A Two-stage Bootloader to Support Multi-application Deployment and Switching in Wireless Sensor Networks[*]

Alan Marchiori and Qi Han
Department of Mathematical and Computer Sciences
Colorado School of Mines, Golden, Colorado 80401
Email: {amarchio,qhan}@mines.edu

Abstract—**Wireless sensor networks are built from highly resource constrained embedded systems. Supporting multiple applications on the sensor network is a desirable goal, however, these constraints make supporting multiple concurrent applications on each node difficult. Therefore, we propose a dynamic application switching approach where only a single application is active on each sensor node at a time. In this paper we present a dynamic application switching framework that can automatically reprogram the sensor node in response to application requests. We implement our framework on the TelosB platform and evaluate its performance using a 52-node sensor network testbed. The implementation of a two-stage bootloader reduces the memory requirements to only 1KiB of program memory and 8 bytes of RAM on this platform. We evaluate the implementation using two different modes of application switching; asynchronous and synchronous. Extensive performance studies indicate that dynamic application switching using our two-stage bootloader is a useful approach to support multiple applications in wireless sensor networks.**

I. Introduction

As wireless sensor networks are becoming useful in real applications, it is often desirable to leverage the deployed nodes to support multiple applications. This reduces the overall cost of the sensor network because the cost can be shared between several users. This also allows the sensor network to be improved by deploying new applications after the initial deployment. There are of course different ways to support multiple applications on a sensor node, such as virtual machines, threading, and scripting languages. However, with these methods each application must contend for the limited resources on the node. Our objective is to support multiple applications on the sensor node by dynamically loading and executing each application in isolation from all other applications stored on the node. We refer to this approach as application switching. This should not be confused with the fine-grained 'switching' a multi-threading operating system employs.

In a wireless sensor network we often times think of the network as a complete system instead of collection of individual nodes. With this network scale view it is clear that executing *every* application on each node is not necessary. The sensor network is more like a multi-core processor where each sensor node represents a single core. The applications can then be assigned to various nodes and executed. This application-node mapping may need to change dynamically. Instead of invoking energy- and time-consuming reprogramming of the nodes or the entire network, we will pre-deploy these applications on each individual sensor node and then switch to the requested one "on-the-fly".

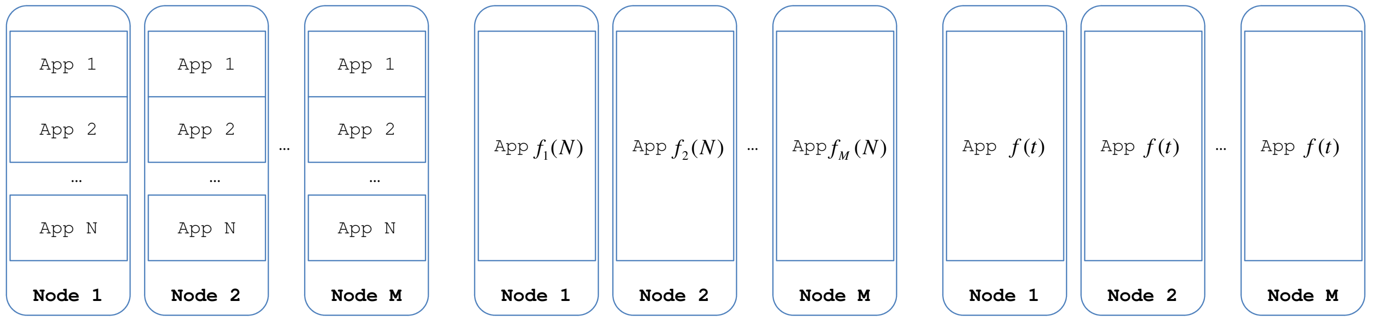
Application switching has been considered previously for code updates and loading failsafe modes [1]. In these scenarios application switching is an unusual occurrence and often initiated by a human operator. We would like the application switching to be fully automated and have minimal impact on the sensor network as a whole. To achieve this, we use a two-stage bootloader. The first stage is highly optimized and only loads the second stage bootloader. The second stage is a full application complete with a user interface. This allows applications to be loaded into memory or executed while minimizing the memory overhead imposed by the bootloader. The primary advantage of our approach is that each application is executed in isolation from every other application. This can prevent complex interactions between applications and simplifies the development process. This allows multiple *complex* applications to co-exist on the sensor node.

In this paper we present the design of a dynamic application switching framework using a two-stage bootloader, discuss the implementation of the framework on the TelosB platform, demonstrate its feasibility and usability on a network testbed of wireless sensor nodes, and evaluate its overhead and efficiency. Our analysis and empirical studies indicate that our approach is a valid and effective approach for multi-application deployment in wireless sensor networks.

II. Multi-application Support

We define an application switch for a sensor node as the complete reprogramming of the node with a new binary code image. Most wireless sensor network operating systems use a monolithic kernel where the application and operating system are compiled into a single binary image, so in our definition there is no distinction between a user's application and the operating system. When an application switch occurs, the currently running application/operating system is halted and a new binary code image is loaded into program memory

[*]This project is supported in part by NSF CSR grant CNS-0720875.



