Proactive Data Dissemination to Mission Sites

Fangfei Chen*, Matthew P. Johnson[†], Amotz Bar-Noy[†], Iris Fermin[‡] and Thomas F. La Porta*

*Department of Computer Science and Engineering, The Pennsylvania State University, State College, PA, USA

[†]Department of Computer Science, The Graduate Center of the City University of New York, New York, NY, USA

[‡]Systems Engineering Associates, Bristol, UK

Abstract-In many situations it is important to deliver information to personnel as they work in the field. We consider such a specialized content distribution application in wireless mesh networks. When a new mission arrives-for example, when an alarm for a fire is reported-data is pushed to storage nodes at the mission site where it may be retrieved locally by responding personnel (e.g., police, firefighters, paramedics, government officials, and the media). It is important that information is available at low latency, when requested or *pulled* by the personnel. The total latency experienced will be a combination of the push delay (if the personnel arrive at the mission site before all the data can be pushed), and the pull delay. Each delay component will in turn be a function of 1) the hop distance traveled by the data when pushed or pulled and 2) the congestion on the links. In this paper, we define algorithms and protocols that trade-off the push and pull latencies depending on the type of application. Our goal is to choose a storage node assignment minimizing the total latency-based cost. We start with a simple model in which cost is a function of distance, and then extend the model explicitly taking congestion into account. Since the problem is NP-hard to approximate, our focus is on developing efficient algorithms and distributed protocols that can be easily deployed in wireless mesh networks. In NS2 simulations, we find that our heuristic algorithms achieve on average a cost within at most 15% of the optimum.

I. INTRODUCTION

Timely dissemination of information in wireless mesh networks required to execute missions is critical in many circumstances. In an example scenario, a search and rescue mission is commenced by personnel who make use of various kinds of information, e.g., maps, road conditions, medical records. Each class of personnel requires access to some (possibly overlapping) subset of the available information, which should be made available to them as quickly as possible upon their arrival at the mission site. The network model works as follows: 1) a mission site lies within a wireless mesh network; 2) data sources push required data into some storage nodes in the network; and 3) personnel (data users) traveling to the mission site pull the data from these storage nodes upon arrival at the mission site. Since the number and storage capacity of storage nodes are limited, intelligent decisions must be made in allocating storage space to the various personnel types. Also, in the context of wireless mesh networks, bandwidth and energy are scarce resources, so we desire our algorithms to be scalable and energy-efficient, with low overhead.

This setting differs from other content distribution environments in which data is pushed to storage nodes on a regular basis. In the applications considered here, the need for data at a specific location arises suddenly with the creation of a mission. Since we push the data only once the information requirements are clear, we can reduce latency costs and congestion by pushing only the data actually needed.

The goal of this paper is to minimize the latency and cost incurred by the ultimate recipient when retrieving the data, while respecting the limitations on storage node capacities. Note that if the users are already at the mission site when data becomes ready, the latency experienced by them is the sum of the pushing and pulling latency. If the users are far from the mission site when data becomes ready, the data may be pushed before the users arrive at the mission site. For this case, the latency to the users is only the pulling latency. We will show how our algorithms handle both these situations.

We start with a simple setting in which the push/pull cost is defined as a function of data item size and the distance between the data's source, destination, and storage nodes. A related problem, called the Data Placement Problem (DPP), is studied in [1]. The goal in one variant of that problem, which is NP-hard even to approximate, is minimizing the sum of installation (push) cost and access (pull) cost. Although our problem formulation differs in some respects, the hardness result carries over to it. Since no constant approximation algorithms are possible unless P=NP, our goal is to find efficient and distributed heuristics which perform well in general cases.

We define heuristics that allow a trade-off between push and pull costs and show that in the general case, where the number of data items is bounded by the number of nodes, we can achieve average costs within 15% of the optimum.

We then consider a more realistic setting in which both the location of the storage nodes and the congestion experienced in pushing and pulling the data contribute to the latency. We refine our model by including constraints that take congestion into account, and develop a corresponding heuristic algorithm. We show that we can effectively trade-off pull and push costs, depending on the type of mission, and that by tuning a parameter we achieve low latencies. Finally, to put the algorithms into the context of wireless mesh networks, we develop a fully distributed protocol and show that it is scalable with little overhead.

