Embedded OSes Must Embrace Distributed Computing

Branden Ghena
brghena@berkeley.edu
University of California, Berkeley

Jean-Luc Watson
jeanluc.watson@berkeley.edu
University of California, Berkeley

Prabal Dutta
prabal@berkeley.edu
University of California, Berkeley

ABSTRACT

Long the case in automotive, aeronautics, and industrial settings, embedded systems in myriad other areas are becoming distributed systems in all but name. Driven by increasingly complex applications with wide-ranging hardware requirements, developers are turning to modular designs that integrate multiple processors onto a single board. Unfortunately, problems already familiar to distributed systems—coordinated execution, portable applications, and unified system-wide administration—have no corresponding embedded support and must be re-implemented in an ad hoc fashion for each application. We argue that future embedded operating system development should explicitly support these multi-microcontroller systems, and can leverage distributed techniques to do so. We also believe that the low-level nature of embedded sensing indicates that OSes should not attempt to completely abstract the presence of separate hardware components and capabilities. Instead, they should provide useful interfaces that support resource-constrained execution and modular application interactions. To that end, we identify existing embedded OS efforts and distributed systems concepts that can inform next-generation OS development.

1 INTRODUCTION

Typical Internet of Things (IoT) devices follow a predictable design. Each has a microcontroller, radio, and sensors and actuators, all wired together on a single circuit board. But even for simple devices, there are many, mostly-independent tasks to be performed: wireless communication, sensing, data processing, control algorithms, and actuation. While these tasks can be, for the most part, developed independently, anyone who has performed system integration knows that simply combining each software component will likely fail; interrupts collide, memory is exhausted, and energy-use estimates are trampled until engineering effort ensures a stable system.

Increasingly, platform designers are solving these integration and resource allocation problems by creating multi-microcontroller devices. Commonly, one microcontroller is dedicated to running a networking stack, while another performs sensing and processing, with physical separation ensuring that software tasks cannot interfere with each other. While this design technique solves several engineering challenges, it also introduces new ones. Embedded systems are becoming distributed systems, facing traditional issues of communication, task placement, and system administration.

Support for these distributed design patterns in current embedded operating systems is limited. Research platforms often eschew an OS altogether in favor of implementing application-specific communication protocols, at the cost of limited flexibility and portability. Commercial cyber-physical systems have seen some efforts to promote interoperability through common interfaces. For example, LonWorks is a platform for managing control networks that communicate and share state with other devices over the network [14]. However, we argue that next generation of embedded operating systems for IoT devices will need to support not only *networked*, but *distributed* applications, for which shared memory, coordinated execution, and reliable communication should be OS services that do not require explicit implementation at the application level.

Embedded operating systems will also need to manage the tasks running on the device. While some tasks are hardware-specific and typically will be co-located with the peripheral with which they are interacting, other functions, such as sensor data processing and control algorithms, are more independent. For these types of tasks, the operating system should enable simple migration between microcontrollers on a platform, even at run time. Similarly, while often only a single microcontroller is connected to the network interface of a device, all microcontrollers on the system need firmware updates. Next generation embedded operating systems should allow code updates to be securely transferred between processors.

Distributed computing has a long history of widespread use, and embedded distributed systems will certainly be able to leverage prior work. However, this new domain has several fundamental, distinctive challenges, such as the heterogeneity of hardware resources and emphasis on low-power operation, which require us to revisit traditional solutions in a new light. Applying these ideas to properly supervise cooperating systems of microcontrollers will be critical to enabling a new generation of complex, but reliable, Internet of Things platforms.

2 MULTI-MICROCONTROLLER PLATFORMS

What is driving the creation of multi-microcontroller platforms? A common reason is to create a physical separation of concerns. In platforms like PowerBlade [6] and SurePoint [12], one microcontroller is devoted to timing-sensitive operations and the other to wireless communications. In Amulet [10] and Flicker [11], one microcontroller solely manages low-power operation. Most of these platforms use software written in bare-metal C without any OS support. Amulet, the exception, provides a runtime system for event-driven applications, but only on the low-power microcontroller.

Alternatively, some platforms use multiple microcontrollers to enable modularity. Burnout [15] has ten wearable accelerometers, each including a separate microcontroller tasked with measuring muscle fatigue. All report measurements to a central microcontroller in charge of logging and wireless communication. Signpost [1], a city-scale sensing platform, expects sensor modules to contain their own microcontroller capable of taking measurements. It uses a custom multi-master I^2C messaging scheme between the microcontrollers as its OS does not provide such a mechanism. The rise of these multi-microcontroller research platforms without corresponding software support for their design demonstrates an unsatisfied need that the operating systems community can fulfill.

NGOSCPS 2019, April 15, 2019, Montreal, Canada

3 OPERATING SYSTEM REQUIREMENTS

Embedded operating systems for multi-microcontroller platforms have many of the same requirements as any other embedded OS. They need to have low memory and processing overhead to accommodate resource-constrained systems, and they need to encourage low-power operation for battery-powered devices. For many cyber-physical systems, the OS must also support real-time, deadline-driven operation. However, distributed embedded systems exhibit several additional domain-specific requirements.

