Developing a Parking Space Finder service on IrisNet

February 3, 2003

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). In response, you receive 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 retrieve images showing how your car was dented and by whom.

The above scenario demonstrates the potential utility of sensors that allow computer systems to monitor and report real-world events. Such sensors can enable services such as a Parking Space Finder, Silent (Accident) Witness, and Meter Enforcement Tracker, as highlighted above. IRISNET(Internet-scale Resource-Intensive Sensor Network Services), a wide area sensor network infrastructure currently under development at Intel Research Pittsburgh, creates the opportunity for real and easy deployment of these services today.

In this paper, we describe how such a service, the Parking Space Finder (PSF) Service, has been developed in IRISNET. The official website `http://www.intel-iris.net` provides source code (of the IRISNET infrastructure as well as the PSF service), documentation, and papers related to IRISNET. The page also describes how to download and install the code in a computer running Linux. At the end of this paper you should see how easy it is to develop a sensor service in IRISNET.

The rest of the handout is organized as follows. Section 2 gives a brief overview of IRISNET and describes the procedure for deploying a service on it. Section 3 describes the PSF service in details. Finally we conclude in Section 4.

2 IrisNet

2.1 The Architecture

IRISNET is a two-tiered architecture composed of two types of nodes: the *Sensing Agents* (SAs), which collect and filter sensor readings, and the *Organizing Agents* (OAs), which perform query processing tasks on the sensor readings. Nodes in the Internet participate as hosts for SAs and OAs by downloading and running IRISNET modules.

A sensor-based service is 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 a particular 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. However, this does not restrict the placement of OAs, because multiple OAs can be hosted on the same physical machine.

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 IRISNET's 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 IRISNET module, Intel's open-source image-processing library [2] which can be used to process the webcam feed to extract useful information. The sensor data is copied into a shared memory segment on the SA, for use by any number of sensor-based services.

IRISNET relies on sophisticated processing and filtering of the sensor feed at the SA to reduce the bandwidth requirements. To greatly enhance the opportunities for bandwidth reduction, this processing is done in a service-specific fashion. IRISNET enables OAs to upload programs, called *senselets*, to perform this processing to any SAs collecting sensor data of interest to the service. These senselets instruct the SAs to take the raw sensor feed, perform a specified set of processing steps, and send the distilled information to the OAs. Senselets can reduce the required 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.

2.2 Running the OA and SA Modules

As mentioned earlier, nodes in the Internet participate as hosts for SAs and OAs by downloading and running IRISNET modules. The code for SAs and OAs can be downloaded form the webpage `http://www.intel-iris.net/download.html`. The page gives detailed instructions for downloading and installing the code on a Linux machine. Each physical machine can run one instance of the SA and one instance of the OA at a time.

Assuming that the OA module has been installed in full (with all the associated packages), you need to do the following steps to run the OA. In the rest of this handout, we use the variable `$IRISNET` to indicate IRISNET's installation directory,

1. Set the environment variables.

```
export XINDICE_HOME= $IRISNET/packages/xml-xindice-1.0
export JAVA_HOME=$IRISNET/packages/jdk1.3.1_06
export PATH=$PATH:$JAVA_HOME/bin:$XINDICE_HOME/bin
```

2. Run Xindice [1], the local XML database engine.

```
cd $IRISNET/packages/xml-xindice-1.0
./xindice.server start
```

3. Run the OA:

```
cd $IRISNET/OA/IrisOA
make run
```

OA uses the configuration file `$IRISNET/OA/IrisOA/oa.cfg`.

Alternatively, you can run the script `oasetup.sh` form the `$IRISNET` directory, that does exactly the same steps as above.

You need to do the following steps to run the SA.