II. RELATED WORK

Our work relates to a body of work on Content Distribution Networks (CDNs) and data dissemination in sensor networks. A CDN consists of a group of servers that try to offload work from origin servers by delivering their content to other servers that are close to expected data request originators [2]. Its goal is to reduce the latency for a user requesting data, which may be formulated as an optimization problem, whether minimizing latency-based costs or maximizing the value to users of stored data. Baev et al. [1] formulated and gave approximation algorithms for a related but simpler variant of the problem we consider. A similar problem in the area of Internet data requests (specifically, using the IP Anycast protocol), in which minimizing time for "redirection" to data item was one of multiple goals, was recently studied by Alzoubi et al. [3]. Our model differs from a typical CDN in that the information needs in our system arise suddenly, and the location of the information requestors is only known once the data need arises. Moreover, we consider both the cost of pushing and pulling data, unlike CDNs in which the pull cost only is typically optimized.

In the networking literature, three canonical data dissemination approaches have appeared for sensor networks. External Storage (ES) stores all event data at an external storage point so queries for data needs to go out of the network; Local Storage (LS) stores all event information locally so queries for data have to flood the network to find the data source; and Data-Centric Storage (DCS) stores different types of data within the network at designated nodes; queries for data can then be sent directly to the node without flooding the network.

Directed diffusion [4] is an example of LS, in which a source stores data locally and sends data only when a sink has sent a query for the data. One example of DCS is Geographic Hash Table (GHT) [5], which hashes key values (such as data names) into geographic coordinates, and stores the data at the sensor node geographically nearest the hash of its key. Queries are directed to that area based on the same procedure. The solution we present is most similar to DCS but differs in that we choose storage nodes based on a cost value. The dissemination is *proactive* in that we push data to these storage nodes ahead of time, so that the user can pull the data when needed at the mission site. Our goal is to minimize the total cost of pushing and pulling and to reduce latency.

At the core of our problem is selecting the correct storage nodes. This requires assigning data items of various sizes to the best nodes. The core optimization problem here is the well studied Generalized Assignment Problem (GAP) [6], which lies in a family of related problems including Multiple Knapsack, Bin Packing and Facility Location.

A natural greedy strategy for the maximization variant of GAP [7], [8] is to take bins one at a time and solve each nearoptimally. This provides a $2+\epsilon$ -approximation when the singlebin problem is approximated within $1+\epsilon$. [9] gave LP-rounding and local-search algorithms for GAP and its generalization the Separable Assignment Problem (SAP), and was motivated by a Distributed Caching Problem (DCP). In that problem, there are data items, caches, and requests; caches have storage capacities and bandwidth limits. Therefore there are two sets of decisions: the assignment of data items copies to each cache and the assignment of each request to some cache (containing data of that type). They choose a request assignment profit based on the associated assignment cost. [1], [10] (see also [11]) studied a minimization caching problem similar to both DCP and our problem in which caches have integral (degree) bounds on the number of assigned data items and clients have numeric demands for data items. The objective is to minimize storage and access costs. They obtain an LP-rounding-based 10-approximation obeying the hard capacity constraints, but with the restriction that data items are unit-size.

Shmoys and collaborators [12], [13] gave approximation algorithms for various special cases and relaxations of GAP, including a polynomial-time (though LP-rounding-based) algorithm that either finds an assignment of cost at most Cwhile possibly violating capacity constraints by as much as the maximum data item size or determines that no such solution exists. The work [3] cited above uses this algorithm and then shifts assignments from overloaded nodes as needed. Recently, [14] gave bicriteria algorithms for a setting allowing capacity constraints to be violated slightly, optimizing both total cost and the amount of capacity violation. Given a hard constraint on either goal, the other can be taken as the optimization goal. Given a cost bound, the problem of minimizing the violation of capacities (or in terms of machine scheduling, minimizing makespan) is known to be NP-hard to approximate better than 3/2 [13]; when capacities are strictly enforced, it is NP-hard to approximate the cost minimization problem to any constant factor, as in our stated formulation.

In this work, we enforce capacities strictly, and we require efficient, combinatorial algorithms that do not require LP-rounding. We therefore develop heuristic algorithms and evaluate them experimentally through simulation.

III. BASIC PROBLEM FORMULATION

In this section we introduce the basic problem. Our goal is to minimize the push-pull cost of the data dissemination where cost is defined as a function of the size of the data and distance it must travel. We extend the basic problem to incorporate congestion costs in Section V.

The number of hops taken by the wireless mesh networks to deliver the data is a natural metric for cost. In our specific mission scenario, because we want to route to a specific area (similar to GHT), we use a geographic routing protocol, similar to GPSR [15], [16], which makes greedy choices based on geographical information, without needing to flood the network. An analytical approach was introduced in [17] to bound the number of hops for a given source-to-destination Euclidean distance, which allows us to use distance as a proxy for hops and avoid having to compute optimal hop counts.