(a) Concurrent executions (b) Sequential executions: asynchronous switching (c) Sequential executions: synchronous switching

Fig. 1: Program memory map at any time instant with multi-application support to execute N different applications on M sensor nodes.

and executed. The newly loaded code may use a completely different operating system.

There are three different ways to support multiple applications on a sensor node. Figure 1 is an example to illustrate the difference when N applications are deployed on a network of M sensor nodes.

1) Concurrent executions (Figure 1a). Multiple applications may reside on the same sensor node and need to be executed concurrently. This implies that all the applications share the node's resources and some form of multitasking is available.
2) Sequential executions with asynchronous switching (Figure 1b). The same applications are deployed on all of the sensor nodes. However, each sensor node selects a single application and executes it using some function $f_n(N)$. The decision of which application to run can be made manually by the operator, automatically by a central server or each node individually. For example, it is likely that two neighboring nodes in a building would have correlated temperature readings. In this case it may not be useful for two neighboring nodes to monitor temperature. Therefore, when a node starts up, it would search its local neighborhood and choose the application based on the distribution of applications already running. This would serve to distribute the applications among the available sensor nodes according to some predetermined goal.
3) Sequential executions with synchronous switching (Figure 1c). The entire network executes the same single application at the same time, $f(t)$. At certain times, all the nodes in the network will switch to a common application. The schedule for application switches could be computed ahead of time and distributed to each node, computed dynamically on the nodes, or delivered over the wireless network. This type of switching could be useful in an office building where the nodes monitor environmental conditions (temperature, humidity, light) during the day; they all switch to a security application that detects and tracks moving objects after hours.

Existing work has considered the first method, i.e., the support of concurrent applications. Section III discusses this approach. In remainder of this paper, we focus on the second and third methods.

III. Related Work

There are two branches of similar work in supporting multiple applications on wireless sensor networks.

The first enables new code images to be downloaded and executed using the wireless interface. Deluge [1] is a data dissemination protocol that allows code images to be downloaded to sensor nodes using the wireless interface. By disseminating a new code image to all of the nodes in a sensor network we can effectively cause an application switch. The drawback to this approach is that it can take a significant amount of time to disseminate a complete code image to the motes. This approach is most like ours, however, our approach is to load all of the code images before their deployment so that we can quickly switch between them.

The second branch focuses on the simultaneous execution of multiple applications on each sensor node. This is distinctly different from our approach and provides some useful features at the expense of resources. This expense is a result of the more complex operating system support required and the fact that the node's resources must be shared between each application.

One way to enable multiple simultaneous application support is to implementing a threading model on the sensor node. Several sensor network operating systems can support multiple simultaneous applications through threading. Mantis OS [2] supports preemptive multi-threading. Contiki [3] and TinyOS [4] support cooperative threading. Finally, FreeRTOS [5] supports both preemptive and cooperative threading in addition to co-routines. However, all of these approaches have significant overheads when compared to the application switching approach we present in this paper.

Another approach is through the dynamic tasking of nodes in wireless sensor networks using a virtual machines, agents, or simple scripting languages. The predominant virtual machine used in wireless sensor networks is Maté [6]. Melete [7] extends Maté with additional features for delivering, storing, and executing concurrent applications on the sensor node. Agilla and Agimone [8] are agent based architectures using a customized Maté virtual machine. Tenet [9] implements a scripting language where motes can be dynamically tasked as needed. Each of these approaches provide strict separation between the underlying hardware and application software. While this separation is good in some respects, such as portability, it is well known that mote class devices have significant resource constraints and operate at a couple of MIPS. The end result of this approach is that much of the available resources are consumed by the framework and application performance is significantly reduced. Our goal is to support application switching without requiring a specific operating system while minimizing the overhead imposed by the framework. This will allow application developers to utilize all of the resources provided by the motes to develop complex applications with whatever tools they prefer.

IV. Our Approach: A Two-stage Bootloader

Our objective is to deploy multiple applications on each sensor node and then dynamically switch to desired application. Our design goals include

- memory usage minimization. Memory usage (both RAM and ROM) should be minimized in order to maximize the available memory for applications.
- operating system independence. The switching framework should not make any assumptions about the operating system (TinyOS, MantisOS, Contiki, etc.) used by the application.
- ease of use. The process to create an application should be no more complicated than creating a single isolated sensor network application.

Wireless sensor nodes are much like any other embedded system where the operating system is stored on the microcontroller's internal flash memory. In many embedded systems no bootloader is even needed. However, to support application switching we need to decide which application should run and reprogram the internal memory as needed. The usual job of a bootloader is to initialize the CPU and begin execution of the operating system. To achieve the stated goals, we design a two-stage bootloader to select and execute a chosen application, and reprogram the node's program memory if necessary. By using a two-stage bootloader we can minimize the overhead of our approach. Since our bootloader needs to run every time the node reboots, it must be resident in program memory. Our approach is to create a simple first stage bootloader that only does one thing - it loads the second stage bootloader from external memory into RAM. This allows us to locate all of the code intensive functions such as downloading a new application in the second stage. Because application switching is implemented in the bootloader, before the application is even executed, we also achieve our second goal of operating system independence. The bootloader has no dependencies on the application; it simply loads machine code into program memory and switches program execution to the application. This also means application development is not complicated, because the application runs exactly in the same way as without the bootloader.

