Adaptive Data Placement for Wide-Area Sensing Services

Suman Nath[*]
Microsoft Research
sumann@microsoft.com

Phillip B. Gibbons
Intel Research Pittsburgh
phillip.b.gibbons@intel.com

Srinivasan Seshan
Carnegie Mellon University
srini@cs.cmu.edu

Abstract

Wide-area sensing services enable users to query data collected from multitudes of widely distributed sensors. In this paper, we consider the novel distributed database workload characteristics of these services, and present IDP, an online, adaptive data placement and replication system tailored to this workload. Given a hierarchical database, IDP automatically partitions it among a set of networked hosts, and replicates portions of it. IDP makes decisions based on measurements of access locality within the database, read and write load for individual objects within the database, proximity between queriers and potential replicas, and total load on hosts participating in the database. Our evaluation of IDP under real and synthetic workloads, including flash crowds of queriers, demonstrates that in comparison with previously-studied replica placement techniques, IDP reduces average response times for user queries by up to a factor of 3 and reduces network traffic for queries, updates, and data movements by up to an order of magnitude.

1 Introduction

Emerging *wide-area sensing services* [18, 26, 27] promise to instrument our world in great detail and produce vast amounts of data. For example, scientists already use such services to make observations of natural phenomena over large geographic regions [2, 5]; retailers, such as Walmart [9], plan to monitor their inventory using RFID tags; and network operators (ISPs) monitor their traffic using a number of software sensors [16]. A key challenge that these services face is managing their data and making it easily queriable by users. An effective means for addressing this challenge is to store the vast quantity of data in a wide-area distributed database, which efficiently handles both updates from geographically dispersed sensors and queries from users anywhere in the world [14].

Like traditional distributed databases, sensing service databases must carefully replicate and place data in order to ensure efficient operation. Replication is necessary for avoiding hot spots and failures within the system, while careful data placement is required for minimizing network traffic and data access latency. Although replication and data placement have been extensively studied in the context of many wide-area systems [10, 12, 24, 32, 36, 37, 40, 41], existing designs are ill-suited to the unique workload properties of sensing services. For example, unlike traditional distributed databases, a sensing service database typically has a *hierarchical organization* and a *write-dominated* workload. Moreover, the workload is expected to be *highly dynamic* and to exhibit *significant spatial read and/or write locality*. These unique workload properties, discussed in further detail in this paper, significantly complicate the design of replication and placement techniques. For example, although replicating data widely can reduce query response times by spreading client load among more host machines and providing replicas closer to queriers, it also increases network traffic for creating replicas and propagating the large volume of writes to them. Similarly, placing data near the sources of reads/writes can reduce network overheads, but it also increases latency and CPU load for answering the hierarchical queries common in sensing services.

In this paper, we present an online, adaptive data placement and replication scheme tailored to the unique workload of wide-area sensing services. This scheme, called *IrisNet Data Placement (IDP)*, has been developed in the context of the IrisNet wide-area sensing system [8]. With IDP, each host efficiently and locally decides when to off-load data to other hosts and what data to off-load. The target hosts are selected such that data accessed together tend to reside on the same host. IDP makes such online decisions based on workloads, locality of queries, proximity of queriers and candidate replica locations, and total load on the hosts running the service. In making these decisions, IDP tries to balance three performance metrics: response time, bandwidth used for creating replicas, and bandwidth used to keep replicas up-to-date, in a way particularly suited to wide-area sensing service workloads. Such automatic and adaptive techniques are essential for a large-scale, widely-distributed sensing service because service administrators cannot be expected to determine and maintain an effective partitioning across all the hosts.

IDP explicitly detects flash crowds (*i.e.,* sudden bursts in queries to spatially localized data), replicates in response to them, and sheds load to newly created replicas, but only as long as the flash crowd persists. Thereafter, if load on the new replicas drops, unneeded replicas are destroyed. IDP places new replicas in proximity to the sources of queries and updates, but routes queries using a

[*] Work done while the first author was a PhD student at CMU and an intern at Intel Research Pittsburgh.

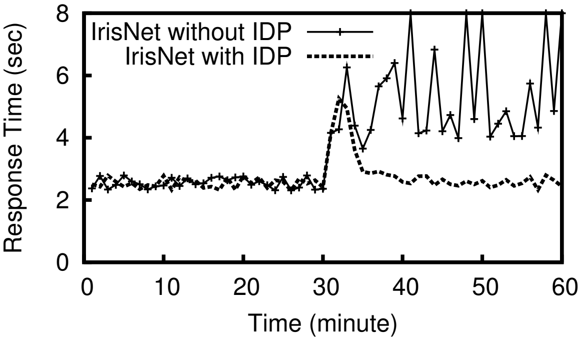


Figure 1: Result of injecting a real query workload into a 310 PlanetLab host deployment of IrisNet with and without IDP. A flash crowd is emulated at time = 30 minutes by increasing query rate by 300 times. A response time of 8 second represents a timeout.

combination of proximity routing and randomization, to ensure that queries favor nearby replicas but new replicas can succeed in drawing query load away from preexisting closer replicas.

We implemented IDP within IrisNet and evaluated its performance under real and synthetic sensing system workloads. Figure 1 presents a sample result demonstrating IDP's effectiveness (details of the experimental setup will be described in Section 7). As shown, IDP helps IrisNet tolerate a flash crowd gracefully, keeping the response times close to their pre-flash crowd levels. Without IDP, IrisNet quickly becomes overloaded, resulting in higher response times and frequent query timeouts. Further results show the effectiveness of various components of IDP and provide a comparison with two previously-studied adaptive placement schemes [36, 41]. Compared to these schemes, IDP reduces average response time for user queries by up to a factor of 3 and reduces network traffic by up to an order of magnitude, while serving user queries, propagating new sensor data to replicas, and adjusting data placements.

In the remainder of the paper, we describe the characteristics of wide-area sensing service workloads and the architecture of a wide-area sensing system, present the algorithms that comprise IDP, describe our implementation of IDP, evaluate IDP through real deployment and simulation, briefly discuss related work, and conclude by summarizing our findings.

2 Wide-Area Sensing Systems

Although IDP has been developed in the context of IrisNet, the techniques are tailored more to the workload than to the specific sensing system, and hence should be effective for many wide-area sensing systems. In this section, we describe the workload in further detail by considering a few example wide-area sensing services. Then we present an architecture for a generic sensing system on which these services can be built; IDP will be discussed in the context of this generic system, in order to emphasize its applicability beyond IrisNet.

2.1 Wide-Area Sensing Services

We consider representative services from five classes of wide-area sensing services, as summarized in Table 1:

Asset Tracker: This service is an example of a class of services that keeps track of objects, animals or people. It can help locate lost items (pets, umbrellas, *etc.*), perform inventory accounting, or provide alerts when items or people deviate from designated routes (children, soldiers, *etc.*). Tracking can be done visually with cameras or by sensing RFID tags on the items. For large organizations such as Walmart or the U.S. Military, the global database may be TBs of data.

Ocean Monitor: This service is an example of a class of environmental monitoring services that collect and archive data for scientific studies. A concrete example is the Argus coastal monitoring service [2, 18], which uses a network of cameras overlooking coastlines, and analyzes near-shore phenomena (riptides, sandbar formation, *etc.*) from collections of snapshot and time-lapse images.

Parking Finder: This service monitors the availability of parking spaces and directs users to available spaces near their destination (see, *e.g.*, [14, 18]). It is an example of a class of services that monitors availability or waiting times for restaurants, post offices, highways, *etc.* Queriers tend to be within a few miles of the source of the sensed data, and only a small amount of data is kept for each item being monitored.

Network Monitor: This service represents a class of distributed network monitoring services where the "sensors" monitor network packets. A key feature of this service is the large volume of writes. For example, a network monitoring service using NetFlow [4] summarization at each router can generate 100s of GBs per day [16]. We model our Network Monitor service after IrisLog [7, 18]. IrisLog maintains a multi-resolution history of past measurements and pushes the distributed queries to the data.

