A Dynamic Operating System for Sensor Nodes

Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler and Mani Srivastava

University of California, Los Angeles

{*simonhan, roy, kohler*}*@cs.ucla.edu,* {*ram, mbs*}*@ucla.edu*

ABSTRACT

Sensor network nodes exhibit characteristics of both embedded systems and general-purpose systems. They must use little energy and be robust to environmental conditions, while also providing common services that make it easy to write applications. In TinyOS, the current state of the art in sensor node operating systems, reusable components implement common services, but each node runs a single statically-linked system image, making it hard to run multiple applications or incrementally update applications. We present SOS, a new operating system for mote-class sensor nodes that takes a more dynamic point on the design spectrum. SOS consists of dynamically-loaded modules and a common kernel, which implements messaging, dynamic memory, and module loading and unloading, among other services. Modules are not processes: they are scheduled cooperatively and there is no memory protection. Nevertheless, the system protects against common module bugs using techniques such as typed entry points, watchdog timers, and primitive resource garbage collection. Individual modules can be added and removed with minimal system interruption. We describe SOS's design and implementation, discuss tradeoffs, and compare it with TinyOS and with the Maté virtual machine. Our evaluation shows that despite the dynamic nature of SOS and its higher-level kernel interface, its long term total usage nearly identical to that of systems such as Matè and TinyOS.

1 INTRODUCTION

Wireless sensor nodes—networked systems containing small, often battery-powered embedded computers—can densely sample phenomena that were previously difficult or costly to observe. Sensor nodes can be located far from networked infrastructure and easy human accessibility, anywhere from the forest canopy [13] to the backs of zebras [12]. Due to the difficulty and expense of maintaining such distant nodes, wireless sensor networks are expected to be both autonomous and long-lived, surviving environmental hardships while conserving energy as much as possible. Sensor network applications will change in the field as well; data returned from the network can influence followup experiment design, and long-lived networks will necessarily be retasked during their lifetimes.

All of this argues for a general-purpose sensor network

operating system that cleanly supports dynamic application changes. But sensor nodes are embedded systems as well as general-purpose systems, introducing a tension between resource and energy constraints and the layers of indirection required to support true general-purpose operating systems. TinyOS [15], the state-of-the-art sensor operating system, tends to prioritize embedded system constraints over general-purpose OS functionality. TinyOS consists of a rich collection of software components written in the NesC language [8], ranging from low-level parts of the network stack to application-level routing logic. Components are not divided into "kernel" and "user" modes, and there is no memory protection, although some interrupt concurrency bugs are caught by the NesC compiler. A TinyOS system image is statically linked at compile time, facilitating resource usage analysis and code optimization such as inlining. However, code updates become more expensive, since a whole system image must be distributed [10].

This paper shows that sensor network operating systems can achieve dynamic and general-purpose OS semantics without significant energy or performance sacrifices. Our operating system, SOS, consists of a common kernel and dynamic application modules, which can be loaded or unloaded at run time. Modules send messages and communicate with the kernel via a system jump table, but can also register function entry points for other modules to call. SOS, like TinyOS, has no memory protection, but the system nevertheless protects against common bugs. For example, function entry points are marked with types using compressed strings; this lets the system detect typing errors that might otherwise crash the system, such as when a module expecting a new version of an interface is loaded on a node that only provides the old. SOS uses dynamic memory both in the kernel and in application modules, easing programing complexity and increasing temporal memory reuse. Priority scheduling is used to move processing out of interrupt context and provide improved performance for time-critical tasks.

We evaluate SOS using both microbenchmarks and the application-level Surge benchmark, a well-known multi-hop data acquisition program. Comparisons of Surge versions running on SOS, TinyOS, and Maté Bombilla virtual machine [14] show comparable CPU utilization and radio usage; SOS's functionality comes with only a minor energy cost compared to TinyOS. Evaluation of code distribution mechanisms in the same three sys-

tems shows that SOS provides significant energy savings over TinyOS, and more expressivity than Maté Bombilla. However, analytical comparisons of the total energy consumed by the three systems over an extended period of time reveals that general operating costs dwarf the above differences, resulting in nearly identical total energy consumption on the three systems.

The rest of the paper is structured as follows. Section 2 describes other systems that have influenced the SOS design. A detailed look at the SOS architecture is presented in Section 3, including key differences between SOS and TinyOS. A brief walk through of an SOS application is presented in Section 4. An in-depth evaluation of SOS, including comparisons to TinyOS and Maté Bombilla, is presented in Section 5. Section 6 closes with ending remarks and directions for future work.

2 RELATED WORK

SOS uses ideas from a spectrum of systems research in both general purpose operating system methods and embedded systems techniques.

2.1 Traditional Operating Systems

SOS provides basic hardware abstractions in a core kernel, upon which modules are loaded to provide both higher level functionality. This is similar to microkernel abstractions explored by the Mach [19] operating system and the Exokernel [6]. Mach modularizes low layer kernel services to allow easy customization of the system. The Exokernel uses a minimal hardware abstraction layer upon which custom user level operating environments can be created. SOS also uses a small kernel that provides interfaces to the underlying hardware at a level lower than Mach and higher than the Exeokernel.

SOS's typed communication protocols were inspired partially by SPIN [2], which makes kernel extensions safe through the use of type safe languages.

2.2 Sensor Network Operating Systems

TinyOS [9] is the de facto standard operating system for sensor nodes. TinyOS is written using the NesC language [8] and provides an event driven operating environment. It uses a component model for designing sensor network applications. At compile time, a component binding configuration is parsed, elements are statically bound and then compiled to make a binary image that is programmed into the flash memory on a sensor node. TinyOS's static binding facilitates compile time checks. Like TinyOS, SOS is event driven and uses a component model, but SOS components can be installed and modified after a system has been deployed.

Our experiences working with TinyOS helped to motivate the development and design decisions of SOS.

As described in section 5.1, approximately 21% of the Mica2 specific driver code and 36% of the AVR specific code included in the current SOS distribution is ported directly from TinyOS.

The need for flexible systems has inspired virtual machines (VM), mobile agents, and interpreters, for sensor nodes. One such system is Maté [14]. Maté implements a simple VM architecture that allows developers to build custom VMs on top of TinyOS and is distributed with Bombilla, which demonstrates a VM implementation of the Surge protocol. Sensorware [3] and Agilla [7] are designed to enable mobile agent abstractions in sensor networks. Unfortunately, these approaches can have significant computational overhead, and the retasking of the network is limited by the expressibility of the underlying VM or agent system. SOS occupies a middle ground with more flexibility and less CPU overhead than VMs, but also higher mote reprogramming cost.

MANTIS [1] implements a lightweight subset of the POSIX threads API targeted to run on embedded sensor nodes. By adopting an event driven architecture, SOS is able to support a comparable amount of concurrency without the context switching overhead of MANTIS.

2.3 Dynamic Code

Traditional solutions to reprogrammable systems target resource rich devices. Loadable modules in Linux share many properties with SOS, but benefit from running on systems that can support complex symbol linking at run time. Impala [17] approaches the devices SOS targets by implementing dynamic insertion of code on devices similar to PDAs. At any given time, Impala middleware chooses a single application to run based on the state of application specific and global node state, allowing Impala to execute an application that is the best fit to current conditions. In contrast to this, SOS both expects and supports multiple modules executing and interacting on top of the SOS kernel at the same time. An operating system for 8-bit devices developed at the same time as SOS and supporting modular dynamic code updates is Contiki [5]. Contiki uses runtime relocation to to update binaries loaded onto a node, as opposed to the completely position independent code used by SOS. This results a different low layer design choices in Contiki and SOS.

