Automatically Exporting TinyOS Modules as Web Services

Michael Okola and Kamin Whitehouse

Computer Science Department, University of Virginia
{mjo2g,whitehouse}@virginia.edu

Abstract. As embedded devices become increasingly useful and ubiquitous, it will be important to incorporate them into enterprise applications through Web services and service-oriented architectures. Existing approaches have limitations in terms of either resource consumption or burden on the programmer. In this paper, we present a new framework to automatically export TinyOS modules as Web Services using a resource-efficient architecture, thereby minimizing both resource usage and the burden on the programmer. As a proof of concept, we demonstrate our framework by creating a TinyOS module that can be accessed by both Python and Java clients with no additional code required from the user. We demonstrate that our framework incurs very little RAM, ROM, CPU, or network overhead, even when the node moves between gateway nodes. The techniques used by our framework currently support both TinyOS 1.x and TinyOS 2.x, and the techniques can easily be generalized to support other wireless embedded programming systems.

Key words: Sensor Networks, Web Services, Service Oriented Architectures

1 Introduction

New industrial applications for wireless embedded sensing devices are continuing to be created, including energy conservation [7], medical monitoring [14], and industrial automation [3]. As embedded devices become increasingly useful and ubiquitous, it will be important to incorporate them into enterprise applications, where programs from many different vendors, service providers and administrative domains must interact in a programmatic fashion to produce new applications. For this reason, several papers have recently proposed techniques to abstract the details of wireless embedded devices as high-level Web services. Web services and service-oriented architectures in general have become a standard for interoperability, allowing programs written in almost any language to programmatically interact with a variety of databases, Web pages, and other resources available on the Web.

In general, two approaches have been used to provide Web services on wireless embedded devices: 1) small-footprint yet standards compliant implementations of the Web service support infrastructure that can be executed on resource constrained devices [11,9], and 2) the use of gateway servers that provide standards

compliance while reducing resource demands on the embedded devices [13,5]. However, each of these approaches has its limitations. The first approach is very easy for the programmer, but places additional burden on the device. On the other hand, the second approach minimizes reduces resource demand but places additional burden on the programmer.

In this paper, we present a new framework to automatically export TinyOS modules as Web Services. The goal is to minimize both resource usage and the burden on the programmer. The framework is based on TinyOS, which is one of the most common ways to program wireless embedded devices [10]. It has a rich library for sensing, actuation, distributed processing, and wireless networking. It also has a modular software architecture that naturally lends itself to exposure through Web services. Our framework allows a user to define a new service simply by creating a new TinyOS software module, compiling it, and programming it onto a node. The framework automatically generates a Web service with the same function signatures as the TinyOS module that is hosted by a gateway device connected to the Internet. The framework also automatically generates a wireless interface for passing an arbitrary number of function arguments and return arguments with arbitrary types between the Web service and the TinyOS module using multi-hop wireless networking. Thus, when an enterprise server calls a function on a Web service created by our framework, the operations are actually performed on a wireless embedded device.

As a proof of concept, we demonstrate our framework by creating TinyOS modules that can be access by both Python and Java clients with no additional code required of the user. Our framework elevates the abstraction of wireless embedded devices for integration with enterprise applications, without incurring any additional code or effort from the programmer other than what would have been required to program the device in the first place. We demonstrate that our framework incurs very little RAM, ROM, CPU, or network overhead on the sensor nodes that provide the Web service. We also propose techniques and estimate the cost of supporting mobility of wireless embedded devices, as they move between gateway nodes. The techniques used by our framework currently support both TinyOS 1.x and TinyOS 2.x, and the techniques can easily be generalized to support other wireless embedded programming systems, such as Contiki [6] and LiteOS [2].

2 Background and Related Work

Most modern programming languages provide powerful rich support for Web services, allowing a programmer to create a new Web service simply by creating a new class or object. The class or object can be automatically exported into a Web service by clicking a button in the IDE, which uses compiler modifications, introspection, and other techniques to automatically load the object into a Web Server and export the functions of the object through a Web services API. The overall architecture is depicted in Figure 1(a): the gray components are required

to be written by the user, while the white components are system infrastructure that is automatically provided and integrated by the development environment.