A. The Integer Programming (IP) formulation

Let $D = \{d_i : i = 1...m\}$ be a set of data items and $S = \{s_j : j = 1...n\}$ be a set of storage nodes or *caches*. Each data item d_i has a size w_i and a source location p_i . Each storage node has a storage capacity b_j . The network is designed to benefit a group of users $U = \{u_k : k = 1...l\}$, each of which must retrieve some subset of the data items $D_k \subseteq D$ at somewhere in the mission site. Each potential placement of a data item d_i at a node s_j is associated with

some cost c_{ij} , which initially is taken to be simply a property of the pair (i, j), i.e. the sum of the cost of placing item d_i in node s_j and the costs of accessing it from there, for all users who desire it.

$$\min \sum_{i,j} c_{ij} x_{ij}$$
(1)
s.t.
$$\sum_{j} x_{ij} = 1 \qquad \forall i,$$
$$\sum_{i} w_{i} x_{ij} \leq b_{j} \qquad \forall j,$$
$$x_{ij} \in \{0,1\}$$

The objective is to minimize the total costs, consistent with the storage capacities. For each pair (i, j), the decision variable x_{ij} indicates whether this assignment is chosen. The first set of constraints indicate that each data item must be placed once. The second set of constraints indicate that each node's storage capacity is respected. If data items can be divided fractionally, then this problem can be solved optimally by Linear Programming (LP). In this case, the relaxed decision variable x_{ij} indicates the fraction of d_i that is placed in s_j .

We assume that there are enough nodes to store all items. One way of justifying this assumption in practice would be to place a cache at each data item's source location. This way, even if the storage nodes are full, the data may be cached at the source itself.

B. The cost function

The cost c_{ij} to store data item *i* on storage node *j* has two components, the push cost s_{ij} and the pull cost r_{ij} (in total, for all users requiring this item). In cases in which users are far from the mission site, and hence will not start pulling data for a much longer time period than the push time, the push cost may not be important. In this case the pull cost is prioritized. Contrariwise, if the users are at or near the storage nodes when the mission is started, then the push cost must be balanced with the pull cost. If we take priority between pushing and pulling into consideration and let the priority of pushing be α and that of pulling be $1 - \alpha$, then the cost is:

$$c_{ij} = \alpha \cdot s_{ij} + (1 - \alpha) \cdot r_{ij} \qquad 0 \le \alpha \le 1$$
(2)

C. Approximate cost

Now we approximate the number of hops by distance. Define d(x, y) to be the distance from object x to object y (where objects may be nodes, users, and/or data sources). With U_i denote the subset of users requesting data i, the cost function can be written as:

$$c_{ij} = \alpha \cdot d(i,j)w_i + (1-\alpha) \cdot \sum_{k:u_k \in U_i} d(j,k)w_k$$
(3)

If α is a constant for each provider, the cost depends on four factors: the source it belongs to, the users who request it, the storage node, and its size.

D. Hardness

A special case of this problem with (metric) access costs only (i.e., $\alpha = 1$) and unit weights was shown to be NPhard in [1]. That paper showed a non-unit weight version of their problem to be NP-hard to approximate, by proving it NP-hard to determine whether a feasible solution exists, given capacity constraints, by reduction from the Partition problem. That result does not *directly* apply here since, as just stated, we assume a feasible solution always exists. Still, their hardness argument can be adapted as follows, which indicates that no constant-approximation algorithm exists unless P=NP.

Proposition 3.1: Our problem is NP-hard to approximate.

Proof: Given is an instance of Partition, i.e., a set A of n numbers such that $\sum_{a_i \in A} = 2S$. We then construct a problem with two users and two unit-capacity storage nodes, all four objects located in the same position P; n data items, with data item i of size a_i/S whose providers are located somewhere other than P; and $\alpha = 1$. Then determining whether there exists a zero-cost solution is the same as solving the Partition problem.

Rather than seeking an optimal solution, therefore, our interest in this paper is efficient algorithms that perform well on realistic problem instances.

IV. HEURISTICS

In this section, we motivate and present a simple heuristic algorithm for the basic storage node selection problem. The costs involved in choosing a particular storage location form contours on a plane. We first discuss these contours and then turn to the algorithm itself.

A. Cost contours

As described above, there are two kinds of costs involved in the basic problem formulation, costs for pushing data from sources to storage nodes and costs for pulling from storage nodes to users. Each point in the plane will correspond to a possible cost value, representing the combined push and pull cost involved in storing a (unit-size) data item in a cache at that location. This ensemble of cost values forms a system of contours on the plane.

