A Distributed Filtering Architecture for Multimedia Sensors

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

*Carnegie Mellon University †Intel Research Pittsburgh

Also appears in First Workshop on Broadband Advanced Sensor Networks (BaseNets), 2004

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 the trusted and untrusted services, and provides mechanisms to hide sensitive sensor data from the 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. Our evaluation shows that our design can: 1) reduce the bandwidth demands of many applications to a few hundred bytes per second on each sensor, 2) support a large number of services on each sensor through the use of redundant computation elimination, and 3) address privacy/security concerns with little additional overhead.

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 that overlook parking lots to determine the availability of parking spaces. 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 and 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 [11, 13, 23].

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 application processes its desired sensor feeds on the CPU of the sensor nodes 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 postprocessed data. For example, the Parking Space Finder service would have cameras overlooking parking lots process a video feed to generate a small bit-vector of parking space availability information. 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 of results *between* sensing services running on the same node. Distinct sensing tasks name the results of their computations, and use these names to retrieve the results computed by other sensing tasks. Our results show that this approach makes computation demand scale sub-linearly with number of applications (e.g., eight simultaneous typical applications sharing a sensor node result in only twice the computation load of running one such application).

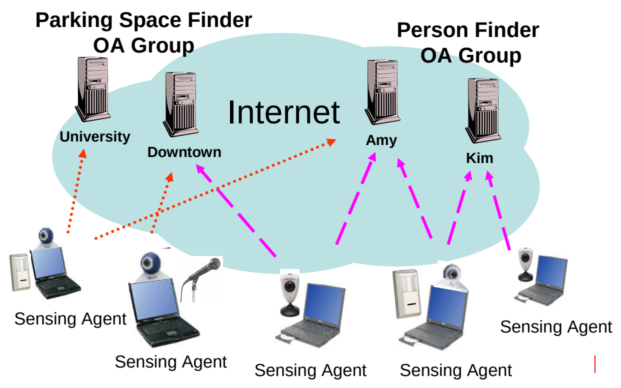


Figure 1: IRISNET Architecture

Finally, while this basic design solves the scalability challenges of sensor data collection, it fails to address the associated security and privacy concerns. For example, the sharing of results between sensor filters allows malicious services to feed incorrect data to other services and the creation of an easily accessible shared sensor infrastructure raises a number of privacy concerns. To mitigate these problems, IRISNET supports different classes (trusted and untrusted) of services that have different privileges for running code and accessing sensor data on the each node. In addition, IRISNET also supports probabilistic checking of shared results to identify malicious services. These techniques place little additional overhead on the system.

The rest of the paper is organized as follows. Section 2 briefly describes the architecture of IRISNET followed by a description of a number of real services built on it (Section 3). Section 4 provides a description of the environment of distributed filtering in IRISNET. We describe how IRISNET addresses scalability and privacy challenges in Section 5 and Section 6 respectively. Section 7 presents the evaluation of our design and implementation. We describe related work in Section 8 and conclude in Section 9.

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[1]), 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.

[1]We use the terms ”SA” and ”SA daemon” interchangeably

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 (PSF) 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 [8], a standard XML query language, to query the sensor database. 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 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 [11, 13, 23].

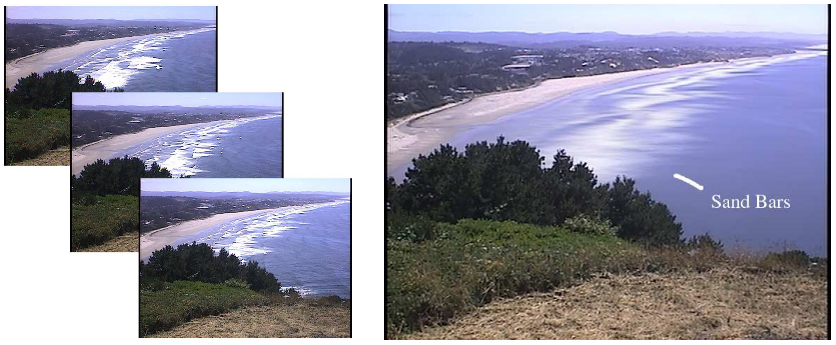
2.2 SA Architecture

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. 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.

