IrisNet: An Architecture for Enabling Sensor-Enriched Internet Service

Suman Nath, Yan Ke,
Phillip B. Gibbons, Brad Karp, Srinivasan Seshan

IRP-TR-02-10
December 2002

Intel Research Pittsburgh

IrisNet: An Architecture for Enabling Sensor-Enriched Internet Services

Suman Nath[†,*] Yan Ke[†,*]
Phillip B. Gibbons[*] Brad Karp[*] Srinivasan Seshan[†,*]

[*]*Intel Research Pittsburgh* [†]*Carnegie Mellon University*

Abstract

The proliferation and affordability of webcams and other smart sensors have created opportunities for novel *sensor-enriched* Internet services, which combine traditional data sources with information collected from live sensor feeds. This paper describes the design, architecture, and implementation of IrisNet, the first general-purpose software infrastructure tailored to the unique demands of worldwide sensing services. IrisNet provides service authors with a very high-level abstraction of the underlying system, to ease authoring of new services. For scalability and decreased bandwidth consumption, IrisNet pushes both sensor feed processing and queries close to the sensor nodes. IrisNet provides distributed query processing, data partitioning, caching, load balancing, and replication schemes optimized for sensor-enriched Internet services. This paper reports on experiments with a working IrisNet prototype running two example services that demonstrate the effectiveness of IrisNet's features in achieving scalability and reducing query response times.

1 Introduction

The availability of low-cost sensing hardware, such as CCDs, microphones, infrared detectors, RFID tags, and accelerometers, has improved dramatically over the past several years. Devices that monitor the physical world can now be deployed pervasively on a global scale. Together with the accelerated trend toward ubiquitous Internet connectivity, the opportunity exists for Internet services that combine traditional data sources with information derived from a rich collection of live sensor feeds. Examples of such *sensor-enriched* services include: a Parking Space Finder service, for directing drivers to available parking spots near their destination; a Bus Alert service, for notifying a user when to head to the bus stop; Waiting Time Monitors, for reporting on the queuing delays at post offices, food courts, etc.; a Lost and Found service, for tracking down lost objects; and a Person Finder service, for locating your colleagues or monitoring your children playing in the neighborhood.

Sensor-enriched services often require the processing of live feeds from high-bit-rate, distributed data sources, and the support for rich expressive queries over these data sources. Unfortunately, there are no effective tools today for querying a large-scale, widely-distributed collection of live sensor feeds and extracting the information needed for such services. The current state of Internet services falls far short of providing this functionality. Users are limited to only viewing isolated live webcam feeds (*e.g.,* Panda cam, Traffic cams), querying real-time single-source numerical data feeds (*e.g.,* stock market tickers), querying days-old crawled data as a single queriable unit (*e.g.,* Google), and participating in real-time peer-to-peer file sharing (*e.g.,* FastTrack, Gnutella). This lack of a suitable infrastructure makes authoring and deploying sensor-enriched services an onerous task, as each service author needs to address all aspects of sensor feed processing, distributed query processing, sensing device deployment, load balancing, fault tolerance, etc.

In this paper, we describe the design, architecture, and implementation of IrisNet, a software infrastructure for enabling sensor-enriched services. IrisNet is the first general-purpose shared infrastructure tailored for developing and deploying new sensor-enriched Internet services. IrisNet is designed to address the unique demands of worldwide sensing services, arising from the following prototypical data source and query characteristics:

- **Data source characteristics:** widely distributed, numerical and multimedia data, frequent data updates.
- **Query characteristics:** hierarchically-scoped queries, soft real-time responses, queries over current data and summarized historical trends, queries involving both dynamic sensor data and more static attribute data.

As an illustration of these characteristics, consider the aforementioned Parking Space Finder service. The service requires cameras or other sensors dispersed throughout a metropolitan area. The queries are geographically scoped around the user's destination, and may request the cheapest available spot near that destination (price is a static attribute). The current up-to-date data are the most relevant, but long term availability trends may be exploited (*e.g.,* street parking begins to open up at 5 pm).

There are three key features of IrisNet that address the demands and challenges in authoring and deploying scalable, sensor-enriched Internet services:

Handling rich data sources. IrisNet provides a means for handling rich, high-bit-rate data sources such as video or audio streams. For scalability, IrisNet processes sensor feeds at or near the source. Because the processing is done locally,

network bandwidth consumption is dramatically reduced and the system scales with the number of sensor nodes. At each sensor node, an IrisNet code module, called a *sensing agent* (SA), provides a common runtime environment for services to share the node's sensor feed(s). Service authors upload to the SAs service-specific *senselets*, binary code fragments for extracting the useful information from the sensor feed (*e.g.,* parking space availability from a webcam overlooking street parking).

Distributed query processing. IrisNet provides a distributed database for each service, with efficient distributed query processing. As with the sensor feed processing, the goal is to push the query processing to the data, for increased parallelism and so that only the query answer (not the raw data) is transferred on the network. IrisNet provides query routing, caching, load balancing, and replication schemes tailored to the unique query characteristics discussed above.

Ease of service authorship. IrisNet provides service authors with a very high-level abstraction of the underlying system. IrisNet explores the extent to which a minimal service specification suffices. At its simplest, a service author writes only a browser front end, a senselet, and an XML database schema for the service. IrisNet derives from these inputs all that is needed to run a scalable sensor-enriched Internet service.

We have built a working prototype of IrisNet, and studied its performance. A number of services are being developed on IrisNet, by users beyond the IrisNet developers. In this paper, we focus on two prototype services from widely different domains: a Parking Space Finder (PSF) service and a Network and Host Monitoring (NHM) service. We also discuss initial efforts to address privacy concerns in sensor-enriched services, through a novel scheme for *privacy-protecting image processing*. To our knowledge, there is no prior published work on automated techniques to hide identity in images.

Enabling sensor-enriched Internet services is a rich area of exploration. In this paper, we focus on fixed-location sensors attached to powered sensor nodes, with queries posed using a standard XML query processing language. There is a large body of literature [14, 12, 5, 15, 24, 22, 26, 28, 29] on ad hoc networked collections of severely resource-constrained sensor motes (*smart dust*), deployed in a localized area (a mote cloud). IrisNet can be viewed as complementary to this work, and in fact, IrisNet is useful as a means of connecting up a worldwide collection of mote clouds.

In the following sections, we describe the high-level abstraction IrisNet provides for service authors (Section 2), give an overview of the IrisNet architecture (Section 3), describe the distributed query processing functionality IrisNet provides to all services (Section 4), and then give details of IrisNet's approach to processing rich data sources (Section 5). Next, we provide descriptions of two diverse prototype services (Section 6), and detailed measurements of IrisNet's performance on these services (Section 7). Finally, we place IrisNet in the context of related work (Section 8) and present conclusions (Section 9).

2 Authoring a Service

A key goal in IrisNet is to greatly simplify the task of authoring a sensor-enriched Internet service. A typical author needs to write only the following:

- Code that processes a sensor feed and outputs the desired information in a self-describing format (*e.g.,* code that takes a video feed as input and outputs a list of empty/full parking spots in XML format), called a *senselet*;
- A database schema that describes and organizes the collected sensor readings (*e.g.,* a schema that describes the characteristics of a parking spot, including its hierarchical, geographic location); and
- A user-friendly browser front end that converts user input into queries upon the service database.

Note that the author is not burdened with the details of collecting the sensor readings, processing queries on the widely distributed data, scaling the service to the wide area, enforcing the appropriate privacy policies, protecting the services from failures and attacks, etc.

A centralized, database-centric view. IrisNet is the first system to explore the extent to which most of a service's specification can be derived from its database schema. Moreover, IrisNet allows the service author to have a simple, centralized view of its database. A service author defines and initializes a single (centralized) XML database (in XML parlance, an *XML document*) for the service, to hold both highly dynamic, sensor-derived data and relatively static, descriptive meta data. We use XML for two reasons: self-describing tags and hierarchy. We envision that services will need a heterogeneous and evolving set of data types, aggregate fields, etc. that are best captured using self-describing tags. Furthermore, it seems natural to organize the data hierachically based on geographic/political boundaries (at least at higher levels of the hierarchy), because each sensor device takes readings from a particular physical location. The XML document for the service defines a way of organizing the data hierarchically that is well-suited to the service.[1] As detailed below, the service author provides hints to IrisNet through the use of certain conventions in the schema. The logical hierarchy and the conventions are essential to IrisNet's scalability since they are used to dynamically distribute the document.