We draw the contour lines by Eq. 3. In the example shown in Fig. 1, we place three users and one source (with users indicated by '*' and sources by 'o'). The three users request one data item from this source, so storing this data item at different places incurs different costs which forms a set of contour lines. The inner lines of the contours always have lower costs than the outter lines. With pull and push costs weighted equally, the shape of the contour resembles a misshapen ellipse lying close to the users (see Fig. 1(a)). The contours converge to misshapen circles as the pull weight increases relative to the push weight (see Fig. 1(b)). The contours lie closer to the users even when we put slightly more weight on pushing (see Fig. 1(c)).



B. The heuristic algorithms

Motivated by the contours, we design our heuristics to work greedily and independently on each data source. Each source determines its best storage nodes, i.e., those with the lowest cost (on the innermost contour line), and attempts to push its data to these nodes in a greedy manner. Because source nodes act independently, conflicts may occur at storage nodes. To manage these conflicts, storage nodes follow a "first come, first serve" rule. If the storage node has enough space, it will accept the storage request; otherwise it rejects the request, and the source has to continue looking for a storage node greedily until it is satisfied.

One thing to consider is that several data items may share a single source, so we need to decide the order of data items to push from a source as well. The base algorithm selects data items for placement in arbitrary order. We consider two variants: *big-to-small* selects items in order of decreasing size while *small-to-big* does the reverse.

Algorithm 1 Greedy Algorithm		
1:	for each d_i on the same source in arbitrary order do	
2:	if d_i can be placed more cheaply in another node than	
	its source then	
3:	place d_i in a node j minimizing c_{ij} and decrease s_j 's	
	capacity by w_i	
4:	else	
5:	d_i stays at its source	
6:	end if	
7:	end for	

Every data source calculates a table with entries c_{ij} (see Eq. 3), which is the cost to put each of its data item *i* to storage node *j*. Each row of this table is sorted in order of increasing cost. The source starts to send requests for its data to the nodes corresponding to the first column of the table. Each node follows a "first come, first serve" rule and the source greedily continues requesting storage space until all its data are placed.

C. Performance evaluation

Here we show the evaluation results for the heuristics for the basic problem. When conflicts occur preventing the placement of data items in the same node due to memory constraints, the selection of storage nodes may not be globally optimal. That is, the first data item arriving at a node and occupying its memory might better be replaced by a more important data item to minimize the overall cost. Nonetheless, we find the performance of the heuristics to be quite good.

To test the performance of these heuristics, we conduct two series of simulations: varying problem size and varying problem hardness. For each set of parameters we test, there are 10 randomly generated mesh grids and the results presented are the average. In the first series of tests, we have a varying size of grids while in the second we have a fixed size of grids with 50 storage nodes. All storage nodes have a uniform capacity of 10 units and data item sizes are selected uniformly at random from [1, 10] units. Each source holds a fixed number of data items and each user requests a fixed number. Which user requests which data items is randomly decided. The location of each source, storage node and user is uniformly distributed in the grid. We test the performance of the heuristic algorithms mentioned above including the two variants: bigto-small and small-to-big.

We compare the performance of the heuristic to a bound we compute on the optimal performance. This bound is the optimal solution value OPT_{LP} to the LP relaxation of Problem 1, i.e., an optimal solution to the easier problem in which data items may be stored fractionally in multiple caches. Note that we do not allow such fragmenting of data items for storage, so OPT_{LP} can only be better than the true optimal value, thus giving us an easy-to-compute lower bound on the optimal cost to compare to.

In Fig. 2(a), we vary the size of the problem instance, while holding the "hardness" constant. That is, we increase the number of nodes along the X axis, holding the relationship between the number of data items and nodes fixed at 1:1 (this ratio is very high). The purpose of this experiment is to test how well the heuristics do for different sizes or complexities of problem instances. We find that the heuristic typically achieves 15% more costs than the optimal.

In Fig. 2(b), we vary the difficulty of the problem instance. That is, we increase the number of data items along the X axis, while holding the number of nodes constant. A complication here is that it may happen that the algorithm runs out of space and fails to place one or more of the data items at all. Although this sort of algorithm failure seems like a natural outcome to test for, due to NP-hardness, we cannot in general be sure that even the optimal algorithm would succeed. Therefore for this test we modify the constraints slightly, introducing the assumption that, if there is no remaining feasible node in which to place a data item, or if all such placements would be



(a) Cost versus problem size (varying # data items and # nodes)



(b) Cost versus problem hardness (varying # data items only)

Fig. 2. Performance Tests

V. CONGESTION

more expensive, then a source may hold the data item locally for direct access by the user. In this case, the placement cost is simply the pull cost experienced by that data item's user(s). Note that we do not introduce an official node co-located with the source; there is no capacity, and other providers may not place their items here.

