SEAP: Sensor Enablement for the Average Programmer

Seth Holloway
Drew Stovall
Angela Dalton
Tony Petz
Christine Julien

TR-UTEDGE-2008-007
SEAP: Sensor Enablement for the Average Programmer

Seth Holloway, Drew Stovall, Angela Dalton, Tony Petz, and Christine Julien
The Mobile and Pervasive Computing Group
The University of Texas at Austin
Austin, TX, USA

E-mail: \{sethh,dstovall,adalton,agoston,c.julien\}@mail.utexas.edu

Abstract

The increasing availability of sensing devices has made the possibility of context-aware ubiquitous 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 ubiquitous computing leads users to demand individualized applications, complexities render programming embedded devices unapproachable to the average programmer. In this paper we introduce the SEAP (Sensor Enablement for the Average Programmer) approach 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.

ACM Classification Keywords

D.2 Software Engineering: Software Architectures;
D.2.11 Software Engineering: Software Architectures-
Data abstraction

General Terms

Standardization

Author Keywords

practical ubicomp, ubiquitous computing software architecture, sensor networks, sensor usability

Introduction

George, a civil engineer, is foreman on a construction site. George is upset because two crane arms recently glanced off one another. The accident did not damage the equipment and no one was hurt; however, the innocent event meant the cranes had to be recertified—an operation that cost $100,000. George has heard of advances in GPS technology that can pinpoint objects outdoors with high accuracy. Software companies have contacted George proposing expensive solutions that will prevent future crane collisions. George is reluctant to purchase these expensive solutions because he understands the basic logic of the problem (thanks to introductory programming courses in college) and wants to be able to easily tailor his solution to future job sites and individualized needs. Unfortunately, the devices necessary to complete this distributed task require several different low-level programming languages with different interaction and communication techniques; George does not understand or have the time to develop these complex multi-device solutions, so he must opt to purchase an expensive, proprietary solution.

If we take a deeper look into the challenges George faces, we can begin to understand how to ease the programming burden for these common application situations. Using existing technologies, George would be required to develop and integrate multiple distinct pieces of software to support his goal: 1) custom applications to collect and generate data (perhaps in different programming languages for different types or brands of devices); 2) a distributed application that connects his devices to some computer set up to read and use the data; and 3) the high-level user interaction application. In George’s specific case, the first piece of the application would instruct individual devices in what type of data to collect and at what frequency. To prevent future crane accidents, George’s application may need to collect GPS data and also potentially crane movement or load data. This is a potentially difficult undertaking because it requires knowledge of specialized device programming languages as well as the (possibly complex and varying) sensor data formats. These devices are distributed throughout George’s job site, so the second piece of his application must control communi-
cation among devices and George’s computer to collect the data to a single location. Different device types or brands may have different programming interfaces for interaction. Finally, when all of the data is collected to George’s computer, the third component of his application is the user interaction code, which may allow George to specify reasonable operating parameters or to simply monitor crane behavior on the site.

Frequently, average programmers like George find programming ubiquitous computing applications that rely on such distributed data and interaction unapproachable. As an additional challenge, bringing existing applications into the world of ubiquitous computing often requires completely rewriting them to take advantage of new sensing and actuating capabilities because of the radical differences in programming paradigms. The complexity associated with developing ubiquitous computing applications is simply too high for average programmers, particularly those looking to rapidly create applications that enhance their own lives or workplaces. Researchers and laypersons alike realize that the availability of even very simple sensing devices increases the number of applications possible, enabling everything from passive context-aware programs to active home automation. Unfortunately, the reality remains that average programmers with the ideas and interest to create such applications are not able or willing to overcome the steep learning curve that currently exists for programming these applications.

