Rapid Prototyping Your Sensor Network

David Chu*, Feng Zhao†, Jie Liu†, Michel Goraczko†,
*EECS Computer Science Division, University of California, Berkeley, CA 94720
Email: davidchu@cs.berkeley.edu
†Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052
Email: {zhao, liuj, michelg}@microsoft.com

Abstract— **Several considerable impediments stand in the way of sensor network prototype applications that wish to realize sustained deployments. These are: scale, longevity, data of interest, and infrastructure integration. We present a tool, Que, which assist those sensor network deployments transitioning from prototypes to early production environments by addressing these issues. Que is able to simulate realistic deployments with faithful data, provide fast and iterative feedback on operations, and compose applications quickly in a platform-independent manner. We demonstrate Que's applicability via tests against our new data center temperature-monitoring deployment, DataCenter.NET.**

I. Introduction

Sensor networks are notoriously difficult to build, deploy and maintain. Early sensor network experiences are not without case studies of deployment that have failed to mature or taken considerably longer to arrive at fruition than originally anticipated.

For example, several geologists, excited by the new science that sensor networks might bring to their field, targeted an initial test deployment in a modest desert cave to collect climatological data. They purchased a packaged sensor network product from a major sensor networking company. The package was billed as the most straightforward off-the-shelf solution offered, so their realistic expectations were that such a system would last several months given the energy provisions once deployed.

Unfortunately, the experiences were not encouraging. After spending several days in the field trying to determine why the product failed to deliver results, the geologists finally established connectivity and collected data for two hours before the product failed permanently. Disillusioned, these users have since reconsidered their sensor network efforts. While the brief two hours of data were beneficial, the costs were very significant [18].

What can we do to remedy this lack of sensor network usability? Let us pursue this question by first examining the development model surrounding sensor network deployments today. Sensor network deployment efforts typically follow a 4-step procedure:

1) *Goals and requirements specifications*
2) *Prototype deployment*
3) *Prototype to production transition*
4) *Production deployment*

Many scenarios are easy to prototype, but difficult to achieve production standards. This indicates that there are significant disparities in the production requirements that are unaccounted for in the prototype phase.

We have built *Que*, a tool which provides a unified scripting and simulation/emulation framework for multi-tier sensor network systems. Que assists the transition from prototype to production by permitting fast iterations on testing system inputs, outputs and whole-system assembly.

Several important factors influence why this transition is not straight forward, and where a tool like Que might provide assistance. The distinctions are primarily in the following dimensions:

Scale: In the prototype phase, it's important to get something working quickly. This often means over-instrumentation with dense of arrays of sensors in a limited area rather than fine-tuning. However, in the production phase, the cost of ownership prohibits over-instrumentation while simultaneously driving scale upward. Thus, determining the minimum density of sensors is a necessary yet often unanswered question.

Longevity: In the prototype phase, the sensornet does not have to be long-lived nor particularly reliable. In production, lifetime and manageability requirements are dominant concerns. Frequently, longevity issues such as minimum sampling interval and duty-cycling are deferred until production deployment, an expensive phase in which to address a fundamental requirement of the sensor system. A short-lived sensing system is often simply not useful [4].

Data: In the prototype phase, raw data is useful especially for exploration. In production, distilled decision making information is most important. Thus, data processing operations which were not present in the prototype must be introduced in production. Furthermore, the operational dynamics are further complicated in the case of online or in-network data processing. Thus, it is important to test with realistic data input and control logic in prototype phase.

Integration: Integration with rest of infrastructure pyramid is not a priority during prototyping. However, realistic production systems often involve many elements in addition to the sensornet. Traditional sensornet development lacks such multi-tier systems integration testing.

The goal of Que is to help answer these question through a combination of two primary mechanisms. First, Que offers a script-based exploration environment for system assembly across multiple tiers. This makes it easy for developers to retask their system for new data processing either on mote- or microserver-class devices. Second, Que provides an easy to use yet flexible simulation and emulation environment for an entire tiered system, so that changes in system scripts can be tested quickly against realistic scenarios. The combination of these two mechanisms lends naturally to rich system and data exploration, which are important in the transition from prototype to production.

To validate our approach, we have applied Que to DataCenter.NET, an entirely new deployment at Microsoft Research with significant potential impact. A significant problem in the modern computer data center is the lack of visibility into power-intensive cooling support systems. DataCenter.NET is a compelling application that assists data center operators with real-time high-granularity feedback into operational metrics such as facilities and server temperatures. As we worked toward a real production deployment with our operations colleagues, we realized that DataCenter.NET required addressing all of the key issues mentioned above: scale, longevity, data and integration, providing a great testing environment for Que.

The next section describes related work. Section III discusses the design principles that drove Que. Section IV introduces the Que environment. Section V describes the system architecture. Section VI and VII discusses our deployment DataCenter.NET and our results in using Que to bring this system to production. Finally, section VIII presents discussion and conclusions.

II. Related Work

Many have addressed the difficulty of deploying sensor networks with "out-of-the-box" sensor network solutions [3]. However, this commoditization does not address the inherent customization necessary for many sensor network scenarios. This means that out-of-the-box solutions are only valid for a narrow, pre-defined segment of the sensornet space.

As a result, a number of efforts provide design and testing support for sensornet application development. [5], [8] both provide two-tier simulation environments. These are similar in spirit to Que though only Que supports the MSRSense platform. Que, like [8], is also able to leverage a user-defined data-source to provide high-fidelity emulation capabilities.

Que's "programming by component wiring" is used in many embedded systems [5], [9], [10], [12], [17]. Such an approach is fitting when well-defined pieces of functionality are reused often. Our scripting is similar in spirit to that proposed in [6], [19]. Whereas [19] uses scripting for debugging and [6] uses scripting for data exploration, Que employs this approach in whole system development, as well as data exploration. In fact, Que offers a convenient bridge to the server-side data manipulation operators offered by [6].

III. Design Goals

The Que functional interface is meant to be an extremely simple yet sufficiently flexible for component wiring and simulation execution.

