Surviving Sensor Network Software Faults

Yang Chen
School of Computing
University of Utah
Salt Lake City, UT USA
chenyang@cs.utah.edu

Omprakash Gnawali
Computer Science Department
University of Southern California
Los Angeles, CA USA
gnawali@usc.edu

Maria Kazandjieva
Computer Systems Laboratory
Stanford University
Stanford, CA USA
mariakaz@stanford.edu

Philip Levis
Computer Systems Laboratory
Stanford University
Stanford, CA USA
pal@cs.stanford.edu

John Regehr
School of Computing
University of Utah
Salt Lake City, UT USA
regehr@cs.utah.edu

Abstract

We describe Neutron, a version of the TinyOS operating system that efficiently recovers from memory safety bugs. Where existing schemes reboot an entire node on an error, Neutron's compiler and runtime extensions divide programs into recovery units and reboot only the faulting unit. The TinyOS kernel itself is a recovery unit: a kernel safety violation appears to applications as the processor being unavailable for 10–20 milliseconds.

Neutron further minimizes safety violation cost by supporting "precious" state that persists across reboots. Application data, time synchronization state, and routing tables can all be declared as precious. Neutron's reboot sequence conservatively checks that precious state is not the source of a fault before preserving it. Together, recovery units and precious state allow Neutron to reduce a safety violation's cost to time synchronization by 94% and to a routing protocol by 99.5%. Neutron also protects applications from losing data. Neutron provides this recovery on the very limited resources of a tiny, low-power microcontroller.

1. Introduction

Sensor networks consist of large numbers of small, low-power, wireless devices, often embedded in remote and inconvenient locations such as volcanoes [35], thickets [25], bird burrows [31], glaciers [32], and tops of light poles [22]. Applications commonly specify that a network should operate unattended for months or years [25, 31]. Software dependability and reliability are therefore critical concerns.

In practice, however, sensor networks operate for weeks or months and require significant attention from developers or system administrators [31, 35]. The discrepancy between desired and actual availability is in part due to difficult-to-diagnose bugs that emerge only after deployment [34]. A recent deployment in the Swiss Alps illustrates this challenge. Network communication failed during mornings and evenings, but worked during the day and night. The cause was temperature response differences for the processor and radio oscillators. Periods of warming and cooling led their clocks to drift too much for their interconnect to be reliable [3].

Unforeseen bugs often manifest as memory errors. For example, a popular radio chip, the ChipCon CC2420, erroneously signals reception of corrupted packets shorter than the 802.15.4 standard permits. Early CC2420 drivers for the TinyOS operating system did not consider this case; receiving a short, corrupt packet triggered an off-by-one error in a loop, overwriting unrelated parts of RAM [28].

As wireless sensors use microcontrollers whose RAM is no larger than a typical MMU page, compiler-enforced safety is the standard mechanism for detecting memory bugs. For example, Safe TinyOS [8] uses Deputy [7] to make all of TinyOS and its applications type-safe, preventing pointer bugs from cascading into memory corruption and random consequences.

Safe execution is an important step towards dependable software, but it raises a difficult question: How should a node respond to a safety violation? These embedded, event driven systems typically have no concept of a process or unit of

SOSP'09 October 11–14, 2009, Big Sky, Montana, USA.

Reprinted from SOSP'09, Proceedings of the 22nd ACM Symposium on Operating Systems Principles, October 11–14, 2009, Big Sky, Montana, USA., pp. 1–16.

code isolation. Thus, on a safety violation, Safe TinyOS spits out an error message (for lab testing) or reboots the entire node (in deployment).

Rebooting an entire node is costly: it wastes energy and loses data. Systems gather state such as routing tables and link quality estimates to improve energy efficiency by minimizing communication. Systems gather this state slowly to avoid introducing significant load. After rebooting, a node may take some time—minutes, even hours—to come fully back online. For example, in a recent deployment on Reventador Volcano, reboots from a software error led to a 3-day network outage [35], reducing mean node uptime from >90% to 69%.

This paper presents Neutron, a version of the TinyOS operating system that improves the efficiency and dependability of wireless sensor networks by reducing the cost of memory safety violations. Neutron has two parts: extensions to the TinyOS compiler toolchain (nesC and Deputy) and extensions to TinyOS itself.

Neutron extends the nesC compiler to provide boundaries between "recovery units." Similarly to microreboots [6], Neutron reboots only the faulting unit on a safety violation. TOSThreads, the TinyOS threading library, helps define application recovery units. Unlike microreboots, which operate only on application-level structures, Neutron must also be able to survive kernel faults, as the TinyOS kernel is typically the largest and most complex part of an application image. In Neutron, the kernel itself is a recovery unit. If the kernel violates safety, Neutron reboots it without disrupting application recovery units.

Rebooting a recovery unit is better than rebooting a node, but it conservatively wastes energy by discarding valid state. Neutron allows application and kernel recovery units to declare memory structures as "precious," indicating that they should persist across faults when possible. The complication is that precious state may be involved in a safety violation and become inconsistent. Neutron uses a combination of static analysis, type safety checks, and user-specified checks to determine which precious structures can be safely retained, and which must be re-initialized on a reboot.

Neutron must provide these mechanisms in the limited code space (tens of kilobytes) and RAM (4–10 kB) typical to ultra low-power microcontrollers. These constraints, combined with embedded system workloads, lead Neutron to take different approaches than are typical in systems that have plenty of available resources. By modifying variables in-place, Neutron introduces no instruction overhead in the common case of correctly executing code. In contrast, transactions would introduce a RAM overhead for scratch space and a CPU overhead for memory copies. Neutron re-initializes possibly corrupt variables, rather than restore them to their last known good states, because logging good states to nonvolatile storage has a significant energy cost.

Neutron minimizes its overhead through compiler techniques that leverage the static nature of TinyOS programs. For example, the component graph of a TinyOS program allows Neutron to infer recovery unit boundaries at compile time. Similarly, Neutron statically analyzes each memory safety check to determine which precious data structures may be in the middle of an update at the program point where the check occurs. When a safety check fails, Neutron does not preserve the contents of any precious data whose invariants are potentially broken. From the user's point of view, Neutron's interface consists of simple, optional annotations, making it easy to retrofit existing sensornet application code.

We evaluate Neutron by demonstrating that it isolates recovery units and increases application availability. We find that Neutron saves energy by preserving precious state across reboots. Our experiments are on a medium-sized network of 56 nodes and use two fault models. First, we model sporadic faults by periodically triggering reboots. Second, we reintroduced a fixed kernel bug back into TinyOS, to verify that Neutron improves dependability.

We find that in comparison to whole node reboots, Neutron reduces time synchronization downtime by 94% and routing control traffic by 99.5%. Furthermore, TinyOS kernel reboots do not lose application state or data. For complex sensor network applications, this increased robustness comes at a 3–8% code size increase over the equivalent Safe TinyOS image, and 1–6% increase in RAM usage. As Neutron only engages when there is a memory safety violation, it introduces no appreciable CPU overhead during normal operation. A Neutron sensor network can survive memory bugs and reboots, continuing to correctly report data without increasing energy consumption.

2. Background

Neutron builds on two recent extensions to TinyOS: Safe TinyOS for memory safety, and the TOSThreads [17] library for application programming. This section provides the necessary background on these systems, TinyOS itself, and core sensornet services that maintain valuable state. This information provides context for Neutron's design, and for how Neutron affects network efficiency and behavior.

2.1 TinyOS

TinyOS is a wireless sensornet operating system. Its mechanisms and abstractions are designed for ultra-low-power microcontrollers with limited RAM and no hardware support for memory isolation. TinyOS typically runs on 16-bit microcontrollers at 1–8 MHz that have 4–10 kB of SRAM and 40–128 kB of flash memory [26].

