A Prototype Distributed Framework for Identification and Alerting for Medical Events in Home Care

Gary Holness
Delaware State University
gholness@desu.edu

Leon Hunter
Delaware State University
xleonhunter@gmail.com

Abstract

The confluence of low-cost embedded sensing and pervasive Internet access and algorithms for machine perception has the potential to revolutionize home healthcare. Wireless connectivity for devices such as inertial measurement units (IMUs), heart monitors, blood O2 sensors, and glucose monitors provides a method for valuable Web-accessible continuous monitoring data that can be used as the basis of indicators for unfolding medical issues in a home care setting. We present a prototype service implementation using a widely available open source hardware reference implementation, the Arduino/eHealth Sensor, for intelligent medical monitoring targeted for home-care environments. This system fits within a larger distributed framework for medical monitoring. In this work, we have solved a number of important technical issues in implementing a stable medical monitoring platform whose sensors are accessible using a typical Web-browser.

Introduction

Aging in place, or the ability to live in one's own home or with family throughout the senior years, is a viable model for reducing strain on long-term care facilities in addition to improving quality of life for the chronically ill. In-home care is also a viable solution for easing the impending burden on the healthcare system imposed by an aging population. Key to improved management and outcomes for care of chronic illness such as diabetes, heart disease, stroke, and Alzheimer's disease includes regular visits with home-care and allied health professionals for the measurement and assessment of vital statistics. Too often, in practice, the day-to-day management of care rests on the patient and his or her family [8]. Depending on condition, a typical elderly patient managing chronic illness can expect only a handful of weekly visits from a home healthcare nurse. While family care is certainly an option, more households from the so-called sandwich generation find themselves caring for both their children and their elderly parents while balancing a career. All of this results in decreased quality of care because subtle changes in health status are easily overlooked.

We have begun to see wireless devices such as inertial measurement units (IMUs), heart monitors, blood O_2 sensors, and glucose monitors provide valuable continuous monitoring data that can be used as the basis of indicators for unfolding medical issues [9]. Armed with such tools, a patient's healthcare team (physicians, allied health, and family) would be better

able to coordinate an early intervention before an issue degrades into a catastrophic event. The proliferation of sensing devices and low-cost embedded systems has spurred the growth in sensory technologies and perceptual robotic systems. Today we are beginning to see the emergence of perceptual applications such as face detection in digital cameras, location monitoring using smart phones, and player motion detection in video game consoles. Machine perception can provide the eyes and ears that monitor daily patient activities and vital measurements, while robotic assistants serve as remote helping hands capable of assisting in the performance of daily activities and certain interventions.

What began in 2005 as a tool for students to experiment with embedded systems, the Arduino platform has grown into a worldwide community of relatively inexpensive controller boards, sensors, and accessories that has fueled embedded systems development in do-it-yourself, hobbyist, and research communities. Because hardware schematics software libraries are freely available through open source licensing, similar to the community of Linux and Linux-based applications, it is inevitable that we will begin to see Arduino-based systems and products in the commercial marketplace. In this experiment, we describe an effort to build a prototype system based on the Arduino platform and purposed with building tools for medical monitoring within home-care environments.

System Platform

The Arduino Platform

The Arduino platform consists of a family of processor-boards differentiated by the processor speed, type, amount of random access memory, and input/output (I/O) signal lines [2, 4]. Completing the hardware development platform is an integrated development environment (IDE) consisting of a C++ like language and software library. Arduino programs are specified using well-defined function entry points that support system initialization, periodic processing, setting and reading of I/O lines, and simple interrupt handling. Arduino processor boards are customized for specific applications through daughter-boards (or shields) that connect by interfacing signal and control lines with the Arduino processor's I/O lines. A growing community of vendors have contributed shields that add diverse capability to the Arduino processor for capabilities such as GSM communication, WiFi communication, motor control, temperature sensing, medical sensing, moisture detection, etc. We have built a prototype that interacts with the Arduino platform in conjunction with an eHealth Sensor available from Libelium Corporation (http://www.cooking-hacks.com) for experimentation with medical monitoring.

