

Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks

Kamin Whitehouse[†], Gilman Tolle^{†‡}, Jay Taneja[†], Cory Sharp^{*}, Sukun Kim[†],
Jaein Jeong[†], Jonathan Hui^{†‡}, Prabal Dutta[†], and David Culler^{†‡}

[†]Computer Science, UC Berkeley Berkeley, California 94720 [‡]Arched Rock Corporation Berkeley, California 94704 ^{*}Moteiv Corporation Berkeley, California 94704

ABSTRACT

A main challenge with developing applications for wireless embedded systems is the lack of visibility and control during execution of an application. In this paper, we present a tool suite called *Marionette* that provides the ability to call functions and to read or write variables on pre-compiled, embedded programs at run-time, without requiring the programmer to add any special code to the application. This rich interface facilitates interactive development and debugging at minimal cost to the node.

Categories and Subject Descriptors

D.1 Programming Techniques, C.3 Embedded Systems

General Terms

Design, Languages, Human Factors

Keywords

Debugging, Embedded Networks, Programming, RPC

1. INTRODUCTION

Wireless embedded systems usually run *batch* programs, which are downloaded to the node in their entirety, executed, and return only the final results. Batch programming is designed to be efficient by minimizing network and node activity during execution. However, this opaque execution environment makes development difficult because the programmer has no visibility into or control over application behavior at run-time.

We propose a tool suite called *Marionette* that enables interactive development by allowing a PC to access the functions and variables of the statically-compiled program on a wireless embedded device at run-time. Our first client implementation provides the equivalent of a *remote terminal* to

an embedded device: the network operator opens an interpreter on the PC and is presented with a set of objects representing the software modules actually running on the node. Through these objects, the node's functions can be called, its variables can be read and written, and its enumerations and data structures can be accessed. This Python-based client can be used for interactive debugging and experimentation, and can also be used to quickly *script* new behaviors for the network. However, the Marionette architecture can be used to provide access to embedded devices through any programming environment, including Java GUIs, web services, or .NET.

With the Marionette architecture, embedded applications seamlessly span the PC and the sensor node; parts of the application can be executed on the PC client, remotely accessing functions and variables on the node only when necessary. Other parts are executed locally on the nodes. The programmer can move aspects of the application across the machine boundary to manage the trade-off between efficiency and run-time visibility and control. When developing new and complex functionality, it may be worthwhile to execute mainly on the PC, where the developer can integrate rich visualization and scientific analysis libraries into the application and utilize more powerful debugging tools. As the logic becomes more mature, it can migrate to the node for efficiency reasons if necessary.

The core of Marionette is Embedded RPC (ERPC), which uses a *fat-client/thin-server* architecture to allow the PC to directly call functions on an embedded application. ERPC is used to provide *poke* and *peek* commands, which allow any variable on the node's heap to be read and written. No extra code must be written by the developer to use Marionette; a minimal set of hooks is automatically added to a nesC application at compile time, consuming only 153 bytes of RAM and less than 4KB of program memory on the node. The PC client software imports all information necessary to access the application from an XML file, which is also automatically generated at compile-time. This XML file is stored directly on the node for the ability to introspect and control a node with unknown or outdated code.

While it is difficult to quantify the degree to which Marionette makes embedded application development easier, we do show that it can reduce code size in existing applications by up to 80%. We also use several case studies to show that it allows common tasks to be executed quickly and easily with only a few lines of code. Finally, Marionette has recently been demonstrated to be effective on a wireless

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IPSN'06, April 19–21, 2006, Nashville, Tennessee, USA.

Copyright 2006 ACM 1-59593-334-4/06/0004 ...\$5.00.

MyMsgs.h

```
struct CommandMsg {
    uint8_t data;
    ...
};

struct ResponseMsg {
    uint8_t data;
    ...
};

enum {
    AM_COMMANDMSG = 33,
    AM_RESPONSEMSG = 34
};
```

MyAppC.nc

```
components GenericComm;
...
Main.StdControl -> GenericComm;
MyApp.CommandMsg -> Comm.ReceiveMsg[AM_COMMANDMSG];
MyApp.ResponseMsg -> Comm.SendMsg[AM_RESPONSEMSG];
```

MyAppM.nc

```
uses interface ReceiveMsg as CommandMsg;
uses interface SendMsg as ResponseMsg;
...
TOSMsg msg;
CommandMsg params;
...
task callCommand(){
    ResponseMsg response = call foo.bar(params.data,...)
    memcpy( msg->data, response, sizeof(ResponseMsg) );
    call ResponseMsg.send(TOS_BCAST_ADDR, &msg);
}

event TOS_MsgPtr CommandMsg.receive(TOS_MsgPtr m) {
    memcpy( &params, m->data, sizeof(CommandMsg) );
    post callCommand();
    return m;
}
```

Figure 1: nesC code required to define a roundtrip messaging protocol without Marionette.

testbed of over 500 nodes. The main cost of using Marionette is that each interaction with a node requires network communication.