The operating system uses components as the unit of software composition [15]. Like objects, components couple code and data. Unlike objects, however, they can only be instantiated at compile time. TinyOS components, written in a dialect of C called nesC [12], have interfaces which define

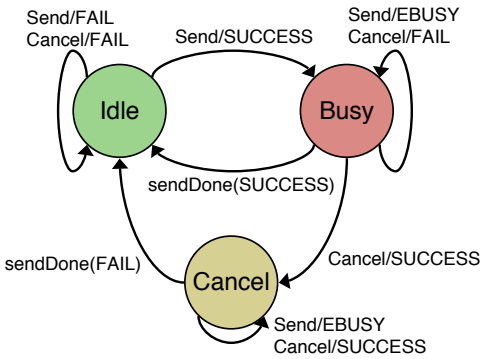


Figure 1. Simplified FSM for the TinyOS interface for sending a packet. A call to Send that returns SUCCESS moves the interface into the Busy state, at which point subsequent calls to Send return FAIL. The interface moves back to the Idle state when it signals the sendDone callback with a SUCCESS argument, indicating the packet was sent successfully.

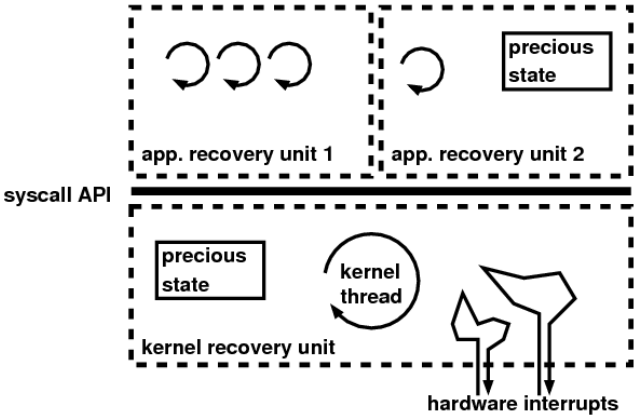


Figure 2. Neutron's structure in TinyOS. TOSThreads are grouped into application recovery units. The kernel thread is its own recovery unit. Recovery units can declare precious state.

downcalls ("commands") and upcalls ("events"). Upcalls and downcalls are bound statically: the absence of function pointers simplifies call graph analysis.

TinyOS interfaces, and components in general, are designed as simple finite state machines. Calls into a component cause state changes and sometimes cause the component to call other components. Many state changes are internal, but some are explicitly visible to other components. For example, Figure 1 shows the finite state machine of the standard packet send interface. Other components can call send() and cancel(); the component implementing the interface calls the sendDone() callback. In some cases, a call made in one state can have multiple results. For example, calling cancel() when the FSM is busy can either fail (packet transmission continues) or succeed (packet transmission is canceled).

TinyOS interfaces typically do not maintain caller state after returning to the idle state. For example, non-volatile storage abstractions do not maintain the equivalent of a seek pointer. Instead, they return an opaque "cookie" to callers on completion, which becomes a parameter to the next call in order to continue at the next byte. This is in contrast to traditional OS interfaces such as POSIX, where the kernel maintains a good deal of state on applications' behalf (sockets, seek pointers, etc.).

The TinyOS core has a highly restricted, purely event-driven execution model. Using a single stack, it supports only interrupt handlers and run-to-completion deferred procedure calls called *tasks*. Tasks are similar to interrupt bottom halves in UNIX implementations: they run at first opportunity and do not preempt one another.

Since tasks and interrupts do not retain stack frames while inactive, they cannot block. Instead, all I/O operations in TinyOS have completion callbacks. For example, in the Send interface shown in Figure 1, send() is an asynchronous I/O call and sendDone() is its completion callback.

2.2 TOSThreads

TOSThreads [17] is a preemptive threading library for TinyOS applications. To remain compatible with existing code, TOSThreads is careful to not break the TinyOS concurrency model. To maintain the invariant that tasks run to completion, TOSThreads runs all TinyOS tasks in a single thread that runs with the highest priority.

An application thread makes a system call by passing a message to the TinyOS thread. Passing a message posts a task: the TinyOS kernel thread runs immediately to handle the message. The application half of the system call stores the blocked thread's state. When the kernel half of a system call executes a completion event, that event resumes the thread. As no call within the TinyOS core blocks, using a single thread does not limit kernel concurrency.

The structure of concurrency in TOSThreads is thus not very different from a traditional uniprocessor microkernel OS. However, as microcontrollers have neither virtual memory nor caches, very simple message passing does not have the context switch overheads seen in traditional computer systems, such as those due to TLB flushes and cache misses.

2.3 Safe TinyOS

Since nesC is an unsafe language and TinyOS nodes lack memory protection hardware, pointer and array bugs lead to corrupted RAM and difficult debugging. Some microcontrollers place their registers in the bottom of the memory map, exacerbating the problem. On these architectures, null pointer dereferences corrupt the register file. For example, writing a zero to address 0x0 on an ATmega128 microcontroller [1] clears register 0 which, depending on the configuration, can reconfigure output pins F0–F7 or write to general purpose register R0.

UU IR Author Manuscript

Safe TinyOS [8] uses the Deputy compiler [7] to enforce type and memory safety using static and dynamic checks. Deputy is based on a dependent type system that exploits array bounds information already stored in memory. Therefore, unlike other memory-safe versions of C, it has no RAM overhead.

When Safe TinyOS detects a safety violation, it takes one of two actions. First, for debugging purposes it can repeatedly display an error message on the node's LEDs. Second, in a deployment setting it can reboot the offending node. If safety violations are infrequent, rebooting can increase the availability of a sensornet application.

2.4 CTP, FTSP, Tenet

Although reboots can increase availability, they are not free. Sensornet systems often build up state to provide useful services or to improve precision or energy efficiency. Rebooting a node clears this state. We present three examples where losing it is costly.

As a first example, applications use the Collection Tree Protocol (CTP) [13] to route data to a collection root. A CTP node maintains two tables. The link estimation table stores an estimate of the expected transmission (ETX) [9] cost of each link. The routing table stores the last heard route cost of a neighbor. Because the set of candidate links and next hops can be much larger than the table sizes, and it takes time to determine the best links and next hops, CTP spends significant effort to continually improve table contents. Rebooting a node re-initializes its tables, forcing a node to ask its neighbors to advertise their presence so it can rediscover good neighbors and link qualities.

As a second example, the Flooding Time Synchronization Protocol (FTSP) establishes a uniform global time over the network [21]. Nodes periodically broadcast packets, and receivers calculate the drift between their local clock and the global clock using linear regression. Each node stores a table of drift coefficients to build an estimate. Rebooting a node flushes this table, forcing a node to recalculate coefficients. While coefficient estimation is in progress, the node is unsynchronized and calls to get the global time fail. Furthermore, if the global time root node reboots, all other nodes in the network fall out of synchronization.

Lastly, Tenet is an application-level programming interface for sensor network applications [14]. Users write programs in a data flow language that compiles to format that is run by an on-node interpreter. In order to save RAM by reducing caching requirements, Tenet's program dissemination protocol operates on a finite time window. Nodes do not execute a new program they hear if it is more than 15 minutes old. If a node reboots due to a safety violation, it will not execute an older program. Therefore, a single node reboot may force an administrator to reprogram the entire network.

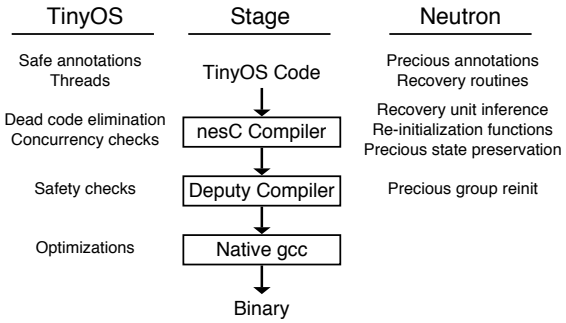


Figure 3. Neutron's extensions to the TinyOS toolchain and build process, shown on the right

3. Neutron Overview

This section presents Neutron, a collection of extensions to TinyOS and its toolchain that address the high cost of whole-node reboots. Figure 2 illustrates Neutron's extensions to TinyOS. Neutron changes how a node responds to a Safe TinyOS memory safety violation. Rather than output an error message or reboot the entire node, Neutron organizes threads and data into "recovery units" that can be rebooted independently.

