IrisNet: An Architecture for Compute-Intensive Wide-Area Sensor Network Services

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

IRP-TR-02-10
December 2002

Intel Research Pittsburgh

IrisNet: An Architecture for Compute-Intensive Wide-Area Sensor Network Services

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

[*]*Intel Research Pittsburgh* [†]*Carnegie Mellon University* [‡]*U.C. Berkeley*

Abstract

Previous work on sensor networks has targeted ad hoc wireless networks of closely-colocated, resource-constrained scalar sensor motes. Such work has overlooked richer sensor types such as webcams and microphones, which are typically attached to Internet-connected machines with significant computing power and storage. In this paper, we describe *IrisNet* (Internet-scale Resource-Intensive Sensor Network services), an architecture for wide-area sensor networks based on these more capable sensing nodes. IrisNet provides a common, scalable software infrastructure that enables the flexible creation of sensor-based Internet services. It dramatically reduces network bandwidth utilization through the use of *senselets*, binary code fragments that perform intensive data filtering at the sensing nodes, leveraging the available processing power and memory. IrisNet employs a two-tier hierarchy of sensing nodes and query processing nodes. Key features of IrisNet include flexible data partitioning, efficient and protected sharing of sensor nodes, low-latency queries, partial match caching, query-specified freshness tolerances, and monitoring and logging support. This paper reports on experiments with a working IrisNet prototype running a parking space finder service, demonstrating the effectiveness of IrisNet's features in achieving scalability and reducing query response times.

1 Introduction

The proliferation and affordability of webcams and other smart sensors has created opportunities for new sensor-based services. Consider for example a Parking Space Finder service for locating available parking spaces near a user's destination. The user specifies criteria for desirable parking spaces (e.g., within two blocks of her destination, at least a four-hour meter), and receives directions to an available parking space satisfying her criteria. If the space is taken before she arrives, the directions are automatically updated to head to a new parking space. Cameras overlooking parking spots may also be used by additional concurrently running services, such as a Meter Enforcement Tracker service for determining when a meter reader last passed by.

While several research projects [14, 11, 7] have begun to explore the use of networked collections of sensors, these systems target the use of densely deployed, energy- and resource-constrained sensor motes (*smart dust*) [15, 21]. Severely limited energy, computational, storage, and network capacities are the key drivers of design choices in these projects. These systems are aimed at ad hoc wireless networks of sensors deployed in a *single contiguous communication domain*, e.g., a battlefield being monitored for tanks, or an island whose habitat is being monitored. Specialized hardware, operating systems, programming languages and database systems have been developed to accommodate this severely constrained environment [18, 22, 25].

In this paper, we discuss a complementary agenda based on more capable sensing devices (which might be called *brilliant rocks*). We describe *IrisNet* (Internet-scale Resource-Intensive Sensor Network services), a *wide-area* sensor network architecture for much more intelligent and capable devices that are widely deployed at the edges of the global Internet. Many of today's Internet nodes already have interfaces that support the attachment of sensors such as webcam video cameras. IrisNet leverages the available processing power and memory at these nodes to perform intensive processing of the sensor data at the nodes. Commodity off-the-shelf (COTS) hardware, operating systems, programming languages and databases are exploited to provide a powerful shared sensor network infrastructure. IrisNet is designed to be a common platform for service developers to create and deploy sensor-based web ser-

smart dust	**brilliant rocks**
mote hardware	PCs/PDAs
TinyOS, TinyDB, etc.	Linux, Java, XML
campus-scale	Internet-scale
minimal sensor processing	intensive sensor processing
energy is a key concern	powered nodes
scalar sensors	multimedia sensors
narrowly focused services	wide variety of services
ad hoc wireless connectivity	direct Internet connectivity

Figure 1: Key differences

vices. The same infrastructure can be shared by a wide variety of services such as Parking Space Finder and Meter Enforcement Tracker. Figure 1 summarizes the key differences between IrisNet and the previous work on smart dust.

There are two fundamental aspects to designing a system for wide-area sensor services, both of which are relevant to IrisNet. First, on the data-producer side of the sensing system, a service designer needs mechanisms to deploy sensor service code to the appropriate sensor nodes, run the code on them, and collect the output. Second, on the data-consumer side, users of a sensing service need to be able to address queries to the sensor network as a single unit, and should receive prompt and accurate responses.

These two broad aspects of wide-area sensing in turn give rise to many criteria for the design of a successful system. For each design criterion, we highlight the key features of IrisNet that satisfy the criterion.

- **Network bandwidth efficiency:** The system will make use of potentially vast numbers of sensor data feeds, distributed across the Internet. These feeds may include rich, high bit-rate sensed datatypes, such as video and audio. To scale, the system must be efficient in its use of network bandwidth as the number of sensing nodes and bit-rate of sensor feeds increase.

  IrisNet approach – Distributed filtering: IrisNet processes high bit-rate sensor feeds on the CPU of the sensor node where the data are gathered. This dramatically reduces the bandwidth consumed: instead of transferring the raw data across the network, IrisNet sends only a potentially small amount of post-processed data. As an added benefit, having each sensor node perform its own processing (in parallel) avoids concentrating the processing of many sensor feeds at a single CPU.

- **Efficient and protected sharing of sensor nodes:** The data feed produced by a particular sensor, by virtue of the sensor's position and sensing modality, may be of interest to multiple distinct sensor services. A successful system must support sharing of sensor devices by multiple mutually distrusting services. Moreover, these services may overlap in the processing they perform on the same data source. Eliminating such redundant computations increases the scalability of the system. This creates a tension between the desire for isolation and for sharing at the sensor nodes.

  IrisNet approach – Sensing code protection: Each sensing service sharing the same sensor CPU is encapsulated in a separate process, to prevent their interfering with one another's address spaces. The entire set of sensing codes may be encapsulated in a virtual machine, to enforce limits on the total resources they consume.

  IrisNet approach – Re-use of intermediate results: IrisNet includes a mechanism for sharing of results *between* sensing services running on the same node. Distinct sensing tasks name the results of their computations, and they may look up, by name, and re-use results computed by other sensing tasks.

- **Low-latency queries:** As with any interactive system, users desire prompt responses to their queries of the wide-area sensing system.

  IrisNet approach – Hierarchical caching and query routing: IrisNet organizes sensor nodes into a hierarchy, and uses the hierarchy to cache previous responses, to reduce response latency for future queries. Queries are routed directly to the nodes of interest to the query. The system incorporates sophisticated caching techniques, such as partial match caching, whereby previously cached data that only satisfy portions of a newly arrived query may be synthesized to build a response.

- **Flexible consistency constraints:** Sensed data can be highly dynamic. Depending on the context, a user may demand up-to-date data, or may prefer a faster response using cached, slightly stale data.

  IrisNet approach – Freshness specifications: A query in IrisNet may explicitly specify the age of the data it desires, trading off response time for data freshness.

- **Efficient monitoring and logging support:** Because of the aggressively distributed nature of a wide-area sensing system, users will need a way to record and collect measurements of the system's behavior, to support testing, debugging, and performance analysis of their sensing services. The system should provide these measurements without reliance on heavyweight coordination, such as globally synchronized clocks.

  IrisNet approach – Logging with logical clocks: IrisNet uses low-overhead logical clocks to record causality between distributed events in the system, and logs these events for subsequent on-demand collection.

IrisNet is the first system to address these challenges in wide-area sensing. We have built a working prototype of the IrisNet system, running on COTS PC desktops, laptops, and webcams. As a proof of concept, our current prototype supports the above-mentioned Parking Space Finder (PSF) service in a controlled environment consisting of toy cars on tabletop parking lots. We present measurements of CPU load, network bandwidth consumption, response time, and sensor code protection overhead gathered on this testbed that demonstrate the utility of the IrisNet design features enumerated above in achieving our stated design criteria.

