Data Caching in Networks with Reading, Writing and Storage Costs

Himanshu Gupta
Computer Science Department
Stony Brook University
Stony Brook, NY 11790
Email: hgupta@cs.sunysb.edu

Bin Tang
Computer Science Department
Wichita State University
Wichita, KS 67260
Email: bintang@cs.wichita.edu

Abstract

Caching can significantly improve the efficiency of information access in networks by reducing the access latency and bandwidth/energy usage. However, caching in too many nodes can take up too much memory, incur extensive caching-related traffic, and hence, may even result in performance degradation. In this article, we address the problem of caching data items in networks with the objective of minimizing the overall cost under the constraint that the data item can be cached at only a limited number of network nodes. More formally, given a network, the access pattern of the data item to be shared (i.e., read and write frequencies to the data item by each node), and the storage cost (cost of caching the data item) at each node, our goal is to select at most P cache nodes so as to minimize the sum of reading, writing, and storage costs. We first consider networks with a tree topology and design an optimal dynamic programming algorithm which runs in $O(n^2P^2)$, where n is the size of the network and P is the allowed number of caches. For the general graph topology, where the problem is NP-complete, we present a centralized heuristic which is amenable to an efficient distributed implementation. Through extensive simulations in general topology graphs, we show that the centralized heuristic performs very close to the exponential optimal algorithm for small networks. In larger networks, we observe that the distributed implementation as well as the dynamic programming algorithm on an appropriately extracted tree perform quite close to the centralized heuristic.

I. Introduction

In recent years, with the advent of wireless technology and distributed file-sharing applications, the traditional client-server model has begun to lose its prominence. Instead, information sharing by spontaneously connected nodes has emerged as a new framework. In such networks, the ownership of the data files is usually not critical

*Preliminary version of the paper appeared in ICC 2006; see Section II (Related Work) for more details.

– an object (data item) does not belong to a specific node or user and hence, and is shared (i.e., read and written) by multiple nodes. For example, in an ad hoc network established for spontaneous meeting, several authors can meet and coordinate to modify the same document (e.g., an article, a powerpoint slides, or a book), in a distributed fashion. Similarly, in interconnected distributed information systems, an object (a web page, an image, a video clip, or a file) may be read and written from multiple distributed locations (network nodes. Maintaining multiple copies of an object across the network (at various locations) is an approach for improving system performance by reducing time to read or write an object.

In ad hoc networks, the problem of cache placement to optimize overall cost is further motivated by the following two 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, wherein access latency can be particularly improved by data caching. Secondly, ad hoc networks are generally resource constrained in terms of wireless bandwidth and battery energy. Data caching can help reduce communication cost, which will result in conserving battery energy and minimize bandwidth usage. However, excessive caching can result in excessive usage of memory resources, may incur excessive caching-related traffic, and thus, result in performance degradation and undesired energy consumption. Thus, in this article, we address the problem of data caching to optimize the overall access and storage cost, under the constraint that the data item can be cached at only a limited number of nodes.

More formally, we address the problem of cache placement in general multi-hop networks wherein the given data item may be read and written by multiple network nodes, and the objective is to minimize the total reading, writing, and storage cost by caching the data item at a limited number of network nodes. Here, the cost of reading the data item by a node is defined as the distance to the closest cache node, the writing cost by a node is defined as the cost of the minimum Steiner tree over the writing node and all the cache nodes times, and the storage cost at a node is the given cost of caching the data item at the node.

The rest of the paper is organized as follows. In Section II, we present our network model, formulate the data caching problem addressed in this article, and present an overview of the related work. Section III presents the optimal dynamic programming algorithm for tree topology networks. In Section IV, we design centralized and distributed heuristics for general graph networks. Simulation results are presented in Section V, and concluding remarks in Section VI.

II. **Data Caching Problem Formulation**

In this section, we present our model of the network, give a formal definition of the problem, and present a discussion on related work. We use the term *cache node* to refer to a network node that caches the data item.

Network Model and Notations. We model the network as a connected general graph, $G(V, E)$, where V is the set of nodes/vertices and E is the set of edges. We use n to denote the total number of nodes in the given network, i.e., $n = |V|$. Each edge has a nonnegative weight associated with it. There is a single data item in the network, which is to be cached at selected network nodes. For each node $i \in V$, the frequency of reading the data item is r_i, the frequency of writing the data item is w_i, and the cost of caching (i.e., storing) the data item

at node i is s_i. Let d_{ij} denote the shortest distance (minimum total weight) between any two nodes i, j, and let $d(i, M) = \min_{j \in M} d_{ij}$ be the shortest distance from i to some node in a set of nodes M. Also, let $S(X)$ be the optimal cost of a Steiner tree over the set of nodes X. Given a set of cache nodes M where the data item is cached, the total cost of reading the data item by a node i is $r_i d(i, M)$, while the cost of writing by node i is $w_i S(M \cup \{i\})$. The tree used by a writer i to write onto the set of caches is refered to as the *write-tree* for the writer i. Note that we do not assume a server for the data item in the network, since in our model, a server can be looked upon as a predetermined cache node.

Data Caching Problem. The *data caching problem* in the above network model can be defined as follows. Given a network graph $G(V, E)$ and a number P $(1 \leq P \leq n)$, select at most P cache nodes such that the total (reading, writing, and storage) cost is minimized. For a given network graph G and a set of cache nodes M, the total cost is denoted by $\tau(G, M)$ and is defined as:

$$\tau(G, M) = \sum_{i \in V} r_i d(i, M) + \sum_{i \in V} w_i S(\{i\} \cup M) + \sum_{i \in M} s_i \tag{1}$$

In the above equation, the terms on the right hand side represent total read cost, total write cost, and total storage cost respectively. Essentially, the data caching problem is to select a set of cache nodes M ($|M| \leq P$) such that the total cost $\tau(G, M)$ is minimized.