1. Set the environment variables.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$IRISNET/packages/opencv-0.9.3/
  cv/src/.libs/:$IRISNET/packages/opencv-0.9.3/otherlibs/highgui/.libs/
```

2. Run the webcam software to periodically store the webcam feed to the shared memory. If you are using a real webcam, do the following.

   ```
   cd $IRISNET/SA/IrisWebcam/
   ./runwebcam
   ```

   The tool `runwebcam` continuously reads the current webcam frame and places it in the shared memory segment from where SA reads the video frames. This decoupling of the webcam and the SA creates an opportunity to simulate a fake webcam for development purpose. You can use the tool `$IRISNET/SA/IrisWebcam/loadImage` to load an image into SA's shared memory segment. The image will be in the shared memory until another image is loaded there, and you can do experiment with your image processing code on that image. You may also use the script `$IRISNET/SA/IrisWebcam/loadImages` to periodically and sequentially load the images in the `$IRISNET/SA/IrisWebcam/images` directory to SA's shared memory segment, and thus to simulate a webcam.

   You can use the tool `$IRISNET/SA/IrisWebcam/showImage` to see the current image in the shared memory.

3. Run the SA.

   ```
   cd $IRISNET/SA/IrisSA/src
   ./runsa
   ```

Alternatively, you can run the script `sasetup_rs.sh` (for real sensor) or `sasetup_fs.sh` (for simulated sensor) from the `$IRISNET` directory, which does the steps described above.

2.3 Deploying a Service on IrisNet

Given the IrisNet infrastructure, it is fairly easy to deploy a new service. More specifically, a service developer needs to perform the following steps.

1. The developer needs to locate the desired webcams and other sensors (more specifically, the SAs attached to the sensors), using some service discovery mechanism.

2. The developer writes the senselet for the SAs connected to the desired webcams. The senselets take the sensor feeds, extract the data relevant to the service, and send the data to the appropriate OAs.

3. The developer then creates the sensor database schema in the XML language. The schema defines the attributes and tags used to describe the sensor readings and the hierarchy used to index the data. IRISNET uses the schema to create a single OA that hosts the overall database for the service. IRISNET provides a mechanism to add new OAs to a service and re-partition the database across the nodes as needed.

4. Finally, the developer needs to provide a user interface for the end users to access the service. This user interface takes some simplified user input and generates the appropriate set of database queries to the OAs for the service. IRISNET provides functionality to efficiently answer these queries over the distributed sensor database.

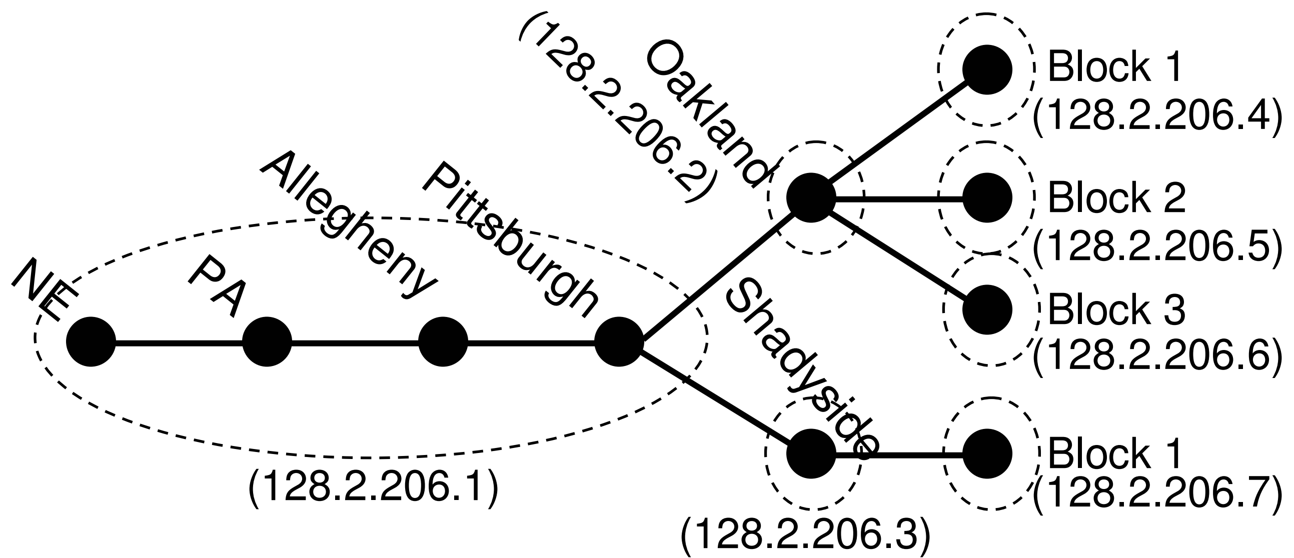

Figure 1: Data hierarchy for the PSF service. The dashed circles define the logical partition of the hierarchy among the OAs with IP addresses shown below each of the partitions.

Step 1 has not yet been implemented in IRISNET. So, for this project we will assume that the service developers (you) know where the desired webcams are. To make the description concrete, Section 3 elaborates how the rest of the steps are performed to deploy the PSF service on IRISNET.

3 The Parking Space Finder (PSF) Service