The performance shown in Fig. 2(b) differs markedly from that in Fig. 2(a). First, when the problem is not too hard, that is, when the ratio of number of data items over number of nodes is below 1, the heuristics perform very close to the optimal. The lower the ratio, the closer to optimal the heuristics perform. For the specific ratio 1:1, as we have also seen from Fig. 2(a), the heuristics incurs about 15% more cost.

As the problem becomes harder, i.e., as it becomes more difficult to find storage nodes because of memory constraints, the heuristic performs more poorly. In the case with a ratio of about 3.3:1, the heuristic actually incurs twice the cost as the optimal.

In these results, we see that *big-to-small* outperforms the other two in most cases. This may be because placing the largest items first reduces the chance that they will be rejected from their optimal storage node and be placed in a node that incurs higher cost. Because the largest data items suffer the highest penalty for being misplaced, the big-to-small heuristic accommodates them first. However, this algorithm still does not match the optimal due to inefficiencies of packing data into storage nodes. Because data fragmentation is not allowed, in some cases space in storage nodes is wasted because no remaining data items may fit, whereas in an optimal solution some alternative packing would be used. For example, one extreme case occurs when storage nodes have capacity of 10 units and three data items of size $5 + \epsilon$, 5, and 5 are placed sequentially. The data item of size $5+\epsilon$ will occupy the lowest cost node, thus blocking the two data items of size 5 which will occupy the second best storage node. In this example, nearly half the storage space of the nearby node is wasted.

We conclude that when the problem is not too hard (there is sufficient memory for storing data items), the heuristics tend to perform well regardless of problem instance size. In this section we extend our formulation and heuristics to explicitly consider the challenge of congestion. This allows us to compare algorithms in terms of the latency experienced by users when retrieving the data.

A. Congestion models

Since latency is theoretically proportional to number of hops, minimizing the costs as discussed in Section III should tend in practice to minimize latency. Latency can also be increased, however, by congestion. For pushing data, the connection model is one-to-many: a source may several data items to their storing nodes simultaneously. Congestion is likely to happen for three reasons: 1) the sources are positioned close to one other; 2) the storage nodes positioned are close to one other; or 3) the routes between the sources and storage nodes cross and interfere with one another. For pulling data, the model is many-to-one: several users may request data from the same node. In this case, congestion may occur for three similar reasons: 1) the users are close to each other; 2) the nodes from which data is being pulled are close together; or 3) the routes between the different storage and users pairs interfere with each other.

Cause 3 in both cases may be handled by specialized routing protocols which are not addressed here. We focus on addressing cause 2: how to pick storage nodes that are close to the ultimate pulling nodes considering the congestion that may be caused during the push and pull operations. Our basic solution is to discourage sources from picking storage nodes that are within close range of other heavily used nodes.

B. New constraints to the problem

To reduce this type of congestion, we add new constraints to formulation (1), which limit the total data size in a certain area. In the following problem formulation, the third set of constraints limit the total data size within a one-hop neighborhood.

$$\min \sum_{i,j} c_{ij} x_{ij} \tag{4}$$

s.t.
$$\sum_{j} x_{ij} = 1$$
 $\forall i,$
 $\sum_{i} w_{i} x_{ij} \leq b_{j}$ $\forall j,$

$$\sum_{k \in H_j} \sum_{i} w_i x_{ik} \le e \qquad \qquad \forall j$$
$$x_{ij} \in \{0, 1\}$$

In this formulation H_j is $\{k | s_k \text{ is one hop from } s_j\}$, i.e., the indices of the nodes within one hop of s_j .

C. Virtual Occupation for the heuristics

Next, we extend our heuristics to deal with congestion. The goal of this extension is to disperse data around the field, in order to avoid congestion, so that heavily used nodes do not lie too near to one another. To discourage this, we introduce the concept of *Virtual Occupation* (VO), which works as follows. When a storage node accepts data, we record that it occupies its own memory, as usual, but also "virtually" occupies some space on its neighbors. When neighboring nodes receive requests to store data, they treat the virtually occupied space as being unavailable. This reduces the amount of data they can store, which naturally spreads data around the mission site. While this may increase the number of hops required for a user to pull data, it may reduce congestion at the storage nodes and thus reduce overall latency.

Algorithm 2, which is performed by each source, behaves as follows. When a node j stores data item d_i of size w_i , it reduces the available space in its one-hop neighbors by $p \cdot w_i$ as seen by other data sources. Note that the available memory of the one-hop neighbors is not reduced with respect to the source of data item d_i because a source will not interfere with itself, and although multiple users may request the same data item, e.g. d_i , this is less likely than different users requesting data items from different sources.