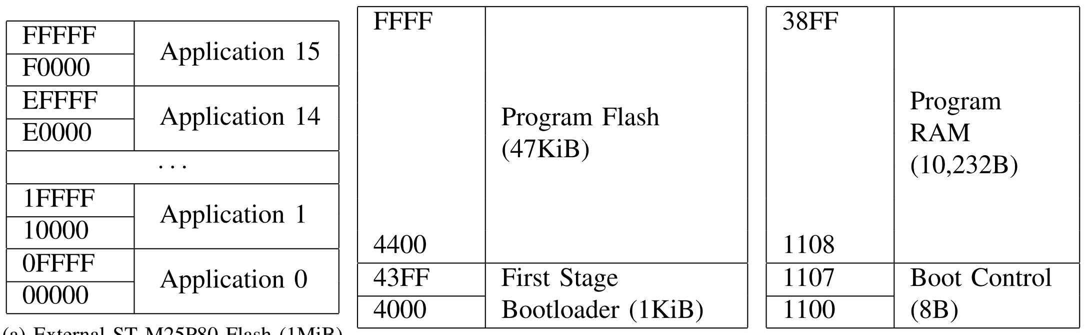
A. Detailed Design

For our design we will focus on the open-source TelosB mote platform. Storing and executing application code is necessarily closely tied to the hardware platform, however, the general approach could be modified to support other mote platforms. At the heart of the TelosB is the Texas Instruments MSP430F1611 microcontroller (MCU) [10]. This provides 48KiB program memory and 10KiB RAM. Application code images are stored on an external 1MiB flash chip [11]. The external flash memory is split into sixteen 64KiB segments. To simplify addressing we assign one application to each of the 16 flash segments. To avoid wasting the $64\text{KiB} - 48\text{KiB} = 16\text{KiB}$ of extra flash memory, the application could use this extra space for persistent data storage. When an application switch is requested, the bootloader copies the program memory segment from external flash to the internal program memory and reboots.

Figure 2 shows the final memory map used on the TelosB platform. The external memory can hold up to 16 applications. The on-chip flash has a 1KiB block reserved for the first stage bootloader. Finally, the first 8 bytes of RAM are reserved for a boot control data structure.

The First Stage Bootloader: The First Stage Bootloader is located at the beginning of program memory (0x4400); the reset interrupt vector always points to the start of the first stage bootloader. This stage runs entirely with interrupts disabled so the remaining interrupt vectors are irrelevant. This forces the first stage bootloader to run on every reset. Its only job is to read the full bootloader from external flash memory and copy it into RAM. The external flash memory is connected on an serial peripheral interface (SPI) bus which requires the overhead of configuring the MCU's USART for SPI mode and then sending the read data command to the external flash. We arbitrarily assume the second stage bootloader is stored in external flash segment 15 (the last segment). After copying the code into RAM the first stage bootloader has the final task of updating the interrupt vectors. This is necessary because the second stage bootloader uses interrupts for buffered serial communication. On the MSP430 the interrupt vectors are stored at the end of program memory (0xFFE0 - 0xFFFF). The second stage bootloader's interrupt vectors are read from external flash and then written into program memory. The reset vector is changed to the first stage bootloader's own start address (0x4400), so that it runs on every reset. Finally, the first stage bootloader sets the program counter to the start address of the second stage bootloader in RAM. This causes the next instruction executed by the CPU to be the start of the second stage bootloader application.

The Second Stage Bootloader: The second stage bootloader executes from RAM avoiding the need to reserve any program



(a) External ST M25P80 Flash (1MiB)
(b) Internal TI MSP430 Flash (48KiB)
(c) Internal TI MSP430 RAM (10KiB)

Fig. 2: TelosB Memory Map

memory and allowing the entire program memory to be rewritten (you cannot modify a memory block that contains the current program counter). To determine if an automated application switch is requested an 8-byte boot control structure is located at the beginning of RAM (0x1100). Because this structure resides in RAM, the last word in the block is set to a magic number to indicate if the data is valid or not. The other values are a flag to indicate an application switch is requested and if so, which application. If the data is not valid the bootloader waits a few seconds to allow entry into an interactive mode. This is indicated by receiving the string '!!!' from the serial port. If the string is not received, the first valid application is selected from external flash, loaded into program memory, and executed. If the data in the boot control block is valid, a second parameter contains the segment containing the requested application. In this case, if the specified application is valid in external flash, the application is loaded into program memory and executed; otherwise, the first valid application is used instead. If no application switch is requested, the last application executed is selected.

The interactive mode implements a simple shell that supports the four commands: *ls*, *rm*, *run*, and *TFTP*.