2. DEVELOPING BATCH PROGRAMS

A network of wireless embedded devices has almost no support for development and debugging. JTAG provides complete debugging support for a single embedded processor, but not for a large network. Many embedded devices can blink LEDs to relay very simple information, but not for complex information or when more than a handful of nodes are being debugged. Eavesdropping on network traffic also provides some clues about network operations, but is not useful for debugging internal logic and is difficult to use in multi-hop networks when the nodes are not all within eavesdropping range. Wireless communication is the only channel for rich interaction with the network. However, a surprisingly complex set of instructions must be manually added to a program for each messaging operation. We distilled the minimal set of instructions which must be added to a nesC and Java application to define a roundtrip message protocol, including message formats, message handler routines, and data marshaling and serialization. These instructions, outlined in Figures 1 and 2, must be replicated

Makefile

```
INITIAL_TARGETS = CommandMsg.java ResponseMsg.java

CommandMsg.java:
    mig java MyMsgs.h CommandMsg -o $$@
    javac $$@

ResponseMsg.java:
    mig java MyMsgs.h ResponseMsg -o $$@
    javac $$@
```

MyApp.java

```
MoteIF mote;
mote = new MoteIF(myGroupID);
mote.registerListener(new OscopeMsg(), this);

CommandMsg msg = new CommandMsg();
msg.data=100;
...
mote.send(MoteIF.TOS_BCAST_ADDR, msg);

public void messageReceived(int addr, Message msg) {
    ...
}
```

Figure 2: PC client code required to define a roundtrip messaging protocol without Marionette. If the user does not configure the Makefile to use MIG, as shown, the Java application would also need to include marshaling and serialization code.

for every function and variable that will be accessed from the PC at run-time.

This unreasonable burden on developers has given rise to several software tools to facilitate messaging between the network and the PC. Tools like DiagMsg/MessageCenter, developed at Vanderbilt University, and MIG, which comes packaged with the nesC compiler, allow a node to marshal data into a message and have it automatically unmarshaled and deserialized when received at the PC. The Config and Command tools, which are freely available in the TinyOS code repository, automatically generate some of the code shown in Figures 1 and 2 for accessing variables and functions. However, none of these four tools provide a fundamentally richer abstraction; the user must still reduce all interaction with the node to a messaging protocol by defining packet formats and message handlers for every operation to be communicated to the node.

One software tool called SNMS [11] does provide a tighter coupling than a simple messaging abstraction by allowing the programmer to export a set of node “attributes” which can be read or written *by name* using a Java client tool. However, SNMS still requires new code to be written for each exported attribute.

3. DEVELOPING WITH MARIONETTE

In contrast to batch programming in which the network program runs autonomously, Marionette allows the developer to observe or change the state of a node at runtime, and to compose the functions of a node application in new ways without downloading any new code to the node. For example, a function may be called on the node, its return value processed on the PC, and the result passed as a parameter to another function on the node or to a function on another node. In this way, Marionette provides a unified

programming architecture in which the network application seamlessly spans the PC and the sensor node. This allows rapid prototyping and experimentation of new application logic in the context of a modern debugging and development environment.

In this section, we illustrate the Marionette user experience through an example application called *Oscilloscope*, which continually reads values from the ADC and sends them out over the radio. This application is the “Hello World” equivalent for sensor network applications in nesC, and is freely available in the TinyOS code repository.

At compile time, the user must enable Marionette scripts, which parse the application code and automatically generate a number of hooks into it. Once the compiled binary is installed on a network, the user can open a Marionette “terminal” into that network by specifying how to connect to the network which, in this example, is through the serial port COM1. Once the terminal is open, a single object called `app` is available, through which the user can access the Oscilloscope application installed on the nodes. As shown here, the `app` variable provides access to all software modules running on the node and their functions and variables as well as all types, enumerations, and messages defined in the application.

```
~/tinyos-1.x/apps/Oscilloscope $ make install
~/tinyos-1.x/apps/Oscilloscope $ marionette.py serial@COM1
>>> app # print the contents of a nesC application
      Enums : 269
      Types : 81
      Messages : 8
      Rpc functions : 21
      Ram symbols : 129
      Modules : ADCM
                AMStandard
                BusArbitrationM
                CC2420ControlM
                CC2420RadioM
                DrainGroupManagerM
                DrainLinkEstM
                DrainM
                DripM
                DripStateM
                FramerAckM
                FramerM
                GroupManagerM
                HPLCC2420M
                HPLUSART0M
                HPLUSART1M
                InternalTempM
                LedsC
                MSP430ADC12M
                MSP430DC0CalibM
                OscilloscopeM
                RamSymbolsM
                RandomLFSR
                RefVoltM
                RpcM
                TimerJiffyAsyncM
                TimerM
                UARTM
                WakeupCommM
```