Related Work. When there are no writers and $P = n$, the data caching problem is exactly the same as well-known facility-location problem. When the number of cache nodes are constrained to be at most P and the cost is comprised only of reading and storage costs, the data caching problem is the well-known P-median problem. Both the problems (facility-location and P-median) are NP-hard, and a number of constant-factor approximation algorithms have been developed for each of the problems [2, 3, 8], under the assumption that the edge costs in the graph satisfy the triangular inequality. Without the triangular inequality assumption, either problem is as hard as approximating the set cover [8, 13], and therefore cannot be approximated better than $O(\log n)$ unless $\mathbf{NP} \subseteq \tilde{\mathbf{P}}$. Several papers in the literature circumvent the hardness of the facility-location and P-median problems by assuming that the network has a tree topology [11, 12, 16]. In particular, the best known algorithm for solving P-median in trees is by Tamir [16], who gives an $O(Pn^2)$ time dynamic programming algorithm. In this article, we essentially generalize Tamir's algorithm for our data caching problem in trees, and also present centralized and distributed heuristics for general graphs.

In a recent work, Wolfson and Milo [18] consider a simpler version of our data caching problem, wherein there are no storage costs, and the write policy uses the minimum spanning tree over the *distance graph* of the cache nodes. They design optimal algorithms for trees, rings, and complete graphs. In addition, in [14], the authors present an adaptive algorithm for replication of a data item; however, their formulation consider only read and write costs without any constraint on the total number of cache nodes. Similarly, writing and storage costs are not considered in the related proxy server placement problem [9, 15].

In the most related work, Kalpakis et al. [10] consider the problem of finding a Steiner-optimal P-replica set in a tree topology in order to minimize the sum of reading, writing, and storing costs. They developed a very complicated (more than 20 pages of case analysis) optimal dynamic programming algorithm that runs in

$O(n^6P^2)$ time and finds a Steiner-optimal replica set of size *exactly* P in tree topologies. In our understanding, their work gives a $O(n^6P^3)$-time algorithm for finding a Steiner-optimal replica set of size *at most* P in trees. In this article, we essentially address the same problem and design a much simpler dynamic programming optimal algorithm that runs in $O(n^2P^2)$ time and finds an optimal set of caches of size at most P. In addition, we design centralized and distributed heuristics to solve the problem in general graph topologies, and show through extensive simulations that our proposed heuristics perform well in practice. In the preliminary version [7] of this work, we proposed an $O(n^2P^3)$ dynamic programming algorithm for the data caching problem in trees with an assumption that read requests are satisfied by an "ancestor" cache node rather than the nearest cache node.[1]

III. Data Caching in Tree Topology

In this section, we study the data caching problem in special case of a tree topology, and present an optimal dynamic programming algorithm. Before we present our algorithm, we first review Tamir's dynamic programming (DP) algorithm for the P-median problem in a tree topology [16], since it forms the basis of our own proposed algorithm.

Tamir's DP Algorithm for P-Median in Trees. As mentioned before, the P-median problem is to select a set S of at most P nodes that minimizes the sum of the storage costs of nodes in S and the access costs. As in our data caching problem, the access cost is defined as the sum of the distances of each node v in the tree to the node nearest to v in S. Tamir [16] presents an $O(n^2P)$ time DP algorithm for the above. The brief description of the DP algorithm in [16] is as follows. First, [16] presents a linear algorithm to transform an arbitrary tree (rooted at some distinguished node v_1) into a full binary tree, wherein each node either has two children or is a leaf. The transformation guarantees that solving the problem on the original tree is equivalent to solving it on the transformed full binary tree. Let $T = (V, E)$ be the resulting binary tree, where $V = \{v_1, ..., v_n\}$. For each node $v_j \in V$, the subtree rooted at v_j is denoted as T_j, and the set of nodes in T_j is denoted as V_j. Then, for each node v_j in V, [16] computes and sorts the distances from v_j to all nodes in V, and denotes the sequence as $L = \{l_j^1, ..., l_j^n\}$,[2] where $l_j^i \leq l_j^{i+1}$ and $l_j^1 = 0$. The node corresponding to l_j^i is denoted as v_j^i. Based on the above notations, [16] defines the following terms G and F, which can be computed recursively from "leaves to root" using a dynamic programming approach.

- $G(v_j, q, l_j^i)$. It is defined as the optimal value of the subproblem defined on the subtree T_j, given that a total of at least 1 and *at most* q cache nodes can be selected in T_j, and that at least one of them has to be in $\{v_j^1, v_j^2, ..., v_j^i\} \cap V_j$. In the above definition, it is implicitly assumed that there is no interaction between the nodes in T_j and the rest of nodes in T.
- $F(v_j, q, l)$. It is defined as the optimal value of the subproblem defined on the subtree T_j under the following constraints: (i) A total of *at most* q cache nodes can be selected in T_j, (ii) There are already some selected cache nodes in $T - T_j$, and the closest amongst them to v_j is at a distance of l from v_j.

[1]The assumption is not stated in the preliminary version [7], since we failed to realize it at the time of publication. This work presents a correct (i.e., without the assumption) and more efficient dynamic programming algorithm based on an entirely different technique.

[2][16] uses the notation $\{r_j^1, \ldots, r_j^n\}$ instead.

Tamir's dynamic programming (DP) algorithm starts from leaves of T, and recursively computes G and F values at each node in T in terms of the G and F values of its children. The optimal *value* of the problem is given by $\min(G(v_1, P, l_1^n), G(v_1, 0, l_1^n))$, where v_1 is the root of the tree and n is the total number of nodes in the network. The algorithm can be easily modified to select the actual set of cache nodes that yields the optimal value.

