A Distributed Filtering Architecture for Multimedia Sensors

Suman Nath*, Yan Ke*, Phillip B. Gibbons†, Brad Karp†, Srinivasan Seshan*
*Carnegie Mellon University †Intel Research Pittsburgh
Pittsburgh, PA-15232, USA
{sknath, yke, srini}@cs.cmu.edu, {phillip.b.gibbons, brad.karp}@intel.com

Abstract

The use of high bit-rate multimedia sensors in networked applications poses a number of scalability challenges. In this paper, we present how IRISNET*, a software infrastructure for authoring wide-area sensor-enriched services, supports scalable data collection from such sensors by greatly reducing the bandwidth demands. The architecture makes a number of novel contributions. First, it enables the use of application-specific filtering of sensor feeds near their sources and provides interfaces that simplify the programming and manipulation of these widely distributed filters. Second, its sensor feed processing API, when used by multiple different services running on the same machine, automatically and transparently detects repeated computations among the services and eliminates as much of the redundancy as possible within the soft real-time constraints of the services. Third,* IRISNET *distinguishes between trusted and untrusted services, and provides mechanisms to hide sensitive sensor data from untrusted services. Using implementations of a number of real world sensor-enriched services on* IRISNET*, we present an evaluation of the benefits of our distributed filtering architecture.*

1 Introduction

The availability and cost of multimedia sensor hardware, such as cameras and microphones, has improved dramatically over the past several years. In fact, such sensors are now regularly incorporated into existing devices such as PCs, laptops and cell phones. While the state of sensor hardware has progressed rapidly, the software needed to make a collection of these devices useful and accessible to applications is still sorely lacking. This lack of a suitable standardized infrastructure of hardware and software makes authoring and deploying sensor-enriched services an onerous task, as each service author needs to address all aspects of data collection, sensor feed processing, sensing device deployment, *etc*.

An example of a sensor-enriched application we would like to enable is a Person Locator service that takes sensor feeds from cameras, indoor positioning systems, smart badges, *etc.*, processes these feeds to determine individuals locations, and organizes the collected position information to answer user queries (with appropriate attention to privacy). Several services could use the same sensor feeds simultaneously. For example, a Parking Space Finder service, which locates available parking spaces near a user's destination, may use a subset of the cameras used by the Person Locator—those overlooking parking lots—in order to determine parking space availability. Authors of these services could benefit from a software infrastructure that aids in collecting and processing sensor feeds, as well as organizing the resulting data and handling user queries. Our system, called IRISNET (Internet-scale Resource-Intensive Sensor Network), handles both these needs. IRISNET is the first system we know of that is tailored for developing and deploying new sensor-enriched Internet services on a shared infrastructure of rich sensors. In this paper, we describe IRISNET's approach to simplifying the task of sensor data collection. Details of IRISNET's support for query processing can be found in [10, 12, 20].

IRISNET's data collection component must address the following requirements:

- **Use of rich, shared data sources.** IRISNET must enable an infrastructure where such sensors can be shared by a number of simultaneously operating services.
- **Scalability up to Internet size.** IRISNET must scale to support a large number of simultaneous users, services and sensors. In addition, it must accommodate a wide heterogeneity in the type and ownership of the sensors. Despite this scaling, developers should be able to use this Internet-scale sensor collection as a seamless platform on which they can deploy services.
- **Efficient use of bandwidth.** IRISNET must support sensors that may be connected to the Internet via low-bandwidth wireless links. Even those that have better connectivity may not be able to support the transfer of multimedia streams for many concurrently running services.

IRISNET addresses these challenges through the use of *application-specific filtering* of sensor feeds at the source. In IRISNET, each service processes its desired sensor feeds on the CPU of the sensor nodes where the data are gathered. This distributed filtering 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. Our experience with an IRISNET prototype has shown that many applications require less than a hundred bytes per second of communication after post processing.

While it solves some problems, this filtering approach creates a new challenge: many services may have interest in the same sensor feeds and their associated sensor feed processing may place excessive demands on the computation resources of the sensor node. To reduce the computation demands of this approach, we take advantage of the observation that sensor feed processing is a relatively narrow domain and, as a result, many services require similar processing of the sensor feed. IRISNET includes a mechanism for sharing results *between* sensing services running on the same node. Our results show that this approach makes computation demand scale sub-linearly with the number of applications (*e.g.*, in one representative scenario, eight simultaneous applications sharing a sensor node result in only twice the computation load of running one of the applications in isolation).