- *ls* returns a 16-bit value where a 1 represents a valid code image. For example, the value 0x8420 has ones at bit positions 5, 10, and 15 meaning applications 5, 10, and 15 are valid in external flash.
- *rm* takes one parameter, e.g.: 'rm 5', and will erase the corresponding segment in external flash memory.
- *run* also takes one parameter, e.g.: 'run 10', and loads the corresponding application into program memory and begins execution.
- Finally, TFTP enables transferring binary images to and from external flash memory. The actual implementation follows the same general approach as RFC1350[12] with a few changes. First, we use 128 byte blocks to reduce the required buffer sizes. Secondly, the RRQ/WRQ packets send a binary word corresponding to the flash memory segment requested rather than a filename; the mode parameter is left out entirely. The TFTP protocol specifies a 16-bit CRC on each block of data ensuring there were no errors in the download. At the completion of the TFTP session, an image descriptor is created and stored at offset 0 in the flash memory segment of the application. The descriptor is used to later validate the code image.

B. Analysis of Internal Flash Memory Lifetime

Each application switch requires the on-chip flash memory to be erased and reprogrammed. Therefore, we must carefully consider the lifetime of this flash memory. The MSP430 datasheet specifies the flash memory endurance as 10,000 cycles minimum and 100,000 cycles nominally. Each reboot causes the last page in flash memory to be erased twice. The first time is when the second stage bootloader is loaded and the second is when the application is loaded. This results in at lifetime of at least 5,000 reboot cycles. If application switches occur once every hour, the flash could wear out in as little as 5,000 hours, or about 200 days. However, nominally we would expect about 2,000 days, or more than 5 years.

Because of the wide variation in flash endurance it is difficult to choose a single good value for maximum switching frequency. A good rule of thumb is switching a few times a day is no problem, but a few times an hour is not recommended. Switching frequencies between these two values is a grey area.

The lifetime could be doubled by rewriting the second stage bootloader to avoid using interrupts at the expense of not using a buffered serial port. If implemented, this would result in only one flash erase operation per application switch instead of two. However, it could impact the time needed to load applications over the serial port because a lower baud rate may be needed. This change could be made later without affecting the operation of the bootloaders.

C. Analysis of Overhead

To understand the performance of dynamic reprogramming in sensor networks, we now analyze the theoretical performance for the TelosB platform. Our goal is to characterize the overhead imposed by our application switching framework. There are two types of overhead: resources (memory) and time to complete the application switch. The resource requirements are difficult to estimate, so we will only present actual results in Section VI.

The time to complete an application switch is constrained by the performance of the flash memory. To complete an application switch, data must be read from external flash memory and written to the internal flash memory. The MSP430 internal internal memory is segmented into 512 byte blocks. An application switch requires three steps: 1) erase internal flash, 2) read external flash, and 3) write internal flash. The read/write step is further split into 96 iterations of 512 bytes to rewrite the entire on-chip flash memory ($96 * 512 = 48Ki$). This calculation represents the time to reprogram the on-chip flash memory, ignoring the time needed to load our two-stage bootloader; it is the ideal case with zero additional overhead.

Each step in the process of switching applications is dependent on CPU frequency, which on the TelosB is selectable using a DCO (digitally controller oscillator) and represented as f_{DCOCLK}. The on-chip flash memory's timings are based on a dedicated clock with frequency f_{FTG} which is a multiple of f_{DCOCLK}. The time to erase a single flash segment is $\frac{4819}{f_{FTG}}$ and writing a full segment is: $\frac{35*256}{f_{FTG}}$. Reading the external flash memory uses the SPI bus. A 5-byte command begins the read sequence and one bit is transfered each SPI clock with frequency f_{SCLK}, therefore a read of 512 bytes takes $\frac{8*(5+512)}{f_{SCLK}}$. The minimum application switch time then can be computed as follows. To get a sense of this time, we use the values for each parameter from our final implementation: $f_{DCOCLK} = 4032000$, $f_{FTG} = 268800$, and $f_{SCLK} = 2016000$.

$$T_{switch_{min}} = 96 * (\frac{13779}{f_{FTG}} + \frac{4136}{f_{SCLK}}) = 5.12 \text{ seconds}$$

V. Usage of Our Two Stage Bootloader

Now that we have described our two-stage bootloader architecture, the last question is how this affects an application designer? Because the bootloaders require certain reserved portions of memory, we need to ensure they are not used by any other application. This is easy to accomplish by modifying the linker script when compiling the application. The two reserved areas are: 0x4000-0x4400 in program memory and 0x1100 - 0x1108 in RAM. The application can then be compiled using the modified linker script and it will not locate anything in the reserved areas.

An application designer also needs to know how to control application switching. This is accomplished by writing data into the boot control block. The second stage bootloader examines the boot control block on every reset and loads the requested application. If there is no requested application or the application cannot be verified the first valid application is loaded instead.

To load an application on the sensor node, we would normally just download the code to the target platform. This would download the application to the on-chip flash memory and allow it to run. However, it would overwrite the interrupt vectors and bypass the first stage bootloader. In addition, it would not be loaded into the external flash memory. To download the application to the external flash memory, we need to reset the device and enter the interactive mode of the second stage bootloader. This is easy to do on the TelosB by using the on-chip bsl (boot strap loader) to cause a reset. Then the interactive bootloader entry command ("!!!") is sent from a host PC. When this is received by the second stage bootloader it enters the interactive mode. The various commands can now be used to list, erase, download, or run a particular application. We have developed tools that automate this entire process.