Epidemic Alert: This service monitors health-related phenomena (number of people sneezing, with fevers, *etc.*) in order to provide early warnings of possible epidemics (see, *e.g.,* [3]). When the number of such occurrences significantly exceeds the norm, an alert is raised of a possible epidemic. We use this service to consider a *trigger-based* read-write workload in which each object in the hierarchy reports an updated total to its parent whenever that total has changed significantly, resulting in a possible cascading of updates. Trigger-based workloads have not been considered previously in the context of data placement, but they are an important workload for sensing services.

	Asset Tracker	Ocean Monitor	Parking Finder	Network Monitor	Epidemic Alert
Global DB Size	large	very large	medium	large	medium
Write Rate	medium	low	low	very high	very low
Read/Write Ratio	low	very low	low	very low	one
Read Burstiness	low	very low	high	low	very low
Read Skewness	building, skewed	root, uniform	neighborhd, skewed	domain, uniform	uniform, uniform
R/W Proximity	neighborhood	root	neighborhood	domain	root
Consistency	variable: secs–mins	relaxed: ≈mins	variable: secs–mins	strict: ≈secs	relaxed: ≈mins

Table 1: Five representative wide-area sensing services and their characteristics.

A Qualitative Characterization. In Table 1, we characterize qualitatively the five representative services along seven dimensions. *Global DB Size* is the size of the global database for the service. Note that for Ocean Monitor, it includes space for a full-fidelity historical archive, as this is standard practice for oceanography studies. For Network Monitor, it includes space for a multi-resolution history. In order to provide different points of comparison, for the remaining services, we assume only the most recent data is stored in the database.[1] *Write Rate* is the rate at which objects are updated in the global database. We assume services are optimized to perform writes only when changes occur (*e.g.*, Parking Finder only writes at the rate parking space availability changes). *Read/Write Ratio* is the ratio of objects read to objects written. A distinguishing characteristic of sensing services is that there are typically *far more writes than reads*, as far more data is generated than gets queried. An exception in our suite is Epidemic Alert, because we view each triggered update as both a read and a write. *Read Burstiness* refers to the frequency and severity of flash crowds, *i.e.*, large numbers of readers at an object in a short window of time. For example, Parking Finder has flash crowds right before popular sporting events. *Read Skewness* is measured in two dimensions: the geographic scope of the typical query (*e.g.*, parking queries are often at the neighborhood level) and the uniformity across the objects at a given level (*e.g.*, parking queries are skewed toward popular destinations). *R/W Proximity* reflects the typical physical distance from the querier to the sensors being queried, as measured by the lowest level in the hierarchy they have in common. For example, an oceanographer may be interested in coastlines halfway around the world (root level proximity), but a driver is typically interested only in parking spaces within a short drive (neighborhood level proximity). *Consistency* refers to the typical query's tolerance for stale (cached) data.

2.2 Sensing System Architecture

We now describe the architecture of a generic wide-area sensing system on which the above services can be built. At a high level, a sensing system consists of three main components.

Data Collection. This component collects data from the sensors, which may be cameras, RFID readers, microphones, accelerometers, *etc*. It also performs *service-specific filtering* close to the sensors, in order to reduce high-bit-rate sensed data streams (*e.g.,* video) down to a smaller amount of post-processed data. In IrisNet, for example, this component is implemented as *Sensing Agents* (SAs) that run on Internet-connected hosts, each with one or more attached sensors. The post-processed data is stored as *objects* in the data storage component, described next.

Data Storage. This component stores and organizes the data objects so that users can easily query them. Because of the large volume of data and because objects are queried far less frequently than they are generated, a scalable architecture must distribute the data storage so that objects are stored near their sources. Sensed data are typically accessed in conjunction with their origin geographic location or time. Organizing them hierarchically, *e.g.,* according to geographic and/or political boundaries, is therefore natural for a broad range of sensing services. We assume that each sensing service organizes its data in its own hierarchies, tailored to its common access patterns. For example, the data of the Parking Finder may be organized according to a geographic hierarchy (for efficient execution of common queries seeking empty parking spots near a given destination), while data in an Ocean Monitor may further be organized according to timestamps (supporting queries seeking all riptide events in the year 2004).[2] Thus, the storage component of a sensing system provides a hierarchical distributed database for a sensing service. Let the *global database* of a service denote a sensor database built from the complete hierarchy for a service and the data collected from all the sensors used by that service. A host in the sensing system contains a part of this database, called its *local database*. Thus, each sensing service has exactly one global database, which can be distributed among multiple hosts. However, each host may own multiple local databases, one for each sensing service using the host.

In IrisNet, for example, this storage component is implemented as *Organizing Agents* (OAs) that run on

[1] This assumption is not crucial for the results that follow.

[2] A service can have multiple hierarchies over the same data.

Internet-connected hosts. The global database is represented as a single (large) XML document whose schema defines the hierarchy for the service. Note that the hierarchy is logical—any subset of the objects in the hierarchy can be placed in a host's local database.

Query Processing. A typical query selects a subtree from the hierarchy. The query is then routed over the hierarchy, in a top-down fashion, with the query predicates being evaluated at each intermediate host. The hierarchy also provides opportunities for in-network aggregation—as the data objects selected by the query are returned back through the hierarchy, each intermediate host aggregates the data sent by its children and forwards only the aggregated results. For example, a query for the total value of company assets currently within 100 miles of a given disaster is efficiently processed in-network by first summing up the total asset value within each affected building in parallel, then having these totals sent to each city object for accumulation in parallel, and so on up the object hierarchy. This technique, widely used in wireless sensor networks [31], is also used in a few existing wide-area sensing systems [14, 17, 39].

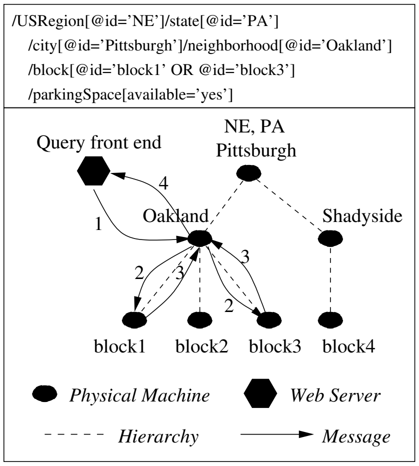
For example, in IrisNet, a query is specified in XPATH [43], a standard language for querying an XML document. Figure 2 shows a portion of an example hierarchy (for a Parking Finder service), a partitioning of the hierarchy among hosts (OAs), and an example of how queries are routed on that particular partitioning. The hierarchy is geographic, rooted at NE (the northeastern United States), continuing downward to PA (state), Pittsburgh (city), Oakland and Shadyside (neighborhoods), and then individual city blocks within those neighborhoods. In the figure, a single physical host holds the NE, PA, and Pittsburgh objects (*i.e.,* the data related to the corresponding regions). The (XPATH) query shown is for all available parking spaces in either block 1 or block 3 of Oakland. The query is routed first to the Oakland host, which forwards subqueries to the block 1 and block 3 hosts, which respond to the Oakland host, and finally the answer is returned to the query front-end. In general, a query may require accessing a large number of objects in the hierarchy in a specific top-down order, and depending on how the hierarchy is partitioned and placed, the query may visit many physical hosts.

3 The Adaptive Data Placement Problem

In this section, we describe the adaptive data placement problem, motivated by the characteristics of wide-area sensing services.

3.1 Data Placement Challenges

Based on the discussion so far, we summarize the distinguishing challenges and constraints for adaptive data placement for wide-area sensing services as follows:



```
/USRegion[@id='NE']/state[@id='PA']
   /city[@id='Pittsburgh']/neighborhood[@id='Oakland']
   /block[@id='block1' OR @id='block3']
   /parkingSpace[available='yes']
```

Figure 2: Top: An XPATH query. Bottom: A mapping of the objects in the hierarchy to seven hosts, and the messages sent to answer the query (numbers depict their relative order).