All variables of all nesC modules are available by default. While all functions could also be made available, there is a cost to the sensor node for each hook that is added for a function and so the current default is to only import those functions or interfaces that are marked with the “@rpc()” tag. Marking functions and interfaces with this tag is the only effort required of the user in order to use Marionette.

Software modules can be accessed as fields of `app`, and functions and variables can be accessed as fields of a software module. Functions can be called normally and do return a

value, although if there are multiple nodes in the network they will return an array of values, one for each node. Functions can also be called only on a specific node by passing an optional `address` parameter. The Marionette libraries automatically perform type checking on function parameters and convert values to the native types of the embedded processor. The values of variables can be read or written through member functions called `poke` and `peek`. Below, we use a software module running on the node called `OscilloscopeM` to view the number of readings that have been collected so far and then use the `TimerM` module to read the local time on the node.

```
>>> app.OscilloscopeM # print the ram symbols and functions
      uint8_t : currentMsg
      TOS_Msg[2] : msg
      uint8_t : packetReadingNumber
      uint16_t : readingNumber

      result_t StdControl.init()
      result_t StdControl.start()
      result_t StdControl.stop()

>>> app.OscilloscopeM.readingNumber.peek() #read a variable
      PeekResponse, nodeID=1:
      uint16_t value : 184

>>> app.TimerM.LocalTime.read() # call a function
      RpcResponse, nodeID=1:
      uint32_t value : 2537364
```

To provide truly seamless integration with the node, the PC also has access to all enumerations, types, structures, and messages defined on the node, all of which are available as fields of the `app` object. In this way, when enumeration or type declarations change in the node application, they also change on the PC and the user does not need to worry about two separate code bases that can fall out of sync. This architecture helps the portion of the application that is running on the PC to deal with new fields being added to a structure or the fact that an integer is a 16-bit value on some platforms and a 32-bit value on others.

```
>>> app.enums.rpcErrorCodes # print the values of an enum
      RPC_SUCCESS = 0
      RPC_GARBAGE_ARGS = 1
      RPC_RESPONSE_TOO_LARGE = 2
      RPC_PROCEDURE_UNAVAIL = 3
      RPC_SYSTEM_ERR = 4

>>> app.types.exception # print the contents of a structure
      int16_t type : 0
      char* name : ptr-> ''
      double arg1 : 0
      double arg2 : 0
      double retval : 0
      int16_t err : 0

>>> app.msgs # print the available network messages
      10 : OscopeMsg
      32 : OscopeResetMsg
      211 : RpcCommandMsg
      212 : RpcResponseMsg
      4 : DrainMsg
      7 : DrainBeaconMsg
      89 : DrainGroupRegisterMsg
      3 : DripMsg

>>> app.msgs.OscopeMsg # print the contents of a message
      TosMsg(am=10) OscopeMsg:
      uint16_t sourceMoteID : 0
      uint16_t lastSampleNumber : 0
      uint16_t channel : 0
      uint16_t[10] data : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

By extending enumeration and type declarations to the PC, Marionette facilitates several tasks that were previously very difficult. For example, TinyOS comes packaged with a very commonly used tool called `Listen`, which eavesdrops on network traffic and prints message bytes to the screen in hexadecimal. Below, we compare the output of the standard Java `Listen` program with a new Python version written using Marionette which, in very few lines of code, uses information about the application to automatically recognize and parse incoming messages. Here, it is parsing the ADC readings which are being broadcast by the node running `Oscilloscope`.

```
$ java net.tinyos.tools.Listen           #execute Java Listen
7E 00 0A 0C 1A 00 00 F8 07 01 00 30 00 34 00 A4 01 58 00 35 ...
7E 00 0A 0C 1A 00 00 F8 07 01 00 30 00 34 00 A4 01 58 00 35 ...
7E 00 0A 0C 1A 00 00 02 08 01 00 DE 00 5F 00 23 03 90 03 54 ...
7E 00 0A 0C 1A 00 00 02 08 01 00 DE 00 5F 00 23 03 90 03 54 ...
7E 00 0A 0C 1A 00 00 0C 08 01 00 17 00 08 03 AD 00 AB 03 1D ...
...
$ Listen.py                               #execute Python Listen
TosMsg(am=10) OscopeMsg:
  uint16_t sourceMoteID : 1
  uint16_t lastSampleNumber : 860
  uint16_t channel : 1
  uint16_t[10] data : [2907, 2906, 2879, ...]