While George may be fictional, this scenario is grounded in our interactions with researchers and practitioners in civil engineering where similar scenarios abound. The potential for ubiquitous computing research to impact the safety and efficiency of construction job sites is real. However, such application scenarios are not limited to construction applications; programming an expressive application for any environment with embedded sensing, computation, and actuation encounters the same challenges. 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 in which an entire sensor network (including software and hardware) is deployed to solve a single problem independent of the availability of other devices or resources already deployed on the site. While sensors can be thought of as simple data sources, these current approaches fuse the entire application stack, creating significant but unnecessary interdependencies. To enable programmability of ubiquitous 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) architecture, makes ubiquitous computing applications easier to develop. SEAP effectively 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 ubiquitous computing applications or to integrate existing applications into a ubiquitous computing environment. Our approach takes advantage of the wealth of experience the average programmer has in web programming. With the growth of the demand for new consumer applications in the ubiquitous computing realm, we anticipate that given appropriate development aids, the adoption of ubiquitous computing applications will be similar to the growth of the Internet. Initially, only large organizations (namely corporations, universities, and governments) had a web presence. However, the Internet age truly burgeoned once individuals could create and maintain web pages. To achieve the same growth in ubiquitous computing, sensors and actuators should be easily accessible through a suite of abstractions usable by the average programmer. The SEAP approach achieves this, allowing programmers to easily integrate sensors and actuators without any awareness of the specific low-level languages and protocols.

We have developed SEAP to empower users and bring the ubiquitous computing paradigm to the average programmer. In this paper, we start by overviewing related work and comparing our approach to previous efforts. We then detail the SEAP architecture and the process by which SEAP can be used to implement and test applications. We illustrate the approach from the ground up using George’s construction site monitoring application. We also discuss our experiences applying the SEAP methodology to an existing application, Ubi-Coffee. Finally, we conjecture about extensions to and applications of SEAP and conclude.

Related Work

As sensing devices have become increasingly capable and available, custom designed applications have been developed. In general, the devices the applications employ range from resource-constrained micro devices such as Mica motes [11] to larger, more capable devices such as cell phones or domain-specific sensors. These devices have been used to create custom applications that include wildlife tracking in ZebraNet [13], structural monitoring of the Golden Gate Bridge [14], and urban walkability [4]. We target these applications and similar ones that benefit from having sensor data accessible to an average programmer who can easily create an individualized ubiquitous computing application.

The notion of sensor/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 [2]. Posts include information about a device’s characteristics (location, for example), enabling a degree of content-based discovery. Another ambitious project has created a centralized website that will accept sensor data generated worldwide [5], a technique cleverly titled slog (sensor log). Here, sensors first communicate their data to a base station supporting a particular sensor network. The base station funnels sensor data to the clearinghouse using a connection to the Internet. Because we are interested in enabling more personal applications, in SEAP, users control their own data. This provides a more ubiquitous computing flavored approach, since data and actuation events are 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 governments and affiliated companies to take advantage of a massive set of federated devices and sensors using SensorML, a sensor model language that defines an XML schema to identify and use sensors [3]. Microsoft’s SenseWeb project [19] has also provided a generic method to push sensor data online. However, both approaches rely on SOAP web services, which can be inflexible, slow, hard to maintain and manage, and heavyweight [6, 8, 15]. For large deployments, economies of scale marginalize the overhead, but SOAP web services present an unnecessary cost that we set out to alleviate. This is especially relevant to ubiquitous computing deployments where devices are often resource-constrained, demanding highly-efficient and streamlined solutions.

Our goal is to enable devices to both publish data and receive configurations and commands. We aim to make both the tailoring of the device behavior and the sensed data easily accessible to application programmers. While other approaches provide mechanisms to publish data, SEAP also provides the ability for sensors to consume and act on data. We minimize the interface for both sensors and programmers, 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 and flexible. REST is an architectural style for network systems that promotes the transmission of domain-specific data over HTTP. Key to REST is the notion of resources; every piece of specific data should be universally accessible as a resource. Users interact with resources using a small set of well-defined commands to manipulate the representation of a resource. Some work has been done to apply the REST style in the sensor domain [7, 16] although this work violates many REST principles with large, 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. Additionally, the aforementioned work on adapting REST to pervasive environments uses an altered version of HTTP. 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.

Our UbiCoffee application, discussed later in this paper, reduces the overhead associated with sharing resources, in our case coffee. Many other projects have achieved similar results [10, 12], and we build on the success of these projects in demonstrating a straightforward and accessible way to construct an obviously useful applications. In fact, a similar application was the runner-up in a mobile computing “killer-app” contest [17].

