A wirelessly programmable actuation and sensing system for structural health monitoring

A wirelessly programmable actuation and sensing system for structural health monitoring

James Long[1] and Oral Büyüköztürk[1]

[1]Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, USA

ABSTRACT

Wireless sensor networks promise to deliver low cost, low power and massively distributed systems for structural health monitoring. A key component of these systems, particularly when sampling rates are high, is the capability to process data within the network. Although progress has been made towards this vision, it remains a difficult task to develop and program 'smart' wireless sensing applications. In this paper we present a system which allows data acquisition and computational tasks to be specified in Python, a high level programming language, and executed within the sensor network. Key features of this system include the ability to execute custom application code without firmware updates, to run multiple users' requests concurrently and to conserve power through adjustable sleep settings. Specific examples of sensor node tasks are given to demonstrate the features of this system in the context of structural health monitoring. The system comprises of individual firmware for nodes in the wireless sensor network, and a gateway server and web application through which users can remotely submit their requests.

Keywords: Wireless sensor networks, smart sensing, python

1. INTRODUCTION

The structural health monitoring of structures, in theory, offers the potential to detect structural changes, give advance warning of dangerous deterioration, and better understand the usage patterns and performance of infrastructure. Although an active area of research since the 1980's, the application of SHM to real systems has not yet gained widespread adoption. Initially, a major inhibiting factor was cost. The instrumentation of the Tsing Ma bridge in Hong Kong with 350 sensing channels is estimated to have cost upwards of $8 million.[1] A significant component of this cost can be attributed to the requirement to connect the network using coaxial cable. Because of this, the use of newly developed wireless sensor network (WSN) technology for SHM, proposed by Straser and Kiremidjian,[2] sparked massive interest in the research community, and has lead to the development of a number of notable systems for SHM, for example the Narada platform[3] developed at the University of Michigan, and the iMote2 based system used at the University of Illinois.[4]

These systems often rely on battery power, and so the power usage of each device in the network must be carefully considered in real deployments. The active current draw of sensor 'nodes' such as those described in,[3,4] is typically two orders of magnitude higher than the sleep mode. For example the iMote2 platform used by Rice et al[4] draws 86 mA in its typical operating mode, and 0.5 mA in sleep mode. In addition to power consumption, storage and archiving of sensor data is required when data is simply transmitted, which for high data rate applications, can become an obstacle. For example, Jeong et al,[5] propose a NoSQL based database system to manage what they describe as a 'enormous amount of sensor information' arising from bridge monitoring.

A key feature of typical wireless sensor nodes is the combination of sensors, radio, and microprocessing unit on one board. Because the sensor mote has computational capability, and because raw time series data is seldom of interest in itself, it is very attractive to interrogate the data on board the sensor mote before wireless transmission, condensing lengthy time series to a much smaller, higher level message. The term 'smart

Further author information: (Send correspondence to James Long)
James Long: E-mail: jjlong@mit.edu
Oral Büyüköztürk: E-mail: obuyuk@mit.edu

Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2016, edited by Jerome P. Lynch, Proc. of SPIE Vol. 9803, 98033H · © 2016 SPIE
CCC code: 0277-786X/16/$18 · doi: 10.1117/12.2219435

sensing' is used to describe such a methodology. For example, Lynch[6] implemented a data-driven damage detection methodology on board a wireless sensor mote and reported up to 50% savings in power consumption. In a similar approach, Kesavan and Kiremidjian[7] proposed embedding a damage detection methodology which combined the extraction of wavelet based features combined with a hypothesis test for anomaly detection. Rice et al[4] developed a solution capable of processing vibration mode shapes and frequency within the wireless sensor network.

