# Poster Abstract: TinyOS 2.1 Adding Threads and Memory Protection to TinyOS

The TinyOS Alliance

(Including all members of the TinyOS 2.x related working groups)

http://www.tinyos.net/scoop/special/working_groups

## 1 Introduction

The release of TinyOS 2.0 two years ago was motivated by the need for greater platform flexibility, improved robustness and reliability, and a move towards service oriented application development. Since this time, we have seen the community embrace these efforts and add support for additional hardware platforms (TinyNode, Iris, Shimmer, BtNode, IntelMote2), and new application level services (CTP[4], Deluge 2.0[3], FTSP[9], ICEM[5], printf, TYMO, DIP[8], DRIP[7], ...). These enhancements are important in the progression of TinyOS as a whole, and have resulted in several minor releases (i.e. TinyOS 2.0.1, 2.0.2).

TinyOS 2.1 is the next stage in the evolution of TinyOS; it takes a step towards addressing the need for easier and more robust application development. TinyOS 2.1 introduces a number of significant enhancements to core TinyOS components and interfaces. The most notable features include a fully preemptable application-level threads library known as *TOSThreads*, and a runtime memory protection service called *Safe TinyOS*. The former aims to ease writing event-driven code while preserving the time-sensitive aspect of TinyOS. The latter aims to make applications more robust through memory safety checks.

## 2 TOSThreads

*Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E., Ramesh Govindan, Andreas Terzis, Philip Levis*

TOSThreads is a complete implementation of a fully preemptive application level threads library for TinyOS. It provides a natural extension to the existing TinyOS concurrency model, requiring only a few minor changes to the TinyOS code base (as documented in [6]). In the existing TinyOS concurrency model two execution contexts exist: synchronous (tasks) and asynchronous (interrupts). These two contexts follow a strict priority scheme: asynchronous code can preempt synchronous code but not vice-versa. TOSThreads extends this concurrency model to provide a third execution context in the form of user-level application threads. Threads synchronize using standard synchronization primitives such as mutexes, semaphores, barriers, and condition variables.

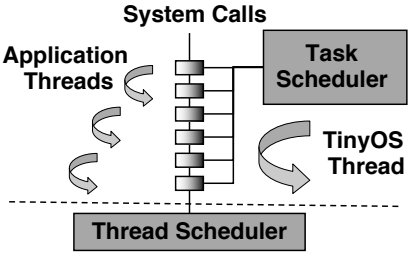


**Figure 1. Overview of the basic TOSThreads architecture. The vertical line separates user-level code on the left from kernel code on the right. Looping arrows indicate running threads, and the blocks in the middle of the figure indicate API slots for making a system call.**

Figure 1 presents the basics of the TOSThreads architecture. Any number of application threads can run concurrently (barring memory constraints), making calls into a single higher priority TinyOS kernel thread through a customizable blocking system call API. Each blocking system call invokes a particular TinyOS service, managing any necessary state across its underlying split-phase implementation. One key feature of TOSThreads is its ability to easily extend this API to include additional TinyOS services. One simply creates a thin shim layer of code (for which many examples already exist) that sits on top of the desired service. Applications can be written against this API in either nesC or standard ANSI-C, enabling developers with no prior knowledge of TinyOS to quickly start writing TinyOS based applications.

Preliminary results show that performing context switches and system calls in TOSThreads introduce a computation overhead of less than 0.92% on representative sensing applications. Furthermore, TOSThreads has been successfully used to reimplement existing sensor network systems such as Tenet, as well as ease the development of

language extensions for sensor network, such as Latte, a JavaScript variant for motes. TOSThreads is also the basis of TinyLD, a dynamic linker and loader for TinyOS. While not ready for inclusion in TinyOS 2.1, TinyLD will soon be used to dynamically deploy and execute TOSThreads based applications at runtime.

