A Dynamic Operating System for Sensor Nodes [*]

Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler and Mani Srivastava
{simonhan@cs.,ram@,roy@cs.,kohler@cs.,mbs@}ucla.edu
University of California, Los Angeles

Abstract

Sensor network nodes exhibit characteristics of both embedded systems and general-purpose systems. As an embedded system, a sensor node must use little energy and be robust to environmental conditions. As a general-purpose system, a node should provide common services that make it easy to write applications. TinyOS, the current state of the art in sensor network operating systems, focuses on traditional embedded system constraints; reusable components implement common services, but a 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 implements 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 for TinyOS. Our evaluation shows that despite the dynamic nature of SOS and its higher-level kernel interface, it performs comparably to TinyOS in terms of energy usage and performance, and better in terms of energy usage during software updates.

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 any networked infrastructure and easy human accessibility, anywhere from the forest canopy [12] to the backs of zebras [11]. Due to the difficulty and expense of maintaining such distant nodes, wireless sensor networks are expected to be both autonomous and long-lived, supporting application changes and 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, and there's a tension between the layers of indirection required to support true general-purpose operating systems and the resource and energy constraints characteristic of sensor nodes. TinyOS [14], the current state of the art in sensor operating systems, effectively prioritizes embedded system constraints over general-purpose OS functionality. TinyOS consists of a rich collection of software components written in the NesC language [7]; these components range 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 [9].

This paper shows that sensor network operating systems can achieve significantly more dynamic, general-purpose OS semantics than TinyOS without sacrificing energy efficiency or performance. Our operating system, SOS, consists of a common kernel and dynamically-loadable application modules; these modules, like conventional applications, can be loaded and 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 typed using compressed strings; this lets the system detect typing errors, such as when a module expecting a new version of an interface is loaded on a node that only provides the old, that would otherwise crash the system. SOS uses dynamic memory both in the kernel and in application modules, easing programing complexity and

[*] This paper is based on research funded in part from ONR through the AINS program and from the Center for Embedded Network Sensing.

increasing temporal memory reuse relative to TinyOS's conservative buffer allocation strategy. 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 application-level benchmarks of Surge, a well-known multi-hop data acquisition program. Comparisons of Surge versions running on SOS, TinyOS, and Maté Bombilla virtual machine [13] 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 systems shows that SOS occupies a nice middle ground: it provides significant energy savings over TinyOS, and more expressivity than Maté Bombilla.

The rest of the paper is structured as follows. Section 2 provides a foundation in general operating systems, sensor network evolution, and dynamic code techniques that have influenced the SOS design. A more detailed look at the architecture used in SOS is presented in Section 3, including key differences between SOS and TinyOS. A brief walkthrough of an SOS application is presented in Section 4. An in-depth evaluation of SOS, including comparisons to TinyOS and Maté/Bombilla, each running a simple network sampling and data transport protocol, 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

The combination of a core kernel and modules used in SOS resembles some earlier operating systems that explored microkernel abstractions. Core kernel functionality can be viewed as basic hardware abstractions while modules provide both higher level system management services and applications.

Two examples of this are the Mach [19] operating system and the Exokernel [5]. 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, but the level of hardware abstraction exported by SOS is lower than that in Mach and higher than that in the Exokernel.

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 [8] is the de facto standard operating system for sensor nodes. TinyOS is written using the NesC language [7] 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. Unfortunately, static binding leads to an inflexible system after deployment. 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.

The need for flexible systems has inspired virtual machines (VM), mobile agents, and interpreters, for sensor nodes. One such system is Maté [13]. 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 [6] 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 [16] implements dynamic insertion of code on devices similar to PDAs, but not the simpler devices targeted by 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 [10]. 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] is a protocol that has been developed to distribute new node images within a sensor network; it uses XNP to reflash nodes. Deluge [9] is a reliable distribution protocol that is used to disseminate large amounts of data, such as TinyOS images, to all other nodes in a 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 natively 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 above the kernel, in SOS modules.

Figure 1 is an overview of the SOS architecture.

