EXPERIENCES FROM A HOME SENSOR NETWORK DEPLOYMENT FOR ASSISTED LIVING

Thiago Teixeira
Electrical Engineering Department
Yale Univerisity
New Haven, CT 06520, USA
thiago.teixeira@yale.edu

Dimitrios Lymberopoulos
Electrical Engineering Department
Yale Univerisity
New Haven, CT 06520, USA
dimitrios.lymberopoulos@yale.edu

Andreas Savvides
Electrical Engineering Department
Yale Univerisity
New Haven, CT 06520, USA
andreas.savvides@yale.edu

Abstract We report on a home wireless sensor network deployment that utilizes cameras to collect activity information. This paper describes the design choices and the general experience of maintaining such a data-heavy network. The system includes camera nodes, a classifier program to distill the packets into appropriate database tables, and a client-side user interface for diagnosing and interacting with the network.

1. Introduction

The accessibility of sensor platforms makes their experimental deployments an attractive approach for better understanding application requirements and challenges for sensor network research. Over the last few months, our research has pursued one such deployment inside a house,

for an assisted living application. In this application our main objective is to devise lightweight sensor networks that can understand people's activities to a level at which they can autonomously provide meaningful services. For practical and cost purposes, our network is expected to have a small number of nodes, but should be able to operate reliably for several months collecting real data for validating our research hypothesis about behavior interpretation.

Our initial deployment consists of 6 sensor nodes, 5 carrying image sensors and 1 acting as a gateway to the base PC. The sensor nodes were attached to the ceiling of the house in the configuration shown in Figure 1. Each node carries a camera module with a wide-angle (162-degree) lens facing perpendicularly down into the room. The nodes compute the locations of people in the house using an image-processing algorithm to detect the centroid of each moving blob. The time-stamped centroids are then forwarded to a base station where they are stored and later on processed for behavior interpretation.

Instead of describing the details of our behavior interpretation work, in this paper we outline the challenges and experiences we have encountered during the first two months of deployment. The paper is divided in three main parts: The first part of the paper introduces our testbed setup and our requirements. The second part of the paper provides a detailed account of the challenges we faced during the actual deployment. The third part describes our solutions to some of the issues we faced together with a list of conventions and recommendations we have established to keep our testbed running in a long-term deployment.

2. Deployment challenges

The prototype network reported in this paper was built with the goal of continuously monitoring the activities that happen inside a home. Moreover, the system should lend to quick deployment, it should be transparent about its operation and vital signs, perform data consistency checks, and, finally, resist unpredictable mishaps as smoothly as possible. Also, for the system to scale to a large number of homes with possibly different functionalities, our architecture should provide mechanisms for reconfiguring and retasking the network after its deployment. This set of constraints should make it possible to sustain a stable deployment for months with little to no maintenance.

In order to fulfil the first constraint, we opted to equip our sensor nodes with cameras with wide-angle lenses. The reasons were twofold: first, given the broad 162-degree angle of view, a small number of camera nodes was able to provide a large coverage area; the entire network con-

sisted of a mere 5 camera nodes plus a base node, distributed over the 2-floor house as shown in Figure 1. As can be seen in that figure, the deployment did not cover sensitive areas such as bedrooms and bathrooms. This is due to the privacy issues that are raised by the presence of cameras, which leads to the second reason why image sensors were picked to begin with: part of our research is to develop an architecture of "blind" image sensors, which provide rich *information* but are not able to *take pictures* [6]. These sensors do not behave like typical imagers: their pixels are capable of asynchronous computation, and provide a summarized form of the data in the image as a continuous data stream — not in frames. Upon completion, this imaging platform will likely replace the off-the-shelf cameras used in this deployment, for added privacy and processing speed.