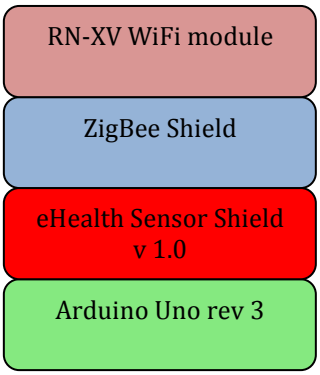

Figure 1. Arduino/eHealth Sensor/WiFi hardware platform

This system consists of an Arduino Uno revision 3 processor board, an eHealth Sensor Shield, a ZigBee wireless communication shield and a Roving Networks RN-XV WiFi module (Figure 1). The Arduino board provides processor, memory, I/O and is programmed through a USB cable attached to a host computer running the Arduino IDE software development platform. The eHealth Sensor shield provides signal lines and interfaces to an array of medical sensor devices including electro cardiogram (ECG), airflow, accelerometer (patient position information), glucometer, pulse-oximeter, and galvanic skin response (moisture/perspiration). The ZigBee shield provides wireless communication using the ZigBee radio standard (IEEE 802.15.4) and the RN-XV module implements WiFi compliant (IEEE 802.11) communication using the ZigBee shield. The eHealth Sensor and ZigBee shields plug into the Arduino's signal lines in a stack (Figure 1). The RN-XV module plugs into a socket on the ZigBee shield.

The software environment consists of an IDE for use in development of programs for the Arduino platform. Associated with each add-on shield is a software library that is added to the IDE software environment in order to extend it with libraries for controlling and interacting with the shield through program implementations. While we selected the credit card-sized Arduino Uno processor board for its price and availability, our intent is to target our application for smaller, more miniaturized Arduino platforms such as the stick of gum-sized Arduino Nano [2]. Because of limited memory resources on our ultimate target platform (Arduino Nano), care was taken in our design to reduce the software memory footprint. To conserve valuable limited program storage space, add-on software libraries employed in our prototype system were limited to the eHealth Sensor library as it was absolutely necessary for reading and controlling the medical sensors attached to the eHealth sensor board. The functions for communication were implemented through the Arduino platform's serial library. We used the Arduino IDE software and libraries contained in version 1.0.5. A program for an Arduino board and shield, or sketch, consists of arbitrary programs that are run from a handful of well-defined execution entry points. These consist of routines resembling standard C++. Arduino programs are executed from three specific entry points and interrupt handlers. The entry points and their execution semantics are

- `void setup()-` called once when Arduino is started, used to initialize your system

  `void loop()-` called periodically as Arduino is running, used to run your system

- `void serialEvent()-` called whenever information is available on the Arduino's UART (serial interface)

- `void interruptHandler()-` function name of your choosing whose execution can be programmatically with a signal input rising or falling

The Arduino Uno board employs a universal asynchronous receiver/transmitter (UART) circuit for I/O connections. The UART is used to get data from the Arduino's attached peripherals (sensors) off the board communicated to the outside world as well as getting data communicated from the outside world (WiFi, USB, or serial port) onto the board.

Software Framework

We set out for a goal of "pluggable connectivity" where component services interact with each other for the formation of a pipeline of components that measure, store, process, and disseminate monitoring data (Figure 2) [3, 6]. This approach gave flexibility in introducing new capabilities as well as separation of concerns for distributed application development, including sensing, storage, processing, and dissemination, in a scalable approach.

