *MOBICOM*, 1998.

[8] G. Cao. Proactive power-aware cache management for mobile computing systems. *IEEE Transactions on Computer*, 51(6), 2002.

[9] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *FOCS*, 1999.

[10] F.A. Chudak and D. Shmoys. Improved approximation algorithms for a capacitated facility location problem. *Lecture Notes in Computer Science*, 1610, 1999.

[11] K. Fall and K. Varadhan (Eds.). The *ns* manual. At http://www-mash.cs.berkeley.edu/ns/.

[12] T. Hara. Effective replica allocation in ad hoc networks for improving data accessibility. In *INFOCOM*, 2001.

[13] T. Hara. Cooperative caching by mobile clients in push-based information systems. In *CIKM*, 2002.

[14] T. Hara. Replica allocation in ad hoc networks with periodic data update. In *Intl. Conf. on Mobile Data Management (MDM)*, 2002.

[15] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MOBICOM*, 2000.

[16] K. Jain and V. V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *Journal of the ACM*, 48(2), 2001.

[17] S. Jin. Replication of partitioned media streams in wireless ad hoc networks. In *ACM MULTIMEDIA*, 2004.

[18] S. Jin and L. Wang. Content and service replication strategies in multi-hop wireless mesh networks. In *MSWiM*, 2005.

[19] K. Kalpakis, K. Dasgupta, and O. Wolfson. Steiner-optimal data replication in tree networks with storage costs. In *IDEAS*, 2001.

[20] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Trans.on Networking*, 8, 2000.

[21] J.-H. Lin and J. Vitter. Approximation algorithms for geometric median problems. *Information Processing Letters*, 44(5), 1992.

[22] P. Nuggehalli, V. Srinivasan, and C. Chiasserini. Energy-efficient caching strategies in ad hoc wireless networks. In *MobiHoc*, 2003.

[23] C. Perkins and P. Bhagwat. Highly dynamic dsdv routing for mobile computers. In *SIGCOMM*, 1994.

[24] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *INFOCOM*, 2001.

[25] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric storage in sensornets with ght, a geographic hash table. *Mobile Networks and Applications*, 8(4), 2003.

[26] C. Swamy and A. Kumar. Primal-dual algorithms for connected facility location problems. In *Intl. Workshop on APPROX*, 2002.

[27] A. Tamir. An $o(pn^2)$ algorithm for p-median and related problems on tree graphs. *Operations Research Letters*, 19, 1996.

[28] A. Vigneron, L. Gao, M. J. Golin, G. F. Italiano, and B. Li. An algorithm for finding a k-median in a directed tree. *Info. Proc. Letters*, 74, 2000.

[29] J. Xu, B. Li, and D. L. Lee. Placement problems for transparent data replication proxy services. *IEEE Journal on SAC*, 20(7), 2002.

[30] L. Yin and G. Cao. Supporting cooperative caching in ad hoc networks. In *INFOCOM*, 2004.

[31] G. K. Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley, 1949.