In summary, there are only a few extra steps to using dynamic application switching:

1) Modify the linker script;
2) Switch by writing data to the boot control block and rebooting the node; and
3) Download code using the interactive bootloader.

VI. Performance Evaluation

To demonstrate the usefulness of our application switching approach we have implemented two sample applications and tested both synchronous and asynchronous switching in our testbed of 52 TelosB motes. The motes are approximately uniformly distributed in an large rectangular building as a 4 by 13 grid. Both stages of the bootloader are implemented in C while the applications are implemented with TinyOS 2.1.0.

A. Sample Applications

We implemented a temperature monitoring application (TempMon) and a light monitoring (LightMon) application. In TempMon, the motes report sensed temperature value once every 30 seconds. Because we do not expect rapid environmental changes, this sampling rate is more than sufficient. In LightMon, the light signal is monitored to detect motion. This is done by rapidly sampling the light sensor and reporting a binary result (motion or not) once every 5 seconds. Our motion detection algorithm works by counting the number of samples that lie outside the previous interval's mean + 3 * standard deviations. If the number of outlying samples is greater than 1/128'th of the total number of samples the node reports motion. This algorithm was developed experimentally and is good at detecting shadows moving over the light sensor.

B. Overhead

There are two types of overhead imposed by application switching: time and memory. We start by examining the application switching times. The Stage 0 bootloader, loads main bootloader into RAM in 912.0 milliseconds. Measured by using an oscilloscope attached to a GPIO sampling at 50K samples per second. At the start of the program the GPIO is set and then cleared at the end. Repeated several times with no more than ± 0.2 millisecond variation. Then the main bootloader loading an application into flash memory takes 5.760 seconds. Measured by using an oscilloscope attached to a GPIO sampling at 2.5K samples per second. Repeated several times with no more than ± 10 millisecond variation. Therefore, the total application switch time is the sum of both bootloaders, which is 6.672 seconds. This is somewhat longer than the minimum application switch time computed in

Section IV, however we leave further optimization for future work.

The memory overhead is reduced by using the two-stage bootloader because only the first stage needs to be in program memory. The second stage is loaded from external flash into RAM only when needed. The first stage bootloader requires 806 bytes of program memory. On the MSP430 program memory is organized as 1KiB segments, so it is natural to round this up to 1 KiB to allow for future growth. Additionally this simplifies loading applications because this sector never contains application code and therefor never needs to be rewritten. The second stage requires 6,700 bytes of program memory and 8 bytes of RAM is used to pass data (such as which application to run) into the bootloader from the application. Our platform has 48KiB program memory and 10KiB RAM. Therefore, the total memory overhead imposed by application switching is about 2% of program memory and less than 0.1% of RAM.

C. *Initial Result of a Controlled Experiment*

Before we implement application switching, we randomly assigned 4 nodes to TempMon and the remaining 48 to LightMon and collected data for several days. These numbers were chosen because we expect very consistent values from TempMon and more variation from LightMon. The results collected are shown in Figure 3. All values are averaged over a 15-minute window for display. The reported average temperatures vary from about 91 to 98 degrees. The absolute value is not very helpful because the sensors were not calibrated beforehand. However, we can clearly see a +6 degree shift at night. The LightMon values are plotted as the percent of all samples indicating activity in the sampling window. This reading turns out to be very noisy in the actual lab conditions. There are also large spikes visible during the first and third nights. As it turns out when the lights are turned off the sensed light values have little variation and the computed standard deviation approaches zero causing the motion detection algorithm to incorrectly report activity. If our intent was to collect valid data these problems would need to be addressed. However, for our investigation of application switching they provide a reasonable data source, if not exactly what we expected.

D. *Results for Asynchronous Switching*

The goal of this experiment is to dynamically choose between TempMon and LightMon. Application switching is useful in this scenario for two reasons. First, LightMon uses more power. It continuously samples the temperature sensor at approximately 400 Hz and updates a running mean and variance which requires one multiply and one divide operation, so there is no time for the mote to sleep. TempMon samples the sensor and reports the raw values once every 30 seconds, leaving a lot of idle time to sleep. We can increase the lifetime of the network by periodically switching to the TempMon application. Second, by switching we can collect both temperature and motion information from each node.