XNP [4] is a mechanism that enables over the air reprogramming of the sensor nodes running TinyOS. With XNP a new image for the node is stored into an external flash memory, then read into program memory, and finally the node is rebooted. SOS module updates cannot replace the SOS kernel, but improve on XNP's energy usage by using modular updates rather than full binary system images, not forcing a node to reboot after updates, and installing updates directly into program memory without expensive external flash access.

Low overhead solutions using differential patching are proposed in [20] and [11]. In these schemes a binary differential update between the original system image and the new image is created. A simple language is used to encode this update in a compressed form that is then distributed through the network and used to construct a new system image on deployed nodes. This technique can be used at the component level in SOS and, since changes to the SOS kernel or modules will be more localized than in tightly coupled systems, result in smaller differentials.

MOAP [22] and Deluge [10] are two protocols that hove been used to distribute new system images to all nodes in a sensor network. SOS currently includes a publish-subscribe scheme that is similar to MOAP for distributing modules within a deployed sensor network. Depending on the user's needs SOS can use MOAP, Deluge, or any other code distribution protocol. SOS is not limited to running the same set of modules or same base kernel on all nodes in a network, and will benefit from different types of distribution protocols including those supporting point-to-point or point-to-region transfer of data.

3 SYSTEM ARCHITECTURE

In addition to the traditional techniques used in embedded system design, the SOS kernel features dynamically linked modules, flexible priority scheduling, and a simple dynamic memory subsystem. These kernel services help support changes after deployment, and provide a higher level API freeing programmers from managing underlying services or reimplementing popular abstractions. Most sensor network application and protocol development occurs in modules that sit above the kernel. The minimal meaningful sensor network application using SOS consists of a simple sensing module and routing protocol module on top of the kernel.

Table 1 presents memory footprints of the core SOS kernel, TinyOS with Deluge, and Maté Bombilla virtual machine. All three of these configurations include a base operating environment with the ability to distribute and update the programs running on sensor nodes. SOS natively supports simple module distribution and the ability to add and remove modules at run time. TinyOS is able to distribute system images and reflash nodes using Deluge. The Maté Bombilla VM natively supports the transfer and execution of new programs using the Trickle [16] protocol. SOS is able to provide common kernel services to external modules in a footprint comparable to TinyOS running Deluge and a space smaller than the Maté Bombilla VM. Note that RAM usage in SOS is broken into two parts: RAM used by the core kernel and RAM reserved for the dynamic memory subsystem.

The following discussion takes a closer look at the key architectural decisions in SOS and examines them

Platform	ROM	RAM
SOS Core	20464 B	1163 B
(Dynamic Memory Pool)	-	1536 B
TinyOS with Deluge	21132 B	597 B
Bombilla Virtual Machine	39746 B	3196 B

Table 1—Memory footprint for base operating system with ability to distribute and update node programs compiled for the Mica2 Motes.
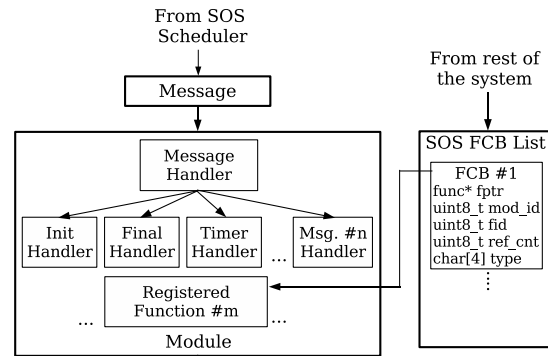
Figure 1—Module Interactions

in light of the commonly used TinyOS.

3.1 Modules

Modules are position independent binaries that implement a specific task or function, comparable in functionality to TinyOS components. Most development occurs at the module layer, including development of drivers, protocols, and application components. Modification to the SOS kernel is only required when low layer hardware or resource management capabilities must be changed. An application in SOS is composed of one or more interacting modules. Self contained position independent modules use clean messaging and function interfaces to maintain modularity through development and into deployment. The primary challenge in developing SOS was maintaining modularity and safety without incurring high overhead due to the loose coupling of modules. An additional challenge that emerged during development, described in more detail in section 3.1.4, was maintaining consistency in a dynamically changing system.

3.1.1 Module Structure

SOS maintains a modular structure after distribution by implementing modules with well defined and generalized points of entry and exit. Flow of execution enters a module from one of two entry mechanisms: messages delivered from the scheduler and calls to functions registered by the module for external use. This is illustrated in figure 1.

Message handling in modules is implemented using a module specific handler function. The handler function

Communication Method	Clock Cycles
Post Message Referencing Internal Data	271
Post Message Referencing External Buffer	252
Dispatch Message from Scheduler	310
Call to Function Registered by a Module	21
Call Using System Jump Table	12
Direct Function Call	4

Table 2—Cycles needed for different types of communication to and from modules in SOS when running on an Atmel AVR microcontroller. The delay for direct function calls within the kernel is listed as a baseline reference.

takes as parameters the message being delivered and the state of the module. All module message handlers should implement handler functions for the *init* and *final* messages produced by the kernel during module insertion and removal, respectively. The *init* message handler sets the module's initial state including initial periodic timers, function registration, and function subscription. The *final* message handler releases all node resources including timers, memory, and registered functions. Module message handlers also process module specific messages including handling of timer triggers, sensor readings, and incoming data messages from other modules or nodes. Messages in SOS are asynchronous and behave somewhat like TinyOS tasks; the main SOS scheduling loop takes a message from a priority queue and delivers the message to the message handler of the destination module. Inter-module direct function calls are used for module specific operations that need to run synchronously. These direct function calls are made possible through a function registration and subscription scheme described in section 3.1.2. Module state is stored in a block of RAM external to the module. Modules are relocatable in memory since: program state is managed by the SOS kernel, the location of inter-module functions is exposed through a registration process, and the message handler function for any module is always located an a consistent offset in the binary.

3.1.2 Module Interaction

Interactions with modules occur via messages, direct calls to functions registered by a module, and ker_* system calls into the SOS kernel. The overhead for each of these types of interactions is presented in table 2. Messaging, detailed in section 3.2, provides asynchronous communication to a module and enables scheduling by breaking up chains of execution into scheduled subparts. Messaging is flexible, but slow, so SOS provides direct calls to functions registered by modules, which bypass the scheduler to provide low latency communication to modules.

Function registration and subscription is the mechanism that SOS uses to provide direct inter-module communication and upcalls from the kernel to modules.

Function Registration Action	Clock Cycles
Register a Function	267
Deregister a Function	230
Get a Function Handle	124
Call to Function Registered by a Module	21

Table 4—Cycles needed for the function registration mechanism in SOS when running on an Atmel AVR microcontroller.
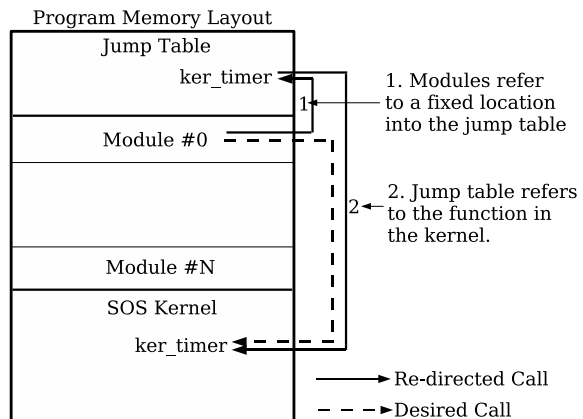
Figure 2—Jump Table Layout and Linking in SOS.