**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 SEAP presents to the average programmer. By relying on well-established programming standards, SEAP brings the seemingly unapproachable task of programming ubiquitous 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.
SEAP Architecture: Devices

As SEAP aims to be generally applicable to a wide variety of ubiquitous computing applications, the SEAP architecture assumes that the devices participating are resource constrained and have little ability for complex data management or processing tasks. A major 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 Figure 1 provides a high level view of the relationships between different hardware components in a SEAP system.

![Figure 1. Data flow in the SEAP architecture.](image)

SEAP participating devices are generally 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 or gathering data from the environment to send to the application is considered a sensor. The main goal behind SEAP is to enable applications to connect to collect this information from remote sensing devices in a straightforward and intuitive fashion. The SEAP enablement components running on the remote device mediate between the often proprietary sensor programming language and interface to the accepted web-based programming approaches. Specifically, when a sensor has data to send, the SEAP enablement component 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 enablement component on the remote device is shown below.

```java
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 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 also runs an enablement component responsible for managing the HTTP connection and the data that in turn invokes native functions. A basic SEAP enablement component for a remote actuator could use the following algorithm.

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

Here, the `command-retry-delay` 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 con-
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 enablement component which downloads configuration changes in much the same way that commands are retrieved by an actuator. This allows a developer to programatically 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 webservice as a simple properties-style configuration page like the one below.

```
http://my.host.name/sensor7/config.properties
  data-report-uri=http://my.host.name/sensor7/readings
  data-report-delay=3
  command-uri=http://my.host.name/sensor7/command.properties
  command-retry-delay=10
  configuration-uri=http://my.host.name/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.

Bootstrapping Devices

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, Figure 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. Using the specified initial configuration parameters, SEAP creates a new SEAP enablement component encapsulating these settings. 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 enablement component is loaded on the device (either through a direct connection or through existing over-the-air programming capabilities).

![Figure 2. Example user interface for sensor configuration.](image)

SEAP Architecture: Application Server

As we discussed in the previous section, the devices participating in a SEAP application are assumed to have constrained resources. Therefore, we shift the application-specific logic to the SEAP server where resources are less constrained. Simply centralizing the application logic to a resource-rich device does not automatically resolve the complexities of coordinating distributed devices. It does, however, allow 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 away 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 the re-framing of the ubiquitous computing problem into a standard web application. 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 sheer quantity of high-quality tools that are available for testing, debugging, documentation, and integration for this highly popular domain.
It should be noted that while we describe a two layer architecture here, the pattern could be applied recursively to produce a layered deployment. 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 ubiquitous 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 entirely sufficient for this purpose. Updates to the files made by hand or by other applications would be immediately reflected by the web server, and thus propagate to the devices on the next download cycle.

SEAP-Based Implementation

The primary goal of the SEAP approach is to ease the implementation of ubiquitous computing applications that use sensing and actuating devices. In this section, we demonstrate the simplicity and straightforwardness of application implementation using SEAP by first outlining the development process of a new ubiquitous computing application. We also compare the traditional approach discussed in the introduction with the SEAP approach. We then further illustrate the benefits of SEAP by showing how the approach can easily augment an existing application to include sensors and actuators. Finally, we recount our experience using embedded devices and the SEAP approach.

Creating new SEAP-based applications

Currently, developers creating ubiquitous computing applications from the ground up must solve difficult issues associated with device interconnection, routing, reliability, and data-handling. SEAP, in contrast, requires only the installation of familiar network components and minimal configuration of devices through a configuration utility; the developer codes the entire application in his or her preferred web programming language (PHP, Perl, Python, Java Servlets, ASP, etc).

As a concrete example of the implementation process of a SEAP-based application, we return to our discussion of the construction foreman, George, who wants to ensure two cranes do not collide. An example of this potentially dangerous scenario is shown in Figure 3. To avoid such accidents, George’s application will track the position of the crane booms with GPS; a siren with a flashing light will signal that the cranes are in imminent danger of colliding.

