ICS203A - Ubicomp
Project Summary
November 29, 2004
Zhengming Tang, Sushil Bajracharya

PetTracker – Pet Tracking System with Motes

1. Abstract

This brief document summarizes the design and implementation of 'PetTracker', a location aware pet tracking application we implemented for our final course project. It also includes as a quick setup guide explaining its usage and deployment.

2. Introduction

Having decided we were going to use motes for our course project, the first task was to determine what kind of application we wanted to build. One of the unique characteristics of ubiquitous computing is its application-driven development. Unlike traditional software systems, an application built on small hardware motes needs to consider how they are going to be deployed and used, and take those considerations into all aspects of the design. The idea of using the motes to track a pet's activities came from watching a pet and wondering what he does when people aren't around. With that application in mind, we designed PetTracker.

The PetTracker system allows pet owners to log and track the activity of their pets throughout a certain period of time (can be couple of days) in an enclosed environment such as a house or apartment. Our current system requires CrossBow motes with environment sensor boards attached to them. One of these motes is attached to the pet and used to track its location and activity and its surrounding environment. The other motes are installed in different areas of the apartment, and these stationary motes act as tracking nodes as well as report their environment conditions. One of the stationary motes is attached to a serial programming board (MIB510) so that it can forward all the sensor data coming in from all the motes to a PC for analysis and visualization.

The PetTracker system is made up of the following three components.

Component 1: Hardware - Crossbow Motes, Sensors, and Interface Boards

The platform consists of Processor Radio boards (MPR) commonly referred to as MOTES. These battery-powered devices run Berkeley's TinyOS and support two-way mesh radio networks. They are manufactured and sold by Crossbow. We used a kit from Crossbow that included these motes (MICA2), sensor acquisition cards (MTS 300 and MTS 310) that are designed to plug into the motes, and gateway and interface boards (MIB510 and MIB600) that provide the interface between Motes and the PC. The interface board is connected to the PC using a serial cable.

Component 2: TinyOS applications running on Motes

Crossbow motes run applications that are written in NesC and compiled as TinyOS applications. We have modified a sample TinyOS application called SurgeReliable that comes along with the

CrossBow Motes toolkit to suit our requirements. SurgeReliable is a NesC application that collects sensor data from various motes and forwards the packets to a PC through a base station. The packets or messages in the sensor network are forwarded using a multihop ad-hoc routing protocol that is implemented as a NesC module.

The sensor data from each mote arrive to the base station in the form of a TinyOS ActiveMessage. This AM further encapsulates a MultiHopMessage in SurgeReliabe, and embedded within MultiHopMessage is an application specific message called SurgeMessage (see Figure 1). We modified the structure of the SurgeMessage to fit our purpose so that it can carry all the required data from the motes. We also modified the routing protocol to enable location tracking of the mobile mote.

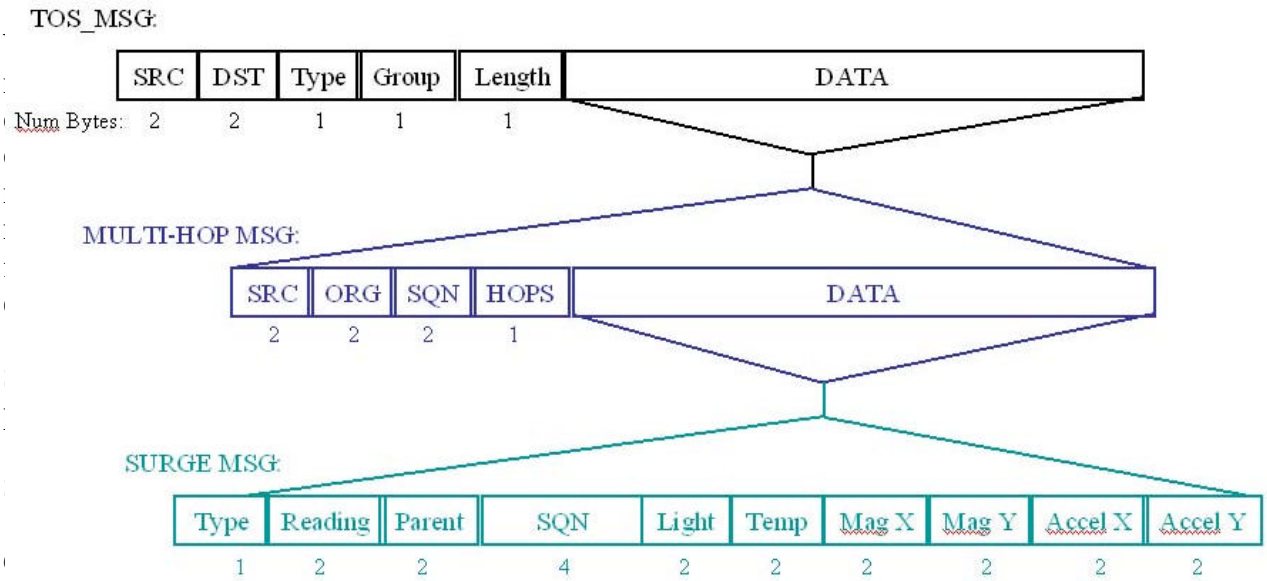


Figure 1. Message Format of packets from motes.

