Opening Pervasive Computing to the Masses Using the SEAP Middleware

Seth Holloway, Drew Stovall, Jorge Lara-Garduno, and Christine Julien
Mobile and Pervasive Computing Group
The University of Texas at Austin
{sethh, dstovall, jlara-garduno, c.julien}@mail.utexas.edu

***Abstract*— The increasing availability of sensing devices has made the possibility of context-aware pervasive computing applications real. However, constructing this software requires extensive knowledge about the devices and specialized programming languages for interacting with them. While the nature of pervasive computing lends users to demand individualized applications, complexities render programming embedded devices unapproachable. In this paper we introduce the SEAP (Sensor Enablement for the Average Programmer) middleware which applies existing technologies developed for web programming to the task of collecting and using sensor data. We show how this approach can be used to create new applications and to update existing web applications to accept sensor data.**

I. Introduction

George, an undergraduate in Computer Science, wants to develop an aware apartment. To start, he would like to automatically turn on lights whenever he is home. George understands the logic perfectly, but he does not know how to use the necessary pervasive computing elements. George has found inexpensive, single-purpose solutions (e.g., a motion-sensitive light switch) or expensive, complex, multi-purpose solutions (e.g., the X10 Ultimate System). George wants a cheap, multi-purpose solution with low complexity.

While George may be fictional, this scenario is real. The potential for pervasive computing research to impact the world is real. Application scenarios are not limited to homes; developing an expressive application for any environment with embedded sensing, computation, and actuation will encounter the same challenges. This complexity is simply too high for average programmers, particularly those looking to create custom applications to enhance their own lives.

Making pervasive computing devices availabile dramatically increases the number of achievable applications, enabling all manner of context-awareness. Unfortunately, the reality remains that average programmers with the ideas and interest to create such applications can not overcome the steep learning curve that currently exists for programming these devices.

Recent sensor integration techniques have improved device interactions but still require significant configuration and programming. In addition, because of the complexity of programming, existing approaches are stovepipe solutions that solve a single problem independent of other devices already deployed on the site. Current approaches fuse the entire application stack, creating significant but unnecessary interdependencies; for example, George cannot reuse the motion sensor from his aware home after it is tied to lighting one area. To enable programmability of pervasive computing applications, we propose a paradigm shift that empowers the *average programmer* rather than highly specialized device programmers.

Our approach, the *Sensor Enablement for the Average Programmer* (SEAP) middleware, makes pervasive computing applications easier to develop. SEAP provides a middleware layer between the developer and the customized hardware, publishing sensor and actuation data using the standard HTTP protocol. The SEAP approach can be used to support the development of entirely new pervasive computing applications or to integrate existing applications into a pervasive computing environment. Our approach takes advantage of the wealth of experience the average programmer has in web programming. With the growth of demand for new consumer applications in the pervasive computing realm, we anticipate that given appropriate development aids, the adoption of pervasive computing will be similar to the growth of the Internet.

In this paper we describe the SEAP middleware, its uses, and qualitative results from our initial prototypes. We start by overviewing related work. We then detail the SEAP architecture and the process by which SEAP can be used to implement applications. Finally, we mention extensions to and applications of SEAP and conclude.

II. Related Work

The notion of sensor and web integration is not entirely new, and several related projects have paved the way for our proposed approach. CoolTown allows networked mobile devices to publish data on the web through a variety of tailored protocols [1]. Data being published includes information about a device's characteristics (location, for example), enabling a degree of content-based discovery. Another project created a centralized website that accepts sensor data generated worldwide [2], a technique cleverly titled *slog* (sensor log). Sensors communicate their data to a base station that funnels sensor data to the clearinghouse. Because we are interested in enabling more personal applications, in SEAP, users control their own data. This provides a more distributed computing approach since data and actuation events are only shared relative to a local space.

Other current approaches to sharing sensor data build on standard web services using SOAP, WSDL, and XML. The

Open Geospatial Consortium (OGC) allows access to sensors using SensorML, a sensor-specific language that defines an XML schema to use sensors [3]. Microsoft's SenseWeb project [4] also provided a generic method to push sensor data online. Both approaches rely on SOAP web services, which can be inflexible, slow, hard to maintain and manage, and heavyweight [5]. SOAP web services present an unnecessary cost for small deployments that we set out to alleviate. This is especially relevant to pervasive computing deployments where resource-constrained devices often demand efficient, streamlined solutions.