In the following sections, we give a more detailed overview of the IrisNet architecture (Section 2), describe the distributed database IrisNet uses to serve user queries (Section 3), and give details of IrisNet's execution environment for sensing code (Section 4). After briefly presenting the logging and monitoring support offered by IrisNet (Section 5), we provide a detailed description of the PSF service (Section 6), as an example of an application of the architecture and features previously set forth, and detailed measurements of this service's behavior and performance (Section 7). Finally, we place IrisNet in the context of related work (Section 8) and present conclusions (Section 9).

2 The IrisNet Architecture

In this section, we describe the basic two-tier architecture of IrisNet (Figure 2), its benefits, and some of the challenges it creates. We also examine how a service developer can build services using this infrastructure. The two tiers of the IrisNet system are the Sensing Agents (SAs), which collect and filter sensor readings, and the Organizing Agents (OAs), which perform query processing tasks on the sensor readings. Service developers deploy sensor-based services by orchestrating a group of OAs dedicated to the service. As a result, each OA participates in only one sensor service (a single physical machine may run multiple OAs), while an SA may provide its sensor feeds and processing capabilities to a large number of such services.

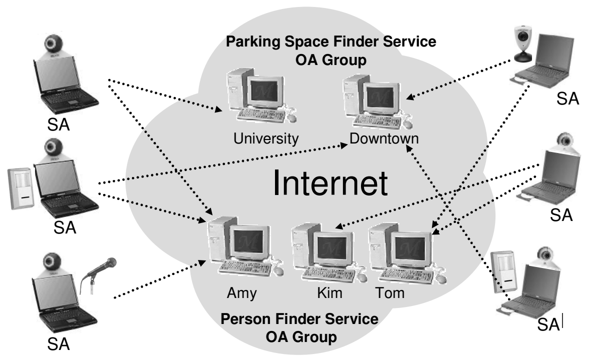


Figure 2: IrisNet Architecture

2.1 OA Architecture

The group of OAs for a single service is responsible for collecting and organizing sensor data in order to answer the particular class of queries relevant to the service (*e.g.,* queries about parking spots for a PSF service). Each OA has a local database for storing sensor-derived data; these local databases combine to constitute an overall sensor database for the service. One of the key challenges is to divide the responsibility for maintaining this Internet-scale sensor database among the participating OAs. IrisNet relies on a hierarchically organized database schema (using XML) and on corresponding hierarchical partitions of the overall database, in order to define the responsibility of any particular OA. Each service can tailor its database schema and indexing to the particular service's needs, because separate OA groups are used for distinct services. The details of the operation of OAs are described in Section 3.

2.2 SA Architecture

SAs collect raw sensor data from a number of (possibly different types of) sensors. The types of sensors can range from webcams and microphones to temperature and pressure gauges. The focus of our design is on sensors such as webcams that produce large volumes of data, and can be used by a variety of services.

One key challenge is that transferring large volumes of data to the OAs can easily exhaust the resources of the network. IrisNet relies on sophisticated processing and filtering of the sensor feed at the SA to reduce the bandwidth requirements. To greatly enhance the opportunities for bandwidth reduction, this processing is done in a *service-specific* fashion. IrisNet enables OAs to upload programs, called *senselets*, to perform this processing to any SA collecting sensor data of interest to the service. These senselets instruct the SA to take the raw sensor feed, perform a specified set of processing steps, and send the distilled information to the OA. Senselets can reduce the needed bandwidth by orders of magnitude, *e.g.,* PSF senselets reduce the high volume video feed to a few bytes of available parking space data per time period. On the other hand, the use of senselets creates three new challenges: (1) what programming language does IrisNet provide for the senselets, (2) how does IrisNet ensure that the senselets do not compromise the security of the SAs, and (3) how does IrisNet enable scaling to a large number of senselets running on the same SA. The details of the SA programming and execution environment are provided in Section 4.

2.3 Developing a Service

We here describe the steps a service developer performs to create a new service using IrisNet to illustrate how our two-tired architecture is used.

1. The developer must first create the sensor database schema. The schema defines the attributes and tags used to describe the sensor readings and the hierarchies used to index the data. IrisNet uses the schema to create a single OA that hosts the overall database for the service. IrisNet provides a mechanism to add new OAs to a service and re-partition the database across the nodes as needed.
2. To populate the database, the developer must write senselet code for the SAs that have sensor coverage relevant to the desired sensor service. The senselets take the sensor feeds, extract just the data relevant to the service, and send the data to the appropriate OA. The developer also initializes any static data in the database.
3. Finally, the developer must provide a user interface for end users to access the service. This user interface takes some simplified user input and generates the appropriate set of database queries to the OAs for the service. IrisNet provides functionality to efficiently answer these queries over the distributed sensor database.

To assist developers in testing, debugging, and evaluating their designs, IrisNet logs events throughout the system. Developers can collect these logs when needed and playback the events in a visualization system that IrisNet provides. The details of this monitoring system are described in Section 5.

The above description highlights how IrisNet achieves its core objective of making it easier to create and deploy new services. IrisNet seamlessly handles many of the common tasks within sensor-based services, such as the query processing, indexing, networking, caching, load balancing, and resource sharing. In the following sections, we describe the details of how IrisNet performs these tasks.

3 OAs as a Distributed Database

One of the core components of almost any sensor service is the ability to make queries about relevant sensor readings. The OAs within IrisNet provide a simple way for a service to incorporate support for rich queries. In this section, we give an overview of IrisNet's support for the database processing of typical sensor services.

3.1 Query Processing

Two important issues for query processing are: (1) the type of data descriptions and queries to support, and (2) the mechanisms for scaling the system to large amounts of data and high update frequencies. Here, we describe how IrisNet handles these two issues.

We envision a rich and evolving set of data types, aggregate fields, etc., within a service and across services, best captured by self-describing tags. In addition, each sensor takes readings from a geographic location, so it is natural to organize sensor data into a geographic/political-boundary hierarchy. Hence, we choose to represent sensor-derived data in XML since it is well-suited to representing such data. We use the XPATH query language, because it is the most-widely used query language for XML, with good query processing support. Figure 3 gives a simple example of an XPATH query

Query: **/usRegion[@id='NE']/state[@id='PA']**
/city[@id='Pittsburgh']/neighborhood[@id='Oakland'
OR @id='Shadyside']/block/parkingSpace[available='yes']

DNS-style name of the LCA:
city-pittsburgh.state-pa.usregion-ne.parking.intel-iris.net

Figure 3: A typical XPATH query and the DNS-style name found from its hierarchical prefix

on a hierachical 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 Pittsburgh, PA. XPATH supports a rich set of query predicates.

3.1.1 Partitioning and naming

In order to support a large number of sensor readings coming from geographically diverse sources, it is critical to enable the sensor database to be widely distributed. IrisNet permits a very flexible partitioning scheme for distributing a sensor database, based on a set of partitioning rules, special tag attributes, and invariants it maintains. In a nutshell, the sensor database can be partitioned at any *IDable* node in the hierarchy. An IDable node has a special `id` attribute. The value of an `id` attribute is a short name that makes sense to the user query (e.g., `Pittsburgh`). The value is unique among its siblings, e.g., there can be only one city whose `id` is `Pittsburgh` among the children of the `PA` state node. Moreover, an IDable node has an IDable parent node (or it is the root node of the hierarchy); thus an IDable node is uniquely identified by the sequence of node names and `id`s on the path from the root to the node.