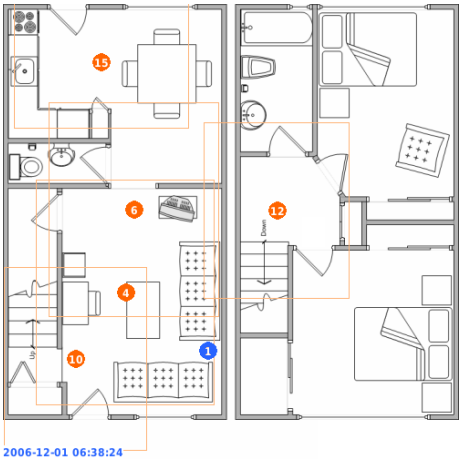

Figure 1. Floorplan of deployment house, including node positions and approximate coverage. The left side shows the bottom (street) level, and the right side shows the top level.

Despite the small number of nodes in the deployment, the sheer amount of data that it must process is sizeable. Considering that each camera takes 8 320 × 240 pictures per second, over 24 Mbits must be processed each second within the network for the cameras alone. Given the well-known capabilities and constraints of WSNs, there is an evident need to select a compact set of attributes to be filtered from the images and transmitted by each node. These attributes should provide information that is pertinent to our behavior-recognition platform [4]. What is more, due to the low transmission rates encumbered by WSNs, there is no possibility to record ground-truth data for comparison and debugging. Therefore it is imperative that the transmitted

data give enough insight on the activities within the network for the developer to look through and understand. With all this in mind, it was decided that the nodes compute and transmit only a stream of the $\{x, y, timestamp\}$ values of the centroids of the moving bodies within a scene, since bandwidth-hogging video streams were out of the question. This reduces each node's data transmissions from nearly 5 Mbps to at most 442 bps.

3. System description

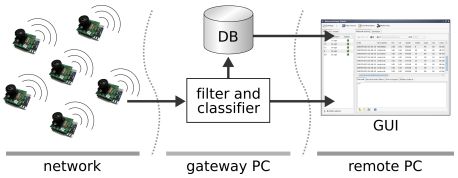

Figure 2. Block diagram of the entire deployment.

Figure 2 shows a block-diagram overview of the configuration of the entire system. A sensor-network is deployed in a home that is also equipped with a local gateway PC. Data can then be streamed to or queried from any other computer for on-line or off-line computation, respectively. Next, we go into more detail about each part of the system.

3.1 The network

The deployed network is composed of iMote2 [5]sensor nodes running the SOS-1.x Operating System [3]and carrying a custom camera-board. The iMote2 is a wireless sensor node made by Intel that contains a PXA271 XScale processor and a 2 GHz 802.15.4 radio from ChipCon, the CC2420 [2]. The frequency and voltage of the PXA are dynamically scalable (13 MHz to 416 MHz), and there are five major power modes. What is more, the iMote2 provides 256 KB of integrated SRAM, 32 MB of external SDRAM, and 32 MB of flash memory. The nodes run the SOS kernel with iMote2-specific drivers as well as the tree-routing module that is standard in SOS.

As mentioned earlier, a custom camera-board sits atop each iMote2. These boards contain an OmniVision OV7649 camera, which can capture color images at 30 fps VGA (640×480) and 60 fps QVGA (320×240). Our experiments with the USB version of the OV show that a resolution of 80×60 is enough for centroid calculation without loss of accuracy, given that it is far above the Nyquist rate for locating humans in these images. Thus, within the PXA we run a linear downsampling algorithm

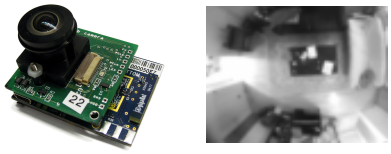

Figure 3. Left: the iMote2 node with a custom camera board and wide-angle lens for added coverage. Right: picture taken by node 4 in the deployment, after downsampling and dropping color information within the node.

to convert from 320×240 to the more manageable 80×60. After each frame is downsampled, the new frame is compared to the one stored in memory and the centroids are extracted from each moving object. The node then time-stamps each centroid with the value of its real-time clock and packs every 10 centroids together for transmission inside a message of type *CENTROID_MSG*. On the other hand, if a node detects 9 or less centroids and 10 seconds have passed since the last detection, the partially filled *CENTROID_MSG* message is sent. This assures that the centroids in each message have close timestamps and describe information from the same context, allowing on-line parsing. Also note that, for data-logging reasons, each *CENTROID_MSG* is provided with a sequence number.

