Que: A Sensor Network Rapid Prototyping Tool With Application Experiences From A Data Center Deployment

David Chu[1], Feng Zhao[2], Jie Liu[2] and Michel Goraczko[2]

[1] CS Division, EECS Department, UC Berkeley, Berkeley, CA, USA
davidchu@cs.berkeley.edu
[2] Microsoft Research, Redmond, WA, USA
{zhao,jie.liu,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 assists 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 environment-monitoring deployment, DataCenter.NET.

1 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 geomorphologists, 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 [24].

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 have difficulty achieving production standards. This indicates that there are significant factors in the production requirements that are unaccounted for in the prototype phase. To address these disparities effectively, it is often more important to know what is wrong *early, often and approximately* rather than late, infrequently and precisely.

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 enabling fast iterations on whole-system assembly and system input/output testing.

Several important factors influence why this transition is not straight forward, and where a tool like Que aims to provide assistance:

Scale: In the prototype phase, it is important to get something working quickly. This often means over-instrumentation with dense arrays of sensors in a limited area rather than finely-tuned capacity planning. However, scale is driven upward in the production phase while the cost of ownership prohibits over-instrumentation. 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 [7].

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 the prototype phase.

Integration: Integration with the 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 scripting-based exploration environment for operator-based system assembly across multiple platform tiers. Quick scripting makes it easy for developers to retask their system for new data processing either on mote- or microserver-class devices. Second, Que provides a simulation and emulation environment for an entire multi-platform tiered system, so that retasking can be quickly tested on the entire system 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 material 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; hence, providing a great testing environment for Que. Our main result here is that Que, positioned as a general rapid prototyping tool, does indeed provide fast insight into these system-wide issues, while leaving definitive and highly-refined answers for special-purpose tools.

The next section describes related work. Section 3 discusses the design principles that drive Que. Section 4 introduces the Que environment. Section 5 describes the system architecture. Section 6 and 7 discusses our deployment DataCenter.NET and our results in using Que to bring this system to production. Finally, section 8 presents discussion and conclusions.

2 Related Work

Observing the practical difficulty of deploying sensor networks, a number of projects provide "out-of-the-box" sensor network solutions [6,9]. Although these solutions are convenient for data collection tasks, they do not address the inherent customization necessary for many sensor network scenarios. Many proposed sensornet programming systems have aimed to facilitate customization with new programming models or APIs. Several have explicitly looked at the benefits toward rapid prototyping [4, 5]. Also, customization may not mean programming each sensor node directly. They can be expressed in declarative ways, such as seen in TinyDB [23], Semantics Streams [26], and DSN [8]. Or, they can be specified via composition languages.

Composition languages, sometimes referred to as "programming by component wiring," is used in many embedded systems programming and designs [10, 13–15, 17, 22]. They are particularly useful at the system prototyping and testing stages, where the users have some ideas of how they system works, but need more hand tuning. EmStar [13] and Viptos [10] both provide two-tier simulation environments. However, Emstar does not provide ways to program at the sensor mote level. Sensors are primarily used as wireless interfaces for microservers. Viptos is a visual programming interface where microserver components are implemented in Java. Que is similar to Viptos in spirit, but by using a unified Python programming language at both sensor mote and microserver levels, users get access to the full capability of Python for experiment control, data archiving, and visualization purposes. The text based general purpose language gives users powerful and intuitive constructs like loops and conditions when building prototypes. Que also takes advantage of MSRSense to support web services and integration with web and enterprise applications.

Some of the preceding composition languages additionally provide model-based semantics in addition to operational semantics [8, 10, 23]. Model-based semantics often permit establishment of program guarantees beyond those available with purely operational semantics, which may help with ensuring that prototype and production systems both conform to user requirements. As a pure composition language, Que itself does not impose any models of computation. It relies on subsystems like TinyOS and MSRSense [22] to provide execution semantics.

Que's scripting environment is similar in spirit to those proposed in Marionette [25] and Tinker [11]. Whereas Marionette uses scripting for debugging and Tinker 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 Tinker.

3 Design Goals

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

Simplicity: Que does not provide yet another programming approach to sensor networks. Rather, Que directly provides the intermediate operator composition language while other languages (*e.g.,* C) provide operator implementations. It has been argued that restricted coordination languages fit well for constructing systems when operator boundaries are well-defined [20]. Indeed, Que users benefit from the safety and simplicity of the language restrictions, yet retain the ability to create new operators in native systems languages. This programming paradigm is common in embedded systems [11, 14, 15, 22].