Simplicity: Que does not provide yet another programming approach to sensor networks. Rather, Que directly provides the intermediate component wiring language while other languages (*e.g.,* C) provide component implementations. It has been argued that restricted coordination languages fit well for constructing systems when component boundaries are well-defined [15]. Indeed, Que users benefit from the safety and simplicity of the language restrictions, yet retain the ability to create new components in native systems languages. This programming paradigm is common in embedded systems [6], [9], [10], [17].

Leverage existing libraries: The lack of full node programmability means that Que relies on others to provide the bulk of component implementations. By default, Que interfaces with three such component libraries: MSRSense [17], TinyOS [10] and Tinker [6] and additionally offers a general adapter to integrate with other component libraries.

Flexibility: Que exposes a Python-like shell for convenient interaction. We utilize it as a flexible platform from which to perform inter-component wiring, sensor network to system integration, and data analysis.

There are some associated limitations with this model as well. Que is best suited for component-based programming. This implies establishment of well-defined component interfaces and libraries. Development of new device drivers for example must still be done in native environments.

object or function	*description*
toslib	TinyOS operator library
mslib	MSR Sense operator library
create	Instantiate a new operator from platform-specific library
link	Bind an input port of operator with the output port of another operator
openlocal	Initialize the network as set of nodes connected to the localhost
run	Simulate the execution of the entire operator graph, possibly by generating source and configuration files, compiling, and simulating.
writecache	Persist simulated results
readcache	Restore simulated results from cache

Fig. 1: Default user objects and functions in Que.

IV. Example User Session

Next we illustrate a user's interaction with Que via an example session. Figure 1 summarizes the main objects and functions of interest in a typical usage scenario. This example session creates and executes the operator graph shown in Figure 2. The operator graph spans both TinyOS motes and MSRSense microservers.

Listing 1 illustrates querying a standard Que object. This particular object, `toslib`, is a factory for creating TinyOS-specific objects some of which are shown in the listing.

The user is able to instantiate components from this factory from the interactive shell. Listing 2 shows the instantiation and linking of several components from this library (lines 2-8 and lines 9-17 respectively). The two important functions above are the operator library `create` call and the `link` call. The `create` call instantiates new operators from the platform-specific operator library `toslib`. The link call binds the output

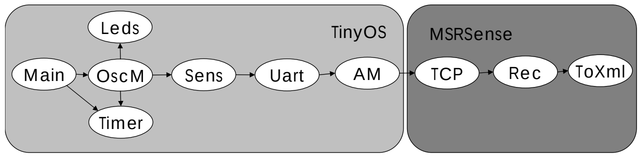


Fig. 2: Operator graph demonstrated in our running example. This corresponds to the prototypical Oscilloscope application. Its function is to periodically send all sensors' measurements over the serial port to the microserver for simple XML canonicalization.

```
>>> toslib.info()
Tinyos Library
        Catalog:
        "apps/Blink/BlinkM"
        "apps/Blink/SingleTimer"
        "apps/BlinkTask/BlinkTaskM"
        "apps/BlinkTask/SingleTimer"
        "apps/CntToLeds/CntToLeds"
        "apps/CntToLedsAndRfm/CntToLedsAndRfm"
        "apps/CntToRfm/CntToRfm"
        "apps/CountRadio/CountDualAckM"
        "apps/CountRadio/CountDualC"
        "apps/CountRadio/CountDualM"
        "apps/CountRadio/CountReceiveC"
        "apps/CountRadio/CountReceiveM"
                ...
```

Listing 1: Querying the standard TinyOS library for available components.

port of one operator to the input port of another operator. There is also a convenient function unlink provided to unlink an object. For example, the Main component and the OscilloscopeM components are linked together through the StdControl interface (line 9). Parameterized interfaces, introduced in TinyOS, are also supported (*e.g.,* line 11).

Listing 3 shows that it is possible to retrieve information about the wirings of an instantiated component. The exposed interfaces for the instantiated component along with the currently wired component are displayed. Some interfaces indicate <NULL> to show that they have not been wired yet (*e.g.,* line 8). Similarly, other interfaces clearly indicate wiring to parameterized component interfaces (line 10).

Que provides integrated support for both TinyOS and MSRSense. MSRSense is a .NET-based compenentized sensor network microserver. Listing 4 shows the instantiation of MSRSense components as Que operators. This is purposely exposed in a similar fashion to the TinyOS operator library above. This uniform support for cross-platform operator composition is where Que facilitates system assembly.

Cross-platform wiring such as MSRSense component to TinyOS component wiring is also permitted. Listing 5 shows the binding of ports between operators of different platforms. In particular, the special operators opamp and oppor, serve as conduits through which communication occurs between the two platforms. Que identifies and appropriately handles this case, as discussed in Section V.

```
1  # create tinyos component graph
2  op_man = toslib.create('tos/system/Main')
3  op_osc = toslib.create('apps/Oscilloscope/OscilloscopeM')
4  op_sen = toslib
5     .create('tos/sensorboards/basicsb/DemoSensorC')
6  op_led = toslib.create('tos/system/LedsC')
7  op_tim = toslib.create('tos/system/TimerC')
8  op_com = toslib.create('tos/system/UARTComm')
9  link(op_man, 'StdControl', op_osc, 'StdControl')
10 link(op_man, 'StdControl', op_tim, 'StdControl')
11 link(op_osc, 'Timer', op_tim, 'Timer[unique("Timer")]')
12 link(op_osc, 'Leds', op_led, 'Leds')
13 link(op_osc, 'SensorControl', op_sen, 'StdControl')
14 link(op_osc, 'ADC', op_sen, 'ADC')
15 link(op_osc, 'CommControl', op_com, 'Control')
16 link(op_osc, 'ResetCounterMsg', op_com,
       'ReceiveMsg[AM_OSCOPEMSG]')
17 link(op_osc, 'DataMsg', op_com, 'SendMsg[AM_OSCOPEMSG]')
```

Listing 2: Instantiating and linking TinyOS components from the component factory.

```
1  >>> op_osc.info()
2  OscilloscopeM_0 [OscilloscopeM]
3          <-StdControl
4                  Main_0.StdControl
5          ->Leds
6                  LedsC_0.Leds
7          ->CommControl
8                  <NULL>
9          ->Timer
10                 TimerC_0.Timer[unique("Timer")]
11         ->DataMsg
12                 <NULL>
13         ->ADC
14                 DemoSensorC_0.ADC
15         ->SensorControl
16                 DemoSensorC_0.StdControl
17         ->ResetCounterMsg
18                 <NULL>
```