Neutron supports multiple *application recovery units* that interact with the TinyOS kernel through a blocking system call interface. It also supports a single *kernel recovery unit*: the event-driven TinyOS kernel which contains one thread, all interrupt handlers, and all kernel data structures. Extensions to the nesC compiler derive these recovery units and automatically generate code for rebooting them.

In addition to limiting reboot scope with recovery units, Neutron allows components to declare memory structures as "precious." By default, Neutron re-initializes a recovery unit's variables when it reboots that unit. If a variable is precious and passes a set of conservative checks, however, Neutron can allow the variable to persist across a reboot. This persistence operates at a component level: if any variable in a component fails a check, Neutron re-initializes all of them on reboot. Extensions to the nesC and Deputy compilers generate these checks and restoration procedures.

Figure 3 shows Neutron's extensions to the TinyOS toolchain. Both applications and kernel are in the nesC language and can specify precious state. The nesC compiler generates C code, which the toolchain passes to the Deputy compiler. Deputy deals with safety annotations, checks types, and generates C code with safety assertions to enforce its safety model. Finally, GCC cross-compiles the output of Deputy to machine code.

In order to correctly reboot recovery units, Neutron needs to properly implement C initializers[1] that normally execute as part of the bootloader. The Neutron nesC compiler generates C functions that re-initialize variables to the same state

[1] Such as `int *a = &b;` in file scope.

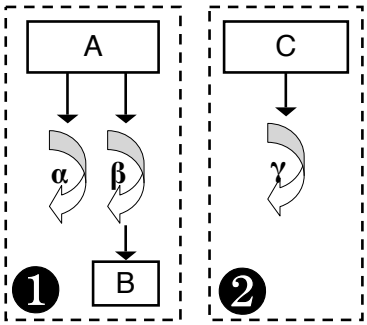


Figure 4. An example TOSThreads application that is forward-compatible with Neutron. Neutron infers two recovery units, 1 and 2, surrounded by dotted lines. Components A and B are both in Unit 1 as they may share state through Thread β. Component C is placed in its own recovery unit, Unit 2.

as if the bootloader had run. Adding this code to a TinyOS image increases its ROM usage; we evaluate this cost in Section 7.

The next three sections detail Neutron's mechanisms and implementation. Section 4 covers how Neutron reboots application recovery units. Section 5 covers how Neutron reboots the kernel recovery unit. Finally, Section 6 covers how Neutron restores precious variables across a recovery unit reboot.

4. Application Recovery Units

An application recovery unit contains one or more TOSThreads and their associated variables. Neutron derives application recovery unit boundaries at compile-time. When one thread in an application recovery unit has a violation, Neutron halts all threads in that unit and cleans up their state, including pending system calls.

4.1 Delineating Application Recovery Units

The nesC compiler, modified for Neutron, infers recovery unit boundaries at compile-time by finding a solution to the following constraints. It minimizes application recovery unit size (puts components in separate units whenever possible), given:

1. An application recovery unit may not call directly into a different recovery unit.
2. An application recovery unit instantiates at least one thread.
3. Every nesC component (and by extension, every thread and variable) above the system call interface belongs to at most one application recovery unit. NesC's reachability analysis and dead code elimination mean that application components not reachable by any thread will not be included in the TinyOS image.
4. Every nesC component below the system call interface belongs to the kernel recovery unit.
5. The kernel recovery unit has one thread.

The Neutron compiler issues an error if the constraints are not satisfiable. For example, if a component makes a direct function call on a kernel component, constraint 1 implies it is part of the kernel recovery unit. If this same component instantiates a thread, it violates constraint 5.

The nesC compiler contains no special support for TOSThreads. On the other hand, for Neutron's recovery unit inference algorithm to work, the compiler has to be aware of the system call boundary. We accomplish this using two special-purpose nesC annotations: `@syscall_base` and `@syscall_ext`. These annotations mark components that implement the Neutron system call API. The former simply marks a component, while the latter marks that a component needs to be parameterized—at compile time—by a small integer identifying an application recovery unit. These "PIDs" are used for bookkeeping purposes inside the Neutron kernel; they are not exposed to applications. The code below shows the use of these two annotations:

```
/* definition */
configuration BlockingActiveMessageC @syscall_base() {}
/* instantiation */
components BlockingActiveMessageC;

/* definition */
generic configuration SafeThreadC(uint16_t stack_size,
    uint8_t pid) @syscall_ext() {}
/* instantiation */
components new SafeThreadC(100);
```

Both `BlockingActiveMessageC` and `SafeThreadC` are labeled as system call APIs, using `@syscall_base` and `@syscall_ext` respectively. Neutron instantiates syscall-related components using the nesC compiler's parameterized component feature to generate a PID for each recovery unit and to generate code passing these PIDs into the kernel as necessary. An additional benefit of this approach is that kernel data structures are automatically sized appropriately, including the special case where there are zero application recovery units. When there are no application recovery units, Neutron introduces no RAM overhead.

4.2 Isolating Application Recovery Units

Neutron isolates recovery units using three mechanisms: namespace control, analysis of the application's component linking graph, and Deputy's memory safety. The nesC language's component model provides namespace control, ensuring that the local state of components is only exposed through components' interfaces. Because nesC requires all components to explicitly declare what functions they call, the compiler can walk the component linking graph to statically determine all component interactions. Neutron exploits this property to ensure that applications only use the system call API, even though applications and the kernel run in the

same physical address space. Finally, Deputy's type safety checks prevent code from exploiting pointer and array violations to cross a recovery unit boundary. In summary, Neutron statically prevents a recovery unit from naming resources in application recovery units, and dynamically prevents it from fabricating pointers or exploiting other backdoors.

4.3 Termination Overview

The TOSThreads system call API provides a natural boundary on which to build termination-safe abstractions. TOSThreads' message passing structure means that application and kernel recovery units are only very loosely coupled: terminating an application does not require unwinding a stack running kernel code.

When an application recovery unit has a memory safety violation, Neutron reboots it in four steps. First, Neutron cancels in-progress system calls and halts all runnable threads in the recovery unit. Second, Neutron reclaims any dynamically allocated memory. Third, Neutron re-initializes the application unit's RAM. Fourth, Neutron restarts the application unit's threads. The rest of this subsection explains these steps in more detail.

4.4 Canceling System Calls and Halting Threads

A thread cannot violate memory safety while it is blocked on a system call. However, an application recovery unit can have one or more blocked threads when the active thread faults. Neutron therefore needs to safely and correctly reboot threads that are blocked on system calls. Doing so is the most complex part of the reboot sequence of an application recovery unit.

The problem of interrupting a thread blocked on a system call is analogous to POSIX signal handling in UNIX-based operating systems. As cleanup can complicate system call code, multitasking OSes usually distinguish "fast" and "slow" system calls [33]. Fast calls, such as disk reads and simple functions, are expected to terminate at some point. Slow calls, such as reading from a terminal or accepting a connection, may never terminate. While a signal can interrupt a slow system call, fast system calls wait for the call to complete before issuing a signal.

Neutron keeps track of which queue each application thread is blocked on, if any. When Neutron reboots a recovery unit, a component in the kernel walks this list and removes threads from the queues. System calls will therefore not resume their waiting threads and threads on the ready queue are halted.

Because Neutron passes messages between application threads and the kernel thread, it is straightforward to interrupt any system call by removing the thread from the system call structure. Neutron performs this step when it halts all threads. In the case of slow system calls that wait on external events (such as packet reception), the application half of the system call discards future events, as if no thread were blocking on it. In the case of fast system calls where there is a completion event (such as sending a packet), the application half of the system call ignores the completion event for the canceled call. However, in the case of fast system calls, the presence of an outstanding request means that Neutron must perform extra bookkeeping.

4.4.1 Re-Execution of Cancelled System Calls

System call cancellation has an important corner case when a system call cannot be cleanly canceled. If a restarted thread tries to re-execute the same system call, Neutron needs to hide the effects of the earlier reboot, as the system call is still pending.

For example, consider recovery unit 1 in Figure 4, with two threads α and β. Thread α is blocked on a call to send a packet. Thread β violates the safety model, causing Neutron to reboot unit 1. Neutron removes α from the system call structure then restarts both threads. Thread α restarts and calls send again. Following the send interface FSM (Figure 1), this call will fail, as the kernel is busy. The reboot is now visible to the application.