The objective of the PSF service is to use feeds from cameras installed in parking lots in a metropolitan area and allow users to make queries about the availability of parking spots at a particular location. The senselet for this service processes the webcam feed and recognize whether the parking spots are empty or full. Users specify a destination and constraints (*e.g.,* a covered lot) of their desired spot. The PSF service returns the driving directions (using the Yahoo Maps service) to the empty parking spot that satisfies the constraints and is the nearest to their destination.

We describe the PSF service using the data hierarchy shown in Figure 1, where each of the blocks has one parking lot with a camera (our current prototype uses webcams overlooking toy parking lots with matchbox cars).

Before going to the implementation details of the PSF service, it would be helpful to know the basic interface between OAs and SAs. OAs and SAs communicate with each other through messages, where a message is a string representation of a command and corresponding parameters. The commands and parameters are delimited by a single white space.

OAs talk to SAs by the following three messages.

- `Load`: This message is used to upload the senselet (and other additional files) to the SAs.
- `Subscribe`: This message asks the SA to dynamically load and execute a senselet.
- `UnSubscribe`: This message asks the SA to stop running a senselet.

SAs talk to OAs by the following message.

- `Update`: This message is sent by the senselet, requesting the OA to apply the series of XUPDATE command to its local database. The XUPDATE commands to be applied are encoded within the `Update` message.

Note that, this is not the complete set of messagess used by IRISNET. However, they will be enough for our description.

3.1 The Senselet

3.1.1 The PSF Senselet

The senselet for the PSF service is written using OpenCV, Intel's open source computer vision library. The source code of the senselet is in `$IRISNET/Applications/parking/SA/parking.cc`. The senselet should be compiled as a shared, dynamically loadable module, so that SA can load and execute them dynamically on receiving a `Subscribe` command.

The PSF senselet uses one configuration file `parking.conf`. The file describes the parking lot specific information, and is different for different SAs. The configuration files specifies the rectangular parking areas, along with the images of those parking spots while empty. The senselet periodically reads the snapshot of the parking lot from SA's shared memory segment, gets the current images of the parking spots (using the rectangular spot description in the configuration file), and for each parking spot, subtracts the current image from the empty parking spot image (defined in the configuration file). Finally, the subtracted value is compared with a threshold to decide whether the parking spot is empty or full. Although this simple algorithm may not be robust enough for using in a real, outside parking lot, it was enough for our tabletop simulated environment.

The senselet needs to encode the sensed data in an `Update` message. The `Update` command encodes a series of XUPDATE commands which the subscribing OA applies on its local XML database. Since the XUPDATE commands to be sent by the senselets depend on the SA location (different SA will update different parts of the global XML tree), the configuration file for the PSF service provides necessary information to construct the `XUPDATE` commands.

The `Update` command sent from the senselet to the OA should have the following format:

```
B <SA's IP> <total # of XUPDATE commands> <XUPDATE commands>
```

where, `'B'` is the OpCode for the `Update` command (`'U'` was taken by some other command before). Different components of a message are separated using a single space.

The first three lines of `parking.conf` are used to construct the `Update` message sent from the SA to the OA. The next integer defines the period (in seconds) of the message. The next integer says how many individual parking spots are in the lot, followed by that many lines defining the rectangular area of each spot and the image of that spot when empty.

3.1.2 Writing the Senselet

A few guidelines to write a senselet code are as follows.

- The main function of the senselet should be named `Start` and should take one argument of type `FilterParameter`. The argument information such as ID of shared memory containing the sensor data is, IP address of the subscribing OA etc. You will find the definition of the type `FilterParameter` in the file `SA/IrisSA/include/filter.h`.
- On receiving a `Subscribe` command, SA dynamically loads the senselet and runs the `Start` function once. If you want the senselet to periodically send the `Update` message for an indefinite time,

it should have an infinite loop. In case the senselet has an infinite loop, it should exit when it can not sent any message to the subscribing OAs (may be the OAs have left the system). OAs can stop the senselet by sending an `Unsubscribe` message to the SA running the senselet.

- The structure of a typical senselet should follow the structure of the PSF senselet. At the beginning, it opens the shared memory segment. Within each round of execution (*i.e.*, one round of the for loop), it reads the current video frame from the shared memory, performs a series of service specific tasks (in `parking.cc`, those steps are commented out as "PSF specific tasks" in the `Start` function) to construct an `Update` message, and then sends it to the subscribing OA by `ConnectToOA` and `SendToOA` APIs. The structure and most of the code of the PSF senselet are reusable by other senselets.
- The senselet should be compiled as a shared library (use the `-shared` option while compiling with `gcc`), so that it can be dynamically loaded by SA.

3.1.3 Uploading the senselet to the SA

The senselet should be placed in the directory `$IRISNET/SA/IrisSA/src/code` of every SA, since this is the default place where an SA looks for the senselet when it receives a `Subscribe` command from an OA. You can upload the code to this specific directory of the SA by using `scp` command provided by Linux, or using the `Load` command. In the second case, you can use the command line tool `$IRISNET/Util/SAControl/SAControl` to send a `Load` message directly to the SA.

3.2 The Database Schema

This section describes how to perform step 3 mentioned in Section 2. It consists of two logical sub-steps: defining the XML schema and populating the OAs with the schema.

3.2.1 The Schema

IRISNET organizes the sensor data in a hierarchical manner. Service developer needs to create an XML schema which defines, along with other information, the hierarchy used by the service. IRISNET envisions a rich and evolving set of data types, aggregate fields, etc., best captured by self-describing tags - hence XML was a natural choice.

The XML schema for an IRISNET service should include at least the following piece of information.

- The geographical hierarchy to be covered by the service.
- The *split points* in the hierarchy.
- IP address of the SAs.
- Static metadata used by the service.
- Sensor data, dynamically updated by the `Update` commands sent by SAs.

Figure 2 shows a part of the XML schema the PSF servce uses for the hierarchy shown in Figure 1. The complete schema will be found in `$IRISNET/Applications/parking/SA/demodataone.xml`. Before describing the schema in details, it would be helpful to know a few basic terminologies of XML. The basic building blocks of XML script are *elements*, *attributes*, and *values*. *Tag* is a general term that can be used for elements and attributes.