Leverage existing libraries: The lack of full node programmability means that Que relies on others to provide the bulk of operator implementations. By default, Que interfaces with three such operator libraries: MSRSense [22], TinyOS [15] and Tinker [11] and additionally offers a general adapter to integrate with other operator libraries.

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

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

```
1 # Create tinyos operator graph
2 op_man = toslib.create('tos/system/Main')
3 op_osc = toslib.create('apps/Oscilloscope/OscilloscopeM')
4 op_com = toslib.create('tos/system/UARTComm')
5 link(op_man, 'StdControl', op_osc, 'StdControl')
6 # ... (3 instantiations and 7 linkages elided)
7 link(op_osc, 'DataMsg', op_com, 'SendMsg[AM_OSCOPEMSG]')
8
9 # Create microserver operator graph
10 op_tpr = mslib.create('ComplexTOSPacketReceiver')
11 op_tpr.setparam('messageType', 'ArrayOscopeMsg')
12 op_d2x = mslib.create('DataToXml')
13 link(op_tpr, 'output', op_d2x, 'input')
14
15 # Bind cross-platform ports
16 op_amp = toslib.createAMPort()
17 op_por = mslib.createTcpPort()
18 link(op_com, 'SendMsg', op_amp, '10')
19 link(op_amp, '10', op_por, '9002')
20 link(op_por, '6001', op_tpr, 'input')
21
22 # Execute emulator
23 emusrc1 = emulator.DataCenterEmulator(connstr)
24 net = openlocal(op_amp, op_por, emusrc=emusrc1)
25 results = run(net, time=60*10, appname='Oscilloscope', dosrcgen=True,
      docompile=True, dosimulate=True)
```

Listing 1.1: Instantiating and linking operators from operator libraries.

4 Example User Session

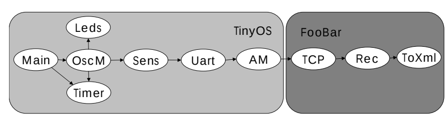


Fig. 1: Operator graph demonstrated in example session.

Next we illustrate a user's interaction with Que via an example session. This comprehensive example session creates, executes and postprocesses the operator graph shown in Figure 1 that spans both mote and microserver platforms, while simultaneously emphasizing the minimal mechanism presented to accomplish these objectives. The application corresponds to the prototypical multi-sensor sampling application. Its function is to periodically send all sensors' measurements over the serial port to the microserver for XML canonicalization.

The user is able to instantiate operators from platform-specific libraries from the interactive shell. Listing 1.1 begins by showing the instantiation and linking of several operators from a particular library, the TinyOS platform library (lines 2-4 and lines 5-7 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 port of one operator to the input port of another operator. There is also a function `unlink` provided to unlink an object. For example, the `Main` operator and the `OscilloscopeM` operators are linked together through the `StdControl` interface (line 5). Parameterized interfaces, introduced in TinyOS, are also supported (*e.g.,* line 7).

Que provides integrated support for both TinyOS and MSRSense. MSRSense is a .NET-based compenentized sensor network microserver. Lines 10-13 show the manipulation of MSRSense operators in Que which by intention is the same as manipulating the TinyOS operator library above. This uniform support for cross-platform operator composition is one point where Que facilitates system assembly.

Cross-platform operator composition such as MSRSense operator to TinyOS operator composition is also easy to accomplish. Lines 16-20 show the binding of ports between operators of different platforms. In particular, the special operators opamp and oppor for Active Message and TCP ports respectively, serve as conduits through which communication occurs between the two platforms. Que identifies and appropriately handles this case, as discussed in Section 5.

In addition to easing system assembly, Que also provides a simulator with great emphasis on ease of use (lines 23-25. 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 23 of Listing 1.1, 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 6. Section 5 describes more about possible emulators.

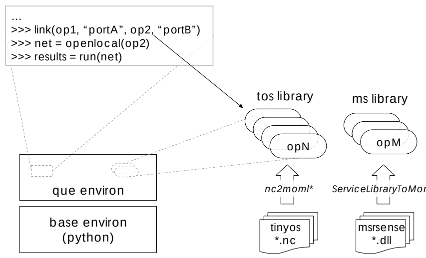
Second in line 24, the openlocal initializes the network topology with a minimum of user intervention. Our ease of use criterion means that the user can either choose a predefined network or can query a preexisting network for its parameters[3]. In addition, openlocal accepts chains of operators and binds these to the nodes initialized in the network.