We use a Java application, SerialForwarder as a packet server. SerialForwarder is an application that comes along with the standard TinyOS distribution. It connects to the serial port and starts an embedded server to which clients can connect to and listen to the packets (TinyOS messages). This is the only basic functionality it provides.

We wrote a Java application that provides the following functionality:

- Connects to the serial forwarder and caches all the packets. Once the cache gets full (current cache size set to 8), it dumps the packets to a local persistent database.

- Process the data from the database and provide visualizations of the pets activity in past. Also show the current status of the pet based on the live packets coming in the serial port in a separate panel.

These two functionalities are described in the following section.

3. PetTracker Java Application

System Architecture

Figure 2 depicts the basic architectural diagram of our Java application. Major software components are shown inside grey boxes, messages inside yellow boxes and message flow as arrows. Dashed arrows represent message flow triggered by events.

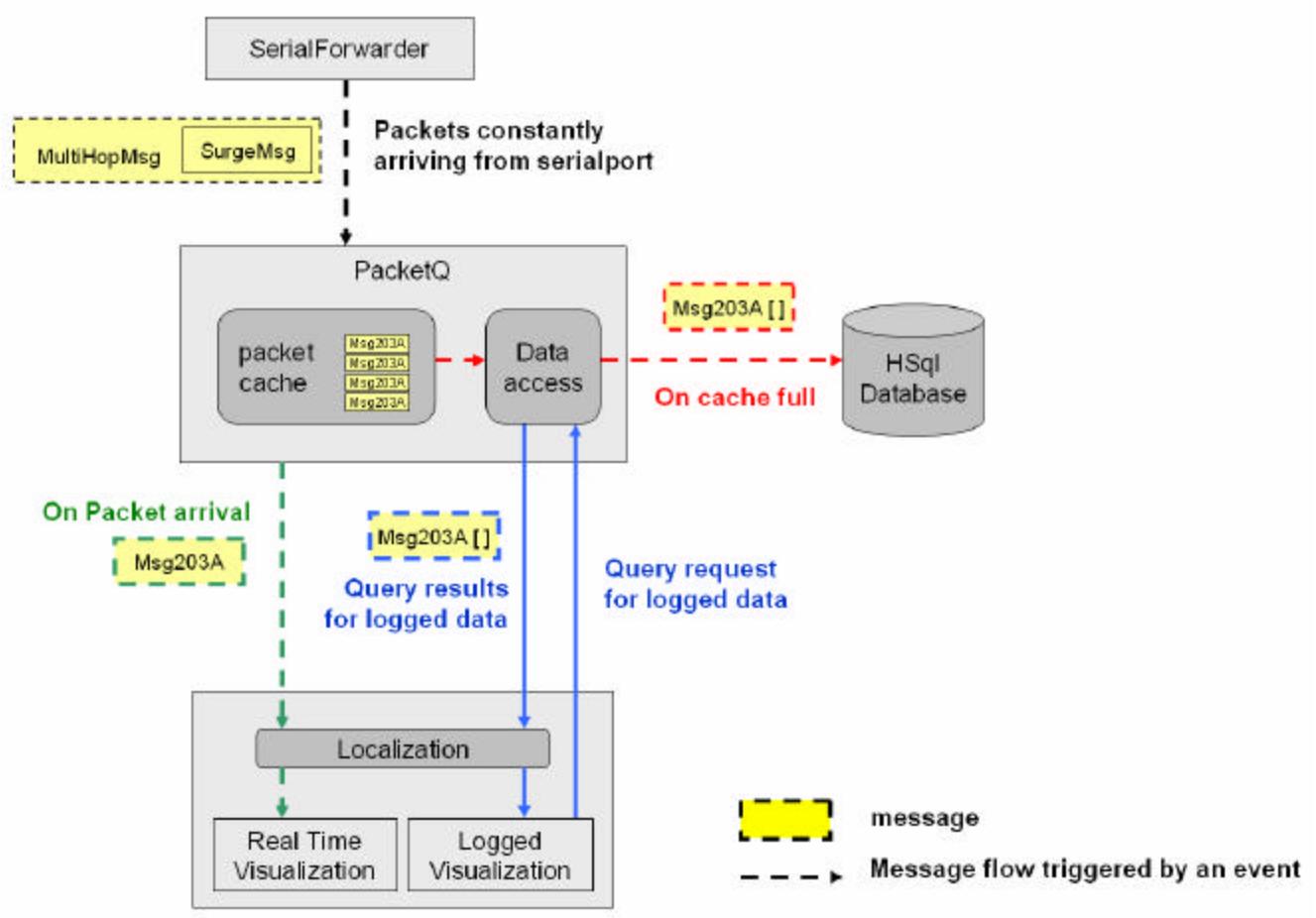


Figure 2. Software Architecture for Java Application

We described SerialForwarder, MultiHopMsg (Multi hop message, the TinyOS active message) and SurgeMsg (SurgeReliable's custom message embedded in MultiHopMsg) in section 2. The rest of the Java application can be divided into two sub-components: (i) caching/persistence module, (ii) visualization module.

i.Caching/persistence module

PacketQ: maintains a cache for incoming messages. The application registers itself as a listener to the SerialForwarder for all the raw packets coming in from the serial port (through the base station). With an arrival of each packet, it first is decoded as a MultiHopMsg and the SurgeMsg is extracted out of it. With all the relevant fields in these messages we create a packet specific to our application in form of an object of Msg203A class (see Appendix B, Table 1 for the fields of Msg203A). PacketQ caches all the Msg203A objects and once the cache grows to a particular size (that can be customized), it flushes all the packets to a database. We have use HSql, an embedded

in memory database server, that allows us to persist all the packets locally in the PC without having us to run a remote database server.

PacketQ also provides required interfaces so that, (i) other modules can query it for the logged packets (during last 'x' hours) and (ii) other modules can listen for the arrival of new Msg203A packets. It is a multithreaded layer of abstraction that hide the details of SerialForwarder, MultiHop and SurgeMessage, and the caching and persistence mechanism going on. The creation and addition of Msg203A packets in the cache, the flushing of the cache to the database, serving packet query results and event notification on new Msg203A packet arrival all are performed in a non-blocking fashion with appropriate synchronization.

ii. Visualization

There are 4 major visualizations that are done are the mote data:

1. Real time statistics – includes current location estimate.
2. Activity of cat for the last 24 hours.
3. Location tracking of cat for the last 24 hours.
4. Environment data from cat and stationary motes.

We use Java Swing for the user interface. The following are details on the various functionalites.