Yet despite this progress, a remaining issue with state-of-the-art wireless sensor networks for SHM (and indeed for other purposes), is that they are notoriously difficult to program for application experts. Rice et al[4] state that '*...it is a very challenging environment for non-programmers to develop network control and application software. The embedded system expertise required to develop SHM applications has limited the use of WSSN technology for monitoring of civil infrastructure.*' Although progress has been made to separate application programming from operating system and networking protocols, software development remains a major obstacle. TinyOS,[8] is an operating system developed specifically for extremely resource constrained low power devices operating in a wireless sensor network. Building upon TinyOS, TinyDB[9] provided a data acquisition system for wireless sensor networks that allows users to specify their requests in Structured Query Language (SQL). However, since the development of TinyOS and TinyDB, the processing speed and memory capabilities of mote devices has drastically improved, and more powerful frameworks are now possible. For example, the mica2 mote referenced in[9] was had a processing speed of 7MHz, and 4KB of RAM. For comparison, many current 32 bit microcontrollers offer upwards of 100 kB of RAM and greater than 100 MHz clock speed. Despite these improvements in hardware capabilities it has not become any easier for application developers to actually use the hardware. However, recent technological developments promise to change this.

In particular, several high level scripting languages have recently been implemented specifically for microcontroller architectures. Espruino[10] and Tessel[11] allow the execution of Javascript code on an embedded device, eLua[12] does the same for the scripting language Lua, and MicroPython,[13] a complete rewrite of Python 3 for microcontrollers, allows near-C like speed in executing Python code. Using a high level scripting language can help make the development of node level software quicker and easier, but importantly can also lead to much more dynamic applications. In this paper we present a new system for wireless sensor nodes, developed using MicroPython. This system allows users to specify custom Python requests to the sensor network. These requests allow users to stipulate the sensor data they want to acquire, but also custom on-board processing logic in the form of python code. The wireless sensor node system presented in this paper is capable of processing multiple requests concurrently, and of scheduling repeating requests in a straightforward manner. It also offers several low power sleep modes, which are automatically entered when the sensor node is idle.

The system also enables users to remotely specify their requests to the network through a web browser. These requests are handled by a server which acts as a gateway to a Zigbee network of wireless sensor nodes. The results of individual user's requests are first stored in a database on the web browser, and can be accessed and downloaded from the web browser interface.

2. SYSTEM OVERVIEW

A schematic of the full wireless sensor network system is shown in Figure 1. The wireless sensor network itself consists of individual sensor nodes comprising of a STM LIS344ALH MEMS accelerometer, an STM 32 bit microcontroller flashed with the MicroPython binary, and an XBee Series 2 radio. The 'coordinator' radio is connected to a gateway server which provides the interface between the zigbee network and the internet. A browser based interface can be accessed via the internet and used to receive updates from the sensor network, and to write and submit custom Python requests to the server, which in turn forwards these requests to the wireless sensor network, collects the response and updates a database, as well as serving the response to the browser.

3. SENSOR NODE SOFTWARE

In this section we will describe the python software developed for the STM32 microcontroller which enables the processing of users requests on board invidivual sensor nodes.

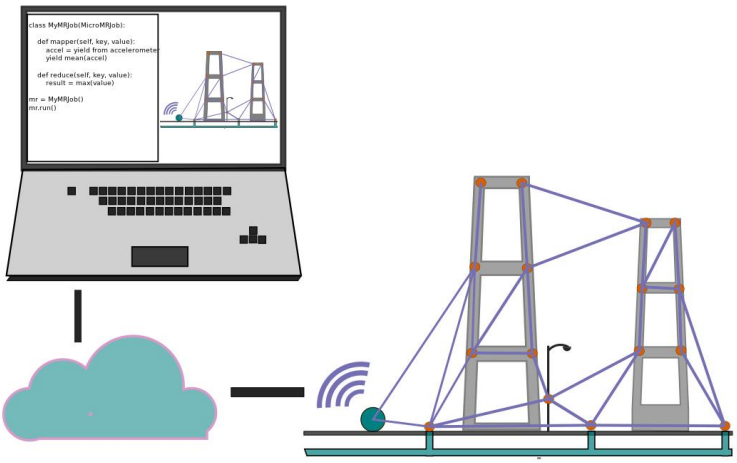

Figure 1. Schematic of integrated browser, server and zigbee network

3.1 Event loop model

At the core of our node level software system is the event loop. The event loop is an infinite main program loop which maintains a priority queue of functions to be executed, ordered by their scheduled execution time. A reader function can be registered on the loop, which watches for read availability on an input object. In our case, an event reader is added to the universal asynchronous receiver/transmitter (UART) bus which connects the microcontroller unit to the Xbee radio. The UART connection is then polled by the event handler, and when data is available to be read the reader function is called by the event loop.