When explicitly registering functions with the SOS kernel, a module informs the kernel where in its binary image the function is implemented. The registration is done through a system call *ker_register_fn* described in table 3 with overheads detailed in table 4. A function control block (FCB) used to store key information about the registered function is created by the SOS kernel and indexed by the tuple {module ID, function ID}.

The FCB includes a valid flag, a subscriber reference count, and prototype information. The stored prototype information encodes both basic type information and whether a parameter contains dynamic memory that needs to undergo a change of ownership. For example, the prototype {'c', 'x', 'v', '1'} indicates a function that returns a signed character ('c') and requires one parameter ('1'). That parameter is a pointer to dynamically allocated memory ('x'). When a registered function is removed, this prototype information is used by kernel stub functions described in section 3.1.4.

The call *ker_get_handle* described in table 3 is used to subscribe to a function. The module ID and function ID are used as a tuple to locate the FCB of interest, and type information is checked to provide an additional level of safety. If the lookup succeeds, the kernel returns a pointer to the function pointer of the subscribed function. The subscriber should always access the subscribed function by dereferencing this pointer. This extra level of indirection allows the SOS kernel to easily replace the implementation of a function with a newer version by changing the function pointer in the FCB, without needing to update subscriber modules.

Prototype	Description
int8_t ker_register_fn(sos_pid_t pid, uint8_t fid, char *prototype, fn_ptr_t func)	Register function 'func' with type 'prototype' as being supplied by 'pid' and having function ID 'fid'.
fn_ptr_t* ker_get_handle(sos_pid_t req_pid, uint8_t req_fid, char *prototype)	Subscribe to the function 'fid' provided by module 'pid' that has type 'prototype'.

Table 3—Function Registration and Subscription API

Modules access kernel functions using a jump table. This helps modules remain loosely coupled to the kernel, rather than dependent on specific SOS kernel versions. Figure 2 shows how the jump table is setup in memory and accessed by a module. This technique also allows the kernel to be upgraded without requiring SOS modules to be recompiled, assuming the structure of the jump table remains unchanged, and allows the same module to run in a deployment of heterogeneous SOS kernels.

3.1.3 Module Insertion and Removal

Loading modules on running nodes is made possible by the module structure described above and a minimal amount of metadata carried in the the binary image of a module.

Module insertion is initiated by a distribution protocol listening for advertisements of new modules in the network. When the distribution protocol hears an advertisement for a module, it checks if the module is an updated version of a module already installed on the node, or if the node is interested in the module and has free program memory for the module. If either of the above two conditions are true, the distribution protocol begins to download the module and immediately examines the metadata in the header of the packet. The metadata contains the unique identity for the module, the size of the memory required to store the local state of the module, and version information used to differentiate a new module version. Module insertion is immediately aborted should the SOS kernel find that it is unable to allocate memory for the local state of the module.

A linker script is used to place the handler function for a module at a known offset in the binary during compilation, allowing easy linking during module insertion. During module insertion a kernel data structure indexed by the unique module ID included in the metadata is created and used to store the absolute address of the handler, a pointer to the dynamic memory holding the module state, and the identity of the module. Finally the SOS kernel invokes the handler of the module by scheduling an *init* message for the module.

The distribution protocol used to advertise and propagate module images through the network is independent of the SOS kernel. SOS currently uses a publish subscribe protocol similar to MOAP.

Module removal is initiated by the kernel dispatching a *final* message. This message provides a module a final opportunity to gracefully release any resources it is holding and inform dependent modules of its removal. After the *final* message the kernel performs garbage collection by releasing dynamically allocated memory, timers, sensor drivers, and other resources owned by the module. As described in section 3.1.4, FCBs are used to maintain system integrity after module removal.

3.1.4 Potential Failure Modes

The ability to add, modify, and remove modules from a running sensor node introduces a number of potential failure modes not seen in static systems. It is important to provide a system that is robust to these potential failures. While still an area of active work, SOS does provide some mechanisms to minimize the impact of potential failure modes that can result from dynamic system changes. These mechanisms set a global error variable to help inform modules when errors occur, allowing a module to handle the error as it sees fit.

Two potential modes of failure are attempting to deliver a scheduled message to a module that does not exist on the node, and delivering a message to a handler that is unable to handle the message. In the first case, SOS simply drops the message addressed to a nonexistent module and frees dynamically allocated memory that would normally undergo ownership transfer. The latter case is solved by the individual modules, which can choose custom policies for what to do with messages that they are unable to handle. Most modules simply drop these messages, return an error code, and instruct the kernel to collect dynamically allocated memory in the message that would have undergone a change of ownership.

More interesting failure modes emerge as a result of intermodule dependencies resulting from direct function calls between modules, including: no correct implementation of a function exists, an implementation of a function is removed, an implementation of a function changes, or multiple implementations of a single function exist.

A module's subscription request is successful if there exists a FCB that is tagged as valid and has the same module ID, function ID and prototype as that in the subscription. Otherwise the subscription request fails and the module is free to handle this dependency failure as it wishes. Actions currently taken by various SOS modules include aborting module insertion, scheduling a later attempt to subscribe to the function, and continuing to

execute with reduced functionality.

A subscription to a function can become invalid should the provider module be removed. When the supplier of a function is removed from the system, SOS checks if any other modules have registered to use the function. If the registration count is zero then the FCB is simply removed from the system. If the registration count is greater than zero, a control flag for the FCB is marked as being invalid to prevent new modules from subscribing and the implementation of the function is redirected to a system stub function. The system stub function performs expected memory deallocations as encoded in the function prototype and sets a global error variable that the subscriber can use to further understand the problem.

Problems can arise after a module has subscribed to a function if the implementation of the function changes due to updates to the provider module. Similar to when a module is removed, functions provided by a module are marked invalid during an update. When the updated module finishes installing, it registers new versions of the provided functions. SOS assumes that a function registration with the same supplier module ID, function ID, and prototype of an already existent FCB is an update to the function implementation, so the existent FCB is updated and the control flag is returned to being valid. This automatically redirects subscribers to the new implementation of the function. Changes to the prototype of a provided function are detected when the new implementation of the function is registered by the supplying module. This results in the old FCB remaining invalid with old subscribers redirected to a system stub and a new FCB with the same supplier module ID and function ID but different prototype information being created. New subscribers with the new prototype information are linked to the new FCB, while any attempts to subscribe to the function with the old prototype fail.

It is important to note that when SOS prevents an error in the situations described above, it typically returns an error to the calling module. The programmer of the module is responsible for catching these errors and handling them as they see fit. Providing less intrusive protection mechanisms is a fascinating and challenging problem that continues to be worked on as SOS matures.

3.2 Message Scheduling

SOS uses cooperative scheduling to share the processor between multiple lines of execution by queuing messages for later execution.