Localization (real time and past)

The localization algorithm is based on proximity. The mobile mote sends a SurgeMsg packet every second. A neighbor who can hear this packet takes that packet and sets the tracking address to itself, adds the received signal strength of the packet, and the battery voltage of itself. The sequence number of all these packets sent by the neighbors is going to be the same. The neighbor then sends this new modified packet to the base station. At the base station, we approximate the location by picking the neighbor mote that had the greatest radio signal strength (RSS).

For location estimates of past data, all packets originating from the mobile mote is retrieved from the database and sorted by timestamp. We essentially iterate through this list of packets, and for all messages that share the same sequence number, create a Msg203A object with the environment data and a list of other Msg203A objects representing packets sent by all the neighbors. The localize() method in Msg203A iterates through this list of neighbor packets and picks the neighbor with the greatest RSS value as the location of the cat. There is logic to take care of the wrapping of sequence number (since it's only given 1 byte), as well as packets that come out of order (fairly rare).

For location estimate of real time data, the algorithm is very similar, except we're getting live data directly from the serial port instead from the database. This means we have to wait to gather enough packets to create a neighbor list. We do this by waiting 0.75 seconds after receiving the first new packet from the mobile mote. We essentially give some time to the neighbors to send their packets before creating the list of neighbors and picking the one with the greatest RSS value. We know that 750 milliseconds is probably long enough to receive all these packets, but not too long that it would conflict with another new packet from the mobile mote.

Graphing and Mapping

For the simple x-y graphs (temperature, light, activity, and room changes), JFreeCharts is used. For the location panel, we have decided to also show a floor plan of the apartment and a slider that the user can control to highlight the appropriate rooms at various times. The slider also controls a vertical bar on the x-y graph that indicates which point on the x-y graph we're highlighting on the floor plan. The regions on the floor plan are defined by the user (see .gifs in /data/ folder), and the areas defining these regions are generated using the MapGenerator class (see the generated maparea.dat file). These areas are used for graphing the various regions on the floor plan.

Data Compression

Because there could be so many data points collected over the course of 24 hours, we employ some data compression of non-essential data before graphing it in the visualization. For example, to detect location and activity more accurately, we have the mobile mote send data every second. The stationary motes, not moving and having no activities, only send packets reporting their environment conditions (temperature and light) every 2 minutes (to save power). So we have much more environment data from the mobile mote than necessary. Thus, before displaying the mobile mote's environment, we average its environment data in 2-minute chunks so graphs can draw faster.