When using the event loop model, care must be taken not to block the event loop. At the simplest level, the event loop maintains a priority queue of functions, checking at every CPU cycle whether the current time is greater than the scheduled execution time of the first function on the queue. When the scheduled execution time is reached, the event loop executes the function. This function runs until completion, upon which the event loop returns to checking whether the next function on the queue is ready for execution. If any single function runs for too long, the execution of the next functions on the queue may be delayed. This also applies to the event handler function, and so the event handler cannot react to input when another function is executing. To ensure that our sensor nodes maintain the ability to quickly react to and read new input, all functions added to the queue should return in a reasonable amount of time.

3.2 Coroutines

In Python, in addition to simple functions, 'coroutines' can be scheduled by the event loop, provided by the library 'asyncio'. A coroutine is a function that can be suspended and resumed without loss of state. This ability is extremely useful in the context of the event loop, as it provides a simple mechanism to schedule long running or repetitive tasks without blocking the event loop. When the coroutine reaches a 'yield' point it is suspended, and added to the priority queue to be resumed at a later point. This allows the event loop to execute any other tasks on the queue in the interim, including the handling of any new input. When multiple different coroutines are added to the event loop, the behaviour emulates a cooperative multitasking operating system. This style of multitasking (used even on PCs in the 1990s), allows switching between different tasks with almost no overhead, but relies on each task voluntarily yielding control. If a single task does not yield, then it will block the execution of all other tasks indefinitely.

To illustrate how the event loop model and coroutines work in the Python language, consider the following simple toy example, where two tasks are scheduled for execution. The first task calculates the sequence of square numbers, printing each element of the series. The second task calculates the fibonacci sequence, again printing each element:

```
@asyncio.coroutine
def square_numbers():
    i = 0
    while i<5:
        print('square series: ',i**2)
        i+=1
        yield from asyncio.sleep(0)

def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)

@asyncio.coroutine
def fibonacci_numbers():
    i = 0
    while i<5:
        print('fibonacci series: ', fib(i) )
        i+=1
        yield from asyncio.sleep(0)
```

When we add these tasks to the asyncio event loop, and run it, the output will be as follows:

```
square sequence:  0
fibonacci sequence:  0
square sequence:  1
fibonacci sequence:  1
square sequence:  4
fibonacci sequence:  1
square sequence:  9
fibonacci sequence:  2
square sequence:  16
fibonacci sequence:  3
```

Here we see that both tasks make interleaved progress, and the event loop alternates between them. Although this is a trivial example, it shows the ability to simply specify a task which can be interrupted and resumed without losing its local state. Now that the concept of the event loop and coroutines has been introduced, we will describe the event loop based cooperative scheduling used to control the individual wireless sensor nodes in our system.