Data Access Pattern. The data access patterns for wide-area sensing are far more complex than traditional distributed file systems or content distribution networks. First, the in-network aggregation within a hierarchically-organized database requires that different parts of the database be accessed in a particular order and the data be transmitted *between* the hosts containing the data. Moreover, in-network aggregation updates may be trigger-based. Second, different queries may select different, possibly overlapping, parts of the hierarchy. As a result, the typical granularity of access is unknown *a priori*.

Dynamic Workload. Since many different sensing services can share the same set of hosts, even the high-level access patterns to the hosts are not known *a priori*. Moreover, the read and write load patterns may change over time and, in particular, the read load may be quite bursty. For example, after an earthquake, there may be a sudden spike in queries over sensed data from the affected geographic region. The data placement should quickly adapt to workload dynamics.

Read/Write Workload. With millions of sensors constantly collecting new readings, the aggregate update rate of the sensor database is potentially quite high (*e.g.*, 100s of GB per day for Network Monitor). Moreover, the number of writes is often *an order of magnitude or more greater* than the number of reads. Thus, any placement must consider the locations of the writers–not just the readers–and the number of replicas is constrained by the high cost of updating the replicas.

Capacity Constraints. Each object has a weight (*e.g.,* storage requirement, read and write load) and each host has a capacity (*e.g.,* storage, processing power) that must be respected.

Wide-area Distribution. The sensors and queriers may span a wide geographic area. On the other hand, many services have high R/W proximity. Thus, data placement must be done carefully to minimize communication costs and exploit any proximity.

Consistency and Fault Tolerance. Each service's consistency and fault tolerance requirements must be met. Based on our representative services, we assume best effort consistency suffices. (Note that most data have only a single writer, *e.g.*, only one sensor updates the availability of a given parking space.) For fault tolerance, we ensure a minimum number of replicas of each object.

3.2 Problem Formulation and Hardness

To capture all these aspects, we formalize the data placement problem for wide-area sensing services as follows. *Given a set of global databases (one for each service), the hosts where data can be placed, a minimum number of copies required for each object, and the capacities (storage and computation) of the hosts, adapt to the dynamic workload by determining, in an online fashion, (1) the local databases, possibly overlapping, to be placed at different hosts, and (2) the assignments of these local databases to the hosts such that the capacity of each host is respected, the minimum number of copies (at least) is maintained for each object, the average query latency is low, and the wide-area traffic is low.*

Given the complexity of this task, it is not surprising that even dramatically simplified versions of this problem are NP-hard. For example, even when there is an unbounded number of hosts, all hosts have the same capacity C, all pairs of hosts have the same latency, the query workload is known, and there are no database writes, the problem of fragmenting a global database into a set of fragments of size at most C, such that the average query latency is below a given threshold, is NP-hard.[3]

Because the global hierarchical database can be abstracted as a tree, selecting fragments is a form of graph (tree) partitioning and assigning fragments is a form of graph (tree) embedding. Many approximation algorithms have been proposed for graph partitioning and graph embedding problems (*e.g.*, in the VLSI circuit optimization literature [25, 38]). None of these proposed solutions address the complex problem we consider. However, we get the following two (intuitive) insights from the existing approximation algorithms, which we use in IDP. First, the final partitions are highly-connected clusters. If, as in our case, the given graph is a tree, each partition is a subtree of the graph. Second, if the edges are weighted (in our case, the weights reflect the frequency in which a hop in the object hierarchy is taken during query routing), and the objective is to minimize the cost of the edges between partitions, most of the heavy edges are within partitions.

4 IDP Algorithms

Due to the complexity of optimal data placement, IDP relies on a number of practical heuristics that exploit the unique properties of sensing services. Our solution consists of three simplifications. First, each host selects what parts of its local database to off-load using efficient heuristics, instead of using expensive optimal algorithms. Second, each host decides when to off-load or replicate data independently, based on its local load statistics. Finally, to mitigate the suboptimal results of these local decisions, we use placement heuristics that aim to yield "good" global clustering of objects. These three components, called *Selection, Reaction, and Placement* respectively, are described below.

4.1 Fragment Selection

The *selection* component of IDP identifies the *fragments* of its local database (*i.e.,* the sets of objects) that should be transferred to a remote host. The fragments are selected so as to decrease the local load below a threshold while minimizing the wide-area traffic for data movement and queries. At a high level, fragment selection involves partitioning trees. However, previous tree partitioning algorithms [30, 38] tend to incur high computational costs with their $O(n^3)$ (n = number of objects) complexity, and hence prevent a host from shedding load promptly. For example, in IrisNet, using a 3 GHz machine with 1 GB RAM, the algorithms in [30, 38] take tens of minutes to optimally partition a hierarchical database with 1000 objects. Such excessive computational overhead would prevent IDP from shedding load in a prompt fashion. On the other hand, trivial algorithms (*e.g.*, the greedy algorithm in Section 7.1.1) do not yield "good" fragmentation. To address this limitation, we exploit properties of typical query workloads to devise heuristics that provide near optimal fragmentation with $O(n)$ complexity. We call our algorithm POST (*Partitioning into Optimal SubTrees*). As a comparison, under the same experimental setup used with the aforementioned optimal algorithms, POST computes the result in a few seconds. Below we describe the algorithm.

Database Partitioning with POST**.** We use the following terminology in our discussion. For a given host, let G_I denote the set of (Internal) objects in the local database, and G_E denote the set of non-local (External) objects (residing on other hosts) and the set of query sources. Define the *workload graph* (Figure 3) to be a directed acyclic graph where the nodes are the union of G_I and G_E, and the edges are pointers connecting parent objects to child objects in the hierarchy and sources to objects. Under a given workload, an edge in the workload graph has a weight corresponding to the rate of queries along that edge (the weight is determined by locally collected statis-

[3] By a reduction from the Knapsack problem.

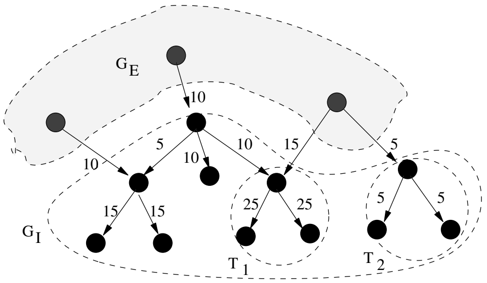


Figure 3: The workload graph of a host. G_I represents the local database and G_E is a set of objects on other hosts. The edges are labeled with the load on the corresponding edge. The circles labeled T_1 and T_2 represent two fragments of the local database of size 3.

tics, as described in Section 6.1). The weight of a node in G_I is defined as the sum of the weights of all its incoming edges (corresponding to its query load) and the weights of all its outgoing edges to nodes in G_E (corresponding to its message load).

For any set T of objects within G_I, we define T's *cost* to be the sum of the weights of nodes in T. The $cut_{internal}$ of T is the total weight of the edges coming from some node in G_I to some node in T, and it corresponds to the additional communication overhead incurred if T were transferred. The $cut_{external}$ is the total weight of the edges coming from some node in G_E to some node in T, and it corresponds to the reduction of load on the node if T were transferred. In Figure 3, the $cut_{internal}$ of T_1 is 10, while the $cut_{external}$ is 15.

Intuitively, to select a good fragment, we should minimize $cut_{internal}$ (achieved by T_2 in Figure 3) so that it introduces the minimum number of additional subqueries or maximize $cut_{external}$ (achieved by T_1 in Figure 3) so that it is the most effective in reducing external load.

To design an efficient fragmentation algorithm for sensing services, we exploit the following important characteristics in their workloads: *A typical query in a hierarchical sensor database accesses all objects in a complete subtree of the tree represented by the database, and the query can be routed directly to the root of the subtree.* This observation is well supported by the IrisLog [7] query trace (more details in Section 7), which shows that more than 99% of user requests select a complete subtree from the global database. Moreover, users make queries on all the levels in the hierarchy, with some levels being more popular than the others. Under such access patterns, the optimal partition T is typically a subtree. The reason is that transferring only part of a subtree T from a host H_1 to another host H_2 may imply that a top-down query accesses objects in H_1 (the top of T) then in H_2 (the middle of T) and then back in H_1 (the bottom of T), resulting in a suboptimal solution.