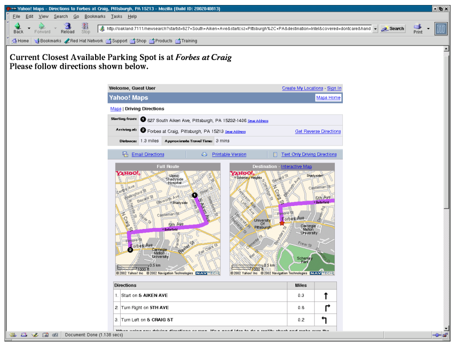
One key challenge is that transferring large volumes of data to the OAs can easily exhaust the resources of the network. IRISNET relies on sophisticated processing and filtering of the sensor feed at the SA to reduce the bandwidth requirements. To greatly enhance the opportunities for bandwidth reduction, this processing is done in a *service-specific* fashion. IRISNET allows service authors to upload programs, called *senselets*, that 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.

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 which can be instructed to go closer to some 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 support untrusted, buggy, or malicious senselets? We address these three questions in Section 4, Section 5, and Section 6 respectively.



(a) Raw video feed input (left) and time-averaged output (right) of the Coastal Imaging service


Yahoo! Maps - Directions to Forbes at Craig, Pittsburgh, PA 15213 - Mozilla (Build ID: 2002040813)
File Edit View Search Go Bookmarks Tasks Help
Back Forward Reload Stop
Search
Print
Home Bookmarks Red Hat Network Support Shop Products Training
Welcome, Guest User
Create My Locations - Sign In
Yahoo! Maps
Maps Home
Maps | Driving Directions
Email Directions
Printable Version
Text Only Driving Directions
Full Route
Destination - Interactive Map
Document: Done (1.138 secs)

http://oakland:7111/newsearch?startst=627+South+Aiken+Ave&startcsz=Pittsburgh%2C+PA&destination=Intel&covered=dontcare&hand

Current Closest Available Parking Spot is at *Forbes at Craig*
Please follow directions shown below.

Starting from: 627 South Aiken Ave, Pittsburgh, PA 15232-1406 Save Address
Arriving at: Forbes at Craig, Pittsburgh, PA 15213 Save Address — Get Reverse Directions
Distance: 1.3 miles — Approximate Travel Time: 3 mins

	Directions	Miles	
1.	Start on **S AIKEN AVE**	0.3	↑
2.	Turn Right on **5TH AVE**	0.8	↱
3.	Turn Left on **S CRAIG ST**	0.2	↰

(b) Output of the PSF service, showing direction to an empty parking spot

Figure 2: Outputs of two services built on IrisNet

2.3 Authoring a Service in IrisNet

To author a sensor-enriched service on IRISNET, a service author needs to first create the sensor database schema that defines the attributes, tags and hierarchies used to describe and organize sensor readings. He then writes senselet code for the SAs having 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, he provides a user interface for end users to access the service. 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-based services, such as the data collection, query processing, indexing, networking, caching, load balancing, and resource sharing.

3 Example Services

A number of IRISNET-based services are being developed and deployed [22]. This section describes two such services and the distributed filtering they perform at the SAs.

3.1 Coastal Imaging Service

In collaboration with oceanographers of the Argus project [1] at Oregon State University, we have developed a coastal imaging service on IRISNET. The service uses cameras installed at sites along the Oregon coastline. These SAs communicate with the OAs through wireless network. The senselets running on the SAs process the video to identify the visible signatures of the nearshore phenomena such as riptides and sandbar formations. One senselet produces 10-minute time-averaged exposure images (Figure 2(a)) that show wave dissipation patterns (indicating submerged sandbars), variance images, and photogrammetric metadata. Senselets for other experiments often perform similar processing with slightly different run-time parameters. Users can also change the data collection and filtering parameters remotely and install triggers to change the data acquisition rates after certain events (for example, to increase sampling frequency when interesting phenomena are detected).