```
<psfROOT status="ownsthis">
  <usRegion id="NE" status="ownsthis">
    <state id="PA" status="ownsthis">
      <county id="Allegheny" status="ownsthis">
        <city id="Pittsburgh" status="ownsthis">
          <neighborhood id="Oakland" status="ownsthis">
              <total-spaces>10</total-spaces>
              <available-spaces type="child-aggregate">
                 <total>8</total>
                 <min-sensor-readtime>20020403133524</min-sensor-readtime>
              </available-spaces>
              <block id="1" status="ownsthis" expiry="999" owner sensor="1.2.3.4">
                 <name>Forbes at Morewood</name>
                 <streetaddress>Forbes at Morewood</streetaddress>
                 <available-spaces type="sensor-aggregate">
                     <total>3</total>
                 </available-spaces>
                 <parkingSpace id="1" status="ownsthis">
                     <usage> <in-use>no</in-use> </usage>
                     <meter>
                          <price-in-dollars>0.50</price-in-dollars>
                          <time-in-hours>0.50</time-in-hours>
                     </meter>
                     <handicapped>yes</handicapped>
                     <covered>no</covered>
                 </parkingSpace>

                 <parkingSpace id="2" status="ownsthis">
                  … … …
                 </parkingSpace>
              </block>

              <block id="2" status="ownsthis" expiry="999" owner sensor="1.2.3.5">
                … … …
              </block>
          </neighborhood>

          <neighborhood id="Shadyside" status="ownsthis">
            …  …  …
          </neighborhood>
        </city>
      </county>
    </state>
  </usRegion>
</psfROOT>
```

Figure 2: XML schema for the PSF service

[Element] A tag that is a container for subelements and attributes. For example, in Figure 2 `usRegion` is an element, that contains all the subelements and attributes in starting from the line line starting with `<state ...` to the line with `</state>`. The root element of an XML script is shown by the opening and closing tags for the entire file. The root element of the schema in Figure 2 is `psfROOT`

[Subelement] This is an element contained within a parent element.

[Attribute] Attributes flesh out the details of an element or subelement. For instance, `status` is an attribute of the root element in Figure 2

[Value] This is the setting of the attribute. For instance, the attribute `status` has the value `ownsthis` all over the schema in Figure 2.

An XML document can be represented as a tree with the root element as the root node of the tree. Each element in the XML document corresponds to a node in that tree. If `Y` is a sub-element of the element `X` in the XML document, then `Y` is a child of `X` in the corresponding tree. From now on, we will use the terms XML-document and XML-tree interchangeably.

The split points in the hierarchy are denoted by the *IDable* nodes in the XML schema. IRISNET allows the sensor database to be partitioned at any IDable node in the hierarchy. An IDable node has a special `id` attribute. The value of an `id` attribute is a short name that makes sense to the user query (*e.g.,* `Pitts-burgh`). The value is unique among its siblings, *e.g.,* there can be only one city whose `id` is `Pittsburgh` among the children of the `PA` state node. Moreover, an IDable node has an IDable parent node (or it is the root node of the hierarchy); thus an IDable node is uniquely identified by the sequence of node names and `id`s on the path from the root to the node.

Figure 2 shows some of the tags as underlined. These special tags and values have specific meaning to IRISNET and so should be used accordingly. Each IDable node (except the root) should have the attribute *id*, whose value must be unique among its siblings. An IDable node should also have the tag `status` which should be initialized with the value `"ownsthis"` in the XML-schema submitted to IRISNET. The node with a corresponding SA should list the IP address of the SA as the value of the tag `owner-sensor`.

IRISNET maintains one global XML document per service. The document is named after the root element, hence it should be unique among services. **The root element must have the substring `ROOT` in its name.**

The schema also should specify the static meta-data (*e.g.,* `total-spaces` in Figure 2) used by the service and the sensor data (*e.g.,* `total, in-use` in Figure 2) dynamically updated by the SAs.

3.2.2 Populating the OAs with the schema

IRISNET provides a very flexible way to partition the database among the participating OAs. IRISNET allows an OA to own (maintain the detail information of) any subset of the IDable nodes. IRISNET also allows an OA to dynamically delegate the ownership of any subset of the IDable nodes it owns to some other OAs. These two in combination allow a service developer to start the service with one OA, and then gradually distribute the schema among the participating OAs.

OA provides a number of APIs for doing this. Following are a few of the APIs provided by OAs to populate them with an XML schema.

1. `addfile <IP> <schema file>`: Adds the full XML schema defined in the `<schema file>` to the OA running on the host given by `<IP>`. The corresponding OA owns all the nodes of the tree. The API also registers the DNS style names corresponding to different nodes of the XML tree. This API is used to populate the first OA of the service.

2. breakfile <from IP> <to IP> <XPATH expression>: This API selects a node using the <XPATH expression> from the XML tree owned by the OA running on the host <from IP>. The subtree rooted at the selected node is then sent to the OA running on the host <to IP>. The API also updates the DNS entries of the nodes transferred to the new OA. The API is used to distribute different parts of the hierarchy within a set of OAs.

3. delete <IP> <document root>: This API removes the complete tree with the root named <document root> from the OA with IP address <IP>. It also removes the DNS entries of the nodes owned by this OA. The command is used to remove a service from IrisNet.

4. file <file name>: Executes the command listed in the file given by <file name>.

The commands can be issued in two different ways. First, from the $IRISNET/OA/IrisOA directory, run make runcl. You will get a command prompt where you can type the commands. Second, you can list the commands in the file $IRISNET/OA/IrisOA/script.cl and then run make runscript to execute all the commands listed in that file. The second approach may be helpful when you want to automate installing the service from a shell script.

For example, assume that the service developer has 7 OAs and she wants to distribute the hierarchy shown in Figure 1 among them. In the corresponding schema file schema.xml, she denotes each of the nodes shown in the hierarchy as IDable node (as shown in Figure 2). Suppose, she wants to delegate the ownership of the IDable nodes according to the partition scheme shown in Figure 1. To do that, she needs to issue the following sequence of commands.

