Greedy is Good: On Service Tree Placement for In-Network Stream Processing

Zoë Abrams
Department of Computer Science
Stanford University, Stanford, CA
zoea@stanford.edu

Jie Liu
Microsoft Research
Redmond, WA 98052
liuj@microsoft.com

Abstract

This paper is concerned with reducing communication costs when executing distributed user tasks in a sensor network. We take a service-oriented abstraction of sensor networks, where a user task is composed of a set of data processing modules (called services) with dependencies. Communications in sensor networks consume significant energy and introduce uncertainty in data fidelity due to high bit error rate. These constraints are abstracted as costs on the communication graph. The goal is to place the services within the sensor network so that the communication cost in performing the task is minimized. In addition, since the lifetime of a node, the quality of network links, and the composition of the service graph may change over time, the quality of the placement must be maintained in the face of these dynamics. In this paper, we take a fresh look at what is generally considered a simple but poor performance approach for service placement, namely the greedy algorithm. We prove that a modified greedy algorithm is guaranteed to have cost at most 8 times the optimum placement. In fact, the guarantee is even stronger if there is a high degree of data reduction in the service graph. The advantage of the greedy placement strategy is that when there are local changes in the service graph or when a hosting node fails, the repair only affects the placement of services that depend on the changes. Simulations suggest that in practice the greedy algorithm finds a low cost placement. Furthermore, the cost of repairing a greedy placement decreases rapidly as a function of the proximity of the services to be aggregated.

1 Introduction

The possibly massive amounts of raw data and the large-scale, distributed, resource constrained nature of sensor networks motivate in-network processing that distills sensor data within the network before sending it to information consumers. We refer to data processing modules as *services*, inspired by a service-oriented abstraction of the sensor network [10]. A natural question is where to place services to achieve good performance and to conserve resources. Communication in sensor networks usually costs significant energy, which leads us to focus on reducing the cost of communication when placing services. We use the notion of cost as a general abstraction. It captures a combination of energy, bandwidth, and reliability concerns.

Service placement is made more challenging in the presence of nodes and links that are unreliable. The network topology may change due to depleted batteries, node failure, or overcapacitated nodes. The cost of communication may also change if nodes are mobile or coupled closely with an evolving environment. In addition, the instantiation of services or the dependencies between services may change in situations such as conditional monitoring. For many applications, it is desirable to *repair* the placements in the network using local and distributed algorithms. An ideal service placement strategy should be both *optimal*, in its placement quality, and *adaptable*, to changes in network and application topology.

As shown in Figure 1, we consider a connected network of sensor nodes that are distributed in the physical world. Each node can collect and process data as well as communicate with its neighbor nodes. A node also serves as a router to relay network traffic. A user can interact with the entire network through any node by, for example, sending queries and receiving answers. The communications have energy, bandwidth and reliability constraints, which introduce a cost for using communication along an edge per unit of data sent. We omit the cost of computation within a node.

A user task is made up of services. These services form a *service composition graph* consisting of services and directed acyclic communication links that represent dependency constraints. The top of Figure 1 is an example of a service composition graph. Some services within a task must run on specific nodes. For instance, tasks that collect sensor data from a certain area must run on a node capable of performing the required sensing function. These services are said to be *anchored*. Other services are *floating* and can be placed on any node in the network. The goal is to find a *service placement* for floating nodes such that the total cost

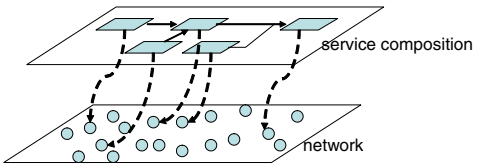


Figure 1. An illustration of service placement

of running the task is minimized. In Figure 1, the service placement is illustrated by dotted arrows from the service composition graph to the network.

This problem formulation encompasses many sensor network applications, where the data collected by the sensors is viewed as streams, and user applications are operators applied on these streams. Since sensor data is distributed, yet usually spatially clustered, processing data near where it is collected can significantly reduce the amount of data transmitted. In a sensor database (e.g. Cougar [6] or TinyDB [11]), streams of data are collected at each sensor. Database operators such as MIN, MAX, and SUM can be applied on these data streams in the network to answer user queries. Since these operators can take an arbitrary number of operands and return only one number, they can significantly reduce network traffic when placed strategically along the paths that data are routed to the end user. In macro-programming paradigms such as regiment [13] and semantic streams [20], sensor streams are hierarchically constructed to inference high-level events and to trigger reactions. These user defined functions or inference units are placed in the network for resource efficiency and timely responses. In this paper we only consider service composition graphs that form a tree, where the sensors are leaves and the end information consumer is the root. Data only flows in one direction, from the sensors to the information consumer. Even with these assumptions, the formulation still covers a wide range of sensor network applications and can be applicable to other systems beyond sensor networks, such as overlay networks, workflow management for web service, and server farms.

1.1 Related Work

The problem of placing modules within a network has recently gained increasing interest in sensor networks due to the trend of interacting with an entire sensor network as one entity through, for example, database queries [5, 16], macroprogramming [13, 2], or service composition [20, 9]. This type of interaction requires information be agreggated, synthesized, or collected *within* the network.

In the distributed computing context, the goal of operator placement is to minimize latency due to computation and communication. The *operator placement* problem, also called the *module assignment* or *task embedding* problem, for distributed tasks with precedence constraints, is one of the classical distributed computing problems [4, 14, 19, 3]. The problem shows up in many contexts such as overlay networks [15], grid computing [8], and streaming databases [12]. It has been show that the general operator placement problem is NP-hard, but polynomial time algorithms (e.g. based on dynamic programming) exist when the service graph is a tree [4].