A. Generalizing Tamir's DP to Our Data Caching Problem

Our data caching problem essentially generalizes the P-median problem by including the concept of writers and writing costs in the overall cost. Below, we present our generalized DP algorithm for the data caching problem in trees. First, we start with an overview of our simplified notations. Then, we generalize the definitions of G and F from [16] for our data caching problem, and present the recursive equations for computing G and F values at each node in the tree. Finally, we will use the values of G and F to define another function $\mathcal{G}$ for each node in the network, which essentially solves our data caching problem.

Simplified Notations. Let $T(V, E)$ be a given binary tree with nonnegative edge weights. For clarity, we drop the subscript j from the notations used in [16]. In particular, we use v to represent a node in T (instead of v_j in [16]), and T_v to denote the subtree (or the set of nodes in the subtree) rooted at v. Without loss of generality, we pick some node $\mathcal{R}$ as the root of the given tree. For each non-leaf node $v \in T$, we use v_1 and v_2 to denote v's left and right children. Finally, for each node $v \in T$, we compute and sort the distances from v to all the nodes in T and denote the corresponding node sequence as $\{v^1, ..., v^n\}$, where $d_{vv^i} \leq d_{vv^{i+1}}$ for $i = 1, ..., n-1$ and $v^1 = v$.

Defining G and F using Γ. For the purposes of defining our generalized versions of G and F functions, we first define the total cost $\Gamma(T_v, M, M_o)$ in a subtree T_v due to M, a set of cache nodes inside T_v, where M_o is the set of cache nodes outside T_v. The cost $\Gamma(T_v, M, M_o)$ is defined as:

$$\Gamma(T_v, M, M_o) = \sum_{k \in T_v} r_k d(k, M \cup \ M_o) + \sum_{k \in M} s_k + \sum_{k \notin T_v} w_k S(\{v\} \cup M) + \sum_{k \in T_v} w_k S(\{k\} \cup \{v\} \cup M)$$

The above expression includes the storage costs of the set M of cache nodes inside T_v, the total reading costs of all the nodes in T_v using the cache nodes M as well as M_o, and total writing cost over the edges in T_v due to all the writers in T. For the writing cost, we assume (even if M_o is empty) that there *are* some cache nodes outside T_v, i.e, v is part of each write-tree.[3] Note that M_o can also be represented by the node in M_o that is closest to v, but we use the above notation for sake of clarity in defining G and F.

Defining $G(v, q, v^i)(1 \leq q \leq |T_v|)$. We define $G(v, q, v^i)$ as the optimal cost Γ in the subtree T_v given that there are *exactly* q cache nodes in T_v and the closest to v among them is at most d_{vv^i} distance away from v. Also, the access costs are computed using only the caches inside T_v (i.e., $M_o = \{\}$). More formally,

$$G(v, q, v^i) = \min_{|M|=q, d(v,M) \leq d_{vv^i}} \Gamma(T_v, M, \{\}).$$

[3]Eventually, we will define another function $\mathcal{G}$ that computes the writing costs assuming no outside caches nodes. For clarity of presentation, we defer definition of $\mathcal{G}$.

Defining $F(v, q, v^i)(0 \leq q \leq |T_v|$. We define $F(v, q, v^i)$ as the optimal cost Γ in the subtree T_v given that there are *exactly* q cache nodes in T_v and the closest outside cache is v^i. More formally,

$$F(v, q, v^i) = \min_{|M|=q} \Gamma(T_v, M, \{v^i\}).$$

Note that $F(v, q, v^i)$ is not defined when $v^i \in T_v$, and

$$F(v, 0, v^i) = \sum_{k \in T_v} (r_k d_{kv^i} + w_k d_{kv}).$$

Recursive Equations for Computing G and F. We now define recursive equations for computing G and F at a node v in terms of the G and F at the children of v. The G and F values will be eventually used to compute the solution of our data caching problem.

G and F Values at a Leaf Node. When v is a leaf node, the value G is defined only for $q = 1$ and F is defined for $q = 0$ or 1. Also, F is not defined for $v^i = v$, i.e., $i = 1$. Now, it is easy to see that:

$$\begin{aligned} G(v, 1, v^i) &= s_v, & i = 1, ..., n \\ F(v, 0, v^i) &= r_v d_{vv^i}, & i = 2, ..., n \\ F(v, 1, v^i) &= s_v, & i = 2, ..., n \end{aligned}$$

Intuition for the Below Recursive Equations. Recall that v_1 and v_2 are used to denote the two children of v. Now, for a non-leaf node v, the cost $\Gamma(T_v, M, M_o)$ can be expressed in terms of the function Γ over T_{v_1} and T_{v_2}, the access and storage cost for node v, and the write cost over the edges (v, v_1) and (v, v_2). The exact expression for the above depends on the composition of M, i.e., whether M includes v, a node in T_{v_1}, and/or a node in T_{v_2}. Based on the above observation, the values G and F at a node v can be appropriately defined in terms of G and F values at its children v_1 and v_2, as shown in the following paragraphs.

Computing $G(v, q, v^1)$ (i.e., for $i = 1$). Here, since i = 1, the node v is also a cache node. First, when $q = 1$, we have

$$G(v, 1, v^1) = F(v_1, 0, v) + F(v_2, 0, v) + d_{vv_1} \sum_{k \in T_{v_1}} w_k + d_{vv_2} \sum_{k \in T_{v_2}} w_k.$$

Note that v is an outside cache node for the subtrees T_{v_1} and T_{v_2}. For $q > 1$, the total cost on T_v includes the storage cost on node v, the cost on the subtrees T_{v_1} and T_{v_2}, and the write cost on the edges of (v, v_1) and (v, v_2). In particular, there are three cases:

(a) There are no cache nodes in T_{v_1}, but there is at least one cache node in T_{v_2}. In this case, the edge (v, v_1) is included in the write-trees of only the writer nodes in T_{v_1}. However, the edge (v, v_2) is included in the write-trees of all writers in the network.