**Figure 3. George’s application scenario.**

Using SEAP, George can quickly create and deploy his application without using any specific knowledge of how to program the GPS devices or the siren. George carries a powerful palmtop device with multiple radios that allows him to effectively communicate with workers on-site and stakeholders off-site. He also has an office on the construction site with multiple laptop and desktop computers. George’s first step in deploying a SEAP application is to determine where to run his application server. Given his particular setup, he has multiple options; all rely on readily available server solutions. The most straightforward approach is to configure a standard application server on his desktop in his fixed office; here George can take advantage of constant power, a fast CPU, and a wired network connection. However, another possibility arises from George’s cell phone. In lieu of a fixed machine, George can set up a lightweight application server on his personal mobile device. Either approach will work, and in fact, the two can work in tandem: switching from one application server to another is simply a matter of updating the configuration files on the participating SEAP devices. The configuration files used to instruct the sensor and actuator devices determine where their connections for sending data and receiving commands go. An example configuration is shown below; the ellipses elide the address of the hosting device of the application server to which future connections should be directed.

```
http://.../safety/crane1/boom/config.properties

gps-report-url=
  http://.../safety/crane1/boom/gps.jsp
gps-report-delay=3
configuration-uri=
  http://.../safety/crane1/boom/config.properties
configuration-delay=60
```

This configuration specifies that within **configuration-delay**, the GPS sensors supporting George’s crane safety application would accept any configuration change, update their internal values, and begin interacting with the newly specified application server. For example, if George is in his office he can configure his sensors’ URIs to point to his desktop machine. Later, as he is walking around the construction site he can update the devices to interact with the application server hosted on his cell phone.
With the application server in place, George can set up the remote sensor and actuator devices that will supply his application with data and the ability to impact the environment. For example, he may attach a GPS unit to his computer to install the necessary SEAP enablement components and provide it its initial configuration. The configuration utility, like the one shown in Figure 2, prompts George to provide the URI to which the attached device will post location readings. George enters the information based on the target URI and the data format his application will consume. The configuration utility loads the necessary program on the device. With the device initially configured, George can deploy it on his site; future changes to the configuration can be made remotely through the SEAP enablement component, as described in the previous section. This process can be repeated for the additional GPS sensors and the site siren.

With his devices deployed and his application server ready to accept connections, George can finalize his application by completing the front end implementation. He constructs a web-based application using the language of his choice. More important than how George creates the user interface is how this user interactive application gets information to and from a distributed set of sensors. In this particular example, when a GPS sensor posts an updated position to the application server, the data is made available to George’s application through the shared URI. In particular, George’s web application can compare the current crane arm positions with one another, add a safety factor based on their maximum speed (a standard piece of domain knowledge for foremen like George), and send an alert to the siren if the cranes are in danger of colliding. George’s development is now reduced to determining if two points overlap—an algorithm that is available online and in textbooks—and managing configuration files—a standard task in programming and systems administration.

Applying SEAP to Existing Applications

Beyond easing the development of new ubiquitous computing applications, SEAP allows existing web applications to be simply reinstrumented to interact with embedded sensing and actuation devices. We demonstrate how to apply the SEAP architecture to an existing web application to integrate physical devices. The application we use for this section is an application we call UbiCoffee, which allow a group to share coffee brewing responsibilities, costs, and benefits. Previously, our CAFFEINE COMMUNE would send email, chat or text messages, or just shout whenever someone brewed coffee. Often, a race ensued to grab a cup before the pot emptied, or alternatively, hours later a half-full carafe was poured out because it was no longer fresh enough to drink.

We created a basic web application to track the amount of coffee left. Users would manually update the amount left (for example, after brewing a fresh pot or pouring a cup) using HTML forms. Users could also check the amount of coffee remaining before heading to the coffee machine. The application remained unchanged for many months; it satisfied our requirements, although we often brainstormed how we could add sensors to reduce our reliance on web forms. SEAP provided an opportunity to reinstrument UbiCoffee to receive information from sensors. The following sections describe our implementation—including a comparison of UbiCoffee before and after integrating sensors—as well possible application extensions.