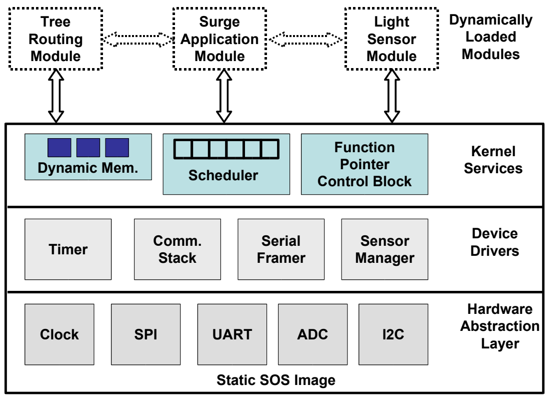


Figure 1—SOS Functional Layout

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. 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 in light of the commonly used TinyOS.

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. These footprints are images compiled for Mica2 Motes using Atmel AVR microcontrollers with 128 KB ROM and 4 KB RAM.

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 are only required when low layer hardware or resource management capabilities must be changed in the system. An application in SOS is composed of one or more modules interacting via asynchronous messages or function calls. Position independent binaries and clean interfaces to self contained modules through messaging and function calls creates a system that maintains its modularity through development and into deployment. The primary challenge in developing SOS was maintaining modularity and safety without incurring from high overhead due to the loose coupling of modules.

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 2.

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

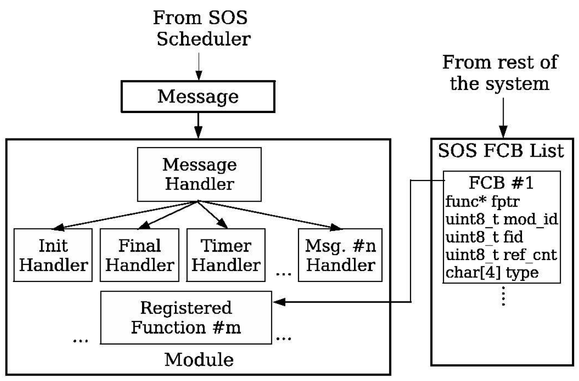


Figure 2—Module Interactions

tion 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 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. Messages are thus delivered asynchronously. 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. This structure helps to maintain modularity in the deployed system since the message handler function for any module can always be located at a consistent offset in the binary, the location of inter-module functions becomes exposed to the system through the registration process, and program state is managed by the SOS kernel, freeing modules of dependencies on where state is stored.

Well defined interfaces to modules make it easy for independently written modules to interact. Figure 3 illustrates a memory debugging module being added to a running node to help locate a suspected memory bug on the running system.

3.1.2 *Module Interaction*

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

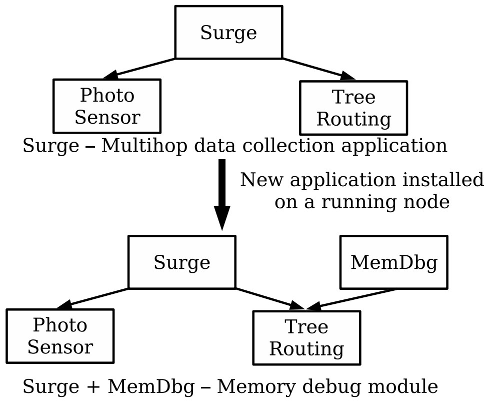

Surge
Photo Sensor
Tree Routing
Surge – Multihop data collection application
New application installed on a running node
Surge
MemDbg
Photo Sensor
Tree Routing
Surge + MemDbg – Memory debug module

Figure 3—Clean interfaces between modules help to allow modular layout on a node to evolve over time.

Communication Method	Clock Cycles
Messaging referencing internal data	149
Messaging referencing external data	305
Subscribed Call to Registered Function	13
Call Using System Jump Table	13
Direct Function Call	5

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

Function registration and subscription is the mechanism that SOS uses to provide direct inter-module communication. When explicitly registering functions with the SOS kernel, the 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. A function control block (FCB) created by the SOS kernel and indexed by the tuple (module ID, function identifier) is used to store key information about the registered function.

The FCB stores prototype information about the function in addition to a provider flag and a subscriber reference count. The stored prototype information also encodes whether a parameter contains dynamic memory that needs to undergo a change of ownership and is used by stub functions that maintain resource semantics of a function call and set a global error variable when a registered function is removed. This is discussed in more detail in section 3.1.4. 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').