Finally, while this basic design solves the scalability challenges of sensor data collection, its provisions for easy access to sensor data raises a number of privacy concerns. To partially mitigate this problem, IRISNET distinguishes between trusted and untrusted services on each node, with differing privileges for running code and accessing sensor data on the node.

The rest of the paper is organized as follows. Section 2 briefly describes the architecture of IRISNET. Section 3 provides a description of the programming environment of distributed filtering in IRISNET. We describe how IRISNET addresses scalability and privacy challenges in Section 4 and Section 5, respectively. Section 6 presents the evaluation of our design and implementation. We describe related work in Section 7 and conclude in Section 8.

2 The IRISNET Architecture

In this section, we describe the basic two-tier architecture of IRISNET (Figure 1), 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-enriched 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.

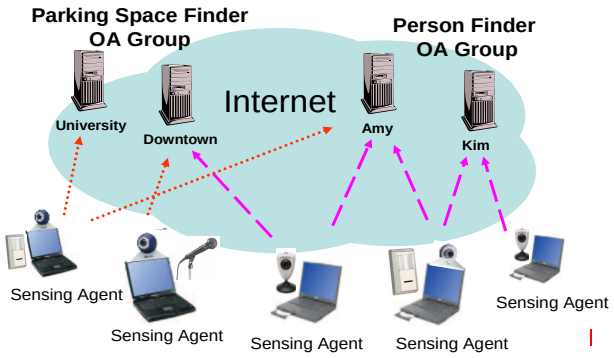


Figure 1. IRISNET **Architecture**

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 spaces for a Parking Space Finder service). In our deployments, an OA is typically a well provisioned PC with a fast connection to the Internet. 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. Users can use XPath [7], a standard XML query language, to query the sensor database. The design of the OAs poses a number of challenges including distributed query processing, caching, data consistency, data placement, replication and fault tolerance, *etc*. The details of how IRISNET addresses these challenges can be found in [10, 12, 20].

2.2 SA Architecture

SAs collect raw sensor data from a number 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. In our deployments, an SA is typically a laptop with one or more such sensors and either a wireless or wired connection to the Internet.

One key challenge is that transferring large volumes of data to the OAs can easily exhaust the resources of the net-

work. IRISNET relies on sophisticated processing and filtering of the sensor feeds at the SAs to reduce the bandwidth requirements. To greatly enhance the opportunities for bandwidth reduction, this processing is done in a *service-specific* fashion. IRISNET enables service authors to upload programs, called *senselets*, that perform this processing. Authors upload senselets 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 required bandwidth by orders of magnitude, *e.g.*, the senselets used by the Parking Space Finder service reduce the high volume video feed to a few bytes of available parking space data per time period.

Many sensors can be actuated by software via a control interface. This actuation can initiate, stop, or configure the data collection. Examples of such sensors include a camera whose viewing angle and focus point can be controlled with software commands, a robot that can be instructed to approach an object and take pictures of it, *etc*. In addition to filtering sensor data, senselets interface with an SA's actuator interfaces to configure and control data collection.

The use of senselets raises three new questions: (1) What programming environment does IRISNET provide for the senselets?, (2) How does IRISNET enable scaling to a large number of senselets running on the same SA?, and (3) How does IRISNET protect sensor data from untrusted senselets? We address these three questions in Section 3, Section 4, and Section 5, respectively.

2.3 Authoring a Service in IrisNet

To author a sensor-enriched service on IRISNET, a service author first creates the sensor database schema that defines the attributes, tags and hierarchies used to describe and organize sensor readings. She then writes senselet code for the SAs with sensor coverage relevant to the desired sensor service. This senselet code converts raw sensor feeds into updates on the database defined by the schema. Finally, she provides a user interface for end users to access the service, which converts user requests into XPath queries. These simple steps highlight how IRISNET makes it easy to create and deploy new services. IRISNET seamlessly handles many of the common tasks within sensor-enriched services, such as data collection, query processing, networking, caching, load balancing, fault tolerance, and resource sharing.

A number of IRISNET-based services are being developed and deployed by IRISNET researchers and others. The services include the aforementioned Parking Space Finder Service, an Ocean Monitoring Service that uses cameras deployed along the Oregon coastline to monitor useful nearshore oceanographic events (*e.g.*, rip tides and sand-bar formation), *etc.*[1] Details of these services can be found in [12].