3.3 Coroutine communication: Producer-consumer relationships

In the previous toy example the two coroutine tasks operated independently, and did not require any shared information. However, in many cases one task produces data, which another task consumes, and a method to share data between these tasks is required. For example, and as will be described in detail in the next section, when data arrives to the MCU from the radio, the input handler reads this data, parses it according to the wireless protocol and then places messages on a queue to be consumed by a task which processes these messages. This relationship can be facilitated through the use of a first in first out queue. This producer-consumer queue itself has a coroutine method for 'getting' data. This coroutine checks for data on the queue, and yields this data if it exists. While the queue is empty the 'get' coroutine simply yields control back to the event loop. To illustrate this relationship, it is useful to consider a modified version of the toy example from the previous section. In this example, one coroutine task calculates and prints the fibonacci sequence before putting them on the queue. The second coroutine task calculates the square of the fibonacci elements produced by the first coroutine.

```
queue = asyncio.Queue()
def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)

@asyncio.coroutine
def fibonacci_numbers():
    i = 0
    while i<5:
        fib_i = fib(i)
        print('fibonacci producer: ', fib_i)
        yield from queue.put( fib_i )
        yield from asyncio.sleep(0)
        i+=1

@asyncio.coroutine
def square_numbers():
    while True:
        i = yield from queue.get()
        print('square sequence: ',i**2)
```

Both tasks again make interleaved progress, and communicate through the queue, producing this output:

```
fibonacci producer:  0
square series:  0
fibonacci producer:  1
square series:  1
fibonacci producer:  1
square series:  1
fibonacci producer:  2
square series:  4
fibonacci producer:  3
square series:  9
```

This producer-consumer relationship enables straightforward communication between different sensor node coroutine tasks, which will now be described in detail.

3.4 Sensor node tasks

In order to provide the desired behaviour and functionality described earlier in this section there are a number of individual tasks that must run cooperatively on the sensor node microcontroller. Multitasking is achieved through the asyncio event loop in the form of cooperative scheduling, as previously described. These tasks are briefly described in this section.

Requests to individual sensor nodes arrive via the UART connection to the Xbee radio. Handling these requests is the primary job of each individual sensor node, and is achieved by the **'input_handler'** task. To respond to radio events an asyncio reader is added to the UART port connected to the radio. The asyncio reader polls the UART for the availability of data. When data becomes available the input handler is called. This input handler reads all available data, parses it, and places the resulting messages on a 'message queue'. and puts it on the data queue, similarly to the toy example discussed previously. Because the UART object has a finite buffer size there is always a risk that the buffer will become full, and data from the radio will be missed. Therefore it is important that all other tasks yield control to the event loop regularly enough such that the input handler has the chance to read any available data, freeing up the UART buffer.

The **'function_scheduler'** task consumes data from the message queue, produced by the input handler. When a user makes a request to a node this message will contain the definition of a python function, which specifies the logic of the request the user wants to make. The message also contains a unique string identifier for the job, as well as two parameters, 'repeat', and 'every', which specify how many times the user wants the task to repeat, and how often the task should repeat (in milliseconds). The function scheduler reads new messages from the message queue, defines the user-specified function and schedules its execution by placing it on the event loop priority queue. Here, we also add networking logic to the user-defined function, which in the simplest configuration, specifies that the returned result of the users function is sent back to the coordinator node of the zigbee network. If the request is repeating, the function is placed on the queue multiple times, with the scheduled time for each repetition calculated easily from the 'every' parameter, the repetition number, and the received time of the message.

The task responsible for acquiring data from the physical sensor is the **'sensor_reader'**. In our system, the main microcontroller unit is connected via UART to the MEMS accelerometer board, which is responsible for the analog to digital conversion of the accelerometer data. The process of collecting and streaming data from the accelerometer board is relatively slow, but requires almost no computational effort from the main microcontroller. To avoid blocking the event loop during this time, some careful consideration is needed. First, an asyncio reader is attached to the UART port connected to the accelerometer board. This reader polls for the availablity of data on the UART and executes a callback function when data is available. The callback function puts the data on a 'sensor queue'. With this machinery in place, the process of reading is relatively simple and is non-blocking. The **'sensor_reader'** task, when called, writes over UART to trigger a data collect. Then the task gets data from the sensor queue as it becomes available, and returns it once the full data set has been acquired. Importantly, in the time between writing to trigger the collect, and when the data becomes available, the main microcontroller is free to make progress on other tasks.