(b) There are no caches nodes in T_{v_2}, but there is at least one cache node in T_{v_1}. This case is similar to the above case (a).

(c) There is at least one cache node in T_{v_1} as well as T_{v_2}; this case is only possible if $q > 2$. In this case, the path (v_1, v, v_1) is included in the write-trees of each writer node in the network.

Based on the above three cases, the value $G(v, q, v^1)$ for $q > 1$ can be defined as below.

$$G(v,q,v^1) = s_v + \min \left(\begin{array}{l} F(v_1, 0, v) + F(v_2, q-1, v) + d_{vv_1} \sum_{k \in T_{v_1}} w_k + d_{vv_2} \sum_{k \in T} w_k, \\ F(v_1, q-1, v) + F(v_2, 0, v) + d_{vv_2} \sum_{k \in T_{v_2}} w_k + d_{vv_1} \sum_{k \in T} w_k, \\ \min_{1 \le q_1 < q-1} \left(F(v_1, q_1, v) + F(v_2, q-1-q_1, v) + d_{v_1 v_2} \sum_{k \in T} w_k \right) \end{array} \right)$$

Computing $G(v, q, v^i)$ for $1 < i \le n$. Here, there are two cases:

1. In the first case, at least one of the nodes in $\{v^1, v^2, \ldots, v^{i-1}\}$ is selected as a cache node. In this case, $G(v, q, v^i)$ is equal to $G(v, q, v^{i-1})$. Note that this case includes the scenario when $v^i \notin T_v$.
2. In the second case, v^i *must* be selected as a cache node. Here, there are two subcases, viz., (2-a): $v^i \in T_{v_1}$, (2-b): $v^i \in T_{v_2}$.

Let us analyze the subcase (2-a); the subcase (2-b) is similar. We denote the total cost for the subcase (2-a) as Q_1, and compute it as a minimum of two values: (i) When there are no cache nodes in T_{v_2}, (ii) when there is at least one cache node in T_{v_2}. The above case analysis yields the following expression for $G(v, q, v^i)$.

$$\begin{aligned} G(v,q,v^i) &= \min\left(G(v,q,v^{i-1}), Q_1\right) && \text{if } v^i \in T_{v_1} \\ G(v,q,v^i) &= \min\left(G(v,q,v^{i-1}), Q_2\right) && \text{if } v^i \in T_{v_2} \end{aligned}$$

where

$$\begin{aligned} Q_1 &= r_v d_{vv^i} + \min \left(\begin{array}{l} G(v_1, q, v^i) + F(v_2, 0, v^i) + d_{vv_1} \sum_{k \in T} w_k + d_{vv_2} \sum_{k \in T_{v_2}} w_k, \\ \min_{1 \le q_1 < q} \left(G(v_1, q_1, v^i) + F(v_2, q-q_1, v^i) + d_{v_1 v_2} \sum_{k \in T} w_k \right) \end{array} \right) \\ Q_2 &= r_v d_{vv^i} + \min \left(\begin{array}{l} G(v_2, q, v^i) + F(v_1, 0, v^i) + d_{vv_2} \sum_{k \in T} w_k + d_{vv_1} \sum_{k \in T_{v_1}} w_k, \\ \min_{1 \le q_1 < q} \left(G(v_2, q_1, v^i) + F(v_1, q-q_1, v^i) + d_{v_1 v_2} \sum_{k \in T} w_k \right) \end{array} \right) \end{aligned}$$

As shown in the above equation for Q_1, when there are no caches nodes in T_{v_2}, the edge (v, v_1) is part of the write-tree for all the writers in the network, and the edge (v, v_2) is part of the write-tree for all the writers in T_{v_2}. On the other hand, when there is at least one cache node in T_{v_2}, the path (v_1, v, v_2) is part of the write-tree of all writer nodes in the network. The cost Q_2 is similarly defined.

Computing $F(v, q, v^i)$. Recall that $F(v, q, v^i)$ is the optimal value of $\Gamma(T_v, M, \{v^i\})$ where M is a set of q cache nodes in T_v, and v^i is not in T_v. If M includes a cache node $u \in T_v$ such that $d_{uv} < d_{vv^i}$, then the optimal value of $\Gamma(T_v, M, \{v^i\})$ is $G(v, q, v^{i-1})$. Else, v^i is the closest cache to v (in particular, v is not a cache node), and there are the following three cases. (i) There are no caches nodes in T_{v_2}, (ii) There are no cache nodes in T_{v_1}, and (iii) There is at least one cache node in T_{v_1} as well as T_{v_2}. For the last case, note that the cache node closest to v_1 (v_2) outside of T_{v_1} (T_{v_2}) is still v^i, since M does not include any node u such that $d_{uv} < d_{vv^i}$. Also, since $q > 1$, there *must* be a cache node in either T_{v_1} or T_{v_2}. The above case analysis and observations yield the following equation for computing F.

$$F(v, q, v^i) = \min\{G(v, q, v^{i-1}), Q_3\}$$

where

$$Q_3 = r_v d_{vv^i} + \min \left(\begin{array}{l} F(v_1, q, v^i) + F(v_2, 0, v^i) + d_{vv_1} \sum_{k \in T} w_k + d_{vv_2} \sum_{k \in T_{v_2}} w_k, \\ F(v_2, q, v^i) + F(v_1, 0, v^i) + d_{vv_2} \sum_{k \in T} w_k + d_{vv_1} \sum_{k \in T_{v_1}} w_k, \\ \min_{1 \le q1 < q} \left(F(v_1, q_1, v^i) + F(v_2, q - q1, v^i) + d_{v_1 v_2} \sum_{k \in T} w_k \right) \end{array} \right)$$