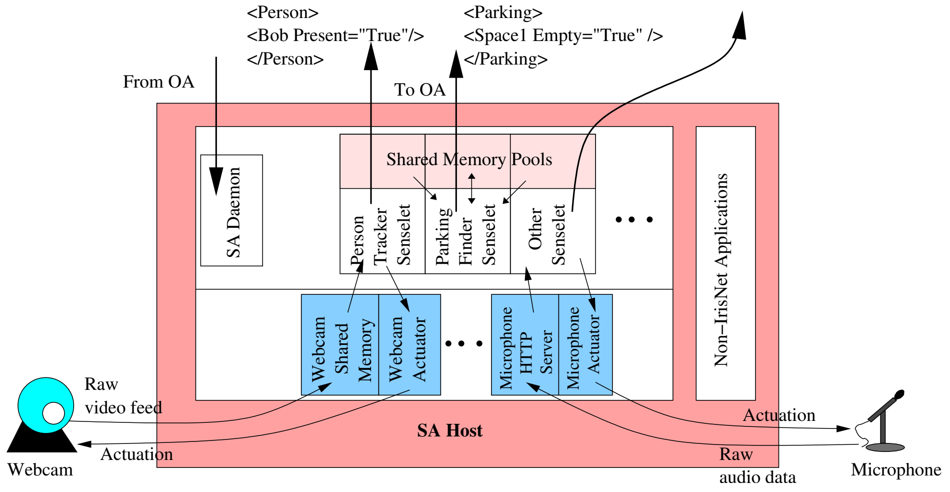


Figure 3: Execution Environment in SA

3.2 Parking-space Finder

The Parking-Space Finder (PSF) service uses cameras throughout a metropolitan area to track parking space availability. Users fill out a Web form to specify a destination and any constraints on a desired parking space (e.g., must be covered, does not require permit, etc.). Based on the input criteria, the PSF service identifies the available parking space nearest to the destination and uses the Yahoo! Maps service to find driving directions to that parking space from the user's current location (Figure 2(b)).

We have implemented multiple versions of the PSF senselets. Each of these uses different image processing algorithms to recognize if a parking spot is empty[2]. One version of the senselet is configured with the locations and background images of the spots. It determines whether a space is full or not by subtracting the current image from the background and comparing it with a predefined threshold. Another version of the PSF senselet uses variance of the color of each pixel with its neighboring pixels to detect the edges of the cars.

4 The SA Execution Environment

This section describes the basic execution environment of an SA in IRISNET (Figure 4).

4.1 Controlling Senselets

Each IRISNET service processes the sensor data at the SAs with application specific code called senselets. Each senselet runs as separate process, collects sensors data from the SAs, filters them, and finally sends the filtered information

[2]Our current algorithms are very simple. We could use more sophisticated image processing algorithms. A possible algorithm [12] 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. However, our simple image processing code is sufficient to demonstrate the important features of the IRISNET infrastructure.

Load(*senselet-code, senselet-id, SA-name*)
: Uploads the specified senselet code to an SA. The senselet can later be referenced by using *senselet-id*.

StartSenselet(*senseied-id, SA-name*)
: Starts the senselet at the specified SA. It assumes that the senselet is already uploaded to the SA.

ControlSenselet(*control-message, senseied-id, SA-name*)
: Sends a control message (*e.g.,* stop, change sampling-rate) to the specified senselet running on the specified SA. It returns the concatenated responses.

Figure 4: Senselet Manipulation APIs.

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 APIs (Figure 4) by which service authors can interact with SAs to install and control senselets. In the simplest case, service authors can use these APIs to interact with a single specified SA. However, in a deployment with thousands of SAs, it might be more useful to interact with a large subset of the SAs with a single command. For example, a service author might want to upload a new senselet to all the SAs in Pittsburgh. This task requires a compact way to select a set of SAs before interacting with them. IRISNET supports this by interfacing the senselet manipulation APIs with its query processing feature. A service author can call the intended senselet manipulation API with a selection query that selects the intended subset of SAs. This simplifies the task of managing a widely deployed service.

In addition to providing access to sensor and actuators via senselets, an SA also provides access via HTTP. Each sensor/actuator is named with a uniform resource identifier (URI). HTTP GET requests (with optional parameters) for this URI provides applications with a simpler interface for configuring data collection and retrieving sensor feeds. However, this interface provides little filtering (other than controlling the sampling rate for sensors) of the sensor data. The input/output syntax for this interface follows the specification given in [25] for the software sensors provided by the PlanetLab [6] infrastructure.

4.2 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. The SA places all readings from a sensor into a shared memory segment associated with that sensor. Senselets gain access to the sensor feed by mapping the shared memory segment into its address space. 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., timestamp), 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.