Algorithm 2 Greedy Algorithm with VO (single source)		
1:	for each d_i in order of decreasing size do	
2:	if d_i can be placed more cheaply in another node than	
	its source then	
3:	place d_i in a node j minimizing c_{ij}	
4:	increase s_j 's one-hop neighbors' VO by $p \cdot w_i$	
5:	decrease s_j 's capacity by w_i	
6:	else	
7:	d_i stays at its source	
8:	end if	
9:	end for	

Figure 3 shows an illustrative example of the use of the virtual occupation algorithm. There are four source-user pairs. The sources are located at the four corners of the field while the users are close to each other. The darker the shading of the



Fig. 3. An example of how VO effects the placement of data items



Fig. 4. Pull latency with/without VO

squares in the figure, the larger the amount of data stored in this area. Without using the virtual occupation (Fig. 3(a)), the area between the two users experiences heavy load (the darker squares). When using virtual occupation (Fig. 3(b)), this area becomes more lightly used, which will reduce the congestion among these source-user pairs.

D. Evaluation

In this subsection we evaluate the virtual occupation algorithm and its effectiveness in mitigating congestion, and evaluate the ability of the cost contours in trading off the pull and push costs when congestion is considered.

1) Virtual Occupation: To see the effect of the virtual occupation clearly, we conducted the following tests in NS2: we fix the four source-user pairs as in Fig. 3 but vary the number of requests from each user. The size of each data item is still uniformly distributed from 1 to 10. The latency results with and without VO are shown in Fig. 4. The users request a varying number of data items as fast as possible; for example, a value of 5 on the X-axis in Fig. 4 corresponds to each user requesting 5 data items. In this example we set α of Eq. 3 to 0 to see the effects of the virtual occupation algorithm when the pull costs are emphasized. We tune the parameter p (as in Algorithm 2) from 0 to 1.

We see that when the request load from each user is low, the virtual occupation technique does not improve performance because there is little congestion near the storage nodes. When the rate of requests increases, in most cases the virtual occupation technique reduces the pull latency. In some cases the reduction in latency is dramatic (40%). Even with high loads, however, in some cases (e.g., with 11 requests) the virtual occupation technique does not improve performance.



Fig. 5. Latency vs. α with/without VM

This is because the virtual occupation technique may increase the number of hops required to retrieve data to a point that the additional latency caused by the increased number of hops outweighs the benefits of reduced congestion.

2) Trade-off between pushing and pulling: As seen in equation 3, a trade-off may be made between the push and pull costs by adjusting the weight α . Similarly, putting different weights on pushing and pulling will cause different latencies. This trade-off is important when considering different application scenarios. In general, the latency T_r in retrieving data as seen by a user may be expressed as:

$$T_r = \begin{cases} T_{ps} - T_u + T_{pl} & \text{if } T_{ps} > T_u \\ T_{pl} & \text{otherwise} \end{cases}$$
(5)

where T_{ps} is the time to push the data, T_{pl} is the time to pull the data and T_u is the movement time for the user to the mission site. Note that if the movement time of the user is longer than the push time of the data, it will not experience any push delay because it cannot retrieve the data until it arrives at the mission site. In this case the latency experienced by the user is just the pull latency T_{pl} . If the user is at the mission site when the mission arrives, T_u is zero and the latency experienced by the user is a combination of the push time and pull time. If the user arrives during the time when the data is still being pushed, then the latency experienced by the user is a combination of a portion of the push cost and the full pull cost.

To evaluate the ability of our algorithms to balance this trade-off, we conducted a series of simulations in NS2 using an 802.11 network in which we use one fixed scenario as in Fig. 3 while changing α from 1 to 0, i.e., we vary the cost smoothly from depending solely on push to depending solely on pull. The number of requests from each user is 12.

The non-VO latency results are shown in Fig. 5(a). As α decreases, the push latency increases and the pull latency decreases, as expected. We also notice that the the sum of the two latencies decreases as α decreases from 1 to 0.5, and then increases. The effect of congestion is also evident in this figure. Note that the pull latency rises slightly as α is decreased from 0.1 to 0; this is a result of increased congestion at the storage nodes when there is a high number of requests.

We also evaluated the impact of the α value with VO turned on (see Fig. 5(b)). The parameter p (as in Algorithm 2) is tuned from 1 to 0. In these simulations, sources are located far from one another while the users are relatively close to one another. Therefore the primary expected cause of congestion is closely spaced storage nodes, i.e., the second congestion type discussed in Subsection V-A. When α is large (> 0.5) and the data is not pushed all the way to the mission site, the chosen storage nodes are likely to lie near the sources and there will be little congestion of this type. When α is smaller, the storage nodes will tend to lie near to the mission sites and hence close to one another, in which case this type of congestion is likelier to appear.

