IRIS: Internet-scale Resource-Intensive Sensor Services

Sigmod'03 Demo Proposal

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

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

ABSTRACT

The proliferation and affordability of smart sensors such as webcams, microphones *etc.*, has created opportunities for exciting new classes of distributed services. A key stumbling block to mining these rich information sources is the lack of a common, scalable networked infrastructure for collecting, filtering, and combining the video feeds, extracting the useful information, and enabling distributed queries.

In this demo, we demonstrate the design and an early prototype of such an infrastructure, called *IRIS* (*Internet-scale Resource-Intensive Sensor services*). IRIS is a potentially global network of smart sensor nodes, with webcams or other sensors, and organizing nodes that provide the means to query recent and historical sensor-based data. IRIS exploits the fact that high-volume sensor feeds are typically attached to devices with significant computing power and storage, and running a standard operating system. Aggressive filtering, smart query routing, and semantic caching are used to dramatically reduce network bandwidth utilization and improve query response times, as we will demonstrate.

The service that we demonstrate here is that of a *parking space finder.* This service utilizes *webcams* that monitor parking spaces to answer queries such as the availability of parking spaces near a user's destination.

1. INTRODUCTION

Imagine driving towards a destination in a busy metropolitan area. While stopped at a traffic light, you query your PDA specifying your destination and criteria for desirable parking spaces (e.g., within two blocks of your destination, at least a four hour meter). You get back directions to an available parking space satisfying your criteria. Hours later, you realize that your meter is about to run out. You query your PDA to discover that, historically, meter enforcers are not likely to pass by your car in the next hour. A half hour later, you return to your car and discover that although it has not been ticketed, it has been dented! Querying your PDA, you get back images showing how your car was dented and by whom.

This scenario demonstrates the potential utility of sensor-based services such as a Parking Space Finder, Silent (Accident) Witness and Meter Enforcement Tracker. While several research projects [5, 4, 2, 3, 8, 9] have begun to explore using and querying networked collections of sensors, these systems have targetted the use of closely co-located resource-constrained sensor "motes" [6, 7]. In this demo, we demonstrate an early prototype of a sensor network system architecture, called *IRIS* (*Internet-scale Resource-Intensive Sensor services*)[1] based on much more intelligent participants. We envision an environment where different nodes on the Internet (standard PCs, laptops and PDAs) have attached sensors such as webcams (video cameras attached to Web-enabled devices). Any sensor-based service can retrieve information from this collection of sensors and provide service to users.

While webcams are inexpensive and easy to deploy across a wide area, realizing useful services requires addressing a number of challenges:

- Preventing the transfer of large data feeds across the network is necessary for system scalability.
- Efficiently discovering relevant data among the distributed collection of sensor and service nodes and delivering it to interested participants is crucial for reasonable response times.
- Efficiently handling static meta-data information (e.g., parking meter details and map directions), live readings from multiple sensor feeds, readings from the recent past, and long term historical data is required in order to answer queries like those in our example scenario.

Our goal in IRIS is to create a common, scalable software infrastructure that allows services to address these challenges in a manageable fashion. This would enable rapid development and deployment of distributed services over a worldwide network of sensor feeds.

IRIS is composed of a potentially global collection of Sensing Agents (SAs) and Organizing Agents (OAs). SAs collect and process data from their attached webcams or other sensors, while OAs provide facilities for querying recent and historical sensor data. Any Internet connected, PC-class device can play the role of an OA. Less capable PDA-class devices can act as SAs. Continued advances in microprocessing technology enable significant processing power and memory to be encapsulated in smaller and cheaper devices that may act as SAs.[2] Key features of IRIS include:

- IRIS provides simple APIs for orchestrating the SAs and OAs to collect, collaboratively process and archive sensor data while minimizing network data transfers.

[1]Note to the reviewers: Our project is not to be confused with a new project, also called IRIS, on building networked systems using distributed hash tables.

[2]Note that SAs (and OAs) need not have keyboards or displays.

- The user is presented with a logical view of the data as a single XML document, while physically the data is fragmented across any number of host nodes (*data transparency*).
- IRIS supports a large portion of XPATH, a standard XML query language, for querying the data in the system.
- IRIS handles issues of service discovery, query routing, semantic caching of responses and load balancing in a scalable manner for all services.

We believe that IRIS can enable a wealth of new sensor-based services. Example future uses include providing live virtual tours of cities, answering queries about the waiting time at different restaurants, unobtrusive monitoring of your children playing in the neighborhood, witnessing whose dog pooped on your lawn, and determining where an umbrella was left behind.

2. THE IRIS ARCHITECTURE

In this section we describe the overall architecture of IRIS. its query processing features, and its caching and data consistency mechanisms.