## 3 Safe TinyOS

*John Regehr, Eric Eide, Nathan Cooprider, Will Archer, Yang Chen, David Gay*

Forming an actionable hypothesis about why a sensor network is malfunctioning is difficult because many different root causes have the same effect: nodes drop out of the network, reboot, and otherwise fail. One class of root causes—pointer and array bugs, or *memory safety violations*—is particularly difficult because the typical consequence of a safety bug is corrupted RAM.

TinyOS 2.1 applications can be optionally compiled in *safe mode* where the Deputy compiler [1] is used to enforce memory safety at runtime. In safe code, the programmer must provide a few extra annotations describing bounds of arrays and branches of unions. The Deputy compiler then adds a safety check before each potentially unsafe operation; failed checks jump to a fault handler. The source code location of the fault can be found by reading an error code from the node's LEDs and entering it into a tool. Safe TinyOS has helped find previously unknown bugs as well as bugs that were known to exist, but whose location was unknown.

Safe TinyOS permits safe code to be freely mixed with unsafe code using new module-level nesC attributes `@safe` and `@unsafe`, with unsafe being the default. The overheads of safety are evaluated in [2].

## 4 Other Additions

*TinyOS Working Groups*

TinyOS 2.1 has numerous features and additions beyond TOSThreads and memory safety. It adds two new platforms, the IRIS from Crossbow, Inc., and SHIMMER, jointly developed by Harvard University and the Intel Corporation. It supports the Flooding Time Synchronization Protocol (FTSP) on most platforms [9]. The Collection Tree Protocol (CTP) has been updated to use the state-of-the-art 4-bit link estimator [4], resulting in a 35% improvement in efficiency over MultihopLQI. It includes a second dissemination protocol, DIP, which has smaller RAM requiremnts and can scalably manage hundreds of dissemination values [8]. The 802.15.4 frame format has changed to be able to support 6lowpan networking [10] in future releases, and there is an optional 802.15.4-compliant MAC layer implementation. Finally, TinyOS 2.1 includes numerous bug fixes, system improvements, and additional documentation.

## 5 Acknowledgements

TinyOS 2.1 was made possible through members of the TinyOS related working working groups around the globe. The institutions involved in its development in no particular order include Vanderbilt University, Johns Hopkins University, Stanford University, UC Berkeley, UCLA, USC, TU Berlin, Harvard University, University of Szeged, MIT, University of Copenhagen, ETH Zürich, EPFL, University of Utah, Rincon Research Inc., Intel Research, Crossbow Inc., and Arch Rock Co.

We would like to thank everyone from the greater TinyOS community for their valuable feedback on the mailing lists during the development of TinyOS 2.1. Without such feedback, TinyOS would not be what it is today.

## 6 References

[1] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proc. of the16th European Symp. on Programming (ESOP)*, Braga, Portugal, March–April 2007.

[2] Nathan Cooprider, William Archer, Eric Eide, David Gay, and John Regehr. Efficient memory safety for TinyOS. In *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys 2007)*, pages 205–218, Sydney, Australia, November 2007.

[3] Deluge: TinyOS Network Programming. Available at `http://www.cs.berkeley.edu/~jwhui/research/projects/deluge/`.

[4] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, and Philip Levis. Four-Bit Wireless Link Estimation, 2008.

[5] Kevin Klues, Vlado Handziski, Chenyang Lu, Adam Wolisz, David Culler, David Gay, and Phil Levis. Integrating Concurrency Control and Energy Management in Device Drivers. In *Proceedings for The 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[6] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Razvan Musaloiu-E., Ramesh Govindan, Andreas Terzis, and Philip Levis. TEP134: The TOSThreads Thread Library.

[7] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *Proceedings of NSDI 2004*, March 2004.

[8] Kaisen Lin and Philip Levis. Data Discovery and Dissemination with DIP. In *IPSN '08: Proceedings of the 2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 433–444, Washington, DC, USA, 2008. IEEE Computer Society.

[9] M. Marot, B. Kusy, Gy. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys)*, pages 39–49, November 2004.

[10] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan Hui, and David Culler. RFC4944: Transmission of IPv6 Packets over IEEE 802.15.4 Networks, September 2007.