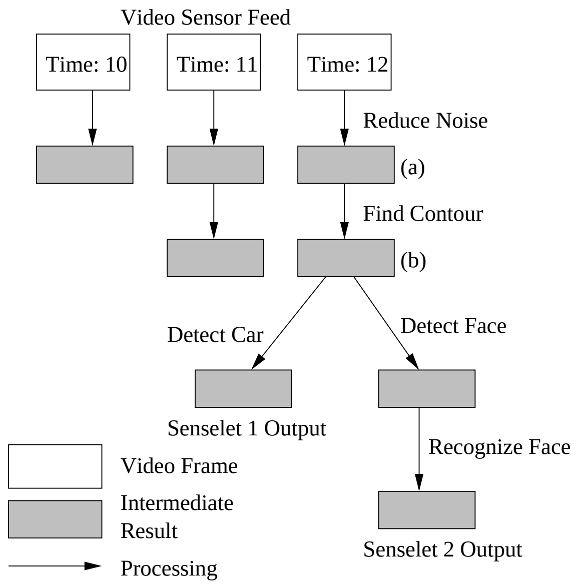
Filtering Libraries The IRISNET also provides sensor feed processing libraries with well-known APIs to be used by the senselets. For example, senselets can perform the image processing task on video data by using an IRISNET customized version of the OpenCV library [3]. We expect typical senselets to 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 in a way 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).

5 Scalable Filtering

So far we have discussed how IRISNET uses senselets to perform distributed filtering to reduce the network overhead of sensor data collection. 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 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 [14]. 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 [12]. Similarly, we have seen that experiments in the Coastal Imaging service have much similarity in their computations. These examples suggest 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.



(a) Computation DAGs for two senselets.

```
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;
}
```

(b) Wrapped version of an OpenCV API.

Figure 5: Cross senselet sharing between PSF and Person Locator. In (a), 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. In (b), the OpenCV API `cvsAbsDiff()` is wrapped to enable cross senselet sharing.

5.1 Cross-Senselet Sharing

Multiple senselets in an SA run continuously on the same sensor feed, such that there may exist many common subtasks across the senselets. For example, consider the two senselets whose data flow graphs (that show the sequence of sensor feed updates, computations and intermediate results) are shown in Figure 5(a). 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 5(a) 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 5(a) 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.

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.

5.2 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 re-use later, without repetition of the same computation.

The SA daemon, which spawns senselets, allocates each new 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 3 shows that the Parking Space 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 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 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 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.

5.3 TStore: Indexing Intermediate Results

To make use of the shared memory store, senselets need to maintain an index in order 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[3]. 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,`

[3]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.

`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. 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(a) 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. 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.

5.4 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 with `Insert`.

We show, in Figure 5(b), how the `cvAbsDiff` API (that finds the pixel-by-pixel difference of two images) of the OpenCV [3] library has been modified. The wrapped version of the API, named `saAvsDiff` has a similar interface as `cvAbsDiff`, except with the additional `TimeSpec` parameter that defines the desired timestamp and slack in the sensor data. For example, if the `TimeSpec` parameter specifies that the timestamp is now and slack is 100 ms, then a previously computed result is returned only if it has been computed on data within the last 100 ms, otherwise the result is computed anew. 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 so that other senselets can reuse them.

Since the wrapped APIs have very similar interface 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. 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.

6 Privacy and Security of Distributed Filtering

Running senselets at sensor hosts is fraught with safety concerns. Buggy or malicious senselets may consume excessive resources on a sensor host or may even exploit security vulnerabilities in the sensor host OS. Currently, IRISNET uses two simple mechanisms for safe execution of senselet code. First, all the senselets are run inside a single User-Mode Linux (UML) virtual machine [17]; this ensures that bugs in the senselets can not compromise the SA host. Second, each senselet is run as a separate process; this ensures process level security among the senselets. Since the code is relatively compute intensive (rather than system call or I/O intensive), this virtual machine (VM) approach imposes a modest 13% reduction in our test senselet's video processing rate.

We should note that IRISNET's VM approach is neither the most fool-proof or efficient solution. However, such safety concerns are far from new and IRISNET can easily take advantage of existing efficient techniques for safe execution of such code [4, 24, 26]. Our efforts focus more on new types of safety concerns that the IRISNET architecture creates rather than on ones that traditional techniques can solve. In this section, we describe our approach to two such issues: 1) the sharing of results between senselets allows malicious senselets to feed incorrect data to other applications, and 2) the creation of an easily accessible shared sensor infrastructure raises a number of privacy concerns.