Neutron solves this problem by maintaining a pending flag in the kernel for system calls with completion events. When a thread that is blocked on such a call is rebooted, Neutron marks that system call as pending. When a thread executes a system call, it checks the pending flag. If the pending flag is true, the kernel blocks the request until the event signaling completion of the prior request has arrived. In case of send(), for example, it blocks on the sendDone() event for the pending send. When a completion event arrives, if the pending flag is set, Neutron clears the flag and then starts the system call of the blocked thread. Neutron does not allow system calls to immediately re-execute after a reboot due to the memory cost of maintaining multiple pending calls.

4.4.2 System Calls with Pointers

System calls with pointer arguments transfer ownership of buffers between application threads and the kernel thread. TOSThreads does not copy across the user/kernel boundary because of the memory cost doing so would entail. Neutron waits for pending system calls with pointer arguments to complete before rebooting the recovery unit. Otherwise, an application could reuse a buffer that belongs to the kernel, resulting in data corruption. Worse yet, the buffer could be on the stack. Therefore, long-running system calls with pointer arguments can introduce latency between a safety violation and reboot. This issue is not unique to Neutron: timeouts on NFS operations in UNIX operating systems, for example, can delay signal delivery by minutes [33].

4.5 Memory and Restarting

After halting all threads, Neutron needs to free any dynamically allocated memory, re-initialize static memory, and restart threads.

Because Neutron builds on Safe TinyOS, it can assume that the heap is not corrupted. Neutron modifies the standard TOSThreads malloc implementation (a boundary tag allocator based on the implementation from msp430-gcc's libc) to add a process identifier to each allocated chunk. When a recovery unit reboots, Neutron scans the heap and frees all chunks allocated by the unit. Because heaps in sensornet nodes are tiny (a few kB), walking the heap is fast.

After recovering memory, Neutron re-initializes the application recovery unit's variables using code generated by the nesC compiler. It then restarts application threads by issuing their boot events as if the node had just booted.

5. The Kernel Recovery Unit

This section describes how Neutron reboots the kernel recovery unit. In a traditional OS, an application may have large amounts of dynamically allocated state in the kernel, such as page tables, file descriptors, I/O streams, and shared memory regions. In TinyOS, however, this state is very limited. There is no virtual memory. Following nesC's model of static allocation, descriptors and state for non-volatile storage are allocated and configured at compile-time.

These allocation approaches lead to very loose coupling between application threads and the TinyOS kernel. Keeping an application recovery unit runnable across a kernel reboot requires maintaining a small number of data structures.

5.1 Application State

The TinyOS kernel maintains four pieces of application state. First, the TOSThreads scheduler, which maintains three thread pointers (the running thread, the kernel thread, and the yielding thread), the head of the ready queue, and a counter of the active application threads. The counter saves energy by disabling the time slice interrupt when no application is active.

Second, there are the thread control blocks and stacks. Because TinyOS statically allocates the state for each TOSThread, these memory structures are defined within the kernel, rather than in application components.

Third, there are system call structures. Active calls describe which application threads are blocked awaiting messages from the kernel. Re-initializing this state removes threads from wait queues but does not make them ready to run: they will never return.

Finally, there are the system call implementations themselves. This is important because system calls will malfunction if kernel components fail to follow their interface state machines. For example, if an application issues a send system call and the kernel reboots the radio mid-transmission, the network stack components will not issue a completion event. This lack of an event will cause the send request to block indefinitely.

5.2 Keeping Applications Runnable

The very limited application state in the TinyOS kernel, combined with the simple FSMs of system calls, makes it possible for Neutron to reboot the TinyOS kernel thread without disrupting applications.

Neutron does three things to keep application threads runnable when recovering from a kernel safety violation. The first is canceling all outstanding system calls. A canceled system call returns a retry error code to the application code, permitting it to remain in sync with the reboot-induced kernel state change. Canceling system calls places once-blocked threads on the ready queue. The second is protecting application-level kernel state, such as application thread control blocks and stacks, from re-initialization on kernel reboot. The third is protecting the thread scheduler itself from re-initialization on kernel reboot.

Canceling pending system calls means it is safe to re-initialize system call structures. When the node reboots, there are no blocked threads. Protecting application thread structures and the thread scheduler by making their values persist across the reboot solves the issues of maintaining thread state. Finally, completely re-initializing the rest of the kernel resets the state machines of kernel components that implement system calls.

5.3 Implementation

Implementing Neutron's kernel thread reboot policy requires changing the TinyOS boot sequence. The standard TinyOS boot sequence goes through three initialization steps: low-level hardware, platform, and software. These respectively include actions such as setting each I/O pin to its lowest power state, setting oscillator sources and clock speed, and initializing queues.

Neutron separates the software initialization step into two parts: kernel state and thread state. On first boot, it runs both. The kernel reboot handler, however, skips thread state initialization.

The memory structures handled by thread state initialization include the application threads, the thread scheduler, and system call gates. Any component that needs to be maintained across kernel reboots can register with this initialization routine: adding new application-dependent kernel state is easy.

6. Precious State

When a safety fault occurs, the state of either an application or the kernel—by definition—violates the safety model. The obvious solution, rebooting the kernel or restarting an application, reverts the faulting recovery unit to an initial, safe state. While separating applications from each other and from the kernel limits how much state a reboot loses, this solution is still highly conservative.

After inspecting a number of applications, we concluded that most state could be reverted to an initial value with little

cost. However, a few key data structures cost substantial time and energy to rebuild. Our hypothesis was that in the common case, expensive soft state would not be corrupted, permitting it to persist across reboots and avoiding the cost of rebuilding it. To support this idea, Neutron introduces a "precious" annotation that is implemented by the Neutron nesC compiler. When possible, precious data retains its value even when the recovery unit containing it is rebooted. The new annotation is used as follows:

```
TableItem @precious() table[MAX_ENTRIES];
uint8_t @precious() tableEntries;
```

Unlike C's *volatile* and *const* type qualifiers, the nesC precious attribute may only be applied at the top level of a variable: struct and union fields cannot be independently precious or not precious.

6.1 Precious Groups

The compiler divides a recovery unit's precious variables into *precious groups*. A precious group is all precious state declared within a single nesC component. When a recovery unit reboots, the Neutron kernel decides if each precious group belonging to that unit is safe or potentially corrupted. If Neutron suspects that the precious state is corrupted, it re-initializes the state as it would on the system's initial boot. Otherwise, it saves and restores the precious state across a reboot.

Since precious groups are separately persistent, they must be semantically independent. The example code above has the invariant that tableEntries specifies the number of valid entries in the table array. If only one of these variables were persistent across a reboot, the invariant would no longer hold in the new instance of the recovery unit. Thus, semantically connected precious data must be declared within the same nesC component. Furthermore, in Neutron it is forbidden—though these rules are not yet checked by the compiler—for pointers to refer: across precious groups, from precious data to non-precious data, or from precious data into the heap. The static design of TinyOS discourages heavy use of pointers and it is fairly uncommon for pointers to cross nesC interfaces.

6.2 Balancing Efficiency and Integrity

Although precious state can be used to reduce the energy and availability penalties of reboots, it carries the risk of propagating corrupted data into the next instance of the kernel or application recovery unit. Neutron's high-level goal is to propagate precious state across a reboot when no reasonable doubt exists about its integrity. Neutron uses several analyses and heuristics to avoid propagating corrupt precious data.

6.2.1 Preventing Termination-Induced Corruption

Persistence and termination have the potential to interact poorly. For example, a thread in an application recovery unit, or an interrupt handler in the kernel recovery unit, can be preempted in the middle of updating a precious data structure. This can leave its invariants broken. The preempting thread or interrupt (which belongs to the same recovery unit) violates safety, causing the unit to be rebooted. On reboot, the recovery unit will see inconsistent precious data.