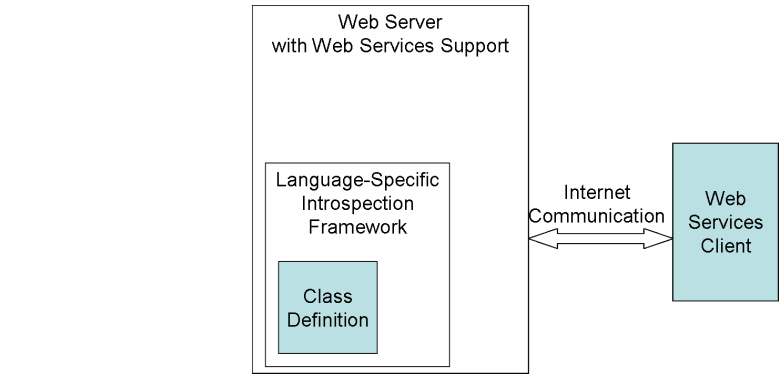
Web services were designed for Internet-scale cooperation, and many existing standards and protocols are too heavy weight for the bandwidth and resource constrained nature of wirless embedded devices. Several approaches are currently being taken to address this concern, including new markeup languages [1], protocols [9], and architectures. Most of the new architectures fall into one of two categories. In the first category, the supporting infrastructure required to support Web services is reduced to minimal yet standards compliant versions that can still operate on resource constrained devices [11]. This architecture does not necessarily incur additional work from the user, but it does require substantial additional resources from the embedded devices that are used solely for the purposes of being standards compliant.

An alternative architecture is to use the gateway device to provide standards compliance, and to use proprietary protocols to communication between the embedded device and the gateway [13,5]. This approach exploits the fact that the embedded devices typically use a specialized radio standard such as 802.15.4 and must therefore have a dedicated gateway device. The Web services infrastructure can therefore be integrated into the gateway device. The shortcoming of this approach is that the creation of new Web services becomes much more complex: the user no longer needs to simply create a new object or class. Instead, the user needs to created a distributed program that maps some of the functionality onto the sensor node, and other functionality onto the gateway device, and manages communication between the two. This type of architecture is depicted in Figure 1(b), and introduces substantial new burden on the user due to the wireless nature of the embedded device.

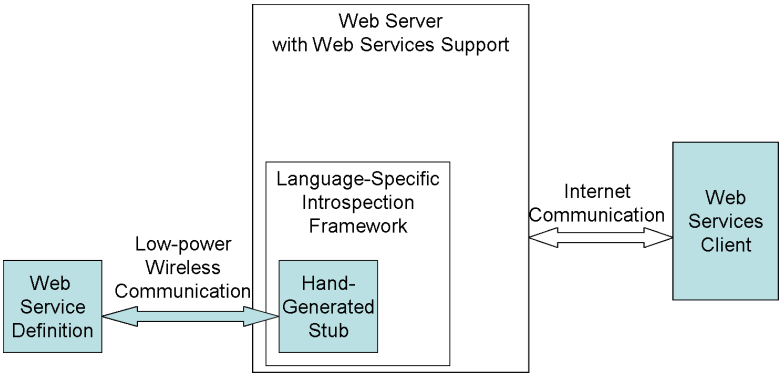
3 Implementation

In this paper, we provide a new software framework that can produce a Web services abstraction for wireless embedded devices without incurring additional burden on the programmer or the embedded device. This framework allows the programmer to define a new Web service simply by creating a new TinyOS module. At compilation time, the framework automatically generates a specialized, efficient interface to the software module through a low-power wireless network, and automatically generates code to execute on the gateway node to support the Web services API.