The above observation enables POST to restrict the search space and run in linear time. POST sequentially scans through all the nodes of the workload graph, and for each node it considers the whole subtree rooted at it. For all the subtrees with costs smaller than the given capacity C, it outputs the one with the optimal objective. The search space is further decreased by scanning the nodes in the workload-graph in a bottom-up fashion, thus considering the lower cost subtrees near the leaves before the higher cost subtrees further up the tree. As mentioned before, in typical settings, POST takes a few seconds to run. Yet, as we will show in Section 7.1.1, the quality of the resulting fragmentation, in practice, is very close to that of the $O(n^3)$ optimal algorithms that take tens of minutes to run.

IDP uses POST with different objective functions for different situations. For example, POST could choose the subtree that maximizes the value of $cut_{external}$. In Figure 3, T_1 denotes such a fragment. Replicating or splitting such fragments would be effective in reducing the external load on the host. Note that this objective tends to choose large fragments since $cut_{external}$ increases with size. Alternatively, POST could minimize the value of $cut_{internal}$. In Figure 3, T_2 denotes such a fragment. Splitting such fragments would minimize any resulting increase in the host hop counts of queries. Unfortunately, this objective tends to pick small fragments which may slow down the load shedding. To avoid this, POST only considers fragments of size greater that $C/2$, where C is a parameter (discussed next) specifying the cost of the fragments POST must select.

Parameters of POST. IDP must choose a value for C to pass to the POST algorithm. C can be chosen to be the smallest amount C_{min} of load whose removal makes the host underloaded. This minimizes the network traffic cost of data placement, but the host may become overloaded again easily. A C far greater than C_{min} would protect the host from overload but increase the overhead and duration of a load shedding event. This choice is equivalent to having two thresholds in the system: the load shedding is triggered when the load is above a *high-watermark* threshold Th_{high}, and at that point the load is shed until the load goes below a *low-watermark* threshold Th_{low}. In addition to these two thresholds, IDP uses a *deletion threshold* Th_{del} that characterizes the lowest load at an object replica that still justifies retaining that replica.

Load Estimation. Each host needs to monitor its load in order to decide when to start shedding load. The load could be estimated using an exponentially-decaying average, L_{avg}, of the instantaneous loads, L_i, on all the objects in the local database. However, after shedding load, L_{avg} overestimates the actual load for a while, because its time-decaying average is unduly inflated by the load just prior to shedding. As a result, unwarranted further shedding could occur repeatedly. A possible fix would be to prevent the host from shedding load until L_{avg} becomes valid; however, this approach would react too slowly to increasing flash crowds. Instead, we adapt a technique used by TCP to estimate its retransmission time out values [20]. We maintain the exponentially-decaying average, V_{avg}, of the value $|L_{avg} - L_i|$, and then estimate the load as $L_{avg} - 2V_{avg}$, until L_{avg} again becomes valid. Our experiments show that this provides a stable estimate of a host's local load.

4.2 Reaction

In IDP, hosts exchange database fragments in order to adapt to changing workloads, using the following four operations:

- *Delete* removes a fragment from the local database. IDP ensures that each object maintains a minimum number of replicas (defined by the service author as a fault tolerance parameter), by not permitting deletes that would violate the specified minimum.
- *Coalesce* merges a fragment from a remote host into the local database. If the local database already contains some part of the fragment, it gets overwritten. *Proactive coalesce* retrieves a fragment and coalesces it. IDP uses this to reduce communication cost and thereby improve performance.
- *Split* deletes a fragment from the local database and sends it to a host where the fragment gets coalesced.
- *Replicate* creates a replica of a fragment and sends it to a host where the fragment gets coalesced.

Naturally, the appropriate action to take at a given time depends on the conditions at the hosts. The following summarizes five host conditions and the corresponding actions taken by IDP:

1. *Read/write load of an object is below Th_{del}*: IDP can delete the object if it has more than the minimum number of replicas.
2. *Read load of the host is above the threshold Th_{high}*: IDP must delete some objects (by applying the above action, as appropriate), replicate a fragment, or split the local database. As discussed in Section 4.1, IDP replicates or splits based on (1) whether POST maximizes $cut_{external}$ or $cut_{internal}$ and (2) the capacity of the target host to handle part or all of the fragment's load.
3. *A local database is too large:* IDP must delete some objects or split the database.
4. *Write load of the host is above Th_{high}*: IDP must replicate a fragment or split the database.
5. *The host is generating a lot of subqueries for an object owned by some other host:* To eliminate these subqueries, IDP proactively coalesces the object. If the host is overloaded as a result, IDP will take additional steps to shed load.

4.3 Fragment Placement

Given a fragment and its load information, IDP must find, through a suitable discovery mechanism, a host that can take the extra load. The simplest approach selects a random host capable of taking the extra load. However, this may require a query to visit multiple hosts, increasing query latency. Therefore, we use two heuristics that exploit the query and data source characteristics to improve the overall performance.

The first heuristic uses our previous observation that subtrees of the global database should be kept as clustered as possible. Thus the host discovery component first considers the *neighboring* hosts as possible destinations for a fragment. A host H_1 is a neighbor of fragment f if H_1 owns an object that is a parent or a child of some object in f. A host H_2, trying to split or replicate a fragment, f, first sees if any of the neighboring hosts of f can take the extra load, and if such a host is found, f is sent to it. If more than one neighboring host is capable of taking the extra load, then the one having the highest *adjacency score* is chosen. The adjacency score of a host H_1 with respect to the fragment f is the expected reduction of read traffic (based on local statistics) if f is placed in H_1. This helps maintain large clusters throughout the system.

When no suitable neighboring host is found, IDP's second heuristic searches for the host capable of taking the extra load that is the closest to the *optimal location* among all such hosts. The optimal location for an object is the weighted mid-point of its read and write sources. We account for the location of sources using the GNP [34] network mapping system. Specifically, each component of the sensing system (sensor, host, or client web interface) has GNP coordinates, and the cartesian distance between hosts is used to estimate the latency between them. If the read and write loads of an object are R and W, and the average read and write source locations are GNP_{read} and GNP_{write}, then the optimal location is given by $GNP_{opt} = \frac{R \cdot GNP_{read} + W \cdot GNP_{write}}{R+W}$. The optimal location of a fragment is the average of the optimal locations of all the objects in the fragment.

4.4 A Simple Run

Intuitively, IDP achieves its goal by letting hosts quickly readjust their loads by efficiently partitioning or replicating objects and by keeping the neighboring objects clustered. We illustrate a simple run of IDP in Figure 4. Rectangular boxes represent hosts and the trees represent their local databases. Initially the global database is placed at one host (1), which experiences high read and write load. To shed write load the host then determines two fragments (one containing object 5 and the other containing objects 3 and 6) for splitting. The fragments are then placed at newly-discovered hosts near the write sources (2). To shed read load, it then determines two more fragments

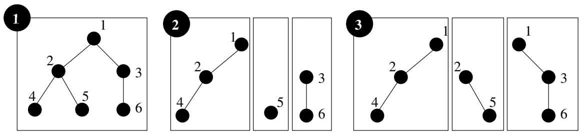


Figure 4: A simple adaptive data placement scenario.

and places them in hosts already owning objects adjacent to the fragments (3).

5 Replica Selection During Queries

While processing a query, if a host's local database does not contain some objects required by the query, subqueries are generated to gather the missing objects. Because the global database is split or replicated at the granularity of individual objects, IDP can route subqueries to *any* of the replicas of the required objects. Two simple replica selection strategies are *proximity selection* that selects the closest replica and *random selection* that selects a random replica. However, as we will show in Section 7, these simple strategies are not very effective for sensing services. For example, in the presence of read-locality, proximity selection fails to shed load from the closest (and the most overloaded) replica. Therefore, we use the following randomized selection algorithm.