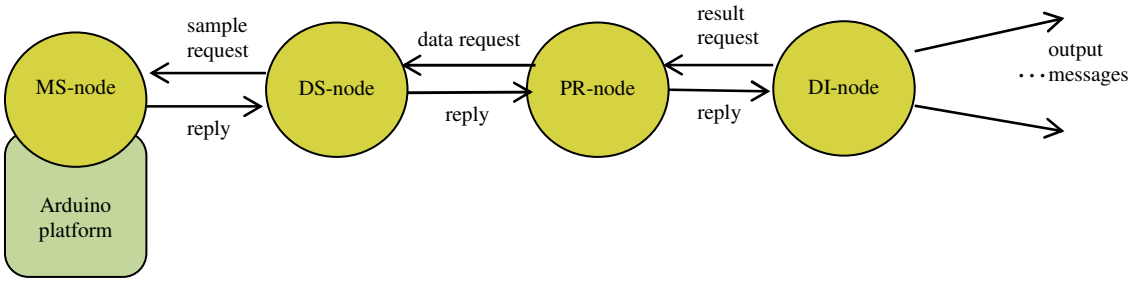

Figure 2. Pluggable service pipeline

The components take form as services that exchange messages in a type of pipelined architecture. The services (or nodes) include capabilities for measurement or sensing, storage, processing, and dissemination. Because nodes in the pipeline implement both client (request initiation) and service (request servicing) roles, the system is best described as a peer-to-peer distributed processing pipeline. The measurement service (or MS-node) consists of a Web service running on an instance of the Arduino/eHealth/WiFi (Figure 1) platform hardware. We have designed and implemented the MS-node as a Web server. The choice of implementation as a Web server facilitated debugging using a standard Web browser. The

Web server accepts incoming HTTP requests, parses and executes the request, and returns an appropriate response. The other nodes in the pipeline are implemented in, but not limited to, the Java programming language.

In order to minimize on the run-time memory footprint, the MS-node server internals were organized as a finite state machine (FSM) that directly accesses the Arduino's UART serial interface rather than relying on communication libraries (such as the WiFly library). When issued an HTTP request, the Web service returns an HTTP response containing sensor readings for each of the attached medical sensors. The added benefit of supporting HTTP in the MS-node is its accessibility from any standard Web browser or client. The storage process consists of a DataStorage peer that retrieves measurements periodically from the MS-node. The DataStorage peer (or DS-node) is configured at run-time in our prototype by specifying the MS-node IP address on the command line at run-time. The processing peer (or PR-node) inputs data samples from the DS-node, runs a processing computation on the data, and produces a result. In our example prototype, the PR-node currently implements simple average. Future versions of the PR-node will consist of algorithms for predictive modeling. Finally the dissemination peer (DI-node) inputs results from the PR-node and delivers them to users. The PR-node in our current prototype implementation outputs information to the console. Future versions of the PR-node will send outputs through email, SMS alerts, or social media. We describe the measurement node in greater detail.

The MS-node is implemented using the subset of C++ supported by the Arduino 1.05 programming environment while the DS-node, PR-node, and DI-node are implemented in the Java programming language. Each node type that accepts incoming requests (server role) implements a standard TCP/IP server. Each node type that initiates requests (client role) implements a standard TCP/IP client. Default values for TCP/IP connections in Java were used for the DS-node, PR-node, and DI-node. For the MS-node implementation, we made use of the Arduino's server-side TCP/IP idle timeout for connection management (expanded upon in Experiments section). The prototype implementations perform the following high-level actions:

MS-node:
1. measurement request
2. obtain readings from medical sensors attached to Arduino
3. compose response
4. accept return response

DS-node:
1. initiate request to MS-node for sensor reading
2. parse HTTP response and extract sensor measurements
3. store record in memory
4. respond to data requests

PR-node:
1. initiate request to DS-node for historical measurements (default of 10)
2. parse response
3. compute average
4. store computed result (current average) in memory
5. respond to processed data requests

DI-node:
1. initiate request to PR-node for computed result
2. parse response
3. print computed result to console