Third in line 25, 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 5. 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 operator op_d2x.

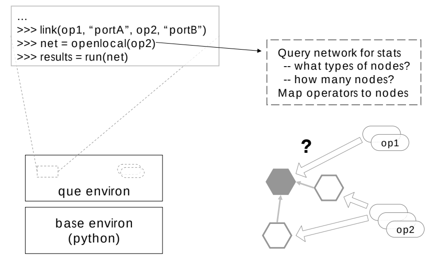
The preceding three commands, and particularly the last one, present simple interfaces for simplicity of use. Yet these allow full flexibility for exercising a custom operator graph on a custom network topology with a custom simulation data source.

By default, simulation results are returned as a sequence of arrays, one for every message from the terminal operator in the graph. After some initial operator-specific data marshalling, the user is able to apply Que's script-based processing to achieve very fast turnaround time for getting initial results. For example, standard utility functions such as 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

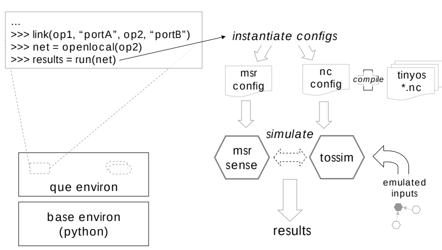
[3] The latter option is not yet implemented.



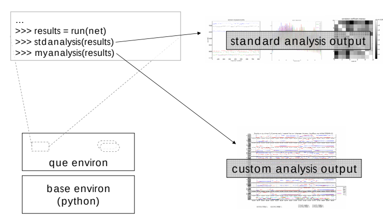
(a) Unified operator composition from interfaces of platform-specific operators.



(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.



(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.



(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. 2: The Que Architecture

wealth of native Python libraries is often a benefit for our scripting environment; `corrcoef` is a built-in Python function that computes correlation coefficients. Additionally, Que can interface to the Tinker and Matlab-like matplotlib Python tools in order to apply more standard data operations [11,16]. Section 7 demonstrates the utility of this rapid data processing.

5 System Architecture

Our architecture, shown in Figure 2, consists of several major components: operator libraries, network libraries, and the simulator. We next discuss the mechanics of each.

5.1 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 operators 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 2a. 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 [19].

The key elements of the interface descriptor 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 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.[4]

5.2 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 as a default.

Another key function of the network object is to pin operators to nodes. As illustrated in Figure 2b, 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 operator virtualization or platform-independent operator implementations). These then permit variable operator placement informed by metrics such as computational speed,

[4] Note that MOML parameters are distinct from parameterized ports. MOML parameters are more akin to NesC generics [12].

energy and sensitivity to network loss. Furthermore, they open the possibility of dynamic operator placement optimization.

5.3 Simulator

The heart of Que is the heterogeneous network simulator. The simulator is initiated with the run command. As shown in Figure 2c, 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 [21]. 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 easy-to-manipulate arrays.

5.4 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 2d. These are exposed as additional user scripts callable from Que. Likewise, we are able to readily adopt Tinker and Matlab-like matplotlib built-in tools [11, 16].

6 DataCenter.NET Deployment

6.1 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, a rapidly rising concern [1, 3]. 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. 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, the density of machinery places an immense cooling requirement placed on the data center facilities; 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.

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 often 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.

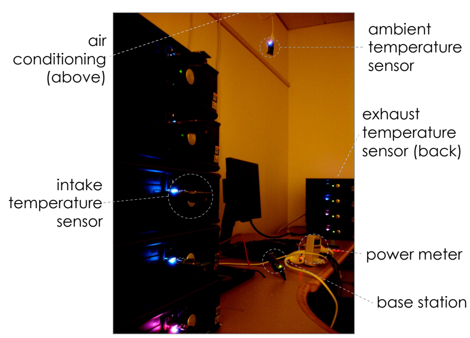


Fig. 3: The DataCenter.NET lab prototype server racks consist of 35 wireless sensors placed on the front and rear of 15 servers and on the ceiling. The servers compromise 3 racks, two of which are visible in the foreground and background here.

6.2 Our Approach

To tackle this problem, we worked with data center managers to develop a wireless network for environment 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 costs (*e.g.,* of traditional thermostats or of upmarket environment sensing-enabled server racks). 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 a live pilot deployment. Our lab prototype, DataCenter.NET, contains 14 servers arranged in racks of 5, 5 and 4 servers each. They are located in a 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 3 shows the components of this setup.