Listing 3: Querying for more detailed linkage information of a particular component.

```
# create microserver component graph
op_tpr = mslib.create('ComplexTOSPacketReceiver')
op_tpr.setparam('messageType', 'ArrayOscopeMsg')
op_d2x = mslib.create('DataToXml')
link(op_tpr, 'output', op_d2x, 'input')
```

Listing 4: Instantiating and linking MSRSense microserver components.

```
# port bindings
op_amp = toslib.createAMPort()
op_por = mslib.createTcpPort()
link(op_com, 'SendMsg', op_amp, '10')
link(op_amp, '10', op_por, '9002')
link(op_por, '6001', op_tpr, 'input')
```

Listing 5: Establishing linkages between TinyOS and MSRSense platforms.

```
emusrc1 = emulator.DataCenterEmulator(connstr)
net = openlocal(op_amp, op_por, emusrc=emusrc1)
results = run(net, time=60*10, appname='Oscilloscope',
      dosrcgen=True, docompile=False, dosimulate=True)
```

Listing 6: Simulation execution of an operator graph across heterogeneous network nodes.

In line with one of Que's primary objectives, heterogeneous networks are very easy to setup, simulate, modify and re-simulate. Indeed, the first step toward this goal has already been presented: the unified component wiring across heterogeneous platforms. Listing 6 shows the second contribution toward this objective. The overall goal is to simulate the operator graph consisting of heterogeneous elements. We next explain these three important commands in detail.

First in line 1 of Listing 6, the user chooses an appropriate simulator for her concerns. The emulator.DataCenterEmulator in particular draws ADC values from traces collected in our new deployment which we describe subsequently in Section VI. Section V describes more about possible emulators.

Second in line 2, the openlocal initializes the network topology with a minimal of user intervention. Our ease of use criterion means that the user can either choose a predefined network or can query a preexisting network for its parameters[1]. In addition, openlocal accepts chains of operators and binds these to the nodes initialized in the network.

Third in line 3, the run simulates the given network, in conjunction with the particular operator graph and sensor inputs. The heavy lifting underlying this command will be explained in Section V. The goal is to provide a very minimal interface through which the details of the simulation are abstracted, but the results are not. At the end of run, the results are brought from the particular platform-specific simulations into the Que environment. The results are naturally emitted by the endpoint(s) of the directed operator graph. For our running example, the results are at the MSRSense component op_d2x.

The preceding three commands, and particularly the last one, are heavy-weight, yet present simple interfaces for simplicity of use. Yet these allow full flexibility for exercising a custom operator graph on a custom network topology with a custom simulation data source.

A. Ad-Hoc Data Processing

By default, simulation results are returned as a sequence of arrays, one for every message from the terminal operator in the graph. There is some parsing required on the part of the users for their specific operator formats. However, once this format standardization is made, some very simple processing can result in very fast turnaround time for getting initial results. For example, Que includes several standard utility functions for quickly displaying standard values of interest.

Listing 7 shows the use of several of these functions. plotresults and plotcorrs generate scatter and topographical plots respectively. ewma computes a tunable exponential weighted moving average that is often useful in real-world data cleaning. In addition, the wealth of native python libraries is often a benefit for our scripting environment; corrcoef is a built-in python function that computes correlation coefficients. For example, Que can interface to the Tinker and Matlab-like matplotlib Python tools in order to apply more standard data operations [6], [11].

[1]The latter option is not yet implemented.

```
# customized result parsing to extract interesting fields
z = customtranslate(results)

# results plotting
plotresults(z)

# correlations
cc = corrcoef(z)
plotcorrs(cc)

# ewma: cleaning
ewmaz = ewma(z)
plotresults(z,ewmaz)
```

Listing 7: Standard result manipulation functions for rapid data analysis.

```
writecache(results)
results = readcache()
```

Listing 8: Caching intermediate simulation results for future data processing.

B. Additional Functionality

Two very simple functions are provided for caching Que emulator results to avoid the heavy-weight acts of rerunning the entire simulation in the case that the operator graphs or the emulation inputs do not change. These are shown in Listing 8.

V. System Architecture

Our architecture, shown in Figure 3, consists of several major components. We next discuss the mechanics of each.

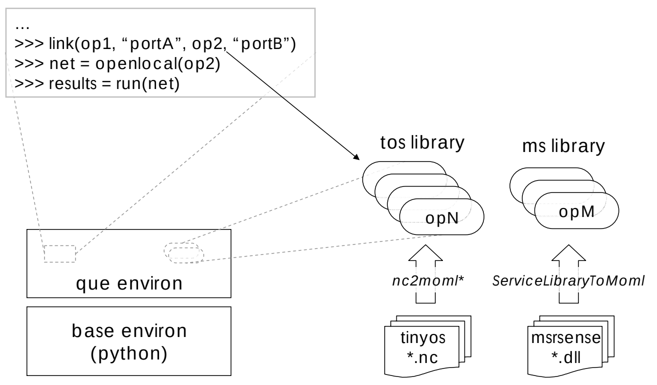
A. Operator Libraries

Operator libraries permit the creation of operators for manipulation in Que. There is an operator library per platform which subclasses oplib. Often these libraries correspond directly to existing software libraries available on the corresponding platforms. For example, the TinyOS and MSRSense platforms both contain a fair number of components in their distributions. In order to expose these platform-specific elements as operators in Que, we provide platform-specific *interface extractors* as illustrated in Figure 3a. For TinyOS and MSRSense, this functionality is provided by the tools nc2moml and ServiceLibrary2Moml respectively. After instantiation from a platform-specific library, all operators behave consistently, resulting in a uniform user experience.

New platforms are straightforward to expose to Que. The only requirements are to subclass oplib for the platform's operator library and populate the library with a platform-specific interface extractor tool.