[1] Note that IrisNet works with arbitrary hierarchies. In fact, multiple hierarchies can be defined over the same data using XML's IDREF feature, but for simplicity in this paper, the XML document can be viewed as defining a single hierarchy (*i.e.,* a tree). In terms of system performance, each hierarchy will serve the same role as a database index or an overlay tree network, in making queries that match the hierarchy more efficient.

```
/usRegion[@id='NE']/state[@id='NY']/city[@id='New York']/neighborhood
  [@id='Soho' OR @id='Tribeca']/block/parkingSpace[available='yes']
```

Figure 1: A simple XPATH query requesting all available parking spaces in Soho or Tribeca

The queries generated by the front end view the database as centralized. We use the XPATH query language, because it is the most-widely used query language for XML, has good query processing support, and supports a rich set of query predicates. Figure 1 gives a simple example of an XPATH query on a hierarchical schema for parking spaces consisting of `usRegion`, `state`, `city`, `neighborhood`, `block`, and `parkingSpace`. Predicates at each level of the hierarchy are given in square brackets. This query requests all available parking spaces in two neighborhoods in New York.

Senselets. Each senselet the service author writes converts a live sensor feed into updates to the sensor database, which again is viewed as centralized. To allow maximum flexibility, senselets can be any Linux binary. For efficiency of sensor feed processing, however, we encourage authors to use our specialized sensor feed processing libraries (Section 5) when writing senselets. Similarly, to minimize the demands on the network, we encourage authors to aggressively filter high-bit-rate sensor feeds down to a smaller amount of derived data (such as a list of all available parking spots).

Resource discovery. In order to create a service, IrisNet must select a number of sensor devices to provide input data and a number of hosts to store the service database. IrisNet requires that service authors specify the important properties of the desired sensor feeds within the leaf nodes of their respective database schemas. Currently, sensor feeds are specified by unique IDs in the leaf nodes of the database schema. Extensions to a content-based sensor description, leveraging recent network service location research [6, 19, 39], are beyond the scope of this paper. As for database hosts, IrisNet typically selects a host that is relatively near the source of the sensor data (to minimize network overhead) and that has relatively low load (to reduce query processing latency). However, the service author can provide hints within the database schema to guide this selection.

The above description highlights the simple tasks that service authors need to perform to create a service using IrisNet. In the following sections, we describe the details of how IrisNet uses the information provided by the author to perform the tasks of query processing, indexing, networking, caching, load balancing, and resource sharing.

3 A Two-Tier Architecture

The IrisNet architecture is implemented using common off-the-shelf hardware and operating systems in combination with our custom user-level software. IrisNet uses a two-tier architecture, comprised of two different types of software modules, sensing agents (SAs) and organizing agents (OAs),

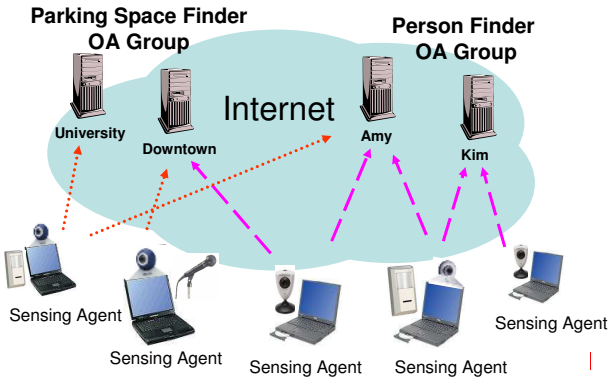


Figure 2: IrisNet Architecture

as depicted in Figure 2. All modules execute on PC-class, Internet-connected hosts.[2] We discuss each tier in turn.

SA Architecture. Each SA host is directly connected to one or more sensing devices. The types of devices can range from webcams and microphones to motion detectors and pressure gauges. An SA running on the host provides a common runtime environment for services to share the host's sensor feed(s). The total collection of SAs create a sensor feed infrastructure that is shared by all services. An SA may transmit data to many OAs for the same or different services (Figure 2).

As discussed in Section 2, service authors write senselets that perform service-specific processing and filtering of sensor feeds. SAs passively await senselets, which are uploaded by the OAs. Multiple senselets can run within the same SA environment on a host; this facilitates work sharing (see Section 5).

OA Architecture. OAs are organized into groups, one group per service. A group of OAs creates the distributed database and query processing infrastructure for a specific service (such as the Parking Space Finder service or the Person Finder service in Figure 2). Each OA participates in one sensor-enriched service (a single physical machine may host multiple OAs). Each OA has a local database for storing sensor-derived data; these local databases combine to constitute an overall service database. Using multiple OAs, *i.e.,* essentially using a distributed database, is necessary to support the high update rates that may result from sensor readings. Using separate OA groups allows each service author to tailor the database schema to the service, and to facilitate service-workload-specific caching and load balancing.

In this paper, we consider environments in which SA and OA hosts are deployed by a small collection of trusted entities similar to ISPs. However, the IrisNet architecture can accommodate other deployment types as well.

[2] We refer to the software modules as the SA and OA, and refer to the physical machines as the SA host and OA host.

4 Distributed Query Processing

Recall from Section 2 that IrisNet provides service authors with a very high-level abstraction of the underlying system, in which the XML database is completely centralized. In this section, we describe how IrisNet takes a service's XML document and creates a *distributed* service database, which supports updates by the service's senselets and answers queries generated by the service's front end. Although distributed databases are a well-studied topic (e.g., [37, 18, 35, 7, 25, 8, 10, 32]), no work prior to ours shows how to dynamically distribute an XML document or how to perform query processing over a distributed XML document.

4.1 Distributing the Database

In order to adapt to query and update workloads, IrisNet can dynamically partition the sensor database among a collection of OAs. The database schema provided by the service author includes special `id` attributes, whose value is a short name that makes sense to include in a user query (*e.g.,* New York). These attributes define potential split points for partitioning the database, *i.e.,* the minimum granularity for a partitioned unit. Unlike approaches with a fixed granularity (*e.g.,* a file block), the granularity can vary widely within the same service database. IrisNet permits an OA to own *any* subset of the nodes in the hierarchy (including non-contiguous subsets), as long as ownership transitions occur at split points and all nodes are owned by exactly one OA. (OAs can also cache/replicate data owned by other OAs, as discussed below.) We envision that service authors will liberally include `id` attributes in their schema, to give IrisNet maximal flexibility in partitioning, while IrisNet will typically partition the document into far fewer actual partitions.

Global naming is achieved by requiring that (1) the `id` of a split node is unique among its siblings with the same parent (*e.g.,* there can be only one `city` whose `id` is `New York` among the `NY` state node's children), and (2) the parent of a non-root split node is also a split node. Thus, the sequence of node names and `id`s on the path from the root uniquely identifies a split node. An important feature of our design is that because the values of `id` attributes make sense in XML queries, we can extract any global name we need directly from the query itself! From the query in Figure 1, for example, we can extract the global name `city-NewYork.state-NY.usRegion-NE` for the New York node by concatenating the name and `ID` pairs from the query. We are not aware of any previous global naming scheme that leverages XML query languages in this manner.

Each OA registers with DNS [31] each split node that it owns, where the registered name is the split node's global name appended with the name of the service and our domain name. DNS provides a simple way for any node to contact the owner of a particular split node of the database. It is the only mapping from the logical hierarchy to physical IP addresses in the system, enabling considerable flexibility in mapping nodes in the document to OAs, and OAs to physical machines. This permits the system to scale to as many machines as needed, each operating in parallel, in order to support large data volumes and high update frequencies. In IrisNet, an OA keeps track of both the computational load on its host machine and the frequency of queries (and of updates, if it is a leaf OA) directed to each split node it owns. An overloaded OA splits off part of the document it owns to a lightly loaded machine (found using the resource discovery mechanism outlined in Section 2). Conversely, a very lightly loaded OA is collapsed into its parent. This simple policy targets reasonable response times, not optimal load balance, and more sophisticated load balancing policies could be used instead (*e.g.,* [33, 17, 16]). Ownership changes are done on-the-fly by exploiting our replication mechanism (Section 4.4) during the transition.