6.1 Sharing in the Presence of Malicious Senselets

A malicious senselet can compute and share incorrect intermediate results to feed false sensor readings to other applications. For example, a malicious senselet processing the image of an empty parking spot can produce incorrect intermediate results that, when used by the PSF senselet, can make it infer that the parking spot is full.

Although IRISNET currently assumes that the senselets are not malicious, it supports mechanisms to deal with this problem. Passing the value of -1 for the `time-spec` parameter of an function supporting sharing forces IRISNET to compute a result from the original sensor data, even if the same result is available in the TStore. Thus, a senselet, S_1, can occasionally compute the intermediate results itself and compare them with those available in the TStore. If the results disagree, S_1 adds the producer of the TStore value into its black-list and avoids sharing intermediate results computed by that producer.

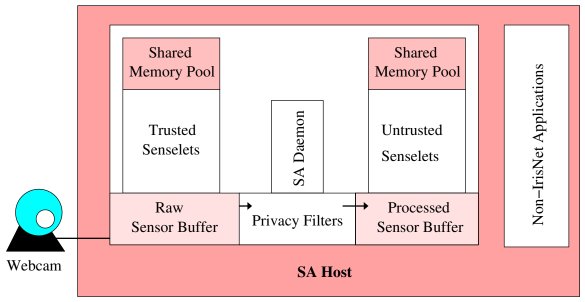


Figure 6: Privacy-Protecting SA

6.2 Protecting Privacy

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 *privacy-protecting image processing* for IRISNET, in which we use image processing techniques to anonymize a video stream. Our prototype uses a face detector algorithm to identify the boundaries of human faces in an image, and replaces these regions with black rectangles. Identifying people in the anonymized image is significantly more difficult. Figure 6 shows the resulting architecture of the IRISNET SA augmented to support privacy-protecting image processing. Also, note that there are separate shared memories for the two senselet classes; intermediate result sharing is done as before, but only among senselets in the *same* class.

One challenge in this design is that if the 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 process 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.

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 the privacy filter, IRISNET incorporates flow-control between the privacy filter and 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

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

the timestamp. However, if no frames are more recent than the 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.

7 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 7.2)? 2) What is the cost or gain of cross-senselet sharing (Section 7.3)?, and 3) What are the overheads of providing privacy through a privacy filter (Section 7.4)?

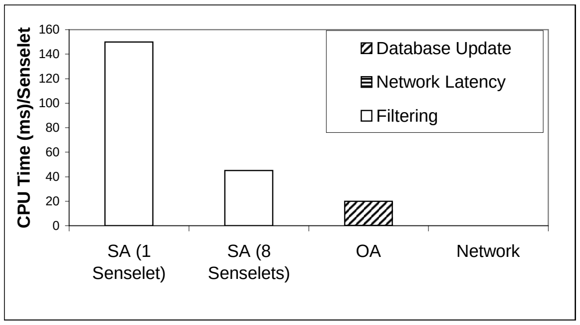
7.1 Experimental Setup

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 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 reads one frame per second. Unless otherwise specified, we use the PSF service described in Section 3.

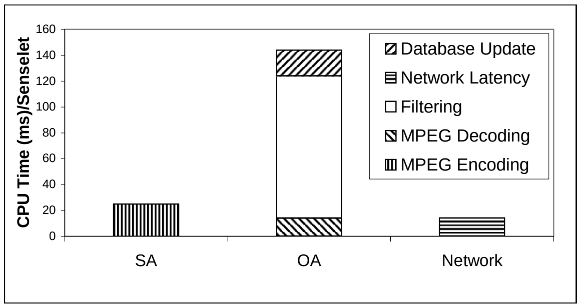
7.2 Processing Webcam Feeds

In our first set of experiments, we show the effectiveness of filtering sensor feeds at 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* [2] library for encoding the video frames into MPEG-4 at the SAs, and the *SMPEG* [7] library for decoding the frames at 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.