[1]We have also implemented a service named IrisLog [4] that uses IRISNET and software sensors (*e.g.*, `/proc`) to monitor the network and the host machines of a large infrastructure.

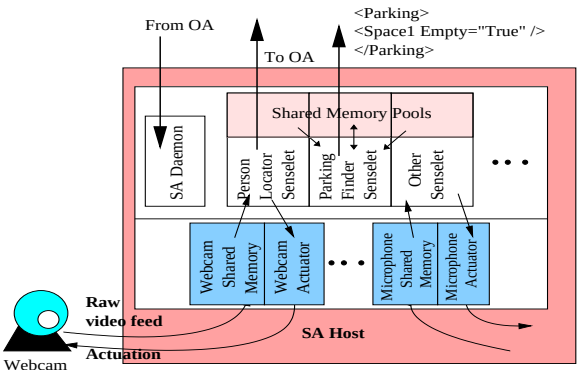


Figure 2. Execution Environment in SA

3 The SA Execution Environment

This section describes the basic execution environment of an SA in IRISNET (Figure 2).

3.1 Controlling Senselets

Each senselet runs as a separate process, collects sensor data from the SAs, filters them, and finally sends the filtered information to the OAs. Senselets can be any executable code and are typically written using standard C and C++ programming languages.

IRISNET provides a set of senselet manipulation APIs by which service authors can interact with SAs to upload, start, and control senselets. In the simplest case, service authors can use these APIs to interact with a single specified SA. To facilitate manipulating multiple senselets in batch, IRISNET interfaces these APIs with its query processing feature such that a service author can call the APIs with a selection query that selects the intended subset of SAs (*e.g.*, all SAs in Pittsburgh). This greatly simplifies the task of managing a widely deployed service.

3.2 Programming Interfaces

While a senselet can be an arbitrary executable, there are a number of important programming interfaces that it must use. These interfaces provide support for a senselet's access to the sensors/actuators, its filtering of the collected sensor data and its scheduling for CPU resources.

Sensor Access. IRISNET exposes the sensors and actuators through well defined interfaces so that senselets can interact with them. For example, the SA places all readings from a sensor into a shared memory segment associated with that sensor[2]. Senselets gain access to the sensor feed by mapping the shared memory segment into its address space.

[2]IRISNET also supports an HTTP based interface compliant with [23].

This interface is especially suitable for high bandwidth sensor feeds (*e.g.*, video) since it minimizes data copying. The shared memory segment keeps a sliding window of sensor data, annotated with relevant metadata (*e.g.*, timestamps), so that senselets can randomly access them for further processing. This well defined interface also hides from the senselets the details and heterogeneity of the drivers by which SA hosts interact with the sensors.

Filtering Libraries. IRISNET also provides sensor feed processing libraries with well-known APIs to be used by the senselets. For example, senselets can perform image processing tasks on video data by using an IRISNET-customized version of the OpenCV library [3]. 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. The computation outside the libraries largely implements service-specific intelligence. For example, while the OpenCV library performs high-level tasks such as edge, object and face detection, the senselet must decide which objects to transmit to the OAs and possibly how to reconfigure data collection when objects are detected. Senselets may also include code to react to external events (*e.g.*, the Parking Space Finder senselet may start archiving video feed when a security alarm is raised).

CPU Scheduling. A typical senselet is written so as to to achieve *soft* real time behavior. A senselet uses periodic deadlines for completing computations, but associates a *slack time*, or tolerance for delay, with these deadlines. A senselet periodically reads a sensor feed, 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. Note that the slack time allows the senselets to make only an approximate guess of its sleeping time between two deadlines. While we use standard UNIX scheduling (*i.e.*, not a real-time scheduler) on the SAs, this typical behavior provides a simple way for senselets to yield computation and provides a metric by which we can evaluate the behavior of an SA schedule (how often the $deadline + slack$ is violated).

4 Scalable Filtering

In this section, we discuss how an SA can support a large number of computationally intensive senselets. This is especially critical as we expect some sensor feeds to be much more popular than others.