Solving the Data Caching Problem. The computation of the above defined G and F values does not solve the data caching problem, since definition of Γ (and hence, G) assumes (for the purposes of write cost) that there is an ouside cache node. Thus, we now define another function $\mathcal{G}$, which is similar to the definition of G but assumes (even for writing costs) that there are no outside cache nodes. More formally, for a given node v and $1 \le i \le n$ and $1 \le q \le |T_v|$, we define $\mathcal{G}(v, q, v^i)$ as

$$\mathcal{G}(v, q, v^i) = \min_{|M|=q, d(v,M) \le d_{vv^i}} \Big(\sum_{k \in T_v} r_k d(k, M) + \sum_{k \in M} s_k + \sum_{k \notin T_v} w_k S(\{v\} \cup M) + \sum_{k \in T_v} w_k S(\{k\} \cup M \Big).$$

The function $\mathcal{G}$ at a node v can be computed in terms of G, F, and $\mathcal{G}$ values at v_1 and v_2, as shown below. The below equations are similar to the recursive equations for G, except that when all the cache nodes are in T_{v_1} (T_{v_2}), the edge (v, v_1) ((v, v_2)) is only used by the write-trees for writers *outside* T_{v_1} (T_{v_2}).

$$\begin{array}{rcll} \mathcal{G}(v, q, v^1) & = & G(v, q, v^1) & \\ \mathcal{G}(v, q, v^i) & = & \min \left(\mathcal{G}(v, q, v^{i-1}), Q_4 \right) & \text{if } \ v^i \in \ T_{v_1} \\ \mathcal{G}(v, q, v^i) & = & \min \left(\mathcal{G}(v, q, v^{i-1}), Q_5 \right) & \text{if } \ v^i \in \ T_{v_2} \end{array}$$

where

$$\begin{array}{rcl} Q_4 & = & r_v d_{vv^i} + \min \left(\begin{array}{l} \mathcal{G}(v_1, q, v^i) + F(v_2, 0, v^i) + d_{vv_1} \sum_{k \notin T_{v_1}} w_k + d_{vv_2} \sum_{k \in T_{v_2}} w_k, \\ \min_{1 \le q_1 < q} \left(G(v_1, q_1, v^i) + F(v_2, q - q_1, v^i) + d_{v_1 v_2} \sum_{k \in T} w_k \right) \end{array} \right) \\ Q_5 & = & r_v d_{vv^i} + \min \left(\begin{array}{l} \mathcal{G}(v_2, q, v^i) + F(v_1, 0, v^i) + d_{vv_2} \sum_{k \notin T_{v_2}} w_k + d_{vv_1} \sum_{k \in T_{v_1}} w_k, \\ \min_{1 \le q_1 < q} \left(G(v_2, q_1, v^i) + F(v_1, q - q_1, v^i) + d_{v_1 v_2} \sum_{k \in T} w_k \right) \end{array} \right) \end{array}$$

Data Caching Problem Solution. The solution of the data caching problem can now be computed as $\min_{1 \le q \le P} \mathcal{G}(\mathcal{R}, q, \mathcal{R}^n)$, where $\mathcal{R}$ is the root and $\mathcal{R}^n$ is the farthest node in the network from $\mathcal{R}$. Starting from the leaves towards the root, for each node v, we compute G, F, and $\mathcal{G}$ values for each q and i. Thus, there are total $3n^2P$ values to be computed. If we *precompute* ($\sum_{k \in T_v} w_k$) and ($\sum_{k \notin T_v} w_k$) terms for all v in total $O(n^2)$ time, then computation of each G or F or $\mathcal{G}$ value can be done in $O(P)$ time. Thus, the overall time complexity of our proposed dynamic programming algorithm is $O(n^2P^2)$.

IV. General Graph Topology

In this section, we address the data caching problem in a general graph topology. In a general graph, the data caching problem is NP-hard, since it reduces to the facility-location problem when the write frequencies are zero. Here, we first design a centralized greedy algorithm, and then present a distributed implementation of the centralized algorithm. We have used similar techniques in our recent work [17] on a related problem of data caching under update cost constraint. We will show through simulations that the centralized heuristic developed in this section perform close to the optimal solution in small general graph networks.

A. Centralized Greedy Algorithm

We now present a polynomial-time Centralized Greedy Algorithm for the data caching problem. We start with defining the concept of benefit of a set of nodes.

Definition 1: (Benefit of Node) Let M be the set of nodes that have been already selected as cache nodes by the Centralized Greedy Algorithm at some stage. The *benefit* of an arbitrary node A, denoted as $\beta(A, M)$, is the reduction in total cost due to selection of A as a cache node. More formally, $\beta(A, M) = \tau(G, M) - \tau(G, M \cup \{A\})$, where $\tau(G, M)$ is the total cost of selecting a set of cache nodes M in graph G, as defined in Equation 1. □

Note that since the minimum-cost Steiner tree problem is NP-hard, we adopt the 2-approximation Steiner tree algorithm [6] to compute writing costs.

Based on the above definition of benefit, our proposed Greedy Algorithm can be described as follows. Let M be the set of cache nodes selected at any given stage. Initially, M is empty. At each stage of the Greedy Algorithm, we add to M the node A that has the highest benefit with respect to M at that stage. The process continues until P caches nodes have been selected or there is no node with positive benefit. The running time of the above described algorithm is $O(Pn^5)$, since the time to compute a 2-approximation Steiner tree over a set of s nodes is $O(sn^2)$.

B. Distributed Greedy Algorithm

In this subsection, we present a distributed localized implementation of the Centralized Greedy Algorithm. To facilitate communication between nodes, we assume presence of a *coordinator* in the network. Our Distributed Greedy Algorithm consists of rounds. During each round, each non-cache node A estimates the benefit (as described in the next paragraph) of caching the data item at A. If the benefit estimate at a node A is positive and is the maximum among all its non-cache neighbors, then A decides to cache the data item. At the end of a round, the coordinator node gathers information about the cache nodes newly added. The number of cache nodes that can be further added is then broadcast by the coordinator to the entire network. The algorithm terminates, when either more than P cache nodes have already been added or no more cache nodes were added in a round.