Devices

Our solution follows the “engaging” paradigm [18] by replacing HTML forms with device interactions. The device-driven UbiCoffee implementation uses Sun SPOT devices [1]. SPOT devices communicate using IEEE 802.15.4 and have several capabilities including two push buttons, eight LEDs, a temperature sensor, and a light sensor. Our UbiCoffee network has one sensor (which posts the amount of coffee left based on user interaction) and one actuator (which displays the amount of coffee left). The sensor is placed by the coffee machine where users press one button to indicate one cup of coffee brewed; the other button corresponds to one cup of coffee taken. The actuator is mounted on the wall in the middle of the laboratory where it can be viewed easily. Although we only describe one actuator we have used up to three actuators as distributed displays. Integrating sensors into UbiCoffee in this manner offers benefits to usability without introducing complexity to the application because we replaced manual user interactions on a webpage with interactions on sensors at the coffee pot.

As this is the first SEAP application, we hand-implemented the SEAP enablement components to be used on the SunSPOT devices. Going forward, device manufacturers or the community will provide these software components for devices. To demonstrate the ease of implementing the data reporting algorithm on a sensor platform, we show our publishData method in Figure 4. This code is a piece of the SEAP enablement component that runs on the SunSPOT, which runs a Java implementation of the Java2 ME specification. For brevity we omit exception handling and the try-catch block to properly close connections.
public void publishData(String URI, String data) {
    String URL = URI + '?data=' + data;
    HttpConnection connection = (HttpConnection)Connector.open(URL);
    connection.setRequestProperty('Connection', 'close');
    InputStream in = connection.openInputStream();
    while (in.read() > 0) {
        //do nothing
    }
}

Figure 4. A Java implementation of the function sensors use to publish data.

Application Server

The UbiCoffee application server is a standard desktop PC running Linux. We use the Apache httpd web server and the MySQL database. The application is written in the Ruby language and executed by the "Ruby on Rails" platform. Sensors publish data to a hidden web form which updates the database. When a web browser or the amount-indicator device requests the status of the system, an appropriate dynamically generated document is rendered using values in the database.

User Interface

The user interface is implemented as a set of Ruby scripts that execute on the central server. Figure 5 shows a screenshot of the UbiCoffee status page for web-browsers. The page prominently displays the amount of coffee left with a numeric readout and color-changing background from red (nearly empty) to yellow to green (nearly full). The application’s page for device interaction is devoid of the HTML used for layout, instead displaying just the information.

http://.../ubicoffee/user/cup-count

Number of cups available at machine: 4.0

http://.../ubicoffee/device/cup-count

4.0

Figure 5. UbiCoffee’s interfaces to users and devices.

In our application, we were able to skip the creation of the configuration update components for several reasons: 1) we have a dedicated device whose sole function is to display the amount of coffee remaining; 2) we do not plan to change the URI or update frequency, and 3) we can manipulate the sensor code easily. The complete SEAP implementation includes the reconfiguration components and properties files containing URIs and delays.

UbiCoffee: Before and After SEAP

As described above, UbiCoffee started off as a basic web page that allowed a group to manually monitor the status of a communal coffee maker; users filled out a simple HTML form in order to update the amount of coffee remaining. Early ideas to automate the update process were ill-defined and represented too much effort for a simple application that was already working. On the other hand accepting sensor data rather than form input from users was remarkably straightforward to implement using SEAP. We added a new form and handler to accept changes in the amount of coffee (rather then the amount of coffee remaining). The existing testcases were still valid, so by simply adding tests for the new form and handler, we were able to quickly verify the program worked properly.

Testing SEAP-based applications. Three distinct interaction types must be examined when testing SEAP-Based applications: 1) device to server: a device publishes data, and the value is received by the application server, 2) server to device: a value is changed on the server, and the device retrieves the new value, and 3) end to end: a device updates a value on the server, and another device retrieves the new value. Categorizing interactions in this concrete manner makes application development well-defined and intuitive. The user may test the application using standard web programming methods and tools. For example, the user can create a web form to emulate a sensor posting data. This is accomplished very easily as seen below

<form action='' method='post'>
    <input type='text' name='cups' value=''/>
    <input type='submit' value='Submit'/>
</form>

UbiCoffee Extensions

UbiCoffee itself is a basic application; it satisfied our requirements both before and after using the SEAP approach. We demonstrated the ease-of-use of SEAP and received very positive feedback from users. UbiCoffee and other one-off applications are a solid foundation for tangible, individually-developed ubiquitous applications that provide many possible extension points. Some methods to expand the system follow.