Measurement Node

The MS-node implementation takes form of a finite state machine that parses incoming requests arriving over the UART/serial interface of the Arduino platform. In our current prototype implementation, the FSM implements a parser that extracts the filename from an HTTP get request [1]. Currently supported is a request for the index.html file. In response to this request, the MS-node composes an HTTP response message containing a webpage (HTML text) with readings from all sensors attached to the Arduino platform. Our future implementations will support requests for individual webpages corresponding to the individual sensors. These will include the non-invasive medical sensors only, as we have received institutional review board approval to run human subject trials for non-invasive experiments. Our noninvasive medical sensors include a pulse oximeter (spo2.html), temperature sensor (temp.html), electro cardiogram (ecg.html), blood pressure (bp.html), airflow (air.html), and accelerometer/position (pos.html). The grammar describing the HTTP request implemented by our FSM parser follows:

```
HTTPRequest → Operation HeaderLine "\r\n"

Operation→ OP Resource Version "\r\n"
OP→ "GET"
Resource→ <file-path>
Version→ "HTTP/1.1"

HeaderLine→Name ":" Val "\r\n"|Name ":" Val "\r\n" HeaderLine

Name→ <valid-header-type-string>

Val→ <valid-header-value-string>
```

The FSM parser is implemented as a standard left associative linear recursive parser with 1-token look ahead [7].

Prototype and Experiments and Experiences

We tested the MS-node on a standard WiFi network that was open (no authentication) for ease of connectivity. We also tested the MS-node using authenticated access (WEP and WPA). A pipeline of processing from MS-node to DS-node to PR-node to DI-node was made as through a series of TCP/IP connections. The focus of this experiment was to build the basic plumbing for the distributed framework (nodes) where the majority of the effort was focused on more complete implementation of the MS-node. This choice was made because the MS-node involves hardware-software codesign.

Configuring the WiFi connectivity was performed by connecting to the Arduino platform using a terminal over the serial port and entering configuration commands. Because our Arduino to host interface was USB, we installed software that maps USB devices to serial ports. This allowed us to use unmodified terminal emulation software to connect to the Arduino. An important detail uncovered through substantial reverse engineering and trial and error not described among vendor documentation for the ZigBee shield is that, in order to connect to or upload programs, the shield must be set to "USB mode." When in USB mode, the interface between the ZigBee shield and the Arduino UART/serial is redirected to the USB port mapped to the serial port of the attached host computer for downloading programs to the Arduino. What this means is that, while programming and testing the system, inputs to the Arduino's serial interface will come from the host computer's serial-mapped USB port. Likewise, outputs from the Arduino's serial interface will be sent to the host computer's serial-mapped USB port. Care was exercised in ensuring the ZigBee Shield's jumper connectors were set such that the board was placed in "USB mode." Once in USB mode, from a terminal emulation program, we configured the network parameters of the ZigBee shield/RN-XV WiFi hardware. This was accomplished through the following commands (text following // are comments):

```
$$$                     //put the ZigBee shield/RN-XV WiFi into command mode
set ip protocol 1\r     //use TCP/IP protocol allow incoming/outgoing connections
set ip localport 80\r   //local TCP port is 80 (for HTTP)
set wlan ssid Robots\r  //set network name to experimental network
set wlan auth 0\r       //no authentication
set wlan join 1\r       //join automatically after power up
set comm idle 1\r       //close TCP connection server side after 1-sec idle
set comm remote 0\r     //turnoff automatic greeting msg for TCP connection
join Robots\r           //join network now
show net\r              //display networking configuration parameter values
save config\r           //save settings to filename "config" to be read at boot
exit\r                  //exit command mode
```

Another important undocumented point is that once you have configured the communications shields (ZigBee/RN-XV), they must be removed before loading the Web service program to the Arduino. Failure to do so will cause the uploaded Web service program to overwrite any ZigBee specific configurations if there are any programmatic WiFi commands in an individual Arduino sketch. At the end of any TCP/IP connection between a client and the server, it is important to close the server-side connection. There was a serious issue

concerning closing the TCP connection on the server. To do so required programmatically switching the Arduino to command mode by sending a "$$$" control character sequence without a <CR-LF> at the end of the string. Once in command mode, the string "close\r" with a single <CR> at the end of the string closes the connection and then and "exit\r" command must be given to programmatically exit command mode on the Arduino. The problem stems from the method by which Arduino uses its UART/serial interface.