Each OA registers with DNS the unique name for each IDable node in its partition, appending the service name (e.g., `parking`) and the domain name `intel-iris.net`. Figure 3 depicts the DNS name for the `Pittsburgh` IDable node. The DNS name server hierarchy can be maintained on the same set of OA nodes if necessary. DNS provides a simple way for any node to contact the owner of a particular IDable node of the database. It is the only mapping from the logical database to physical IP addresses in the system, enabling considerable flexibility in mapping databases 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.

3.1.2 Routing a query and gathering the answer

Due to our flexible 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. Due to our naming scheme, 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 information is needed to produce the DNS name of the LCA! To see this, we observe that each XPATH query contains a (maximal) hierarchical prefix, which specifies a single path from the root of the hierarchy to the LCA. From the node names and `id` fields in the query, IrisNet constructs the DNS name for the LCA. For example, for the query in Figure 3, the hierarchical prefix is shown in bold font, the LCA node is Pittsburgh, and the DNS name is as shown. IrisNet uses a simple parser to scan the query for a sequence of `id` predicates, and create the DNS name. After looking up the IP address, IrisNet transmits the query to that OA, which we call the *starting point* OA for the query.

Upon receiving a query, the starting point OA queries its portion of the overall database and evaluates the result. However, for many queries, a single OA may not have enough of the database to respond to the query. Based on the invariants and tags it maintains for its local database, the OA determines which part of a user's query can be answered by the local database and where to gather the missing parts. 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 different responses are collected and combined by the starting point OA and the result is sent back to the user. For the example in Figure 3, if the `Oakland` and `Shadyside` nodes reside on different machines than the `Pittsburgh` node, then the Pittsburgh OA sends subqueries to the Oakland OA and the Shadyside OA, who each return a list of available parking spaces, to be combined at the Pittsburgh OA and returned to the user.

3.2 Caching and Data Consistency

Like many other network services, it is obvious that there will be a great deal of locality in the user requests to a sensor service. For example, in a PSF service, there are likely to be many more queries about downtown parking than rural/suburban parking. 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. Each OA maintains partitioning and cache invariants that ensures that even the partial matches on cached data can be exploited.

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, the IrisNet service developer for PSF might specify a large tolerance when the user is far from her destination or there are a large number of available spots, because the user probably would not mind an old cached response. However, as the user approaches her destination or when available spots are scarce, a low tolerance is specified. The ability to specify a tolerance for data timeliness is a natural and useful feature for sensor-based services. For example, it enables users to ask queries such as "Have you seen Joe?" and specify whether the desired timeframe is this morning, today, this week, etc. We store timestamps along with the cached data, so that an XPATH query specifying a tolerance is automatically routed to the data of appropriate freshness.

The combination of XML, hierarchical data organization, and effective response caching allows IrisNet to support large-scale wide-area sensor databases while providing interactive response times to user queries. In this paper, we overview IrisNet's database processing support. For further details on the XML processing and mechanisms needed to support querying and caching in IrisNet, we refer the reader to [12].

4 The SA Execution Environment

We envision a decentralized deployment of SAs on the IrisNet network, where loosely federated users will participate in IrisNet by hosting sensing code on their hosts. Loose federation is a basic requirement for a system intended to scale to the wide-area Internet; it would be unreasonable to assume that all participating nodes are controlled by the same administrative authority, with perfect trust for one another. Senselets, the sensor feed processing code running at SAs, may be mutually distrusting, and the author of a senselet and operator of an SA node may not trust one another perfectly. Moreover, bugs in senselets may (unmaliciously) exhaust the resources available on an SA, eliminating that SA's availability to IrisNet. To address these difficulties, IrisNet supports *limited protection* of senselets from one another, and of non-IrisNet applications (run by the SA owner on the SA node) from the entire set of IrisNet senselets executing on an SA.

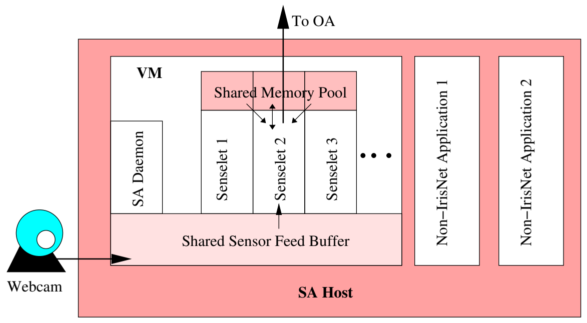


Figure 4: Execution Environment in SA

Moreover, a sensor feed may be of interest to multiple IrisNet users who use different services: *e.g.,* a video feed in a particular location may be used by one user to monitor parking spaces, and by 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 SAs among multiple senselets. Because senselets will often be CPU-intensive, and may perform identical processing on identical data, naively running multiple senselets in time-shared fashion may waste significant CPU resources needlessly on duplicated execution. To scale well when multiple senselets with overlapping computation share an SA, IrisNet *shares intermediate results* between senselets that are willing to do so.

In the following, after defining the structure of the SA execution environment in Section 4.1, we describe the details of how IrisNet meets the above two challenges. Section 4.2 presents the mechanism IrisNet uses to reduce the load on SA hosts by sharing computation across senselets, and Section 4.3 describes protection in IrisNet.

4.1 Filtering with Senselets

Figure 4 shows the execution environment in an SA host. All SA code runs within a single virtual machine (VM) (*e.g.,* VMWare or UML), that shares the SA's CPU alongside user applications. Inside the VM, an *SA Daemon* accepts commands (to download senselets, to start execution of senselets, *&c.*) from OAs. Each senselet runs as a separate process within the VM.

Senselets are written in the standard C and C++ programming languages[1]. These binary executables filter and process the raw sensed data available at an SA. The IrisNet execution environment on SAs provides sensor feed processing libraries with well-known APIs to be used by the 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. Raw sensed data are periodically copied into a circular shared memory buffer within the VM, readable by senselets; the sharing model for senselets is discussed in detail in Section 4.2.

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.

4.2 Cross-Senselet Sharing

Multiple senselets in an SA run continuously on the same sensor feeds, such that there may exist many common sub-tasks across the senselets. For example, consider the two senselets whose data flow graphs are shown in Figure 5. 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. More generally, a sequence of operations on a set of raw sensor data feeds $\{V\}$ can be represented as a directed acyclic graph (DAG), where the nodes with zero in-degree are in $\{V\}$, the remainder of the nodes

[1]In principle, they can be any executable code.

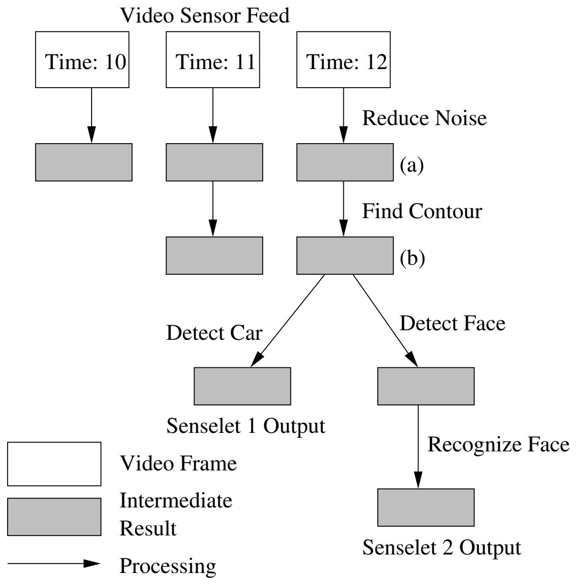


Figure 5: Computation DAGs for two senselets. The complete DAG is shown for the video frame at time 12. A few tuples of the computation DAGs for previous frames are also shown.

represent intermediate results, and the edges are the operations on intermediate results. If multiple senselets use the same sensor data feed set $\{V\}$, their corresponding DAGs can be merged into a single DAG referred to as the *computation DAG*. Figure 5 shows such a computation DAG where two scripts are processing the same sensor data with timestamp 12.