The process of subscribing to a function uses the sys-

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

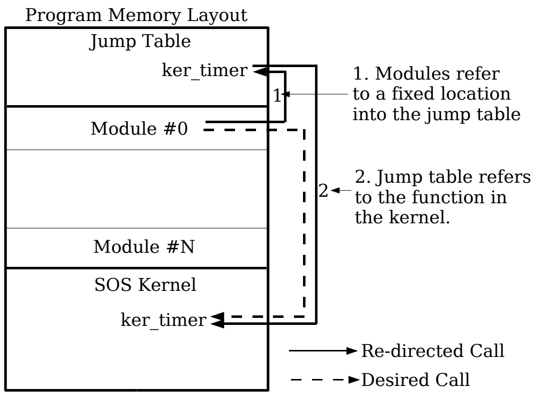

Program Memory Layout
Jump Table
ker_timer
1
1. Modules refer to a fixed location into the jump table
Module #0
2
2. Jump table refers to the function in the kernel.
Module #N
SOS Kernel
ker_timer
Re-directed Call
Desired Call

Figure 4—Jump table layout and linking in SOS.

tem call *ker_get_handle* described in table 3. 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.

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 4 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 callback module listening for advertisements of new modules in the network. When the callback 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 callback 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 and 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 simple linker script for all modules forces the message handler to use a known offset in the binary at compile time allowing easy linking from the SOS kernel. The registration process stores the absolute address of the handler, pointer to the dynamic memory holding the module state, and the identity of the module in a kernel data structure indexed by the unique module ID included in the metadata. Finally the SOS kernel invokes the handler of the module by scheduling an *init* message for the module.

The nature of advertisements and the distribution protocol used to 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 by dispatching a *final* message. This message provides the module a final opportunity to gracefully release any resources it is holding and inform dependent applications or protocols 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. FCBs with nonzero subscriber counts are linked to stub functions. Thus, other modules that depend on the removed module will not crash when they access a missing function; instead, the stub is called, releasing memory in function arguments as appropriate, and returns an error to the caller. Those with zero subscriber counts are removed.

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. It is important to provide a system that remains stable in the presence of problems. SOS provides mechanisms to minimize the impact of failure modes so that nodes do not crash due to dynamic system changes.

SOS also provides 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 collects any dynamically allocated memory in the message data. 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 messages when this occurs and exit the handler with an error code. The SOS kernel will then collect any dynamically allocated memory in the message data.

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, the implementation of a function is removed, the implementation of a function changes, or multiple implementations of a single function exist.

A subscription request is successful if there exists a FCB that has the same module ID, function ID and prototype as that in the subscription, and that has not been flagged as having no implementation. If these conditions are not met at the time of module subscription the attempt fails. Modules are free to handle this failure as they wish. Actions currently taken by SOS modules include aborting module insertion or scheduling a later attempt to subscribe to the function.

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 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. To change an implementation of a function the module providing the function first deregisters the function, then the module itself is updated, and finally the new implementation of the module can be registered. When the new provider is installed, the new provider registers the new version of the function. 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 and redirecting old subscribers 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 when the invalid function flag is checked on the old FCB.

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 at the cost of only supporting FIFO scheduling and the inability to pass parameters to functions. SOS instead implements priority queues, which can provide responsive servicing of interrupts without operating in an interrupt context. 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 4.

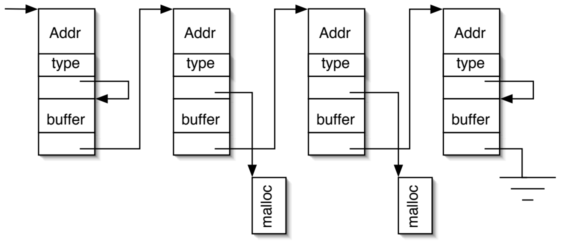


Figure 5—Memory Layout of An SOS Message Queue

Figure 5 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 di-

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 4—Messaging API

rectly 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 original mbuf design). SOS message headers also include a series of flags to describe the priority of the message, identify incoming and outgoing radio or UART 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 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. The priority queues have allowed SOS to minimize processing in interrupt contexts by writing interrupt handlers that quickly construct and schedule a 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 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. An example of this is TinyOS, which statically allocates memory for easier compile time analysis. Unfortunately, static memory allocation results in fixed length queues sized for worst case scenarios and an awkward programing environment where some common tasks are difficult to implement, such as passing a data buffer down a stack of protocol modules. 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 memory blocks provides constant time memory allocation and deallocation, reducing the overhead of using dynamic memory. Failed memory allocation returns a null pointer.

