Poster: Enabling Computational Jewelry for mHealth Applications

Andres Molina-Markham
Dartmouth College

Ronald A. Peterson
Dartmouth College

Joseph Skinner
Dartmouth College

Ryan J. Halter
Dartmouth College

Jacob Sorber
Clemson University

David Kotz
Dartmouth College

Many of the most compelling mHealth applications are designed to enable long-term health monitoring for outpatients with chronic medical conditions, for individuals seeking to change behavior, for physicians seeking to quantify and detect behavioral aberrations for early diagnosis, for home-care providers needing to track movements of elders under their care in order to respond quickly to emergencies, or for athletes monitoring their physiology to improve performance. Developing BAHN applications that require consistent presence and strong security, without depending on a smartphone or without building lots of computation/communication resources into every BAHN device presents a critical challenge for the wide-spread adoption of mHealth technologies. The smartphone is not always with its user [1]: many people set aside their phone while at home or while driving, exercising, or bathing. According to a Pew study, a third of smartphones have been lost or stolen [2]! When the smartphone is not present, the BAHN could lose its foundation; valuable data could be lost, critical events may go unrecognized. Second, smartphones have limited means to authenticate or identify the person holding them; if the phone has been lost or stolen, an app could inappropriately disclose personal health information about the phone's owner. Third, smartphones are general-purpose devices, not dedicated to health-related applications; it is thus more difficult to evaluate the safety and security of a system when it is sharing resources with other applications.

For these reasons, we are developing wearable devices as the foundation for a consistently present and highly available body-area mHealth network. Our vision is that a small device, such as a bracelet or pendant, will provide the availability and reliability properties essential for successful body-area mHealth networks. We call this class of device *computational jewelry*, and expect it will be the next frontier of mobile systems.

MobiSys'14, June 16–19, 2014, Bretton Woods, New Hampshire, USA.
ACM 978-1-4503-2793-0/14/06.
http://dx.doi.org/10.1145/2594368.2601454.

We prototyped our first piece of computational jewelry, which we call *amulet*, to enable our previously proposed vision [3]. It runs applications that may collect sensor data from built-in sensors or from other devices, analyze and log the data, queue information for later upload, and interact with the wearer. Independent developers can develop applications that can be vetted and installed on an amulet.

We achieve our goal of allowing applications to run on a small ultra-low-power device by means of (1) a multiprocessor hardware architecture and (2) an event-driven software architecture that allows applications to survive routine processor shutdowns. (1) We use a two-processor architecture: an application processor capable of performing computationally intensive tasks and a coprocessor that manages radio communications and internal sensors (in our prototype: accelerometer, gyroscope, magnetometer, temperature sensor, light sensor, and microphone). To save power the application processor is powered off most of the time, while the coprocessor handles all real-time device interactions. Minutes or hours of sensor data and other messages from the coprocessor are staged in queues, stored in shared Ferroelectric RAM, until it is necessary to boot the application processor. (2) Most mHealth applications are reactive, running only when an event of interest occurs. Our architecture provides a state-machine event-driven programming model. Programmers define an application as a finite-state machine and a set of functions to handle events of interest. Our architecture allows applications to identify computational state that should be retained between events. Explicitly managing program state (rather than implicitly managing state in a thread's run-time stack) allows our run-time system to efficiently save application state to persistent memory and power down the main processor, with no harm to applications. This approach dramatically extends battery life, increasing overall application availability.

Prototype Implementation

Key goals of our previously proposed vision [3] were to create a system that is consistently present and highly available, while providing a flexible application programming model. To achieve consistent presence, an amulet must be able to be implemented in a small package so it will have a wearable form factor. Therefore, the battery must be small: e.g., a typical 150 mAh battery requires only 1.71 cm^3. To achieve high availability, despite a tiny battery, an amulet must be extremely power efficient so it can function for sev-

eral days. To achieve independence from the smartphone, for interesting applications, an amulet must be capable of performing digital signal processing, cryptographic operations, and, sensor-data classification. Thus, our approach involves a dual-processor design: an ultra-low-power processor tends to communications and sensing (I/O coprocessor), and a more-capable application processor runs applications. Although this approach is not uncommon, our hardware-software architecture is specifically designed to shut down the application processor – not just put it to sleep – most of the time. This approach drives another requirement: the application processor (and its OS) must transition quickly from off mode to active mode. Our approach drives two requirements: first, applications must be able to survive routine system reboots; second, the application processor must be able to return from off state to active state, load its operating system, and reload an application extremely fast.[1] Our architecture achieves these requirements by providing an event-driven programming model. This approach works well for mHealth applications, many of which are idle most of the time, waiting for something to happen.