Data Conversion and Processing

The data from the Msg203A packets are raw when the visualizer receives them. These raw values need to be converted to real values (specifically for light and temperature). We used equations provided by Crossbow to convert these raw values. In addition, the acceleration data simply tells us the magnitude of the acceleration in x and y directions. Unless the accelerometer plane is perfectly perpendicular to gravity, a cat who is not moving at all is still going to have non-zero acceleration (due to the effect of constant 9.8 gravity it's measuring). In order to factor this out and display real activity, we calculate jerk, which is defined to be the change in acceleration, and display that as well.

For the overview panel, we also computed the following statistics:

- Max, min, and average temperature and light data for both the cat and the other stationary motes.
- Percentage of time the cat spends in each room.

4. Deployment - setting up and running the system

Our system depends on following third party software:

- SerialForwarder
- TinyOS Java Components
- HSql Database engine
- JFreeCharts

We have provided the entire source code for this system in a zip file. The NesC applications can be found in the folder /nesc/, and the Java application in the folder /java/edu/uci/ics/ICS203A/. We had the motes and base station setup according to the following configurations:

1. Base mote with ID 0 connected to the MIB 510 programming board. Also had MTS 300 sensor board attached to the base mote (through the programming board).
2. Programming board connected to the PC (windows XP) with a serial cable at COM1.
3. Other stationary tracking motes installed in various rooms with IDs 2 through 10.
4. The mobile mote was attached to the cat using a harness and had ID 1.
5. SerialForwarder was started on the PC to which the base station was connected (with mote communication set to serial@COM1:mica2).

If setup is correctly done, the application can be started by running the PetTracker.class file. Figure 3 shows the installation of the system that was deployed on the cat.

Figure 3 Installation of motes (Stacy's apartment)

Detailed Instructions – follow if you'd like to install and run your own system.

Part 1. Setting up the motes.

You need to have TinyOS installed as well as files from Crossbow copied on your computer.

1. Copy the two folders under /nesc/, MobileMote and TrackingMote, to your xbow apps directory (/tinyos-1.x/contrib/xbow/apps).

2. Hook up the MICA2 mote (no batteries and turned off) to the programming board and the board to the computer (follow instructions in Crossbow manual for different board types).

3. cd into /xbow/apps/MobileMote and run the make command to download to the motes.
   "make mica2 install,1 <programming board type>,<connection addr>"
   (If using programming board MIB510, then programming board type is "mib510" and connection address is the COM number, ie "com1". If using MIB600, then programming board type is "eprb", and you need to find the ip address of the board. Refer to Crossbow manual.)

4. Hook another mote to programming board and cd into /xbow/apps/TrackingMote and run the following make command to download to the motes. Repeat this one mote at a time.
   "make mica2 install,<stationary mote ID> <programming board type>,<connection addr>"

5. Put battery in all the motes except for the base mote with ID 0, which should be attached to the programming board. Attach sensor boards to all the motes (including the base mote if possible). Make sure to use MTS310 sensor board with the accelerometer on the mobile mote.

6. Put the stationary motes in various rooms where the mobile mote is expected to go. Put the mobile mote (ID=1) on the pet.

7. Make sure the programming board is set to OFF. Turn all the motes on (except for base mote).

Part 2. Running the Java Application.

Option 1) If you have want to compile the source code from scratch, do the following:

1. (optional) If you're deploying this (not in my apartment), then you should create your own floor plan and define regions using Photoshop, and then use the MapGenerator class to find the areas of those regions. You'll also need to change the source code (MoteData.java) to associate mote ID's with room names. If you're not deploying this, then just use the provided maparea.dat and maplabel.dat files.

2. Set classpath to include jfreechart (2 jars) and hsqldb (1 jar) before compiling:
   .jar files are in /ICS203A/java/jars/
   hsqldb.jar
   jcommon-0.9.6.jar
   jfreechart-0.9.21.jar

3. Compile all the java classes in /ICS203A/java/edu/uci/ics/ICS203A/
   cd /ICS203A/java/
   javac edu/uci/ics/ICS203A/*.java

4. Do some prep work by creating a local persistent database. You can skip this step completely if you just use the database files in the /ICS203A/java/edu/uci/ics/ICS203A/ folder (recommended).

   However, if you want to create your own database from scratch, then delete those database

files (hsqldbdata.*) and do the following:

cd /ICS203A/java

java edu/uci/ics/ICS203A/CreateDBTables

This will create some database files in whereever your compiled code lies. If you don't have motes and need some sample data, then run:

java edu/uci/ics/ICS203A/StoreHSqlDB

and this will fill the tables in so you'll have some fake (and not that useful) data for the visualizer. This won't look like much in the visualizer but at least the graphs will be there.