We exploit the following observation to achieve scalability. In general, we expect sensor feed processing primitives (*e.g.* on video streams, color-to-gray conversion, noise reduction, edge detection, *etc.*) to be reused heavily across

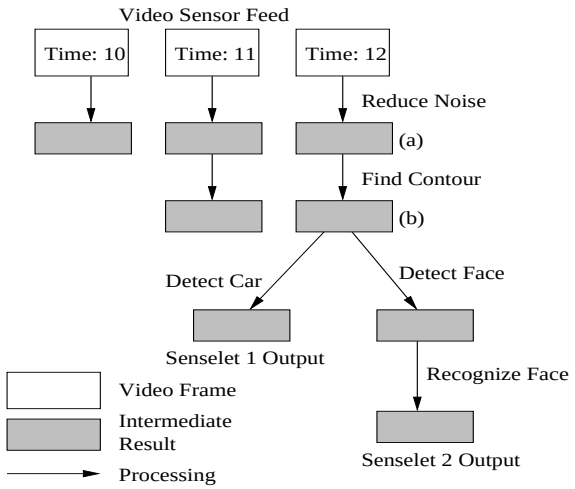


Figure 3. Cross senselet sharing between Parking Space Finder and Person Locator.

senselets working on the same sensor feed or video stream. If multiple senselets perform very similar jobs (*e.g.*, tracking different objects), *most* of their processing would overlap [13]. 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 [11]. This illustrates a large degree of shared computation across services we are currently considering. Because most of a senselet's time is spent within the sensor feed processing APIs, using this simple mechanism to optimize these APIs will reduce computation and storage requirements significantly. In this section, we describe our design for supporting shared filtering across multiple senselets.

4.1 Cross-Senselet Sharing

We abstract the sharing among senselets by using the concept of a *computation DAG*. Consider the two senselets whose data flow graphs (that show the sequence of sensor feed updates, computations and intermediate results) are shown in Figure 3. The figure shows the complete computation DAG for the video frame at time 12, and portions of the DAGs for the previous frames. 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 formally, 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 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 3 shows such a computation DAG where two scripts are processing the same sensor data with timestamp 12.

We wish to enable senselets like the pair shown in Figure 3 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 the 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.

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 subsections.

4.2 Shared Buffering of Intermediate Results

In IRISNET, intermediate results generated by the senselets are kept in shared memory regions so that all senselets can use them. 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 later re-use, without repetition of the same computation.

The SA allocates each newly started senselet a shared memory region. 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. This allocation strategy prevents one senselet from overwriting intermediate results generated by other senselets. Figure 2 shows that the Parking Finder senselet 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 shared memory allocation calls (based on [5]) that allocate memory from the calling senselet's own shared memory pool. Note that intermediate results are not self-contained – they may often contain pointers to other objects which may, in turn, contain additional pointers. Fortunately, pointers within the 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 discussed in the next section. Each intermediate result is marked with the timestamp of the original sensor feed (e.g., video frame) from which it is generated.

IRISNET evicts intermediate results from shared memory when it is full. The replacement policy for shared memory is to evict the item with the oldest timestamp. If multiple such results exist 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.

4.3 TStore: Indexing Intermediate Results

Senselets use an index in order to publish 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. 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 memory 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 finds tuples with the specified tuple-name and time-spec (timestamp and slack) in the TStore. The slack in the time-spec allows a senselet to take advantage of its tolerance for accepting any of number of close together sensor readings. `Lookup` returns a result as long as the appropriate computed value exists for any of the sensor readings in timestamp $\pm$ slack.

The names of intermediate results (*i.e.*, the `name` fields of tuples) must be consistent across senselets, uniquely describe results, and be easily computable. Note that 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 3 is the hash of the function name `Find Contour`, concatenated with the name of the tuple marked (a), concatenated with the names of other operands to `Find Contour`. 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 kept sorted in decreasing order of their timestamps which improves the performance of `Lookup` and `Insert` operations. Tuples

are evicted from TStore when the corresponding intermediate results are evicted from the 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 the shared memory.

4.4 APIs to Enable Sharing

The sharing of the intermediate results through TStore are completely hidden from the senselet authors. The sharing is automatically enabled if the authors use the sensor feed processing library provided by IRISNET. The APIs of this library are built from the APIs of widely used libraries with the addition of a simple wrapper that enables sharing. The wrapper 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 using `Insert`.

Because the wrapped APIs have very similar interfaces as the original APIs, it is relatively easy for a senselet author to modify his code to use the IRISNET-provided library and enable sharing. However, this sharing is optional, and a senselet can always compute its own results by passing the value of -1 for the `time-spec` parameter, that forces the APIs to compute a result from the original sensor data, even if the same result is available in the TStore.