In sensor networks, energy constraints and node reliability are often crucial. Along these lines, the work of [16, 17] considers optimum placement of filters with different selectivity rates so that the cost of execution and communication is minimized. Their model is similar to ours in its notion of filters with selectivity rates that operate over a pre-existing aggregation tree. However, their solution exploits the freedom of re-ordering operators. Our work is different in that the data flow tree represents a work flow that must be executed in a specific order and must form an exact structure.

This paper is organized as follows. In section 2 we formally define the service placement problem and outline a dynamic programming based optimum solution and an in-network relaxation heuristic. In section 3.1, we introduce the greedy placement algorithm and show its performance bound for cases where data reduction rate at each service is high. In section 3.2, we propose a modified greedy algorithm that can be used even when the data reduction rate does not satisfy the conditions in section 3.1. Section 3.3, describes how to implement the greedy algorithm in a distributed manner. We simulate and compare the performance of these algorithms in section 4.

2 Service Placement Problem

We formally define the service placement problem as follows. An underlying network is given as an undirected communication graph $G = (V, E)$, where V is the set of nodes and $E \subset V \times V$ is the set of edges connecting the nodes. Let w_e on edges $e \in E$ be the weight on the edge, i.e., the cost of communicating one unit of data across the edge. Let u and v be two nodes, $w_{(u,v)}$ is the sum of the weights on the shortest path from u to v in G.

We are also given a service graph in the form of a rooted tree $T = (O, L)$ where O is a set of services, and $L \subset O \times O$ is the directed dependency links among the services. To make the presentation clear, we call elements of the communication graph *nodes* and *edges* and elements of the service graph are called *services* and *links*. Link $l = (q, p) \in L$ represents that the outputs of service q feed into the input of service p, where p is called the parent of q and q a child of p. We denote $\mathcal{R}$ the root of the tree (i.e. the only node in the tree with no outgoing links), and C_p the (direct) children services of p. For each $(q, p) \in L$, d^o_q represents the amount of data communicated on link (q, p), and d^i_p the total amount of data fed into node p. That is $d^i_p = \sum_{q \in C_p} d^o_q$. The ratio $r_p = \frac{d^o_p}{d^i_p}$ denotes the data reduction rate at service p, defined as the data out of p divided by the data from

all children into p. We further make the following assumptions: (i) The communication graph is connected. (ii) All leaf services S and the root service $\mathcal{R}$ are anchored in the communication graph. This assumption simplifies the problem but does not exclude situations where an interior node of the service graph is anchored, since we can consider that this interior service is the root of a subproblem and then merge subproblem solutions together. Merging is possible because placements below an anchored service are independent of the placements above it. (iii) The edge weights in the communication graph satisfy triangle inequality and are symmetric (that is, $w_{(u,v)} = w_{(v,u)}$). (iv) The computation cost within a node is ignored. In particular, we assume the computational power on each node is sufficient to host the entire user task. That is, we do not constrain the capacity of the nodes, and can place multiple services on the same node.

Definition 1 ***Service Placement Problem****: Find an onto function $f : O \rightarrow V$ satisfying anchor assumptions and such that*

$$\sum_{p \in I} \sum_{q \in C_p} d_q^o \cdot w_{(f(q),f(p))} \quad (1)$$

is minimized, where I is the set of interior services in the service tree including the root, i.e. $I = O - S$.

We refer to the value of equation (1) as the *cost* of placement f, $f(p)$ the *host* of p, and f^* the function f that minimizes equation (1).

2.1 Optimum Placement

When the service graph is a tree, there exists a polynomial-time algorithm for optimum placement using dynamic programming. Define function $C(p, u)$ on services $p \in O$ and nodes $u \in V$ to be the minimum possible communication cost of routing all descendants of p to node u. For every leaf service $s \in S$ anchored at node v in the communication graph, define $C(s, v) = 0$; and $C(s, u) = \infty$ if $u \neq v$. Then, considering service p is placed at node u and given values $C(q, x)$ for $q \in C_p$ and for all $x \in V$, then the optimum placement of q is the placement for which the sum of the cost of communicating data from C_p to p plus the cost of communicating all desendant data to C_p is minimized. Note that the optimum placement for each child of p is independent of the optimum placement for the other children of p. Precisely, the function C can be computed recursively using the following equation:

$$C(p, u) = \sum_{q \in C_p} min_{x \in V}(w_{(x,u)} d_q^o + C(q, x))$$

After computing $C(p, u)$, $\forall p \in O, u \in V$, define the map g with the set of pairs $p \in O, u \in V$ used in the recursive unfolding of $C(\mathcal{R}, v_{\mathcal{R}})$, where the root service is anchored at node $v_{\mathcal{R}}$. Then, g is the optimum placement. This algorithm has running time $O(|V|^2|O|)$. Although it finds the optimum in polynomial time, it has three problems: (1) The algorithm is centralized. It requires the precise knowledge of the shortest path between all pairs of nodes. (2) The algorithm is global. A change of weights in the communication graph or a change in the topology or data rates of the service graph could potentially trigger an entire recalculation for the function C. (3) Although polynomial, the in-network cost of running the algorithm may be prohibitively large in sensor network contexts where energy is at a premium.

2.2 In-Network Relaxation

When complete knowledge of the underlying network is unknown, researches have suggested relaxation-based placement heuristics. Relaxation can be centralized by abstracting the communication costs among the nodes into values in a metric space [15]; or it can be in-network [5]. In an in-network relaxation scheme, based on an initial placement, a service hosting node locally decides whether placing the service on a neighbor node can reduce the overall cost, assuming that no other services are moving concurrently. A local migration usually will have a chain effect that triggers up-stream or down-stream service migration. The algorithm iterates through these local adjustments and tries to settle on a placement tree of minimal total cost.