Along with the *CENTROID_MSG*s, other messages circulate in the network, including *HEARTBEAT_MSG*s, and messages of type *GET_RTC* and *SET_RTC*. There are also additional messages and facilities such as those involved in snapping and transmitting an image of either 320×240 or 80×60 resolution. Although it is currently possible to download a picture from the network at any point in time, we typically do so during the set-up process only. Therefore, a logical addition to this deployment is a mechanism that blocks picture downloads at later stages, again, due to privacy concerns.

The aforementioned *HEARTBEAT_MSG*s were introduced after the first point version of the deployment to periodically supply information relating to the internal state of each node and expose the vital signs of the network allowing the computation of network statistics. Nodes dispatch a *HEARTBEAT_MSG* 15 seconds after the transmission of the latest *CENTROID_MSG*, and then every 15 s after that repeatedly. Thus, no *HEARTBEAT_MSG*s are sent from a node when there is activity within that node's field of view.

Each *HEARTBEAT_MSG* contains the timestamp and sequence number information for itself as well as for the last transmitted *CENTROID_MSG*. With this data at hand, the status of the network can be always inferred.

Finally, the two remaining message types, *GET_RTC* and *SET_RTC*, are part of a simplistic time-sync protocol that was implemented. The former is transmitted periodically every time a node boots. In the meantime, no other activities are allowed to take place in the node. Upon receiving a *SET_RTC* response message, the node then synchronizes its real-time clock and commences capturing centroids and sending heartbeats. All RTCs are expressed in seconds since the Epoch (midnight UTC of January 1st, 1970). This detail allows nodes to be started and stopped at will, with the assurance that the recorded centroids will always have unambiguous timestamps.

This feature is crucial for the implementation of the last line of defense of the network: watchdogs. Each node's watchdog forces a reboot when the PXA's internal counter (OSCR0) matches the values of the watchdog timer (OSMR3). This way, the watchdog timer may be advanced by a predefined time interval within a reocurring routine to ensure that the node reboots if that operation does not execute within a specified timeout. In the camera-nodes, this routine is the *picture ready* function that is called by the camera module. Hence, the node reboots if some mishap caused it to stop taking its 8 pictures per second. At this point the node synchronizes its RTC then starts collecting centroids, as discussed in the previous paragraphs.

The only node that does not operate as described above is the base node. This node acts as a gateway between the network and a local PC. It simply forwards to the USB all radio messages that are addressed to it, and transmits into the network all messages dispatched by the PC. This node also has a watchdog service, to force a reboot after long periods of inactivity.

3.2 The local and remote PCs

Connected to the base node is a gateway PC equipped with a MySQL 5 database management system. As is common with SOS networks, this computer constantly runs the SOS Server daemon, which provides a standard socket interface. This way, all it takes is a standard TCP client to send messages to and from the WSN. Note that, due to its ubiquity, the SOS Server is not directly represented in Figure 2.

The next component in the gateway PC's software stack is a custom program called the Classifier. This component acts as a bridge between the network and the database and GUI. That is, the Classifier reads the incoming messages from the SOS Server (ie. the messages sent by any node to the base node) and properly redirects them to the appropriate table in the database. All packets are stored in an all-purpose

raw-packet table with a column for each SOS message field, plus an additional column recording the reception time (see Figure 4). Additionally, centroid packets are also parsed into centroid arrays that are stored in a distinct table. The centroid table stores all the information from the centroid $(x, y, timestamp)$ as well as from the packet in which the centroid was transmitted. This redundant schema was chosen for added performance when parsing the table data, since it does away with expensive operations such as joins.

Another function of the Classifier is to provide the current time to the base node's RTC. Hence, the Classifier is able to respond to *GET_RTC* messages that are issued by the base. Much like the other nodes, the base does not perform any other function while it waits for its RTC to be synchronized, for the reasons already stated.