Inter-Microcontroller Communication. A primary enabler for multi-microcontroller systems is communication. Operations running on two separate microcontrollers need to send commands and share results with each other. On a single processor, communication is as simple as calling a function and passing a data buffer, and we advocate that OSes provide abstractions that emulate this simplicity. The introduction of memory ownership concepts in modern systems programming languages like Rust can make this communication simpler, transferring control of a slice of memory rather than implementing more complex shared memory techniques.

Inter-chip communication will introduce additional latency into task coordination. While reducing latency is good, more important is that connections are deterministic so that developers can have an expectation of reliable behavior and long-term system stability. A related requirement necessary for many systems is a priority mechanism. Platform designers should be able to configure whether an incoming message or the task at hand is more important, and take steps to reduce interfering bus contention as necessary. Handling of communication failures, although rare in a single circuit board context, is also necessary for long-term system robustness.

An example of inter-microcontroller communication in practice is CoMOS [9], a multi-microcontroller OS developed for a sound source localization device. CoMOS provides an explicit `send_msg()` function and handles the routing of messages between chips. We argue that communication should instead be more implicit. In our view, communication between two loosely-coupled application components, even across multiple processors, should be no different to implement than for communication within a single microprocessor, with the OS managing interaction behind the scenes. Making inter-microcontroller communication seamless may also require toolchain support during the software development and compilation stage in addition to runtime support by the OS.

Task Migration. Embedded OSes should additionally provide an abstraction for software portability, so that application code can be re-targeted to any available, appropriate microprocessor in the system. Even during the design process, changing system requirements can shift application tasks between microcontrollers to reduce workload or meet energy consumption constraints. Developing software components that can be easily transferred when necessary avoids expending unnecessary additional engineering effort. However, we caution that many tasks are so closely tied to the physical platform they execute on that making them agnostic to the presence of a multi-microcontroller system is likely to yield unintuitive results. For example, a task that samples a sensor it assumes is connected directly to "its" processor may receive data with a vastly longer delay than expected because components elsewhere on the device require immediate higher-priority service. Thus, tasks must be aware of module boundaries at a logical level. Tock [13], a multi-processing OS for embedded systems, demonstrates one possible mechanism of software independence. Processes in Tock interact with a microcontroller-specific kernel through a restricted set of system calls, allowing the same application code to run on any platform as long as the hardware resources it needs are available.

Runtime migration of tasks between microcontrollers is also sometimes necessary for optimal efficiency. Some tasks, such as control algorithms or data processing, are inherently movable, and when a platform's workload changes, optimal placement for applications can shift. CoMOS describes an example of this, in which an FFT can be executed in a more energy efficient manner on either an MSP430 or an ARM7 core depending on the window size [8].

Platform Management. Finally, operating systems should support management of microcontrollers anywhere on a platform, regardless of the physical interconnect. Frequently, only one microcontroller actually interfaces with a network, but firmware updates are needed for all processors on a system. Future embedded OSes should enable authenticated, reliable firmware updates over inter-processor communication channels just as they should over the network, ensuring that updated components retain full compatibility with the remainder of the system. Other management aspects, such as the ability to watchdog and reset other microcontrollers in the system are also important for increasing platform robustness.

4 LESSONS FROM CLASSICAL SYSTEMS

Multi-microcontroller systems differ dramatically from the general-purpose distributed computing systems of traditional computer science research. The primary concern is not that of providing a generic execution environment and memory access abstraction [4, 16], but of facilitating interaction between an array of heterogeneous microcontrollers. In particular, distributed systems operate at scale by coordinating thousands of servers to, for example, maintain global consistency, efficiently distribute user data, or provide low latency, failure resistant web services. Multi-microcontroller designs do not face the same scaling challenges given the localized nature of embedded sensing. Likewise, faced with unreliable networks and frequently changing topologies, previous systems have required significant fault tolerance [20]; embedded microcontrollers are more stable in operation due to their static, well-defined hardware layout. Nevertheless, multi-microcontroller sensors remain, fundamentally, distributed systems and face many of the same challenges.

Message-Passing Interfaces. Distributed operating systems have long used message-based protocols to coordinate system behavior. Decades ago, both the LOCUS [20] and Amoeba [16] OSes served requests to a global file system with simple remote procedure calls (RPCs), transparently managing serialization and addressing. The same has been applied to work on a heterogeneous multicore OS where RPCs provided an efficient alternative to shared-memory communication due to their compact representation [2], and in NUMA systems where cores hosted independent kernel instances [17]. Since the physical nature of embedded devices precludes shared memory access, a well-designed messaging interface is critical for an embedded OS to build more powerful abstractions.