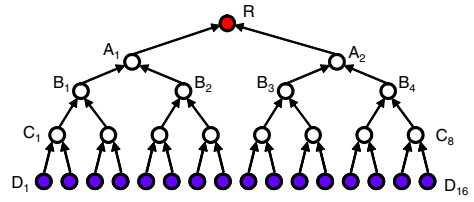
Relaxation-based algorithms are extremely simple and adaptive. They can also handle cases where the service composition graph is not a tree. However, the quality of the placement highly depends on the initial placement of the services, since it can fall into local minima when the communication costs on edges are not uniform or when the network topology is irregular (e.g. with holes).

3 A Distributed Greedy Algorithm

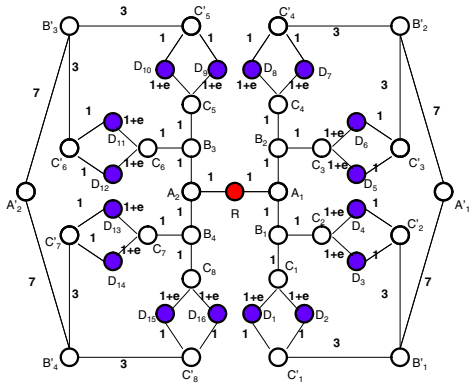
One of the key challenges of service placement in sensor networks is the ability to adapt to changes in the communication and service graphs without expending too much energy. In this section, we develop a modified greedy placement algorithm and a decentralized adaptation strategy to repair a greedy placement. The algorithm is simple and efficient and has proven guarantees on performance, regardless of the number of nodes in the network and the complexity of the service composition tree. We first describe a straightforward greedy approach. We then build upon the greedy algorithm with a simple, yet potentially crucial, modification. This section ends with a description of a local distributed repair strategy using the modified greedy algorithm.

3.1 The Greedy Algorithm

We now present an $O(|V| \cdot |O|)$ greedy heuristic that places services hierarchically. For each service p, if $f(q)$ is defined for all children $q \in C_p$, then assign p to node



(a) A service composition tree.



(b) A communication graph.

Figure 2. Greedy placement scenario

$$u = \text{argmin}_v \{ \sum_{q \in C_p} d_q^o w_{(f(q),v)} \}.$$

We say that u is the optimum median for nodes C_p. Since placement of each service only depends on the placement of its children, the greedy algorithm has the potential to be easily distributed and local changes in the service tree only affect the assignment of its ancestors. Furthermore, since each service can be definitively placed based on its children's placement, the storage and communication costs are minimized.

However, the greedy algorithm can be arbitrarily more costly than the optimum solution. Figure 2 shows a binary tree service composition tree (a) that is to be placed on a communication graph (b), where the weights of the communication graph are labeled on the edges, and e is an arbitrarily small number. The leaf nodes `Di` and the root `R` are anchored to the corresponding network node with the same label. The optimum placement routes the data "inward", i.e. each service is assigned to the node in the communication graph with the same name. The greedy algorithm, on the other hand, assigns all the nodes outward, until reaching A_i', and then routes the data all the way back inward to the root `R`.

Let $0 \leq h \leq H$ be the depth of a service in the tree where $h_{\mathtt{R}} = 0$ and H is the height of the tree. Let the amount of data sent out from each leaf be 1, and let the data reduction rate at each intermediate node be r. The amount of data leaving a node at height h is $(2r)^{H-h}$ and the number of nodes at level h is 2^h giving the following cost of the optimum assignment:

$$C^* = \sum_{h=1}^{H} 2^h (2r)^{H-h} = 2^H \sum_{h=1}^{H} (r)^{H-h}$$

while the cost of greedy assignment is

$$C = 2^H \sum_{h=1}^{H} (r)^{H-h} (2^{H-h+1} - 1)$$

because the weight of edges in the greedy placement at height h is $(2^{H-h+1} - 1)$ as opposed to 1 in the optimum. If $r \geq 1/2$, then $C/C^* \to \infty$ as $H \to \infty$, which indicates that the greedy algorithm can provide a solution with cost that is arbitrarily worse than the optimum. However, it is also interesting to observe that if $r < 1/2$, $C/C^* < \infty$. In particular, for $r = 1/4$, $C/C^* \leq 2$. That is, for this example, when the data reduction rate is $1/4$, the greedy algorithm is at most twice as bad as the optimum placement.

3.2 A Modified Greedy Algorithm

There are two factors working against each other in the Greedy Algorithm from section 3.1. First, the sub-optimum placement incurs a cost penalty due to the additional distance the data must travel. Second, the data reduction rate reduces the costliness of communicating the data. In our Modified Greedy Algorithm, the benefit from the second factor is used to mitigate any possible damage due to the first factor.

Analysis We will now give a bound on the cost of the greedy solution for arbitrary service composition trees and communication graphs. This bound will show that the greedy solution is not too far from the optimum when there is a high data reduction rate. The key observation is that *the communication cost of routing placed children $q \in C_p$ to the greedily chosen host for p is at most the communication cost of routing those same greedily placed children to the host for p in the optimum solution*. This observation is true because the host for p in the optimum is also a host option for the greedy algorithm. The cost of routing to the host for p in the optimum is broken into (a) the cost of routing the data from the greedily placed children back to the leaves, and (b) the cost of routing from the anchored leaves of the service composition tree, through the optimum service placement tree, to the host for p in the optimum solution. We first give a bound for the quantity (b) and then use this bound to recursively build up a bound for (a) since each previously chosen portion of the greedy placement has already been bound.