Amulet applications are defined as finite-state machines (FSMs) with memory. Thus, each application is defined as a set of *states*, a small set of *variables*, and a set of *event handlers* by which the application responds to events of interest. When an event is delivered to the application, the system calls the appropriate handler function and transitions the application to the designated next state. The non-blocking handler is a function that may consume data arriving with the event, update application variable(s), or send events to system services (or to itself), in any combination. The application subscribes to certain events when it is initialized, and can add or adjust subscriptions as part of the action in an event handler. This approach makes application state explicit; because handlers run to completion,[2] there are no threads with stack-based state information to preserve between events, let alone across processor reboots. Application code and variables are kept in persistent storage, as is a record of the current state of each application; thus, when the event queue becomes empty, the application processor simply shuts off.

The application processor remains off until the I/O coprocessor wakes it up. On request, the I/O coprocessor produces messages to carry the output from internal sensors (e.g., temperature, accelerometer), an input from the wearer (e.g., a button press), or the reception of a network message from an external mHealth device. The coprocessor inserts each new message into the interprocessor message queue; the memory controller uses a set of queue-management policies to determine when to wake the application processor (examples include the insertion of a high-priority message or the queue being near full). While the application processor is awake, it draws messages from the interprocessor queue and copies it into a new event message inserted into its internal event queue.

We implemented the following OS services and managers in C: Sensor Manager, Actuator Manager, Storage Manager, Network Manager, Interprocessor Communication Manager and Authorization Manager. Each consist of two parts, an event-driven application – which we implemented in a similar way to applications – and a set of routines that have access to lower level drivers. The event-driven part allows these managers to receive and process event messages. The set of routines with low level access are not available to applications. We implemented the I/O coprocessor as a single finite state machine with timers and hardware interrupts. We also implemented three devices to simulate a heart-rate monitor, a galvanic skin response sensor, and a nicotine sensor that communicate via ANT with our prototype. The three devices are development boards that use the same ANT SoC as in our prototype (Nordic Semi nRF51422). We implemented the message queue using an ultra-low-power MCU with FRAM (MSP430). FRAM offers persistent storage without power, and it is one hundred times faster than flash. On writes FRAM uses 250 times less power than flash ($\approx \mu A$ at 12kB/s). The FRAM MCU buffers data until one of two conditions are met: (i) the FRAM is full, or (ii) the I/O coprocessor observes an event that must wake up the application processor. This may happen when an application subscribes to a particular data value from a sensor.

Our first wearable prototype is 3.7×3.5 cm. We optimized it for development and not yet for size. Our programming model allows multiple independently-developed applications to run simultaneously and efficiently. Our current prototype provides 14 to 26 hours of battery life with a 150mAh battery when running two mHealth applications (for stress management and smoking cessation) with realistic workloads. For news about the Amulet project, visit amulet-project.org and 'follow' our blog.

[1] In the applications we consider, an application may execute a task every few seconds and the task may only require a few milliseconds to complete.

[2] A supervisor enforces a timeout to prevent handlers from blocking or running too long.

Acknowledgements

We thank Kevin Freeman, Bhargav Golla, Emily Greene, Hilary Johnson, Adam Labrie, Tim Pierson, and Tianlong Yun for their participation in this project. This research results from a research program at the Institute for Security, Technology, and Society, supported by the National Science Foundation under award numbers CNS-1314281, CNS-1314342, and TC-0910842, and by the Department of Health and Human Services (SHARP program) under award number 90TR0003-01. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsors.

1. REFERENCES

[1] A. K. Dey, K. Wac, D. Ferreira, K. Tassini, J. H. Hong, and J. Ramos. Getting closer: an empirical investigation of the proximity of user to their smart phones. In *Proceedings of the International Conference on Ubiquitous Computing (UbiComp)*, pages 163–172, Sept. 2011. DOI 10.1145/2030112.2030135.

[2] K. Hill. Sorry, smartphone owners, but you're more likely to have your privacy invaded. Forbes, Sept. 2012. Online at http://tinyurl.com/hill-smartphone.

[3] J. Sorber, M. Shin, R. Peterson, C. Cornelius, S. Mare, A. Prasad, Z. Marois, E. Smithayer, and D. Kotz. An Amulet for trustworthy wearable mHealth. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (HotMobile)*, Feb. 2012. DOI 10.1145/2162081.2162092.