Prototype	Description	Cycles
void *ker_malloc(uint16_t size, sos_pid_t id)	Allocate memory	62
void ker_free(void* ptr)	Free memory	63
int8_t ker_change_own(void *ptr, sos_pid_t id)	Change ownership	46
sos_pid_t ker_check_memory()	Validate memory (after a crashes)	1842

Table 5—Cycles needed for common dynamic memory tasks in SOS

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 5. 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 those 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.

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

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 and UART libraries.

The loosely coupled design used in SOS has resulted in a platform that is very portable. SOS currently has support for the full line of Mica 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

```
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 6—Surge Source Code

Figure 6 contains a source code listing of the Sample_Send module of the Surge application[2]. The program is 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.

Line 2 shows the conversion of the generic state stored and passed in from the SOS kernel into the modules internal representation. As noted in section 3.1.1, the SOS kernel always stores a module's state to allow modules to be 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 used 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 performs 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 is far less expensive in terms of energy than the comparable solution in TinyOS with Deluge. This section tests these hypotheses using Surge, a prototypical multihop tree builder and data collector.

Initial macrobenchmarks show 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. However, the active CPU usage, and hence the energy consumption, of a Surge-like application on SOS is greater than Surge on TinyOS but less than Bombilla. Furthermore, the process of remotely updating an application's functionality in SOS is about 200 times more energy efficient than in TinyOS. We argue that this gain in update energy offsets the execution energy overhead of SOS in many situations. Along the way examples show that SOS provides more flexibility in updating application's functionality than Maté.

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

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. All the 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 with the neighborhood information.

The Surge application in SOS is implemented with three modules: Sample_Send, Tree_Routing and PhotoSensor. A three-hop network of Mica2 motes with the blank SOS kernel was deployed. The Sample_Send, Tree_Routing and PhotoSensor 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 Deluge 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 through the routing protocol. Bytecode for periodically sampling and sending the data to the base station was injected into the network and installed on all relevant machines.

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

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.

5.2 Macro Benchmarks

We first measure the time it takes to complete application-level benchmarks by examining the time needed to form a routing tree and to send a single packet from a node to the base station. Next we measure the number of packets delivered from each node to the base station in a duration of 40 minutes. These tests help to verify that Surge is running correctly on the three test platforms. Since all three systems are executing the same application, we expect differences in application performance to be small enough to be lost in the noise, as in fact they are.

Figure 7 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 for this test given our 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 a packet reception.

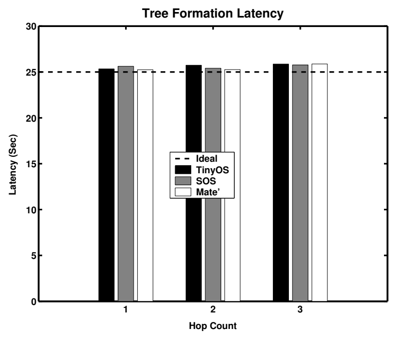
Figure 7 shows the average time it takes to deliver the first packet from a node to the base station. Since all the systems implement a queue for sending packets to the radio, the sources of per-hop latency are queue service time, wireless medium access delay, and packet transmission time. Queue service 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. The medium access delay depends on MAC protocol, but this is 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.

Figure 7 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.

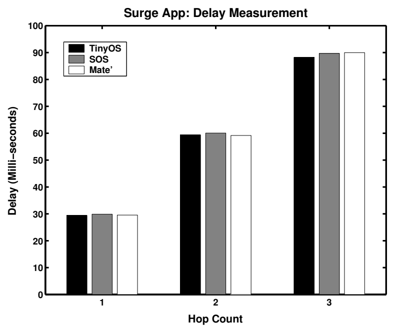
These results verify our hypothesis that the application level performance of a typical sensor network application executing on SOS is comparable to the other systems such as TinyOS and Maté. The overheads introduced in SOS for supporting run time dynamic linking and message passing do not affect application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

5.3 CPU Overhead

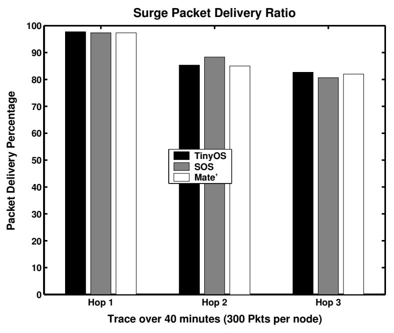
In this subsection, we measure CPU active time during a one-minute span of Surge's execution. This begins to address the question of how much energy SOS applications use. Sensor node energy usage can be divided among several causes, including radio usage (bytes sent and time spent in low-power listening mode, if available), CPU usage, flash memory/external flash access (writes can be particularly expensive), and sensor activity. SOS and TinyOS treat the radio and sensors essentially identi-