We define the following notation: D_p are the descendants of p (excluding both p and the leaf nodes). $c(p)$ is the cost of routing data from the host nodes of children C_p to the host of p in the greedy algorithm. In other

words, for function f defined by the greedy algorithm, $c(p) = \sum_{q \in C_p} d_q^o w_{(f(q),f(p))}$. Similarly, $c^*(p)$ is the cost for the optimum placement. π_{pq} (with q being a descendant of p) is the set of service along the path from p to q, including q but not p. If $p = q$, then the set is empty. $r_{pq} = \prod_{h \in \pi_{pq}} r_h$ is the data reduction rate along the path from q to p, again including q but not including p. If $p = q$ the product over the empty set is 1. $\gamma^*(p)$ is the cost of routing a total data amount d_p^i from leaves to p in the optimum solution. The total amount of data is distributed among the leaves proportional to the real traffic. Precisely, $\gamma^*(p) = \sum_{q \in (D_p \cup p)} r_{pq} c^*(q)$.

By definition, for each node p, the greedy algorithm picks a placement that minimizes the cost of sending data d_p^i from p's child nodes to p. So, the cost $c(p)$ should be no greater than the cost of shipping the data from each $q \in C_p$ along the greedy assignment path all the way back to the leaf nodes, then along the optimum path according to $f^*(p)$. This implies the following inequality:

$$c(p) \leq \gamma^*(p) + \sum_{q \in D_p} r_{pq} c(q). \tag{2}$$

Using this inequality, we can derive the following lemma.

Lemma 1 *For each node p, the greedy placement cost and the optimum placement cost satisfies:*

$$c(p) \leq \sum_{q \in (D_p \cup p)} r_{pq} 2^{|\pi_{pq}|} c^*(q) \tag{3}$$

The proof is given in [1]. Applying this lemma recursively, the cost of the entire tree is: $C(T) = \sum_{p \in I} c(p) \leq \sum_{p \in I} \sum_{q \in (D_p \cup p)} r_{pq} 2^{|\pi_{pq}|} c^*(q) \leq \sum_{p \in I} \sum_{q \in (\pi_{\mathcal{R}p} \cup \mathcal{R})} r_{qp} 2^{|\pi_{qp}|} c^*(p)$. The second inequality follows because, in trees, the set of all (internal node, descendant) pairs is exactly the same as the set of all (ancestor, internal node) pairs. In both summations, the pair (p, q) occurs once iff p is an ancestor of q. If all data reduction rates are the same value R, $C(T) \leq \sum_{p \in I} \sum_{q \in (\pi_{\mathcal{R}p} \cup \mathcal{R})} (2R)^{|\pi_{qp}|} c^*(p) \leq \sum_{p \in I} \frac{(2R)^{|\pi_{\mathcal{R}p}|+2} - 1}{2R-1} c^*(p)$. So, we have the following theorem.

Theorem 1 *For a tree service placement problem, if the data reduction rate at every service satisfies $r \leq R < 1/2$, then $C(T) \leq \frac{1}{1-2R} C^*(T)$.*

In particular, if $R = 1/4$, we have $C(T) \leq 2C^*(T)$. This result is independent of the size of the communication graph and the height of the service tree.

The Modified Greedy Placement Algorithm When the data reduction rate is greater than or equal to $1/2$, the result from Theorem 1 does not apply. In this situation, we can cluster the original service tree and introduce "super-services" that have stronger data reduction rates. We then perform the greedy assignment algorithm on the modified service tree. In particular, given a service tree $T = (O, L)$ and a desired data reduction rate R, we create a modified tree $T' = (O', L')$ by applying the following graph transformation:

1. Initialize $O' = \{s | s \in S\}$, $L' = \emptyset$.
2. For each service $p \in O$ with $C_p^T \subset O'$
   - If $r_p < R$: Add p to O', add links from C_p^T to p into the set L'. Delete p from O. Here C_p^T are the children of p in T.
   - Otherwise, if $r_p \geq R$: For each $c \in C_p^T$, remove link (c, p) from L and add link (c, q), where q is p's parent in T. Delete p from O. Redefine $d_q^i = d_q^i - d_p^o + \sum_{c \in C_p^T} d_c^o$ and $r_q = \frac{d_q^o}{d_q^i}$.
3. If O contains only S and $\mathcal{R}$, end; else, go to Step 2.

The new tree T' thus created will have all the leaves of T, and all the data reduction rates less than R. We can then perform the greedy placement algorithm for T'.

Theorem 2 *The optimum tree for routing T' costs at most $\frac{1}{R}$ times the optimum tree for routing T. Precisely, $C^*(T') \leq \frac{1}{R} C^*(T)$.*

Proof: Take the original service tree T. For all nodes in $O - O'$, give the node a data reduction rate of 1 and call this tree $\bar{T}$. Clearly, $C^*(\bar{T}) \leq \frac{C^*(T)}{R}$ since the data rate at any node is increased by at most $\frac{1}{R}$. Now, any solution for $\bar{T}$ can be used for T' to get the same total cost.

Theorem 3 *For $R = 1/4$, the Modified Greedy Algorithm solution for T, $\hat{C}(T)$, has cost at most $8OPT$. Precisely, $\hat{C}(T) \leq 8C^*(T)$.*

Applying the greedy algorithm to T', $C(T') \leq \frac{1}{1-2R} C^*(T')$. Combining with Theorem 2, $C(T') \leq \frac{1}{R-2R^2} C^*(T)$. Furthermore, $\hat{C}(T) \leq C(T')$, because we can route T using cost at most $C(T')$ by placing all excluded services at the host for the included service closest along the path to the root. We choose $R = 1/4$ to minimize $\frac{1}{R-2R^2}$, giving an approximation of 8. Again, this guarantee is independent of the height of the tree.