To avoid this problem, updates to precious data must occur within an `atomic` block supported by the nesC language. Modifications to precious variables outside an atomic block are a compilation error. Atomic blocks in nesC are independent and serializable with respect to concurrency, but they are not transactional in terms of durability. If a thread or interrupt violates safety while updating a precious data structure, any stores issued before the violation are persistent and will leave the precious data in an inconsistent state.

To solve this problem, Neutron uses a lightweight static analysis to compute, for each memory safety check inserted into an application, a conservative estimate of the precious data structures that could be in the middle of being updated. This analysis operates in two stages. In the first stage, the compiler analyzes each atomic block to find which, if any, precious data structures it may update. In the second stage, the compiler marks all code reachable from each atomic block as being "tainted." If a memory safety check in tainted code fails, the associated precious data structures are re-initialized to a clean state. At the implementation level, the Neutron compiler associates a bit-vector with each memory safety check in an application, where the bits specify which groups of precious state must be re-initialized when that check fails. Typically, little TinyOS code is actually reachable from atomic blocks that update precious state, due to TinyOS's tendency to use small, short-running events. Therefore, although our analysis is quite conservative, it does not lead to significant false sharing and unnecessary loss of precious state. Even so, it is critical for dependability that the interaction of rebooting and precious state avoids corrupting that state.

6.2.2 Defending Against Other Sources of Corruption

Deputy's memory safety model cannot defend against all types of data corruption: a stack overflowing into another memory region, a safety violation in trusted code, or buggy-but-safe application logic could all lead to inconsistent precious state. Neutron uses three checks to avoid propagating inconsistent state across a reboot.

For each precious group, the developer can optionally write a `check_rep` function that checks a collection of data for application-level consistency. Neutron calls this function following a safety violation; Neutron re-initializes the precious group if the check fails.

Second, when Neutron detects a safety violation that directly involves a precious data structure (such as a negative array index in a routing table), the precious group containing that data structure is automatically re-initialized as part of the reboot.

Lastly, one consequence of propagating corrupted state across a reboot is that subsequent reboots are likely to happen more often. To address this, Neutron could be parameterized by a time window and a *maximum reboot count*. If a violation occurs more than the specified number of times within the time window, a clean reboot is performed, wiping out all precious state. Our implementation of Neutron does not include this feature, as the time scales of our experiments in Section 7 would trigger it unnecessarily.

6.3 Implementation

Neutron modifies the nesC compiler to generate routines that save and restore precious state. Neutron also instructs the C compiler to place different precious groups' initialized and uninitialized variables into separate .data and .bss segments. When a recovery unit reboots, Neutron re-initializes all non-precious variables and also each precious group that fails any of the heuristic checks described in Section 6.2. As the reboot sequence has already re-initialized variables, failing to restore a variable returns it to its initial state.

Neutron goes through five steps to restore precious data:

1. check precious variables for possible corruption;
2. push persisting variables on the stack;
3. copy initial values from ROM to the recovering .data section;
4. zero the recovering .bss section; and
5. pop persisting variables, replacing initial values.

Placing a recovery unit's variables in a contiguous region of memory has two benefits. First, aggregation makes recovering a precious group simple and fast, as it is a single memcpy operation rather than a series of individual writes. Second, RAM is saved by copying precious data onto the stack, which is nearly empty during a reboot. In the common case where precious data fits into the existing stack memory region, no additional stack memory needs to be allocated to support precious data.

7. Evaluation

This section evaluates Neutron's improvements to the dependability, efficiency, and robustness of TinyOS applications. It measures the cost of whole-node reboots for two TinyOS services, FTSP and CTP, and a sample Tenet program. It measures the extent to which precious state allows Neutron to reduce the cost of rebooting the recovery unit containing the service, in the case where the precious state is not corrupted. Through controlled and uncontrolled reboot scenarios, it verifies that Neutron limits the effect of a safety violation to the corresponding recovery unit. Finally, it measures the RAM, ROM, and CPU overhead Neutron introduces due to the need to save state and selectively re-initialize variables.

Service	Variable	Purpose
CTP	routingTable	Network-layer route costs of neighbors
	routingTableActive	Count of valid table entries
	linkEstSeq	Counter for beacon packets
	currentInterval	Current beaconing interval
	prevSentIdx	Last neighbor estimate sent in beacon
FTSP	table	Table of clock drift coefficients
	tableEntries	Number of coefficients in table
	skew	Aggregate result from table
	localAverage	Aggregate result from table
	offsetAverage	Aggregate result from table
	heartBeats	Freshness of table entries
	TBR	Timer B counter register

Figure 5. Precious variables in CTP and FTSP

7.1 Methodology

All experiments use Telos rev. B sensor nodes [26]. A Telos has an MSP430F1611 microcontroller with 10 kB of RAM, 48 kB of program flash, and a 2.4 GHz 802.15.4 radio. Since the radio operates in the same band as 802.11b, WiFi traffic can interfere with 802.15.4. All experiments use 802.15.4 channel 16, with overlaps with WiFi channels 4–8.

For network experiments, we use the Tutornet testbed at USC, consisting of 56 TelosB sensor nodes deployed above the false ceiling of a single floor of a large office building. Small embedded PCs provide debugging and instrumentation backchannels. The combination of a false ceiling and heavy WiFi interference makes the testbed a realistic setting for evaluating network protocols.

As a first step, we evaluate Neutron by inducing controlled reboots through randomized triggers. In a real system, the reboot could be caused by safety violations but the result is the same: interrupted execution while the whole node reboots.

To explore a less controlled test case, we also re-introduce a fixed bug from TinyOS. An SPI (serial peripheral interface) bus driver in TinyOS 2.0.0 had an off-by-one error. If the SPI master asked to read a zero-byte message, the driver would read 65,535 bytes instead. Reading 65,535 bytes overwrites the stack, making the return address of the stack frame 0x0 (the reset vector). An unforeseen edge condition in the CC2420 stack would trigger this bug. We recreate this bug in the drivers for the MSP430 microcontroller of the TelosB.

To validate Neutron's ability to detect corrupted precious state, we introduce safety violations in code that accesses precious state. This causes a reboot where Neutron must recognize that a precious structure might be corrupt and re-initialize it.

7.2 Precious State

This section describes the variables we declared precious in CTP, FTSP, and Tenet. Each of the system services we evaluate (introduced in Section 2.4) maintains valuable in-memory structures. In the case of FTSP and CTP, nodes fill data into these structures over time. Losing this state

therefore harms efficiency. To a smaller degree, it also harms uptime, as a node can be unsynchronized, unable to deliver data, or improperly configured until it regenerates the proper state. Tenet, in contrast, stores a task description that, if lost, cannot be recovered without manual intervention.

CTP has three precious data structures: its routing table, its link estimation table, and its beaconing interval. Two variables govern the routing table: `routingTable`, an array of table entries, and `routingTableActive`: the number of valid entries. The link estimation table is stored in an array `neighborTable`. For beaconing, `linkEstSeq` is a sequence number for inferring beacon losses while `currentInterval` stores the interval between beacons. Finally, `prevSentIdx` is an index into the neighbor table for cycling through entries to report.

FTSP has three precious data structures: its neighbor drift estimation table, its current time estimates, and the local clock. The drift estimation table is stored in `table` and `tableEntries` counts how many entries are valid. The variables `skew`, `localAverage`, and `offsetAverage` are the aggregate estimates from the table. The counter variable `heartBeats` validates that the table values are up-to-date. Finally, since FTSP uses the local node clock, it marks the counter register as precious.

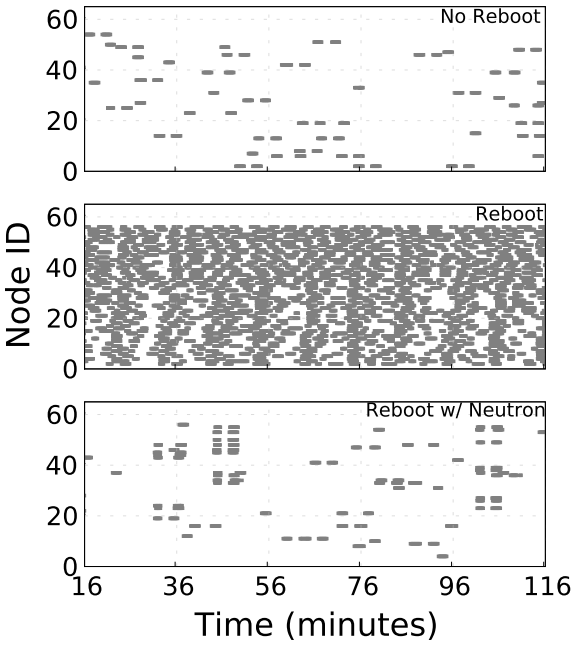
Tenet has one precious structure, its program descriptions. When Tenet reboots, it maintains loaded programs. To support this feature, we had to modify the Tenet interpreter slightly: on boot it checks if it has programs in memory, and if so executes them.