Estimation of $\beta(A, M)$. A non-cache node A considers only its "local" traffic and estimation of distance to the nearest cache node, to estimate $\beta(A, M)$, the benefit with respect to an already selected set of cache nodes M. In particular, a node A observes its local traffic, i.e., the data access requests that A forwards to other cache nodes. Of course, the local traffic of a node includes its own data requests. We estimate the benefit of caching the data item at A as

$$\beta(A, M) = fd - s_a - d \sum_{i \in V} w_i,$$

where f is the frequency of the local data access traffic observed at A, d is the distance to the nearest cache from A (which is computed as shown in the next paragraph), s_a is the storage cost at A, and w_i is the write frequency at a node i in the network. In the above equation, we have estimated the increase in total writing cost due to caching at A as $d \sum_{i \in V} w_i$. The local traffic f can be computed if we let the normal network traffic (using only the already selected cache nodes in previous rounds) run for some time between successive rounds.

Estimation of d – the distance to the nearest cache from A. Let A be a non-cache node, and T_A be the shortest path tree from the coordinator to the set of communication neighbors of A. Let $C \in M$ be the cache node in T_A that is closest to A. In the above Distributed Greedy Algorithm, we estimate d to be $d(A, C)$, the distance from A to C. The value $d(A, C)$ can be computed in a distributed manner at the start of each round as follows. As mentioned before, the coordinator initiates a new round by broadcasting a packet containing the remaining number constraint to the entire network. If we append to this packet all the cache nodes encountered on the way, then each node should get the set of cache nodes on the shortest path from the server to itself. Now, to compute $d(A, C)$, each node only needs to exchange the above information with all its immediate neighbors.

V. **Performance Results**

In this section, we evaluate the relative performances of the various cache placement algorithms proposed in our article.

Experiment Setup. We use a network of 50 to 400 nodes placed randomly in a square region of size 30×30. We consider unit-disk graphs wherein two nodes can communicate with each other if the distance between them is less than a given number (called the *transmission radius*). For our simulations, we use a transmission radius of 9, which is the minimum to keep even small networks of size 50 connected. We vary various parameters such as network size, the maximum number of cache nodes P, percentage of readers and writers in the network, and the *ratio* R of average write frequency to average read frequency. Note that in practical settings we expect R to be low. The read frequency of a reader node is chosen to be a random number between 0 and 100, the write frequency of a writer node is chosen to be a random number between 0 and $100R$, and the storage cost at a node is chosen to be a random number between 0 and 100. Each data point in the graph plots is an average over five different random graph topologies. In our simulations, we compare the performance of various data caching placement algorithms, viz., Centralized Greedy Algorithm, Distributed Greedy Algorithm, and Dynamic Programming Algorithm (DP) on the spanning tree with near-minimum stretch factor (as described below).

Computing a Spanning Tree with Near-Minimum Stretch Factor. Before presenting the algorithm from [1] for constructing a spanning tree with near-optimal stretch factor, let us first define *stretch factor*. Consider a graph $G = (V, E)$; the *stretch factor* of an edge $(u, v) \in E$ in a subgraph $G'(V, E' \subset E)$ is defined as the shortest distance between u and v in G'. The *stretch factor* of the subgraph G' is defined as the maximum stretch factor over all edges in G. The minimum stretch-factor spanning tree problem is to find a spanning tree with minimum stretch factor in the given graph. The problem is known to be NP-hard [1].

We now describe the approximation algorithm due to Boksberger et al. [1] for the above problem in unit-disk graphs. We will use this algorithm to construct a near-minimum stretch-factor spanning tree, which will be input to our dynamic programming algorithm (since it runs only on tree topologies). The approximation algorithm consists of the following steps.

1) Construct a dominating set of the given unit-disk graph.
2) Connect the nodes in the dominating set that are at most three hops away. This results in a connected dominating graph.

3) Extract the Gabriel Graph (which is planar) from the above connected dominating graph.
4) Compute the dual graph of the Gabriel Graph. The *dual graph* contains a vertex for every face of the Gabriel Graph, and an edge between any two adjacent faces. The weight of the edge in the dual graph is the number of common edges of the corresponding faces in the Gabriel Graph.
5) We now associate an appropriate defined weight with each vertex in the above dual graph, and then, construct a "shortest path tree" in the above dual graph.
6) Finally, in the Gabriel Graph, we delete a common edge between any two adjacent faces that are connected in the above constructed shortest-path tree in the dual graph.

The resulting graph can be shown [1] to be a spanning tree with a stretch factor of $(OPT)^4$, where OPT is the optimal (minimum) stretch factor.[4]

Comparison with Optimal Algorithm in Small Networks. An optimal solution for the data caching problem can be computed by looking at all $O(n^P)$ subsets of nodes of size at most P, and picking the subset of nodes that gives the minimum total cost as the solution. Due to the high time complexity of the above algorithm, we choose the network size $n = 50$ and vary P from 1 to upto 6. We pick R (the ratio of average write frequency to the average read frequency) as 0.1, since it was just small enough to result in maximum number of cache nodes being selected. We observe in Figure 1 that the Centralized Greedy Algorithm performs very close to the optimal cost. Thus, in the following experiments, we use the Centralized Greedy Algorithm as a benchmark of comparison. We also observe that the DP algorithm performs only about 15% worse than the optimal algorithm.