3.3 A Distributed Implementation

The goal of this section is to describe an efficient way to find a new placement for a service, and all services that depend on it, in a distributed manner such that the communication cost of the new placement has the same performance guarantee as that of the modified greedy algorithm. We consider the situation where there is a change in a single

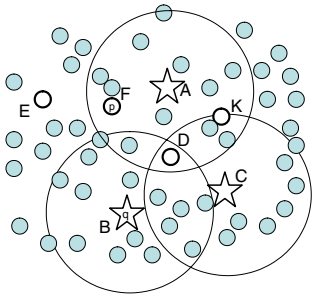


Figure 3. A limited range flooding

service that is part of a larger service graph that has already been placed. The change may come from the node that is hosting the service or from a change in the structure of the service tree.

A brute force implementation when service p changes is for every child of p to flood the entire network. Each flooding sets up the gradient of cost to the particular child node. Every node computes the total cost if it were to be the host of p. Then, a leader election process chooses the host with minimum total cost. This process is repeated for every ancestor of p.

We now show how, when the child services are proximate, the optimum median can be found without flooding the entire network. The more closely located the set of siblings, the less flooding is used to find the minimum cost median. For many sensor network applications, sibling services are proximate since usually the users are interested in data that is relatively local.

The general idea of the algorithm is to flood some small area around the siblings until the flooding ranges intersect at $some$ median (not necessarily the minimum cost median). Then, the found median is used to limit the search space for the optimum median. Figure 3 shows a limited range flooding emanating from three child nodes A, B, and C, represented by the stars. The circles are flooding ranges with values C_A, C_B, and C_C respectively. D is a node where the three flooding ranges intersect. It may not be the minimum cost median, but the minimum cost median will be within the *union* of the three flooding ranges, since for each node, like E, that is outside the union, every edge weight in the cost function $c(E)$ will be larger than the corresponding edge weight in the cost function for node D.

In the following algorithm, we assume that the nodes are reasonably time synchronized. We denote the children, siblings, and parent of a service p with F_p and call the hosts of services F_p the *immediate relatives* of p. A slight abuse of notation, we assume sending $f(F_p)$ in a message includes the ID numbers of the immediate relatives of p along with information about which host holds which service (similarly for C_p). We assume that if p is hosted at node h, the broadcast distances to the immediate relatives of p are known to h. If this assumption cannot be made, then the ID numbers of the immediate relatives are known and can be used to determine distances using standard distributed algorithms such as doubling broadcast distance until the requested node responds. We also assume that when a message is sent from some location, it will first be received by a node along a shortest path. It is assumed that all nodes know α, a parameter of the algorithm that is used as the initial value in limited cost flooding.

The algorithm performs in two stages: a limited cost local flooding from each child to find an initial median and a flooding to the union of the first flooding zone to find the optimum median. The algorithm is defined by 4 states. Each state has an event that triggers entry into the state, some activity that the node performs while in the state, and an event that triggers exit from the state. Note that the states are not exclusive, that is, a node can be in more than one states at the same time. In addition, there are 5 types of messages that nodes might send during the course of the algorithm. Where applicable, the sender and recipient of the message is indicated using unique node ID numbers. The $newHost(p, h, f(F_p))$ message indicates that service p, with immediate relatives $f(F_p)$, has a new host with ID number h. Regardless of the state, a node hosting service p that receives a $newHost(p, h, empty)$ message sends a $newHost(p, h, f(F_p))$ message to h and deletes p from the set of services it hosts. Upon receipt of a $newHost$ message, all nodes update the new service location if the information is relevant for the services it hosts. To prevent the description from becoming cumbersome we have omitted details that can either be inferred or implemented using standard techniques. These details include the precise content of the messages and the manner of calculating distance to a node using a message received from that node.

INITIATE STATE

Entry Event: A change in the network occurs requiring a new placement of service q.

Activity: Choose arbitrary child $p \in C_q$ with host $f(p)$. Send $newHost(p, h, f(F_p))$ message to $f(p)$ and $f(q)$ with $h = f(p)$ and $f(F_p)$ empty. Essentially, this message tells the node hosting p to move the service p from itself to itself in order to initiate the greedy algorithm.

Exit Event: Send $newHost$ message.

LEAD STATE

Entry Event: A $newHost(p, h, f(F_p))$ message is received and the new host ID matches my own ID, $f(F_p)$ is not empty, and p is not the root of the service graph.

Activity:

1. Set q to be the parent of p in the service graph.
2. Broadcast $initiate(self, q, f(C_q))$ message to hosts of C_q.
3. Upon first receipt of a *medianValue(v,id)* message, set w equal to the time elapsed between entrance into the leader state until the current moment. Use w and knowledge of the algorithms to determine the time-towait needed until all $medianValue$ messages are received. Choose the $medianValue$ message with the smallest v, and set $f(q)$ (the new host for q) to be the id from this message. Send message $newHost(q, f(q), empty)$ to nodes in $f(F_p)$.

Exit Event: Send $newHost$ message.

FLOOD STATE
Entry Event: Receive $initiate(p, q, f(C_q))$ message, where my ID w is in $f(C_q)$.
Activity: Set $J = \alpha$. J is a cost bound on the reach of the message such that the message will be received by node u iff $w_{(w,u)}d^o_w \leq J$.
LOOP: Broadcast a $flood_{radius}(J, p, q, f(C_q))$ message. If a $terminate(q)$ message is not received within given timeout, let $J = g(J)$ where g is monotonically increasing in J, and return to LOOP (perform another bounded depth flooding).
Exit Event: Receive $terminate(q)$ message.

SEARCH STATE
Entry Event: Receive $flood_{radius}(J, p, q, f(C_q))$ message.
Activity:

1. Retransmit any $flood_{union}(union, p, q, f(C_q), v)$ message received.
2. If receive $flood_{radius}$ or $flood_{union}$ messages from all hosts of services C_q
   - If the all messages are $flood_{radius}$ messages, send $terminate(q)$ message to all nodes $f(C_q)$.
   - Compute $v = \{$the cost of hosting q myself$\}$.
   - Compute $v^* = \{$the smallest from amongst v and any median value received in any $flood_{union}$ message so far$\}$.
   - Send $medianValue(v, self)$ message to the leader p iff $v \leq v^*$
   - Send $flood_{union}(union, p, q, f(C_q), v^*)$ messages to all neighbors.