Finally, the **'sleep_manager'** task is responsible for placing the sensor node into its sleep modes. There are two different sleep modes; stop and standby. Stop mode reduces the power consumption to approximately 0.3 mA, and can be exited through either a real time clock event, or an external interrupt, for example the arrival of data on the radio. During stop mode state is preserved. Standby mode reduces the power consumption to less than 50 μA, but can only be exited through a real time clock event, and all program state is lost. The choice of sleep mode, and sleep period can be altered remotely as will be discussed in the next section. The **'sleep_manager'** task first checks whether there is any data available to be read or whether a user-request is still being processed. If either of these conditions are true the sleep manager simply yields control back to the event loop, to allow completion of outstanding tasks before sleep. If all available data has been read, and there is no request being processed, the sleep manager then checks whether there are any queued requests. If there are queued requests, and the sleep mode is set to standby, we first save these requests to file, so they can be reloaded and requeued when the node wakes up. The sleep manager then places the node into the chosen sleep mode for a fixed period of time. If there are no queued requests, no save to file is required and the sleep manager can immediately place the node into the sleep mode.

3.5 Features

In this section we will describe some examples of user-specified requests to the node level software that are enabled by the event loop model and the coroutine tasks described in the previous section. For each request we will also show an illustration of the time spent in each coroutine task by the processor, to demonstrate how cooperative multitasking is achieved by the system. To help visualise the cooperative multitasking, a colour coded schematic of the various coroutine tasks (somewhat simplified from the previous section) is shown in Figure 2.

3.5.1 A simple request

First, a simple request which acquires a 1000 point long acceleration time series, and then calculates and returns the sum of this series is shown. This request is obviously simpler that a typical SHM application, but is nonetheless instructive. The logic contained in this function can be replaced by any valid python code by the user. On the right of this figure, an illustration of the progression of tasks is provided. Moving anticlockwise from the top, we see that initially the sensor node is in sleep mode. It wakes upon the arrival of data, and the