TosMsg(am=10) OscopeMsg:
  uint16_t sourceMoteID : 1
  uint16_t lastSampleNumber : 870
  uint16_t channel : 1
  uint16_t[10] data : [2906, 2895, 2866, ...]
...
```

In order to function, Marionette requires access to the XML files generated from the code that is running on the node. If the user does not have these files, they can be downloaded wirelessly from the node itself. This provides introspective capabilities in which a user can contact a node with an unknown application and quickly and easily observe its state, control it, or extend the application with new functionality.

4. IMPLEMENTATION

The Marionette development environment described in the previous section is provided by the cooperation of six independent tools which are described in this section. At compile-time, hooks for ERPC and poke/peek are automatically added to the nesC application, and all information necessary for a PC client to use ERPC is exported to an XML file. After the binary has been compiled, the symbol table is parsed and all information necessary for poke/peek is also added to the XML file. This XML file is loaded onto the node along with the actual code image when the node is programmed. At run-time, the PC client tools import all information from the XML file, first downloading it from a node if necessary. Then, the user can send commands to and receive responses from the nodes through the PC client over a multi-hop routing layer, or over a wired backchannel if possible. An overview of this process is provided in Figure 3.

We call the architecture just described a *fat client* and *thin server* architecture because as much is done on the PC as possible to reduce the burden expected of the node. For example, the PC client bears the full burden of serializing network transmissions. Furthermore, by automatically generating the ERPC component and extracting the symbol

table at compile-time on the PC, we eliminate the overhead of the node looking up functions, function parameters, or symbols at run-time.

4.1 Embedded RPC

The core of this tool suite is Embedded Remote Procedure Call (ERPC), an implementation of RPC specially designed for embedded systems. RPC is a completely general abstraction that allows procedure calls across language, protection, and machine boundaries, a powerful abstraction that incurs costly overhead [2]. Lightweight RPC (LRPC) is designed to reduce the cost of RPC when used for communication across protection boundaries on a single machine, but not across machine boundaries [1]. The design constraints for the embedded networking domain are different still: 1) ERPC commands need to cross machine boundaries, but not necessarily protection boundaries; since the client and server are in the same administrative domain, they can trust each other, for example, with pointer values 2) the ERPC client is generally several orders of magnitude more powerful than the ERPC server 3) a single client may be interacting with multiple servers, all of which will probably be programmed in the same language. These circumstances encouraged the following differences from traditional RPC:

- Instead of using a Network Data Representation (NDR), all data is transmitted over the network using the native types of the ERPC server. This adds complexity to the ERPC client, which must be able to convert to the format of all CPU architectures that it might interact with. However, it relieves significant burden from the ERPC server, which no longer must perform any serialization or deserialization.
- Instead of using an Interface Definition Language (IDL), we parse the RPC interface directly from the source code running on the embedded device. By not making ERPC general across all languages, we relieve the user from specifying the interface twice. Instead, the user must mark each native function with a “@rpc()” tag in the original source code and a ERPC server stub is automatically generated and linked into the embedded application at compile time.
- A client stub is not generated at all. Instead, the types and RPC interface are stored in an XML file, which can be read by a client program. The client uses this information to marshal parameters and serialize them into server types. This XML file is language independent and allows the client to switch between ERPC servers at run-time without being recompiled.
- Because of the limited resources of the server, ERPC does not support threading or queuing of incoming requests. It also has an optional parameter to not send a RPC response in order to reduce unnecessary burden on the network, especially when sending a RPC request to all nodes in a large sensor network.

In nesC terms, the automatically-generated ERPC server stub *uses* each RPC function or interface, and these are *wired* to the modules that *provide* them. It contains a single message handler function that determines the function that is being called, unmarshals the function parameters, calls the

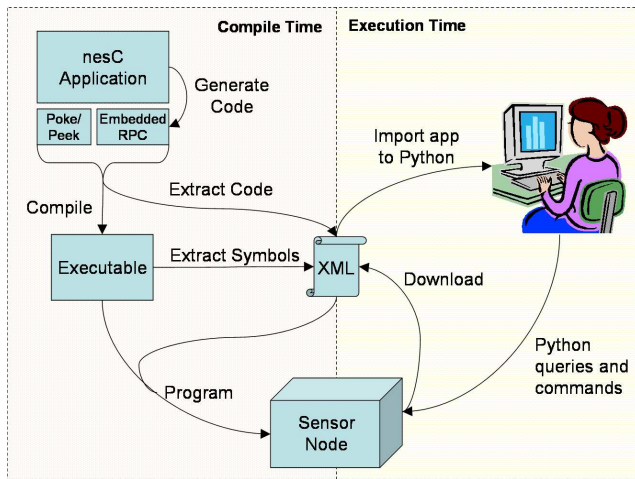


Figure 3: The structure of the application is extracted at compile time and encoded in XML. The executable and XML file are loaded onto the node. The pc-side tools import the application structure from the XML file, and can interact with the node.