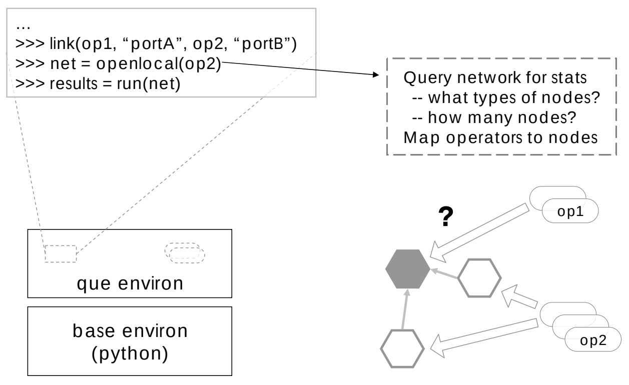
The goal of a platform-specific interface extractor is to generate operator interface descriptor files which are used by operator library subclasses. We have adopted a variant of the Ptolemy2 standard MOML interface [14].

Listing 9 shows an example interface descriptor. The key elements of this interface are the exposition of named input and output ports and operator parameters. We have found that the two platforms we tested offer fairly natural mappings to



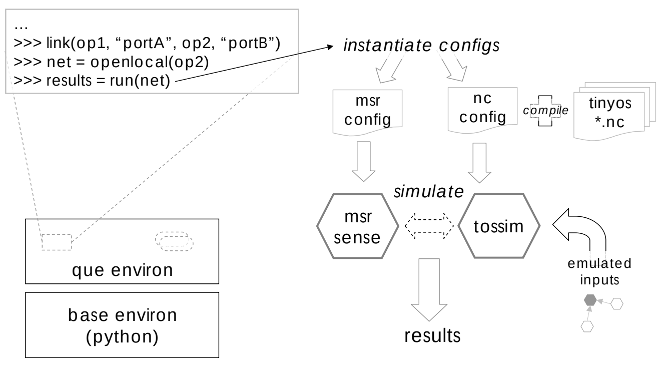
```
...
>>> link(op1, "portA", op2, "portB")
>>> net = openlocal(op2)
>>> results = run(net)
```

(a) Unified operator composition from interfaces of platform-specific components.



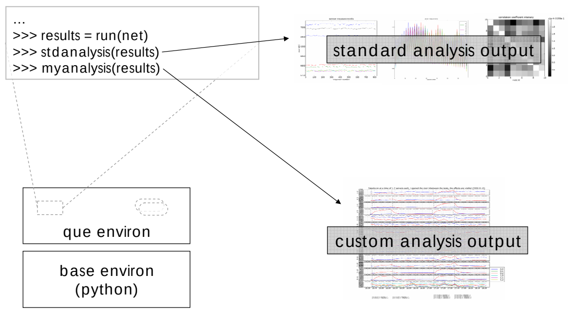
```
...
>>> link(op1, "portA", op2, "portB")
>>> net = openlocal(op2)
>>> results = run(net)
```

(b) Creation of a network object based upon a static network configuration or querying of a live network. The operator graph is assigned to platform-specific network nodes.



```
...
>>> link(op1, "portA", op2, "portB")
>>> net = openlocal(op2)
>>> results = run(net)
```

(c) The operator graph is run. This involves instantiating operators (possibly involving compilation) and invoking platform-specific simulation environments. Results are retrieved back into the Que environment.



```
...
>>> results = run(net)
>>> stdanalysis(results)
>>> myanalysis(results)
```

(d) The results are fed into standard analysis and visualization tools. In addition, the user has very flexible options for scripting her own post-processing.

Fig. 3: The Que Architecture

this interface. MSRSense input and output ports map directly to MOML input and output ports; TinyOS uses and provides interfaces correspond to input and output ports respectively. In addition, to support NesC-style interface parameterization, input and output ports are permitted to be parameterized, such that a single port proxies for a number of instances of the port determined at compile time.[2]

B. Network Libraries

The network library provides the network abstraction for the user. Subclasses of netlib define a set of heterogeneous nodes and the interconnecting network. For example, a subclassed network object may correspond to a predefined static set of nodes, a set chosen from an asset catalog, or a dynamic set established from querying an online prototype network. Currently, we provide a subclass that supports a predefined static set.

Another key function of the network object is to pin operators to nodes. As illustrated in Figure 3b, this determines the mapping of what operators each node runs. Typically this assignment proceeds by associating platform-specific operators with the nodes on which they are capable of running. At present, every operator is targeted for only one platform so the mapping is straightforward. However, cross-platform operators are also possible (*e.g.,* with virtualization or platform-independent implementations). These then permit variable operator placement informed by metrics such as computational speed, energy and sensitivity to network loss. Furthermore, they open the possibility of dynamic operator placement optimization.

[2]Note that MOML parameters are distinct from parameterized ports. MOML parameters are more akin to NesC generics [7].