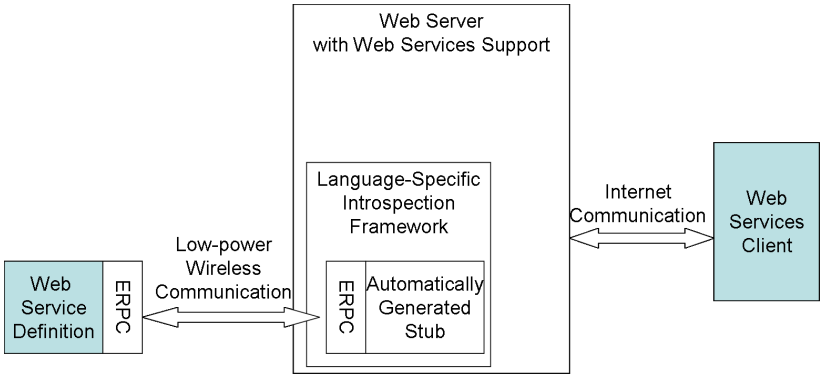
Figure 1(c) depicts the architecture of our solution, and illustrates that the user must only write the TinyOS module and can then access it directly using a standard Web services client. All other infrastructure is automatically supported and integrated, including the gateway code that supports standards compliance. Thus, this architecture allows standards compliance overhead to be offloaded to a more powerful gateway node, relieving the embedded device of undue resource burden. At the same time, the burden on the developer remains just as simpe as it is for a developer of a non-embedded Web service.



(a) Standard Web Services



(b) Web Services for Embedded Devices



(c) New Architecture for Web Services for Embedded Devices

Fig. 1. a) Standard Web services architectures provide supporting infrastructure, so the programmer only needs to define a class and a client. b) Web services for embedded devices are more complex to create because it must be defined as a node and gateway program. c) Our framework generates the gateway program and all wireless protocols automatically, reducing the burden on the programmer while maintaining efficient operation and standards compliance.

3.1 Automatic Stub Generation

Once a TinyOS module is created, our framework automatically creates a software stub that will execute on the gateway node and will be served as a Web service. When an enterprise application accesses the stub Web service, the function parameters are automatically forwarded to the TinyOS module on the embedded device using an efficient wireless protocol. After the functionality is executed, the return arguments are automatically returned to the stub. An example of a Web service created using TinyOS and NesC [8] is shown below, that takes a `uint8_t` argument, changes the values of the LEDs on the node, and returns *SUCCESS* as a return value.

TestWebServiceC.nc

```
#Sample Web service written in NesC
includes Rpc;
module TestWebServiceC {
  provides command error_t testFunction(uint8_t foo);
  uses interface Leds;
}
implementation {

  command error_t* testFunction(uint8_t foo){
    call Leds.set(3);
    return SUCCESS;
  }
}
```

To interface the gateway node with this module, our framework uses the embedded RPC (ERPC) capabilities provided by Marionette [12]. The compiler automatically parses all function names, argument names, and argument types of the TinyOS module. It automatically generates a ERPC stub on the gateway side that has the same function signatures. When a stub function is called on the gateway, it automatically marshalls all arguments into a TinyOS packet and sends them to the embedded device using multi-hop wireless networking. It also automatically generates a ERPC stub on the embedded device that is *wired* to all interfaces and functions provided by the TinyOS module. When a new message is received, the arguments are unmarshalled and passed to the appropriate function. When the function has completed, the return argument is marshalled into a TinyOS packet and sent back to the gateway. The ERPC stubs on both the embedded device and the gateway device are depicted in Figure 1(c).

Because our framework makes use of the Marionette ERPC tools, it inherits a very low overhead implementation. It requires only 153 Bytes of RAM and less than 4KB of program memory on the node. It introduces less than 400 lines of code on the node with no loops or recursion, and so it introduces essentially no additional CPU overhead. Finally, the message arguments are passed in a byte-packed format using the standard TinyOS message headers. Thus, the messaging overhead is the minimum that can be achieved without deep changes to the TinyOS networking stack. In order to fully support Web services for TinyOS

modules, we needed to make several changes to the Marionette framework. Two of the major changes are described in the subsections below.