Randomized Response-Time Selection. Another possible policy is to select the replica with the smallest expected response time (network latency plus the host processing time). Each host models the response time of the replicas using the moving averages of their previous actual response times. Note that when the load of all the replicas are similar (*i.e.*, their host processing times are comparable), the choice is determined by network latencies. This may cause load oscillation of the optimally placed host due to a herd effect [13]. To avoid this oscillation, we instead use the following randomized response-time (RRT) algorithm. Each host, when it contacts a replica r_i, measures its response times and uses them to estimate r_i's expected response time e_i. Suppose a host is considering the replicas in increasing order of response time, $r_1, r_2, \ldots, r_n$. The algorithm assigns a weight w_i to replica r_i according to where e_i falls within the range e_1 to e_n as follows: $w_i = (e_n - e_i)/(e_n - e_1)$. Finally it normalizes the weights to compute the selection probability p_i of r_i as $p_i = w_i / \sum_j w_j$. Note that if all the hosts are similarly loaded, RRT prefers nearby replicas. If the replicas are at similar distances, RRT prefers the most lightly-loaded host. In addition, the randomization prevents the herd effect.

6 Implementation Details

We now sketch our implementation of IDP in IrisNet, including the various modules and parameter settings.

6.1 Implementation in IrisNet

Our implementation of IDP consists of three modules.

The Stat-Keeper Module. This module runs in each IrisNet OA (*i.e.,* at each host). For each object, we partition its read traffic into *internal* reads, in which the object is read immediately after reading its parent on the same OA, and *external* reads, in which a (sub)query is routed directly to the object from the user query interface or another OA. For each object in an OA's local database, the stat-keeper maintains (1) the object's storage requirements, (2) its external read load, defined as an exponentially-decaying average of the rate of external reads, (3) its internal read load, defined as an exponentially-decaying average of the rate of internal reads, (4) its (external) write load, defined as an exponentially-decaying average of the rate of writes, (5) GNP$_{\textit{read}}$, and (6) GNP$_{\textit{write}}$. (GNP$_{\textit{read}}$ and GNP$_{\textit{write}}$ are defined in Section 4.3.)

Moreover, the stat-keeper maintains the current size, aggregate read load, and aggregate write load for each local database. When an OA with multiple local databases (from multiple co-existing services) incurs excessive load, the local database with maximum size or load sheds its load.

The module is implemented as a light weight thread within the query processing component of the OA. On receiving a query or an update, the relevant statistics (maintained in the main memory) are updated, incurring very negligible overhead ($< 1\%$ CPU load).

The Load Adjusting Module. This module is at the center of IDP's dynamic data placement algorithm. It uses different statistics gathered by the stat-keeper module, and when the OA is overloaded, performs the tasks described in Section 4.1 and Section 4.2.

The OA Discovery Module. This module performs the tasks described in Section 4.3 and acts as a matchmaker between the OAs looking to shed load and the OAs capable of taking extra load.

Currently, the first heuristic described in Section 4.3 (selecting neighboring nodes) is implemented in a distributed way. Each OA exchanges its current load information with its neighboring OAs by piggybacking it with the normal query and response traffic. Thus each OA knows how much extra load each of its neighboring OAs can take, and can decide if any of these OAs can accept the load of a replicated or split fragment. The second heuristic (selecting optimally located nodes), however, is implemented using a centralized directory server where underloaded OAs periodically send their location and load information. Our future work includes implementing this functionality as a distributed service running on IrisNet itself.

6.2 Parameter Settings

We next present the tradeoffs posed by different POST parameters and their settings within our implementation. The values are chosen based on experiments with our implementation subjected to a real workload (more details in Section 7.1 and in [33]).

Watermark Thresholds. The value of the high watermark threshold Th_{high} is set according to the capacity of an OA (*e.g.,* the number of objects per minute that can be processed by queries). The value of the low watermark threshold Th_{low} poses the following tradeoff. A smaller value of Th_{low} increases the data placement cost (and the average response time) due to increased fragmentation of the global database. However, a larger value of Th_{low} causes data placement thrashing, particularly when the workload is bursty. Our experiments in [33] show that a value of $Th_{low} = \frac{4}{5} Th_{high}$ provides a good balance among different degrees of burstiness.

Deletion Threshold. The deletion threshold Th_{del} for an object presents additional trade-offs. A small Th_{del} keeps replicas even if they are not required. This increases the local database size and hence the OA processing time.[4] A high Th_{del} can remove replicas prematurely, increasing the data placement cost. Based on our experiments, we set the threshold such that an object gets deleted if it receives no queries in the past 30 minutes (and the object has more than the minimum number of replicas).

We use these parameter settings in our experiment evaluation, discussed next.

7 Experimental Evaluation

Our evaluation consists of the following two steps. First, we use a real workload with our deployment of IDP (within IrisNet) running on the PlanetLab to study how IDP's different components work in practice. Then, we use simulation and synthetic workloads to understand how sensitive IDP is to different workloads.

7.1 Performance in Real Deployment

In this section, we evaluate IDP using IrisLog, an IrisNet-based Network Monitor application deployed on 310 PlanetLab hosts. Each PlanetLab host in our deployment runs an IrisNet OA. Computer programs running on each PlanetLab host collect resource usage statistics, and work as sensors to report to the local OA once every 30 seconds. We drive our evaluation by using IrisLogTrace, a real user query trace[5] collected from our IrisLog deployment on the PlanetLab from 11/2003 to 8/2004 (Table 2). Because the trace is relatively sparse, we replay this trace with a *speedup factor*, defined as the ratio between the original duration of the trace and the replay time. For the experiments requiring longer traces than what we have, we concatenate our trace to construct a longer trace. At the beginning of each experiment, the global database is randomly partitioned among a small group of hosts. As the workload is applied, IDP distributes the database among the available hosts, with the data placement costs decreasing over time. We say that the system has reached *initial steady state* if, under our reference workload, no data relocation occurs within a period of 2 hours. We experimentally ensured that this choice of period was reasonable. In all our experiments, we measure the cost of data placement from the very beginning of the experiment, while all other metrics (*e.g.*, response time) are measured after the system has reached its initial steady state, at which point the specific workload to be studied is started.

Total queries	6467 (100%)
Queries selecting a complete subtree	6409 (99.1%)
Queries selecting all the data	401 (6%)
Queries selecting a country	1215 (19%)
Queries selecting a region	3188 (49%)
Queries selecting a site	1469 (23%)
Queries selecting a host	136 (2%)
Queries not selecting a complete subtree	58 (0.9%)

Table 2: IrisLogTrace: Trace of user queries from 11/10/2003 to 8/27/2004 for the IrisLog application deployed on 310 PlanetLab hosts.

Figure 1 demonstrates the overall performance of IDP. As shown, IDP helps IrisNet maintain reasonable response times even under a high flash-crowd. Without IDP, IrisNet becomes overloaded and results in high response times and timeouts.

We now evaluate the individual components of IDP.

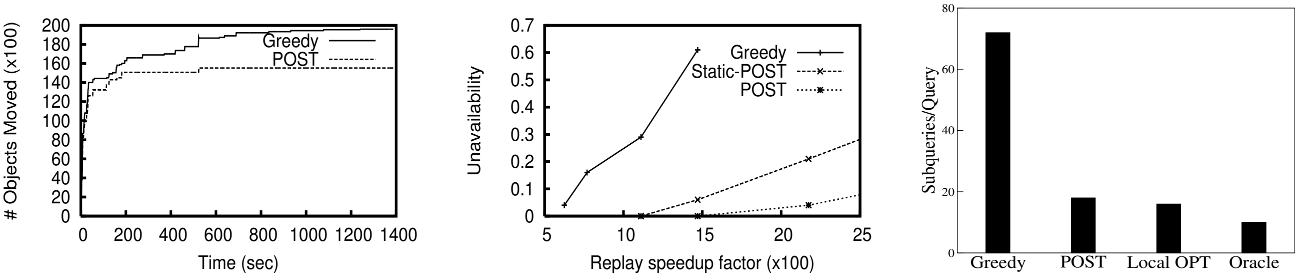
7.1.1 Partitioning Heuristics

This section evaluates POST and compares it with three hypothetical algorithms: GREEDY, LOCALOPT and ORACLE. In GREEDY, overloaded hosts evict individual objects in decreasing order of their loads. As a result, hosts make decisions with finer granularity, but do not try to keep the hierarchical objects clustered. In LOCALOPT, each host partitions its XML fragment using an optimal tree partitioning algorithm [30] with $O(n^3)$ complexity. However, because each invocation of LOCALOPT takes tens of minutes of computation time, we do not use it in our live experiments on the PlanetLab. ORACLE is an offline approach that takes the entire XML database and the query workload and computes the optimal fragmentation (again, using the algorithm in [30]). ORACLE cannot be used in a real system and only serves as a lower bound for comparison purposes.