In this case, the virtual occupancy algorithms work to reduce the congestion, and hence latency, for both push and pull. The reduction in push latency as α decreases in Fig. 5(b) as compared to Fig. 5(a) is clear, as is the reduction in total latency (e.g., when $\alpha = 0$, the total latency is reduced from over 50 seconds in Fig 5(a) to under 40 in Fig 5(b)). There is also reduction in the pull latency (as shown in bar 12 of Figure 4), but this is not as visible in Figs. 5(b) and 5(a) due to scaling. We observe that the pull latency does not reduce much when the virtual occupation algorithm is used as α decreases beyond 0.2. This is because although the reduction in α drives data closer to the mission site, the virtual occupation algorithm is preventing the data from completely reaching the mission site, thus maintaining the lowest pull latency, unlike the case in Fig. 5(a) in which the pull latency increases slightly when $\alpha = 0.$

The the sum of push and pull latency is minimized when $\alpha = 0.5$. We draw the following conclusions. First, if a user has a longer time to travel to the mission site than the push time, i.e. $T_u > T_{ps}$, then setting $\alpha = 0$ tends to result in the lowest latency because the user only experiences pull latency. If the user is at the mission site when the mission occurs, i.e. $T_u = 0$, then a setting of $\alpha = 0.5$ results in the best latency because the user will experience the sum of T_{ps} and T_{pl} . If the user arrives at the mission site while the data is being pushed, the optimal setting of α depends on $T_{ps} - T_u$ which is then added to T_{pl} .

These observations suggest the following guidelines. If the mission is of a type in which personnel typically must travel to the mission site after a mission commences, for example, fire fighters and police responding to a fire, a setting of $\alpha = 0$ is

appropriate. If the personnel are already at a mission site when the mission commences, for example guards in a building noticing an alarm, then α should be set to 0.5.

VI. PROTOCOLS

In the following subsections we present protocols to implement the algorithms previously discussed.

A. Centralized Protocols

To implement a centralized protocol for this problem, an offline agent makes the optimization and then distributes the results to sources and users so that they can send or respond to requests to the right nodes. As noted above, computational hardness prevents us from guaranteeing an optimal solution. Beyond the time requirements, another challenge in producing a (near-)optimal solution is to do so without the benefit of global knowledge of such things as node locations and capacities, data item sizes, and source locations. If implemented in a wireless mesh network, this information must be passed among nodes and users which will incur high overhead.

B. Distributed Protocols

In the context of wireless mesh networks, we prefer a distributed protocol. In this section we define two distributed protocols that are differentiated by which nodes execute the distributed heuristics. These protocols do not require global knowledge. We assume the sources know the size of all the data items they will push, the location of storage nodes, and have a reasonable approximation of the location of their users. In the first protocol, the data sources run the heuristic and pick the candidate storage nodes. In the second, a node near the mission site called the center node (CN) runs the heuristic and directs the source on where to store its data. The CN may be a node in the center of the expected storage area or the first node to which a source sends a storage request.

1) Directed by the source: A simple protocol is shown in Fig. 6(a). The source runs the heuristic and determines the best candidate node for each of its data items. It then requests that each candidate node store its data, and if they accept the request, pushes the data items. There will be at least as many such requests as there are data items because if the storage nodes are already occupied, the provider will contact its subsequent choices for storage. When the user desires to pull data, it contacts the Provider to get an index of where all data is stored and then requests the data it requires from each storage node directly.

2) Directed by the CN: In the general case, storage nodes are likely to be near the users, e.g. when there is more weight on pulling. In this case the source-directed protocol described above will cause request and reply messages from sources to storage nodes to travel many hops. To reduce this effect, we develop a CN-directed protocol in which every source picks a CN as its delegate near the user. Users will retrieve the location index from the CN and pull desired data directly from the nodes (see Fig. 6(b)).

C. Overhead Simulation

To see how the overhead values depend on the problem instance, we conduct two experiments: increasing the number of users and increasing the number of requests from each user. In the tests, all storage nodes have a uniform capacity of 10 units and data items are randomly generated with sizes between 1 and 10 units. Each user requests a certain number of data items randomly from all the data items.

In the first test (see Fig. 7(a)), we increase the number of users with each user having a fixed number of requests. The total overheads increase nearly linearly. In the second test, we increase the number of requests from each user while holding the number of sources, users and data items constant. As we can see from Fig. 7(b), the overheads also increase linearly. When each user only requests one data item, the source-directed protocol has slightly less overhead than the CN-directed protocol.