Adding Events to ERPC The first major change we made to Marionette was to add support for `events` to ERPC. ERPC was originally designed for a *thin server / heavy client* architecture, in which the sensor node was a lightweight server and commands could be called on it from a heavyweight client, which was a Python user terminal. Thus, ERPC originally only supported TinyOS *commands*, which could be called by the user terminal. However, this framework does not support the need for sensing and actuation through Web services, which could require a *split phase* operation. In other words, the original command could pass control of a sensing or actuation command to the hardware, and return a value to the client before the sensing and actuation command has been completed. In TinyOS, an *event* is triggered by a hardware interrupt to indicate that the command has completed, and this event can be handled in software. We added support for events in ERPC, so that the server could initiate a transmission to the client to complete a split phase operation.

The key challenge to supporting events in ERPC is that events can occur at any time due to interrupts, even when not triggered by a Web services call. Thefore, we used a special *enable flag array* for every TinyOS event. By default, events do not trigger ERPC calls to the gateway. This allows timers, clocks, and other standard system operations to be performed without introducing spurious network traffic. However, when a command is called through the Web services server, the corresponding enable flag on the appropriate event is enabled by the ERPC stub on the embedded devices, so that the split-phase response message will be generated. The enable flag array introduces one byte of additional RAM overhead to the embedded device for every event provided in the TinyOS program. If necessary, only those TinyOS events that are necessary for Web services can be made accessible through ERPC, reducing that overhead to less than a few bytes.

Adding Support for TinyOS 2.x The original implementation of ERPC was written for TinyOS 1.x. We added support for ERPC to TinyOS 2.x, since much of the new development in TinyOS occurs with this newer version. Since TinyOS 2.x is not backwards-compatible with 1.x, several changes had to be made to add support for ERPC. Aside from syntactical changes, the most significant changes were are in the communication code between the PC and node. For example, we removed all dependencies on legacy routing protocols such as Drip and Drain, as well as other legacy libraries. Furthermore, we integrated a new Python-based communication stack for the PC to eliminate a previous tool dependency on Java. TinyOS 1.x includes a Java serial communication implementation, but lacks an implementation in Python. Therefore, JPype, a tool that gives Python programs access to Java class libraries, is used to send messages to nodes from Python. This has several disadvantages, however. Java and, more importantly, JPype must be installed in order to use Marionette. Mes-

sages must be wrapped into Java classes, which requires several extra methods to create and parse them. It also introduces significant memory and processing overhead when running Marionette, due to the need to run not only the Python interpreter but also the just-in-time compilation required by the Java Virtual Machine. Finally, debugging tools such as pdb are much less useful when using JPype, as line-by-line debugging across lanuages is not possible. Unlike TinyOS 1.x, a Python serial communication implementation is included in TinyOS 2.x. For Marionette, this toolchain is utilized instead of the original method. Due to the removal of reliance on Java, the newer implementation no longer relies on external packages, significantly decreasing the amount of effort needed to install, set up, and execute Marionette. It also increases execution speed by removing the overhead of wrapping Python objects in Java form and the neccessity of running the Java Virtual Machine. Therefore, installing of Marionette in TinyOS 2.x is achieved in many fewer steps than it was in TinyOS 1.x. Only one directory structure needs to be downloaded, either from an online repository or by other means, and a single line inserted into a shell script. This has been accomplished by a combination of improvement of the scripts used to set up the environment for Marionette and of the removal of JPype, the only external, independently-installed tool previously required, from use. This simplicity did not require that any functionality be removed from Marionette; the steps for utilizing Marionette in a given nesC program are the same in most cases, and where they differ they are simpler. These improvements to the Martionette code base make our Web services framework painless to install and seamless to integrate with the existing TinyOS development environment.