In general, we expect image processing primitives (*e.g.,* color-to-gray conversion, noise reduction, edge detection, *&c.*) 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 [16]. 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]. These examples suggest a large degree of shared computation across services we are currently considering.

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. IrisNet uses names of sensor feed processing API calls to identify commonality in execution, rather than attempting to determine commonality across *any* arbitrary piece of C code. Because most of a senselet's time is spent within the sensor feed processing APIs, using this simple mechanism to optimize across only those APIs will reduce computation and storage requirements significantly.

Two mechanisms are required for sharing intermediate results between senselets: a data store that is shared between separate senselets (which run as distinct processes), and an index whereby senselets can publish results of interest to other senselets, and learn of ones of interest to themselves. We describe these mechanisms in the following two sections.

4.2.1 Shared Buffering of Intermediate Results

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.

In IrisNet, intermediate results are generated and kept in shared memory regions so that all senselets can use them. The SA Daemon, which spawns senselets, allocates each new senselet a shared memory region. The SA execution environment provides senselets with a memory management library that allocates memory from the calling senselet's own shared memory pool. A senselet has read/write access to memory allocated from its own shared memory pool, but read-only access to memory allocated in other senselets' shared memory pools; the SA Daemon configures senselets' shared memories this way, with the support of the shared memory library [5]. This allocation strategy prevents one senselet from overwriting intermediate results generated by other senselets. Figure 4 shows that senselet 2 can read and write the memory allocated from its own shared memory pool, but can only read from other shared memories.

To generate intermediate results in the shared memory, we replace standard dynamic memory allocation calls in the sensor feed processing libraries with our shared memory allocation calls. Note that intermediate results are not self-contained – they often may contain pointers to other objects which may in turn contain additional pointers. These pointers, in general, are not meaningful across senselets running as separate processes. Fortunately, pointers to shared memory regions are valid for all senselet processes since they map each others' shared memory regions at identical addresses. This equivalence of pointers across address spaces is also essential for indexing the shared memory, as will be revealed in the next section. All intermediate results are marked with the timestamp of the original sensor feeds they are generated from.

When allocation of shared memory for a new result fails, IrisNet evicts an intermediate result from shared memory. The replacement policy for shared memory is to evict the item with the oldest timestamp. If multiple such results exist (because they all are from the same DAG), the one generated most recently is selected. The intuition here is that old results are relatively less likely to be used by other senselets, and within the same computation DAG, the ones generated more recently (farther down in the computation DAG) are less likely to be common across senselets.

The maximum amount of advantage that can be achieved from the commonality of computation across senselets depends on the size of the slack and the amount of shared memory allocated to store the intermediate results. In Section 7.3, we present experimental results measuring the effect of these two factors on system performance.

4.2.2 TStore: Indexing Intermediate Results

To make use of the shared memory store, senselets need an index for it, to advertise and find intermediate results. IrisNet indexes intermediate results as *tuples* in a *Tuple Store* (TStore), which is itself in a shared memory region mapped into all senselets' address spaces[2]. Tuples are of the form `(name, timestamp, result)`, where `name` is a unique name for the `result` computed from a sensor feed with timestamp `timestamp`. The `result` may contain a value (if the intermediate result is a scalar) or point to a shared memory address where that intermediate result is stored; recall that shared store pointer values are valid across all senselets. Conceptually, TStore is a black box with two operations: `Insert(tuple)`, which inserts a tuple into TStore, and `Lookup(tuple-name, time-spec)`, which

[2]In our current implementation, all senselets have read and write permissions to the TStore. However, we are moving to a model where only the SA daemon has write permission and performs all writes to the TStore.

finds tuples with the specified tuple-name and time-spec (timestamp and slack) in the TStore.

The names of intermediate results (*i.e.,* the `name` fields of tuples) must be consistent across senselets, uniquely describe results, and be easily computable. Recall that senselets are comprised of API function calls to libraries provided by the IrisNet SA platform. Senselets leverage the function names in this well-known library API to name their intermediate results for sharing with other senselets.

A tuple within TStore represents the result of applying a series of API function calls to some particular sensor feed. We name a tuple using its *lineage*, which is an encoding of the path from the original sensor feed to the result in the computation DAG. The encoding should preserve the order of the non-commutative function calls. IrisNet names the intermediate result produced by a function by hashing the concatenation of the names of the function and its operands. For example, the name of the tuple marked (b) in Figure 5 is the hash of the function name `saFindContour`, concatenated with the name of the tuple marked (a), concatenated with the names of other operands to `saFindContour`. Note that TStore may contain multiple tuples with the same name, but they will have different timestamps.

We implement TStore as a hash table keyed on tuple `name` fields. Within a hash chain, tuples are stored as a linked list in decreasing order of their timestamps. This ordering improves the performance of `Lookup` and `Insert` operations.

A senselet uses TStore by preceding calls to the sensor data processing libraries with `Lookup` calls for tuples with names for the appropriate function and data source, and the desired time-spec. If TStore contains a matching intermediate result previously computed by another senselet within the appropriate time range, `Lookup` returns the requested intermediate result from shared memory. Otherwise, the senselet calls the actual sensor data processing library function and stores the result in TStore with `Insert`.

Similarly, tuples are evicted from TStore when the corresponding intermediate results are evicted from shared memory, or when TStore itself exhausts storage for new tuples. The TStore tuple replacement policy for selecting a victim tuple is similar to that for intermediate results in shared memory.

4.3 Protected Execution

While the core SA code is trusted, senselets may not be, and users of IrisNet services may only be loosely federated with one another, or with the administrators of SAs. There are two relationships between code running on SAs of interest: senselet-to-senselet, and IrisNet-to-non-IrisNet. While we have discussed above the performance benefits of sharing data between senselets, a senselet should retain the option of refusing to use results from other senselets, or refusing to make its results available to other senselets. The owner of an SA node has an interest in isolating his own applications from downloaded senselets that share the SA node's CPU; even non-malicious bugs in senselets might otherwise exhaust resources on the SA node. We now describe IrisNet's support for protected execution, which addresses these needs.

IrisNet supports encapsulating all senselets in a single User-Mode Linux (UML) virtual machine [19], within which each concurrently executing senselet is a single process, as shown in Figure 4. UML is easily integrated into IrisNet, because its VM environment is binary-compatible with native Linux, on which IrisNet was initially developed. We run all senselets in a single VM instance because of the high memory and processing overhead of multiple VM instances executed concurrently; each VM runs a complete copy of the operating system.

Between senselets, there exists standard UNIX-style process protection, so that senselets cannot overwrite one another's address spaces. Shared memory buffers, used in sharing of intermediate results, may be mapped into multiple senselet processes. Each senselet controls whether it elects to make its results available to other senselets in such segments, or whether it elects to use results from other senselets' shared memories.

The UML VM that encapsulates senselet execution runs alongside native applications executed by the SA node owner. This separation allows an SA node owner to limit the CPU consumption of the senselets in the aggregate, to ensure an SA node devotes a particular portion of its execution time to the SA owner's tasks. Should a faulty senselet greedily consume CPU, the UML also permits the SA node owner to retain control of the SA. This degree of control is important in cases where an SA may be deployed in a location not easily accessible, such that rebooting the SA may be impractical.

The upcoming release of UML supports bounding the consumption of CPU, memory, disk, and network bandwidth for a UML VM. These resource limits will thus be enforceable for the set of senselets executing on an SA.