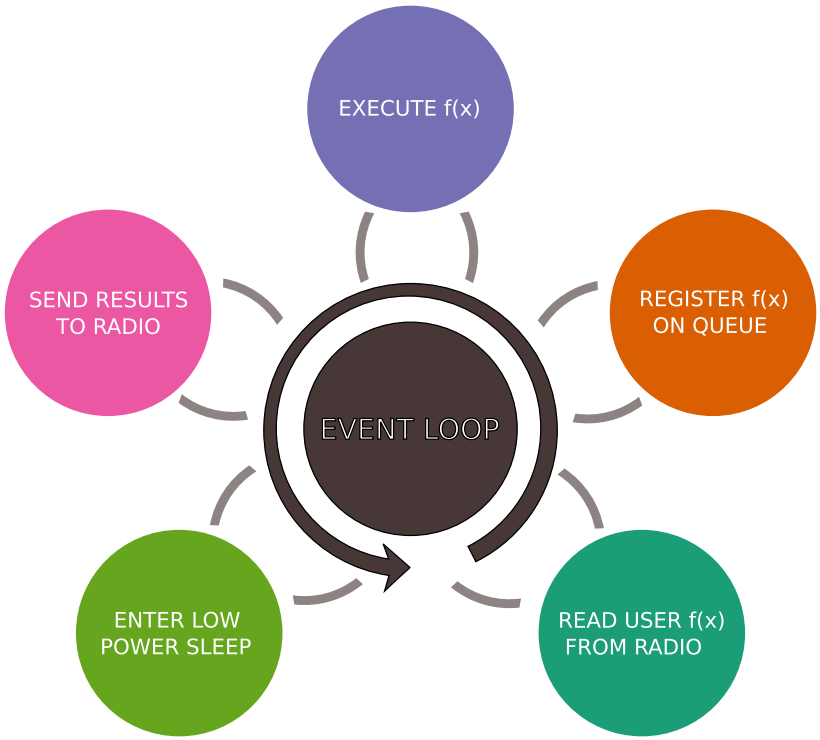

Figure 2. Illustration of the event loop and sensor node coroutine tasks

input handler is triggered, which reads as much data as is in the buffer, before yielding control back to the event loop. This continues until all the input data has arrived and been read. This data is consumed by the function definer, shown in orange, which defines the function and schedules its execution. This task then begins executing, as shown in blue. For simplicity, we omit the CPU cycles spent acquiring the accelerometer data. We can see that the remainder of this task does not yield at any point to the event loop, and therefore all CPU cycles until its completion are spent in this task. Finally, the result of the task is sent back to the user. This task is shown in pink, and illustrated here in the 'send latency' portion, is a scenario in which the write buffer becomes full, and control is yielded back to the event loop until the remaining data can be sent. Finally, with the request fully completed, the sensor resumes its fallback behaviour of intermittently waking between sleep periods.

```
def sum_accel(self):
    accel = yield from accelerometer(1000)
    count = 0
    for i in accel:
        count += i
    yield count
```

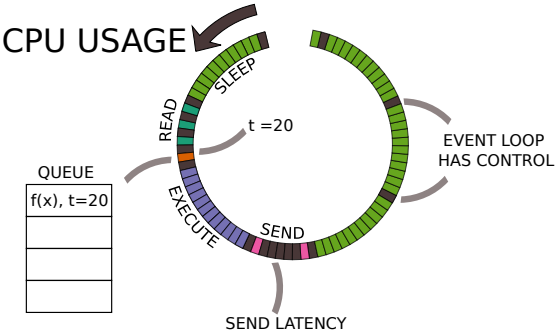

Figure 3. Processing a simple request

3.5.2 Non blocking request

If a user wants to specify a relatively time consuming request, they can improve the performance of the system by periodically yielding to the event loop during this request. This can be achieved by adding a simple 'yield None' line to any iterative computation, allowing the event loop to read any new input data and make progress on other tasks. This syntax is shown in Figure 3.5.2. Here we see that the addition of this line allows the event loop to resume control at regular intervals during the execution of the users task.

```
def sum_accel(self):
    accel = yield from accelerometer(1000)
    count = 0
    for i in accel:
        count += i
        yield None
    yield count
```

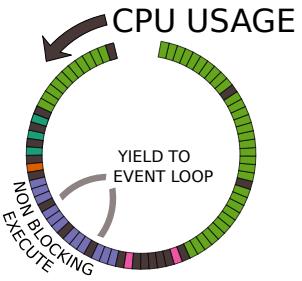

Figure 4. Processing a simple non-blocking request

3.5.3 Repeating request

In Figure 3.5.3, the syntax required to schedule a repeating task is shown. Here, the use of the keywords 'repeat', and 'every', are used to instruct the function definer to place the task on the queue twice: Once for immediate execution, and once 30 seconds in the future.

```
def sum_accel(self):
    accel = yield from accelerometer(1000)
    count = 0
    for i in accel:
        count += i
        yield None
    yield count
repeat = 2
every = 30000
```

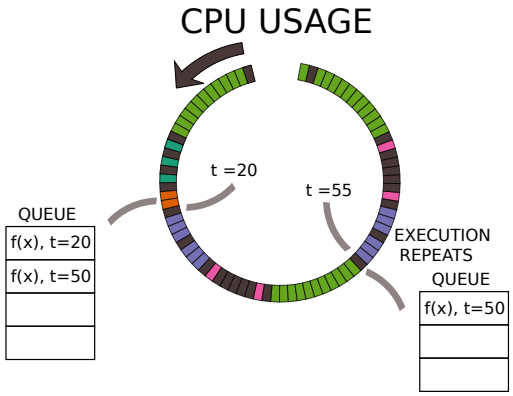

Figure 5. Processing a repeating request

3.5.4 Sleep configuration

Finally, in addition to tasks comprising of data acquisition and computation, the default behaviour of the sensor nodes can be altered through simple python functions. This can be triggered by any user with appropriate privileges, or in an automated fashion from the gateway server. In Figure 3.5.4, a function that specifies that the sleep period be changed to 10 seconds, and that the sleep mode be set to standby is shown. On the right of Figure 3.5.4, we see that although this function serves a different purpose from those previously described, it is read, queued, and executed in exactly the same manner. After execution of this function, the changed sleep behaviour is illustrated on the right hand side.

```
def change_sleep(self):
    self.sleep_mode = 0
    self.sleep_for = 10000
```