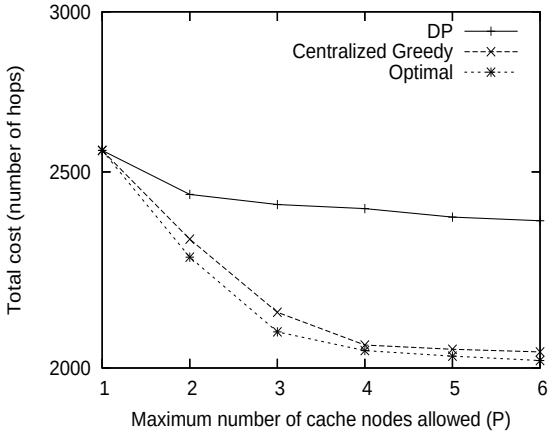

Fig. 1. Comparison of Centralized Greedy Algorithms with the optimal algorithm. Here, the network size is 50, R (the ratio of average write to average read frequency) as 0.1, and percentage of readers and writers is 50%.

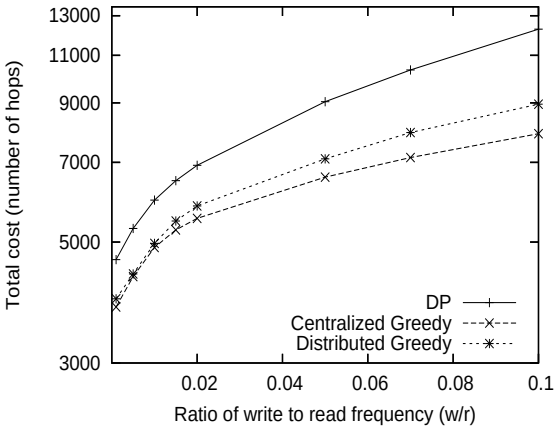

Fig. 2. Varying R, the ratio of average write to average read frequency. Here, the network size is 200, $P = 25$, percentage of readers and writers is 50.

Varying R. In this experiment, we vary R (the ratio of average read frequency to the average write frequency) from 0.001 to 0.1 in a network of size 200 with P (the maximum number of cache nodes allowed) as 25. We keep the percentage of readers and writers in the network at 50%. Figure 2 plots the total cost $\tau(G, M)$

[4]We note that the best known approximation for the minimum stretch-factor spanning tree problem is $\log n$ [4, 5]; however, we choose the technique from [1] for the sake of its relative simplicity.

corresponding to the set M of cache nodes delivered by various algorithms for given parameters. We see that the Centralized Greedy outperforms the Distributed Greedy Algorithm only by about 15%. However, when R is small, the centralized and distributed greedy algorithms perform very closely, but their relative performance becomes almost constant after $R = 0.02$. This implies that the estimation of writing costs done by the Distributed Greedy Algorithm is not as accurate as the estimation of reading costs. In contrast, we see that the DP algorithm actually performs close to the Centralized Greedy for very low values of R. For higher values of R, the DP algorithm performs worse than the Distributed Greedy. Thus, the strategy of extracting the shortest path tree rooted at an appropriate node seems effective when the writing cost is relatively very low. For $R = 0.1$, we observed that the number of cache nodes selected by any algorithm was very low (1 or 2). Thus, we did not increase the value of R beyond 0.1. Based on Figure 2, we fix R as 0.02 for all the remaining experiments, since for $R = 0.02$ the number of cache nodes is large enough (around 10) and the relative performance observed at $R = 0.02$ is representative of the general trend.

Varying Network Size. In Figure 3, we vary the network size from 100 to 400 and plot $\tau(G, M)$ corresponding to the solution M delivered by various algorithms. As suggested before, we fix $P = 25$ and $R = 0.02$. Also, the percentage of readers and writers in the network is kept as 50%. In Figure 3, we can see that the Centralized Greedy Algorithm and the Distributed Greedy Algorithm perform quite closely; both perform better than the DP algorithm. More importantly, we observe that the relative performance of the various algorithms remains relatively stable, and hence, in all other simulations, we fix the network size to be 200.

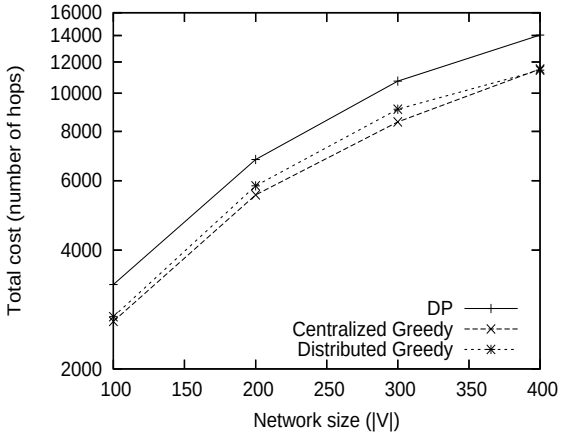

Fig. 3. Varying network size. Here, $P = 25$, R (the ratio of average write to average read frequency) is 0.02, and percentage of readers and writers is 50.

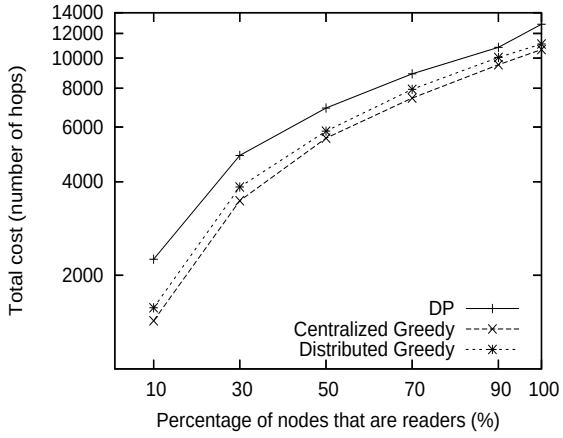

Fig. 4. Varying percentage of reader nodes in the network. Here, the network size is 200, $P = 25$, $R = 0.02$, and the percentage of writer nodes is 50%.