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE plot >
<class name="EWMA"
  extends="ptolemy.domains.ptinyos.lib.NCComponent">
  <port name="input" class="ptolemy.actor.IOPort">
    <property name="input" />
  </port>
  <port name="output" class="ptolemy.actor.IOPort">
    <property name="output" />
  </port>
  <parameter name="windowSize" />
</class>
```

Listing 9: MOML standardized interface definition example. This example shows the interface definition for the microserver exponential weighted moving average component.

C. Simulator

The heart of Que is the heterogeneous network simulator. The simulator is initiated with the `run` command. As shown in Figure 3c, the simulator executes the following sequence of operations:

Operator configuration: The simulator first generates platform-specific configuration files from operator graph specifications given as input. For example, for TinyOS, the simulator generates NesC component wirings. For MSRSense, the simulator generates XML operator configuration files.

Binary compilation: The simulator then enacts platform-specific compilation for the configured system. This possibly involves multiple compilations for multiple platforms.

Native execution: The simulator next executes the compiled operator graphs in low-level native platform-specific simulators. The TOSSIM simulator is used for TinyOS binaries [16]. Since MSRSense microserver is already contained within the .NET virtual machine, it is natively executed. Also, the simulator draws data inputs from its user-specified data source for either preset, trace-driven, or emulated sensor readings. This provides for a customizable degree of fidelity. We highlight that similar emulator drivers can also be provided for the network.

Channel establishment: A myriad of communication channels are needed for interoperability in a mixed environment of heterogeneous platforms. For instance, appropriate connection bindings are needed between the MSRSense runtime and Serial Forwarder, a standard TinyOS communication channel, in order to achieve heterogeneous network simulation. As another example, data input from the user-specified data source also needs to be connected with the simulator. The Que simulator establishes all of these channels and extra plumbing on the user's behalf when the `run` command is invoked.

At the conclusion of this process, the operator graph is transformed into a set of results over the specified network and data source. These results are populated back into the Que environment as arrays.

D. Analysis tools

The standard analysis tools provide helpful first-level diagnostics that go toward answering the general prototype to production questions. These tools are: visualizing the resulting output for each node; calculating and visualizing the correlation map for the nodes of interest; and performing basic data cleaning of the resulting data. Examples of there application are shown in Figure 3d. These are exposed as additional user scripts callable from Que. Likewise, we are able to readily adopt Tinker and Matlab-like matplotlib builtin tools [6], [11].

VI. DataCenter.NET Deployment

A. The Problem

We focused the use of Que in a particular deployment, DataCenter.NET. The goal of DataCenter.NET is to reduce energy costs in the computer data center. The typical data center is an intense environment consisting of thousands to tens of thousands of physical compute and storage servers, arranged in vertical racks. There are typically several dozen servers per rack, and several hundreds to thousands of racks per data center. This density of servers creates two compounding problems. First, the servers require an intense amount of power to run. Hundreds of watts per square foot is not uncommon. Second, due to the great density of power in a very confined amount of space, there is an immense cooling requirement placed on the data center facilities environment. The Heating, Ventilation and Air Conditioning (HVAC) energy expenditure is a sizable fraction of the overall facilities energy expenditure. Therefore, both are significant sources of energy consumption, and hence present significant opportunities for energy reduction.

Currently, the core data center client industries (e.g. Internet services, health care services, etc.) are growing rapidly every year. Subsequently, the development of new data centers is also occurring extremely aggressively. According to [1], a fully loaded 24 Megawatt data center costs about 20 million USD per year in electricity. The cost of powering the HVAC in such a fully utilized data center can be as high as 5 million USD per year. With such an enormous energy appetite, data centers are even drawing political attention; the US Congress recently passed a bill asking for investigations into how to reduce data center energy consumption [2].

Unfortunately, data center managers have relatively scarce information on which to base facilities HVAC decisions. Traditional thermostats are generally deployed at a very coarse grain, with one thermostat canvassing several thousand servers. This means that HVAC settings are naturally adjusted to local phenomenon first, and only slowly adapt to global temperature changes. Since there is a hard requirement to run all machines under certain machine-specific temperatures or else risk overheating and hardware failure, facilities managers are loathe to experiment aggressively with new thermostat settings. Unfortunately, zeroing in on the right temperature setting is exactly a key factor in saving data center energy consumption.

Further compounding the problem, data center operators often have little visibility into future request loads that are being executed by data center clients. In addition, each rack is configured to contain a mix of varied processing and storage elements, all of which exhibit different workloads. This leads to unpredictable fluctuations in the space of optimal HVAC settings over time.

B. *Our Approach*

To tackle this problem, we worked with data center managers to develop a wireless network for temperature sensing. Wireless sensors are a suitable fit for this scenario for several reasons. First, the wireless sensors can be deployed incrementally and flexibly. This is important for gradual rollout and avoiding high upfront fixed costs for traditional thermostats. Second, wireless sensors can cover a very fine-grained spatial setting, and this density can be flexibly chosen and reconfigured.

With detailed temperature heat maps, data facilities managers are able to make more informed decisions affecting data center operations. First, managers gain visibility into better ways to design facilities, such as where to optimally place new racks and improve HVAC distribution systems. Second, managers and the server's users can control job scheduling better so as to not only take into account server load, but also heat displacement effects. With a flexible job allocation mechanism such as virtualization, we might even apply optimization algorithms to job placement.

Commissioned with these high level goals, we proceeded to build a modest prototype data center before embarking on an actual deployment. Our lab prototype server racks, DataCenter.NET, contains 14 servers arranged in racks of 5, 5 and 4 servers each. They are located in a very small 10 ft by 15 ft contained testing environment. We fully instrumented each server with a wireless temperature sensor mote near the front intake fan, and a mote near the back exhaust fan. Similarly, we deployed 6 ambient temperature sensors along the ceiling in a grid arrangement. Along with a base station to transmit all the data, this formed a 35 mote deployment. Figure 4 shows the components of this setup.

C. *Using Que in DataCenter.NET*

DataCenter.NET is a fitting scenario in which to test Que. In fact, DataCenter.NET highlights the importance of each of the areas of concern when transitioning from prototype to production which we previously outlined in Section I:

Scale: Our initial prototype consists of 35 motes deployed on 3 racks. However, we are facing a massive scale-up to tens of thousand racks and a proportional increase in the number of wireless sensors.

Longevity: Energy requirements are not initially a concern. However, as we transition to production, battery replacement becomes an increasingly important concern. In particular, the number radio messages sent, an energy intensive operation, becomes important to monitor.

Data: Our prototype is capable of delivering all of the data to the end users. However, facilities managers are only interested in faithful temperature trends as opposed to noisy and lossy raw readings.

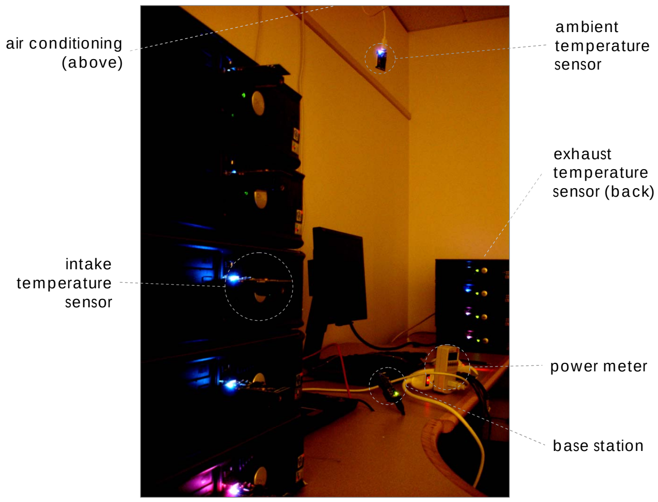

air conditioning (above)
ambient temperature sensor
exhaust temperature sensor (back)
intake temperature sensor
power meter
base station

Fig. 4: The DataCenter.NET lab prototype server racks consist of 35 wireless sensors placed on the front and rear of 15 servers and on the ceiling. The servers compromise 3 racks, two of which are visible in the foreground and background here.

Integration: Lastly, a wireless sensing system is but one part of many tools for facility managers. This must integrate cleanly with their other preexisting tools and infrastructure.

In the Section VII, we address how Que answers these questions we had about our deployment.

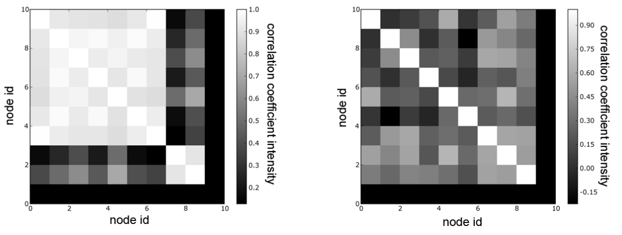
Presently, we illustrate some example temperature traces in Figures 5 and 6. Figure 5 displays three subfigures 5a, 5b and 5c, each of which corresponds to a rack of machines. For example, Subfigure 5a corresponds to five machines Server A, B, C, D and E whose physical arrangement corresponds to the vertical ordering of their plots. For each machine, the red plot indicates the exhaust temperature measurements and the blue plot indicates the intake temperature measurements across time.

Figure 5 illustrates a scenario in which a single machine, Server A, is turned on at the start of the experiment at 9:12 PM. Server A's intake quickly cools while its exhaust quickly rises. The *exhaust* of machines above *decreases*, which is due to the impact of moving, albeit hot, air from the exhaust of Server A. Moreover, the *intake* of machines in Rack 2 *increases* noticeably faster than that of the other monitored locations. Clearly this is a cause for concern since hot intakes are not optimal for cooling. This interplay Server A's exhaust on Rack 2's intake is an unexpected result and reminds us of the unpredictable nature of thermodynamics.

Figure 6 also demonstrates several interesting features and is arranged similarly to Figure 5. In this experiment, all servers in Rack 2 are turned on at 1:50 PM. As expected, this causes a universal rise in room temperature which is seen at all racks. However, slightly after 1:55 PM, Racks 1 and 3 proceed to cool down(!). Further investigation revealed that as the temperature rose, the building thermostat sensed the change and actuated the building AC, causing a depression in temperature. This