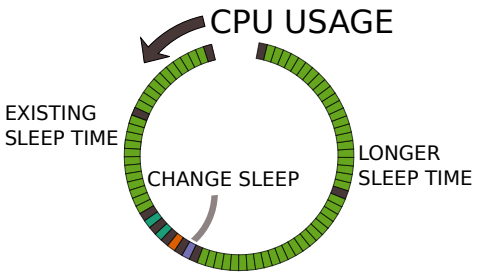

Figure 6. Configuring the sleep parameters

4. GATEWAY SOFTWARE

To allow users of the network to submit requests and view results via the internet, a server application is required. This server acts as a gateway between the zigbee network and the internet, handling http requests from users, and also providing access to the coordinator node of the zigbee network. Authenticated users can log in via a web application, after which they are given access to a code editor from which http post requests are made to the server.

The server, (developed using the popular javascript framework node.js), then handles the http 'POST' requests containing python code, converts this request into zigbee readable frames, adds a unique identifier, and sends the source code to the nodes specificed by the user. When the results of these requests are returned to the server via the zigbee coordinator node they are saved to a database. This ensures that the data is available to the user, even if the connection is lost or the browser is closed between the time the request is submitted and the result is returned. Users can then access this data at any time through the browser application. A screenshot of the browser application developed in conjunction with this server is shown in Figure 7.

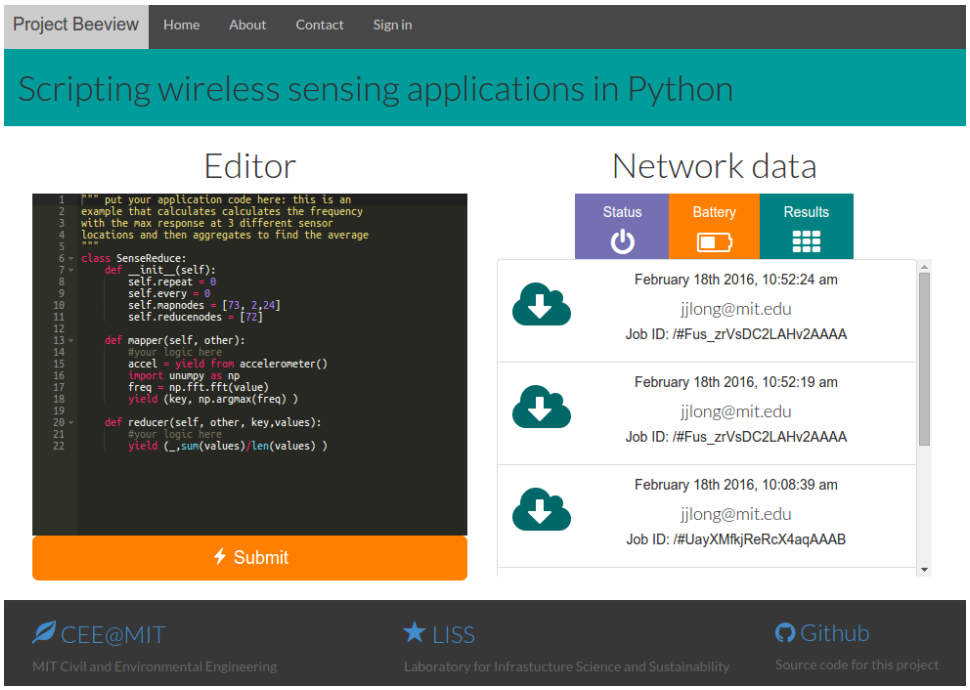

Figure 7. Screenshot of browser interface

5. CONCLUSIONS

In this paper a python based software system for individual wireless sensor nodes in a zigbee network is presented. This system enables multitasking through cooperative scheduling implemented with an event loop and coroutines. The key functionality of this system is the capability to listen for and process data acquisition and computational tasks specified by users in the form of python functions. While enabling this dynamic behaviour, the wireless sensor nodes still retain the ability to autonomously enter low power sleep states when idle. The system also allows for the scheduling of repeating tasks, a feature of significant practical use for structural health monitoring applications.