6.3 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 1:

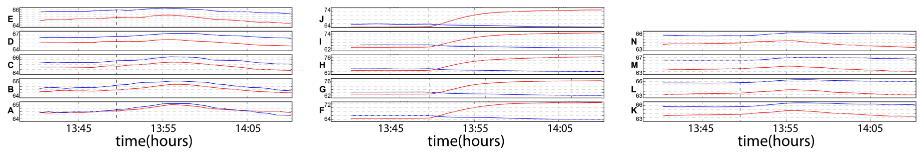
Scale: Our initial prototype consists of 35 motes deployed on 3 racks. However, we are facing a massive scale-up to tens of thousand racks and a proportional increase in the number of wireless sensors.

Longevity: Energy requirements are not initially an issue. However, as we transition to production, battery replacement becomes an increasingly important concern. In particular, the number of 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.

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 7, we address how Que answers these questions we had about our deployment. Section 7 also shows how it was often necessary to modify existing mote and microserver operator graphs because some amount of customization was necessary. Hence, it was not feasible to use readily available off-the-shelf solutions [2, 9].



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

Fig. 4: 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.

Presently, we illustrate some example temperature traces in Figure 4. Three subfigures 4a, 4b and 4c, each correspond to a rack of machines. For example, Subfigure 4a 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.

In the experiment corresponding to Figure 4, 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 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 common, yet difficult to identify without a rich coverage of sensors.

7 Evaluation

We first built an application that we ran on the nodes in the lab data center testbed. This application was previously described in Section 4 and shown in Figure 1. 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. For evaluation purposes, we compare each area to the original base application of Figure 1; in an actual Que usage scenario, each iteration would improve upon the former. While rarely providing the final word on any single topic, we argue that Que delivers on its ability to retask and reevaluate systems quickly. We cover the results of each area of concern in-depth below, and illustrate how Que was applied.

7.1 Scale

Scaling up deployments introduces many new issues. Presently, we use Que to address just 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 [18].

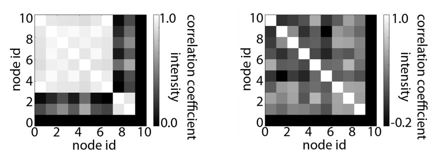
Our test environment, as described in Section 6, embeds motes in a wealth of locations in the environment: six on the ceiling, and two per server, for an average of ten motes per rack. While this finely captures transitions in temperature across space, the number of sensors may be saturating the environment for the utility of the information provided.

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 [18], 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.

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

Listing 1.2: Que script to compute correlations between monitoring nodes in the server room. Some optional parameters have been omitted.

Listing 1.2 and Figure 5 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 darker intensities correspond to stronger correlations. For example, in Figure 5a, 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 5b, we tested a different workload that varied across racks. Here, no clear clusters immediately emerge. While more thorough investigation is warranted to determine the optimal configuration for various server workloads, Que's ease of data analysis permitted us to quickly gain valuable ballpark intuition on the scaling issue.



(a) Clustered sensors (b) Weakly correlated sensors

Fig. 5: Histogram visualization of node correlations.

7.2 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 1. The first option is the MSRSense operator, ewma. An operator graph involving ewma is shown in Figure 6. 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 1.3.

We ran the latter data cleaning procedure and converted initial results shown in Figure 7a to those shown in Figure 7b. 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 1.3. 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.

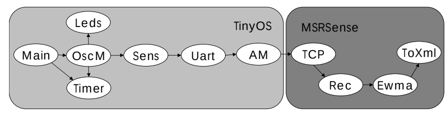


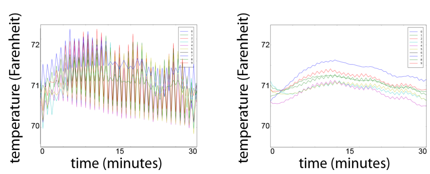
Fig. 6: Operator graph with MSRSense-based EWMA.

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

Listing 1.3: EWMA applied as data processing script.

7.3 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 by performing application-level data reduction and decreasing cross-traffic. In addition, this reduces the energy spent transmitting messages.

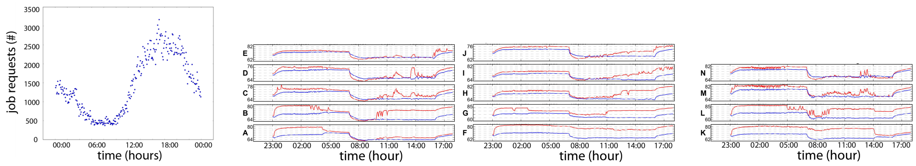


(a) Without EWMA (b) With EWMA

Fig. 7: A sample time series of data with simple EWMA data cleaning applied. Ten sensors are shown, each represented by a line.