Architecture. IRIS is composed of a dynamic collection of SAs and OAs. Nodes in the Internet participate as hosts for SAs and OAs by downloading and running IRIS modules. Sensor-based services are deployed by orchestrating a group of OAs dedicated to the service. These OAs are responsible for collecting and organizing the sensor data in a fashion that allows for a particular class of queries to be answered (e.g., queries about parking spaces). The OAs index, archive, aggregate, mine and cache data from the SAs to build a system-wide distributed database for that service. Having separate OA groups for distinct services enables each service to tailor the database schema, caching policies, data consistency mechanisms, and hierarchical indexing to the particular service. This does not restrict the placement of OAs, because multiple OAs can be hosted on the same node.

In contrast, SAs are shared by all services. An SA collects 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 that produce large volumes of data and require sophisticated processing, such as webcams. SAs with attached webcams include, as part of the IRIS module, Intel's open-source image-processing library [1]. The sensor data is copied into a shared memory segment on the SA, for use by any number of sensor-based services.

OAs upload scripting code to any SA collecting sensor data of interest to the service, basically telling the SA to take its raw sensor feed, perform the specified processing steps, and send the distilled information to the OA. For video feeds, the script consists primarily of calls to the image processing library. Filtering data at the SAs prevents flooding the network with high bandwidth video feeds and is crucial to the scalability of the system. Even compressed video consumes considerable bandwidth, whereas aggressive filtering can reduce 10 minutes of video down to under a kilobyte of data, depending on the service. For example, the Parking Space Finder service distills the video down to a bit vector indicating which spots are empty.

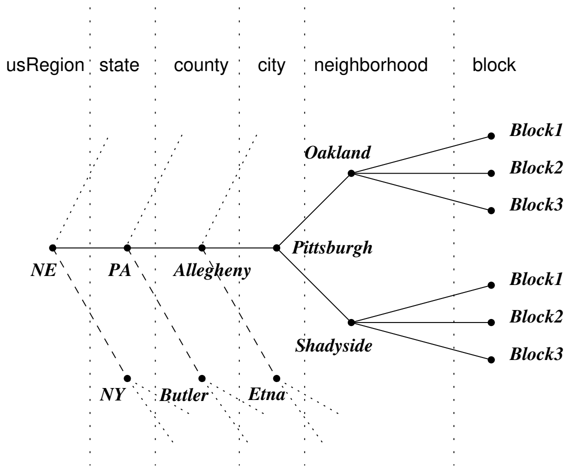

Figure 1: OA Hierarchy

One key concern is that because multiple services utilize each SA, the relatively limited computing power of the SA may be overloaded. Therefore, it is important that SAs are used efficiently and avoid repetitious work. Thus, our programming API for the SA allows downloaded code from different OAs to cooperate and avoid redundant processing and storage. For example, both the Parking Space Finder service and the Silent Accident Witness service perform common steps of filtering for motion detection and for identifying vehicles. Opportunities for work sharing can be detected automatically by the SA, by observing matching sequences of library calls on the incoming stream of frames. We anticipate significant work sharing, because the downloaded codes are all specialized to the processing of the SA's particular sensor feeds.

Query Processing. Central to IRIS is distributed query processing. Data is stored in XML databases associated with each OA. We envision a rich and evolving set of data types, aggregate fields, etc., best captured by self-describing tags – hence XML was a natural choice. Larger objects such as video frames are stored outside the XML databases; this enables inter-service sharing, as well as more efficient image and query processing.

Data for a particular service is organized hierarchically, with each OA owning a part of the hierarchy. An OA may also cache data from one or more of its descendants. A common hierarchy for OAs is geographic, because each sensor feed is fundamentally tied to a particular geographic location.[3] Figure 1, for example, shows a geographical hierarchy for Parking Space Finder.

A user's query, represented in the XPATH language, selects data from a set of nodes in the hierarchy. The query in Figure 2, for example, selects data from the *Oakland Block1* and *Shadyside Block1* nodes in the hierarchy of Figure 1. We exploit the hierarchical nature of the OA organization to expedite the routing of queries to the data, as follows. Observe that each query contains a (maximal) hierarchical prefix, which specifies a single path from the root of the hierarchy to the lowest common ancestor (LCA) of the nodes potentially selected by the query. For the query in Figure 2, this is the prefix of the query up to `city[@id='Pittsburgh']`.

[3]A service may define indices based on non-geographic hierarchies. In IRIS, such hierarchies are reflected in the XML schema.

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

Figure 2: Query asking for all available parking spaces in *Oakland Block1* or *Shadyside Block1*.

We denote this LCA node (in our example, the *Pittsburgh* node) as the *starting point* for the query.