An internet interface to this network of wireless sensor nodes is also presented. This interface is comprised a web server and database, and a browser based web application. Using this interface individual users of the sensor network can remotely specify their sensor node applications, which are then sent to the nodes. The returned results are stored in a database, and can be accessed through the web application. This interface greatly simplifies the process of requesting data from the sensor network.

In conclusion, a system for remote and high level programming of low power wireless sensor network applications has been proposed and discussed in this paper. This system enables rapid and easy development of sensor network applications without frequent costly firmware updates, and provides a flexible and reusable framework for developing applications that are distributed across the sensor network. This system will allow researchers to specify their algorithms in a scripting language, without explicitly specifying networking, or writing low level microcontroller code. We believe that this will allow the rapid deployment of SHM applications while still preserving scarce power and bandwidth resources

ACKNOWLEDGMENTS

The authors acknowledge the support provided by Royal Dutch Shell through the MIT Energy Initiative, and thank chief scientists Dr. Sergio Kapusta and Dr. Dirk Smit for their oversight of this work.

REFERENCES

[1] Lynch, J. P. and Loh, K. J., "A summary review of wireless sensors and sensor networks for structural health monitoring," *Shock and Vibration Digest* **38**(2), 91–130 (2006).

[2] Straser, E. G. and Kiremidjian, A. S., [*A modular, wireless damage monitoring system for structures*], John A. Blume Earthquake Engineering Center Stanford, CA, USA (1998).

[3] Kim, J., Swartz, A., Lynch, J. P., Lee, J.-J., and Lee, C.-G., "Rapid-to-deploy reconfigurable wireless structural monitoring systems using extended-range wireless sensors," *Smart Structures and Systems* **6**(5-6), 505–524 (2010).

[4] Rice, J. A., Mechitov, K., Sim, S.-H., Nagayama, T., Jang, S., Kim, R., Spencer Jr, B. F., Agha, G., and Fujino, Y., "Flexible smart sensor framework for autonomous structural health monitoring," *Smart Structures and Systems* **6**(5-6), 423–438 (2010).

[5] Jeong, S., Byun, J., Kim, D., Sohn, H., Bae, I. H., and Law, K. H., "A data management infrastructure for bridge monitoring," in [*SPIE Smart Structures and Materials+ Nondestructive Evaluation and Health Monitoring*], 94350P–94350P, International Society for Optics and Photonics (2015).

[6] Lynch, J. P., Sundararajan, A., Law, K. H., Kiremidjian, A. S., and Carryer, E., "Embedding damage detection algorithms in a wireless sensing unit for operational power efficiency," *Smart Materials and Structures* **13**(4), 800 (2004).

[7] Kesavan, K. N. and Kiremidjian, A. S., "A wavelet-based damage diagnosis algorithm using principal component analysis," *Structural Control and Health Monitoring* **19**(8), 672–685 (2012).

[8] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., et al., "Tinyos: An operating system for sensor networks," in [*Ambient intelligence*], 115–148, Springer (2005).

[9] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W., "Tinydb: an acquisitional query processing system for sensor networks," *ACM Transactions on database systems (TODS)* **30**(1), 122–173 (2005).

[10] "Espruino." http://www.espruino.com/. Accessed: 2015-04-30.
[11] "Tessel2." https://tessel.io/. Accessed: 2015-04-30.
[12] "Elua." http://www.eluaproject.net/. Accessed: 2015-04-30.
[13] "Micropython: Python for microcontrollers." https://micropython.org/. Accessed: 2015-04-30.