Figure 7 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



(a) Filtering in SAs



(b) Filtering in OAs

Figure 8: Breakdown of time spent extracting information from video frames and updating the database.

actual measurements. Although cameras feed a large volume of raw video data to the SAs[4], 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 8 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 8(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 8(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. For example, 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 that filtering at SAs is far more scalable than filtering at OAs. The advantage is two-fold: 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.

7.3 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.

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

7.3.1 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 created a SA workload based on four different image processing senselets we have developed. For example, the PSF service described in Section 3 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 $\rightarrow$ Reduce noise $\rightarrow$ Convert to gray $\rightarrow$ Find contour $\rightarrow$ Compare contours $\rightarrow \cdots$
- Parking Space Finder 2 (PSF2): Current frame $\rightarrow$ Reduce noise $\rightarrow$ Convert to gray $\rightarrow$ Get image parts $\rightarrow$ Subtract background $\rightarrow \cdots$
- Motion Detector (MD): $\{$Current frame $\rightarrow$ Reduce noise $\rightarrow$ Gray, 1 second old frame $\rightarrow$ Reduce noise $\rightarrow$ Gray$\}$ $\rightarrow$ Subtract images $\rightarrow \cdots$
- Person Tracker (PT): Current frame $\rightarrow$ Reduce noise $\rightarrow$ Gray $\rightarrow$ Find Contour $\rightarrow$ Get image parts $\rightarrow$ Subtract background $\rightarrow \cdots$

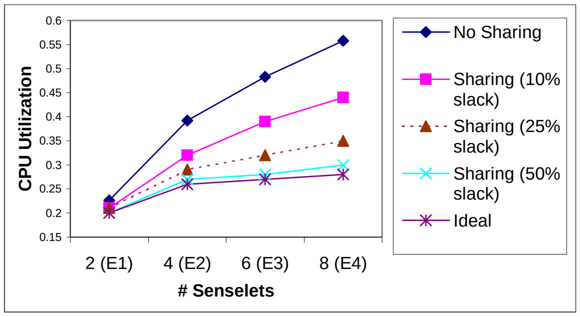
We average all measurements in this section over 20 30-minute executions. We report the results of four sets of experiments. The combinations of senselets in each set, and their deadline intervals in seconds are as follows:

[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$\}$

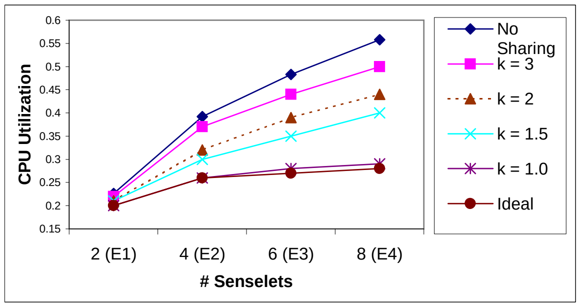
Shared Memory Size The optimal size of shared memory needed to achieve the maximum sharing depends on a senselet's sensor feed access pattern, execution pattern (deadline and slack values), and intermediate result generation rate. For a small shared memory, arrival of a new intermediate result may force 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.

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

Figure 9: Average time required by different operations.



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



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

Figure 10: Plots showing the effect of sharing on CPU time.

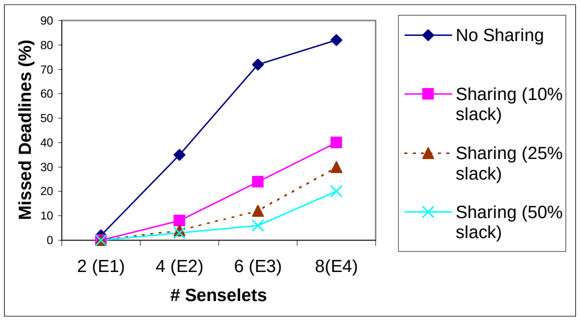
Slack As mentioned before, the senselets have soft real time behavior; they process data periodically, and the deadlines have small slack periods. This slack in execution time is the same slack that is used in retrieving results from the TStore. In this evaluation, slack is defined as a percentage of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