In IRIS, a query from a user anywhere in the world is first routed to its starting point. But how do we find the starting point OA, given the large number of OAs and the dynamic mapping of OAs to host machines? Our solution is to have DNS-style names for OAs that can be constructed from the queries themselves, to create a DNS server hierarchy identical to the OA hierarchy, and to use DNS lookups to determine the IP addresses of the desired OAs. For our example query, we construct the DNS-style name `pittsburgh.allegheny.pa.ne.parking.intel-iris.net`, perform a DNS lookup to get the IP address of the *Pittsburgh* OA, and route the query there.

Upon receiving a query, the starting point OA queries its local database and cache, and evaluates the result. If necessary, it gathers missing data by sending subqueries to its children OAs (the *Oakland* and *Shadyside* OAs in our example), who may recursively query their children, and so on. Finally the answers from the children are combined and the result is sent back to the user. Note that the children IP addresses are found using the same DNS-style approach, with most lookups served by the local host. The key technical challenge overcome in our approach is how to efficiently and correctly detect, for general XPATH queries, what parts of a query answer are missing from the local database, and where to find the missing parts.

XPATH queries supported. In our current prototype, we take the common approach of viewing an XML document as *unordered*, in that we ignore any ordering based solely on the linearization of the hierarchy into a sequential document. For example, although siblings may appear in the document in a particular order, we assume that siblings are unordered, as this matches our data model. Thus we focus on the *unordered* fragment of XPATH, ignoring the few operators such as *position*() or axes like *following-siblings* that are inappropriate for unordered data. We support the entire unordered fragment of XPATH 1.0.

Partial-Match Caching and Data Consistency. An OA may cache query result data from other OAs. Subsequent queries may use this cached data, even if the new query is not an exact match for the original query. For example, the query in Figure 2 may use data for *Oakland* cached at the *Pittsburgh* OA, even though this data is only a partial match for the new query. Similarly, if distinct *Oakland* and *Shadyside* queries result in the data being cached at *Pittsburgh*, the query may use the merged cached data to immediately return an answer.

Due to delays in the network and the use of cached data, answers returned to users will not reflect the absolutely most recent data. Instead, queries specify a consistency criteria indicating a tolerance for stale data (or other types of approximation). For example, when heading towards a destination, it suffices to have a general idea of parking space availability. However, when arriving near the destination, exact spaces are desired. We store timestamps along with the data, so that an XPATH query specifying a tolerance is automatically routed to the data of appropriate freshness. In particular, each query will take advantage of cached data only if the data is sufficiently fresh.

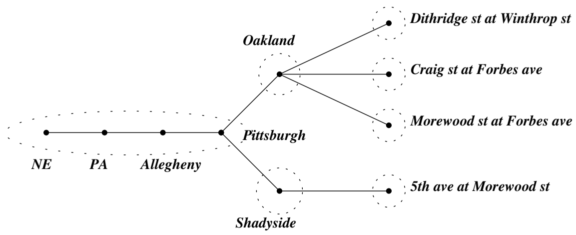

Figure 3: The hierarchy used in the demonstration and the mapping of the hierarchy onto the OAs

Figure 4: Webcams monitoring toy parking lots

3. A PARKING SPACE FINDER SERVICE

The service that we demonstrate in this demo is that of a parking space finder. This service utilizes webcams that are monitoring parking spaces to gather information about the availibility of the parking spaces.

Sensing Agents. We use 4 cameras that are monitoring toy parking spaces set up as part of our demo (Figure 4). These cameras are attached to 1.6 GHz laptop machines that process the video feed, and perform image processing to decide whether a parking spot is full or empty. Figure 5 shows the actual locations of the four parking lots that we simulate using the above setup. These are parking lots near Intel Research Pittsburgh.

Organizing Agents. The organizing agents that we use for this demo are 7 PCs scattered throughout the Intel research labs. Figure 3 shows the part of the hierarchy that is used in this demonstration. This logical hierarchy is mapped onto the 7 machines as follows : (1) the four blocks corresponding to the parking lots are mapped onto one OA each, (2) the two neighborhoods, Oakland and Shadyside, are mapped onto one OA each, and (3) the rest of the nodes in the hierarchy are mapped onto one OA.