When running as a Web server, the ZigBee/RN-XV, is also (similar to the board's USB port that to the host) interfaced to the Arduino through its UART/serial interface. This means that incoming data over WiFi passes through the UART and to the serial interface. Reading the serial interface on the Arduino accesses this data. Responses from the Arduino communicated to the outside world is sent through its serial interface to the UART on to the ZigBee/RN-XV and out over WiFi. The problem with switching to command mode, closing the TCP server-side connection, and exiting command mode programmatically is that these are achieved by a special print command, namely `Serial.println()`, which binds formatted outputs to the serial interface. The same print function results in data being sent through the serial interface. The resulting issue is that the control commands appear in the server's output at the end of the HTTP response message (Figure 4). Because these commands don't belong in the formal definition for a valid HTTP response, the client/ browser on the other end of the TCP Web-connection interprets them as an ill-formed HTTP response. Depending on the browser type or client, the response is interpreted as invalid. We developed and tested a method for closing server-side TCP connections that do not have this problem.

We use the TCP idle timer as the method for closing server-side connections instead of programmatically closing it using the ZigBee/RN-XV's command mode (the method published in technical documentation is to use command mode). We tested this approach and found it to be very reliable method for closing server-side connections (Figure 5).

Because inbound data arrives through a serial interface, it was important to store inbound messages on the server in a buffer before operating on it with the parser. The `serialEvent()` routine is a built-in interrupt handler for the Arduino platform. It is triggered whenever data is available to the serial interface via the Arduino's UART. We use this trigger to fill a buffer with the character data for the incoming HTTP request. When attached to the ZigBee/RV-XN/WiFi shield, incoming data over the WiFi network reaches the Arduino processor through the UART. From the UART, data is stored in buffers accessible using the Arduino's serial interface.

Programmatically, HTTP responses are implemented by sending data through the Arduino's serial interface. Such outbound data then passes through the UART to the ZigBee/RV-XN/WiFi shield and out to the requesting client as an HTTP response over the network. We tested browser interaction with the MS-node using both Safari on MacOS X, Firefox on MacOS X, and Firefox on Ubuntu Linux 12.04 LTS. Wireless connectivity (WiFi) tested using a Cisco Aironet 1142 router operating in open mode (no authentication) and an ActionTech MI424 operating in open mode and authenticated (WEP 128) mode.