5 Privacy of Distributed Filtering

Providing easy access to live video and other sensor feeds raises a number of obvious privacy concerns. Ensuring, with full generality, that a sensor feed *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.

Towards this goal, we divide the senselets into two classes *trusted* and *untrusted*. Senselet authors cryptographically sign senselets, and SAs classify them into one of these two categories according to the (verified) identity of the author. While trusted senselets are given access to the raw sensor feeds, untrusted senselets can only access sensor feeds that have been pre-processed. This pre-processing attempts to remove any data that affects the privacy of an individual. For example, we have implemented a face detector to identify human faces in an image, and replaces them with black rectangles. Identifying people in such anonymized images is significantly more difficult. Finally, 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.

One challenge in this design is that if such a privacy filter and untrusted senselets are free-running, the resulting naive CPU allocation may be inefficient. For example, if when sharing the CPU equally, the privacy filter produces 10 frames per second of video and an untrusted senselet processes 5 frames per second of video, the privacy filter wastes half the CPU it consumes. These cycles might instead have been used by the untrusted senselets to increase their output frame rates. However, carefully coordinating the demands of the different senselets can be difficult. For example, supporting two senselets each requiring 5 frames per second may result in the privacy filter generating anywhere between 5 to 10 frames per second depending on how cleverly the demands are processed, as described next.

To support scheduling that maximizes the output frame rate of the untrusted senselets (the "useful work" done by the system), and eliminates wasted work by a privacy filter, IRISNET incorporates flow-control between the privacy filter and the untrusted senselets. 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 more recent than that timestamp. If no such frame is available, the senselet tries to retrieve the newest unused frame that is more recent than the timestamp. However, if no frames are more recent than that timestamp, the untrusted senselet sets the used bit on the newest frame and 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.

6 Experimental Results

We present a performance evaluation of the IRISNET's SA architecture that seeks to answer the following three questions: 1) What are the performance gains in intelligently filtering at the SAs *vs.* performing the work at the OAs, (Section 6.2)? 2) What is the cost or gain of cross-senselet sharing (Section 6.3)?, and 3) What are the overheads of providing privacy through a privacy filter (Section 6.4)?

6.1 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. Unless otherwise specified, we use the Parking Space Finder (PSF) service described in Section 1. Each SA samples the webcam feed 10 times per second, to support services that require up to that frame rate, and writes 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 senselet we examine reads one frame per second.

6.2 Processing Webcam Feeds

In our first set of experiments, we show the effectiveness of filtering sensor feeds in the SAs. We compare two scenarios. In the first scenario, filtering is done in the SAs with senselets. In the second scenario, filtering is done in the OAs — SAs send compressed video frames to the OAs, which then decode the frames, process them with the senselet code, and update their local databases. We use the *FAME* [1] library for encoding the video frames into MPEG-4 in the SAs, and the *SMPEG* [6] library for decoding the frames in the OAs. We assume that the SAs are in the same local area network as the OAs and the OA database is updated once per second.

As mentioned in Section 2.2, filtering in the SAs significantly reduces the required bandwidth between SAs and OAs. In our experimental setup, even a compressed video frame took tens of kilobytes, whereas senslets encoded the required information in several bytes. This not so surprising result shows one major advantage of our distributed filtering architecture.

Figure 4 shows the breakdown of the time spent on the various stages of extracting information from a video frame and updating the database under the scenarios of filtering in SAs and OAs, respectively. Here, we measure the execution time required to run one senselet on the SA (`SA(1)`), 8 senselets on the SA (`SA(8)`, the scenario is described in the next section), and one senselet on the OA. Not only does filtering in the SAs save network bandwidth; it also parallelizes sensor feed processing across SAs, rather than concentrating processing in the OAs. The graph shows that an OA takes the same amount of time to process a video frame as an SA, but intuitively, aggregation of feeds from many SAs in an OA can easily overwhelm the computational capability of even the fastest processor. This poor scaling is exacerbated when multiple OAs run on the same physical machine. The graph also reveals that while filtering in the SAs incurs a high load on SA hosts, even a moderate sharing across the senselets reduces the per-service computational load significantly. For example, the second bar in the graph shows that enabling result sharing across 8 concurrently running senselets significantly reduces the per-service costs.