function, marshals the return argument, and sends the response message. The RPC server stub constitutes the bulk of the RAM and program memory requirements of the sensor nodes by this tool suite. It requires 140 bytes of RAM for buffering packets and about 3.6KB of program memory in a large application. The minimal program memory requirements are near 1KB, and each RPC function adds approximately 100 bytes of program memory for marshaling and unmarshaling the function arguments.

4.2 Poke and Peek

Besides calling functions on the node, the user may also need to get or set variables on the heap. Similar to ERPC, we can provide this functionality such that most of the work is performed by the PC. When the embedded application is compiled, the name and type of each variable declaration is extracted from the source code, and the memory address on the heap is extracted from the symbol table of the executable. The name, type, and memory address of every variable is stored in an XML file that can be read by PC client tools.

In our system, we assume that the client and server are in the same administrative domain, in contrast to LRPC, and that the server can completely trust the client. In addition, nesC does not enforce any protection boundaries between components running on a single node. With these freedoms in mind, Marionette includes a minimalist nesC library on the node called `RamSymbolsM` that provides the ability to read and write directly to any memory address. The read and write commands are called *poke* and *peek*, following the BASIC naming convention to indicate that these are direct memory accesses and that *poke* may be a dangerous operation. Most of the work is done by the PC client, which reads the variable sizes and memory addresses from the XML file and calls the *poke* and *peek* functions on the node using the ERPC system described earlier. Because the client knows the type of the variable, it is responsible for indexing into

arrays, dereferencing pointers, and casting return arguments to the appropriate type. The nesC module on the node, in contrast, is only 60 lines of code and requires 13 bytes of RAM and about 200 bytes of program memory.

4.3 Extracting nesC Declarations

When the embedded application is compiled, the code is parsed for all nesC declarations that might be useful to the PC client tools, including enumerations, constants, data structures, typedefs, message formats, and module and interface names. The names, fields and byte alignment information of all C-structs are provided directly by the nesC compiler using tools similar to those used for MIG. We use Perl to parse the source code for enumeration, type, typedef, and nesC module and interface declarations. The data structures and enumerations are then combined to infer message formats: following the MIG convention, an AM message format is defined by struct *StructName* if an enum with the name `AM_<StructName>` is defined. All extracted nesC declarations are written to an XML file called `nescDecls.xml`, which can be read by any PC client. Extracting this information from the application eases the burden on the programmer, who no longer needs to repeat enumeration definitions in the PC client tools or to manually perform type casting or byte alignment.

4.4 PyTOS: The Python Client

The user can use a normal Python shell or script as a Marionette client simply by importing our Python libraries called PyTOS. This allows the user to access node functions and variables through an interactive, scriptable, multi-threaded, object-oriented programming environment, including rich visualization and scientific analysis libraries. The core of the PyTOS client is a parameterizable type system which reads the XML file for an application and dynamically creates new types in Python that are identical to the native types on the embedded processor. This provides type checking on the PC client tools, eg. an exception is raised if an integer-typed variable for a 16-bit processor is assigned a value outside the range `[-32,767,32,767]`. Besides the basic types, the PyTOS typing system also supports complex types such as arrays, pointers, and structs. Structs and arrays use the same byte alignment used on the embedded processor, and the sizes of pointers change based on the size of the native integer type.

The PyTOS typing system allows the PC client tools to easily serialize and deserialize data that is communicated with the sensor nodes. Typed variables can be used to dynamically construct new `TosMsg` objects, which are similar to structs except that they can be serialized and deserialized to raw bytes for sending over the network. PyTOS can automatically parse nested message structures, such as when the MAC and routing protocols each add header bytes to a packet. This functionality has long been needed in TinyOS.

Once PyTOS can generate variables and messages of the same types as those used on the sensor nodes, it reads the ERPC interface definition and makes new Python objects with functions identical to those on the node. These functions take nesC typed variables as arguments and, when called, pack the arguments appropriately and unpack the response messages. A similar procedure is performed for each variable on the heap. This set of objects provide a basis for user interaction or Python scripts that use functions and variables from the sensor nodes.

MyAppM.nc