Raw Packets Table							
time	did	sid	daddr	saddr	type	len	data

Centroids Table									
time	did	sid	daddr	saddr	type	len	x	y	rtc

Figure 4. Structure of the database tables that are currently used in the deployment.

Finally, we have created a Python module called PySOS for interacting with the network, as well as the graphical front-end to PySOS seen in Figure 5. Through PySOS, it is possible to send/receive SOS messages, post remote procedure calls (and return the reply) and set up message listeners for asynchronous handling. Accordingly, the PySOS Control Center front-end provides all these features through an embedded Python console, but also displays all received packets in a sortable grid view, keeping track of the time passed since each node last communicated. These utilities that proved essential in quickly determining the status of the network are described in the section that follows.

3.3 Messaging and PySOS

As explained earlier, PySOS is a Python module that allows one to send and receive messages to and from an SOS network. Sending a message is as easy as issuing a `post(daddr=15, did=140, type=32, data='\x00\x01\x02\x03')`, for example, to send a message of type 32 to module 140 on node 15. The contents of the message are 4 Bytes containing the numbers 0, 1, 2 and 3. The parameters names ('daddr', 'did', etc) coincide with the typical SOS messaging vernacular, so the

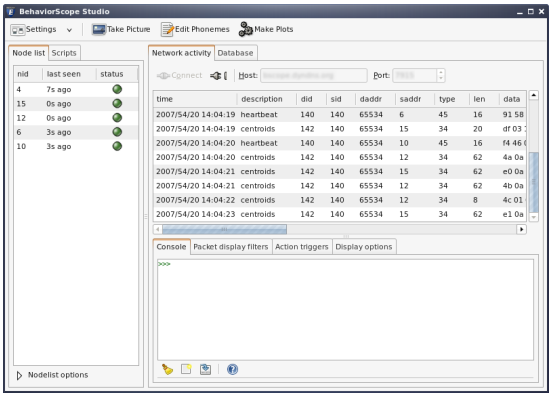

Figure 5. Screenshot of the graphical user interface developed for monitoring and interacting with the deployment.

`post` command should be instantly understandable to those accustomed with SOS.

Similarly, PySOS makes it easy to receive messages from the network (either by registering asynchronous listeners, or by using "blocking" function calls) as well as to issue remote procedure calls. The full source code and manual to PySOS are made available through our website, at `http://enaweb.eng.yale.edu/drupal/pysos`.

At that URL one will also find the PySOS Control Center GUI, which works as a front-end to PySOS. The GUI can connect to any local or remote SOS server, automatically monitoring all incoming traffic and displaying it in a list box while keeping track of each node's status (Figure 5). A quick look to this node list is enough to ascertain which nodes are "alive" and which are having trouble communicating.

Through the GUI, the user may also create message filters that describe the messages that should be appended to the network activity list box, and action triggers for designating an action that should be executed for each matching message. These triggers can be used to parse data in real-time, through user-defined Python functions.

An embedded Python console is also present on the PySOS Control Center, so that any other Python and PySOS command may be issued from the same interface. Any Python module installed in the system may be utilized from the embedded console. As an example, the Python module for the scripting and reconfiguration framework described in [1]can be directly accessed in order to retask the network as appropriate.

The PySOS Control Center can be further utilized to query the stored database of received packets. This way the GUI handles both on-line and off-line data in a seamless manner. What is more, given the object-

oriented nature of the PySOS control center, programmers may sub-class it such that other features may be easily added without the need to modify the GUI's source code. In this way, we have extended our internal version of the PySOS Control Center to accomodate capabilities that are pertinent only to our specific deployment configuration, such as downloading pictures from the network, plotting live and stored statistics, making animations of the centroids on the deployment floorplan, etc.