(a) Surge Tree Formation Latency



(b) Surge Forwarding Delay



(c) Surge Packet Delivery Ratio

Figure 7—Macrobenchmark Comparison of Surge Application

cally and Surge does not access flash memory or external flash. Therefore, the only differences in the energy consumption of the Surge application on the three systems is due to the differences in the CPU usage. In all three systems, the source code was instrumented to raise a GPIO pin in the microcontroller whenever the CPU was performing any active operation (executing a task or interrupt). A high speed data logger was used to capture the waveform generated by the GPIO pin and measure the active duration.

System	Active Time (In 1 min.)	Active Time (%)	Overhead Rel. TOS (%)
TinyOS	3.31 sec	5.52%	N/A
SOS	3.50 sec	5.84%	5.7%
Maté Bom.	3.68 sec	6.13%	11%

Table 6—Surge App: CPU Active Time

The experiment results are shown in Table 6. The CPU was active only 3.68 seconds in one minute for Maté Bombilla VM, and even less for SOS and TinyOS. Clearly the Surge application is not CPU intensive. Bombilla has an overhead of 11% over TinyOS for executing the Surge application. This is mainly due to two factors: First and the major factor of the overhead is the interpretation of the high level Maté bytecode in comparison to the native instructions. Second, Maté was executing the Trickle [15] protocol for bytecode updates while TinyOS was executing the Deluge [9] protocol for binary updates. Some variation in the active time is caused due to the differences in the amount of processing that needs to be done in the two different protocols. SOS has an overhead of 5.7% over TinyOS for executing the Surge application. This was an unexpectedly large overhead as SOS also executes instructions natively. We believe the overheads in SOS over TinyOS are primarily due to the following reasons.

1. Scheduling Overhead - SOS takes more cycles to insert and dispatch messages than TinyOS.
2. Number of messages dispatched - SOS exits an interrupt context by dispatching a high priority message which increases the number of messages dispatched in a system.
3. Dynamic Memory - Table 5 shows how each dynamic memory related operation consumes CPU cycles, is an overhead not in TinyOS
4. Indirection Overheads - Table 2 shows the overhead of indirection in function pointer pointer calls and jump table calls, which take slightly more CPU cycles than a direct function calls used in TinyOS.
5. Source Code Optimizations - SOS performs almost no source code optimizations compared to TinyOS, which uses the NesC compiler to analyze the whole program to inline function calls for improved performance.

SOS is still an early system with many new ideas being tested and we are confident that there is plenty of room for optimizing the above sources of overhead. It is interesting to note that energy costs of code updates to the deployed system can offset this cost, as we show in section 5.5.

Note that the CPU active time measurements were performed without any duty cycle of the application or the Mica2 radio stack. Therefore, some of the CPU active time was spent in the idle mode of the Mica2 radio stack in all the three systems. It can be argued that in the low power mode of operation, the CPU active time would be further reduced as the radio would be turned off when it is idle. The CPU in the idle mode of the Mica2 radio stack spends 1.6μs for every interrupt. The interrupts are received every 416μs in the Mica2 radio stack. Thus the theoretical upper limit on the active CPU time spent in the idle mode of the radio stack is about 231 ms. This is the upper limit because packet reception, packet transmission and synchronization mechanisms upon wakeup would take away time spent by the radio stack in the idle mode. By subtracting the upper limit from the active time of all the three systems, we obtain that the SOS overhead with respect to TinyOS to be about 6.11%. Similarly, the Maté Bombilla overhead with respect to TinyOS would be about 11.9%.

5.4 Code Updates

We now present an evaluation of the energy needed for propagation and installation of the Surge application. We begin by looking at installing Surge over a single hop in SOS. The Surge implementation consists of three modules: Sample_send, Tree_routing, and Photo_Sensor. Table 7 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 Mica node to sample the current consumed by the node. SOS's write cost per page incur flash overhead, while TinyOS's incurs flash and external flash overhead.

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

Table 7—SOS Surge Remote Installation

We continue with a comparison of the energy and latency costs of performing a similar code update in our three benchmark systems—SOS, TinyOS with Deluge [9], and Maté Bombilla VM with Trickle [15]. 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.