5. Then start the app in the following order:

   a. Make sure to turn on all motes (except for base). If motes are not available, then skip this step and you just won't get live data.

   b. Run SerialForwarder (in /tinyos-1.x/tools/java/net/tinyos/sf/)
   - set Mote Communications to "serial@COM1:mica2" (if using MIB510 and COM1).
   - start server

   c. Lastly, start the app:

      cd /ICS203A/java/

      java edu/uci/ics/ICS203A/PetTracker

      The visualizer's default settings will graph data from the past 24 hours. You can reset the date range in the overview pane if you want another time period.

Option 2) Use the compiled version of PetTracker with or without live motes.

1. Run the SerialForwarder (see above step 5b).

2. Then run the already compiled PetTracker Application.

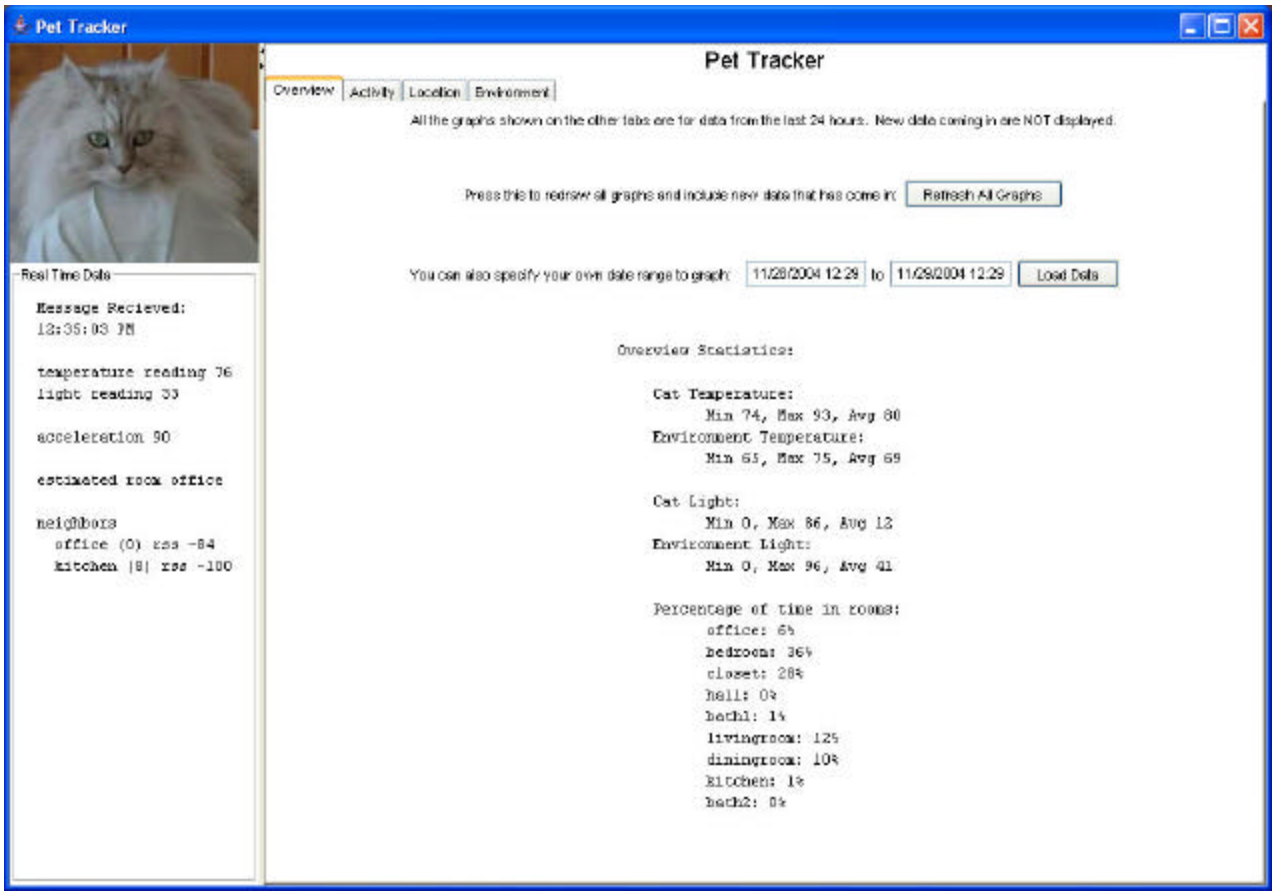
   cd /ICS203A/java/

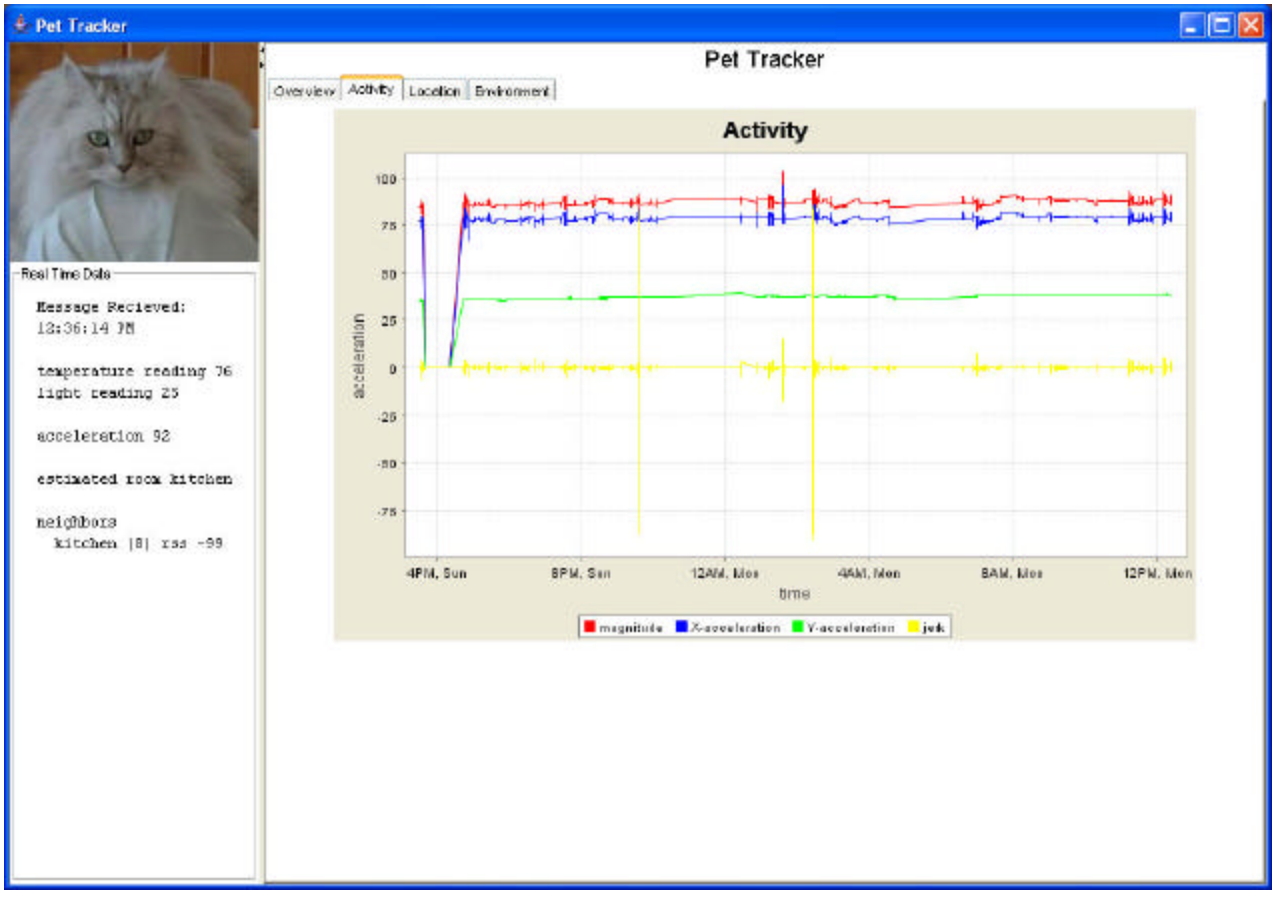
   java edu/uci/ics/ICS203A/PetTracker

   Specify the date range to: "11/28/2004 15:43" to "11/29/2004 12:17"