Exit Event: Send $flood_{union}$ messages to all neighbors.

Consider the distributed algorithm at work in the example from Figure 3. Node F will soon run out of batteries and the service p it is hosting must be placed at a different node. Node F chooses to assign leadership to node B, because B is hosting $q \in C_p$, an arbitrarily chosen child of p. Leader node B sends a message to the hosts of p's children, A, B, and C, telling them to initiating their flooding. When node D hears from A, B, and C, it sends them all a $terminate$ message to stop their flooding processes. D computes the cost of hosting p itself, and sends this value to leader node B. Node D also forwards its median value and distances to A, B, and C on to its neighbors so that the gradients will continue to grow into the union space. Similarly, all of the nodes inside the union space will forward the gradients within the union while they are waiting for the gradient from all children of p. Once all gradients are received, they too will send the cost of hosting p themselves to the leader B if the hosting cost is better than D. Once B receives the median value message from D, it can anticipate the time to wait for other nodes since weights are symmetrical. B chooses a host for p that has minimum cost, say node K and sends this decision to the hosts of the immediate relatives of q so that they can update their information, including the old host site for p (node F). Node F fills in information about the hosts of F_p and retransmits this message to K so that K has all the information it needs to properly host p. Now K is the leader of a new median search for a node to host the parent of p, unless p is the root.

For each service p placed in this process, $2|C_p|$ bounded depth floodings are needed. If this amount of flooding is still too expensive and the optimality of the solution can be further relaxed, we can use the first median found. In fact, we can improve the algorithm by using time varying flooding where a message is delayed at each node proportional to the amount of data transmitted along the incoming link. In this way, the node where the flooding messages meet is closer to services sending more data.

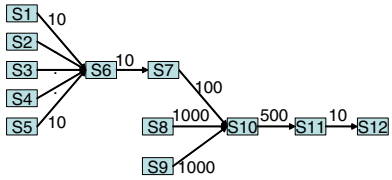
4 Simulations

We evaluate both the placement and adaptation costs of the greedy algorithm through simulation. The placement cost is the value of the placement given by equation (1) from section 2, and the adaptation cost is the amount of energy expended in adapting the placement after a change or failure in the network. We use four kinds network topology: perturbed grid, perturbed grid with a hole, random topology, and random topology with a hole. In a square region of 1000x1000, various numbers of sensors are deployed. The communication range of each node is chosen uniformly from $[0.5\phi, 1.5\phi]$, where ϕ is a parameter to control different network densities. The cost on the edges are uniformly distributed in [10, 15]. The root of the service tree is always set at the bottom left corner of the network, i.e., at coordinate (0, 0). Examples of the topologies used and some placement results can be found in [1].

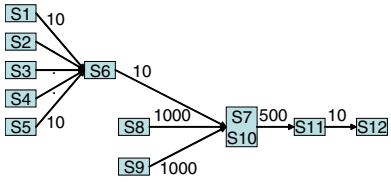
4.1 Placement Cost

We use two sets of examples to study placement performance in the simulations: a sparse tree that is inspired by a parking garage application, and a regional data collection tree inspired by TinyDB. In the first example, we compare the greedy placement results with the optimum placement and the in-network relaxation based placement from 2.2.

Sparse service tree We place 64 nodes in the field and pick $\phi = 100$. We use a service graph as shown in Figure 4(a) as a representative for a class of distributed event detection applications. This particular graph is inspired by the parking garage scenarios in [10] and [7]. Assume that a set of wireless webcams are installed in a parking structure. A set of sensors (e.g. break beams or RFID readers, labeled as $S1, ..S5$ in the figure) are deployed near the entrance that detect (service $S6$) and identify (service $S7$) incoming vehicles. Camera images from $S8$ and $S9$ are stitched together at $S10$ and filtered by the car detection events (which may also contain the driver's parking preference) at $S10$. Open slots are counted (service $S11$), and their locations are returned to a display (service $S12$).



(a) Original service tree.



(b) Modified service tree.

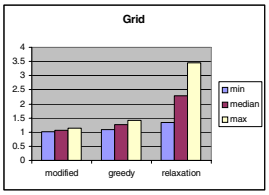
Figure 4. A service tree example

The output data rate from each sensor and service is labeled on the links. The data reduction rates $r_{S6} = 1/5$, $r_{S7} = 10$, $r_{S10} = 5/21$, and $r_{S11} = 1/50$. Mapping to the sensor network layout, we assume S1, ..., S5 are near (700, 700); cameras $S8$ and $S9$ are near (900, 100), and the root service $S12$ is at the origin (0,0). Note that not all data reduction rates are less than $1/4$. We convert the service graph into the one shown in Figure 4(b) using the modified greedy algorithm.

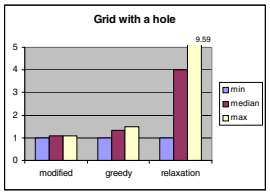
We compare the results from the optimal, a simple greedy, a modified greedy and a in-network relaxation methods, and lable the assignment cost C_{opt}, C_{greedy}, $C_{modified}$, and C_{relax}, respectively. To collect performance statistics, we run each algorithm on 32 randomly generated networks in each topology category and compute the ratio of the total cost over the optimal cost. Figure 5 shows the statistics of the results. Each network topology is shown in a separate subfigure. The min, median, and max ratio are plotted. In all these examples, the modified greedy placement performs better than greedy placement, which is better than in-network relaxation.