Both communication energy and latency depend 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. In comparison, the number of bytes of code required to be transferred to any 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 update process to be directly proportional to the number of bytes of the code that need to be transferred and written to the program memory. The design of the protocol is orthogonal to the operating system design and SOS can use any of the existing code propagation approaches [22, 15]; it currently uses a custom publish/subscribe protocol similar to MOAP [22].

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 8—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 9—Surge Application Update

5.4.1 *Updating low level functionality*

To test updating low level functionality, we install 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 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, so the total energy consumption of writing the magnetometer driver module is $1.86\text{ mJ} = 6 \cdot 0.31\text{mJ}$.

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. The results are summarized in table 8.

5.4.2 *Updating application functionality*

We test updates to applications by 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. The results of an analysis similar to the previous experiment are presented in Table 9. As before, TinyOS/Deluge requires requires the transfer of a new Surge application image with the modified functionality, expending almost 400 times as much energy as SOS. However, the Maté Bombilla VM requires just a bytecode update. 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.

SOS offers more flexibility than Maté; operations such as driver updates not already encoded in Maté's high-level bytecode cannot be implemented. However, Maté's code updates are an order of magnitude cheaper 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 even to upgrade the core kernel components, which is not currently possible with 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. These design choices must be considered by application designers to best choose an sensor network operating system for a particular deployment.

5.5 Discussion

From table 6, we see that application execution on SOS has an energy overhead of about 5.7%[3] over TinyOS. However, the cost of code update is higher for TinyOS in comparison to SOS as shown in Table 9. Clearly, there exists a minimum interval between successive code updates at which the energy consumption of the two systems is matched. If updates occur more frequently, then SOS has a lower energy consumption. We attempt to characterize this interval based upon our measurements.

We assume an ideal distribution scenario in which an external base station transmits the update such that each node receives the source update exactly once with no protocol overhead, no packet headers, and no data retransmission. The above assumptions were made to make the analysis protocol independent.

For an application that executes for T_{Active} seconds in TinyOS, the difference in the execution energy consumption for the two systems ΔE_{CPU}, is given by the equation:

$$\Delta E_{CPU} = P_{CPU} \cdot T_{Active} \cdot SOS_{overhead}$$

P_{CPU} describes the power consumed by the CPU and equals 24 mW [21] for AVR micro-controllers; $SOS_{overhead}$ was measured to be 0.057. The code update energy difference, ΔE_{Update}, is given by the equation:

$$\Delta E_{Update} = \Delta E_{Flash} + \Delta E_{Comm}$$

We assume ΔE_{Comm} to be directly proportional to the difference in the number of bytes received in the two systems. Therefore,

$$\Delta E_{Comm} = \Delta Bytes_{RX} \cdot E_{ByteRX} \cdot K_{Comm}$$

For Surge application, the values for ΔE_{Flash} and $\Delta Bytes_{RX}$ can be obtained from the Table 9. E_{ByteRX} equals 0.02 mJ/byte [21] for Mica2 sensor nodes. We initially set the constant of proportionality K_{Comm} to 1. The two systems will have an equal energy consumption when $\Delta E_{CPU} = \Delta E_{Update}$. This gives us an active time (T_{Active}) of 536 seconds. In the Surge application, the CPU is active for only 3.3 seconds in a minute. Therefore, 536 seconds of active CPU time will be expended in 162 minutes. Hence, if Surge were to operate continuously, then code updates would have to be done every 162 minutes for TinyOS and SOS to have identical energy consumption. However, most of the real sensor network deployments have a duty cycle of about 1% [17]. If we assume that the Surge application is active for one minute every hour (dropping the duty cycle to 1.67%) to sample and transmit data to the base station, then the 536 seconds of active CPU time will be expended in 162 hours or approximately one week. Therefore, if code updates occur more frequently than once in one week, SOS has a lesser overall energy consumption.

An important point to be noted about the above analysis is that the strong assumptions used to make the analysis protocol independent by using an ideal distribution protocol result in an absolute lower bound on SOS performance. Furthermore, ΔE_{Comm} was assumed to be directly proportional to the difference in the number of bytes to be received in the two systems ($K_{Comm} = 1$). Packet losses are a common occurrence in wireless channels and communication energy is a strong function of the nature of this loss (random, bursty, etc.) and the response of the individual protocols to such losses. Typical packet losses for any operating environment are hard to characterize, making the above analysis difficult. However, we can definitely claim that the packet losses will have a greater impact on energy consumption for TinyOS and Deluge in comparison to SOS, due to the increased number of bytes that need to be transported in the former. The overall impact of this would be to increase the interval between successive code updates in SOS before it breaks even with TinyOS.