4. Network analysis

The deployment has been running for over 8 months, with many software upgrades in the first two months. The latest version dates from early November 2006. Through those upgrades, some features were added (such as tracking statistics with the heartbeat messages) and a few bugs were fixed. Interestingly, the majority of the bugs encountered — and the hardest to track down — were in the MAC layer of the CC2420 code from SOS. After the discovery of these bugs, the radio code was further modified for increased robustness with the addition of CRC checks, and the implementation of ACKs and retries.

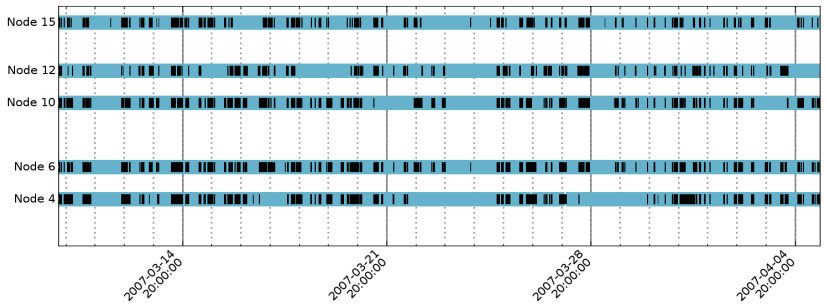

Figure 6. Plot depicting the type of traffic produced by each node over time. Each packet is represented by a thin vertical line and the colors show the packet type. Blue indicates *HEARTBEAT_MSG*s while black is for *CENTROID_MSG*s.

Figures 6, 7 and 8 depict the behavior of the deployment over a window of 25 days. The first one, Figure 6, shows the nature of the traffic on the network. Each heartbeat packet sent by a node is drawn as a blue vertical line, while each centroid packet is represented in black. From this depiction it is clear that centroid packets are typically sent by multiple nodes at nearly the same time, as it is common for actions to span the coverage area of more than one node.

Figure 7 shows a cumulative plot of the number of centroid packets that never reached the base. The figure shows that, as expected, there is a rough correspondence between packet drop rates, distance from the

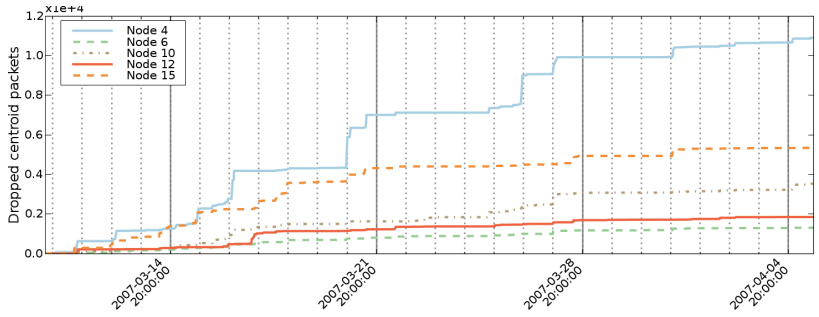

Figure 7. Cumulative plot showing the number of dropped centroid-carrying packets over the course of 25 days. Y-scale is $\times 10^{-4}$.

base, and number of obstacles in between. Two nodes, however, do not follow this analysis: nodes 4 and 10. Remarkably, despite being close to the base, node 4 showed the worst centroid drop rate. Also worth noting is the apparent correlation between the sudden spikes in packet drop across multiple nodes. This is especially when comparing the behavior of nodes 4 and 10. Incidentally, these two nodes are the two that are the closest, geographically. The source for these jumps must be associated with the observation from the previous paragraph: that centroids are usually sent in bursts by many nodes at the same time. In other words, given the physical properties of the events being sensed, medium contention may be ocurring each time a person moves from one sensor to the next. This would be especially accentuated when multiple people are in the house. If this is the reason for the packet drops, then the number of dropped heartbeat packets (Figure 8) must exhibit a much different behavior, due to the regular traffic pattern of the heartbeat transmissions.