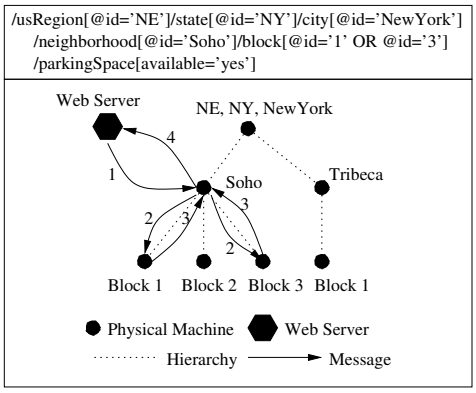


Figure 3: Top: An XPATH query. Bottom: A mapping of logical nodes to seven machines, and the messages sent to answer the query (numbers depict their relative order).

4.2 Answering Queries

Due to our dynamic partitioning, providing fast and correct answers to user queries is quite challenging. The goals are to route queries directly to the nodes of interest to the query, to take full advantage of the data stored at each OA visited, and to pass data between OAs only as needed. We show how IrisNet addresses each of these goals.

An XPATH query selects data from a set of nodes in the hierarchy. In IrisNet, the query is routed directly to the lowest common ancestor (LCA) of the nodes potentially selected by the query. IrisNet uses a simple parser to scan the query for its maximal sequence of `id`-only predicates (*e.g.,* up to the New York node for the query in Figure 1). This constitutes a non-branching path from the root of the hierarchy to the node, such that the query answer will come from the subtree rooted at that node. In the common case where the `id`-only chain ends at the place where the query no longer selects just a single path, the node is the desired LCA for that query. In Figure 1, for example, the New York node is the LCA because the query selects from two of the node's

children (Soho and Tribeca). The simple parser constructs the DNS name for the node (as detailed above), performs a DNS lookup to get the IP address of the machine hosting the New York node, and routes the query to the OA. We call this the *starting point* OA for the query. The key point is that for an arbitrary XPATH query posed anywhere in the Internet, IrisNet can determine where to route the query with just a DNS lookup: no global or per-service state at the web server is needed to produce the DNS name of the LCA. Also, the root of the hierarchy is not a bottleneck, because queries are routed directly to the LCA, which, for the typical hierarchically-scoped query, is far down in the hierarchy.

Upon receiving a query, the starting point OA queries its portion of the overall XML document and evaluates the result. However, for many queries, a single OA may not have enough of the document to respond to the query. The OA determines which part of a user's query can be answered from the local document (discussed below) and where to gather the missing parts (extracting the needed global names from the document). The OA looks up the IP addresses of the other OAs to contact and sends subqueries to them. These OAs may, in turn, perform a similar gathering task. Finally, the starting point OA collects the different responses and the combined result is sent back to the user. For the example in Figure 3, the Soho OA receives the query from the web server, sends subqueries to the Block 1 and Block 3 OAs, who each return a list of available parking spaces, to be combined at the Soho OA and returned to the user.[3]

While the above recursive process is straightforward at a high level, the actual details of getting it to work are rather complicated. The crux of the problem is to determine which part of an XPATH query answer can be extracted from the OA's local document (a fragment of the overall service document). Simply posing the query to the XML database will not work because the fragment may be missing nodes in the document that are part of the answer or fields in the document that are needed to correctly evaluate a predicate. In short, existing query processors are not designed to provide negative information about missing parts of an answer. Our solution to this problem is to tag the nodes in a fragment with status information that indicates various degrees of completeness and to maintain tagging/partitioning invariants (such as partitioning only at split points). Then, we convert the XPATH query into an XSLT program that walks the OA's XML document tree and handles the various tags appropriately. Complete details on the approach appear in [1].

4.3 Caching and Data Consistency

Like many other distributed services, it is obvious that there will be a great deal of locality in the user requests to a sensor-enriched service. For example, in a PSF service, there are likely to be many more queries about downtown parking than rural/suburban parking, as well as spikes in requests for near the stadium on game day. To take advantage of such patterns, OAs may cache data from any query gathering task that they perform. Subsequent queries may use this cached data, even if the new query is not an exact match for the original query. During the gathering task, IrisNet generates subqueries that fetch partitionable units; in this way, its query processing mechanisms (described above) work for arbitrary combinations of owned and cached data.

Due to delays in the network and the use of cached data, answers returned to users may not reflect the most recent data. A query may specify consistency criteria indicating its tolerance for stale data. For example, a PSF service can specify a decreasing tolerance for stale data as a user approaches her destination. We store timestamps along with the cached data, indicating when the data was created, so that an XPATH query specifying a tolerance is automatically routed to the data of appropriate freshness.

4.4 Fault Tolerance and Replication

While a hierarchical database organization enables the scalable processing of rich queries, it is more susceptible to failures. For example, the failures of nodes high up in the hierarchy can cause significant portions of the database to become unavailable. Other hierarchical systems, such as DNS [31], rely on replication to tolerate such failures. However, DNS relication is relatively simple since DNS records are rarely updated. Unfortunately, sensor readings change frequently making replication more difficult in IrisNet.

IrisNet achieves fault tolerance through two mechanisms. First, queries are first sent to the LCA OA, and hence failure of the OAs containing nodes above the LCA node in the hierarchy does not affect the query processing. Second, IrisNet replicates nodes in the logical hierarchy on multiple OAs. This replication requires us to relax the restriction that there be a single owner for each node in the hierarchy. We call these multiple owners *primary replicas.* All data on the primary replicas must be kept strongly mutually consistent. When the DNS name for a node is resolved, it returns the addresses of all primary replicas. Typically, primary replicas are assigned to nodes that are optimal in placement. For example, a leaf node primary replica is likely to be near its associated SAs.

IrisNet also uses secondary replicas. Secondary replicas only maintain a weakly consistent copy of the data corresponding to the node. Secondary replicas are often placed far from the primary replicas to avoid of simultaneous failures of these hosts. We store the location of these secondary replicas separately from the primary replicas by using an alternative trailing domain name. Querying this alternate name retrieves the address of all primary and secondary replicas for the node. One other important distinction is that the DNS records for the alternate name are given a TTL of 0 while those for the normal name are given a TTL of 10 minutes.

[3] This is a simple example in which the children of the Soho node reside on different machines, but in general, a subquery may be sent to any descendant in the hierarchy, or even a descendant of an ancestor, depending on the query and how the document is fragmented.

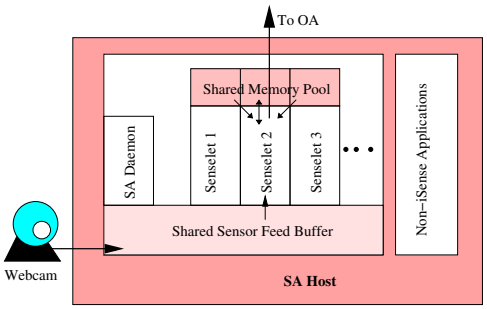


Figure 4: Execution environment in sensor host

The low TTL for the alternate DNS records ensures that an up-to-date list of replicas is retrieved during the critical periods when the secondary replicas are used.

When a host attempts to route a query to an OA, it may fail in a number of ways: the TCP connection setup may fail, the query may timeout, the contacted host may no longer be responsible for the node, etc. Upon any failure, the host reissues the query to the next primary replica. If all primary replicas fail, the host performs a DNS lookup on the alternate node name and queries any newly discovered replicas in a similar fashion. The current IrisNet prototype does not spawn new replicas when a failure is detected.

5 Handling Rich Data Sources

Because rich sensor feeds such as video streams and audio streams are potentially high-bit-rate, IrisNet faces a fundamental scaling challenge: how to make numerous distributed, rich sensor feeds available to users while minimizing the use of network bandwidth. As we state in the introduction, central to the IrisNet architecture is the aggressive filtering of raw data sources *at the node where the data originate,* to achieve drastic bandwidth savings in their transmission across the network. In this section, we describe in detail how IrisNet accomplishes this filtering efficiently.

5.1 Senselets: Service-Specific Filtering

IrisNet exploits the observation that *service-specific filtering* of sensor feeds yields the greatest reduction in their data rates. For example, a PSF service may use a 10 frame-per-second (fps) video stream from a camera pointed at a parking lot as its input, but the service need only know which parking spaces are full and which are empty. Running filtering code tailored to the PSF service on the host to which the camera is attached will reduce the sensor feed from a stream of 10 fps video to a series of vectors of { full, empty } bits, one per parking space.