TinyOS uses a streamlined scheduling loop to pop function pointers off of a FIFO message queue. This creates a system with a very lean scheduling loop. SOS instead implements priority queues, which can provide responsive servicing of interrupts without operating in an interrupt context and more general support for passing parameters to components. To avoid tightly integrated modules that carefully manage shared buffers, a result of the inability to pass parameters through the messaging mechanism, messaging in SOS is designed to handle the passing of parameters. To mitigate memory leaks and simplify accounting, SOS provides a mechanism for requesting changes in data ownership when dynamically allocated memory is passed between modules. These design goals result in SOS choosing a more expressive scheduling mechanism at the cost of a more expensive scheduling loop. The API for messaging is SOS is shown in figure 5.
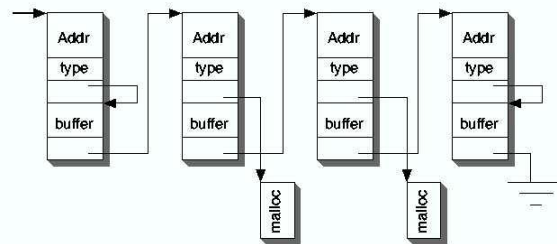

Figure 3—Memory Layout of An SOS Message Queue

Figure 3 provides an overview of how message headers are structured and queued. Message headers within a queue of a given priority form a simple linked list. The information included in message headers includes complete source and destination information, allowing SOS to directly insert incoming network messages into the messaging queue. Messages carry a pointer to a data payload used to transfer simple parameters and more complex data between modules. The SOS header provides an optimized solution to this common case of passing a few bytes of data between modules by including a small buffer in the message header that the data payload can be redirected to, without having to allocate a separate piece of memory (similar to the mbuf design). SOS message headers also include a series of flags to describe the priority of the message, identify incoming and outgoing radio messages, and describe how the SOS kernel should manage dynamic memory. These flags allow easy implementation of memory ownership transfer for a buffer moving through a stack of components, and freeing of memory on completion of a scheduled message.

SOS uses the high priority queue for time critical messages from ADC interrupts and a limited subset of timers needed by delay intolerant tasks. Priority queues have allowed SOS to minimize processing in interrupt contexts by writing interrupt handlers that quickly construct and schedule a high priority message and then drop out of the interrupt context. This reduces potential concurrency errors that can result from running in an interrupt context. Examples of such an errors include corruption of shared buffers that may be in use outside of the interrupt context, and stale interrupts resulting from operating with

Prototype	Description
int8_t post_short(sos_pid_t did, sos_pid_t sid, uint8_t type, uint8_t byte, uint16_t word, uint8_t flag)	Place a message on the queue to call function 'type' in module 'did' with data 'byte' and 'word'.
int8_t post_long(sos_pid_t did, sos_pid_t sid, uint8_t type, uint8_t len, void *data, uint8_t flag)	Place a message on the queue to call function 'type' in module 'did' with *data pointing to the data.

Table 5—Messaging API

Prototype	Description	Cycles
void *ker_malloc(uint16_t size, sos_pid_t id)	Allocate memory	69
void ker_free(void* ptr)	Free memory	85
int8_t ker_change_own(void *ptr, sos_pid_t id)	Change ownership	43
sos_pid_t ker_check_memory()	Validate memory (after a crash)	1175

Table 6—Cycles needed for dynamic memory tasks in SOS when running on an Atmel AVR microcontroller.

interrupts disabled.

3.3 Dynamic Memory

Due to reliability concerns and resource constraints, embedded operating systems for sensor nodes do not always support dynamic memory. Unfortunately, static memory allocation results in fixed length queues sized for worst case scenarios and complex program semantics for common tasks, such as passing a data buffer down a protocol stack. Dynamic memory in SOS addresses these problems. It also eliminates the need to resolve during module insertion what would otherwise be static references to module state.

SOS uses dynamic memory with a collection of bookkeeping annotations to provide a solution that is efficient and easy to debug. Dynamic memory in SOS uses simple best fit fixed-block memory allocation with three base block sizes. Most SOS memory allocations, including message headers, fit into the smallest block size. Larger block sizes are available for the few applications that need to move large continuous blocks of memory, such as module insertion. A linked list of free blocks for each block size provides constant time memory allocation and deallocation, reducing the overhead of using dynamic memory. Failed memory allocation returns a null pointer.

Queues and data structures in SOS dynamically grow and shrink at run time. The dynamic use and release of memory in SOS creates a system with effective temporal memory reuse and an ability to dynamically tune memory usage to specific environments and conditions. Self imposed hard memory usage limits prevent programing errors that could otherwise result in a module allocating all of the dynamic memory on a node.

Common memory functions and their clock cycle overheads are presented in table 6. All allocated memory is owned by some module on the node. This value is set during allocation and used by SOS to implement

basic garbage collection and watch for suspect memory usage. Modules can transfer memory ownership to reflect data movement. Dynamic memory blocks are annotated with a small amount of data that is used to detect basic sequential memory overruns. These features are used for post-crash memory analysis[1] to identify suspect memory owners, such as a module owning a great deal of system memory or overflowed memory blocks. SOS also supports the use of a hardware watchdog timer used to force a soft boot of an unresponsive node and triggering the same post-crash memory analysis. Finally, memory block annotations enable garbage collection on module unload.

3.4 Miscellaneous

The SOS kernel includes a sensor API that helps to manage the interaction between sensor drivers and the modules that use them. This leads to more efficient usage of a node's ADC and sensing resources. Other standard system resources include timer multiplexing, UART libraries, and hardware management of the I2C bus.

The loosely coupled design used in SOS has resulted in a platform that is very portable. SOS currently has support for the Mica2 and MicaZ motes from Crossbow, and the XYZ Node from Yale. The most difficult portion of a port tends to be the techniques for writing to program memory while the SOS core is executing.

Limitations on module development result from the loose coupling of modules. An example of this is a 4KB size limitation for AVR module binaries, since relative jumps used in position independent code on the AVR architecture can only jump up to 4KB. Moreover, modules cannot refer to any global variables of the SOS kernel, as their locations of may not be available at compilation time.

4 PROGRAMMING SOS APPLICATIONS

Figure 4 contains a source code listing of the Sample_Send module of the Surge application[2]. The program uses a single *switch* structure to implement the message handler for the Sample_Send module. Using the standard C programing language reduces the learning curve of SOS while taking advantage of the many compilers, development environments, debuggers, and other tools designed for C. C also provides efficient execution needed to operate on resource limited 8-bit microcontrollers.

```
01 int8_t module(void *state, Message *msg) {
02   surge_state_t *s = (surge_state_t*)state;
03   switch (msg->type){
04     //! System Message - Initialize module
05     case MSG_INIT: {
06       char prototype[4] = {'C', 'v', 'v', '0'};
07       ker_timer_start(SURGE_MOD_PID, SURGE_TIMER_TID,
08             TIMER_REPEAT, INITIAL_TIMER_RATE);
09       s->get_hdr_size = (func_u8_t*)ker_get_handle
10             (TREE_ROUTING_PID, MOD_GET_HDR_SIZE,
11              prototype);
12       break;
13     }
14     //! System Message - Timer Timeout
15     case MSG_TIMER_TIMEOUT: {
16       MsgParam *param = (MsgParam*) (msg->data);
17       if (param->byte == SURGE_TIMER_TID) {
18         if (ker_sensor_get_data(SURGE_MOD_PID, PHOTO)
19            != SOS_OK)
20           return -EFAIL;
21       }
22       break;
23     }
24     //! System Message - Sensor Data Ready
25     case MSG_DATA_READY: {
26       //! Message Parameters
27       MsgParam* param = (MsgParam*) (msg->data);
28       uint8_t hdr_size;
29       uint8_t *pkt;
30       hdr_size = (*(s->get_hdr_size))();
31       if (hdr_size < 0) return -EINVAL;
32       pkt = (uint8_t*)ker_malloc
33             (hdr_size + sizeof(SurgeMsg), SURGE_MOD_PID);
34       s->smsg = (SurgeMsg*)(pkt + hdr_size);
35       if (s->smsg == NULL) return -EINVAL;
36       s->smsg->reading = param->word;
37       post_long(TREE_ROUTING_PID, SURGE_MOD_PID,
38             MSG_SEND_PACKET, length, (void*)pkt,
39             SOS_MSG_DYM_MANAGED);
40       break;
41     }
42     //! System Message - Evict Module
43     case MSG_FINAL: {
44       ker_timer_stop(SURGE_MOD_PID, SURGE_TIMER_TID);
45       ker_release_handle(s->get_hdr_size);
46       break;
47     }
48     default:
49     return -EINVAL;
40   }
51   return SOS_OK;
52 }
```

Figure 4—Surge Source Code

Line 2 shows the conversion of the generic state stored in and passed from the SOS kernel into the module's internal representation. As noted in section 3.1.1, the SOS kernel always stores a module's state to allow modules to be easily inserted at runtime.