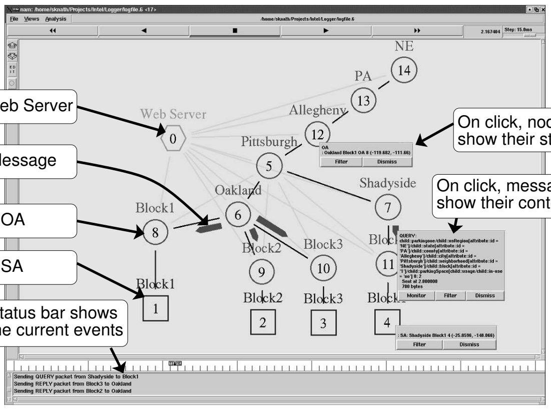
UML imposes one overhead on IrisNet beyond the expected virtualization overhead of any VM system, because it doesn't virtualize many sensor input devices, such as webcams. That is, only the host Linux instance (running on the bare hardware) can read from, *e.g.*, USB devices. To make sensor inputs available to senselets inside the guest UML VM instance, we've implemented a *tunnel writer* for the host Linux instance, that reads raw sensor inputs and sends them over a network socket. This socket connects to a corresponding *tunnel reader* process inside the UML VM, which writes the stream arriving on the socket into a shared memory accessible to senselets executing inside the UML VM. Copying the raw sensor data across the socket in this fashion is not required when senselets execute directly on the host Linux instance, and thus represents additional CPU overhead we pay for encapsulating senselets in a UML VM. We plan in future to add support to UML for direct reading of sensor input devices, to eliminate the data copy overhead of the tunnel reader and writer.

5 Service Monitoring

As can be seen over the previous sections, the interaction between the SAs and OAs can be quite complex. Monitoring a distributed service, such as a collection of OAs and SAs, for debugging, testing, and performance evaluation purposes is always challenging [20]. In this section, we describe the tools that IrisNet provides to aid service developers in this task. First, IrisNet provides mechanisms to log the events at each node (*e.g.* exchange of messages) and to gather logs to a central place. Second, IrisNet provides tools to replay the events and visualize them in a graphical interface.

5.1 Logging

Each OA independently logs the *events* triggered by the services running on it. The current prototype of IrisNet only logs message transmission and reception events at the OAs; however, the set of events that are logged can easily be extended. Information about the operation of the SAs is collected by recording the reception of SA messages at some OA. Each log entry includes at least the following: type of the message (Query, Sub-query, Reply etc.), sender and receiver names, name of the service related to the event, and query ID. Log entries also need to be timestamped so that distributed operations (*e.g.* a query followed by the response) can be replayed later in correct order. However, generating this timestamp raises several issues.



```
Sending QUERY packet from Shadyside to Block1
Sending REPLY packet from Block3 to Oakland
Sending REPLY packet from Block2 to Oakland
```

Figure 6: Animation of the events

Since different nodes in the system may not be time synchronized, physical timestamps may not represent the global causal order of the events. To overcome this problem, IrisNet uses logical clocks instead of physical timestamps to maintain the event order. Each OA maintains a logical clock (Lamport clock [23]) which is incremented on each event. Messages sent by an OA are timestamped with the value of the clock at the time they were sent. On receiving a message, an OA updates its timestamp to be the maximum of the received timestamp and its current timestamp. Such a clock maintains the causal order of the events, which can later be used to replay the events.

In our system, the collected logs are sent to a central server (maintained by the service developer) as they are generated by the OAs.[3] In addition, we plan to have each OA store a sliding window of the log entries in its local XML database. When needed, a developer can collect these logs using IrisNet's query processing support. This would make it possible to retrieve the logs for a failed request by querying all the events generated by the failed query (specified by the query ID) for a specific service (*e.g.* PSF). This approach may be more attractive since the logs are only collected when needed and the possible bottleneck of central log collection is avoided.

[3]Note that, even with this setting, physical timestamps by the central server may not provide the correct event ordering.

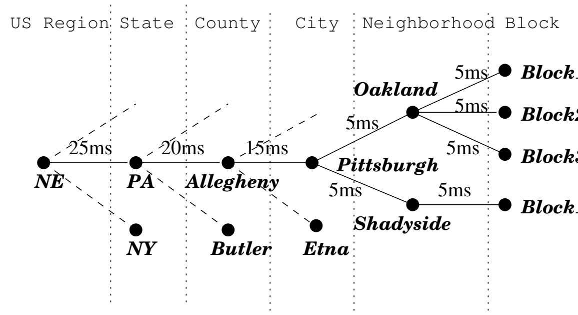
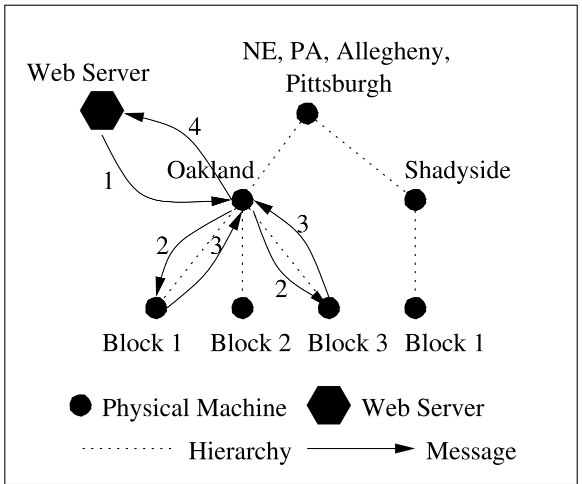


Figure 7: OA Hierarchy for the PSF service

5.2 Replay and Visualization

Since the log of events for many services will have a large volume, they may be difficult to examine for the purposes of debugging. IrisNet provides two tools to process and replay the logs once they are collected. The first tool is the *Trace Generator* which sorts the logs in ascending order of their logical timestamps and generates NAM-style [4] trace. The second tool, the *Trace Animator*, displays the topology of OAs and SAs as a graph and animates the events of message exchanges among nodes. The trace animator is created by modifying NAM to support a few extra commands (e.g., to provide a view of the content cached at a node). Figure 6 shows a snapshot of this animator, replaying the logs collected from the PSF service described in Section 6.



/usRegion[@id='NE']/state[@id='PA']
/county[@id='Allegheny']/city[@id='Pittsburgh']
/neighborhood[@id='Oakland']/block[@id='1' OR
@id='3']/parkingSpace[available='yes']

Figure 8: Processing of a query in the PSF service. One physical machine contains one or more nodes in the hierarchy. The numbers beside the messages show their relative order. The corresponding XML query is given below.

6 An Example Service

We have built a working prototype of the IrisNet system and a PSF service using commodity, off-the-shelf PC desktops, laptops, and webcams. The objective of the PSF service is to use feeds from cameras installed in parking lots in a metropolitan area and allow users to make queries about the availability of parking spots at a particular location. Our prototype of this service uses simulated parking lots on a tabletop with toy cars. This simulated setup was chosen in the interest of having a controlled experimental environment. Here, we describe the implementation details of PSF.

6.1 Database processing

A service developer creates the XML schema which defines, along with other information, the hierarchy used by the service. Figure 7 shows part of the hierarchy used in our current deployment of the PSF service and Figure 8 shows one instance of the physical distribution of the nodes in the hierarchy, where (a) the four blocks corresponding to the parking lots (leaves in the hierarchy) are mapped onto one OA each, (b) the two neighborhoods, Oakland and Shadyside, are mapped onto one OA each, and (3) the rest of the nodes in the hierarchy are mapped onto one OA. The OAs maintain a distributed XML database based on the supplied schema. This database includes dynamically changing parking space availability information from the SAs as well as static data describing each of the parking spots monitored (e.g., meter restrictions on the spot).