IrisNet's senselets perform service-specific filtering. The IrisNet execution environment on sensor hosts provides sensor feed processing libraries with well-known APIs to be used by senselets. We expect typical senselets to be sequences and compositions of these well-known library calls, such that the bulk of the computation conducted by a senselet occurs inside the processing libraries. Note that we do not *require* that a senselet use these libraries; they are merely a convenience to authors, in that they represent a predictable development platform. Using well-known library calls plays a role in efficient sharing of CPU resources by distinct senselets, however, as discussed in Section 5.2.

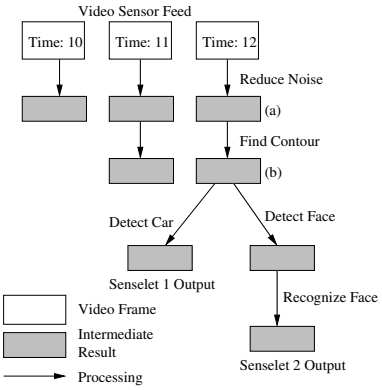


Figure 5: Computation flows for two senselets. The complete graph is shown for the video frame at time 12. A few computation results for previous frames are also shown.

Figure 4 shows the execution environment in an IrisNet sensor host. A sensor host receives one or more raw sensor feeds from directly attached sensors. One instance of the IrisNet (SA), the code that manages filtering of sensor feeds, runs on each sensor host as a root-privileged, user-level process. Each senselet runs as a separate user-level process.

One circular shared-memory buffer for each locally attached sensor is mapped into the address spaces of all senselets. Raw sensed data are periodically written into this shared memory, so that all senselets may read them without incurring a memory-to-memory copy. We discuss the data sharing model for senselets in more detail in Section 5.2.

A typical senselet is written in a way to achieve *soft* real-time behavior: the senselet uses periodic deadlines for completing computations, but associates a *slack time*, or tolerance for error, with these deadlines. A senselet periodically reads a sensor feed from shared memory, processes it, sends output information to an OA, and sleeps until the next deadline. Senselets dynamically adapt their sleep times under varying CPU load to target finishing their next round of processing within the window defined by the next deadline, plus-or-minus the slack time.

5.2 Cross-Senselet Sharing

One sensor feed may be of interest to multiple different IrisNet services: *e.g.,* a video feed in a particular location may be used in one service to monitor parking spaces, and in another to track passersby in the same visual field. To make sensor feeds maximally available to users of heterogeneous services, IrisNet must support *sharing* of sensor feeds among

multiple senselets. We expect image processing primitives (*e.g.,* color-to-gray conversion, noise reduction, edge detection, *etc.*) to be reused heavily across senselets working on the same video stream. If multiple senselets perform very similar jobs (*e.g.,* tracking different objects), *most* of their processing would overlap. For example, many image processing algorithms for object detection and tracking use background subtraction. Multiple senselets using such algorithms need to continuously maintain a statistical model of the *same* background [13].

Consider the two senselets whose computation graphs are shown in Figure 5. Senselet 1 finds images of cars in a video stream, while senselet 2 finds images of human faces in the same video stream. Note the bifurcation at time 12, step (b) between senselets 1 and 2; their first two image processing steps, "Reduce Noise" and "Find Contour," are identical, and computed over the same raw input video frame.

We wish to enable senselets like the pair shown in Figure 5 to cooperate with one another. In the figure, one senselet could share its intermediate results (marked as (a) and (b)) with the other, and thus eliminate the computation and storage of redundant results by the other.

Each senselet owns a shared-memory region where it has read and write permissions, and has read-only access to other senselets' shared-memory regions. These mappings are enforced by the SA. IrisNet stores senselets' intermediate results in shared memory at run time. This technique is quite similar in spirit to the memoization done by optimizing compilers, where the result of an expensive computation is stored in memory for re-use later, without repetition of the same computation. One key difference between results sharing in IrisNet and traditional memoization is that IrisNet shares intermediate results *between senselets*, whereas memoization shares intermediate results between exactly matching function calls within a *single running executable*. Two different senselet binaries may overlap in only a portion of their computation, and results should only be reused if they were computed using *identical* functions on identical input data. Moreover, senselets are soft real-time processes that act on time series, and may not be willing to use pre-computed results derived from stale raw data.

If a senselet's author uses the libraries for sensor feed processing provided by the IrisNet infrastructure, most of that senselet's time will be spent within these libraries' functions. IrisNet uses *names* of sensor feed processing API calls to identify commonality in execution. Senselets name intermediate results using their *lineage*, which is a hashed encoding of the entire sequence of library function calls along the path from an original sensor feed to a result in the computation graph. This encoding preserves the ordering on non-commutative function calls. The libraries implement results sharing.[4] Each library function begins by looking for a previously computed intermediate result of the appropriate name in shared memory. If the result is found, and is timestamped within the requesting senselet's expressed tolerance for stale results (slack), it's returned from shared memory, and the computation is saved. Otherwise, the library performs the computation, and the result is stored under the appropriate name in shared memory for others to use.

Note that the senselet author need not explicitly refer to the shared store of intermediate results to reap the efficiency benefit of sharing. Rather, the sensor feed processing libraries used by senselets hide this complexity from the author, and do the work transparently.

Two factors most significantly affect the degree of increased computational efficiency offered by intermediate results sharing: the size of the shared memory in which results are stored, and the tolerance of senselets for stale results. We measure their performance implications experimentally in Section 7.1.

5.3 Privacy-Protecting Image Processing

Monitoring of human activities raises privacy concerns. Video streams from cameras in public places will often contain human faces, automobiles' license plates, and other data that can be used to identify a person. As an infrastructure to ease deployment of sensor services, IrisNet as described thus far makes raw sensed data equally available to all senselets. A senselet for a PSF service digests a video stream of a parking lot into an anonymous vector of empty and full parking spaces. But an ill-behaved senselet could extract the identities of people who appear in the same video stream, and thus violate their privacy. Ensuring, with full generality, that a video stream *cannot* be used to compromise the privacy of any individual is out-of-scope for our work on IrisNet. Nevertheless, we believe that IrisNet must provide a framework for helping to limit the ability of senselets to misuse video streams for unauthorized surveillance.

Toward that end, we have implemented *privacy-protecting image processing* for IrisNet, in which we use image processing techniques to anonymize a video stream. Our prototype uses a face detector algorithm to identify the boundaries of human faces in an image, and replaces these regions with black rectangles. Identifying people in the anonymized image is significantly more difficult.

Figure 6 shows the architecture of the IrisNet SA augmented to support privacy-protecting image processing. Here, IrisNet offers raw video data to *trusted senselets* and anonymized video data to *untrusted senselets.* The privacy filter anonymizes the raw video stream and makes the result available to untrusted senselets. Senselet authors cryptographically sign senselets, and SAs classify them into one of these two categories according to the (verified) identity of the author. Note that there are separate shared memories for the two senselet classes; intermediate result sharing is done as before, but only among senselets in the *same* class.

[4] IrisNet provides a parallel, non-shared API to the sensor feed processing libraries. If a senselet wishes not to share its results with others, it uses this API rather than the standard API, and allocates storage on the heap rather than in shared memory.

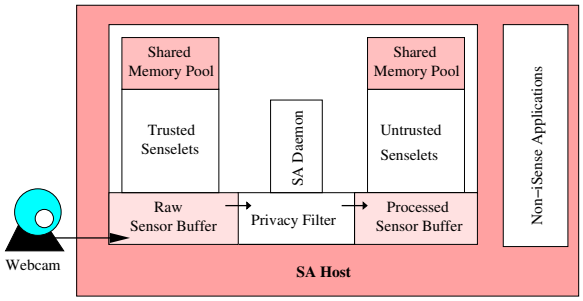


Figure 6: Privacy-Protecting SA

If the privacy filter and untrusted senselets are free-running, the resulting allocation of CPU among them under naive multiprogramming may be inefficient. For example, if the privacy filter produces 10 frames per second of video, but the untrusted senselets only can process 5 frames per second of video, the privacy filter wastes half the CPU it consumes—and these cycles might have been used by the untrusted senselets instead to increase their output frame rates. We observe that there are thus two goals for the scheduling of the privacy filter and untrusted senselets: maximizing the output frame rate of the untrusted senselets (the "useful work" done by the system), and eliminating wasted work by the privacy filter.