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 by the TrigM operator. This is shown in Figure 8. The corresponding Listing 1.4 is also shown.

When we ran this series of operators, Que immediately produced odd graphs, shown in Figure 9. In this case, Que allowed us to quickly identify an operator that behaved strangely and produced nonsensical results before we deployed into the field.



(a) 24-hour workload (b) Rack 1 (c) Rack 2 (d) Rack 3

Fig. 10: 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.

7.4 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.

We have already tested the response of a realistic job load on this controller. Figure 10a is a deployed Internet service workload trace representative of one day. We scaled it appropriately to fully load our servers at peak requests.

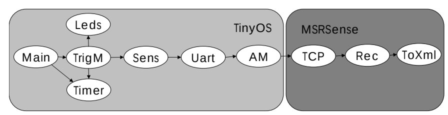


Fig. 8: Operator graph of threshold-triggered reporting

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

Listing 1.4: Event trigger script.

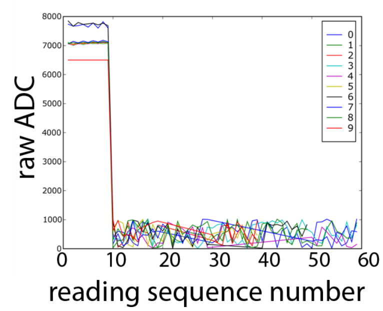


Fig. 9: A bug revealed in trigger program. Ten sensors are shown, each represented by a line.

The temperature fluctuations displayed by our controller are shown in Figure 10. 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. Quick and frequent guidance such as this that Que provides has been very useful for guiding our systems integration rapid prototyping efforts.

8 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 argued that a primary ingredient in healthy application maturation is fast diagnoses of areas of concern as they arrise throughout the prototype process. Four areas of concern which we focused on were scale, longevity, data of interest and infrastructure integration. With Que, users can quickly assess application peformance in these areas. By leveraging several Que features such as unified system assembley, iterative data processing, and high-level interfacing, we have explored our new DataCenter.NET deployment, and validated the utility of Que as a general rapid prototyping tool.

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. Veriteq humidity and temperature mapping and validation with wireless data loggers, 2007. http://www.veriteq.com/validation-data-loggers.
3. 109th U.S. Congress. H.r. 5646: To study and promote the use of energy efficient computer servers in the united states, 2005-2006.
4. U. Arumugam. *Rapid Prototyping and Quick Deployment of Sensor Networks*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 2006.
5. S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(4):563–579, 2005.
6. P. Buonadonna, J. Hellerstein, W. Hong, D. Gay, and S. Madden. Task: Sensor network in a box. In *The Proceedings of 2nd European Workshop on Sensor Networks*, 2005.
7. D. Carlson. Keynote address. In *SenSys '06*, 2006.
8. D. C. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *The Proceedings of 5th ACM Conference on Embedded Networked Sensor Systems*, Nov 2007.
9. A. R. Corporation. Demo abstract: A new embedded web services experience for wireless sensor networks. In *The Proceedings of 4th ACM Conference on Embedded Networked Sensor Systems*, 2006.

10. 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.
11. 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.
12. 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation, 2003*, 2003.
13. 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 *2004 USENIX Technical Conference*, Boston, MA, 2004.
14. 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 *The Proceedings of 4th ACM Conference on Embedded Networked Sensor Systems*, 2006.
15. 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.
16. J. D. Hunter. Matplotlib, 2006. http://matplotlib.sourceforge.net.
17. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. In *The 17th Symposium on Operating Systems Principles*, 2000.
18. 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.
19. 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.
20. E. A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, January 10 2006.
21. P. Levis, N. Lee, M. Welsh, , and D. Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems.*, 2003.
22. J. Liu and F. Zhao. Towards semantic services for sensor-rich information systems. In *Proceedings of Second IEEE/CreateNet International Workshop on Broadband Advanced Sensor Networks (Basenets)*, November 2005.
23. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *Transactions on Database Systems (TODS)*, March 2005.
24. D. Malmon. Personal correspondence, 2006. United States Geological Survey.
25. 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.
26. K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: a framework for composable inference over sensor data. In *The Proceedings of Third European Workshop on Wireless Sensor Networks*, February 2006.