Figure 8 shows how query processing is done for the PSF service. The illustrated query asks for the available parking spots in blocks 1 and 3 in Oakland. The web server initiates the query on behalf of the user. The query is first sent to the Oakland OA, which is the LCA for the supplied query string. The Oakland OA then sends subqueries to its children OAs, aggregates and caches the responses, and sends the final answer to the web server.

6.2 Sensor feed processing

In our current prototype, we use four webcams to monitor four parking lots – one at each block in the hierarchy shown in Figure 7. The webcams are attached to laptops which act as the SAs.

The block-level OAs upload image processing senselets to each of their associated SAs. The senselets are written using the Intel OpenCV library [3], which performs the necessary image processing. The OpenCV library was modified to make effective use of the tuple store described in Section 4.2. The `saAbsDiff` (Figure 9) call shows how the OpenCV calls have been modified. Before calling the `cvAbsDiff` function provided by the OpenCV library, the `saAbsDiff` function uses the `getName` and `Lookup` calls to determine if the result of the call is already available in the tuple store. Similarly, if the result is computed, the `Insert` call is used to add the result to the tuple store.

The senselets for the PSF application process the webcam video feeds to determine which parking spots are available and send this availability information to the appropriate OAs. The processing is done via a sequence of calls to the OpenCV library. The senselet is configured using a fixed set of background images of each parking spot. The senselet takes the difference of the current image and the fixed background image, and uses a threshold to decide if the parking spot is empty or full. While the senselet could use more sophisticated image processing algorithms[4], our simple image processing code is sufficient for our tabletop environment and to demonstrate the important features of the the IrisNet infrastructure.

6.3 The frontend

The web frontend for this service presents the user with a form that he/she can fill out to specify a location and any other constraints (*e.g.*, that the parking spot must be covered) for the desired parking spot. The frontend uses IrisNet to find an available parking spot close to the desired location that satisfies the user's constraints, and then uses the Yahoo Maps Service [8] to find driving directions to that parking spot from user's current location. The driving directions and destination parking spot are displayed in response to the user query.

[4]A possible algorithm [13] would be to maintain different statistical models for each pixel in the background image based on the time of day. This model could more easily compensate for changes in sunlight, shadows, etc.

```
void cvsAbsDiff(IplImage* srcA,IplImage* srcB,
                                IplImage* dst);

void saAbsDiff(TimeSpec ts, IplImage* srcA,
              IplImage* srcB, IplImage* dst) {
  // pre-processing
  name = getName(srcA, srcB,SA_ABSDIFF);
  foundTuple = Lookup(name, ts);
  if (foundTuple != NULL) {
     dst = foundTuple->result;
     return;
    }

  // call the OpenCV API
  cvAbsDiff(srcA, srcB, dst);

  // post-processing
  tuple->name = name;
  tuple->result = dst;
  Insert(tuple);
  return;
}
```

Figure 9: Wrapped up version of an OpenCV API shared across scripts

The driving directions are continuously updated as the user drives towards the destination, if the availability of the parking spot changes, or if a closer parking spot satisfying the constraints is available. We envision that a car navigation system using the PSF service could periodically repeat the query, along with its current GPS coordinates, as the user nears the destination.

7 Experimental Results

We present a performance evaluation of the IrisNet infrastructure that seeks to answer the following four questions:

1. What are the performance gains in intelligently filtering at the SAs *vs.* performing the work at the OAs (Section 7.2)?

2. What is the cost or gain of cross-senselet sharing (Section 7.3)?

3. What is the raw performance of the prototype on simple queries, in terms of processing time, communication time and querying time (Section 7.4)?

4. What is the overhead of providing protection using a VM (Section 7.5)?

Method	Bandwidth (bps)
Raw camera feed (30 FPS)	221184000
1 FPS sampling	7372800
Compressing in SA (1 FPS)	143000
Filtering in SA (1 FPS)	256

Figure 10: Bandwidth requirements for data sent from the SA to the OA under four scenarios.

7.1 Experimental Setup

In our experiments, 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 are written in C and OAs are written in Java. We run the off-the-shelf Xindice XML database [1] at OAs. 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 elect to sample frames at a lower rate. For example, the PSF service we examine herein reads one frame per second. All measurements presented are for experiments *without* UML, with the exception of Section 7.5, where we specifically measure the overhead of UML on senselet execution.

We study the PSF service described in Section 6 that uses the hierarchy shown with solid lines in Figure 7. The nodes in the hierarchy are distributed among seven machines on a LAN as shown in Figure 8. To simulate wide-area network latencies, we add appropriate artificial delays on the links between OAs, as shown in Figure 7. Each SA runs on a separate host with a connected webcam.

7.2 Processing Webcam Feeds

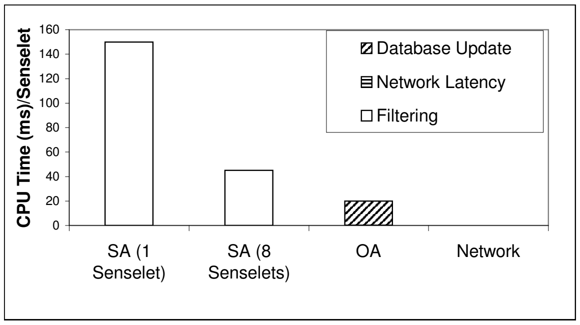
In our first set of experiments, we show the effectiveness of filtering sensor feeds at the SAs. We compare SA filtering with filtering at the OAs. For OA filtering, SAs send compressed video frames to the OAs, who then decode the frames, process them with the senselet code, and update their local databases. We use the *FAME* [2] library for encoding the video frames into MPEG-4 at the SAs, and the *SMPEG* [6] library for decoding the frames at the OAs. We assume that the OA database is updated once per second.

Figure 10 shows how filtering at the SAs reduces the required bandwidth between SAs and OAs. The first two rows in the figure show numbers estimated using 640×480 RGB video frames, while the last two rows show numbers from actual measurements. Although cameras feed a large volume of raw video data to the SAs[5], our PSF service samples the frames at only 1 frame per second. Still, sending these uncompressed frames to the OAs demands a vast amount of bandwidth. The figure reveals that encoding the frames in MPEG-4 format reduces the traffic. While the compression ratio depends on the dynamic behavior of the video feed, we found the average compression ratio to be approximately 50. However, filtering the frames in the SAs produces the least volume of traffic—as low as a few bytes per frame.

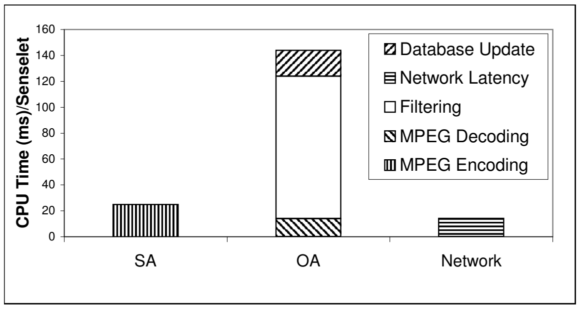
Figure 11 shows the breakdown of time spent on stages of extracting information from a video frame and updating the database under the strategies of filtering in SAs and OAs, respectively. Here we measure the execution time required to run one senselet on the SA, 8 senselets on the SA (the scenario is described in the next section), and one senselet on the OA. Not only does filtering at SAs save network bandwidth; it also parallelizes sensor feed processing across SAs, rather than concentrating processing at OAs. Figure 11(b) shows that an OA takes the same order of time to process a video frame as an SA, but intuitively, aggregation of feeds from many SAs at an OA can easily overwhelm the computational capability of even the fastest processor. This poor scaling is exacerbated in the case where multiple OAs run on the same physical machine. Figure 11(a) also reveals that while filtering at SAs puts high load on SA hosts, even moderate sharing across the senselets reduces the per-senselet computational load significantly. To wit, the second bar in the graph shows that running 8 concurrent senselets and enabling result sharing across them significantly reduces the per-senselet costs.