A similar comparison between Maté Bombilla and SOS gives us a minimum update interval of 2.6 hours with a duty cycle of 1.67%. Therefore, if the code is updated less frequently than once in every 2.6 hours, it is more energy efficient to use SOS. Once again, the losses in the wireless medium will increase this interval.

This analysis examines energy usage, which provides a quantifiable comparison of these systems. Choosing an operating system for a deployment requires a broad examination of additional factors. Stable deployments that

[3]Using the theoretical upper limit of 6.11% described in section 5.3 does not alter the results.

must be long lived may better benefit from the use of the aggressive optimizations in TinyOS. Deployments that wish to concurrently run applications developed by independent researchers, explore the use of heterogeneous software deployments, or update functionality in deployed nodes will benefit from SOS. Finally, deployments that need flexibility within a narrow application class can benefit from Maté.

5.6 Summary

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 the flexibility and the cost associated with remote code updates. Second, the application performance of the SOS kernel is comparable to TinyOS and Maté for common sensor network applications that do not stress the CPU. Third, the SOS kernel has a small overhead with respect to TinyOS during application execution. Fourth, code updates in SOS consume less energy than similar updates in TinyOS.

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. Experimentation shows that the energy usage of SOS, TinyOS and the Maté Bombilla virtual machine is dependent on the frequency of updates required in a deployed system. SOS has a superior energy profile for deployments that need updates but do not need constant retasking. Thus, to choose between SOS and another system, it is important for developers to consider how dynamic the deployed system might be and to what extent they value the services provided by SOS and the modularity it enables.

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. 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 is 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 might end up utilizing all 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. In this way SOS sits at a dynamic point on the design spectrum of sensor node operating systems.

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] 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 conference on Wireless sensor networks and applications*, pages 50–59. ACM Press, 2003.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.

[3] A. Boulis, C.-C. Han, and M. B. Srivastava. 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*, pages 187–200. ACM Press, 2003.

[4] Crossbow Technology, Inc. *Mote In-Network Programming User Reference*, 2003.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for

application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[6] C. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. Technical Report WUCSE-04-59, Washington University, Department of Computer Science and Engineering, St. Louis, 2004.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation*, 2003.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104. ACM Press, 2000.

[9] J. W. Hui and D. Culler. 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*. ACM Press, 2004.

[10] J. Jeong and D. Culler. 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.

[11] P. Juang, H. Oki, Y. Wang, M. Martonosi, L.-S. Peh, and D. Rubenstein. 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.

[12] W. Kaiser, G. Pottie, M. Srivastava, G. S. Sukhatme, J. Villasenor, and D. Estrin. Networked Infomechanical Systems (NIMS) for ambient intelligence. 2004.

[13] P. Levis and D. Culler. 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. To appear.

[14] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, pages 1–14. USENIX Association, 2004.

[15] P. Levis, N. Patel, D. Culler, and S. Shenker. 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*, pages 15–28. USENIX Association, 2004.

[16] T. Liu and M. Martonosi. 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*, pages 107–118. ACM Press, 2003.

[17] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, Sept. 2002.

[18] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *To appear in the Second ACM Conference on Embedded Networked Sensor Systems*, 2004.

[19] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[20] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67. ACM Press, 2003.

[21] V. Shnayder, M. Hempstead, B. rong Chen, and H. Matt Welsh. Powertossim: Efficient power simulation for tinyos applications. In *Sensor Networks. In Proc. of ACM SenSys 2003.*, 2003.

[22] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.