An example of the *init* message handler appears on lines 5-13. These show the module requesting a periodic timer from the system and subscribing to the MOD-_GET_HDR_SIZE function supplied by the tree routing module. This function pointer is used on line 30 to find the size of the header needed by the underlying routing layer. By using this function, changes to the underlying routing layer that modify the header size do not break the application. Lines 43-47 show the *final* message handler releasing the resources it had allocated: the kernel timer and the subscribed function. If these resources had not been explicitly released, the garbage collector in the SOS kernel would have soon released them. Good programing style is observed on lines 31 and 35 where potential errors are caught by the module and handled.

5 EVALUATION

Our initial performance hypothesis was that SOS would perform at roughly the same level as TinyOS in terms of latency and energy usage, despite the greater level of functionality SOS provides (and the overhead necessary to support that functionality). We also hypothesized that SOS module insertion would be far less expensive in terms of energy than the comparable solution in TinyOS with Deluge. This section tests these hypotheses using a prototypical multihop tree builder and data collector called Surge. A baseline evaluation of the operating systems is also performed.

Benchmarking shows that application level performance of a Surge-like application on SOS is comparable to Surge on TinyOS and to Bombilla, an instantiation of the Maté virtual machine specifically for the Surge application. Initial testing of CPU active time in baseline evaluations of SOS and TinyOS showed a significant discrepancy between the two systems, motivating an in depth search into the causes of overhead in SOS. This search revealed simple optimizations that can be applied to improve both systems and bring their baseline performance to almost the same level. The process of remotely updating an application's functionality in SOS is more energy efficient than updates using Deluge in TinyOS, but less efficient than updates using Bombilla. While our hypotheses that SOS can more efficiently install updates than TinyOS with Deluge proves true, further analysis shows that this difference does not significantly impact the total energy usage over time and that the three systems have nearly identical energy profiles over long time scales. From these results, we argue that SOS effectively provides a flexible solution for sensor networking with energy usage functionally equivalent to other state of the art systems.

5.1 Methodology

We first describe the workings of the Surge application. Surge nodes periodically sample the light sensor and send that value to the base station over multi-hop wireless links. The route to the base station is determined by setting up a spanning tree; every node in the tree maintains only the address of its parent. Data generated at a node is always forwarded to the parent in the spanning tree. Surge nodes choose as their parent the neighbor with the least hop count to the base station and, in the event of a tie, the best estimated link quality. The estimate of the link quality is performed by periodically broadcasting beacon packets containing neighborhood information.

The Surge application in SOS is implemented with three modules: Sample_Send, Tree_Routing and Photo-Sensor. For SOS, a blank SOS kernel was deployed on all motes. The Sample_Send, Tree_Routing and Photo-Sensor modules were injected into the network and remotely installed on all the nodes. For TinyOS, the nodes were deployed with the Deluge GoldenImage. The Surge application was injected into the network using the Del-

Type of Code	Percentage
Kernel	0%
Mica2 Specific Drivers	21%
AVR Specific Drivers	36%

Table 7—Approximate number of lines of code (including comments) used from TinyOS in different parts of SOS.

uge protocol and remotely installed on all the nodes. For Maté, the Bombilla VM was installed on all the nodes. Bombilla VM implements the tree routing protocol natively and provides a high-level opcode, Send, for transferring data from the node to the base station via a spanning tree. Bytecode for periodically sampling and sending the data to the base station was injected into the network and installed on all relevant nodes.

In all versions of Surge, the routing protocol parameters were set as follows: sensor sampling happens every 8 seconds, parents are selected and route updates broadcast every 5 seconds, and link quality estimates are calculated every 25 seconds. Therefore, upon bootup, a node will have to wait at least 25 seconds before it performs the first link quality estimate. Since the parent is selected based on link quality, the expected latency for a node to discover its first parent in the network is at least 25 seconds.

The experiments in section 5.2 use the setup described above to examine application level performance of Surge. Section 5.3 examines the base SOS kernel and a simple TinyOS application to find the base CPU active time in the two operating systems when no application is present. The section continues with an examination of how the Surge application effects CPU active time. Section 5.4 finishes with a numerical examination of the update overhead in SOS, TinyOS and Maté. In an attempt to isolate operating system specific overhead, the analysis in section 5.4 does not use Deluge or the module distribution protocol currently used in SOS.

SOS reuses some of the TinyOS driver code for its Mica2 target and AVR support. Table 7 shows the approximate percentage of code reused from TinyOS in SOS. For these experiments SOS uses nearly direct ports of the TinyOS CC1000 radio stack, sensor drivers, and applications. This helps to isolate performance differences to within the kernel and module framework of SOS.

5.2 Macro Benchmarks

We first measure the time needed to form a routing tree and then measure the number of packets delivered from each node to the base station in a duration of 40 minutes. These tests act as a macro benchmark to help verify that Surge is running correctly on the three test platforms. Since all three systems are executing the same application and the CPU is not being heavily loaded, we ex-

pect differences in application performance to be small enough to be lost in the noise, as in fact they are.

For this experiment, Mica2 motes are deployed in a linear topology, regularly spaced at a distance of 4 feet from each other. The transmit power of the radio is set to -25dBm, the lowest possible transmit power, which results in a range of approximately 4 feet for our indoor laboratory environment.

Figure 5 shows the average time it took for a node to discover its first parent, averaged over 5 experiments. All three systems achieve latencies quite close to 25 seconds, the minimum latency possible given the Surge configuration parameters. The differences between the three systems are mainly due to their different boot routines. The source of the jitter results from the computation of the parent being done in a context which is often interrupted by packet reception.

Figure 5 also shows the average time it takes to deliver the first packet from a node to the base station. Since all three systems implement a queue for sending packets to the radio, the sources of per-hop latency are queue service time, wireless medium access delay, and packet transmission time. Queue service time depends upon the amount of traffic at the node, but the only variation between the three systems is due to the differences in the protocols for code dissemination given the otherwise lightly loaded network. The medium access delay depends on the MAC protocol, but this is nearly identical in the three systems [18]. Finally, packet transmission time depends upon the packet size being transmitted; this is different by a handful of bytes in the three systems, causing a propagation delay on the order of 62.4 μs per bit [21]. The latency of packet delivery at every hop is almost identical in the three systems. The small variations that can be observed are introduced mainly due to the randomness of the channel access delay.