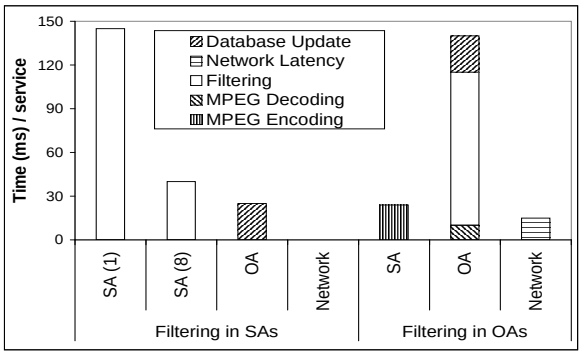


Figure 4. Breakdown of the time spent extracting information from video frames and updating the database for SA vs. OA filtering.

These results suggest that filtering in the SAs is far more scalable than filtering in the OAs, particularly when cross-senselet sharing is used, because the network overhead is significantly reduced and the computational overhead is distributed among the SAs (which are expected to outnumber the OAs, as multiple SAs may report to the same OA).

6.3 Effectiveness of Sharing Among Services

In this section, we evaluate the overhead introduced by wrapping the OpenCV image processing APIs in TStore calls, and the performance gains we achieve from sharing across senselets.

6.3.1 Experiment Parameters

Our evaluation of sharing has three critical parameters: workload, shared memory size and execution slack. We describe each below.

Workload. In order to evaluate the effectiveness of sharing, we create an SA workload based on four different image processing senselets we have developed. For example, the PSF service uses the senselet PSF2 below. 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 Locator (PL): Current frame → Reduce noise → Gray → Find Contour → Get image parts → Subtract background → · · ·

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

Figure 5. Average time required by different operations.

We average all measurements in this section over twenty 30-minute executions. We report the results of four sets of experiments below. 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} + {PL, 1 sec}
[E3] 6 senselets: E2 + {PSF1, 2 secs} + {MD, 2 secs}
[E4] 8 senselets: E3 + {PSF2, 2 secs} + {PL, 2 secs}

Shared Memory Size. 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 in each run — but is varied between experiments) 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. Unless otherwise specified, we use $k = 2$ which means that the allocated shared memory has a size half the maximum total required by all the concurrent senselets.

Slack. As mentioned before, the senselets have soft real time behavior; they process data periodically, and the deadlines have small slack periods. This slack in execution time is the same slack that is used in retrieving results from the TStore. In this evaluation, slack is defined as a percentage of a senselet's execution interval. We vary this slack between experiments.

6.3.2 Overhead of Wrapping APIs

Figure 5 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 function 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 function in most of the cases. As we show later in this section, we make significant gains for this small cost.

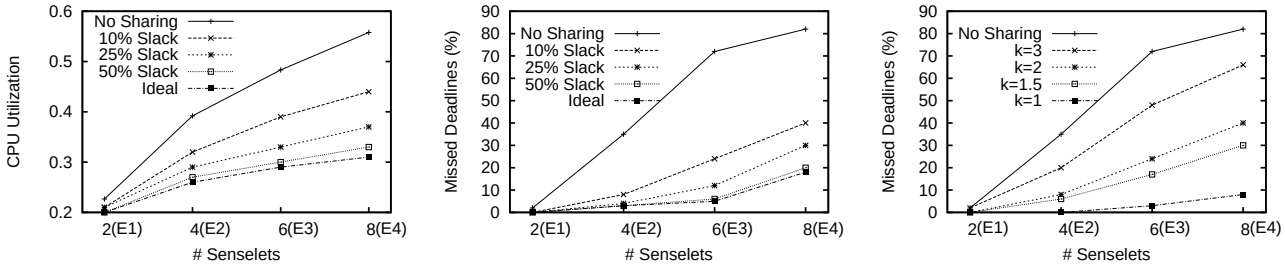
6.3.3 The Effect of Sharing on CPU Load

Figures 6(a) shows 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. In addition, the ideal case assumes that every senselet is scheduled to execute exactly periodically (*i.e.*, it ignores CPU scheduling conflicts). 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 6(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. The effectiveness of sharing can be further increased by increasing the size of the shared memory. We omit the supporting results here for lack of space (see [21] for details).

6.3.4 The Effect of Sharing on Missed Deadlines

As described in Section 3.2, 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 and senselets miss more deadlines. Figures 6(b) and 6(c) show how the number of missed dead-



(a) Slack and CPU time ($k = 2$) (b) Slack and missed deadlines ($k = 2$) (c) Missed deadlines (slack = 10%)

Figure 6. Plots showing the effect of sharing.

lines 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 6(b)) and larger shared memories (Figure 6(c)).