All these results suggest filtering at SAs is far more scalable than filtering at OAs. The advantage is twofold: first, the network and computational loads are distributed over the SAs (expected to outnumber the OAs, as multiple SAs may report to the same OA), and second, co-locating senselets at SAs creates the opportunity to share computation among senselets.

[5]Most webcams compress the video to less than 12Mbps to transfer it across a USB bus.



(a) Filtering in SAs



(b) Filtering in OAs

Figure 11: Breakdown of time spent extracting information from video frames and updating the database.

7.3 Effectiveness of Sharing among Services

We now report measurements of the performance of cross-senselet sharing. Through these experiments, we try to understand the amount of overhead we introduce by wrapping the OpenCV image processing APIs in TStore calls, and the performance gains we achieve from sharing across senselets.

7.3.1 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 and the sequences of major image processing operations they perform are as follows:

- Parking Space Finder 1 (PSF1): Get current frame → Reduce noise → Convert to gray → Find contour → Compare contours → ⋯
- Parking Space Finder 2 (PSF2): Current frame → Reduce noise → Convert to gray → Get image parts → Subtract background → ⋯
- Motion Detector (MD): {Current frame → Reduce noise → Gray, 1 second old frame → Reduce noise → Gray} → Subtract images → ⋯
- Person Tracker (PT): Current frame → Reduce noise → Gray → Find Contour → Get image parts → Subtract background → ⋯

The PSF service described in Section 6 uses the senselet PSF2.

Operation	Time (ms)
`cvCvtColor()`	1.78
`cvAbsDiff()`	2.85
`cvFindContour()`	4.95
`Lookup() + Insert()`	0.02

Figure 12: Average time required by different operations.

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.

7.3.2 Overhead of wrapping the APIs

Figure 12 shows the execution times for a few typical functions in the OpenCV API and the overhead of wrapping them. The numbers reported in the figure are the averages of performing the operations on a lightly loaded SA on 20 different 640×480 24-bit images. A typical OpenCV API takes 1-5 ms, whereas the overhead we introduce by wrapping them is around 0.02 ms, less than 1% of the time taken by the original API in most of the cases. As we reveal later in this section, we make significant gains for this small cost.

7.3.3 The amount of slack and shared memory

For all the experiments, slack is defined as a percentage of senselet's execution interval.

The optimal size of shared memory needed to achieve the maximum sharing depends on a senselet's sensor feed access pattern, execution pattern (deadline and slack values), and intermediate result generation rate. For a small shared memory, arrival of a new intermediate result may force the discarding of an old intermediate result, before that prior result has been used by other senselets. In these cases, the prior result will be recomputed redundantly. Let us assume that around $1/k$ (k is a constant) of the intermediate results generated by one senselet will eventually be used by some other senselet. In the case where most senselets use input from the same sensor data feed, we estimate that a senselet should allocate $(\text{Period}_{\text{max}}/\text{Period}_{\text{senselet}} \times \text{Size}_{\text{IR}})/k$ bytes of shared memory, where $\text{Period}_{\text{max}}$ is the maximum of the periods of all the concurrent senselets, $\text{Period}_{\text{senselet}}$ is the per-iteration running time of the senselet under consideration, and Size_{IR} is the size of the intermediate results the senselet generates in each execution round for other scripts to share.

7.3.4 The effect of sharing on CPU load

Figures 13(a) and 13(b) show that cross-senselet sharing significantly reduces the CPU load on SAs. In accordance with intuition, 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 for the same set of senselets, where the ideal load 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 TStore 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 argument 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 13(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 13(b) reveals that as the shared memory size increases (k decreases), the performance of sharing again approaches the ideal case, as shared memory holds progressively more results for later re-use.

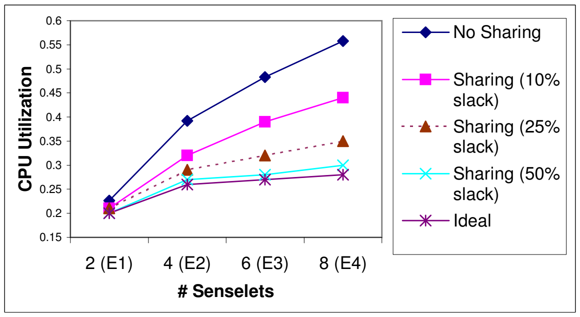
7.3.5 The effect of sharing on missed deadlines

As described in Section 4.1, 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. Figures 14(a) and 14(b) show how the number of missed deadlines increases with the number of 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. As before, the number of missed deadlines can be reduced by using longer slack times (Figure 14(a)) and larger shared memories (Figure 14(b)).

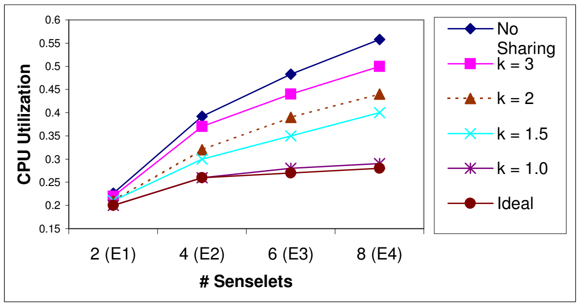
7.4 Query Execution

Despite IrisNet's distributed nature and hierarchical queries, the system offers response times for queries well within the range demanded by users of interactive applications. For a simple query in the PSF service, asking for available parking spots in block 1 of Oakland, IrisNet responds in approximately 70 ms. Even if we artificially route the same query to an ancestor OA two levels above the relevant block's OA in the OA hierarchy, the response time only increases to 150 ms. These results suggest that even queries that must traverse multiple layers of the OA hierarchy receive reasonable interactive response times. They also suggest that routing queries directly to the OA with the desired data significantly reduces response time, as was our goal in the design of IrisNet.

For further details concerning IrisNet's query processing performance, we refer the reader to [12].

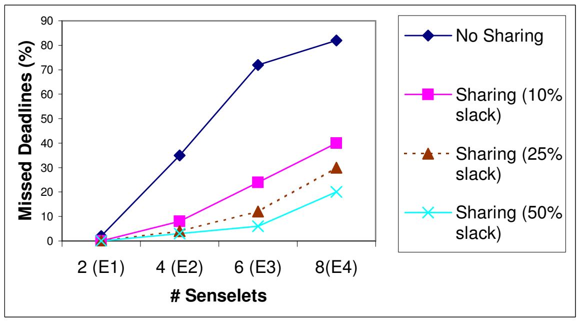


(a) Effect of slack on CPU time ($k = 2$)

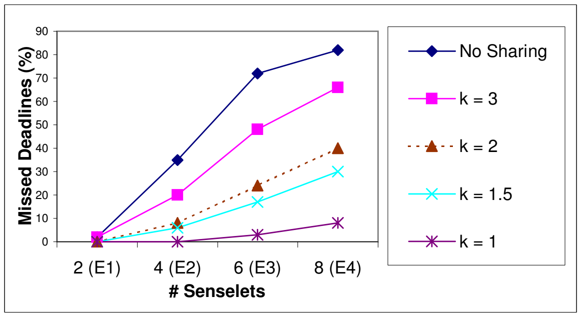


(b) Effect of memory size on CPU time (slack = 10%)

Figure 13: Plots showing the effect of sharing on CPU time.



(a) Effect of slack on missed deadlines ($k = 2$)



(b) Effect of memory size on missed deadlines (slack = 10%)

Figure 14: Plots showing the effect of sharing on missed deadlines.