Our evaluation consists of two phases. In Phase 1, we

[4] In existing DOM-based XML database engines, query processing time increases with the local database size.

[5] Available at `http://www.intel-iris.net/ilog-trace/`.



(a) Fragmentation cost (b) Unavailability due to overload (c) Network overhead per query

Figure 5: Comparing different fragmentation algorithms.

fragment the IrisLog database by using all but the last 1000 queries of IrisLogTrace as warm-up data. To do this within a reasonable amount of time, we perform this phase on the Emulab testbed [6] on a LAN. This enables us to use a large speedup factor of 1200 and finish this phase in a short period of time. We also assign each Emulab host its GNP coordinates that can be used to model the latency between any two emulated hosts if required.

In Phase 2 of our experiment, we place the fragments created in Phase 1 in our PlanetLab deployment and inject the last 1000 queries from IrisLogTrace.

To understand the advantage of POST's adaptive fragmentation, we also use Static-POST which, under all speedup factors, starts from the same warm-up data (generated from Phase 1 with a speedup factor of 1200) and does not further fragment the database during Phase 2.

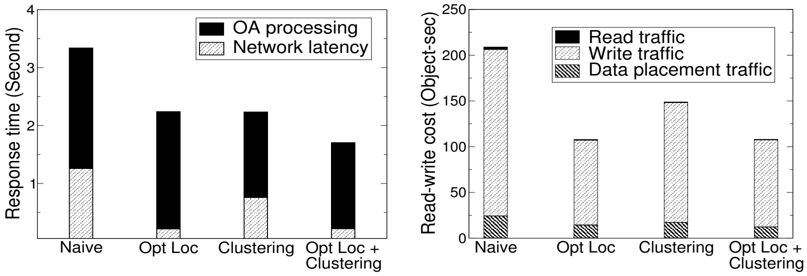
Figure 5(a) plots the cumulative overhead of the fragmentation algorithms over time during the warm-up phase of the experiment. The graph shows that the amount of fragmentation decreases over time, which means that our load-based invocations of the algorithms do converge under this workload. GREEDY incurs higher overhead than POST because GREEDY's non-clustering fragmentation increases the overall system load which makes the hosts fragment more often. We do not use LOCALOPT or ORACLE in this experiment due to their excessive computation overhead.

Replaying IrisLogTrace with a high speedup factor on IrisLog overloads the hosts. Because of this overload, the response of a typical query may return only a fraction of the requested objects. The remaining objects reside in overloaded hosts that fail to respond to a query within a timeout period (default 8 seconds in IrisNet). We define the unavailability of a query to be the fraction of the requested objects not returned in the answer. Figure 5(b) shows the average unavailability of IrisLog under different fragmentation algorithms and under different speedup factors. GREEDY is very sensitive to load and suffers from high unavailability even under relatively smaller speedup factors (*i.e.*, smaller load). This is because GREEDY produces suboptimal fragments and overloads the system by generating a large number of subqueries per query (Figure 5(c)). POST is significantly more robust and it returns most objects even at a higher speedup factor. The effectiveness of POST comes from its superior choice of fragments, which generate a near optimal number of subqueries (as shown in Figure 5(c)). The difference between Static-POST and POST in Figure 5(b) demonstrates the importance of adaptive load-shedding for better robustness against a flash-crowd-like event.

7.1.2 Data Placement Heuristics

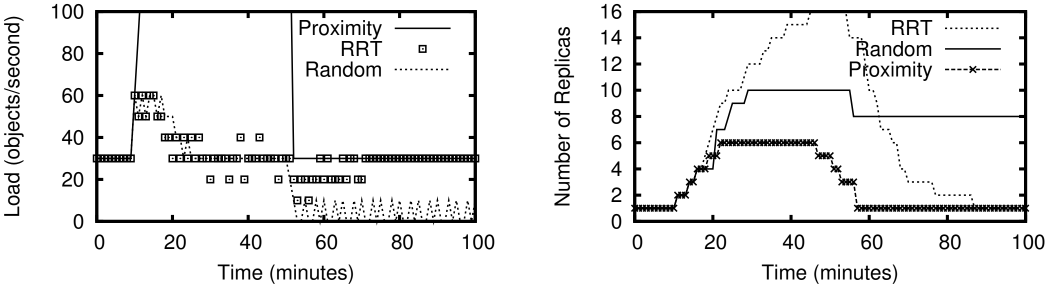
We now evaluate the effectiveness of different heuristics described in Section 4.3 for choosing a target host for a fragment. The heuristics are compared to the naive approach of choosing any random host that can accept the extra load. We start this experiment by placing the fragments generated by Phase 1 of the previous experiment and then replaying the last 1000 queries from IrisLogTrace with a moderate speedup factor. In different settings, we use a different combination of placement heuristics. We represent the cost of a data transfer using the metric *object-sec*, which is the product of the number of objects transferred and the latency between the source and the destination. Intuitively, this metric captures both the amount of data transferred and the distance over which it is transferred. The results of this experiment are shown in Figure 6. As mentioned before, response times and read/write traffic are computed only in the second phase of the experiment (done on the PlanetLab), while the data placement cost includes the cost incurred in both phases.

The first heuristic `Opt Loc`, choosing a host near the optimal location, reduces network latency and traffic, as shown in Figure 6. The second heuristic `Clustering`, choosing a neighboring host, reduces the number of hops between hosts. This is due to the fact that the heuristic increases the average cluster size (25% increase on average in our results). To achieve the best of both heuristics, we use them together as follows. Because host processing time dominates network latency, we first try to find a neighboring host capable of taking the extra load; only if no such host exists, we use the second heuristic to find a host near the optimal location. This has the following interesting implication: because the read and write sources



(a) Average query response time (b) Average network overhead

Figure 6: Effect of data placement heuristics on query response time and network overhead.



(a) Current Load (b) Replica Count

Figure 7: Change in load of a host and replica count of a fragment after a burst of read traffic.

of the objects that are adjacent in the global database are expected to be nearby (because of read-write locality of sensor data), first placing an object in the optimal host and then placing the adjacent objects in the same host automatically places the objects near the optimal location. As shown in the graph, the combination of the two heuristics reduces the total response time and read network traffic, with a slight increase in write network traffic. Also note that the data placement cost is many times smaller than the read/write cost, showing that IDP has little overhead.

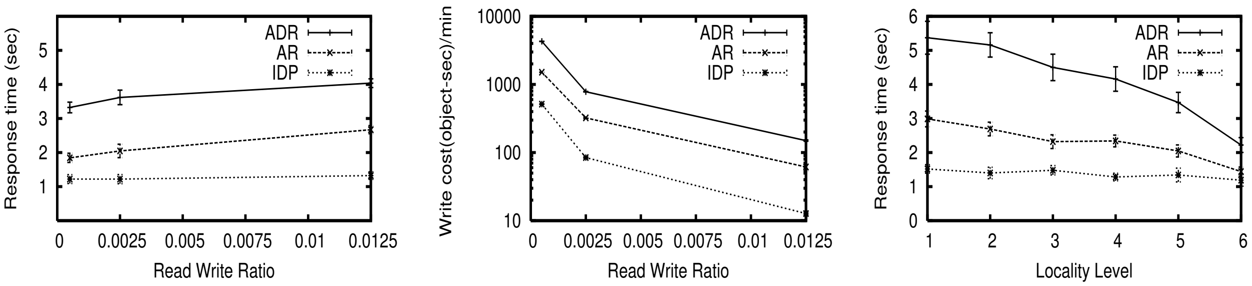
7.1.3 Replica Selection Heuristics

To demonstrate the effect of different replica selection heuristics, we do the following experiment. We place a fragment in a host H on the PlanetLab. We start issuing queries from a nearby location so that H is the optimal host for the reader. At time $t < 10$, the load of the host is below the threshold (= 35, the maximum load H can handle). At $t = 10$, we increase the query rate by a factor of 10. This flash crowd exists until $t = 50$, after which the read load drops below the threshold. As the query rate increases, the host creates more and more replicas. In different versions of this experiment, we use different heuristics to select a replica to which the query is sent. We plot the load of the host H and the number of replicas at different times in Figure 7.