```
addfile 128.2.206.1 schema.xml

breakfile 128.2.206.1 128.2.206.4 /psfROOT/usRegion/state/county/city/neighborhood[@id='Oakland']/block[@id='1']
breakfile 128.2.206.1 128.2.206.5 /psfROOT/usRegion/state/county/city/neighborhood[@id='Oakland']/block[@id='2']
breakfile 128.2.206.1 128.2.206.6 /psfROOT/usRegion/state/county/city/neighborhood[@id='Oakland']/block[@id='3']
breakfile 128.2.206.1 128.2.206.7 /psfROOT/usRegion/state/county/city/neighborhood[@id='Shadyside']/block[@id='1']

breakfile 128.2.206.1 128.2.206.2 /psfROOT/usRegion/state/county/city/neighborhood[@id='Oakland']
breakfile 128.2.206.1 128.2.206.3 /psfROOT/usRegion/state/county/city/neighborhood[@id='Shadyside']
```

Note the level order breaking of the schema file, with the lowest level broken first.

3.3 The User Interface

A set of web pages ($IRISNET/Applications/parking/UserInterface/*.html) provides the user interface for the PSF service. One web page provides the users with a form to fill up with her current address, destination address (since the prototype has only a few webcams, it allows the user to choose from a set of predefined destinations near the webcams), and the constraints (*e.g.,* a covered lot) of the desired spot. IrisNet returns the driving directions (using the Yahoo Maps service) to the empty parking spot that satisfies the constraints and is the nearest to her destination. The direction is updated periodically to reflect possible changes in the parking spot availability (*e.g.,* if the spot is taken up by someone or a better spot opens up).

The second set of web pages allows user to choose from a set of predefined XPATH queries (this can easily be extended to a form to be filled up by arbitrary XPATH query on the global XML document). The answer to the query is then showed in a formatted way.

The backend program for these web pages are written in Java (you can find them in `$IRISNET/Applications/Parking/FontEnd/*.java`). `YahooMaps.java` is used to serve the first webpage and `CannedQueries.java` is used to serve the second set of web pages mentioned above. The directory also contains a Makefile which you can use for building and running the programs. The backend programs listen on specific ports. The front end webpage should send the query request to those ports. The backend programs then query IRISNET and returns the formatted answer to it.

4 Conclusion

In this paper, we have described the technical details of developing a Parking Space Finder application on IRISNET. This should be used as guidelines to develop new sensor services on IRISNET. For more information, visit the official web site `www.intel-iris.net`, or mail to `irisdev@intel-iris.net`.

References

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

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