• Employ an increased range of sensors and devices. Our no-frills web interface allows any web-enabled device (such as cell phones and PDAs) to interact with the system; however, we could exper-
iment with other novel display devices. We could increase the possible functionality of the system by accepting additional or alternative sensor readings; for example, by adding RFID the UbiCoffee application could also automatically identify users.

- **Augment the system with cost information to develop a pricing model.** With user tracking, we could identify consumers and present them a bill or ensure they contribute by buying and brewing coffee.

- **Apply security mechanisms to the system.** Input from specific sensors could be authenticated before being applied to the application server.

- **Add a recommendation engine.** Using usage history and other context information, a recommendation engine could guide the user in the amount and type of coffee to brew.

### Experience

UbiCoffee was developed with minimal web-programming experience (three web applications developed collectively). We believe our experiences reasonably reflect the capabilities of an average programmer. Sensor development, which SEAP expects to be performed only once and by device experts, took two weeks while the web application required only minor changes when we switched the input method to use sensors.

The general development process mirrored existing approaches so we could utilize the vast amounts of information available online. Ruby on Rails proved to be a great platform for rapid database-based web development, although the database may be overkill for simple applications.

Sensor programming was not nearly as user-friendly. The ability to work in Java afforded by our use of Sun SPOT devices greatly eased development. We encountered some difficulties in debugging the sensor functionality. The primary challenge was the lack of a console to monitor system output. Physically connecting the devices to the development machine allowed us to view print lines on a console. Future revisions will provide emulators and simulators to ease the process. We found the Sun SPOT was a smart choice for rapidly developing sensor applications. The SPOT features were well documented and adequate for our purposes. There is a great deal of information available in highly-responsive forums, a development manual, and a published API. Additionally, programming in Java, even a slim version, was far easier than out past experiences in sensor-specific languages. After refactoring, the sensor integration required around 250 lines of non-commented code for full functionality.

Since UbiCoffee first went online, two students (one graduate and one undergraduate) have used SEAP in similar applications using JSP and PHP—two languages that the students had never used before; these applications were extremely easy to program despite the lack of a priori knowledge of either language. Given our experience, we believe that a variety of context-aware ubiquitous computing applications could be developed quickly and easily using the SEAP system. Programmers are already familiar with this paradigm when the input device is a personal computer, so integrating diverse hardware will be intuitive.

### Future Work

We see several extensions to and applications of the SEAP system. Currently we are using the technique to control unmanned vehicles in a large-scale pervasive computing testbed. We are also investigating the use of this technology in other ubiquitous computing scenarios; in particular, we see an opportunity to apply the SEAP approach in the absence of infrastructure—SEAP is particularly well-suited to ad hoc 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.

Simultaneously, we are developing the configuration utility and accompanying sensor code. We have made all source code freely available at [http://mpc.ece.utexas.edu/SEAP/](http://mpc.ece.utexas.edu/SEAP/). We would like to extend this architecture to other platforms and create tutorials for use. Ultimately, we would like to publish prototypes for end-user customization that allow novice programmers to create exciting, useful ubiquitous computing applications. Another possibility is ready-to-use sensor data provided via blog integration and widgets to be used in mash-ups or composite applications. The Internet experience could be more personal by adding sensor-provided context information.

### Conclusion

In this paper we presented the Sensor Enablement for the Average Programmer (SEAP) architecture, which lowers the barrier for entry into programming ubiquitous 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 architecture allows people to interact with sensors and actuators without learning new or proprietary languages or
procedures. SEAP allows rapid prototyping of ubiquitous computing applications as shown in the construction site application that was derived from real-world requirements. SEAP also provides a method to quickly add multiple devices to existing web applications as shown with UbiCoffee.

The scalability and accessibility of the Internet make the web an ideal platform for increasing the number of active ubiquitous computing applications. SEAP takes advantage of the vast body of work on web programming to provide an architecture that is easy-to-understand, easy-to-use, language-agnostic, robust, reusable, and immediately achievable. People are ready for ubiquitous computing applications; we simply need an accessible method to enable multi-device developments. SEAP is that method.

Acknowledgments

The authors would like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment. 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