Data aggregation The next set of experiments test the greedy placement algorithm in data aggregation applications. The network is set up similar to the previous evaluation and a base station is placed at the bottom left corner. The tasks are aggregation queries, such as MAX, computed over a subset of sensor inputs. We compare the performance of greedy placement with TinyDB [11] like aggregation trees, which we call TAG trees. A TAG tree is built without prior knowledge of what sensor data will be collected. In the simulation, we used the tree formed by the shortest paths from all nodes to the root.

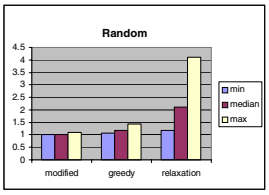
An aggregator, like MAX, is placed at every node on the



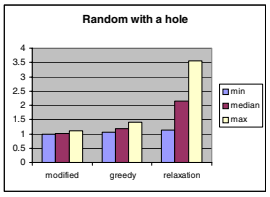
(a) Perturbed grid topology.



(b) Perturbed grid with a hole.



(c) Random topology.



(d) Random topology with a hole.

Figure 5. Placement algorithm statistics

TAG tree. Since these aggregators take an arbitrary number of inputs and produce only one output of the same data type, their data reduction rate equals the inverse of the number of their children in the TAG tree. In these data aggregation scenarios, the optimum placement of aggregators is a minimum-cost Steiner tree [18]. In this section, we apply the greedy placement algorithm to an n-ary balanced tree. We compare our placement cost with the cost of a TAG tree. By using an n-ary tree, we effectively create aggregation services with data reduction rate $1/n$. Given a network of N nodes in a square field, we select $m = \lfloor \sqrt{N} \rfloor$ nodes as source sensors. We place these sources "evenly" along the anti-diagonal line (from upper left to lower right), so that we give the greedy algorithm enough space (i.e. the upper-right half of the field) to make sub-optimal choices. To achieve this effect, we divide the field into an $m \times m$ grid, and select one sensor from every anti-diagonal grid. We run 32 experiments for each $N = 64$, 128, 256, and 512 under a perturbed grid topology. In each configuration we vary the data reduction rate R (i.e. the branching factor in the aggregation tree) to be 2, 4, 6, and 8.

Figure 6 plots the ratios between the cost of the greedy algorithm placement on trees with varying branching factors and the cost of placement using a predefined TAG tree. In almost all cases, the greedy placement performs better than the TAG tree. The result is not surprising, since by breaking up the aggregator, we have created a specific tree for the specific set of data sources. A notable exception is in Figure 6(a) with group size 8. Since there are only 8 data sources in the configuration, by building a 8-ary tree, we are not doing any in-network aggregation.

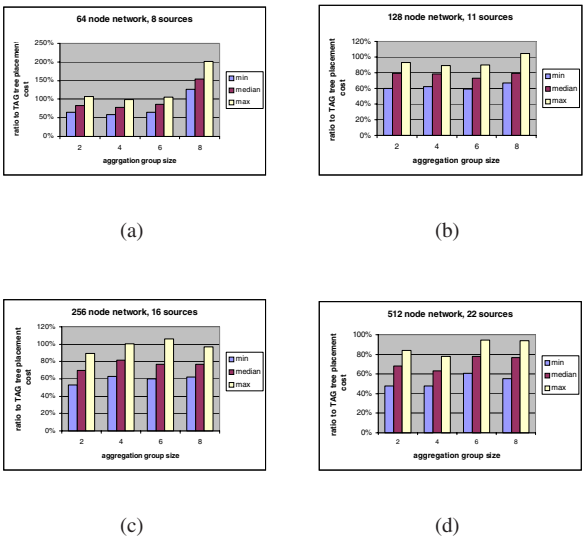


Figure 6. Greedy vs. TAG trees

4.2 Adaptation Cost

In this section, we evaluate the performance of our distributed greedy algorithm and, in particular, the effectiveness of using bounded depth flooding to find the minimum cost weighted median. There are three key metrics to evaluate: the flooding zone union, the number of $medianValue$ messages sent, and the quality of the initial median. The flooding zone union is defined as the number of nodes reached in the flooding processes, or, the number of nodes that enter the SEARCH STATE. The smaller this zone, the less energy and network traffic is spent forwarding flooding messages. The number of $medianValue$ messages sent is the number of nodes in the flooding zone union that have cost smaller than the cost of the first median found at the intersection of the individual flooding zones. A node only sends its median value to the leader node if it is less than the median value found initially. Therefore, the fewer the number of nodes that would cost less than the initial median, the fewer $medianValue$ messages will be sent. Reducing the number of $medianValue$ messages conserves valuable energy and communication resources. Finally, the quality of the initial median is defined as the ratio between the minimum cost median and cost of the initial median found. The better the quality of the initial median, the more promise there is in using it as a substitute for the optimum median when communication resources are severely constrained.

We run the simulation in networks with 512 nodes randomly scattered in a physical space of size 1000×1000. In a square region of size 200×200 at the center of the space, we randomly place a number of child services who need to find a host for their parent node. These children are in the center of the space so that the flooding process can reach a large number of nodes without hitting the edge of the network. All children services have the same amount of outgoing data. The number of children is a simulation parameter, chosen from $\{3, 4, 5\}$. We run 32 simulations for each parameter.