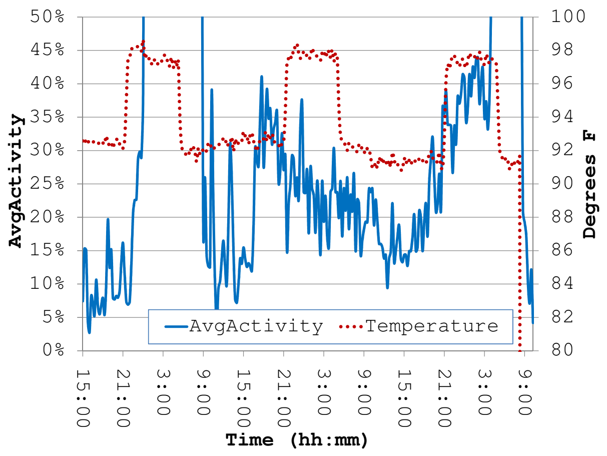


Fig. 3: Results from a static multi-application deployment

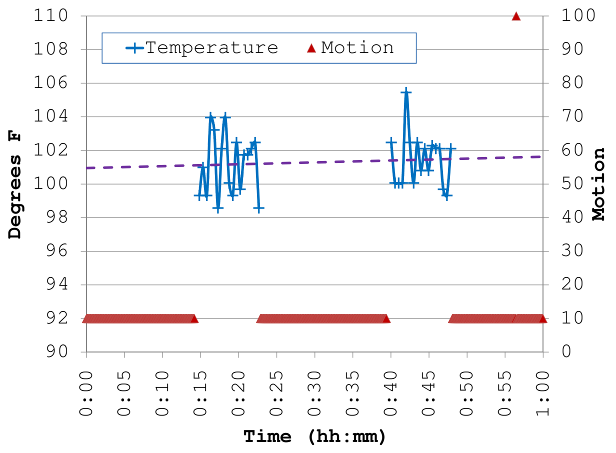
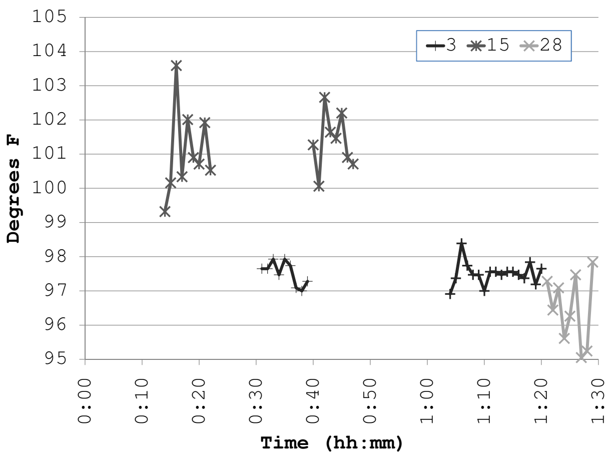
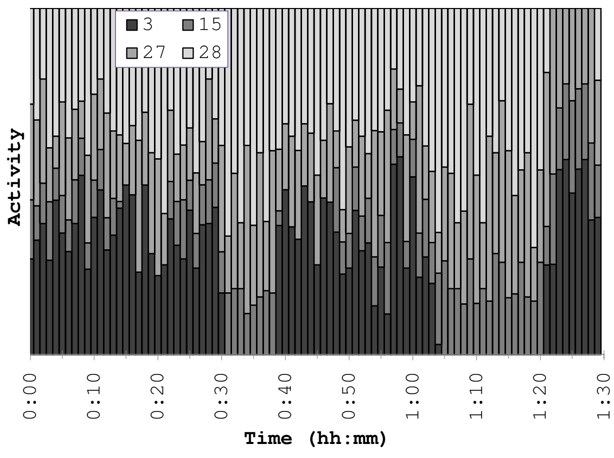


Fig. 4: One node with asynchronous switching

In this experiment we arbitrarily decide that we want one application switch per hour with a total of approximately 4 TempMon nodes at any time. This could be solved by computing a global switching schedule, however by using a randomized algorithm we can achieve similar results with much less overhead. The randomized algorithm chooses an application to run independently with $Pr(TempMon) = \frac{4}{52}$. This results in 4 nodes selecting TempMon, on average. To realize one switch per hour we need to compute a switching interval. Each time the switching interval elapses we repeat the application choice where $Pr(TempMon) = \frac{4}{52}$. The interval computation, where N is the number of switches in one hour becomes $N * Pr(Switch) = 1$. The value $Pr(switch)$ is the probability an application switch occurs, this is when the node transitions from TempMon to LightMon or vice versa. If the node was running TempMon and randomly chooses TempMon again a switch does not occur. Since we know $Pr(LightMon) = \frac{48}{52}$ and $Pr(TempMon) = \frac{4}{52}$



(a) Temperature



(b) Light

Fig. 5: Four nodes with asynchronous switching

we can compute $Pr(Switch) = 2 * Pr(LightMon) * Pr(TempMon) = \frac{24}{169}$. Then solving we get $N = \frac{169}{24}$ switches per hour, this is $\frac{24}{169} * 3600 = 511.2426...$ seconds between potential switches. We round this up to 512 seconds and randomly choose a new application. If the application randomly selected is different, the node reboots and runs the other application.

One Node with asynchronous switching. Figure 4 shows data from a single node (id=15). Temperature values are plotted with a + (plus) sign in degrees Fahrenheit; a linear trend line (dashed) is fit to this data. Output from LightMon is binary so we map this as [false=10 and true=100]. In this data there is only one motion event detected at 56:28. There are four application switches in this data; two from LightMon to TempMon and two from TempMon to LightMon. We can estimate the switching time by computing the duration between receiving two different type packets at the basestation. This value includes the switching time plus the time taken for the application to send the first data packet. Because this measured time includes several delays in addition to the switching delay it is useful only to understand the delays that would be present in real sensed data when employing application switching.