Figure 5 summarizes these annotations.

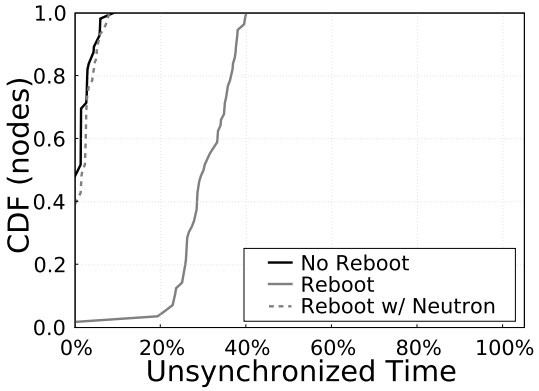
7.3 Cost of Reboots

This section quantifies the cost of reboots. For each of the three services that we consider, we examine the system when there are no reboots, when there are reboots and no precious state, and when there are reboots and precious state. Bugs in deployed sensornets are typically rare (e.g., occurring every few days), but running statistically significant multi-day experiments is infeasible. Given a particular fault rate, we measure the relative cost increase Neutron observes as a fraction of the cost increase due to a whole node reboot. Thus, the measured increase in the cost of faults is independent of the fault rate.

For FTSP, reboots increase the time for which nodes are unsynchronized. Figure 6 shows that without precious state, reboots cause significant desynchronization. These data are from a two-hour, 56-node run of FTSP where each node rebooted every five minutes with some randomization to prevent synchronized behavior. The first 1000 seconds of data were discarded to let the network reach a steady state. Packet losses and other real-world variations cause a non-zero amount of unsynchronized time even in the no-reboot case. Without reboots, the median node is out of synch 1.3% of the time. With reboots, it is out of synch 30% of the time: this time is very large due to the artificially high reboot rate.



(a) Unsynchronized time for each node is marked in gray



(b) CDFs of unsynchronized time

Figure 6. For the Flooding Time Synchronization Protocol (FTSP), precious state reduces the penalty of a reboot, in terms of time spent without time synchronization, by a factor of 17

With this high reboot frequency and precious state, however, the median node is out of synch 1.7% of the time. Precious state reduces the cost of reboots by 94%, a factor of 17.

For CTP, the cost of a reboot is a burst of control packets for a node to rediscover its neighbors and estimate link qualities. Figure 7 shows the cumulative number of beacons sent during three 25-minute experiments on the same 56-node testbed. At around 450, 930, and 1400 seconds into two of the three runs, a single node reboots. Without precious state, three reboots nearly triple the total number of

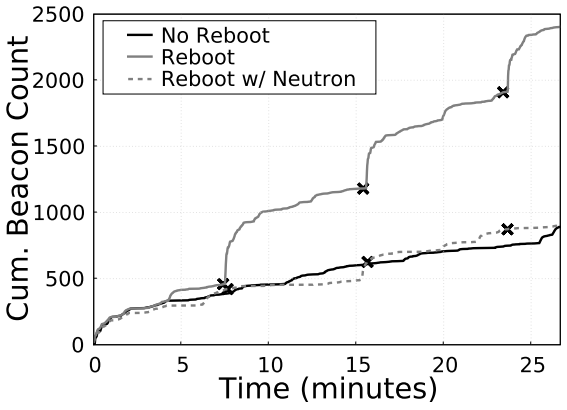


Figure 7. For the Collection Tree Protocol (CTP) without precious state, each reboot of a single node leads to a surge in control traffic; three reboots more than doubles the total number of beacons sent. Precious state eliminates the surges in traffic.

beacons the network sends, from 894 to 2,407. Rebooting with precious state, however, causes 901 beacons to be sent. Precious state reduces the increase in beacons triggered by reboots by 99.5%.

For Tenet, a reboot's cost is the need to reprogram an entire network. Without Neutron, a safety violation clears Tenet programs. Using Neutron, Tenet programs should persist across a reboot.

To validate this behavior, we performed two experiments using a Tenet program that collects and sends light sensor readings. First, we introduced code that triggers a safety violation six seconds after Tenet starts running a program. After six seconds, the node stopped reporting. In the second experiment, key data structures that describe programs and the state of task execution are precious. With these precious structures, the node continued to report light samples after the reboot. This experiment demonstrates that we are able to preserve execution state of an application and avoid loss of application control information or data loss.

7.4 Validating Recovery Unit Isolation

We perform a series of experiments to validate that Neutron properly isolates recovery units from each other and that it correctly re-initializes corrupt precious state. These tests use a Neutron application that takes sensor readings and stores them into a buffer. When a buffer fills, it is put into a packet and sent to the base station using CTP as the routing layer. The CTP service runs in the kernel recovery unit, and the sampling and buffering run in an application recovery unit.

Isolating the kernel from application faults: A modified version of the application periodically accesses an out-of-bound array element, violating memory safety. Without Neutron, the resulting node reboot re-initializes CTP; this is visible at the base station because CTP's packet sequence numbers get reset to zero. With Neutron, the sequence numbers increase monotonically.

Isolating applications from kernel faults: For this experiment, the kernel sporadically violates memory safety via the SPI bug described in Section 7.1. Without Neutron, a kernel reboot wipes out the application's buffer, losing samples. With Neutron, the reboot does not affect the application recovery unit: all samples arrive successfully at the base station.

Correctly preserving/re-initializing precious state: We further modify the sampling application to read from two sensors at different rates, placing the results into separate buffers. The two buffers are marked as precious along with the corresponding indices. Since the components share a timer, they are in the same application recovery unit. However, since the buffers reside in different components, they are in different precious groups. That is, Neutron separately decides whether each buffer (along with its current index) is to be persistent across a reboot. As before, an out-of-bound access causes a safety violation and the application reboots. The application logs show that Neutron correctly preserves one of the buffers, while re-initializing the other one.

7.5 Programmer cost

This section evaluates the implementation burden that Neutron imposes on sensornet application developers.

Application recovery units: Because Neutron automatically infers the boundaries of recovery units, existing TOSThreads applications can be ported to Neutron with near zero programmer effort. The behavior of a fault-free application is never changed by Neutron: Neutron's code runs in the context of the safety violation failure handler. Rebooted application recovery units may have their behavior changed by Neutron if, for example, they interact with a stateful hardware device that is outside of Neutron's framework, such as non-volatile storage. A practical solution to this kind of problem is to add, for example, atomic flash memory manipulation primitives to Neutron's system call API.

Precious state: To fully benefit from Neutron, a programmer must annotate applications to express precious state. Depending on the complexity of an application, the number of annotated data structures and variables can be different. However, each annotation only requires a simple `@precious()` code addition.

Pre-existing annotations in TinyOS, such as those in CTP and FTSP, reduce programmer burden. The simplest TinyOS application, Blink, has no precious annotations. FTSP uses 11 and CTP uses seven. The kernel uses an additional six precious annotations to avoid disrupting applications when the kernel reboots.

Our experience is that for complex TinyOS components, such as the ones implementing FTSP or CTP, finding the right set of precious variables is not incredibly difficult. The difficult cases requiring some trial-and-error involve components that include many loosely-coupled variables,

	Safe TinyOS	Neutron	Increase
Blink	6402	8978	40%
BaseStation	26834	31556	18%
CTPThreadNonRoot	39636	43040	8%
TestCollection	44842	48614	8%
TestFtsp	29608	30672	3%

Figure 8. Code size of applications (bytes)

	Safe TinyOS	Neutron	Increase
Blink	1031	1090	6%
BaseStation	3580	3764	5%
CTPThreadNonRoot	2890	3000	4%
TestCollection	3098	3228	4%
TestFtsp	1354	1352	0%

Figure 9. RAM use of applications (bytes). As TestFtsp has no application threads, it has a single recovery unit and introduces no RAM overhead.

each of which could plausibly be made either precious or non-precious.