Finally, figure 5 shows the packet delivery ratio for the three systems when the network was deployed for 40 minutes. As expected, the application level performance of the three systems was nearly identical. The slight differences are due to the fluctuating wireless link quality and the different overheads introduced by the code update protocols.

These results verify our hypothesis that the application level performance of a typical sensor network application running at a low duty cycle in SOS is comparable to other systems such as TinyOS and Maté. The overheads introduced in SOS for supporting run time dynamic linking and message passing do not affect application performance.

5.3 CPU Overhead

We proceed by measuring CPU active time on the SOS and TinyOS operating systems without any application
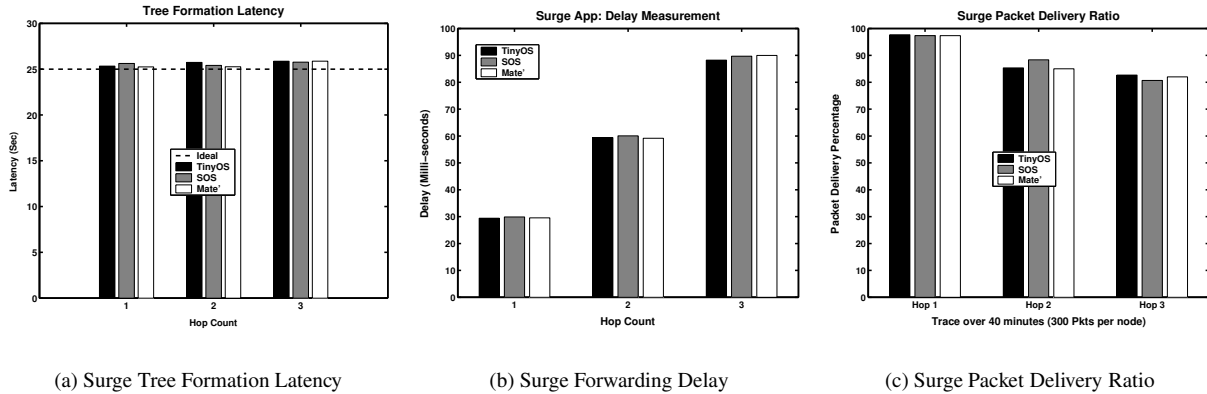
| (a) Surge Tree Formation Latency | (b) Surge Forwarding Delay | (c) Surge Packet Delivery Ratio |

Figure 5—Macro Benchmark Comparison of Surge Application

activity to examine the base overhead in the two systems. Bombilla is not examined in these base measurements since the virtual machine is specific to the Surge application. The evaluation is finished by examining the CPU active time when running Surge on each of SOS, TinyOS, and Maté. In all three systems, the source code is instrumented to raise a GPIO pin in the microcontroller when the CPU is performing any active operation (executing a task or interrupt). A high speed data logger is used to capture the waveform generated by the GPIO pin and measure the active duration. All experiments are run for 10 intervals of 60 seconds using the same Mica2 motes. We expect both systems to have nearly identical base overheads since the SOS kernel should be nearly as efficient as the base TinyOS kernel.

To examine the base overhead in SOS we installed a blank SOS kernel onto a single mote and measured CPU active time to be $7.40\% \pm 0.02\%$. In contrast to this, running the closest TinyOS equivalent, TOSBase, resulted in a CPU active time of $4.76\% \pm 0.01\%$. This active time discrepancy came as a great surprise and prompted an evaluation into the cause of overhead in SOS.

Detailed profiling using the Avrora [23] simulator revealed that the overhead in SOS may be related to the SPI interrupt used to monitor the radio for incoming data. The SPI interrupt is triggered every $418\mu s$ in both SOS and TinyOS when the radio is not in a low power mode, as is the case in the above experiments. Micro-benchmarking showed that the SPI interrupt handler in SOS and TinyOS are almost identical. However upon exit from the SPI interrupt, control is transferred to the scheduler that checks for pending tasks. On a lightly loaded system the common case is to perform this check and go to sleep, since the scheduling queue is usually empty. Further micro-benchmarking revealed that in the common case of empty scheduling queues, the SOS scheduling loop takes 76 cycles compared to 25 cycles in TinyOS. This additional overhead acquired every $418\mu s$

accounts for the discrepancy in performance between SOS and TinyOS.

The SOS scheduling loop was then optimized to better handle this common case reducing the CPU active time to $4.52\% \pm 0.02\%$ and resulted in SOS outperforming TinyOS. For a fair comparison we modified the TinyOS scheduling loop and *post* operation [3] to use a similar scheduling mechanism, reducing the TinyOS CPU active time to $4.20\% \pm 0.02\%$. This difference is small enough that it can be accounted for by two additional push and pop instruction pairs in the SOS SPI handler that are introduced due to the slight modifications to the radio stack implementation.

The above active times are summarized in table 8. Of the most interest are the last two entries, which show that when both scheduling loops are optimized the base operation overhead of SOS and TinyOS are within a few percent, as expected.

Having verified that the base operating systems have very similar overheads, we continue by examining the CPU active time when Surge is run on SOS, TinyOS and Maté. Versions of both SOS and TinyOS with optimized schedulers are used for all three tests and Maté Bombilla runs on top of the optimized TinyOS core. Surge is loaded onto two nodes, and the active time is again measured by instrumenting the code to activate a GPIO pin when active and monitoring this pin with a high speed data logger. The experiment results are shown in Table 9. This comparison shows that this low duty cycle application has less effect on the base performance of SOS than for TinyOS. The cause of this larger increase for TinyOS is not yet fully understood. One hypothesis is that the initial baseline measurement between SOS and TinyOS do not accurately reflect the same base functionality.

5.4 Code Updates

We now present an evaluation of the energy needed for propagation and installation of the Surge application. We

OS Version	Percent Active Time
SOS Unoptimized	$7.40\% \pm 0.02\%$
TinyOS Unoptimized	$4.76\% \pm 0.01\%$
SOS Optimized	$4.52\% \pm 0.02\%$
TinyOS Optimized	$4.20\% \pm 0.02\%$

Table 8—CPU Active Time on Base Operating System

OS Version	Percent Active Time
SOS Optimized	$4.64\% \pm 0.08\%$
TinyOS Optimized	$4.58 \pm 0.02\%$
Maté Bom.	$5.13 \pm 0.02\%$

Table 9—CPU Active Time With Surge

SOS Module Name	Code Size
Sample_Send	568 bytes
Tree_Routing	2242 bytes
Photo_Sensor	372 bytes
Energy (mJ)	2312.68
Latency (s)	46.6

Table 10—SOS Surge Remote Installation

begin by looking at installing Surge onto a completely blank node over a single hop in SOS. Surge on SOS consists of three modules: Sample_send, Tree_routing, and Photo_Sensor. Table 10 shows the size of the binary modules and overall energy consumption and the latency of the propagation and installation process. The energy consumption was measured by instrumenting the power supply to the Mica2 to sample the current consumed by the node.

We continue with an analytical analysis of the energy costs of performing similar code updates in our three benchmark systems—SOS, TinyOS with Deluge [10], and Maté Bombilla VM with Trickle [16]. It is important to separate operating system-related energy and latency effects from those of the code distribution protocol. Overall latency and energy consumption is mainly due to the time and energy spent in transmitting the updated code and storing and writing the received code to the program memory.