3.2 Supporting Mobility

In order to support mobility, a description of the TinyOS module, all function signatures, types, and ERPC interfaces are stored in a XML file in the node's flash. If the node moves to a new network, it can associate with a new gateway device by uploading this XML file. The new gateway device can then automatically recreate the full stub class and ERPC interface, which can be served as a Web service from the new gateway device. Thus, the cost of moving between networks is the cost of uploading the XML description of the TinyOS module to the new gateway. This XML file does not need to conform to any Web services standards, such as WSDL [4], and therefore can represent the information as compactly as possible. In our case, the representation of a function is stored using a specialized and efficient basic XML format. The example function from the example TinyOS module shown above is represented as:

```
<function command="" provided="0" name="testFunction" ref="0x406b3088">
  <type-function size="I:1" alignment="I:1">
    <type-int cname="unsigned char" unsigned="" size="I:1" alignment="I:1">
      <typename><typedef-ref name="error_t" ref="0x403098d0"/></typename>
    </type-int>
    <function-parameters>
```

```
      <type-int cname="unsigned char" unsigned="" size="I:1" alignment="I:1">
        <typename><typedef-ref name="uint8_t" ref="0x401a82f8"/></typename>
      </type-int>
    </function-parameters>
  </type-function>
  <parameters>
    <variable name="something" ref="0x406b5ef0" loc="4:TestInterface1.nc">
      <type-int cname="unsigned char" unsigned="" size="I:1" alignment="I:1">
        <typename><typedef-ref name="uint8_t" ref="0x401a82f8"/></typename>
      </type-int>
    </variable>
  </parameters>
</function>
```

This function description includes the name of the function, the parameters, the parameter byte alignment, and all other important information needed to automatically generate a stub class and corresponding ERPC interface. In this case, the entire function description requires about 1KB of storage and, especially if compressed, can be transmitted in a relatively small number of TinyOS packets. As the number of functions and arguments increases, therefore, the cost of migrating between networks increases. Thus, with our framework, devices with a relatively sparse Web service interface can therefore move more frequently, while devices that contain a more complex interface would need to pay more in terms of energy and bandwidth in order to move frequently between networks.

3.3 Integration with Existing Web Services Infrastructure

Once a stub has been created for execution on the gateway node, it can be integrated into any existing tools and supporting infrastructure for Web services or other service oriented architectures, such as the Java or .NET framework support. As a proof-of-concept, we demonstrate the use of the stub class with the Python XML-RPC server application as well as clients written in Python and Java to demonstrate the ease with which our framework can be used to integrate TinyOS modules with existing tools. XML-RPC is an XML-based protocol for calling functions between programs written in different programming languages. It is the predecessor to the current SOAP standards typically used for Web services.

Our framework utilizes Marionette and a XML-RPC server to directly translate function calls from a remote client into calls to the TinyOS module executing on the node. When a request is received, our framework checks the function to confirm that the function exists and is available to be called. It then calls the function with the provided arguments, if any, and returns the results to the client. Any errors are returned back to the XML-RPC client.

Our framework also exports the Python *str* method, which takes any valid PyTos object as the argument. This invokes the object's default *__str__* method, which returns a user-readable string detailing the object's contents. This enables a user to utilize introspection to determine which functions, variables, and

other attributes are available. This operation demonstrates the flexibility of our framework.

Because our framework can be accessed by clients running in almost any language, there are complexities involved in handling complicated types, such as structs, because they may be defined in terms of other structs. This presents a non-trivial portability issue caused by the limitations of XML-RPC. Due to the difficulty of representing these types in a way that can be universally supported by all clients, only simple types and arrays of these types are currently supported. In general, the limitations imposed on our framework are inherent in the tools and supporting infrastructure being used. For example, if the stub object were passed to a .NET Web services framework, the limitations of that framework would apply.