```
net = openlocal(opgraph, ...)
results = run(net, ...)
z = myappconverter(results)
cc = corrcoef(z)
plotcorrs(cc)
```

Listing 10: Que commands to compute correlations between monitoring nodes in the server room. Some optional parameters have been omitted.



(a) Clustered sensors (b) Weakly correlated sensors

Fig. 7: Histogram visualization of node correlations. The lighter intensities indicate stronger correlations. For example, nodes 9 and 10 are strongly correlated with each other, but otherwise are not correlated with other nodes. Nodes 1-8 are strongly correlated.

effect was more heavily felt at Rack 1 then at Rack 3 because the AC ventilation was much closer to Rack 1. This sort of complex interaction is difficult to assess without a rich coverage of sensors.

VII. Evaluation

We first built an application that we ran on the nodes in the lab data center testbed. In fact, it is the same application composed of operators shown in Figure 2 and described in Section IV. The application simply collects temperature readings periodically, and send these back to a base station where they are canonicalized into a standard XML format. We ran this application for approximately eighteen days.

Next we evaluated Que with respect to DataCenter.NET in the four important areas of concern for sensor networks that we have outlined: scale, longevity, data and integration. We cover the results of each issue in-depth below, and illustrate how Que was applied.

A. Scale

Scaling up deployments introduces many new issues. We use Que to address one particular issue in this process: what density of spatial coverage is necessary in a production deployment? This has previously been formulated as a theoretical optimization problem [13].

Our test environment, as described in Section VI, embeds motes in a wealth of locations in the environment: six on the ceiling, and two per server, for an average of ten motes per rack. While this finely captures transitions in temperature across space, the number of sensors may be saturating the environment for the utility of the information provided.

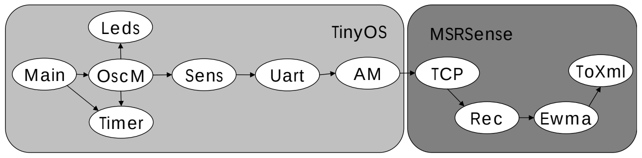
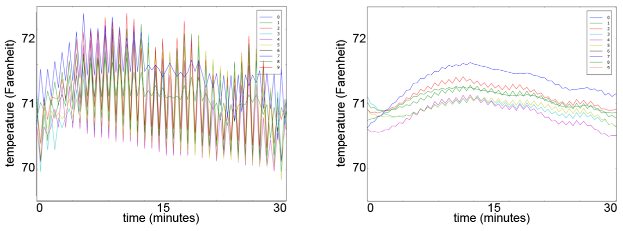


Fig. 8: Operator graph with MSRSense-based EWMA.