While rooted in different technologies, there are a number of other designs to reduce the efforts required to develop pervasive computing applications. For example, Weis et al. [6] use visual programming techniques to reduce the learning curve typically required. Other approaches [7], [8] provide additional layers of abstraction to manage complexities that can be hidden from the developer. We anticipate that these techniques will be complementary to the SEAP middleware architecture and might be combined to further ease software development for pervasive computing.

With SEAP we minimize the interface for both devices and programmers by relying on a simple but expressive form for data movement, HTTP GET and POST commands. Our approach is consistent with representational state transfer (REST) principles [9] in an effort to be lightweight, flexible, and compatible. REST is an architectural style that promotes the transmission of domain-specific data over HTTP. Users interact with resources using a small set of well-defined commands to manipulate the resource. Some work has been done to apply REST to pervasive computing [10], however, this work violates many REST principles with complex systems that require a great deal of configuration and knowledge; this reduces benefits innate in the initial REST proposal. SEAP, on the other hand, approaches the problem with a minimalist perspective: get the data online in a form that entry level web programmers can already use. By using standard HTTP, we inherit the benefits of both past and future work on HTTP and allow programmers to begin writing ubiquitous applications immediately.

III. SEAP Architecture

In this section, we describe the SEAP approach in detail. Specifically, we present the necessarily simple software architecture that underlies the sensing, actuation, communication, and interaction capabilities.

By relying on well-established programming standards, SEAP brings the seemingly unapproachable task of programming pervasive computing applications into the hands of domain programmers who are experts in their applications' requirements. SEAP hides complexities associated with data collection and actuator command with familiar web programming patterns, using lightweight software components deployed on resource-constrained devices to manage the distributed coordination tasks. Through SEAP's abstractions, an individualized application can tailor device and network configurations to a particular task by parameterizing the software running remotely. At the same time, participating remote devices can use standard posting procedures to exchange sensor data and actuation commands in simple formats.

We divide our description of the SEAP architecture into two aspects: the behavior of devices participating in a SEAP supported application, and the application server that hosts the applications.

A. SEAP Architecture: Devices

As SEAP aims to be generally applicable to a wide variety of pervasive computing applications, a goal of the SEAP architecture is therefore to ensure that the functionality required for these devices is kept to a minimum. To this end, devices are not required to accept arbitrary inbound connections; instead each device controls its own communication costs through the outbound connections it creates. This includes both the transmission of data requested by an application and the reception of configuration and actuation commands. The SEAP data flow depicted in Fig. 1 provides a high-level view of the relationships between different hardware components in a SEAP system.

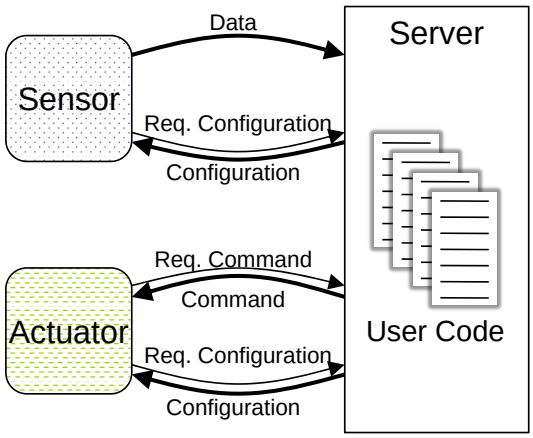

Fig. 1. Data flow in the SEAP architecture.

SEAP participating devices are classified as *sensors* (data producers) and *actuators* (command consumers). While it is possible for a node to perform both sensing and actuation duties simultaneously, this distinction allows clarity in the architectural description.

Sensors. A device that is creating its own data or gathering data from the environment to send to the application is considered a sensor. SEAP enables applications to collect this information from sensors in an intuitive fashion. The SEAP components running on the remote device create an interface to the accepted web-based programming approaches. Specifically, when a sensor has data to send, SEAP packages the data and opens an HTTP connection to a preconfigured Uniform Resource Identifier (URI). The sensor data is encapsulated as parameters in the connection and is transferred to

the application. A basic algorithm to be implemented by the SEAP component on the remote device is shown below.

```
while(true) {
   connect to data-report-uri
   send readings
   disconnect
   sleep data-report-delay
}
```

The connection to the server is successful when communication paths are available (SEAP delegates communication to an underlying network infrastructure). When the central web server receives the connection and its parameters, SEAP deployments use common web application frameworks to handle communication and parse the parameters and present the data to the user's application. Because the user's application appears to be interacting only through web programming constructs, interaction with sensor data is reduced to familiar and routine operations.