3.4 Accessing the TinyOS Module as a Web Service

In this section, we demonstrate how to access a TinyOS module as a Web Service using both a Python and a Java client. Because of Python's dynamic typing system and its simple XML-RPC package, utilizing our framework with Python is extremely simple. The XML-RPC package is contained in the core Python distribution, so it requires little time to begin development. An example Python client that calls the test function provided by the sample TinyOS module above is shown below. In this example, only a single line of code is required to connect to the TinyOS module as a Web service, and another line of code is required to call a function on it.

MarionetteClient.py

```
#Sample Python XML-RPC Client
server = ServerProxy("http://localhost:21777", allow_none=True)
print server.app.TestMarionetteC.testFunction(8)
```

Java's XML-RPC calls are more complex than those of Pythons, due to Java's strict typing system. Once the syntax is understood, however, making RPC calls using the Java client is still relatively simple. One disadvantage of using Java is that several lines of overhead are required to load the XML-RPC library, in contrast to the single line of code required in Python. This overhead is language specific, and relates to the level of integration of each library with each language. The overhead required to use a Web service is much lower for Java than the use of XML-RPC.

MarionetteClient.java

```
#Sample Java XML-RPC Client Function Calls
XmlRpcClientConfigImpl config;
XmlRpcClient client;
config = new XmlRpcClientConfigImpl();
client = new XmlRpcClient();
try{
    config.setServerURL(new URL("http://localhost:21777"));
}
```

```
catch(Exception e)
   { System.out.println("Cannot configure connection: " + e);}
client.setConfig(config);
Integer result;

try{
    result = (Integer) client.execute(
       "app.TestMarionetteC.testFunction", new Object[](new Integer(2)));
    System.out.println(result);
}
catch(Exception e){ System.out.println("ERPC exception: " + e);}
```

4 Conclusions

In this paper, we propose a new framework for providing wireless embedded devices as Web services in a standards compliant fashion, without introducing undue burden on either the device or the programmer. We demonstrate that our framework incurs very little RAM, ROM, CPU, or network overhead, even when the node moves between gateway nodes. Our approach can generally be used with any service-oriented archtecture – including Web services – simply by integrating the stub program into existing automated tools and supporting infrastructure such as the Java or .NET tool chain. As a proof of concept we demonstrate the framework using XML-RPC, which is well supported by Python and which is a more simplified predecessor to modern Web services protocols. The techniques used by our framework currently support both TinyOS 1.x and TinyOS 2.x, and the techniques can easily be generalized to support other wireless embedded programming systems.

References

1. M. Botts et al. Sensor Model Language (SensorML) for In-situ and Remote Sensors. *OGC document reference number*.
2. Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS operating system: Towards Unix-like abstractions for wireless sensor networks. In *Proceedings of the 7th international conference on Information processing in sensor networks*, pages 233–244. IEEE Computer Society Washington, DC, USA, 2008.
3. C. Chong and S. Kumar. Sensor networks: Evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003.
4. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1, 2001.
5. F. Delicato, P. Pires, L. Pirmez, and L. da Costa Carmo. A flexible web service based architecture for wireless sensor networks. In *Distributed Computing Systems Workshops*, pages 730–735, 2003.
6. A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, volume 2004, 2004.

7. G. Gao and K. Whitehouse. The Self-Programming Thermostat: Optimizing Setback Schedules based on Home Occupancy Patterns.
8. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, June 2003.
9. N. Glombitza, D. Pfisterer, and S. Fischer. Integrating wireless sensor networks into web service-based business processes. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, pages 25–30. ACM, 2009.
10. P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, 35, 2005.
11. N. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 253–266. ACM New York, NY, USA, 2008.
12. K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *Proceedings of the 5th international conference on Information processing in sensor networks*, page 423. ACM, 2006.
13. A. Woo. Demo Abstract: A New embedded web services approach sensor networks. In *Sensys '07*.
14. A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. Fang, Z. He, S. Lin, and J. Stankovic. ALARM-NET: Wireless sensor networks for assisted-living and residential monitoring. *University of Virginia Computer Science Department Technical Report*, 2006.