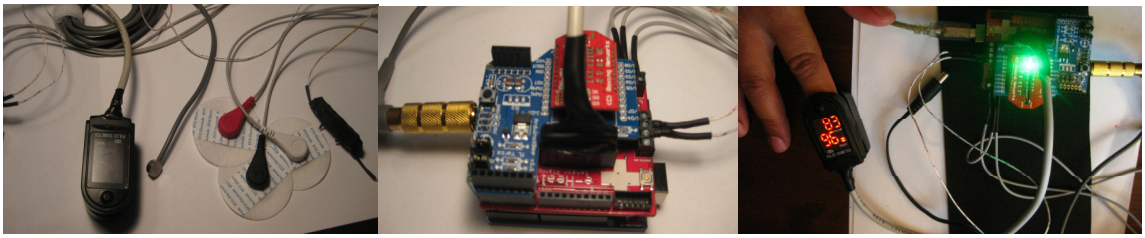

Figure 3. SP02/ECG/temp/airflow sensor, Arduino platform, system operation

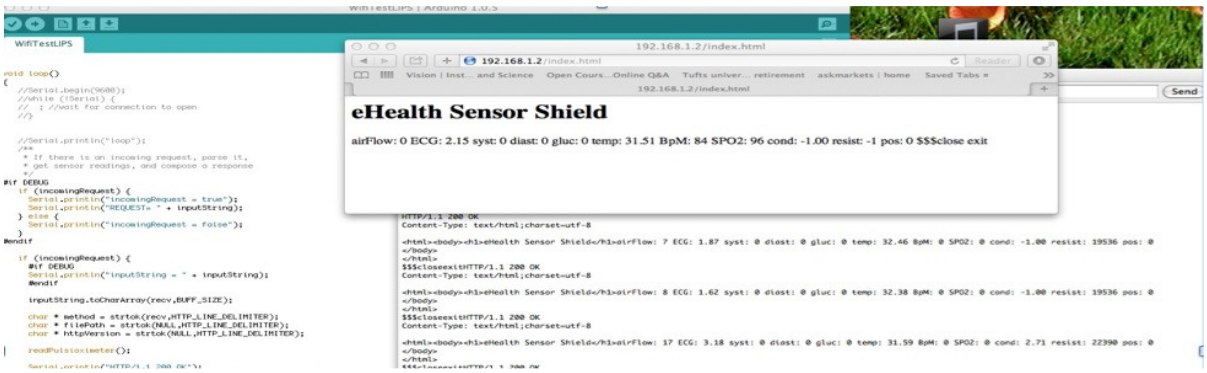

Figure 4. Closing TCP connection via command mode pollutes HTTP response

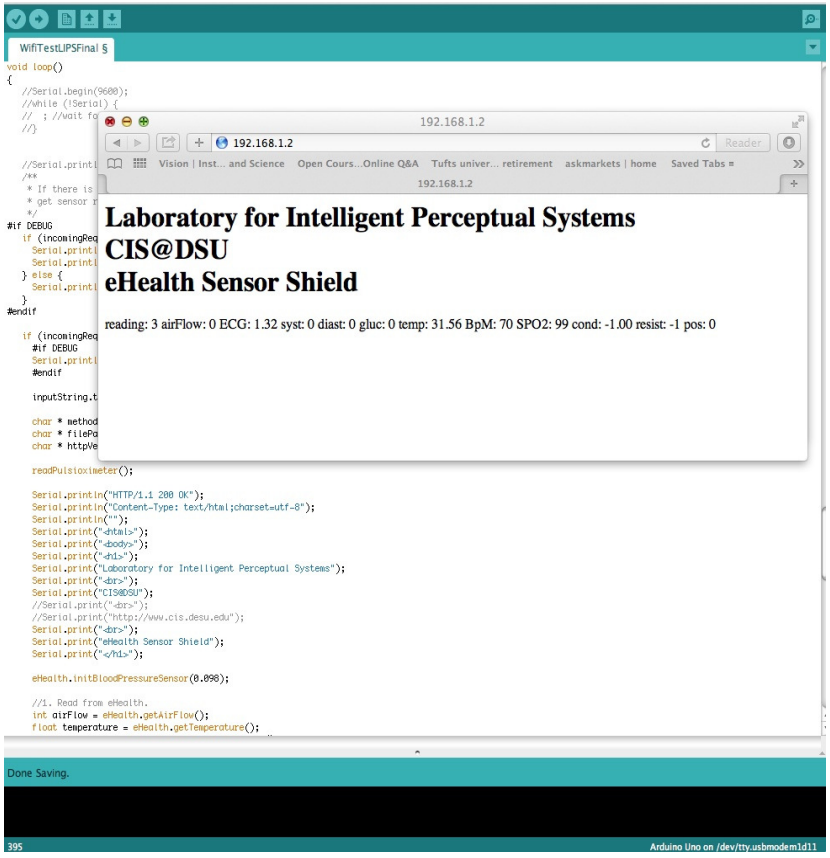

Figure 5. Using TCP idle timer reliably closes server-side TCP connection

Discussion

We have described a prototype implementation and the successful solution to a number of important undocumented practical issues for the Arduino/eHealth Sensor/ZigBee RN-XV hardware stack that participates within a larger distributed framework intended for medical monitoring for home-care environments. We have designed and implemented a prototype Web service measurement node (MS-node) that responds to HTTP GET request and responds with properly formatted HTTP responses as well as simple DS-node, PR-node, and DI-node peers that implement a peer-to-peer distributed processing pipeline. The HTTP response of the MS-node contains information sampled from health sensors interfaced to the Arduino platform. Many months have been invested investigating, reverse engineering, and hacking the platform in order to resolve issues largely undocumented in the vendor's technical documentation as well as the worldwide Arduino development community. Particularly, we have addressed critical issues of how to get the platform to connect automatically to the network.