```
net = openlocal(opgraph, ...)
results = run(net, ...)
z = myappconverter(results)
# ewma: cleaning
ewmaz = ewma(z)
plotresults(z,ewmaz)
```

Listing 11: EWMA applied as data processing script.

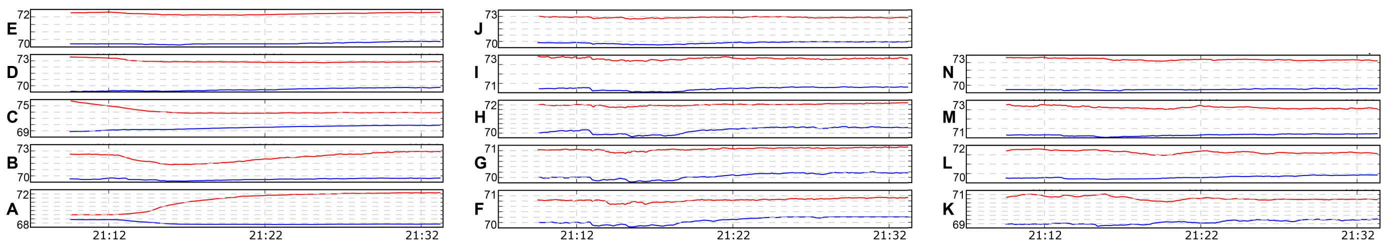


(a) Before. (b) After.

Fig. 9: A sample time series of data with simple EWMA data cleaning applied.

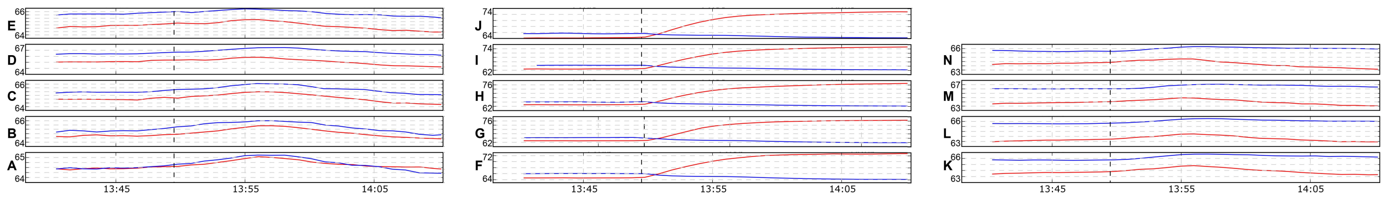
The task is then to determine which sensors to retain if one were to scale to many thousands of racks. We focus on a primitive to mutual information criteria used in [13], the correlation coefficients between every pair of sensors. We are less concerned with network costs since a single-hop base station suffices for all communication in our scenario.

Listing 10 and Figure 7 show the steps we performed in Que to drill down on this question, and the results generated respectively. The correlations between pairs of nodes are illustrated in a 3D histogram where lighter intensities correspond to stronger correlations. For example, in Figure 7a, two clusters emerge: one which contains the majority (eight nodes) and another that contains the minority (two nodes). The larger cluster corresponds to the front and back of one rack during a period of time when no server in the rack was active. The smaller cluster corresponds to two nodes associated with another rack which did have servers activated during the investigated period. Hence, we can start to suspect that if the server workload is highly localized to particular racks, then clusters emerge around nodes of the same rack. In Figure 7b, we tested a different workload that varied across racks. Here, no clear clusters immediately emerge. While more investigation is warranted to determine the optimal configuration for various server workloads, Que's ease of data analysis permitted us to explore this scaling issue quickly.



(a) Rack 1 (b) Rack 2 (c) Rack 3

Fig. 5: A single machine, Server A, turned on by itself. Notice the heat impact on surrounding machines, both in the same rack and in surrounding racks, especially the intake of Rack 2.



(a) Rack 1 (b) Rack 2 (c) Rack 3

Fig. 6: An entire rack, Rack 2, actuated simultaneously during the day. Notice the strange *decrease* in temperature after the initial temperature increase, especially at Rack 1. Later investigations revealed the involvement of the building thermostat AC, underscoring the nontrivial dynamics of the seemingly simple test deployment.

B. Data

The data is the key benefit that draws clients to use sensor networks. One prerequisite of providing data of interest is extracting first level base data from noisy sensor measurements. In particular, data cleaning and calibration is often a mundane but necessary step.

There are two approaches to data cleaning in Que for the data collection operator graph of Figure 2. The first option is the MSRSense operator, ewma. An operator graph involving ewma is shown in Figure 8. Alternatively, Que provides simple ewma as part of the standard set of data analysis tools, in case MSRSense is not part of the operator graph. Its use is shown in Figure 11.

We ran the latter data cleaning procedure and converted initial results shown in Figure 9b to those shown in Figure 9b. This offered a significant improvement in the usable data values, as evidenced by the reduction in variance. The procedure involved no more than a handful of scripting calls shown in Figure 11. Que is effective at quickly performing data processing that, once tuned in the scripting environment, can then be applied in a straightforward fashion as a operator on the actual running platform.

C. Longevity

Next, we investigate ways to improve the longevity and reliability of our system. While many methods to increase system longevity and reliability are possible, we focused on one in particular: we attempted to increase network reliability be performing application-level data reduction and decreasing cross-traffic. In addition, this reduces the energy spent transmitting messages.

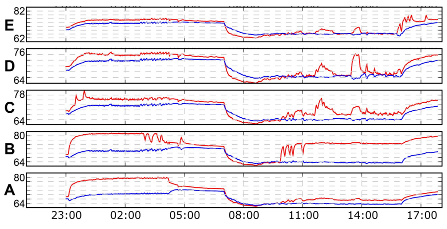
Our approach here is a moving threshold reporting scheme: we convert collection from a periodic event to one in which data is only reported if the measurements are some threshold beyond the previous report. Our main changes to the previous operator graphs was the replacement of the OscM operator for the TrigM operator. This is shown in Figure 10. The corresponding Listing 12 is also shown.

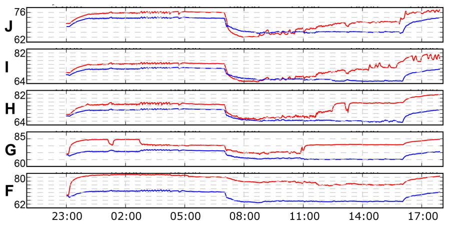
When we ran this series of operators, Que immediately produced odd graphs, shown in Figure 11. In this case, Que allowed us to quickly identify an operator that behaved strangely and produced nonsensical results before we deployed into the field.

D. Integration

Lastly, we are concerned with the lack of support testing end to end systems with traditional sensor network prototyping systems. In the case of DataCenter.NET, this means that a system controller should function as part of the running simulation in an entirely integrated system.

We explored this area by developing and deploying an open-loop controller alongside our sensor network. This controller assigns jobs to servers in a predetermined fashion, without input from the environment, much like existing controllers used in commercial data centers. At present, this controller is a separate application. As a next step, it is natural to incorporate the controller as a MSRSense microserver operator. In this way, it may be manipulated just like any other operator in Que.


(a) Rack 1


(b) Rack 2

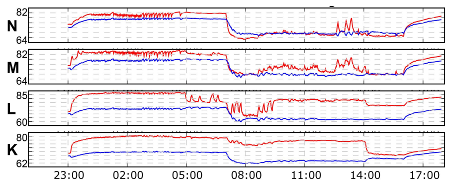

(c) Rack 3

Fig. 13: Open-loop controller measurement results over day and half period. Results for part of this time are shown. Notice the large irregularities in local server and rack temperatures as jobs are scheduled without knowledge of environmental conditions.

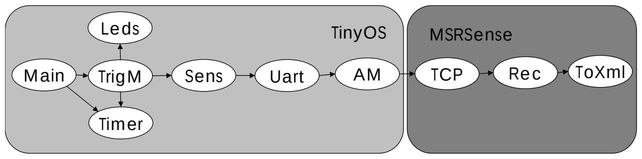


Fig. 10: Operator graph of threshold-triggered reporting