```
provides interface foo @rpc();

command result_t foo.bar( uint8_t data, ... )
...
return SUCCESS;
```

MyApp.py

```
result = app.MyAppM.MyCommand( 101, ... )
```

Figure 4: Marionette code required to access a function on a node from the PC. This replaces all the code shown in Figures 1 and 2

4.5 Drip and Drain

Marionette requires a transport layer to pass queries and responses between the PC client tools and the network nodes. It functions in two modes: local communication or multi-hop communication. In local communication mode, the queries and responses are both simply broadcast to and from the base station node. In multi-hop communication, Marionette uses two routing protocols called *Drip* and *Drain* that were designed specifically for network management by providing simple routing with low overhead to the node [11]. Drip uses an epidemic protocol to reliably transmit query messages to the entire network. If a query message is destined to only a single node, the query message is still flooded to the entire network, but only processed by the destination node. Drain rapidly builds a spanning tree to allow all nodes to send response messages back to the base station. This tree remains indefinitely and must periodically be rebuilt as the tree quality degrades over time. By not rebuilding automatically, Drain may only consume resources when response messages are expected.

These routing protocols are designed to be extremely simple in order to have a small code footprint on the node. Because the entire network must be flooded to send a message to only a single node, however, the number of messages required to send a single request/response is unnecessarily high. In ongoing work, these protocols are being extended to allow a *session* to be opened with a small number of nodes such that query messages can more efficiently be sent to nodes with open sessions.

4.6 The Supplement

The client tools in Section 4.4 rely on the XML file that is extracted from the embedded application at compile time, as described in Section 4.3. Without the XML file, the developer cannot interact with the node. We have added functionality to Deluge, the popular network reprogramming libraries [5] so that the XML file is downloaded to the node’s external flash as a *supplement* to the actual code image. This is feasible because, even for very large applications, this file can be compressed into less than 20KB. The supplement can be retrieved by the user in an emergency or when the XML file has been lost, and is critical for long-term maintainability of sensor networks in which nodes may be running old versions of code, specialized code, or simply unknown code images.

	Oscope	Rssi	Calamari	Peg
Funtion calls	0	3	34	5
Variable writes	1	1	42	47
Variable reads	0	0	45	47
TinyOS code	12	178	537	116
PC code	6	71	392	234

Table 1: The number of times an application remotely accessed functions or variables on the node, and how much extra code was required to do this, for four existing applications

5. EVALUATION

5.1 Code Analysis

Marionette provides an architecture that shortens the development cycle and makes it easier to interact with and program sensor networks. This contribution is difficult to objectively convey, but we do quantify ease of use through two different analyses. First, we analyze the code of existing applications to identify 1) how often existing applications might access functions and variables on node applications and 2) how much code was required to do so. Second, we present several case studies to demonstrate that common tasks can be performed quickly and easily using Marionette’s Python scripting interface.

In Section 2, we distilled the minimum 37 lines of code required for each remote function or variable access. Using Marionette, these 37 lines of code are replaced by 1 line of code, as illustrated in Figure 4. The nesC module provides an interface `foo` and defines the function `foo.bar` the same way it is normally done in nesC. The user can then call this function directly through the `app` variable in a python script. The only extra code required for this functionality is the `@rpc()` tag which is added to the interface declaration. Remote access to variables does not require any extra code.