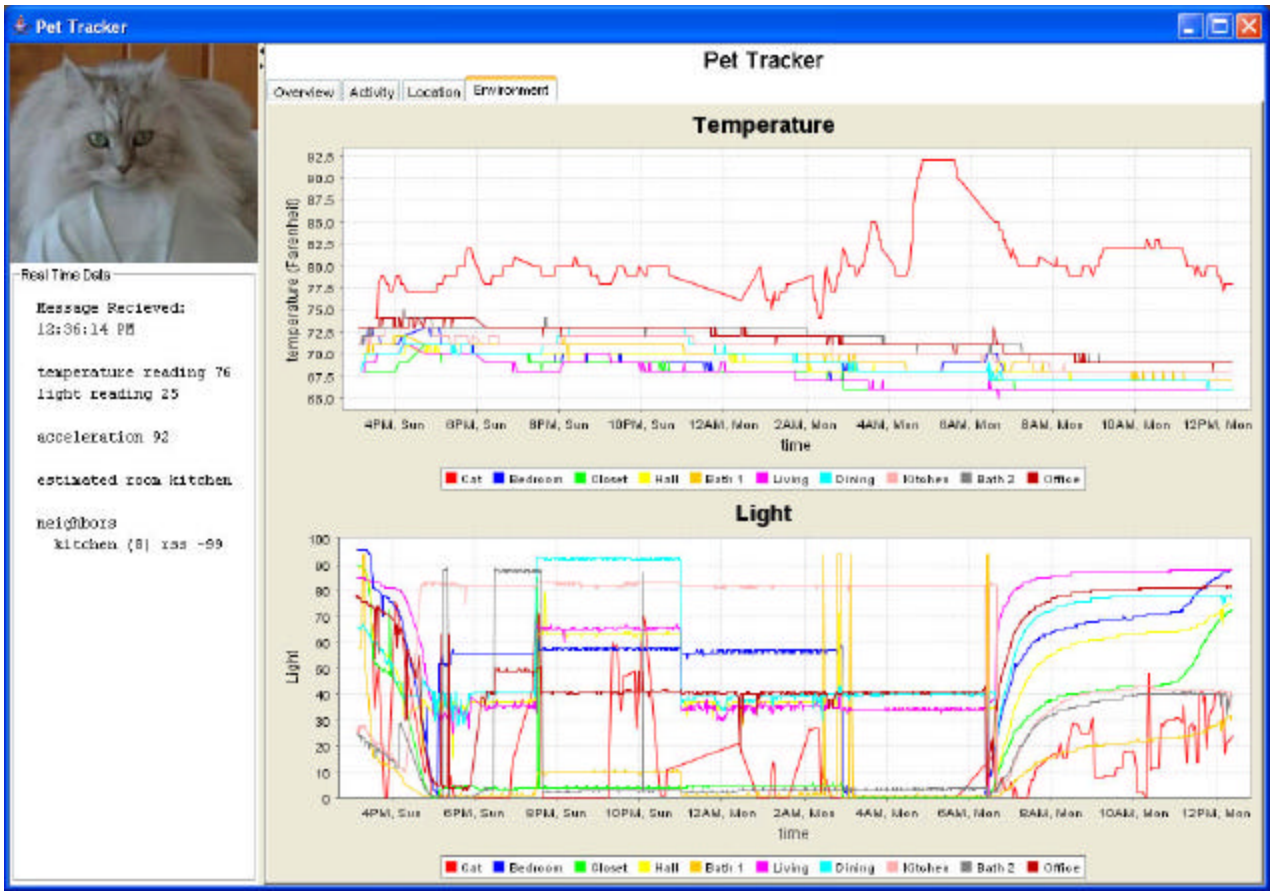
   We have included the database files with real data from deployment on the cat for this time period (Sunday afternoon to Monday noon), so you can play with the visualization on real collected data.

5. Sample Visualization Output






Pet Tracker
Pet Tracker
Overview
Activity
Location
Environment
PATIO
Walk-in Closet
W/D
Walk-in Closet
Linen
MASTER BATH
ENTRY
DW
Pantry
KITCHEN
Ref
BATH 2
Real Time Data
no current data
Location
bath2
kitchen
diningroom
livingroom
bath1
hall
closet
bedroom
Cat
office
room
4PM, Sun
6PM, Sun
8PM, Sun
10PM, Sun
12AM, Mon
2AM, Mon
4AM, Mon
6AM, Mon
8AM, Mon
10AM, Mon
12PM, Mon
time


Pet Tracker
Pet Tracker
Overview
Activity
Location
Environment
Real Time Data
Message Recieved:
13:36:14 PM
temperature reading 76
light reading 25
acceleration 92
estimated room kitchen
neighbors
kitchen (8) rss -89
Temperature
temperature (Farenheit)
time
Cat
Bedroom
Closet
Hall
Bath 1
Living
Dining
Kitchen
Bath 2
Office
Light
Light
time

6. Discussion

After deploying the system, we noticed a number of weaknesses with the system. Here's a summary of the issues and for future work, and some possible solutions for addressing the issues.

Inaccuracy with Radio Signal Strength

Because the received radio signal strength that is used for localization can vary due to so many factors, the localization that is currently performed is not very accurate. There are often times when it gives wrong estimates for the location of the mobile mote. The inaccuracy in the signal is likely due to the signal reflecting off walls and obstacles before it gets to the other mote (having direct line of sight between the 2 motes would prevent this but is impractical), as well as variations in the transmitters and receivers. These are well known problems with using RSS to estimate location [Dankwa; Hile].

Sound has proven to be much more accurate for localization and using the microphone in the motes for localization would yield much better results. If we can move the sound emitted by the motes to be out of the audible range (for people and animals), then sound is certainly a better option.

Mobile Mote transmission unreliable

Even though the mobile mote is suppose to be sending data every second, very often for minutes at a time it will disappear and not send packets at all. Most of these incidences occur after the mobile mote is moved, which leads us to believe that sometimes it just takes it a long time to determine who the parent node is for routing before sending out packets again.