To realize these two goals, we implement a simple flow-control system between the privacy filter and untrusted senselets as follows. The privacy filter timestamps each video frame it produces, and marks the frame as *unused*. Any untrusted senselet that reads a frame marks that frame as *used*. An untrusted senselet requests a video frame by specifying the oldest timestamp value it can accept. It retrieves the newest used frame newer than that timestamp. If no newer frame has been used yet, the newest unused frame is retrieved. When no newer frame is available, an untrusted senselet sets the used bit on the newest frame, but cedes the CPU until a sufficiently new anonymized frame is produced by the privacy filter. This preference for retrieving previously used frames reduces the aggregate frame rate requested by the set of untrusted senselets by increasing sharing of frames, within their frame freshness constraints. The privacy filter monitors the number of unused frames in its output buffer. It only generates a new frame when there are *no* unused frames in the output buffer. In this way, we can ensure that the privacy filter produces frames at a rate no greater than the rate the fastest senselet consumes them.

5.4 Protected Execution

Running senselets at sensor hosts is fraught with safety concerns. Buggy or malicious senselets may consume excessive resources on a sensor host, or may even exploit security vulnerabilities in the sensor host OS, and thus compromise the sensor host. Such safety concerns are far from new. Active Network systems such as ANTS [43] download code to Internet routers to extend their functionality, and must contend with similar code safety risks.

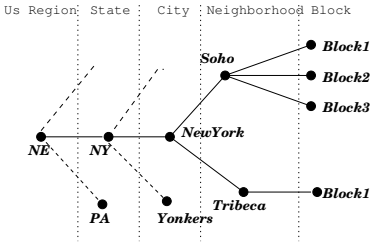


Figure 7: Hierarchy used in the PSF service

In the current IrisNet implementation, we assume trust between the administrator of a sensor host where senselets execute and the author(s) of those senselets. This model makes sense in cases where a single organizational entity administers the sensor hosts, and has a formal trust relationship with the senselet authors.

We believe that techniques applied in the Active Networks arena will be useful in evolving IrisNet to a model where senselets are less trusted by sensor host administrators. Senselets are signed by a trusted authority in IrisNet, as is the case for router extension code in ANTS. Executing senselets in a virtual machine (VM) would provide sandboxing, to limit the resources a senselet can consume and the operations it can execute. ANTS executes packet forwarding code in a Java VM for similar reasons.

Virtualization comes at the price of execution overhead, as was found in the evaluation of the ANTS system. However, senselets should pay minimal virtualization overhead, because of their compute-intensive (rather than system-call- or I/O-intensive) nature. A preliminary experiment verifies this intuition: when running the senselet for the PSF service previously mentioned under a User-Mode Linux (UML) VM [23], we observe a relatively modest 13% reduction in the senselet's video processing rate.

6 Prototype Services

We here present two services built on IrisNet, from two very different application domains.

6.1 Parking Space Finder (PSF)

The objective of the PSF service is to collect information from cameras installed in parking lots in a metropolitan area and to allow users to make queries about the availability of parking spaces. In the interest of having a controlled experimental environment, our PSF prototype uses simulated parking lots on a tabletop with toy cars.

Database schema. Figure 7 shows part of the hierarchy used in our current deployment of the PSF service. The hierarchy is defined by an XML schema which also includes dynamically changing parking space availability information from

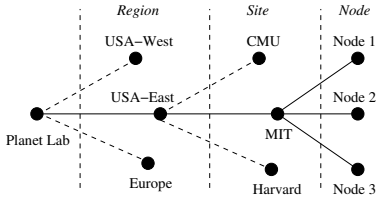


Figure 8: Hierarchy used in the NHM service.

the SAs as well as static data describing each of the parking spaces monitored (e.g., meter restrictions on the space).

Senselets. Senselets process the webcam video feeds to determine which parking spaces are empty and send the availability information to the appropriate OAs. After initial calibration of the reference background image, the senselet detects the presence of cars by differencing the current image of the spaces and corresponding background image [36]. All the image processing tasks are done using the Intel Open Source Computer Vision (OpenCV) [3] library.

Browser front end. The front end presents a web page on which the user can specify search criteria such as location, cost, etc. PSF converts this input into an XPATH query on the service database to discover matching spaces, and then uses the Yahoo Maps Service to find driving directions to a matching parking space from a specified location. The driving directions to the parking space are displayed and periodically updated with current parking space availability. We envision that a car navigation system could use this service, in combination with a GPS input, to provide up-to-date directions as the user nears the destination.

6.2 Network and Host Monitor (NHM)

The NHM service collects data from host and network monitoring tools (which act as sensing devices) running on a widely dispersed set of hosts, and allows users to efficiently query that data. Our current prototype of NHM runs on PlanetLab [34], an open, shared planetary-scale application testbed consisting of 102 nodes distributed across 42 sites in three continents: North America, Europe, and Australia. NHM currently supports a superset of the queries supported by the currently deployed Ganglia [30] PlanetLab monitoring service, but incurs far less network overhead.

The deployment of NHM demonstrates the ease of authoring a service on IrisNet. Although we did not have this service in mind while developing IrisNet, we created and deployed the NHM service on 102 PlanetLab nodes with only 1 person-day of effort.

Database schema. The schema for NHM describes the metrics to be monitored on PlanetLab nodes and organizes these metrics into a geographical hierarchy. This hierarchy, part of which is shown in Figure 8, allows efficient processing of geographically scoped queries (*e.g.,* find the least loaded MIT node).

To support historical queries on the monitored data, the schema uses multi-resolution vectors to store each monitored metric. These vectors provide higher resolution samples of recent data than older data.

Senselets. Each PlanetLab node runs an SA. The SA uses the output of the local Ganglia daemon to create a sensor feed describing 30 different performance metrics (*e.g.,* CPU and memory load, bandwidth usage, etc.) for the node. The NHM service takes these different metrics, translates them to match the NHM's XML schema, and periodically sends the XML to the local OA.

Browser front end. The front end for NHM presents a form on which the user can specify interest in: sets of PlanetLab nodes (*e.g.,* all the nodes at *MIT*), particular metrics (*e.g.,* CPU load), and time periods over which the data have been collected (*e.g.,* last one hour). The front end also allows the user to type arbitrary XPATH queries on the global XML database.

Note that it is impossible to make an apples-to-apples comparison between NHM and Ganglia. For example, Ganglia provides exact historical measurement data, while NHM provides a multi-resolution approximation of that data. Ganglia has a single point of failure but can provide historical data about a currently down node if the central server is up. NHM does not have any single point of failure, but can answer historical queries about a currently down node only if the data are cached in other OAs.

7 Experimental Results

In this section we present a performance evaluation of the IrisNet infrastructure using the two services described in Section 6. We seek to answer the following four questions:

(1) What are the performance gains in intelligently filtering at the SAs *vs.* doing that at the OAs (Section 7.1.1)?
(2) What is the gain of cross-senselet sharing (Section 7.1.2) and cost of privacy protection mechanisms (Section 7.1.3)?
(3) What is the query response time and bandwidth consumption of a service built on IrisNet (Section 7.2.1)?
(4) How much fault tolerance can be achieved by the IrisNet replication strategy (Section 7.2.2)?

In our current prototype of IrisNet, SAs are written in C and OAs are written in Java. OAs use the Xindice XML database [2]. Senselets for both services are written in C.

7.1 SA Experimental Evaluation

To evaluate SA performance, we use the PSF service, since it uses real webcam feeds and its senselets perform expensive image processing tasks. We run SAs on 1.2 GHz and OAs on 2.0 GHz Pentium IV PCs, all with 512 MB RAM. All the machines run Redhat 7.3 Linux with kernel 2.4.18. SAs sample the webcam feed 10 times per second, to support services that require up to that frame rate, and write frames into a shared buffer sized to hold 50 frames. Note, however, that senselets may choose to sample frames at a lower rate. For example, the PSF service reads one frame per second.

7.1.1 Processing Webcam Feeds

The PSF senselet processes video frames at the SA at one frame per second. It requires about 145 ms of processing time per frame and only a few bytes per second to transmit the availability information of 6 parking spots to the OA. An alternative design would be to compress the video data at the SA, send it to the OA, and process the video frame there. With this design, the SA would need around 18 KB/s to transmit the compressed video frame to the OA, which introduces more network load. Moreover, filtering at the SAs distributes the computational load over the SAs which are expected to outnumber the OAs, as multiple SAs may report to the same OA. These factors suggest that pushing sensor feed processing to the SAs is a more scalable design.