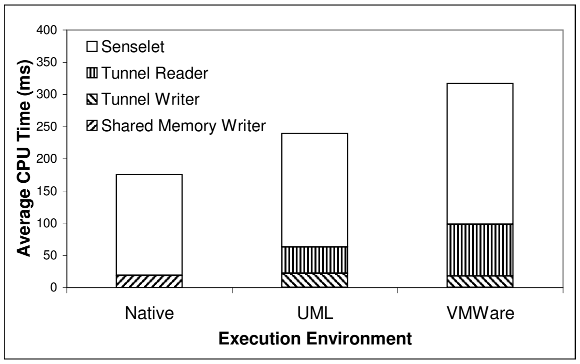


Figure 15: VM Performance Comparison

7.5 Overhead of Protected Execution

The entire evaluation presented thus far has been for senselets running natively on the host machine. We now present results to argue that the additional computational overhead of running senselets in a virtual machine is reasonable, in exchange for added control over the resource consumption of senselets running alongside non-IrisNet applications.

Figure 15 compares the time required to process one frame of a video feed in a senselet when running natively, *vs.* under a UML VM. We also include measurements for VMWare, a commercial VM emulator, for comparison. As compared with native execution, the parking space finder senselet's execution slows by 13 and 40 percent under UML and VMWare, respectively. This modest slowdown can be attributed to the math-intensive nature of senselets; it's system calls that have the greatest overhead under VMs, whereas arithmetic operations are relatively unaffected by virtualization. Note that the tunnel reader portion of execution under the two VMs may be reduced by virtualizing the camera device for direct use by senselets, rather than tunneling over network sockets, as described in Section 4.3.

8 Related Work

Several of the subcomponents of IrisNet share techniques and goals similar to those of other systems. Many of these similarities have been highlighted throughout this paper. Here, we describe large systems with similar goals to IrisNet's. These efforts fall into a few categories: 1) work on numerous but geographically limited deployments of resource-constrained sensors; 2) work on creating large queriable databases with distributed updates; and 3) work on limited deployments of intelligent surveillance systems.

The work on networked sensors 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 of past efforts on resource-constrained motes, earlier key contributions have been in the areas of tiny operating systems [18] and low-power network protocols [22]. Mote-based systems have relied on techniques such as directed diffusion [17] 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 [24], to address the limitations of sensor motes. None of this work considers sensor networks with intelligent sensor nodes, high-bitrate sensor feeds, and global scale.

The distributed database infrastructure in IrisNet shares much in common with a variety of large-scale distributed databases. For example, DNS [26] 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 [27] 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.

A number of large research projects have explored automated video surveillance. The Video Sureveillance and Monitoring (VSAM) [10] project at CMU is a good example of the work in this area. Most of these efforts 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 ours, as we focus on wide-area scaling and development tools, rather than sensor feed processing algorithms.

9 Conclusions

The goal of the IrisNet system is to enable the development of large-scale sensor-based services. We have discussed features in IrisNet that greatly simplify many common tasks in these services, such as collecting, filtering and combining sensor feeds, and enabling distributed queries with reasonable response times.

IrisNet achieves scalability and flexibility through the effective use of service-specific processing and filtering of the sensor feeds at the SAs. This approach introduces the additional challenges of eliminating duplicate or redundant sensor processing at the SAs and isolating a senselet from the SA software and from other senselets. IrisNet makes use of data structures in shared memory segments to pass results between different senselets. We use a combination of process protection and virtual machine technology to securely isolate the senselets.

IrisNet supports low latency query times through the novel marriage of hierarchical schemas, OA hierachies, DNS-style names extracted from the hierarchical prefixes of queries, and partial match caching within the hierarchy.

We have reported performance numbers demonstrating the efficacy of our approach. We have also described an example service (Parking Space Finder) to illustrate how IrisNet makes the development of services simpler.

Three areas that we are currently exploring are service discovery, integration with smart dust sensors, and privacy. We plan to use service discovery to enable the dynamic matching of OAs to SAs with appropriate sensor coverage. We also plan to explore how the capabilities of smart dust sensors, such as distributed query processing, can be used effectively to connect them to IrisNet. Finally, privacy is a concern in any wide deployment of rich sensor recording devices. We believe that the SAs in our architecture can help address this concern. For example, we can limit the senselet's access to the raw sensor data or inspect the data transmitted from the SA and remove sensitive information.

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

References

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

[2] The FAME project. http://fame.sourceforge.net/.

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

[4] Nam: The network animator. http://www.isi.edu/nsnam/nam/.

[5] OSSP mm shared memory allocation library. http://www.ossp.org/pkg/lib/mm/.

[6] SDL Mpeg Player Library. http://www.lokigames.com/development/smpeg.php3.

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

[8] Yahoo! maps and driving directions. http://maps.yahoo.com/.

[9] BONNET, P., GEHRKE, J. E., AND SESHADRI, P. Towards sensor database systems. In *MDM* (2001).

[10] COLLINS, R., LIPTON, A., FUJIYOSHI, H., AND KANADE, T. Algorithms for cooperative multisensor surveillance. *Proceedings of the IEEE 89*, 10 (Oct. 2001), 1456–1477.

[11] CULLER, D., BREWER, E., AND WAGNER, D. Berkeley Wireless Embedded Systems (WEBS). http://webs.cs.berkeley.edu/.

[12] DESHPANDE, A., NATH, S., GIBBONS, P. B., AND SESHAN, S. Cache-and-query for wide area sensor databases. Submitted for publication.

[13] ELGAMMAL, A., DURAISWAMI, R., HARWOOD, D., AND DAVIS, L. S. Background and foreground modeling using nonparametric kernel density estimation for visual surveillance. In *Proceedings of the IEEE* (July 2002), vol. 90, pp. 1151–1163.

[14] ESTRIN, D., GOVINDAN, R., AND HEIDEMANN, J. SCADDS: Scalable Coordination Architectures for Deeply Distributed Systems. http://www.isi.edu/scadds.

[15] ESTRIN, D., GOVINDAN, R., HEIDEMANN, J., AND KUMAR, S. Next century challenges: Scalable coordination in sensor networks. In *MOBICOM* (1999).

[16] HEBERT, M. http://www.cs.cmu.edu/ hebert/,Personal communication, November, 2002.

[17] HEIDEMANN ET AL., J. Building efficient wireless sensor networks with low-level naming. In *SOSP* (2001).

[18] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. System architecture directions for network sensors. In *ASPLOS* (2000).

[19] HOXER, H. J., BUCHACKER, K., AND SIEH, V. Umlinux - a tool for testing a linux system's fault tolerance. In *LinuxTag 2002* (Karlsruhe, Germany, June 2002).

[20] JOYCE, J., LOMOW, G., SLIND, K., AND UNGER, B. Monitoring distributed systems. *ACM Transactions on Computer Systems 5*, 2 (May 1987), 121–150.

[21] KAHN, J., KATZ, R. H., AND PISTER, K. Next century challenges: Mobile networking for 'smart dust'. In *MOBICOM* (1999).

[22] KULIK, J., RABINER, W., AND BALAKRISHNAN, H. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks . In *MOBICOM* (1999).

[23] LAMPORT, L. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 27*, 7 (July 1978), 558–565.

[24] MADDEN, S., AND FRANKLIN, M. J. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE* (2002).

[25] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. Tag: A tiny aggregation service for ad hoc sensor networks. In *OSDI* (2002).

[26] MOCKAPETRIS, P. V., AND DUNLAP, K. J. Development of the Domain Name System. In *SIGCOMM* (1988).

[27] WAHL, M., HOWES, T., AND KILLE, S. Lightweight Directory Access Protocol (v3). Tech. rep., IETF, 1997. RFC 2251.