Actuators. A device that receives commands from an application to alter the logical or physical environment is considered an actuator. As before, SEAP's goal is to allow applications to pass commands to remote actuators using simple and intuitive techniques. Like sensors, a remote SEAP actuator manages the HTTP connection and invokes native functions. A basic SEAP component for a remote actuator could use the following algorithm.

```
while(true) {
   connect to command-uri
   while (connection open) {
      read command
      apply command
   }
   sleep command-retry-delay
}
```

Here, the *while (connection open)* loop is used to eliminate the resource usage of continuously polling the server for new commands. When the connection is closed explicitly by the host or implicitly due to a link failure, the device will wait before attempting to reestablish the connection. This pause could incur an undesired delay between sequential commands if a SEAP server were only allowed to send a single command per connection. To address this issue, devices read sequences of commands from a single connection as they are sent by the server. Assuming the *read* command above is able to parse the commands as they are delivered, the inner while loop explicitly provides support for batching many commands together. Any sequence of commands returned to the device will be processed in order, without encountering the *sleep* statement.

However, even batching commands together may still not provide the immediate response that some applications desire. If the connection is closed after each batch of commands is processed, the device must still wait for the *command-retry-delay* before receiving a new batch of commands. This concern is easily addressed by allowing the *read* command to block until new commands are received. The server simply distributes commands with arbitrary delays, each of which is read and applied as soon as it is received. When the end-to-end connections are stable (compared to the *command-retry-delay*), this gives the application much finer control over the timing of actuation commands.

Reconfiguration. Both sensors and actuators rely on a small set of pre-configured variables to properly integrate and support a SEAP application. However, while the application is running, it may become necessary to change the values of such a variable on a remote device. For example, an application may request more frequent sensor readings due to an event external to the remote device. To enable these scenarios, devices also host a SEAP component which downloads configuration changes in much the same way that commands are retrieved by an actuator. This allows a developer to programmatically alter any of the variables mentioned in this section, even the reconfiguration URI, at runtime.

In the example sensor and actuator algorithms above, the *uri* and *delay* variables can be reassigned by providing new values on the webserver as a simple properties-style configuration page like the one below.

http://my.host.name/sensor7/config.properties

```
data-report-uri=\
   http://myhost/sensor7/readings
data-report-delay=3
command-uri=\
   http://myhost/sensor7/command.properties
command-retry-delay=10
configuration-uri=\
   http://myhost/sensor7/config.properties
configuration-delay=60
```

In this case the configuration file contains the URIs and delays for a fictitious device "sensor7." The device parses the configuration and updates its internal variables. Once the sensor has applied this configuration, we expect it to post readings every three seconds, and download its configuration every 60 seconds.

Even devices that can be reconfigured still require initial settings; a process we call *bootstrapping*. When a device participating in SEAP is initially put into service for an application, the user configures the device by specifying variables necessary to coordinate with the web application. An example configuration utility, Fig. 2, demonstrates the interaction that is required for this process. Here, a device with a temperature sensor and an LED is configured to publish temperature readings and retrieve commands to control the LED. The nature of this component depends on the particular target device; it's main purpose is to translate between the low-level hardware and protocols and the simple SEAP interface. As the final step of the configuration utility, the SEAP component is loaded on the device (either through a direct connection or through existing over-the-air programming capabilities).

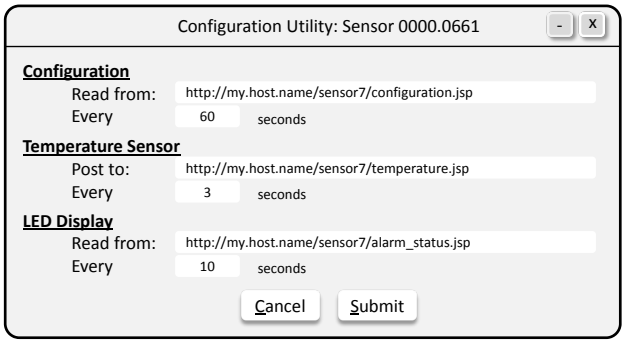

Fig. 2. Example user interface for sensor configuration.

B. SEAP Architecture: Application Server

Because average, hobbyist programmers do not have device-specific knowledge, we shift the application logic to the application server where programmers are more comfortable. By centralizing the logic to an application server we mitigate complexities of coordinating distributed devices. This also allows us to apply existing high-quality tools and techniques to the problem that would not otherwise be available. Because SEAP reduces interactions with both sensors and actuators to a standard web-style interaction paradigm, constructing applications becomes much simpler and more familiar.