7.3.2 Overhead of Wrapping APIs

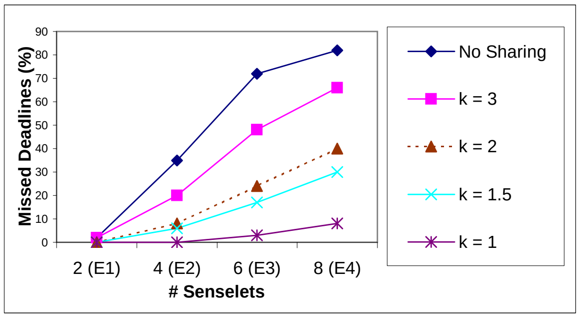
Figure 9 shows the execution times for a few typical functions in the OpenCV API and the overhead of wrapping them. The numbers reported in the figure are the averages of performing the operations on a lightly loaded SA on 20 different 640×480 24-bit images. A typical OpenCV API takes 1-5 ms, whereas 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 show later in this section, we make significant gains for this small cost.

7.3.3 The Effect of Sharing on CPU Load

Figures 10(a) and 10(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. 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



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



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

Figure 11: Plots showing the effect of sharing on missed deadlines.

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 10(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 10(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.

7.3.4 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 and senselets miss more deadlines. Figures 11(a) and 11(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 11(a)) and larger shared memories (Figure 11(b)).

7.4 Overhead of Privacy Protection

To evaluate the potential overhead of the privacy-protecting filter, we have constructed a filter using the OpenCV face detector. The filter detects all human faces in a video frame and replaces them with a gray rectangle. We measured the effects on three different untrusted senselets, each requiring different amounts of processing time per frame. We modified the PSF senselet into three different senselets that differ in the frequency of camera calibration, a desirable

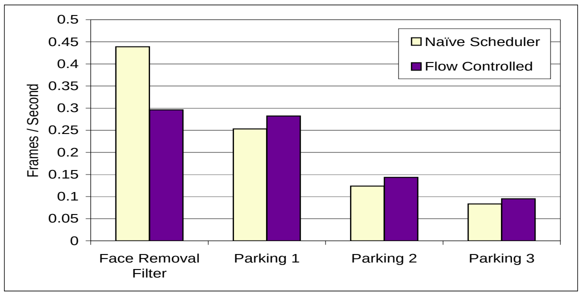


Figure 12: Effect of flow control on senselet processing rates.

functionality when the cameras may move (by wind, for example). Camera calibration uses a few predefined landmarks to infer positions of the parking spots in the video frames. We deliberately choose this compute-intensive function and disabled sharing to illustrate the effects of flow control.

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

8 Related Work

In this section, we explore related efforts in the following areas of work: 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) [10] 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 [26] 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 [16] and low-power network protocols [18]. Mote-based systems have relied on techniques such as directed diffusion [15] 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 [19, 20, 21], 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.

9 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 and security 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 argus program. http://cil-www.oce.orst.edu:8080/.

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

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

[4] Java technology. http://java.sun.com.

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

[6] Planetlab. http://www.planet-lab.net/.

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

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

[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] DESHPANDE, A., NATH, S., GIBBONS, P. B., AND SESHAN, S. Cache-and-query for wide area sensor databases. In *ACM SIGMOD* (2003).

[12] 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.

[13] GIBBONS, P. B., KARP, B., KE, Y., NATH, S., AND SESHAN, S. Irisnet: An architecture for world-wide sensor web. *IEEE Pervasive Computing 2*, 4 (2003).

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

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

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

[17] 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).

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

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

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

[21] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. The design of an acquisitional query processor for sensor networks. In *SIGMOD* (2003).

[22] NATH, S., DESHPANDE, A., KE, Y., GIBBONS, P. B., KARP, B., AND SESHAN, S. Irisnet: An architecture for internet-scale sensing services. Demo and abstract in *Proceedings of Very Large Databases (VLDB)*, 2003.

[23] NATH, S., KE, Y., GIBBONS, P. B., KARP, B., AND SESHAN, S. Irisnet: An architecture for enabling sensor-enriched internet service. Intel Research Pittsburgh Technical Report IRP-TR-03-04, 2003.

[24] NECULA, G. C. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France, jan 1997), pp. 106–119.

[25] ROSCOE, T., PETERSON, L., KARLIN, S., AND WAWRZONIAK, M. A simple common sensor interface for planetlab. PlanetLab Design Notes PDN-03-010.

[26] WETHERALL, D. Active network vision and reality: lessons from a capsule-based system. In *SOSP* (1999), pp. 64–79.