7.1.2 Effectiveness of Sharing among Services

We now report measurements of the amount of overhead we introduce by looking up and storing intermediate results in library functions, and the performance gains we achieve from sharing across senselets.

The Workload. For the experiments in this section, we use four different image processing senselets we have developed using the image processing library provided on SAs. These senselets perform image processing tasks (e.g., detecting an empty parking spot, detecting motion, etc.), and constitute a realistic synthetic workload for SAs. The four senselets all begin by performing the same sequence of image processing operations:

- Get current frame → Reduce noise → Convert to gray

They then continue their image processing as follows:

- Parking Space Finder 1 (PSF1): $\cdots \rightarrow$ Find contour → Compare contours $\rightarrow \cdots$
- Parking Space Finder 2 (PSF2): $\cdots \rightarrow$ Get image parts → Subtract background $\rightarrow \cdots$
- Motion Detector (MD): (read two frames) $\cdots \rightarrow$ Subtract images $\rightarrow \cdots$
- Person Tracker (PT): $\cdots$ Find Contour → Get image parts → Subtract background $\rightarrow \cdots$

The PSF service in Section 6.1 uses the senselet PSF2.

We report the results of four sets of experiments. The combinations of senselets in each set, and their deadline intervals in seconds are as follows:

[E1] 2 senselets: {PSF1, 1 sec} + {MD, 1 sec}
[E2] 4 senselets: E1 + {PSF2, 1 sec} + {PT, 1 sec}
[E3] 6 senselets: E2 + {PSF1, 2 secs} + {MD, 2 secs}
[E4] 8 senselets: E3 + {PSF2, 2 secs} + {PT, 2 secs}

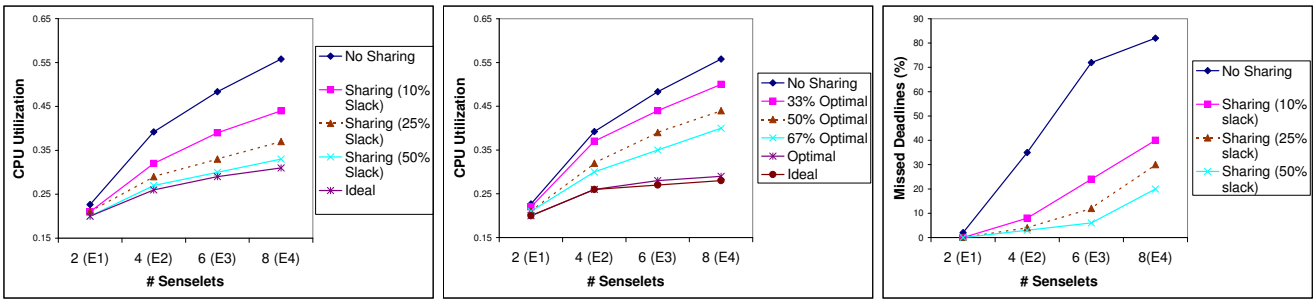
We average all measurements in this section over 20 30-minute executions. In all the results in this section, slack is given as a percentage of senselet's execution interval and the *optimal size* of the shared buffer is defined to be the size just big enough to hold all the intermediate results generated by the workload over the maximum deadline interval (2 secs for the above mentioned workload).

Overhead of sharing intermediate results. We measured the execution times for a few typical functions in the OpenCV API, and the cost of using memoization of intermediate results for these functions. We performed `cvCvtColor()`, `cvAbsDiff()`, and `cvFindContour()` operations on a lightly loaded SA on 20 different 640×480 24-bit images. These typical OpenCV library calls take between 1 and 5 ms to execute. The cost of looking up an intermediate result in shared memory, failing to find it there, and storing the freshly computed result in shared memory is 0.02 ms, less than 1% of the time taken by the original API in most cases.

The effect of sharing on CPU load. Figures 9(a) and 9(b) show that cross-senselet sharing significantly reduces the CPU load on SAs. As expected, the gain from sharing increases as the number of senselets increases, and more redundant computation is saved by result reuse. The graphs also show the *ideal* CPU load which is computed assuming that no two tuples with the same lineage and timestamp are ever generated. However, in IrisNet, a result computed by one senselet may be evicted from the fixed-size TS and shared memory before it is needed by another senselet, and thus must be computed again. Also, if a senselet working on the current frame misses its deadline and is scheduled later, it may not find a tuple fresh enough to use, even though it could have used the tuple if scheduled within the deadline. The likelihood of these occurrences increases with the number of concurrent senselets, as at higher CPU loads, senselets requiring the same tuple may be scheduled to execute far apart in time from each other. This explains why the load with sharing in IrisNet is higher than the ideal load, and why the gap between the two curves grows with the number of concurrent senselets.

We note that the performance gap between sharing and the ideal case can be reduced by using greater slack values on senselet deadlines or larger shared memory buffers. Figure 9(a) shows that the CPU utilization under result sharing approaches the ideal CPU utilization as the slack value increases. Greater tolerance of older results increases the likelihood of finding an intermediate result with a timestamp falling in the desired window. Figure 9(b) reveals that as the shared memory size increases, the performance of sharing again approaches the ideal case, as shared memory holds progressively more results for later re-use.

The effect of sharing on missed deadlines. As described in Section 5.1, senselets exhibit soft real time behavior by dynamically adjusting the length of the period they sleep between two successive rounds of processing. However, because the SAs do not run under a real-time OS, scheduling of SAs may become unpredictable at high CPU loads, such that senselets miss more deadlines. Figure 9(c) show how the number of missed deadlines increases with the number of



(a) Slack and CPU time (b) Memory size and CPU time (c) Slack and missed deadlines

Figure 9: Plots showing the effect of sharing on CPU time (a and b) and missed deadlines (c). The default slack is 10% and the default shared memory size is 50% of the optimal size.

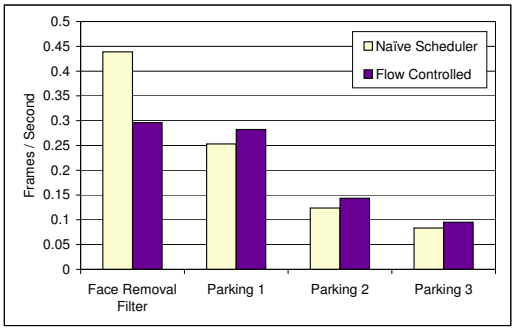


Figure 10: Effect of flow control on senselet processing rates.

concurrent senselets. Without sharing, the SA host becomes overloaded quickly and senselets miss more and more deadlines. Cross-senselet sharing significantly reduces missed deadlines by shedding redundant CPU load and re-using tuples computed previously to meet deadlines.

7.1.3 Privacy Protection Overhead

To evaluate the potential overhead of the privacy-protecting filter, we constructed a filter using the OpenCV face detector. The filter detects all human faces in a video frame and replaces them with a gray rectangle. We measured the effects on three different untrusted senselets, each requiring different amounts of processing time per frame. We modified the PSF senselet into Parking 1, 2, and 3 senselets, which include multiple calls to a camera calibration function for each frame. We deliberately chose this compute-intensive function and disabled sharing to illustrate the effects of flow control.

The first bar of each group in Figure 10 shows the frame rate of each component when they run concurrently and without any flow control. They are scheduled using the default Linux process scheduler. Because there are four concurrent processes running, the frame rate of each component running individually is four times of what is shown in the graph. With no flow control, the face removal filter runs at 0.44 fps while Parking 1 runs at 0.25 fps. The filter is wasting 43% of its work. After adding flow control between the face filter and the senselets, the face filter's frame rate drops to 0.30 fps, while Parking 1's frame rate increases to 0.28 fps. We see a 12, 16, and 14 percent increase in frame rates for parking apps 1, 2 and 3, respectively. As expected, the CPU time given up by the filter is evenly distributed among the senselets.

7.2 OA Experimental Evaluation

To understand how IrisNet's query execution performs with a wide-area service, we use the NHM service since it has been deployed on a large set of nodes in the Internet. The NHM service we use in this section uses the hierarchy shown in Figure 8. The nodes in the hierarchy are distributed among OAs running on 102 PlanetLab nodes across three continents. Every OA owns the leaf node corresponding to the host PlanetLab node and possibly a few non-leaf nodes in the hierarchy. For brevity, we here omit the complete mapping of non-leaf nodes in the hierarchy to the OAs. To achieve robustness against PlanetLab node failure, each site-level node in the hierarchy has one primary and one secondary replica. The primary replica is placed in a geographically optimal OA. For example, the primary replica of the site-level node `MIT` is placed in one of the PlanetLab nodes at MIT, whereas the secondary replica is placed in some random node outside MIT. Similarly, each region-level node in the hierarchy has two primary and two secondary replicas. Finally, the root node in the hierarchy is replicated in five geographically dispersed PlanetLab nodes.