A web application framework (e.g., Tomcat, Ruby on Rails, ASP.Net, etc.) simplifies the design of the server by addressing many of the traditional server-programming concerns (e.g., connection management, data-stream parsing, etc.). Many of these frameworks also provide support for advanced features such as load balancing and clustering. By including web application frameworks in the SEAP architecture, we are encouraging the separation of non-application concerns from the user's code base and into a purpose-built tool. Pragmatically, the key advantages of using a web application framework are derived from re-framing misunderstood pervasive computing problems into a standard web applications. This enables developers to use almost any of the popular programming languages and the manuals, tutorials, and guides that are available for them. Developers also benefit from the high-quality tools available for testing, debugging, documentation, and integration for this popular application domain.

By introducing elements that serve as both *device* and *server*, applications can adopt a more hierarchical design. Intermediate nodes could perform data filtering, aggregation, or even pre-processing.

At the other end of the spectrum of pervasive computing applications, a SEAP system that does not require sensors can use simple static content based web servers instead of web application servers. Commands and configurations requested by remote devices could be read directly from the central server's file system. Any web server (Apache httpd for example) would be sufficient for this purpose. Updates to the files made by hand or by other applications would be immediately reflected by the web server and propagate to the devices.

IV. SEAP-Based Implementations

SEAP eases software development by moving the logic of pervasive computing applications to an environment that is accessible and familiar to junior programmers. In this section we detail how this is done in practice for two different applications. One application was developed before the SEAP architecture was conceived and then adapted to it after the fact. The other was developed from scratch using the SEAP patterns to build a robust, flexible, and easily testable application.

A. SEAP Applied to an Existing Application

The *UbiCoffee* application was previously developed by graduate students to monitor coffee levels at several locations in our building. Originally users of the application updated the "amount available" through a web-based interface. Since a web browser was often not available near the coffee pot, most users traveled back to their desk to update the application. Often these trips were interrupted leaving the application unaware of the actual state of the coffee pots.

To increase the quality of the data reported to the application, we wanted to place two buttons near the coffee pots to indicate the addition and removal of coffee. Users would simply press a button labeled "+1" several times when brewing new coffee (8 presses to indicate brewing 8 cups) or press the "-1" button when taking a cup from the pot. To provide this functionality, we selected SunSPOT [11] devices. However, this choice left us with a problem. The server is written with Ruby on Rails, a language used to enable rapid prototyping of web applications, while the SunSPOT is programmed with a variant of JavaME.

To bridge this language divide, we used the SEAP architecture. As a first step, the *UbiCoffee* application was extended to include an "amount changed" form in addition to the existing "amount available" form. This work was completed quickly by the original developers and could be independently tested and verified without any interaction with the SunSPOT.

The second step was programming the SunSPOT to behave as a simple pervasive computing device with "+1" and "-1" buttons. This task was not difficult, but was performed by a student familiar with the nuances of these devices. Now, when the buttons are pressed the device posts the appropriate value to a given web URI as a given parameter.

To link our new pervasive device to our web application, we simply provided the URI of the new *UbiCoffee* page and the parameter name to the device. The resulting system now accepts data from users via the SunSPOT device or web browsers at their discretion.

B. SEAP From Scratch

The *Spot-to-Bot* application was developed by a undergraduate researcher in our lab to steer a Roomba robot using the accelerometer on a SunSPOT. In this case, the SEAP architecture was used to break the complete application into three distinct parts, each independently testable and exchangeable. The resulting application is flexible and easily extended to accept new sensors, actuators, or behaviors.

The first of the three components is a web application with three pages. The first page is a web form that accepts three values: "X", "Y", and "Z". Eventually these values were to be provided by the SunSPOT. In the meantime, these values were provided by submitting the form from a web-browser. The second and third pages of the web application display one value each; representing the requested speed and turning angle for the robot. The web application thus contains the logic to translate the raw data provided by the sensor into the commands given to the actuator.

The second component of the system delivers accelerometer data from the SunSPOT device to the web server. This component is a simple extension of the program developed for the *UbiCoffee* application described in the previous section. The third system component drives the Roomba robot. Using existing code as a basis, the majority of this work fell to retrieving and parsing values provided by the web server.

The resulting application uses three components which can be independently tested and verified. In fact, the three components can even use different programming languages, each specifically suited to the component's primary task (e.g., JavaME for the SunSPOT, PHP for the server, C++ for the Roomba controller).