6.4 Overhead of Privacy Protection

To evaluate the potential overhead of a privacy-protecting filter, we construct a filter using the OpenCV face detector. The filter detects all human faces in a video frame and replaces them with a black rectangle. We measure the effects on three different untrusted senselets, each requiring different amounts of processing time per frame. We modify the PSF2 senselet into three different senselets (Parking 1, Parking 2, and Parking 3) that differ in the frequency of camera calibration, a desirable functionality when the cameras may move (by wind, for example). Camera calibration uses a few predefined landmarks to infer the positions of the parking spots in the video frames. We deliberately choose this compute-intensive function and disable sharing in order to illustrate the effects of flow control.

The first bar of each group in Figure 7 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. 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 senselets Parking 1, Parking 2 and Parking 3, respectively. As expected, the CPU time given up by the face filter is evenly distributed among the senselets.

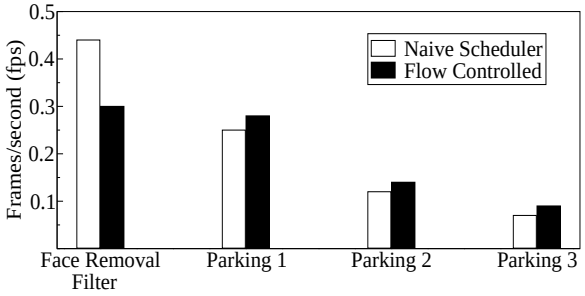


Figure 7. Effect of flow control on senselet processing rates.

7 Related Work

In this section, we explore related efforts in the following areas: video surveillance, active networks, and sensor networks. 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.

Video Surveillance. The use of video sensors has been explored by efforts such as the Video Surveillance and Monitoring (VSAM) [9] 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.

Active Networks. The Active Network architecture [24] 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 [15] and low-power network protocols [16]. Mote-based systems have relied on techniques such as directed diffusion [14] to direct sensor readings to interested parties or long-running queries [8] 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 [17, 18, 19], 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. Since the initial IRISNET paper [10], other work has begun to consider more powerful sensor networks (*e.g.*, [2, 22]), addressing aspects other than distributed filtering.

8 Conclusion

Distributed filtering is the key to creating sensing services that can scale to employ a large number of high bit-rate sensors such as webcams. In this paper, we have described several techniques that address the challenges of efficiently supporting this filtering near the sensors. In the context of IRISNET, we have presented the APIs required to perform distributed filtering, techniques required to scale the infrastructure to a large number of concurrent sensor-enriched services, and mechanisms to address the privacy issues raised by untrusted services. The deployment of a number of real world services on IRISNET indicates that our solutions place few restrictions on the type of services that IRISNET can support. Finally, we have shown the significant benefits of our design through experiments with our IRISNET implementation.

References

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

[2] Hourglass: A data collection network for scalable sensor applications. http://www.eecs.harvard.edu/ syrah/hourglass.

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

[4] IrisLog: a distributed systelog. http://www.intel-iris.net/irislog.

[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] XML path language (XPATH). http://www.w3.org/TR/xpath.

[8] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards sensor database systems. In *Mobile Data Management*, 2001.

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

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

[11] A. Elgammal, R. Duraiswami, D. Harwood, and L. S. Davis. Background and foreground modeling using nonparametric kernel density estimation for visual surveillance. 90(7):1151–1163, July 2002.

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

[13] M. Hebert. http://www.cs.cmu.edu/~hebert/, personal communication, November, 2002.

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

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

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

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

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

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

[20] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. Irisnet: An architecture for enabling sensor-enriched internet service. Intel Research Pittsburgh Technical Report IRP-TR-03-04, 2003.

[21] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. A distributed filtering architecture for multimedia sensors. Intel Research Pittsburgh Tech. Report IRP-TR-04-16, 2004.

[22] L. E. Parker, B. Kannan, X. Fu, and Y. Tang. Heterogeneous mobile sensor net deployment using robot herding and line-of-sight formations. In *IROS*, 2003.

[23] T. Roscoe, L. Peterson, S. Karlin, and M. Wawrzoniak. A simple common sensor interface for planetlab. PlanetLab Design Notes PDN-03-010.

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