Indeed, this is what is seen in Figure 8. Had the cause of the centroid drops for any of the nodes been an independent factor such as antenna orientation, then the number of dropped heartbeats would follow the suit. Instead, what is seen is that nodes 4, 10, 12 and 15 drop around 4000 packets each, with node 4 dropping most of them at once, very early on. After that moment, nodes 4 and 6 show a similar drop rate, as evidenced by the slope of their plots.

In the 25-day window, there were close to 23 thousand dropped centroid packets, making up 10.2% of all attempted centroid transmissions. Since heartbeats are only sent when there is no centroid activity, and since they are transmitted in relatively sparse intervals, the smaller number of dropped heartbeats (around 16.5 thousand, or 1.56% of all attempted transmissions) seems to agree with the analysis that the major

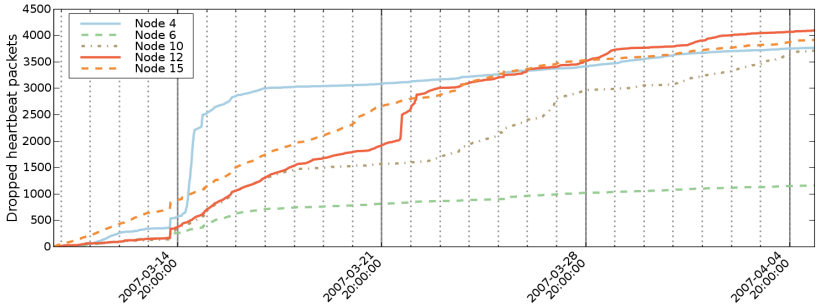

Figure 8. Cumulative plot showing the number of dropped statistics-carrying (heartbeat) packets over the course of 25 days. The plot shows a smoother behavior that contrasts with the bursty quality of the dropped rates in Figure 7

culprit of packet drops is packet collisions due to the traffic pattern imposed by centroid transmissions.

5. Conclusion

We developed and deployed a sensor network for home activity monitoring, complete from the camera board to the graphical user interface. The network has been functioning properly for extended periods encompassing several months. This is due to the preventive measures taken in the design phase, ranging from the use of a watchdog (which allowed the network to function despite the initial bugs in the radio code), to the simple act of protecting the nodes by a plastic enclosure. Oftentimes, such details are overlooked and end up raising many issues. Moreover, inspite of moderate packet loss rates and simplistic approach to time synchronization, the data contains the right information for use in our research.

Future improvement may include a more appropriate routing algorithm to be decided after careful analysis of the longterm network statistics. Also present on the list are a time synchronization method with better accuracy, improvements to the sensing layer and the addition of debug messages to be sent through the radio each time an "assert" statement fails.

Returning to the initial requirements set forth in Section 1.2, all but one have been tackled in the course of this paper: there is still the question of whether the data produced by the network is consistent. Consistency checking (*does the data make sense?*) differs from simple validity checks (*is the data within the appropriate values?*), and typically goes

unhandled. Part of our ongoing research is to employ a data consistency check that arises naturally from our behavior recognition platform.

References

[1] A. Bamis, N. Singh, and A. Savvides. An architecture for dynamic reconfiguration of data flows in sensor networks. In *Submitted to the Fourth Workshop on Embedded Network Sensors (EmNets '07)*, 2007.

[2] Chipcon: CC2420 802.15.4 compliant radio. http://www.chipcon.com.

[3] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. Technical Report NESL-TM-2004-11-01, University of California Los Angeles, Networked Embedded Systems Lab, November 2004.

[4] D. Lymberopoulos, A. Ogale, A. Savvides, and Y. Aloimonos. A sensory grammar for inferring behaviors in sensor networks. In *Proceedings of Information Processing in Sensor Networks, IPSN*, April 2006.

[5] L. Nachman. Imote2, http://www.tinyos.net/ttx-02-2005/platforms/ttx05-imote2.ppt, 2006.

[6] T. Teixeira, E. Culurciello, E. Park, D. Lymberopoulos, and A. Savvides. Address-event imagers for sensor networks: Evaluation and modeling. In *Proceedings of Information Processing in Sensor Networks, IPSN*, April 2006.