With *proximity-based replica* selection, host H becomes overloaded and creates new replicas. However, as expected, all the load shedding efforts are in vain because all the queries continue to go to H. As the figure shows, its current load promptly goes up (the peak value 400 is not shown in the graph) while all the newly created replicas remain unused and eventually get deleted.

With *random* selection, the load is uniformly distributed among the replicas. As shown in Figure 7, as the load increases, fragments are replicated among the hosts until the load of each host goes below the threshold ($t = 20$). However, after the flash crowd disappears ($t = 50$), all the replicas still get the same equal fraction of the total load. In this particular experiment, at $t = 51$, the average load of each replica is slightly below the threshold used for deletion of objects. Because of the randomized deletion decision, two replicas get deleted at $t = 57$. This raises the average load of each replica above the deletion threshold. As a result, these replicas remain in the system and consume additional resources (Figure 7(b)).

Randomized response-time (RRT) selection shows the same good load balancing properties of random selection. Moreover, after the flash crowd disappears, the replica at the optimal location gets most of the read load. Thus, the other replicas become underloaded and eventually get deleted. Furthermore, because RRT incorporates estimated response times of replicas in its replica selection, it significantly reduces the query response time (by around 25% and 10% compared to proximity-based and random selection, respectively). This demonstrates the effectiveness of RRT for sensing services.



(a) Average response time with different read rates

(b) Average write traffic with different write rates

(c) Average response time varying R/W proximity

Figure 8: Effect of varying the Read rate, the Write rate, and the R/W proximity.

7.2 Sensitivity Analysis with Simulation

We now use simulation to understand how IDP performs as the different parameters of the workload change. Our IrisNet simulator enables us to use a larger setup (with 10,000 hosts) than our PlanetLab deployment of IrisLog. Each simulated host is given multi-dimensional coordinates based on measurements from the GNP [34] network mapping system, and the cartesian distance between hosts estimates the latency between them. We use the GNP coordinates of the 867 hosts from [34]. Because the simulated hosts and sensors outnumber the available GNP coordinates by two orders of magnitude, we emulate additional GNP coordinates by adding small random noise to the actual GNP coordinates. To approximate the geographic locality of sensors, we make sure that sensors that report to sibling leaf objects are assigned nearby GNP coordinates (*e.g.*, the sensors that report to block 1, 2, and 3 of Oakland have nearby coordinates). The simulator does not model the available bandwidth between hosts in the network. However, most messages are small and, therefore, their transmission times will be limited more by latency than by bandwidth. Unless otherwise stated, we use IrisLogTrace with an appropriate speedup factor for our read workload. For our write workload, we assume that every sensor reports data once every 30 seconds.

We compare IDP with the following existing adaptive data placement approaches. In the first approach, *aggressive replication* (AR), a host maintains replicas of the objects it has read from other hosts. The second approach, *adaptive data replication* (ADR), uses the algorithm proposed in [41]. Although ADR provably minimizes the amount of replication compared to aggressive replication, it incurs a higher communication cost. This is because, for general topologies, it requires building a spanning tree of the hosts and communicating only through that tree, which is crucial for the optimality of its data placement overhead. In all the schemes, if a host has insufficient capacity to store the new objects, it replaces a randomly-chosen least-recently-used object.

Note that comparing IDP with ADR or AR is not an apples-to-apples comparison, since ADR and AR were designed for different workloads. Still, our experiments provide important insight about their performance under a sensing workload. For lack of space, we present only a few interesting results. More details of the simulation setup and additional results can be found in [33].

Read and Write Rate. Figure 8(a) shows that for a given write rate ($16,000$ objects/min),[6] the average query response time of IDP increases slightly with the read rate. This is because of the increased fragmentation due to higher system load. Surprisingly, even with AR, the response time increases with the read rate. The reason behind this counter-intuitive performance is that AR is oblivious to data clustering. ADR has the worst response time because it communicates over a spanning tree, ignores data clustering, and does not consider the hierarchical data access patterns. At all the read rates shown in Figure 8(a), IDP performs significantly better than the other algorithms. Similarly, our experiments have shown that IDP incurs significantly less read traffic than these algorithms across all the read rates.

In Figure 8(b), we vary the write rate with a constant read rate (40 objects/min). As expected, the write cost increases as the write rate increases. Although both ADR and IDP create similar numbers of replicas, ADR incurs a higher write cost because updates are propagated using a spanning tree overlay and, therefore, they traverse longer distances. AR incurs more cost than IDP, because it fails to capture the locations of write sources while placing replicas.

Read-Write Proximity. Read-write proximity reflects the physical distance between the reader and the writer. As mentioned in Section 2.1, we model it by the lowest level (called a locality level) in the hierarchy they have in common. Our simulated database has 6 levels, with the root at level 1. Thus, a locality level of 1 signifies the smallest proximity (because the reader and the writer for

[6] To be more accurate, we here use the number of objects touched, instead of the number of updates or queries per minute.

the same data can be anywhere in the whole hierarchy), and a level of 6 means the highest proximity.

Figure 8(c) shows that increasing the locality level reduces the average response time. This is intuitive, because queries with higher levels of locality select fewer objects, and, thus, need to traverse fewer hosts. Because it keeps objects clustered, IDP outperforms the other algorithms. The difference in performance becomes more pronounced as the locality level is low, because such queries select large subtrees from the global database and so the advantages of clustering are amplified.

Representative Services Workload. We have also evaluated IDP, AR, and ADR with a wide variety of synthetic workloads and a heterogeneous mix of them (more details in [33]). By tuning different parameters, we have generated workloads outlined in Table 1. Our results show that IDP outperforms the other two algorithms in both response time and network overhead for all the workloads. AR has worse response time because it ignores data clustering and optimal placement, and worse network overhead because it creates too many replicas. The performance of AR approaches that of IDP when both (1) the write rate is low (*e.g.,* Epidemic Alert) so that eager replication has minimal cost, and (2) the read locality is near the leaf objects (*e.g.,* Parking Finder) so that typical queries select only small subtrees and hence clustering objects is not that crucial. On the other hand, ADR has worse response time and network overhead for all the services, mainly because it communicates over a spanning tree, ignores data clustering, and does not consider the hierarchical data access patterns.

8 Related Work

The issues of data replication and placement in wide-area networks have been studied in a variety of contexts. In this section, we first discuss the relevant theoretical analyses and then present relevant efforts in Web content delivery, distributed file systems, and distributed databases.

Theoretical Background. The off-line data allocation problem (*i.e.*, the placement of data replicas in a system to reduce a certain cost function) has been studied extensively (see [15] for a survey). The general problem has been found to be NP-Hard [42], and several approximate and practical solutions have been proposed [21, 28, 36]. The similar problem of optimally replicating read-only objects has been addressed in the context of content distribution networks [22] and the Web [35].

Other studies [11, 29, 41] have explored the on-line replication of objects in distributed systems. The competitive algorithm in [11] is theoretically interesting, but has little practical application since on every write, all but one replica is deleted. [29] uses expensive genetic algorithms to approximate optimal replication, and requires global information (*e.g.,* current global read and write loads). The ADR algorithm in [41] provides optimal data placement when the network topology is a tree. However, its performance on a general topology is worse than IDP, as shown in Section 7.