```
net = openlocal(opggraph, ...)
results = run(net, ...)
z = myappconverter(results)
plotresults(z)
```

Listing 12: Event trigger. Some optional parameters have been excluded for brevity.

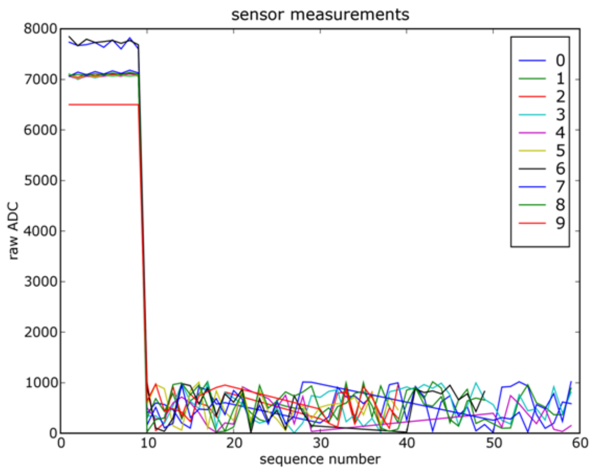


Fig. 11: A bug revealed in our trigger program.

We have already tested the response of a realistic job load on this controller. Figure 12 is a deployed Internet service workload trace representative of one day. We scaled it appropriately to fully load our servers at peak requests.

The temperature fluctuations displayed by our controller are shown in Figure 13. We note several features of this dataset, in particular the high degree of fluctuation of the exhaust measurements, and also the uneven degree to which servers are actuated. On several occasions, the fluctuations are on the order of tens of degrees in several minute's time, suggesting that the variance is indeed very great in a very short time span. These results strongly encourage investigation of more informed closed-loop controllers that incorporate temperature feedback.

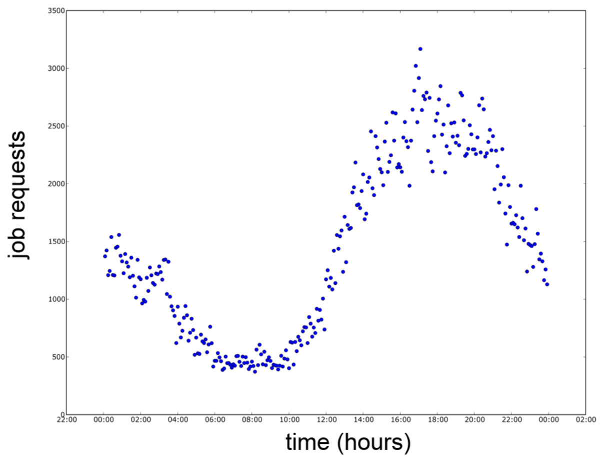


Fig. 12: Deployed Internet service workload trace for one machine during a 24-hour period.

VIII. Conclusion

We have presented Que, an environment in which promising prototypes may be grown into substantial production deployments with relative ease through a simple yet flexible operator wiring and general-purpose scripting. We have leveraged several Que features, including unified system assembley, iterative data processing, and high-level interfacing to explore our new DataCenter.NET deployment, and validate the utility of Que.

The research reported here is based on David Chu's summer intern project at Microsoft Research in 2006.

References

[1] How to choose the location for your next data center (it's the power stupid). *The Data Center Journal*, March 2006. www.datacenterjournal.com.

[2] 109th U.S. Congress. H.r. 5646: To study and promote the use of energy efficient computer servers in the united states, 2005-2006.

[3] P. Buonadonna, J. Hellerstein, W. Hong, D. Gay, and S. Madden. Task: Sensor network in a box. In *In Proceedings of European Workshop on Sensor Networks*, 2005.

[4] D. Carlson. Keynote address. In *SenSys '06*, 2006.

[5] E. A. L. Elaine Cheong and Y. Zhao. Viptos: A graphical development and simulation environment for tinyos-based wireless sensor networks. Technical Report UCB/EECS-2006-15, EECS Department, University of California, Berkeley, February 15 2006.

[6] J. Elson and A. Parker. Tinker: a tool for designing data-centric sensor networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 350–357, New York, NY, USA, 2006. ACM Press.

[7] 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 *In ACM SIGPLAN Conference on Programming Language Design and Implementation, 2003.*, 2003.

[8] 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 *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004.

[9] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The tenet architecture for tiered sensor networks. In *Sensys*, 2006.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[11] J. D. Hunter. Matplotlib, 2006. http://matplotlib.sourceforge.net.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of the 17th Symposium on Operating Systems Principles*, 2000.

[13] A. Krause, C. Guestrin, A. Gupta, and J. Kleinberg. Near-optimal sensor placements: maximizing information while minimizing communication cost. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 2–10, New York, NY, USA, 2006. ACM Press.

[14] E. A. Lee. Overview of the ptolemy project. Technical Report Technical Memorandum No. UCB/ERL M03/25, EECS Department, University of California, Berkeley, July 2 2003.

[15] E. A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, January 10 2006.

[16] P. Levis, N. Lee, M. Welsh, , and D. Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003).*, 2003.

[17] J. Liu and F. Zhao. Towards semantic services for sensor-rich information systems. In *Second IEEE/CreateNet International Workshop on Broadband Advanced Sensor Networks (Basenets)*, November 2005.

[18] D. Malmon. Personal correspondence, 2006. United States Geological Survey.

[19] 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 *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM Press.