One thing we can do is to modify the mobile mote NesC code so it simply broadcasts its packets all the time (and never receives or relays packets). And modify the tracking mote code so they know about this. Currently the mobile mote still participates in the ad-hoc multi-hop routing scheme, which means it will find a new parent when it has moved and relay packets from other stationary motes if it's a parent of some other mote. However, we know all the neighbors of the mobile mote relays packets from the mobile mote, not just the parent, so it hardly matters if we find the parent at all. Thus, we essentially make the mobile mote not participate in the routing protocol. We can take the extra routing algorithm out of the mobile mote and only have it broadcast its message to whoever happens to be nearby. This would likely fix this disappearing problem.

Mote too large for cat

One thing we learned from deployment is the MICA2 motes are too large for a cat or any small size dog. Because of its large and heavy size, the only way to attach the mobile mote to the cat is to strap it to its back. This is done by first attaching the mote to a cat harness, then strapping the harness on the cat. However, the mote is rather heavy and will not stay put. It often slides off to the side and thus became accessible to the cat, who tried very hard to lick and pull it off. We amended this temporarily by enclosing the mote in cardboard (so the mote won't be damaged by the cat), but this solution meant worse radio signals and loss of light data for the cat.

A much better solution to this problem does exist. Crossbow manufactures another type of mote, the MICA2DOT, which is quarter sized and use a small battery (instead of 2 AA), and also a sensor board made for this smaller mote (MTS510). The MICA2DOT with the MTS510 would be much more appropriate for attaching to an animal. They are small and light enough to hang from a collar and would be much more comfortable and likely not bother a cat that is already used to wearing a collar.

7. Acknowledgments

We'd like to thank Raja Jurdak (fellow ICS student) for writing the NesC application for the motes as well as stimulating discussions and giving us advice on localization. We are grateful for his knowledge of the motes that he generously shared with us.

We'd like to thank Harlan Hile (co-owner of cat and Ph.D student at UW) for his practical advice throughout the project and his help in deploying the application.

8. Credits

Raja Jurkak – NesC applications
Sushil Bajracharya – Cache/data storage
Zhengming Tang – Localization and visualizations; deployment
Harlan Hile – Map generator (for floor plan visualization); deployment
Gandalf (the cat) – for being the initial inspiration and a patient and tolerant subject!

9. References

www.tinyos.net

www.xbow.com

Nana Dankwa, "An Evaluation of Transmit Power Levels for Node Localization on the Mica2 Sensor Node" (http://zoo.cs.yale.edu/classes/cs490/03-04b/nana.dankwa.ee.pdf).

Alex Yates, Jonathan Ko, Harlan Hile. "WiFi Peer-to-Peer Location Service", (http://www.cs.washington.edu/homes/harlan/561paper.pdf)

Appendix A.

NesC Application Component Diagrams

Please see files **Mobile_Mote.htm** and **Tracking_Mote.htm**. Note these diagrams are generated using the "make docs mica2" command in TinyOS. Only these two html files are included with this turn in, so do not follow the links on these pages.

Appendix B.

Table 1: Fields in Msg203 class

Java Data type	Field name	Description
long	timestamp	Indicates when the packet arrived in the PC
int	originAddress	Mote ID where the packet was created [0-10].
int	trackingAddress	The first tracking mote to receive this packet from the mobile mote. Used for localization (stored in Reading field of SURGE_MSG) and not used by packets that originates from stationary motes.
long	parentAddress	Address of the routing parent of the mote
long	SurgeSeqNum	seqNum field of SurgeMsg (first 3 bytes is battery voltage, last byte is actual packet sequence number).
long	battVoltage	The battery voltage of the tracking mote that relayed the mobile mote's message. First 3 bytes of SurgeSeqNum.
long	seqNum	Sequence number of the packet and last byte of SurgeSeqNum.. All tracking motes that relay the packets from mobile mote will use the same seq num assigned by the mobile mote. (ie. mobile mote send a packet with seq num 128, 3 neighbors including parent hears the packet and each relays it. In the end, 3 packets arrive at base station with seq num 128.)
short	Light	Raw light value (percentage of greatest light value detectable).
short	temperature	Raw temperature value.
short	signalStrength	Radio signal strength of stationary mote that received the packet from the mobile mote.
short	acclX	Acceleration of the mobile mote in the X direction
short	acclY	Acceleration of the mobile mote in Y direction
long	multiHopSeqNum	Sequence number given to the packets for multi-hop routing.
short	hopCount	Used for routing only.
int	sourceAddress	Address of the last mote that sent the packet. Used for routing only.
int	locationId	Location of mote that sent packet. Filled in by localization code during visualization.
ArrayList	neighbours	Neighbors of mobile mote. Filled in by localization code during visualization.