Communication energy depends upon the number of packets that need to be transferred in order to propagate the program image into the network; this number, in turn, is closely tied to the dissemination protocol. However, the number of application update bytes required to be transferred to update a node depends only on the architecture of the operating system. Therefore, to eliminate the differences introduced due to the different dissemination protocols used by SOS, TinyOS, and Bombilla, we consider the energy and latency of communication and the update process to be directly proportional to the number of application update bytes that need to be transferred and written to the program memory. The design of the actual distribution protocol used is orthogonal to the operating system design and SOS could use any of the existing code propagation approaches [22, 16]; it currently uses a custom publish/subscribe protocol similar to MOAP [22].

5.4.1 *Updating low level functionality*

To test updating low level functionality, we numerically evaluate the overhead of installing a new magnetometer sensor driver into a network of motes running the Surge application. The SOS operating system requires the distribution of a new magnetometer module, whose binary is 1316 bytes. The module is written into the program

System	Code Size (Bytes)	Write Cost (mJ/page)	Write Energy (mJ)
SOS	1316	0.31	1.86
TinyOS	30988	1.34	164.02
Maté VM	N/A	N/A	N/A

Table 11—Magnetometer Driver Update

System	Code Size (Bytes)	Write Cost (mJ/page)	Write Energy (mJ)
SOS	566	0.31	0.93
TinyOS	31006	1.34	164.02
Maté VM	17	0	0

Table 12—Surge Application Update

memory, one 256-byte page of the flash memory at a time. The cost of writing a page to the flash memory was measured to be 0.31 mJ, and 6 pages need to be written so the total energy consumption of writing the magnetometer driver module is 1.86 mJ.

TinyOS with the Deluge distribution protocol requires the transfer of a new Surge application image with the new magnetometer driver. The total size of the Surge application with the new magnetometer driver is 30988 bytes. The new application image is first stored in external flash memory until it is completely received. The cost of writing and reading one page of the external flash is 1.03 mJ [21]. Thereafter, the new image is copied from the external flash to the internal flash memory. This makes the total cost of installing a page of code into the program memory 1.34 mJ, and the total energy consumption for writing the updated Surge application 164.02 mJ.

Lastly, the Maté Bombilla VM does not support any mechanism for updating low level functionality such as the magnetometer driver. These results are summarized in table 11.

5.4.2 *Updating application functionality*

We examine updates to applications by numerically evaluating the overhead of updating the functionality of the Surge application. The modified Surge application samples periodically, but transmits the value to a base station only if it exceeds a threshold. The SOS operating system requires the distribution of a new Sample_Send module with the updated functionality. As before, TinyOS/Deluge requires the transfer of a new Surge application image with the modified functionality. However, the Maté Bombilla VM requires just a bytecode up-

date. The total size of the bytecode was only 17 bytes, and since it is installed in the SRAM, it has no extra cost over the running cost of the CPU. The results of an analysis similar to that in section 5.4.1 is presented in table 12.

SOS offers more flexibility than Maté; operations such as driver updates not already encoded in Maté's high-level bytecode cannot be accomplished. However, Maté's code updates are an order of magnitude less expensive than SOS's. TinyOS with Deluge picks the extreme end of the flexibility/cost trade off curve by offering the greatest update flexibility at the highest cost of code update. With a Deluge like mechanism, it is possible to upgrade the core kernel components, which is not possible using only modules in SOS; but in SOS and Maté, the state in the node is preserved after code update while it is lost in TinyOS due to a complete reboot of the system.

The above analysis does not consider how differential updates described in section 2.3 could impact static and dynamic systems. There is potential for differential updates to significantly decrease the amount of data that needs to be transmitted to a node for both systems. It is also unclear how efficiently a differential update, which may require reconstructing the system image in external flash, can be implemented. Differential updates are an area of research that both SOS and TinyOS could benefit from in the future.

5.4.3 Analytical analysis of update energy usage

The previous evaluation of CPU utilization and update costs in SOS, TinyOS, and Maté prompts asking how significant those factors are in the total mote energy expenditure over time. Application energy usage on a mote can be divided up into two primary sources: the energy used to install or update the application, and the power used during application execution multiplied by application execution time. An interesting point of comparison is to find the time, if any, when the total energy usage of two systems running the same application becomes equal. Starting with an update to a node, the total energy consumption of the node is found using:

$$E_{total} = E_{update} + P_{average} \times T_{live} \qquad (1)$$

$P_{average}$ denotes the power consumption of the mote averaged over application execution duration and the sleep duration (due to duty cycling). T_{live} is the time that the node has been alive since it was updated. E_{update} is the energy used to update or install the application.

The average power consumption during application execution depends upon the active time of the application, duty cycle of operation, and the power consumption of the hardware platform in various power modes. The Mica2 power consumption in various modes was obtained from [21] and is summarized in table 13. We first

Mode	Active (mW)	Idle (mW)	Sleep (mW)
Mica2 Power	47.1	29.1	0.31

Duty Cycle(%)	TinyOS (mW)	SOS (mW)	Maté (mW)
100	29.92	29.94	30.02
10	3.271	3.272	3.281
1	0.6057	0.6058	0.6067

Table 13—Average Power Consumption of Mica2 Executing Surge

compute P_{awake}, the average power consumption of the Surge application assuming a 100% duty cycle i.e. the system is operating continuously without sleep:

$$P_{awake} = P_{active} \times \frac{T_{active}}{T_{active} + T_{idle}} + P_{idle} \times \frac{T_{idle}}{T_{active} + T_{idle}}$$

Table 9 provides the ratio $\frac{T_{active}}{T_{active}+T_{idle}}$, which can be used to find $\frac{T_{idle}}{T_{active}+T_{idle}}$. When the system is operated at lower duty cycles, the average power consumption is given by:

$$P_{average} = P_{awake} \times DutyCycle + P_{sleep} \times (1 - DutyCycle)$$

Table 13 summarizes the average power consumption of the Surge application at various duty cycles.

Now we compute the energy used during the Surge application installation, E_{update}, which is the sum of the energy consumed in receiving the application over the radio and subsequently storing it on the platform. For the Mica2 Mote, the energy to receive a byte is 0.02 mJ/byte-[21]. Using table 12 we can compute the energy required to update the Surge application on SOS assuming that communications energy has a one to one correlation with application bytes transmitted:

$$E_{update}(SOS) = 0.02\frac{mJ}{byte} \times 566bytes + 0.93mJ = 12.25mJ$$

Similar computations can also be performed for TinyOS and Maté using the numbers in table 12.

Using the numbers computed thus far, it is possible to understand the total energy consumption of a system, E_{total}, as a function of the elapsed time T_{live}. From equation 1, it can be seen that energy consumption is linear with the slope being equal to $P_{average}$ with an initial offset equal to E_{update}. The average power consumption for each of the three systems examined is very similar and results in nearly identical total energy usage over time, despite the initially significantly different update costs. For example, by solving the linear equations for the 10% duty cycle, it is easy to see that TinyOS becomes more energy efficient than SOS after about 9 days. Similarly, for a 10% duty cycle, SOS becomes more energy efficient than Maté after about 22 minutes. But looking at the absolute difference in energy consumed after 30 days at a 10% duty cylcle, SOS has consumes less than 2 J more energy than TinyOS and Maté is only about 23 J beyond SOS. This analysis reveals that, contrary to our

initial intuition, differences in both CPU utilization and update costs are not significant in the total energy consumption of the node over time.