We consider four different classes of queries: *global* (Q_g), *region*-(Q_r), *site*-(Q_s) and *node-level* (Q_n) queries, involving all the nodes, all nodes in a region, all nodes in a site and a specific node respectively. We use the following two different workloads to evaluate the query response time and communication overhead.

- Synthetic: The workload consists of 1000 instances of all possible queries of the class. For this workload, we consider two types of queries: *historical queries* ask for data monitored over the last year[5], and *latest-value queries* ask for the most recent monitored data.

[5]NHM supports historical queries over smaller periods of time, *e.g.,* last one hour.

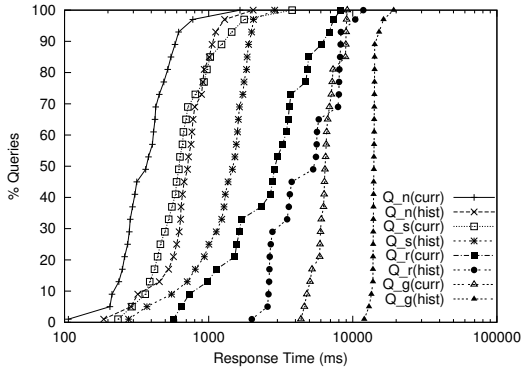


Figure 11: Response times for different classes of queries (using synthetic workload); hist denotes historical queries; curr denotes latest-value queries

- Ganglia: We derive a workload from the Ganglia web server log, which lists all user-requests from 8-7-2002 to 2-26-2003. The log consists of 90014 queries of which 4015 are of class Q_g, and the rest are of class Q_s. The queries in this log are all historical queries with different time periods. Since this workload is very sparse (0.31 queries per minute), and we want to see how NHM performs under a higher query rate, but a similar query distribution, we *compress* the overall timescale of the Ganglia log. For example, to simulate 10 times the query rate, we assume that two queries 10 seconds apart in the original Ganglia log appear 1 second apart in the *compressed* log.

7.2.1 Query Execution

Response Times. Figure 11 shows the distribution of response times for different classes of queries. The average response time for historical queries is more than that for latest-value queries. This is due to the extra processing and communication overhead for retrieving and post-processing historical data. The average response times for Q_n queries are on the order of milliseconds and those for Q_s and Q_r queries are on the order of seconds. Only the historical Q_g queries have response time longer than 10 seconds, which is still reasonable considering the amount of information *pulled* (around 3MB total) from a large number of nodes.

Figure 12(a) demonstrates the effectiveness of caching in reducing response times. The average response time to queries from the Ganglia log is slightly less than 2 seconds. However, with caching, response time decreases significantly and goes below a second at a query rate of more than 12 queries per minute. This is due to the fact that under the increased query rate, more and more queries are served from the starting point OAs in the hierarchy without requiring the data to be pulled from the other OAs containing the descendant nodes.

Bandwidth Consumption. In Figure 12(b), the three curves `Ganglia-5`, `Ganglia-15` and `Ganglia-60` show the network bandwidth consumed by Ganglia on PlanetLab, with a 5-, 15- (default value used by Ganglia) and 60-second polling interval. `Ganglia-15` data has been taken from [30], while the data for `Ganglia-5` and `Ganglia-60` have been extrapolated from `Ganglia-15` data. As the figure shows, Ganglia on Planetlab consumes more than 6 Kbps per node. Under the same query rate and distribution, NHM incurs two orders of magnitude less network overhead[6].

Figure 12(b) also shows the result of replaying the Ganglia log in NHM and summarizes how the network bandwidth consumption increases with query rate. Our single-threaded front end can answer 30 queries per minute on average[7] (which is 100 times more than the query rate in the Ganglia log), and even at that rate NHM consumes less bandwidth than Ganglia. Extrapolating the curves in Figure 12 shows that NHM can support as many as 120 queries per minute while consuming less network bandwidth than Ganglia on PlanetLab. This shows the potential advantage of a *pull*-based system like NHM.

The effectiveness of caching is evidenced by the the reduction of slope of the bandwidth curve in Figure 12(b). The benefit of caching increases with load for the same reasons that response time improves.

7.2.2 Fault Tolerance

To evaluate the effectiveness of the replication strategy discussed in section 4.4, we use the metric *queryability*. Let L_q be the total number of non-faulty leaf nodes specified by a query workload. Since queries are propagated from the starting point OAs down to the leaf OAs (OAs containing the leaf nodes), in the presence of faulty OAs, all the L_q nodes may not be included in the responses. Let L_a denote the total number of leaf nodes actually present in the responses ($L_a \leq L_q$). Then the queryability of an OA hierarchy for the given query workload is given by the ratio L_a/L_q. Intuitively, the greater the ratio, the more fault-tolerant the OA hierarchy.

Figure 12(c) shows how the queryability of NHM's OA hierarchy decreases as the fraction of faulty OAs increases. The figure assumes uncorrelated OA failures, but since one OA may contain multiple nodes in the hierarchy, failure of an OA renders all the nodes owned by that OA unreachable by a query. Each point in the graph denotes the average queryability of 50 different random combinations of the same number of faulty OAs. The graph shows that even the simplest static replication strategy used by NHM provides very high queryability (> 0.99) in the presence of a reasonable number of faulty OAs ($\leq 10\%$).

The presence of faulty OAs may degrade response time because a faulty OA may be tried first before a message is sent to a non-faulty OA, and the chosen non-faulty OA may be in a geographically non-optimal location. However, our

[6] Although NHM collects measurement data from the Ganglia daemon, each SA uses only the part of the Ganglia output related to its host. NHM could have collected the data locally. Therefore we do not charge NHM with the network traffic that Ganglia introduces for monitoring the whole cluster.

[7] IrisNet can support, *e.g.,* through multiple front ends, a higher query rate, since many simultaneous queries are disjoint to each other.

experiment shows that this degradation of response time is minimal. For example, each time a query is routed through a secondary replica, it incurs a penalty of about 500 ms.

8 Related Work

In this section, we explore related efforts in the following areas of work: Internet services, Active Networks, sensor networks, distributed databases, and video surveillance. Note that while each of these related efforts addresses a subset of the issues in creating sensor services, only IrisNet provides a complete solution for enabling such applications.

Internet Services. A number of different efforts, *e.g.*, [42, 40, 4, 16] have developed frameworks for simplifying the development of scalable, robust Internet services. In general, these projects target a lower level of the architecture than IrisNet. They concentrate on issues that are generic to all Internet services, such as load balancing, resource allocation and network placement. In contrast, IrisNet addresses issues that are unique to services that need to collect vast amounts of data and process queries on the data. In this way, IrisNet is largely complementary to these previous efforts and could, in fact, be implemented using these frameworks.

Active Networks. The Active Network architecture [43] shares much in common with our SA design. In both systems, a shared infrastructure component (routers and SA hosts) is programmed by end-users and developers to create a new service. However, the differences between the target applications of packet forwarding and sensor data retrieval result in significant differences in the requirements for Active Networks' code capsules and IrisNet's senselets. In order to protect the resources of the router, capsules need to complete execution quickly, typically before the arrival of the next capsule. Capsule code is also limited to using soft-state at the router across invocations. In contrast, the very purpose of senselets forces them to be long-running and store hard state. Another important difference is that capsule code is fetched on demand (and cached) upon arrival of a packet. This fact and resource constraints force capsule code to be relatively small. The loading and execution of a senselet is performed once—upon the initialization of the sensor service. In general, the programming environment of SAs is far less constrained than that of capsules.