The most reliable approach was to manually configure the platform, store the configuration, and retrieve and apply it under program control. We found that a pure programmatic solution to configuring the hardware to attach to the network was not reliable. This stemmed from timing issues concerning contacting a local WiFi access point, the electrical noise in the environment around the system, and a relatively low power antenna on the ZigBee module. We also found that using command mode to close TCP connections (the only documented method in the vendor documentation) polluted the server's HTTP response message with the command string sequences. We found the TCP idle timer as the perfect method for reliably silently closing the server-side TCP connection.

Future versions of the distributed framework will include implementation of more sophisticated processing capability for the PR-node including a library for linear interpolation, Gaussian processes, and pattern recognition for streaming data. Moreover, the storage node (DS-node) in future implementations will include database-backed persistence. Finally, the dissemination node (DI-node) in future implementations will include targeted wide-area communication through mechanisms such as integration with the Twitter APIs.

The Arduino platform is relatively stable and in the future we will add features for retrieving individual sensor readings by implementing access to multiple server file system objects, one for each sensor. The platform is a promising system that participates easily within our distributed pipeline framework. Further information about the project may be found among the webpages for the Laboratory for Intelligent Perceptual Systems at Delaware State University (http://www.cis.desu.edu).

Acknowledgments

The authors acknowledge the funding agents. This research was supported by the National Science Foundation (Award #s CNS-1205426, HRD-1242067). Support for the undergraduate researcher working on this project is provided by the National Science Foundation (Award # HRD-928404).

References

[1] Kurose, J., & Ross, K. (2012), *Computer Networking A Top-Down Approach*. (6th ed.). Upper Saddle River, NJ: Pearson.

[2] Blum, J. (2013). *Exploring Arduino: Tools and Techniques for Engineering Wizzardry*. New York: Wiley.

[3] Holness, G., Karuppiah, D., Uppala, S., Ravela, S., & Grupen, R. (2001). A Service Paradigm for Reconfigurable Agents. *Proceedings of 2nd Workshop on Infrastructure for Agents, MAS, and Scalable MAS*.

[4] Timmis, H. (2011). *Practical Arduino Engineering*. New York: Apress.

[5] Noshadi, H., Dabiri, F., Meguerdichian, S., Potkonjak, M., & Sarrafzadeh, M. (2013, April 12). Behavior-Oriented Data Resource Management in Medical Sensing Systems. *ACM Transactions on Sensor Networks*, *9*(2), 1-26.

[6] Vasseur, J.-P., & Dunkels, A. (2010). *Interconnecting Smart Objects with IP: The Next Internet*. Burlington, MA: Morgan Kaufman.

[7] Aho, A., Lam, M. S., Sethi, R., & Ullman, J. D. (2066). *Compilers:Principles, Techniques, and Tools*. (2nd ed.). Reading, MA: Addison-Wesley.

[8] National Institute on Aging, NIH. (2010). *There's No Place Like Home—For Growing Old*. Retrieved from http://www.nia.nih.gov/health/publication/theres-no-place-home-growing-old

[9] Ko,, G., Lu, C., Srivastava, M. B., Stankovic, J. A., Terzis, A., & Welsh, M. (2010, November). Wireless Sensor Networks for In-Home Healthcare: Potential Challenge. *Proceedings of the IEEE*, 1948-1957.

Biographies

GARY HOLNESS is an assistant professor and directs the Laboratory for Intelligent Perceptual Systems. His research expertise and experience rests at the intersection of machine learning, machine perception, distributed systems, and statistics. His research group investigates algorithms that endow systems with the ability to discern and act upon the content of their environment (learn more at http://www.cis.desu.edu).

LEON HUNTER is an undergraduate student at Delaware State University majoring in computer science. His research interest includes computer networking, sensor systems, and mobile devices.