The measured packet delays for each transition (in seconds) are: 40, 13, 39, and 13. The variation in these numbers is due to the report interval for each application. The first and third transitions are from LightMon to TempMon. The report interval for TempMon is 30 seconds which accounts for 30 seconds of the 40 second delay to receive the first packet. As we have previously measured the actual switching time to be under 7 seconds we can surmise the extra 3 second delay is due to the node startup, connection to the network, and possibly routing delays before the first packet is received. The TempMon to LightMon transition was 13 seconds in both cases. Therefore, these two samples demonstrate only a 1 second additional startup delay. A detailed analysis of startup delays are beyond the scope of this paper. However, this result demonstrates that application startup times, which are typically ignored, may become significant when applied to application switching.

From this data we can also see that a single node can execute two applications and provide more information than running a single application alone. The temperature data is sufficient to compute long-term trends without sacrificing the ability to collect motion data. The long term temperature trend is shown by the fit line (dashed). We clearly see an approximate 1 degree per hour temperature rise while still spending most of the time looking for motion.

Four nodes with asynchronous switching. We have previously looked at the data from a single node (id=15). The next step in exploring our data is to examine data from neighboring nodes. For these results we use nodes 3, 15, 27, 28, which include the three spatially closest nodes to 15. The data has been processed into one-minute blocks. Light and temperature values are averaged over the one-minute interval.

Figure 5a shows the temperature data from each node in our nodes of interest. Node 27 did not report any temperature data during this interval, so it is not shown. Nodes 3 and 28 supplement the data from 15 well. There are three gaps of 14, 8, and 16 minutes respectively. There is clearly a large temperature differential between the nodes; which can be explained because the temperature sensors were not calibrated and that actual temperature variations likely exist in our environment.

Figure 5b shows the LightMon output from each node on a stacked bar graph. For each one minute interval we should receive 12 binary LightMon values. These binary values are mapped as [false=10 and true=100] and averaged over the interval and then normalized to show the relative output at each node. This mapping was chosen to differentiate false

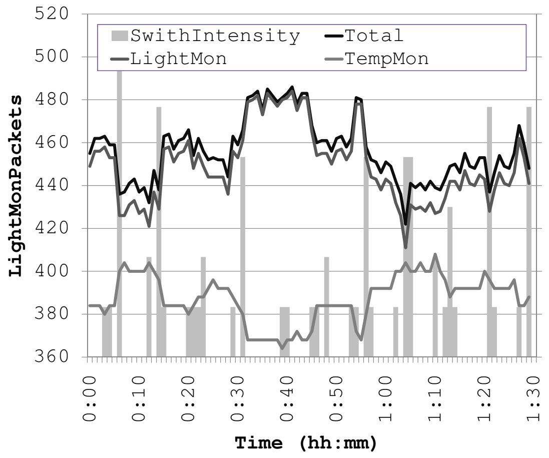


Fig. 6: Full network with asynchronous switching

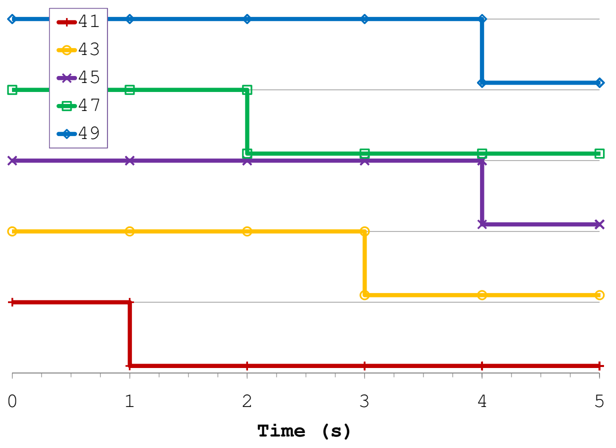


Fig. 7: Synchronous switching

from no data. We call this result 'activity' because when there is a lot of activity many nodes will detect motion. The data gaps are more easily visualized by comparing to 5a, when a node reports temperature value no motion values are reported. The relative activity level between the nodes can be explored. The result in terms of application switching is that because the gaps are relatively small and non-overlapping, we have reasonable activity samples in the neighborhood of node 15. So, if these nodes were part of a security monitoring application we have good coverage of this area while still executing dynamic application switches.

Full Network asynchronous switching. We have previously explored results from a single node and then a group of four nodes, finally, we will consider the network as a whole. Figure 6 shows the total number of packets received in each one-minute interval. The value for TempMon is shown on the secondary axis because this application sends fewer packets than LightMon. The results show significant variation; to explore this further we added another metric, 'SwitchIntensity' (bars). This value is computed from the total number of application switches in the interval normalized to use the full scale of the graph. With the addition of this metric the results become clear.