7.6 Overhead

Neutron introduces several types of overhead to TinyOS applications. First, Neutron's mechanisms can increase program size. The selective initialization vector, application recovery unit clean-up, and reboot code all introduce extra code. Second, because Neutron must selectively re-initialize variables on reboot, its memory initialization sequence can take longer than the standard assembly routine produced by the C compiler.

We evaluate these overheads on five representative TinyOS applications. The first is Blink, a trivial application that toggles three LEDs. As Blink is effectively only a timer, it represents the boot time dedicated to hardware initialization. The second application, BaseStation, bridges packets between the radio and serial port. CTPThreadNonRoot is the third application, a threaded application that sends packets using CTP; as it is not a root of the collection tree, it does not use the serial port. The TestCollection application extends CTPThreadNonRoot to also support being a collection sink: it includes the serial stack in its image. Finally, TestFtsp is a simple test of FTSP, which periodically sends messages to the serial port describing whether it is successfully synchronized, its local time, and its perceived global time. TestFtsp has no application threads, so the TinyOS kernel is its only recovery unit.

ROM: The increase in code size is due to three types of code additions. The majority of the overhead comes from code managing the recovery units within a node. There is also code for handling re-initialization of global variables after a reboot and routines for copying and restoring precious data across failures. Figure 8 shows the results. Neutron increases code size by 3–40%.

Over 90% of the additional code in our applications is the constant overhead of unit management. While small applications see a large percentage increase in their code size, larger applications—the ones which actually struggle with code size limits—see only a small percentage increase.

RAM: Precious state and recovery also increase application RAM use, as shown in Figure 9. The RAM cost varies within a relatively small range, from a very slight decrease for TestFtsp to 6% for Blink.[2] Neutron requires very little extra state to correctly handle pending system calls. In the case of Blink, for example, where there is 59 byte (6%) overhead, 32 bytes of this is for tracking pending system calls and 25 bytes is for tracking threads. By placing precious variables on the stack during a reboot, Neutron minimizes its RAM needs. Furthermore, as the stack is very small when they are pushed, Neutron typically does not increase the maximum stack size.

Reboot time: Figure 10 shows the reboot times of the applications. Because Neutron has to copy back preserved precious state and selectively re-initialize variables on a reboot, reboots can take up to 1.54x as long as an initial boot.

However, Neutron's recovery units mean that application failures do not reboot the kernel. Figure 11 presents a breakdown of reboot time. In all cases, the majority of the time

	Boot	Reboot	Increase
Blink	10.3	12.2	18%
BaseStation	16.2	22.1	36%
CTPThreadNonRoot	10.1	15.6	54%
TestCollection	10.8	15.6	44%
TestFtsp	12.6	14.8	17%

Figure 10. Boot and reboot times for TinyOS applications using Neutron (milliseconds). Reboots take up to 54% longer than boot due to the need to selectively re-initialize variables.

	Node	Kernel	Application
Blink	12.2	11.4	1.16
BaseStation	22.1	14.1	9.18
CTPThreadNonRoot	15.6	15.5	1.01
TestCollection	15.6	15.5	0.984
TestFtsp	14.8	-	-

Figure 11. Reboot times (milliseconds) for sample threaded applications. In the case of application faults, Neutron can recover much faster than a whole node reboot by avoiding the cost of rebooting the kernel.

[2] The decrease in RAM usage for TestFtsp is an accident resulting from the way in which data alignment requirements interact with our reorganization of recovery units' data into new ELF sections.

is spent on kernel re-initialization. Applications other than BaseStation can be rebooted in 1–2 milliseconds. BaseStation takes about nine milliseconds to reboot the application recovery unit, which has five worker threads; about seven milliseconds are spent re-starting those threads. In general, when a Neutron application recovery unit violates safety, the node comes back to full operation rapidly.

8. Related Work

This section describes some of the prior work that motivated us to build a language-based OS for sensor network nodes. Safe TinyOS [8] and TOSThreads [17] form the foundation for Neutron. Broadly speaking, Neutron draws its ideas from three main sources: language-based operating systems, reboot-based mechanisms for improving reliability, and system support for persistent state.

8.1 Language-Based Protection in an OS

Most operating systems use the MMU to isolate processes, but some OSes have instead used language-level safety. For example, Singularity [16], KaffeOS [2], and SPIN [4] are respectively based on the type-safety provided by C#, Java, and Modula-3. Even so, each of these systems depends on low-level unsafe C code that is significantly larger than a typical TinyOS application. In contrast with these from-scratch systems, Neutron builds a protected-mode OS that is almost entirely backwards compatible with TinyOS. For this reason, Neutron is perhaps most closely related to projects such as SafeDrive [37] and Nooks [30] that support rebootable execution environments in legacy kernels.

Unlike most previous language-based OSes, Neutron does not rely on garbage collection. In fact, Neutron does not require any dynamic memory allocation at all (although applications that use a heap are supported).

8.2 Recovering from Faults

Microreboots [6] examined how to reboot fine-grained system components in Java Enterprise Edition [29] Internet services. Microreboots depend on individual operations being idempotent and stateless, and requires a backing transactional store for completed operations. Where microreboots store all persistent state in a transactional database, Neutron modifies variables in place and relies on integrity checks to detect when they are inconsistent. Furthermore, rather than focus on high-level Internet services, Neutron operates on the lowest levels of an embedded operating system.

Rx [23] and recovery domains [18] use a combination of checkpointing and re-execution to recover from software errors. While similar in goals, Rx tackles user applications and recovery domains tackle the kernel, and so they respond to faults differently. Rx changes the re-execution environment in order to hopefully avoid the same error, while recovery domains return an error to the user process. Like both of these techniques, Neutron hopes that re-execution after some cleanup will avoid the problem. But as Neutron nodes do not have the storage for efficient checkpointing and rollback nor the hardware support to make it efficient (e.g., copy-on-write pages), it instead reboots part of the system.

Failure-oblivious computing [24] dynamically detects violations of a memory safety model; faulting stores are suppressed and data is fabricated to satisfy faulting loads. The resulting system is analogous to what could be a variant of Neutron where all data is precious and there are no integrity checks. The difference is one of goals; failure-oblivious computing improves availability with zero developer overhead, but can give incorrect results. Neutron asks the developer for help, but does not sacrifice correctness in trying to improve availability. Furthermore, failure-oblivious computing mostly addresses user-readable or user input data, such as mail subject headers, URLs, and file names. Because these data are user-centric, mistakes or errors are expected, such that masking the failure often results as a user-level error rather than a crash. In contrast, the data in embedded systems are consumed by less forgiving computer programs: corrupting the last byte of a MAC address has more serious consequences than the last byte of a subject line.

8.3 OS Support for Persistence

Most operating systems use one interface to access RAM and another to access non-volatile storage like disks and tapes. A persistent OS, such as Eumel/L3 [19], EROS [27], Grasshopper [10], or KeyKOS [5], provides a uniform interface to both kinds of storage. Neutron—which provides a uniform interface to reboot-volatile and reboot-persistent storage—is peripherally related to these systems but is much simpler. In particular, Neutron does not have to deal with the problem of providing transparent and high-performance access to slow, stable media.

Like EROS, Neutron attempts to protect the consistency of persistent state. However, unlike a traditional persistent OS, Neutron explicitly does not attempt to make all data persistent. Our view is that minimal persistent state is useful for improving efficiency and availability; beyond that, all state should be volatile with respect to reboots so it can be wiped clean on a safety violation.