Note that this analysis is only for a single application, Surge, running under a given set of conditions. Different applications and evolving hardware technology will change the results of the above analysis. Applications with very high CPU utilization can magnify the effects of different CPU utilization on $P_{average}$ resulting in total energy variances of 5% to 10% on current mote hardware. Similarly, improvements to duty cycling that reduce node idle time can magnify the effects of CPU utilization on $P_{average}$. Hardware innovations that reduce peripheral energy usage, especially when the CPU is idle, will also lead to a noticeable discrepancy in total energy consumption. Only as the above improvements to systems are made, leading to more energy efficient execution environments, will energy savings from efficient updates become significant.

5.5   Discussion

This analysis examines energy usage, which provides a quantifiable comparison of these systems. This section also shows that these differences are not significant when taking into account average power consumption resulting from the different power modes of current mote technology. Choosing between operating systems for these systems should be driven more by the features provided by the underlying system, such as flexibility of online updates in SOS or full system static analysis in TinyOS, rather than total system energy consumption. As software and hardware technologies advance, the effects of differing CPU utilization and update costs between systems will play a more significant role in choosing a system.

The results of the experiments performed in this section validate the following claims about SOS. First, the architectural features presented in section 3 position SOS at a middle ground between TinyOS and Maté in terms of flexibility. Second, the application level performance of the SOS kernel is comparable to TinyOS and Maté for common sensor network applications that do not stress the CPU. Third, code updates in SOS consume less energy than similar updates in TinyOS, and more energy than similar updates in Maté. Fourth, despite different update costs and CPU active times, the total energy usage in SOS is nearly identical to that in TinyOS and Maté.

6   CONCLUSION

SOS is motivated by the value of maintaining modularity through application development and into system deployment, and of creating higher-level kernel interfaces that support general-purpose OS semantics. The architecture of SOS reflects these motivations. The kernel's message passing mechanism and support for dynamically allocated memory make it possible for independently-created binary modules to interact. To improve the performance of the system and to provide a simple programming interface, SOS also lets modules interact through function calls. The dynamic nature of SOS limits static safety analysis, so SOS provides mechanisms for run-time type checking of function calls to preserve system integrity. Moreover, the SOS kernel performs primitive garbage collection of dynamic memory to improve robustness. Fine grain energy usage of SOS, TinyOS and the Maté Bombilla virtual machine are dependent on the frequency of updates and stressing of the CPU required for a specific deployment. However, analysis of overall energy consumed by each of the three systems at various duty cycles shows nearly identical energy usage for extended deployments. Thus, to choose between SOS and another system, it is important for developers to consider the benefits of static or dynamic deployments independently of energy usage.

SOS is a young project under active development and several research challenges remain. One critical challenge confronting SOS is to develop techniques that protect the system against incorrect module operation and help facilitate system consistency. The stub function system, typed entry points, and memory tracking protect against common bugs, but in the absence of any form of memory protection it remains possible for a module to corrupt data structures of the kernel and other modules. Techniques to provide better module isolation in memory and identify modules that damage the kernel are currently being explored. The operation of SOS is based on the notion of co-operative scheduling, but an incorrectly composed module can utilize all of the CPU. We are exploring more effective watchdog mechanisms to diagnose this behavior and take corrective actions. Since binary modules are distributed by resource constrained nodes over a wireless ad-hoc network, SOS motivates more research in energy efficient protocols for binary code dissemination. Finally, SOS stands to benefit greatly from new techniques and optimizations that specialize in improving system performance and resource utilization for modular systems.

By combining a kernel that provides commonly used base services with loosely-coupled dynamic modules that interact to form applications, SOS provides a more general purpose sensor network operating system and supports efficient changes to the software on deployed nodes.

We use the SOS operating system in our research and in the classroom, and support users at other sites. The system is freely downloadable; code and documentation are available at:

http://nesl.ee.ucla.edu/projects/sos/.

REFERENCES

[1] ABRACH, H., BHATTI, S., CARLSON, J., DAI, H., ROSE, J., SHETH, A., SHUCKER, B., DENG, J., AND HAN, R. Mantis: system support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications* (2003), ACM Press, pp. 50–59.

[2] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles* (Copper Mountain, Colorado, 1995), pp. 267–284.

[3] BOULIS, A., HAN, C.-C., AND SRIVASTAVA, M. B. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services* (2003), ACM Press, pp. 187–200.

[4] CROSSBOW TECHNOLOGY, INC. *Mote In-Network Programming User Reference*, 2003.

[5] DUNKELS, A., GRÖNVALL, B., AND VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors* (2004).

[6] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles* (1995), pp. 251–266.

[7] FOK, C., ROMAN, G.-C., AND LU, C. Rapid development and flexible deployment of adaptive wireless sensor network applications. Tech. Rep. WUCSE-04-59, Washington University, Department of Computer Science and Engineering, St. Louis, 2004.

[8] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation* (2003).

[9] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (2000), ACM Press, pp. 93–104.

[10] HUI, J. W., AND CULLER, D. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the second international conference on Embedded networked sensor systems* (2004), ACM Press.

[11] JEONG, J., AND CULLER, D. Incremental network programming for wireless sensors. In *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON* (2004).

[12] JUANG, P., OKI, H., WANG, Y., MARTONOSI, M., PEH, L.-S., AND RUBENSTEIN, D. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet. In *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)* (San Jose, California, Oct. 2002).

[13] KAISER, W., POTTIE, G., SRIVASTAVA, M., SUKHATME, G. S., VILLASENOR, J., AND ESTRIN, D. Networked Infomechanical Systems (NIMS) for ambient intelligence.

[14] LEVIS, P., AND CULLER, D. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA* (Oct. 2002).

[15] LEVIS, P., MADDEN, S., GAY, D., POLASTRE, J., SZEWCZYK, R., WOO, A., BREWER, E., AND CULLER, D. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of the First Symposium on Networked Systems Design and Implementation* (2004), USENIX Association, pp. 1–14.

[16] LEVIS, P., PATEL, N., CULLER, D., AND SHENKER, S. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the First Symposium on Networked Systems Design and Implementation* (2004), USENIX Association, pp. 15–28.

[17] LIU, T., AND MARTONOSI, M. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming* (2003), ACM Press, pp. 107–118.

[18] POLASTRE, J., HILL, J., AND CULLER, D. Versatile low power media access for wireless sensor networks. In *Second ACM Conference on Embedded Networked Sensor Systems* (2004).

[19] RASHID, R., JULIN, D., ORR, D., SANZI, R., BARON, R., FORIN, A., GOLUB, D., AND JONES, M. B. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON* (San Francisco, CA, USA, 1989), IEEE Comput. Soc. Press, pp. 176–178.

[20] REIJERS, N., AND LANGENDOEN, K. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications* (2003), ACM Press, pp. 60–67.

[21] SHNAYDER, V., HEMPSTEAD, M., RONG CHEN, B., AND MATT WELSH, H. Powertossim: Efficient power simulation for tinyos applications. In *Sensor Networks. In Proc. of ACM SenSys 2003.* (2003).

[22] STATHOPOULOS, T., HEIDEMANN, J., AND ESTRIN, D. A remote code update mechanism for wireless sensor networks. Tech. Rep. CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[23] TITZER, B. L., PALSBERG, J., AND LEE, D. K. Avrora: Scalable sensor network simulation with precise timing. In *Fourth International Conference on Information Processing in Sensor Networks* (2005).

NOTES

[1] Soft reboot of many microcontrollers, including the Atmel AVR, preserves on chip memory.

[2] This source code is trimmed for clarity. Complete source code can be downloaded separately.

[3] Patch is being submitted to the TinyOS developers should they wish to optimize for the case of a lightly loaded radio that is not in a low power mode.