We conclude that as the number of requests rises, either by increasing the number of users or the number of requests per user, the overhead increases linearly. In both of the two tests, the source-directed protocol is outperformed by the CNdirected protocol, again as expected.

VII. CONCLUSION

To disseminate data to mission sites for efficient access, two latency factors are taken into consideration: hop distance and congestion. In this paper we developed heuristics to minimize hop distances and use a virtual occupation technique to disperse data and thereby reduce congestion. We find that our heuristics achieve good performance in a distributed way and that latency can be reduced effectively. We also discussed how a trade-off between push and pull cost/latency can be made, so that in varying kinds of situations, users may consistently experience low data-retrieval latency. Finally, we define distributed protocols to implement these algorithms and show that they exhibit acceptable overheads even as the problem size grows.

Several open problems and extensions suggest themselves:

- We deal with congestion in the IP formulation by proxy, using the "virtual occupation" technique to discourage packing items too densely together. Given a linear (or perhaps merely convex) model of congestion costs, the congestion influence could be placed directly in the IP formulation.
- Algorithms could be tailored to more complex cost functions (as are allowed in our problem definition), such as costs depending on varying importance of missions.
- In difficult problem instances, it may be prohibitively expensive to place all data items. In some situations it may be preferable to refuse some items rather than pay this cost, thus transforming the hard placement constraint into a soft constraint, i.e. *filtering* the data.
- Finally, geometry could be invoked in new ways. If users are moving towards mission sites while data is being pushed, for example, it is more proactive if we push data on their path.



Fig. 6. Two different protocols



Fig. 7. Overheads

ACKNOWLEDGMENTS

This research was sponsored by US Army Research laboratory and the UK Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the US Government, the UK Ministry of Defence, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

REFERENCES

- I. Baev, R. Rajaraman, and C. Swamy, "Approximation algorithms for data placement problems," *SIAM Journal on Computing*, vol. 38(4), pp. 1411–1429, 2008.
- [2] B. Krishnamurthy, C. Wills, and Y. Zhang, "On the use and performance of content distribution networks," *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pp. 169–182, 2001.
- [3] H. A. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. E. van der Merwe, "Anycast cdns revisited," in *Proceedings of WWW 2008*, 2008.
- [4] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM 2000)*, pp. 56–67, 2000.
- [5] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, "GHT: a geographic hash table for data-centric storage," *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pp. 78–87, 2002.
- [6] G. T. Ross and R. M. Soland, "A branch and bound algorithm for the generalized assignment problem," *Mathematical Programming*, vol. 8-1, 1975.

- [7] R. Cohen, L. Katzir, and D. Raz, "An efficient approximation for the generalized assignment problem," *Information Processing Letters*, vol. 100(4), pp. 162–166, 2006.
- [8] B. Lehmann, D. J. Lehmann, and N. Nisan, "Combinatorial auctions with decreasing marginal utilities," in ACM Conference on Electronic Commerce, 2001, pp. 18–28.
- [9] L. Fleischer, M. X. Goemans, V. S. Mirrokni, and M. Sviridenko, "Tight approximation algorithms for maximum general assignment problems," in *Proceedings of SODA 2006*, pp. 611–620.
- [10] A. Meyerson, K. Munagala, and S. A. Plotkin, "Web caching using access statistics," in *Proceedings of SODA 2001*, pp. 354–363.
- [11] S. Guha and K. Munagala, "Improved algorithms for the data placement problem," in *Proceedings of SODA 2002*, pp. 106–107.
- [12] D. B. Shmoys and va Tardos, "An approximation algorithm for the generalized assignment problem," *Mathematical Programming*, vol. 62, 1993.
- [13] J. K. Lenstra, D. B. Shmoys, and va Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical Programming*, vol. 46, 1990.
- [14] M. H. Bateni and M. Hajiaghayi, "Assignment problem in content distribution networks: Unsplittable hard-capacitated facility location," in *Proceedings of SODA 09*, 2009.
- [15] P. Bose, P. Morin, I. Stojmenović, and J. Urrutia, "Routing with Guaranteed Delivery in Ad Hoc Wireless Networks," *Wireless Networks*, vol. 7, no. 6, pp. 609–616, 2001.
- [16] B. Karp and H. Kung, "GPSR: greedy perimeter stateless routing for wireless networks," *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MOBICOM 2000)*, pp. 243–254, 2000.
- [17] S. De, A. Caruso, T. Chaira, and S. Chessa, "Bounds on hop distance in greedy routing approach in wireless ad hoc networks," *International Journal of Wireless and Mobile Computing*, vol. 1, no. 2, pp. 131–140, 2006.