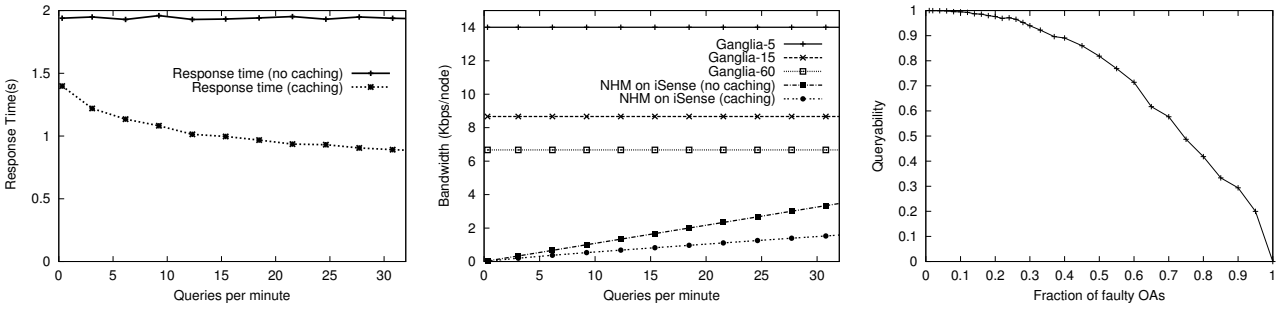
Sensor Networks. Sensor networks and IrisNet share the goal of making real world measurements accessible by applications. The work on sensor networks has largely concentrated on the use of "motes," small nodes containing a simple processor, a little memory, a wireless network connection and a sensing device. Because of the emphasis on resource-constrained motes, earlier key contributions have been in the areas of tiny operating systems [22] and low-power network protocols [26]. Mote-based systems have relied on techniques such as directed diffusion [21] to direct sensor readings to interested parties or long-running queries [9] to retrieve the needed sensor data to a front-end database. Other groups have explored using query techniques for streaming data and using sensor proxies to coordinate queries [27, 28, 29], to address the limitations of sensor motes. None of this work considers sensor networks with intelligent sensor nodes, high-bit-rate sensor feeds, and global scale.

Distributed Databases. The distributed database infrastructure in IrisNet shares much in common with a variety of large-scale distributed databases. For example, DNS [31] relies on a distributed database that uses a hierarchy based on the structure of host names, in order to support name-to-address mapping. LDAP [41] addresses some of DNS's limitations by enabling richer standardized naming using hierarchically organized values and attributes. However, it still supports only a relatively restrictive querying model. Harren *et al.* [20] have investigated peer-to-peer databases that provide a richer querying model than the exact-match queries supported by existing DHT systems, to limited success (significant hotspots in storage, processing, and routing were reported). DHT-based databases are more robust than the replicated hierarchically-based approach we propose, but are less well suited to leveraging XML and the hierarchically-scoped queries typical in sensor services. Distributed databases supporting a full query processing language such as SQL are a well-studied topic [38], with the focus on supporting distributed transactions or other consistency guarantees (*c.f.* [37, 18, 35, 7, 25, 8, 10]). None of the previous work addresses the difficulties in distributed query processing over an XML document. There is also considerable work on query processing over a federation of heterogeneous databases [38], dealing with issues such as incompatible schemas. These issues do not arise in our current architecture, because the service author defines the schema for an entire service.

Video Surveillance. The use of video sensors has been explored by efforts such as the Video Surveillance and Monitoring (VSAM) [11] project. Efforts in this area have concentrated on image processing challenges such as identifying and tracking moving objects within a camera's field of vision. These efforts are complementary to our focus on wide-area scaling and service authorship tools.

9 Conclusions

We have described the design, architecture, and implementation of IrisNet, the first general-purpose software infrastructure for enabling sensor-enriched Internet services. With IrisNet, a strikingly simple service specification defines all IrisNet needs to create an efficient, highly distributed Internet service. We presented novel techniques for data partitioning, distributed query processing, fault tolerance, service-specific sensor feed processing with sharing, and privacy-protecting image processing—all tailored for sensor-enriched services. Finally, we reported on a performance evaluation of IrisNet running two very different services, demonstrating the effectiveness of IrisNet's features in



(a) Response time (b) Bandwidth consumption (c) Queryability

Figure 12: Plots showing the results of replaying the Ganglia log.

reducing network bandwidth, increasing sensor processing throughput, tolerating faults, and reducing query response times. For example, using an IrisNet-based tool to monitor PlanetLab instead of Ganglia reduced the network bandwidth of a real-world workload by two orders of magnitude, while providing 1-second response times.

Our ongoing research includes improved resource discovery mechanisms, additional real-world services (*e.g.,* we are currently working with a team of oceanographers to deploy IrisNet along the Oregon coast, to detect and analyze sandbar formation, riptides, etc.), additional privacy mechanisms, and integration with smart dust sensors.

Acknowledgements. Thank Amol and Rahul. We thank M. Satyanarayanan for many valuable suggestions and C. Helfrich for helping with the experimental setup.

References

[1] Accepted for publication. Reference omitted for anonymity. For an anonymized copy of this paper, please contact the program chair.

[2] Apache xindice. http://xml.apache.org/.

[3] Intel Open Source Computer Vision Library. http://www.intel.com/research/mrl/research/opencv/.

[4] Libra: Scalable advanced network services based on coordinated active components. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/cmcl/www/team/.

[5] Webdust: Automated construction and maintenance of spatially constrained information in pervasive microsensor networks. http://athos.rutgers.edu/dataman/webdust.

[6] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *SOSP*, pages 186–201, 1999.

[7] D. Agrawal and S. Sengupta. Modular synchronization in distributed, multi-version databases: Version control and concurrency control. *IEEE TKDE*, 1993.

[8] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM TODS*, 1990.

[9] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards sensor database systems. In *MDM*, 2001.

[10] S. W. Chen and C. Pu. A structural classification of integrated replica control mechanisms. *ACM TODS*, 1992.

[11] R. Collins, A. Lipton, H. Fujiyoshi, and T. Kanade. Algorithms for cooperative multisensor surveillance. *Proceedings of the IEEE*, 89(10):1456–1477, Oct. 2001.

[12] D. Culler, E. Brewer, and D. Wagner. Berkeley Wireless Embedded Systems (WEBS). http://webs.cs.berkeley.edu/.

[13] A. Elgammal, R. Duraiswami, D. Harwood, and L. S. Davis. Background and foreground modeling using nonparametric kernel density estimation for visual surveillance. In *Proceedings of the IEEE*, volume 90 (7), pages 1151–1163, July 2002.

[14] D. Estrin, R. Govindan, and J. Heidemann. SCADDS: Scalable Coordination Architectures for Deeply Distributed Systems. http://www.isi.edu/scadds.

[15] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *MOBICOM*, 1999.

[16] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[17] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP*, 1997.

[18] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[19] Guttman, P. E., and C. Service location protocol. Technical report, IETF, 1999. RFC 2608.

[20] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *IPTPS*, 2001.

[21] J. Heidemann et al. Building efficient wireless sensor networks with low-level naming. In *SOSP*, 2001.

[22] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *ASPLOS*, 2000.

[23] H. J. Hoxer, K. Buchacker, and V. Sieh. Umlinux - a tool for testing a linux system's fault tolerance. In *LinuxTag 2002*, Karlsruhe, Germany, June 2002.

[24] J. Kahn, R. H. Katz, and K. Pister. Next century challenges: Mobile networking for 'smart dust'. In *MOBICOM*, 1999.

[25] N. Krishnakumar and A. Bernstein. Bounded ignorance in replicated systems. In *PODS*, 1991.

[26] J. Kulik, W. Rabiner, and H. Balakrishnan. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks . In *MOBICOM*, 1999.

[27] S. Madden and M. J. Franklin. Fjording the stream:an architecture for queries over streaming sensor data. In *ICDE*, 2002.

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

[29] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

[30] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. Submitted for publication, February 2003, http://berkeley.intel-research.net/bnc/.

[31] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *SIGCOMM*, 1988.

[32] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD*, 2002.

[33] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *ASPLOS*, 1998.

[34] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Hotnets-I*, 2002.

[35] C. Pu and A. Leff. Replica control in distributed system: An asynchronous approach. In *SIGMOD*, 1991.

[36] Shapiro and Woods. *Computer Vision*. Prentice-Hall, 2001.

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

[38] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw Hill, 2002.

[39] Sun Microsystems. Jini technology architectural overview. Technical report, 1999. http://www.sun.com/jini/whitepapers/architecture.html.

[40] J. R. von Behren, E. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A framework for network services. In *Proceedings of 2002 USENIX Annual Technical Conference*, 2002.

[41] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). Technical report, IETF, 1997. RFC 2251.

[42] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP*, pages 230–243, 2001.

[43] D. Wetherall. Active network vision and reality: lessions form a capsule-based system. In *SOSP*, pages 64–79, 1999.