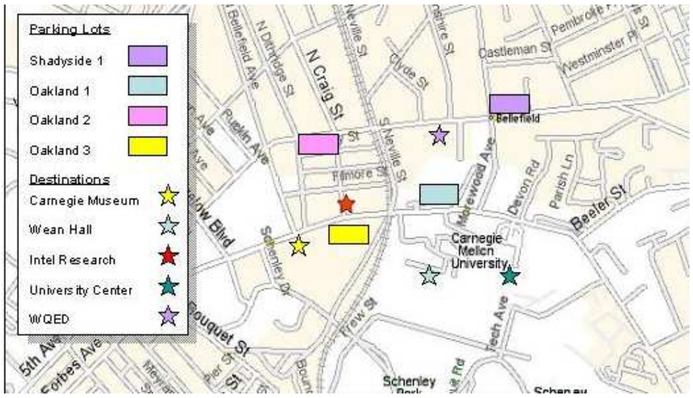

Figure 5: Parking lot locations, and user destinations (Pittsburgh, PA)

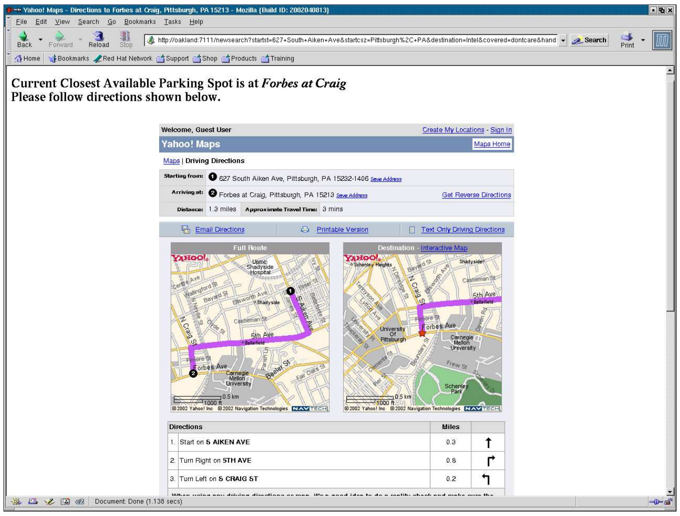

Figure 6: Driving directions to the parking spot are displayed using the Yahoo Maps Service

Web Frontend. The web frontend for this service essentially presents the user with a form that the user can fill out to specify her destination, and also other constraints that she might have (*e.g.*, that the parking spot must be covered). Currently, we only allow the user to pick from 5 destinations near the parking lots (Figure 5) using a drop-down menu. Once the user specifies her criteria and submits the query, the frontend finds the nearest available parking spot that satisfies the user's constraints using IRIS, and then uses Yahoo Maps Service to find driving directions to that parking space from the user's current location. These driving directions are then displayed to the user (Figure 6).

The driving directions are continously updated as the user drives towards the destination, if the availability of the parking spot changes, or if a closer parking spot satisfying the constraint is available. We envision that a car navigation system will repeatadly and periodically ask the query as the user nears the destination. Lacking that, we currently simulate such a behaviour by resubmitting the query periodically by assuming that the user has reached the next intersection along the route (the next intersection is obtained by parsing the driving directions provided by the Yahoo Maps Service).

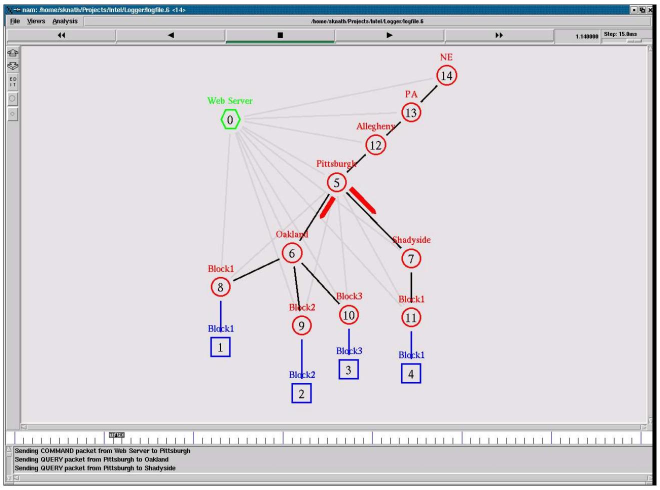

Figure 7: A modified version of NAM is used to show the messages during a query execution

Logging and replaying messages. We also demonstrate a mechanism that we have built for logging and replaying the messages exchanged by the web frontend and by the OAs. The collected log information during execution of a query is used to lazily replay the messages that were sent during the execution of the query. We use the NAM network simulator to show these messages. NAM is part of the popular open-source network simulator, *ns*, with a graphical display that shows the configuration of the network under consideration (Figure 7), and uses animation to show messages being communicated in the network.

A series of XPATH queries of increasing complexity are used to demonstrate visually various aspects of our system such as routing to the starting point, recursive query processing, partial-match caching, and query-based consistency.

4. REFERENCES

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

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

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

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

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

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

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

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

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