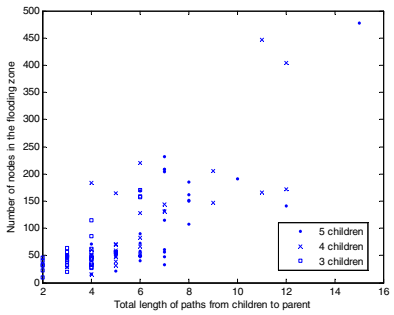
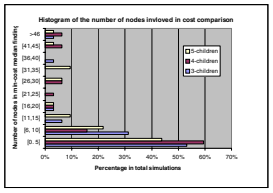


Figure 7. Flooding zone statistics

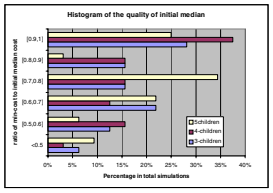
Figure 7 shows the size of the flooding zone union under different runs. The X-axis is the total length of the communication paths from child hosts to the chosen parent host. This is used to measure the closeness of the child hosts. The smaller this number, the closer the child hosts are, and thus the smaller the flooding zone union should be. Although not used in our simulations, the unweighted median cost could also serve as a lower bound on the amount of communication that must be expended in finding this median (assuming the only way to determine distances is through direct communication between nodes). Since the flooding zones stop growing when all children gradients intersect, the more children, the larger the flooding zone union tends to be. The significant reduction in the size of the flooding zone union as the proximity of the children nodes increases, emphasizes that the performance of the distributed greedy algorithm is dependent on the proximity of the children nodes.

Figure 8(a) shows a histogram of 96 simulations, bucketed according to the number of $medianValue$ messages in the simulation. In the majority of the simulations, less than 10 nodes had a cost that was smaller than the cost of the initial median. Less than 9% of the simulations had more than 46 nodes send medianValue messages. Out of a total of 512 nodes, this shows a significant reduction in communication and processing costs as a result of the low cost of the initial median. Figure 8(b) shows the distribution of the cost ratio between the optimum median and the initial median. In the majority of the simulations, the ratio is more than $\frac{2}{3}$, with about $\frac{1}{3}$ of the simulations resulting in an optimum median that is more than .9 the initial value. Although there is still more to understand about the full implications of using the initial median in place of the optimum, these results indicate that this option has real potential.

It should be pointed out that sometimes the objectives of minimizing the flooding zone union and minimizing the cost of the initial median are in conflict. For instance, often setting the individual flooding radii to be the same for all nodes, independent of the amount of outgoing data at that



(a) The distribution of the number of nodes whose cost is smaller than the cost of the initial median.



(b) The distribution of the cost ratio between the minimum median and the initial median.

Figure 8. Initial median statistics

node, will result in a smaller union flooding zone but will increase the value of the initial median found. Further exploration into the tradeoffs between these two metrics and their relative importance in various application scenarios is an interesting direction for future work.

5 Conclusion

We studied the performance bound for a modified greedy algorithm for service tree placement and proposed a distributed scheme to adapt to network and service tree changes. We recommend using the the optimum dynamic programming solution when all information is centralized, the network is relatively stable, or initially when a task is deployed. When the network is dynamic, the greedy algorithm is a fast, simple, distributed, and efficient alternative with both provable guarantees and easy adaptivity.

The simulations in data aggrgation scenarios suggest that performance improves when the routing tree is defined based on knowledge about the modules to be aggregated. As would be expected, tailoring the routing to the specific user task leads to significant reductions in energy consumption. This motivates further exploration into the possibility of a new paradigm in which routing structures are defined in the network, in real time, and can evolve, adapt, and change as the task at hand changes.

References

[1] Z. Abrams and J. Liu. Greedy is Good: On service tree placement for in-network stream processing. Technical Report MSR-TR-2005-171, Microsoft Research, November 2005.

[2] A. Bestavros, A. D. Bradley, A. J. Kfoury, and M. Ocean. snBench: A development and run-time platform for the rapid deployment of video sensornet applications. In *Proc. of 2nd Intl. Workshop on Broadband Advanced Sensor Networks (Basenets'05), Boston, MA*, October 2005.

[3] A. Billionnet. Allocating tree structured programs in a distributed system with uniform communication costs. *IEEE Trans. Parallel Distrib. Syst.*, 5(4):445–448, 1994.

[4] S. H. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Trans.Software Eng.*, SE-7(6):583–589, 1981.

[5] B. J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Proc. Information Processing in Sensor Networks (IPSN'03), Palo Alto, CA*, pages 47–62, April 2003.

[6] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. *Lecture Notes in Computer Science*, 1987:3–14, 2001.

[7] J. Campbell, P. B. Gibbons, and S. Nath. IrisNet: An internet-scale architecture for multimediasensors. In *Proc. of ACM Multimedia (MM'05), Singapore*, November 2005.

[8] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow management for grid computing. In *Proc. Intl. Symposium on Cluster Computing and the Grid (CCGrid'03), Tokyo, Japan*, pages 198–205, May 2003.

[9] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An architecture for a world-wide sensor web. *IEEE Pervasive Computing*, 2(4):22–33, 2003.

[10] J. Liu and F. Zhao. Towards semantic services for sensor-rich information systems. In *Proc. of the 2nd Intl. Workshop on Broadband Advanced Sensor Networks (Basenets'05), Boston, MA*, October 2005.

[11] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, December 2002.

[12] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. In *Proc. Intl. Conf. Database Engineering (ICDE'05), Tokyo, Japan*, April 2005.

[13] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the First International Workshop on Data Management for Sensor Networks (DMSN), Toronto, Canada*, Augest 2004.

[14] M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, 1993.

[15] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. 22nd Intl. Conference on Data Engineering (ICDE'06), Atlanta, GA*, April 2006.

[16] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proc. 24th ACM Symposium on Principles of Database Systems (PODS'05), Baltimore, MD*, June 2005.

[17] P. T. Uriel Feige, Laszlo Lovasz. Approximating min-sum set cover. In *Proc. 5th Intl. Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX 2002), Rome, Italy*, pages 94–107, September 2002.

[18] V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[19] B. Veltman, B. J. Lageweg, and J. K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16(2-3):173–182, 1990.

[20] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: a framework for the composable semantic interpretation of sensor data. In *Proc. European Workshop on Wireless Sensor Networks (EWSN'06), Zurich, Switzerland*, Feb. 2006.