After analyzing nearly a dozen existing applications available in the TinyOS code repository, we found that applications vary significantly in how many message protocols they define for communication between the node and the PC. The applications that use software tools such as MIG, DiagMsg, Config, and Command that reduce the required amount of hand-written code necessary to perform a remote function or variable access, as discussed in Section 2, tended to have an order of magnitude more such accesses than applications that do not use these tools. This demonstrates that the amount of code required to perform remote accesses is a real barrier to doing so. The number of remote function and variables accesses for several applications, and the number of lines of code required to provide them, are summarized in Table 1.

Calamari and *Peg*, which are distributed localization and tracking algorithms respectively, had the most remote accesses and used all of the tools listed above. Both applications implemented over 100 remote accesses in less than 1000 lines of hand-written code, although the amount of automatically generated code exceeds 3000 lines for both applications. Not coincidentally, these applications both have similar properties which make development and debugging particularly difficult: 1) they process real sensor data, meaning that they must be developed and debugged on the nodes

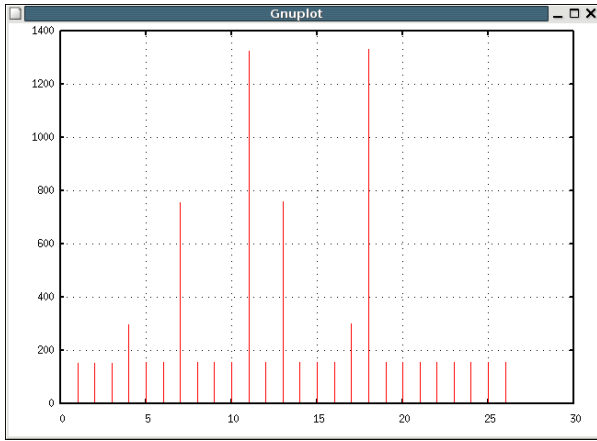


Figure 5: This graph shows the most recent number of messages that were forwarded by each node in a routing tree built on a 28 node testbed.

themselves, not in a simulator 2) they perform distributed computations in the network, meaning that they must be developed and debugged as a network, not as a single node.

ChipconRssi is an application that commands each node in the network in turn to send a certain number of radio messages while all other nodes record the received signal strength. When all messages have been sent, the application reliably queries each node for the recorded information, repeating requests for data that has been lost. This application requires access to only three functions and one variable, which were provided with approximately 250 lines of hand-written code. The number of lines required can be different from 37 when extra marshaling or serialization is required, when a single protocol is reused for multiple purposes, or when the protocol is not a complete round trip. For example, the *Oscilloscope* application that we explored in Section 3 sets only a single variable with only 18 lines of code because it does not expect a return variable.

Reducing the amount of hand-written code needed to remotely access the program, however, is only part of the benefit of using Marionette. For example, the total lines of node/pc code (respectively) in the *ChipconRssi* application would be reduced from 502/437 to 324/366 by simply removing overhead for remote accesses. When we re-implemented this application in the Marionette framework, however, we found a reduction to 279/134 lines of code instead. We also re-implemented a reliable bulk data collection library called *Straw* that was used for the structural health monitoring [6], and saw a reduction from 515/396 total lines of code to 136/48. That is, an application of over 900 lines of code was reduced to less than 200 lines of code. This reduction in total application size of almost 80% illustrates that Marionette does more than remove the need to hand write code for remote access to functions and variables. By removing the barrier between the node and the PC, it promotes simpler application architectures.

5.2 Case Studies

Marionette does more than reduce code size, it also allows the user to create new functionality for the sensor network without reprogramming the nodes. In this section, we demonstrate simple Python scripts that create network

functionality for which the nodes were not originally programmed. These examples are very simple due to space constraints, although we do show an example in each of three key areas: 1) network health monitoring 2) debugging and 3) scripting new functionality.

5.2.1 Monitoring Traffic Patterns

Surge is a multi-hop data collection application in which each node in the network reads a sensor value and routes it back to the base station through a routing tree. In such applications, the routing nodes near the root of the tree may send many times more packets than the leaf nodes, and the user may want to monitor this difference. This can be achieved by monitoring a state variable called `forwardPackets` in the `DrainM` software module, which keeps track of the number of packets sent.