Varying Percentage of Readers and Writers. In Figure 4 and Figure 5, we vary the percentage of reader and writer nodes respectively in the network and plot the values of $\tau(G, M)$ for the solution delivered by various algorithms. As suggested in previous paragraphs, we fix R as 0.02 and the network size as 200. In addition, we use P as 25. In Figure 4, we vary the percentage of reader nodes from 10 to 100%, while keeping the percentage of writer nodes fixed at 50%. Similarly, in Figure 5, we vary the percentage of writer nodes from 0 to 100%, while keeping the percentage of reader nodes fixed at 50%. We observe that the relative performance

of the various algorithms remains largely unchanged with the change in percentages of readers or writers. In generally, we see the performance gap between various algorithm to be limited by 10-15%.

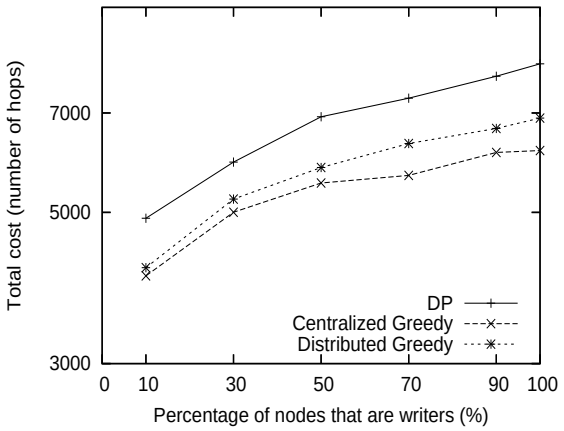

Fig. 5. Varying percentage of writer nodes in the network. Here, the network size is 200, $P = 25$, $R = 0.02$, and percentage of reader nodes is 50%.

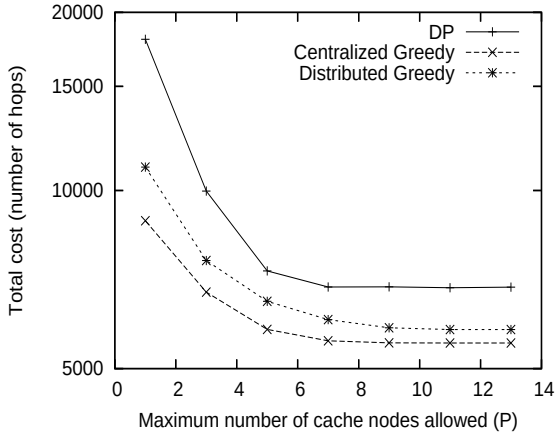

Fig. 6. Varying P. Here, the network size is 200, $R = 0.02$, and percentage of readers and writers is 50%.

Varying P. In Figure 6, we vary P, the maximum number of cache nodes allowed, and plot $\tau(G, M)$ for various algorithms. We see that with the increase in P, the relative performance gap between the Centralized and Distributed Greedy Algorithms reduces. After $P = 10$, the performance of the various algorithms remains unchanged since for the given parameter values all algorithms place at most 10 caches. Again, we see the performance gap between various algorithm to be limited by 10-15%.

VI. Conclusions

In this paper, we addressed the problem of selection on nodes to cache a data item in ad hoc networks, wherein multiple nodes can read or update the data items, individual nodes have storage limitations, and there is a limit on the number of nodes that can be selected to cache the data item. The objective of our problem was to minimize the sum of appropriately defined total reading cost, writing cost, and storage cost. For the above data caching problem, we designed an optimal dynamic programming algorithm for tree networks. In addition, for general network graphs, we proposed Centralized Greedy and Distributed Greedy heuristics, and evaluated the performance of our proposed algorithms through extensive simulations. We observe that the Centralized Greedy performs very close to the optimal algorithm for small networks, and for larger networks, the Distributed Greedy and the dynamic programming algorithm on an appropriately extracted tree perform very close to the Centralized Greedy.

References

[1] P. Boksberger. Minimum stretch spanning trees. http://dcg.ethz.ch/theses/ss03/minimumStretchSpanningTrees_report.pdf.

[2] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Proc. of IEEE Conference on Foundations of Computer Sciences*, pages 378–388, 1999.

[3] F. A. Chudak and D. Shmoys. Improved approximation algorithms for a capacitated facility location problem. *Lecture Notes in Computer Science*, 1610:99–131, 1999.

[4] M. Elkin, Y. Emek, D. A. Spielman, and S. Teng. Lower-stretch spanning trees. In *Proc. of the 37th ACM Symposium on Theory of Computing (STOC 2005)*.

[5] Y. Emek and D. Peleg. Approximating minimum max-stretch spanning trees on unweighted graphs. In *Proc. of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*.

[6] E. N. Gilbert and H. O. Pollak. Steiner minimal trees. *SIAM J. Appl. Math*, 16:1–29, 1968.

[7] H. Gupta and B. Tang. Data caching under number constraint. In *Proc. of ICC*, 2006.

[8] 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):274–296, 2001.

[9] S. Jamin, C. Kurc, A. R. Raz, and Y. Shavitt. Constrained mirror placement on the internet. In *Proc. of InfoCom'01*, pages 285–293, 2001.

[10] K. Kalpakis, K. Dasgupta, and O. Wolfson. Steiner-optimal data replication in tree networks with storage costs. In *Proc. of IDEAS*, pages 285–293, 2001.

[11] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Trans.on Networking*, 8:568–582, 2000.

[12] B. Li, M. J. Golin, G. F. Italiano, and X. Deng. On the optimal placement of web proxies in the internet. In *Proc. of INFOCOM*, volume 3, pages 1282–1290, 1999.

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

[14] S. Jajodia O. Wolfson and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

[15] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *Proc. of INFOCOM*, volume 3, pages 1587–1596, 2001.

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

[17] B. Tang, S. Das, and H. Gupta. Cache placement in sensor networks under update cost constraint. In *Proc. of AdHoc-Now*, 2005.

[18] O. Wolfson and A. Milo. The multicast policy and its relationship to replictated data placement. *ACM Transactions on Data Base Systems*, 16(1):181–205, 1991.