The first large spike in SwitchIntensity is at 0:07, during this interval the number of TempMon packets nearly doubles (therefore the number of LightMon nodes has fallen). Because TempMon sends packets every 30 seconds versus every 5 seconds for LightMon the total packet counts fall significantly. The next spike in SwitchIntensity is at 0:15 where the number of TempMon packets falls back to its previous value and the total packets return to their initial values. This pattern continues, however, at 1:05 there are two intervals back-to-back with high SwitchIntensity. This normally indicates a change in packet counts, however, the TempMon packets stay within one packet of the previous value. The total packets and LightMon packets show a large decrease at the same time. This result can be attributed to switching. Nearly the same number of nodes switched from LightMon to TempMon as TempMon to LightMon. The TempMon totals are not affected because the switching time is relatively small to the TempMon report interval. However, the switching time is significant when compared to the LightMon report interval. The end result is that we see a drop in LightMon packets and no change in TempMon packets.

E. *Results for Synchronous Switching*

To explore synchronous switching in the network we need to implement a way to coordinate the switching between nodes. Depending on the application there are two possible ways to accomplish this. If the application switching occurs in response to sensed data we could flood the sensor network with an application switch message. The second approach is useful for applications where switching is predetermined according to some schedule. In this case we can program each node with a common schedule and synchronize all nodes in the network to a global clock.

We chose the second approach to illustrate an application where all of the nodes follow the same global schedule. This could be the scenario where environmental data is collected during the day and switches to a security application at night. To implement this we added a node to act as the global timekeeper. This was necessary to maintain the global system time across application switches where all of the nodes in the network restart. The timekeeper node ran FTSP [13] and did not collect data or switch applications so that the time would remain consistent. Each of the applications included FTSP code to synchronize their local clocks to the global time and the same application switching schedule. We made a small change to the FTSP code so that the timekeeper node was always the root of the FTSP tree.

Figure 7 shows when (in seconds) the received packets change type (LightMon to TempMon) from five of the nodes in the network (41, 43, 45, 47, 49). Packets received by the base station are timestamped by the server with one second resolution. These nodes are arranged in a linear topology spanning the length of the deployment area. The timekeeper node was located nearest to node 41. Multihop communication is required to reach node 49. The variation of three seconds is reasonable because the decision to switch applications is made before sending a packet, which occurs once every 5 seconds in LightMon. Because this data was taken after many application switches, the data collection times would be randomly distributed in a five second window due to variations in node startup and connection to the network. By synchronizing the application switching to the FTSP global clock we could achieve better coordination of the application switches. However, if close synchronization is not required checking the application schedule at the report interval is sufficient.

VII. Conclusion

In this paper we presented a network-scale approach to multi-application deployment in wireless sensor networks. This has the advantage of allowing each application to exploit the full resources of a sensor node without worrying about interactions between different applications. Our two-stage bootloader requires only 1KiB of program memory and 8-bytes of RAM and there is little extra effort required for each application to support dynamic switching. At the same time applications can be developed, tested, and deployed without considering the other applications currently deployed on the sensor network. We have evaluated our approach on a testbed of 52 nodes using two different switching modes: asynchronous and synchronous. Both approaches ran for several days and executed thousands of application switches without failure. The collected data verified that the nodes were switching applications and the observed switching times were consistent with our calculations. We believe that for many sensor network applications dynamic application switching is a competitive alternative to other approaches for multiple application deployment.

References

[1] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *In Proceedings of the 2nd ACM international conference on Embedded networked sensor systems (SenSys '04)*, 2004.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han, "Mantis: System support for multimodal networks of in-situ sensors," in *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '03)*, 2003.

[3] A. Dunkels, B. Grnvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, 2004. [Online]. Available: http://www.sics.se/~adam/dunkels04contiki.pdf

[4] TinyOS Alliance, "TinyOS 2.1 adding threads and memory protection to TinyOS," in *In Proceedings of the 6th ACM conference on Embedded network sensor systems (SenSys '08)*, 2008. [Online]. Available: http://dx.doi.org/10.1145/1460412.1460479

[5] R. Barry, "FreeRTOS.org Features," http://www.freertos.org/FreeRTOS_Features.html, access date: April 2009.

[6] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, 2002.

[7] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun, "Supporting concurrent applications in wireless sensor networks," in *In Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06)*, 2006.

[8] G. Hackmann, C. liang Fok, G. catalin Roman, and C. Lu, "Agimone: Middleware support for seamless integration of sensor and ip networks," in *Proceedings of the 2nd International Conference on Distributed Computing in Sensor Systems (DCOSS '06)*, 2006.

[9] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler, "The tenet architecture for tiered sensor networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, 2006.

[10] Texas Instruments, "SLAS368E - MSP430x15x, MSP430x16x, MSP430x161x Mixed Signal Microcontroller," 2006.

[11] STMicroelectronics, "M25P80 8Mbit, Low Voltage, Serial Flash Memory With 25MHz SPI Bus Interface," 2002.

[12] K. Sollins, "The TFTP Protocol (revision 2)," RFC1350, 1992.

[13] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *In Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys '04)*, 2004.