Web Content Delivery. Web proxy caches [40] replicate data on demand to improve response time and availability, but deal with relatively infrequent writes and simpler access patterns. The replication schemes used by distributed cache systems and content distribution systems (CDNs) [1, 23] place tighter controls on replication in order to manage space resources even more efficiently and to reduce the overhead of fetching objects. These approaches may be more applicable to read/write workloads. However, they do not support the variable access granularity and frequent writes of sensing services.

Distributed File Systems. Some recent wide-area distributed file systems have targeted supporting relatively frequent writes and providing a consistent view to all users. The Ivy system [32] provides support for the combination of wide-area deployment and write updates. However, while creating replicas in the underlying distributed hash table (DHT) is straightforward, controlling their placement is difficult. The Pangaea system [36] uses aggressive replication along with techniques (efficient broadcast, harbingers, *etc.*) for minimizing the overhead of propagating updates to all the replicas. However, our experiments in Section 7 show that the less aggressive replication of IDP provides better read performance while significantly reducing write propagation overhead. These existing works do not consider the complex access patterns and dynamic access granularity that we consider.

Distributed Databases. Distributed databases provide similar guarantees as file systems while supporting more complex access patterns. Off-line approaches [10, 24] to the replication problem require the complete query schedule (workload) *a priori*, and determine how to fragment the database and where to place the fragments. However, neither [10] nor [24] considers the storage and processing capacity of the physical sites, and [24] considers only read-only databases. Brunstrom et al. [12] considers data relocation under dynamic workloads, but assumes that the objects to be allocated are known *a priori* and does not consider object replication. It also allocates each object independently and thus fails to exploit the possible access correlation among objects. The PIER system [19], like Ivy, relies on an underlying DHT for replication of data. The Mariposa system [37] considers an economic model for controlling both the replication of table fragments and the execution of queries across the fragmented database. While Mariposa's mechanisms are flexible and could create arbitrary replication policies, the work does not evaluate the policies. All of these past efforts are very specific to relational databases, and, therefore, are not directly applicable to the hierarchical database systems that we explore.

9 Conclusion

In this paper, we identified the unique workload characteristics of wide-area sensing services, and designed the IDP algorithm to provide automatic, adaptive data placement for such services. Key features of IDP include proactive coalescing, size-constrained optimal-partitioning, TCP-time-out-inspired load estimators, GNP-based latency estimators, parent-child peering soft-state fault tolerance, placement strategies that are both cluster-forming and latency-reducing, and randomized response-time replica selection. We showed that IDP outperforms previous algorithms in terms of response time, network traffic, and responsiveness to flash crowds, across a wide variety of sensing service workloads.

Acknowledgements. We thank Brad Karp and Mahadev Satyanarayanan for helpful discussions on this work.

References

[1] Akamai web site. http://www.akamai.com/.

[2] Argus. http://www.netcoast.nl/info/argus/argus.htm.

[3] Biomedical security institute. http://www.biomedsecurity.org/info.htm.

[4] Cisco IOS NetFlow - Cisco Systems. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[5] Collaborative adaptive sensing of the atmosphere. http://www.casa.umass.edu/.

[6] Emulab - network emulation testbed home. http://www.emulab.net.

[7] IrisLog: A distributed syslog. http://www.intel-iris.net/irislog.php.

[8] IrisNet: Internet-scale resource-intensive sensor network services. http://www.intel-iris.net/.

[9] Trends in RFID technology at Walmart. http://www.rfida.com/walmartrfid.htm.

[10] Apers, P. M. G. Data allocation in distributed database systems. *ACM Trans. on Database Systems 13*, 3 (1988).

[11] Awerbuch, B., Bartal, Y., and Fiat, A. Optimally-competitive distributed file allocation. In *ACM STOC* (1993).

[12] Brunstrom, A., Leutenegger, S. T., and Simha, R. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In *4th Int. Conf. on Information and Knowledge Management* (1995).

[13] Dahlin, M. Interpreting stale load information. *IEEE Trans. on Parallel and Distributed Systems 11*, 10 (2000).

[14] Deshpande, A., Nath, S., Gibbons, P. B., and Seshan, S. Cache-and-query for wide area sensor databases. In *ACM SIGMOD* (2003).

[15] Dowdy, L., and Foster, D. Comparative models of the file assignment problem. *ACM Computing Surveys 14*, 2 (1982).

[16] Feldmann, A., Greenberg, A., Lund, C., Reingold, N., Rexford, J., and True, F. Deriving traffic demands for operational IP networks: Methodology and experience. *IEEE/ACM Trans. on Networking 9*, 3 (2001).

[17] Franklin, M. J., Jeffery, S. R., Krishnamurthy, S., Reiss, F., Rizvi, S., Wu, E., Cooper, O., Edakkunni, A., and Hong, W. Design considerations for high fan-in systems: The HiFi approach. In *CIDR* (2005).

[18] Gibbons, P. B., Karp, B., Ke, Y., Nath, S., and Seshan, S. IrisNet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing 2*, 4 (2003).

[19] Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S., and Stoica, I. Querying the internet with PIER. In *VLDB* (2003).

[20] Jacobson, V. Congestion avoidance and control. In *ACM SIGCOMM* (1988).

[21] Kalpakis, K., Dasgupta, K., and Wolfson, O. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. on Parallel and Distributed Systems 12*, 6 (2001).

[22] Kangasharju, J., Roberts, J., and Ross, K. Object replication strategies in content distribution networks. In *Int. Web Caching and Content Distribution Workshop* (2001).

[23] Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., and Yerushalmi, Y. Web caching with consistent hashing. *Computer Networks 31*, 11-16 (1999).

[24] Karlapalem, K., and Pun, N. M. Query-driven data allocation algorithms for distributed database systems. In *Database and Expert Systems Applications* (1997).

[25] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (1998).

[26] Kumar, P. R. Information processing, architecture, and abstractions in sensor networks. SenSys'04 Invited Talk.

[27] Kurose, J. Collaborative adaptive sensing of the atmosphere. SenSys'04 Invited Talk. http://www.casa.umass.edu/.

[28] Lam, K., and Yu, C. T. An approximation algorithm for a file-allocation problem in a hierarchical distributed system. In *ACM SIGMOD* (1980).

[29] Loukopoulos, T., and Ahmad, I. Static and adaptive data replication algorithms for fast information access in large distributed systems. In *ICDCS* (2000).

[30] Lukes, J. A. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development 18*, 3 (1974).

[31] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. Tag: A tiny aggregation service for ad hoc sensor networks. In *OSDI* (2002).

[32] Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B. Ivy: A read/write peer-to-peer file system. In *SOSP* (2002).

[33] Nath, S., Gibbons, P. B., Karp, B., and Seshan, S. Adaptive data placement for wide-area sensing services. Tech. Rep. IRP-TR-04-05, Intel Research Pittsburgh, October 2004.

[34] Ng, T. S. E., and Zhang, H. Predicting internet network distance with coordinates-based approaches. In *INFOCOM* (2002).

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

[36] Saito, Y., Karamanolis, C., Karlsson, M., and Mahalingam, M. Taming aggressive replication in the Pangaea wide-area file system. In *OSDI* (2002).

[37] Sidell, J., Aoki, P. M., Sah, A., Staelin, C., Stonebraker, M., and Yu, A. Data replication in Mariposa. In *ICDE* (1996).

[38] Sze, C. N., and Wang, T.-C. Optimal circuit clustering for delay minimization under a more general delay model. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 22*, 5 (2003).

[39] van Renesse, R., Birman, K. P., and Vogels, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. on Computer Systems 21*, 2 (2003).

[40] Wang, J. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review 25*, 9 (1999).

[41] Wolfson, O., Jajodia, S., and Huang, Y. An adaptive data replication algorithm. *ACM Trans. on Database Systems 22*, 2 (1997).

[42] Wolfson, O., and Milo, A. The multicast policy and its relationship to replicated data placement. *ACM Trans. on Database Systems 16*, 1 (1991).

[43] WWWC : The World Wide Web Consortium. XML Path Language (XPATH). http://www.w3.org/TR/xpath, 1999.