Rio Vista [20] provides ultra-high performance transactions by combining a persistent file cache with an associated transaction library. The persistent file cache, like recovery units, protects user application state from kernel crashes. While Neutron and Rio Vista provide a very similar abstraction—persistent memory—their different hardware assumptions cause them to have very different implementations. Rio Vista uses the swap partition to store the buffer cache across reboots and Neutron has a special reboot sequence that does not lose RAM contents. Furthermore, Neutron introduces the concept of precious state, such that data within a faulting recovery unit can persist across a reboot.

9. Discussion

The constraints of low-power embedded devices—very limited RAM and program memory, the static structure of TinyOS—lead Neutron to use a different set of tradeoffs than used in traditional systems. Rather than relying on execution rollback [18], re-execution [23], or a transactional store [6], Neutron uses conservative, compile-time techniques whenever possible. Doing so has two advantages: first, it requires few RAM and ROM resources, and second, it does not introduce any CPU overhead during normal execution. The tradeoff to these advantages is that Neutron is not fully transactional and can lose state. If a memory bug corrupts precious variables or leaves them inconsistent, Neutron must re-initialize them. Precious data can also be lost due to conservative static analysis.

The underlying assumption in this choice of tradeoffs is that memory faults are uncommon. While the systems studied in this paper are very different from those in failure-oblivious computing [24], Rinard's ideas on testing versus deployment apply similarly: developers should not use Neutron during testing, as it masks bugs that could be fixed, but in deployment Neutron can provide efficient recovery from the bugs that testing does not catch.

At present, there are no data or studies on sensornet failures in the field. In most cases, developers and engineers are simply unable to pinpoint the root causes due to the inherent lack of visibility into embedded, event-driven systems that respond to real-world phenomena [34]. For example, while researchers at Harvard were able to identify the downtime on the Reventador volcano network being due to node reboots from the reprogramming system, to this day they are unable to identify the exact cause [36]. As another example, the SPI bug described in Section 7.1 took a TinyOS developer over one month to identify, and doing so required approximately 30 hours of experiments on a controlled testbed with a wired debugging backchannel. The bug was crippling because it lost MAC and topology state, such that CTP's performance was determined more by reboots than algorithms.

Neutron's approach of restarting individual recovery units is a good match for TinyOS's FSM-based interfaces and strongly decoupled components. In monolithic systems with large amounts of state sharing, Neutron would be forced to create very large recovery units, reducing their utility. However, event-driven systems, such as Internet services, which have well-defined boundaries between independent execution units, could be another promising domain. Similarly, component-based operating systems and OS toolkits, such as the OSKit [11], or strongly decoupled microkernels, may also benefit from Neutron's approaches.

10. Conclusion

This paper presents Neutron, a set of extensions to the TinyOS operating system and its toolchain. Neutron enables sensor network software running on tiny microcontrollers to recover from memory safety violations by rebooting "recovery units": subcomputations that are isolated from each other using namespace control and memory-safe execution. The Neutron kernel—a modified version of the TinyOS operating system—is itself a recovery unit and can be rebooted with minimal application disruption. The Neutron compilation toolchain automatically infers boundaries between application recovery units to minimize recovery unit size without compromising safe, separate termination.

Neutron allows applications and kernel services to declare "precious" state that persists across recovery unit reboots. Neutron uses static and dynamic methods to ensure the consistency of precious data before restoring it. In the presence of memory errors, recovery units and precious state can increase availability, reduce energy consumption, and avoid losing application data. In comparison to whole node reboots, Neutron reduces time synchronization downtime by 94% and routing control traffic by 99.5%.

Neutron achieves these goals by leveraging the static nature and finite-state machine behavior of TinyOS programs. Mechanisms that might be too expensive at runtime for a microcontroller, such as tracking precious data structure updates, are approximated using compile time analyses. Similarly, leveraging language extensions and simple, optional annotations makes Neutron easy to incorporate into existing code.

Acknowledgments

This work was supported by Microsoft Research, Intel Research, DoCoMo Capital, Foundation Capital, the National Science Foundation under grants #0615308, #0121778, #0627126, #0846014, and #0448047, as well as a Stanford Terman Fellowship.

References

[1] Atmel, Inc. ATmega128 datasheet, June 2008. `http://www.atmel.com/atmel/acrobat/doc2467.pdf`.

[2] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, pages 333–346, San Diego, CA, Oct. 2000.

[3] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proc. of the 6th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 43–56, Raleigh, NC, Nov. 2008.

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

[5] A. C. Bomberger and N. Hardy. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 95–112, Apr. 1992.

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot—A technique for cheap recovery. In *Proc. of the 6th Symp. on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[7] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proc. of the 16th European Symp. on Programming (ESOP)*, Braga, Portugal, Mar.–Apr. 2007.

[8] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proc. of the 5th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 205–218, Sydney, Australia, Nov. 2007.

[9] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proc. of the Intl. Conf. on Mobile Computing and Networking (MobiCom)*, pages 134–146, San Diego, CA, Sept. 2003.

[10] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.

[11] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 38–51, Saint-Malô, France, Oct. 1997. `http://www.cs.utah.edu/flux/papers/oskit-sosp16.ps.gz`.

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

[13] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection Tree Protocol. In *Proc. of the 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Berkeley, CA, Nov. 2009.

[14] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The TENET architecture for tiered sensor networks. In *Proc. of the 4th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 153–166, Oct. 2006.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Nov. 2000.

[16] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[17] K. Klues, C.-J. M. Liang, J. Paek, R. Musăloiu-E., R. Govindan, P. Levis, and A. Terzis. TOSThreads: Safe and Non-Invasive Preemption in TinyOS. In *Proc. of the 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Berkeley, CA, Nov. 2009.

[18] A. Lenharth, S. T. King, and V. Adve. Recovery domains: An organizing principle for recoverable operating systems. In *Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2009.

[19] J. Liedtke. A persistent system in real use—experiences of the first 13 years. In *Proc. of the 3rd Intl. Workshop on Object Orientation in Operating Systems*, pages 2–11, Dec. 1993.

[20] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP)*, pages 92–101, Saint-Malô, France, Oct. 1997.

[21] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi. The flooding time synchronization protocol. In *Proc. of the 2nd ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 39–49, Baltimore, MD, Nov. 2004.

[22] R. Murty, G. Mainland, I. Rose, A. R. Chowdhury, A. Gosain, J. Bers, and M. Welsh. CitySense: A vision for an urban-scale wireless networking testbed. In *Proc. of the 2008 IEEE Intl. Conf. on Technologies for Homeland Security*, Waltham, MA, May 2008.

[23] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3), Aug. 2007.

[24] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. B. Jr. Enhancing server availability and security through failure-oblivious computing. In *Proc. of the 6th Symp. on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[25] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young, and J. Porter. LUSTER: Wireless sensor network for environmental research. In *Proc. of the 5th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 103–116, Sydney, Australia, Nov. 2007.

[26] Sentilla, Inc. Telos rev. B datasheet, 2007. `http://www.sentilla.com/moteiv-transition.html`.

[27] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, Kiawah Island, SC, Dec. 1999.

[28] SPI driver for ATmega128, version 1.5, 2007. `http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-2.x/tos/chips/atm128/spi/Atm128SpiP.nc`.

[29] Sun Microsystems. Java Platform, Enterprise Edition (Java EE). `http://java.sun.com/javaee/`.

[30] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), Nov. 2006.

[31] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proc. of the 1st European Workshop on Wireless Sensor Networks (EWSN 2004)*, pages 307–322, Berlin, Germany, Jan. 2004.

[32] I. Talzi, A. Hasler, S. Gruber, and C. Tschudin. PermaSense: Investigating permafrost with a WSN in the Swiss Alps. In *Proc. of the 4th Workshop on Embedded Networked Sensors (EmNets 2007)*, Cork, Ireland, June 2007.

[33] E. Troan. The ins and outs of signal processing. *Linux Magazine*, Dec. 1999.

[34] M. Wachs, J. I. Choi, K. Srinivasan, M. Jain, J. W. Lee, Z. Chen, and P. Levis. Visibility: A new metric for protocol design. In *Proc. of the 5th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 73–86, Sydney, Australia, Nov. 2007.

[35] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2006)*, Nov. 2006.

[36] G. Werner-Challen. Private correspondence, 2009.

[37] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

UU IR Author Manuscript