V. Future Work

We see several extensions to, and applications of, the SEAP middleware. We are investigating the use of this technology in other pervasive computing scenarios; in particular, we see an opportunity to apply the SEAP approach in the absence of infrastructure—SEAP is particularly well-suited to pervasive networks. Although HTTP is normally run over TCP/IP, the specification is compatible with any networking stack, and there are lightweight web servers capable of running on handheld and other resource-constrained devices. We plan to explore the ability to quickly deploy a sensor network for environmental monitoring in a disaster-relief circumstance.

We would like to extend this architecture to other platforms and create tutorials. Ultimately, we will publish prototypes for end-user customization that allow novice programmers to create exciting, useful pervasive computing applications.

Perhaps the most ambitious ideas are blog widgets and an SMS-proxy. Ready-to-use sensor data provided via blog integration and widgets to be used in mash-ups or composite applications. The Internet experience will be more personal by adding sensor-provided context information. The SMS interface would allow access to your devices from all over the world; for example, a user could send a text message from home that would start brewing coffee at work.

VI. Conclusion

In this paper we present the Sensor Enablement for the Average Programmer (SEAP) middleware, which lowers the barrier to entry for programming pervasive computing applications. While we could have developed a new, heavyweight framework aimed at making ubiquitous computing more accessible, we instead focused on a broad solution using existing technologies where possible. As a result, the SEAP middleware allows people to interact with sensors and actuators without learning new languages or procedures. We also described two existing SEAP deployments. The `UbiCoffee` application shows how SEAP was used to add sensors to an existing web-based application while the `Spot-to-Bot` application shows the benefits of using SEAP as an architectural pattern for system development.

The scalability and accessibility of the Internet make the web an ideal platform for increasing the number of active pervasive computing applications. Initially, only large organizations had a web presence. However, the Internet age truly burgeoned once individuals could create and maintain web pages. To achieve the same growth in pervasive computing, sensors and actuators should be easily accessible through a suite of abstractions usable by the average programmer. The SEAP middleware achieves this, allowing programmers to easily integrate sensors and actuators without any awareness of the specific low-level languages and protocols.

SEAP takes advantage of the vast body of work on web programming to provide an approach that is easy-to-understand, easy-to-use, language-agnostic, robust, reusable, and immediately achievable. People are ready for pervasive computing applications; we provide an accessible method to enable multi-device developments. SEAP is that method.

VII. Acknowledgments

The authors would like to thank the Center for Excellence in Distributed Global Environments for providing research facilities. This work was funded, in part, by the National Science Foundation (NSF), Grant # CNS-0620245. The views and conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] J. Barton, T. Kindberg, H. Dai, and N. Priyantha, "Sensor-enhanced mobile web clients: an XForms approach," *WWW*, pp. 80–89, 2003.

[2] K. Chang, N. Yau, M. Hansen, and D. Estrin, "SensorBase.org-A Centralized Repository to Slog Sensor Network Data," *DCOSS/EAWMS*, 2006.

[3] M. Botts, "Sensorml," http://vast.uah.edu, 2007.

[4] A. Santanche, S. Nath, J. Liu, B. Priyantha, and F. Zhao, "SenseWeb: Browsing the Physical World in Real Time," *Demo Abstract, IPSN*, 2006.

[5] C. Kohlhoff and R. Steele, "Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems," *WWW*, pp. 03–2002, 2003.

[6] T. Weis, M. Knoll, A. Ulbrich, G. Muhl, and A. Brandle, "Rapid prototyping for pervasive applications," *IEEE Pervasive Computing*, vol. 6, no. 2, pp. 76–84, April–June 2007.

[7] R. Handorean, J. Payton, C. Julien, and G.-C. Roman, "Coordination middleware supporting rapid deployment of ad hoc mobile systems," in *MCM*, May 2003, pp. 362–368.

[8] F. J. Ballesteros, E. Soriano, G. Guardiola, and K. Leal, "Plan B: Using files instead of middleware abstractions," *IEEE Pervasive Computing*, vol. 6, no. 3, pp. 58–65, July-September 2007.

[9] R. Fielding and R. Taylor, "Principled design of the modern Web architecture," *TOIT*, vol. 2, no. 2, pp. 115–150, 2002.

[10] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin, "pREST: a REST-based protocol for pervasive systems," *MASS*, pp. 340–348, 2004.

[11] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java on the bare metal of wireless sensor devices," in *VEE*, June 2006.