```
forwardPackets = [0 for i in range(28)]
while True:
    responses = app.DrainM.forwardPackets.peek(timeout=60)
    for response in responses:
        forwardPackets[response.sourceAddress] = response.value
    gplt.plot(forwardPackets, 'notitle with impulses')
```

The script above illustrates how a user could script this functionality in 6 lines of Python code using Marionette. Every 60 seconds, it queries all nodes for the number of packets forwarded so far, and updates a plot. We ran this script on a testbed of 28 nodes and generated the graph shown in Figure 5, which shows the number of packets sent by each of the 27 nodes in the network besides the base station. This graph shows that the routing tree was four levels deep, and that the two nodes in the first level forwarded almost 8 times more packets than the 21 leaf nodes.

5.2.2 Stress Testing

Tree based routing algorithms can be very susceptible to failing nodes, especially if those nodes are near the root of the tree. A programmer that is designing a new routing algorithm may want to deliberately remove nodes from the network that are near the root of the tree in order to stress test the algorithm. Clearly, this functionality should not be built into the nodes themselves. Instead we present below an 11 line script that continually plots the total number of messages received, and repeatedly removes the node that is closest to the root of the routing tree.

```
numReceived=[]
while True:
    responses = app.DrainM.forwardPackets.peek(timeout=60)
    numReceived.append( len(responses) )
    for i in range( len( responses) ):
        if response[i].value > 50:
            responses = app.DrainM.forwardPackets.poke(0)
            app.DrainM.StdControl.stop( address=i )
            print "Killed node %d" % i
            break
    gplt.plot(numReceived)
```

The script keeps track of the number of messages received each minute in the `numReceived` array and plots this value over time. Once every 60 seconds, the script asks all nodes for an update of the number of forwarded packets and removes the first one that has forwarded more than 50 packets. It notifies the user that a node has been removed and updates the graph showing how many packets have been received.

5.2.3 Motion Detector

In a recent deployment, we deployed a large number of sensor nodes in a field with Passive Infrared sensors (PIR) that could detect moving objects. However, it was unclear how to convert the output of the sensors into reliable detection events. In the script below, we cause the nodes to beep every time a filtered PIR value exceeds some threshold. This script could be used to quickly and easily test for reasonable threshold values or to experiment with more complicated filters without reprogramming the nodes.

```
while True:
    responses = app.PIRFilter.value.peek(timeout=10)
    app.SounderM.StdControl.stop( )
    for i in range( len( responses ) ):
        if response[i].value > 300:
            app.SounderM.StdControl.start( address=i )
```

This script simply queries each node for its filtered PIR value every fifteen seconds. If any of the nodes have a value greater than a threshold, the node is commanded to turn on its sounder until the next update is received.

5.3 Costs and Limitations

The most significant limitation to Marionette is the requirement to remotely access a node's functions or variables over a wireless network. Increased traffic not only translates to energy consumption, but can also adversely affect the behavior of the network algorithm that is being developed or debugged. Furthermore, because of the unreliable network, the Drip and Drain protocols cannot guarantee against latency or loss, which may lead to unforeseen effects on the application such as unsynchronized behavior between nodes. While Marionette provides a full, modern development environment for the aspects of the application running on the PC, complete debugging functionality such as break points, variable watches, and stack traces are not possible for the aspects running locally on the nodes. Finally, Marionette cannot read the variable at any arbitrary point during program execution but can only read variables in nesC's *task* context. This is not a fundamental limit of PyTOS or nesC, but a limit of the current implementation which may be overcome in future work.

6. DISCUSSION

To maximize efficiency during deployment, sensor nodes do not support most development and debugging techniques. Typically, two separate programs are executed on the PC and the sensor nodes, loosely coupled through a messaging interface. The sensor node provides no other programmatic interface. This opaque, minimalist interface, however, does not provide adequate visibility and control for practical debugging and development. Marionette uses Embedded RPC to tightly couple the executables on the PC and the node, to the extent that parts of the sensor network application are actually executed on the PC client, remotely accessing functions and variables on the node only when necessary. This approach consumes more bandwidth than batch programming, but provides the programmer with increased visibility and control during execution.

This approach is similar to that taken by EM*, which defines an interface to the sensor node's hardware which can be used over a wired testbed by an application running centrally on a PC [4]. With Marionette, however, this interface is very flexible, allowing the user to choose which functionality runs on the PC and which runs on the sensor nodes,

while EM* defines the interface just above the hardware level. Marionette can therefore be used on wired testbeds like EM* or, by migrating more complex operations onto the nodes, can also be used at an efficiency point suitable for multi-hop, wireless networks.

In a way, Marionette is also similar to macro-programming approaches such as TinyDB and Region Streams [9, 10], and Virtual machines such as VM*, Mate, and SensorWare [3, 7, 8], which similarly address the challenges of developing sensor applications. These approaches install a number of libraries on the sensor nodes which can be composed by scripts that are run by a virtual execution layer on the node. The main difference in Marionette is that this script is executed on the PC instead of the node, which allows the user allows the user to use any standard programming environment such as an interpreter or web interface, and all standard debugging and development techniques. A second difference with these other techniques is that Marionette hooks into a normal application running on the node; programmer does not need to format functions or algorithms into special libraries in order to make them composable. This enables an incremental type of development in which each iteration of the node application is compiled to native code so that the final application does not have to pay the overhead of running through a virtual execution layer on the node.

Acknowledgment

Special thanks to Shawn Schaffert for help starting PyTOS.

7. REFERENCES

- [1] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *TOCS*, 1990.
- [2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *TOCS*, 1984.
- [3] A. Boulis, C. C. Han, and M. B. Srivastava. Design and implementation of a framework for programmable and efficient sensor networks. *MobiSys*, 2003.
- [4] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *USENIX*, 2004.
- [5] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SensSys'04*.
- [6] S. Kim. Wireless Sensor Networks for Structural Health Monitoring. Master's thesis, University of California at Berkeley, March 2005.
- [7] J. Koshy and R. Pandey. VM*: synthesizing scalable runtime environments for sensor networks. *SensSys '05*.
- [8] P. Levis and D. Culler. Mate: a Virtual Machine for Tiny Networked Sensors. In *ASPLOS*, October 2002.
- [9] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, December 2002.
- [10] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *DMSN '04*.
- [11] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *EWSN*, January 2005.