However, the embedded applications for which these systems are designed present unique challenges. Complex sensing (e.g. lo-

calization [12]) often executes in tight control loops and is very sensitive to messaging delays [18]. Thus, while language—and operating system—independent interoperation mechanisms such as CORBA [19] or Protocol Buffers [7] are useful tools to structure communication, the OS must effectively manage them to expedite critical communication and identify worst-case messaging overhead. Further, in low-power systems, the energy cost of frequent inter-microcontroller communication is a consideration that may determine where software tasks are placed. Exposing communication primitives (e.g. Protocol Buffers) directly to the application artificially freezes the application structure and reduces the ability of the OS to dynamically pair tasks locally if possible.

Transparent Abstractions. Presenting a distributed collection of computing nodes as a single machine is a core function of general-purpose distributed operating systems. This provides a familiar abstraction to user applications: computation and memory accesses occurring in parallel across the system retain the semantics of single-server execution. It is commonly achieved by transparently intercepting local system calls to perform an RPC [20], invoke distributed consistency protocols [2], or page in remote memory [5]. Similarly, an embedded operating system will need to provide abstractions that ease application development across microcontroller boundaries, but it is not clear that traditional mechanisms can be directly applied. Sensing applications cannot realistically execute in a completely transparent environment, as they may depend on specific low-level hardware. Instead, we argue that a distributed OS should primarily implement *opaque* abstractions: interfaces that easily bridge between logical components of an application without hiding the distributed nature of the microcontrollers. For example, a humidity sensor that wishes to export a data buffer through a Bluetooth radio driver need not concern itself with the location of the radio (be it on the same chip or external). Instead, a call to the radio driver would transfer ownership of the buffer, with the OS transparently triggering over-the-wire communication if necessary.

Load Distribution. Efficiently managing the placement of application tasks across physical nodes is a recurring theme. In considering execution on multiprocessor chips, Bertozzi et al. [3] require application developers to define explicit process migration points within the application such that it could be safely moved if necessary to avoid overloading a processor or to pair it with a sibling task. The Cilk-NOW runtime is an example of a global-level scheduler that detects idle processors and dynamically steals upcoming tasks that have not yet been executed to increase application parallelism [4], and Baumann et al. identify that systems may contain fundamentally different processors (e.g. GPUs or FPGAs) that should be assigned tasks for which they are best suited [2]. In an embedded setting, the ability to flexibly direct where execution occurs will be limited due to the physical design, power and timing constraints, and sensor availability, but leveraging the same task migration techniques could provide useful capabilities, for example, to batch timing insensitive operations on a processor that offers a higher startup cost but lower energy per cycle.

5 CONCLUSION

Multi-microcontroller systems are becoming more popular as an embedded design pattern that provides strong separation of concerns. Building modular devices that isolate tasks, operate at low power, and satisfy real-time requirements is critical to enabling complex IoT sensing applications. In this paper, we assert that embedded operating systems need to provide support for these use cases by leveraging distributed system concepts. Specifically, coordinated execution, processor-agnostic task migration, and system-wide management capabilities are primitives that will substantially simplify embedded software development. Relevant work in distributed systems is encouraging in that it suggests a modular, distributed operating system is possible, having demonstrated the utility of message-based communication, abstractions over hardware boundaries, and dynamically-placed execution. While care must be taken to ensure that introducing distributed systems techniques to the embedded domain does not compromise existing guarantees of real-time and low-power operation, doing so has the potential of reducing the development burden for future embedded systems and enabling the creation of robust, capable platforms.

REFERENCES

[1] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. 2018. The signpost platform for city-scale sensing *(IPSN'18)*.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems *(SOSP'09)*.

[3] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. 2006. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the conference on Design, automation and test in Europe*.

[4] Robert D Blumofe, Philip A Lisiecki, et al. 1997. Adaptive and reliable parallel computing on networks of workstations. In *USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*.

[5] David Cheriton. 1988. The V distributed system. *Commun. ACM* (1988).

[6] Samuel DeBruin, Branden Ghena, Ye-Sheng Kuo, and Prabal Dutta. 2015. Powerblade: A low-profile, true-power, plug-through energy meter *(SenSys'15)*.

[7] Google. 2008. Protocol Buffers | Google Developers. . Accessed: 2019-02-18.

[8] Michel Goraczko, Jie Liu, Dimitrios Lymberopoulos, Slobodan Matic, Bodhi Priyantha, and Feng Zhao. 2008. Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems *(DAC 2008)*.

[9] Chih-Chieh Han, Michel Goraczko, Johannes Helander, Jie Liu, Bodhi Priyantha, and Feng Zhao. 2006. *CoMOS: An operating system for heterogeneous multiprocessor sensor devices.* Technical Report. Microsoft Research.

[10] Josiah Hester, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearndon, Kevin Freeman, Sarah Lord, et al. 2016. Amulet: An energy-efficient, multi-application wearable platform *(SenSys'16)*.

[11] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things *(SenSys'17)*.

[12] Benjamin Kempke, Pat Pannuto, Bradford Campbell, and Prabal Dutta. 2016. SurePoint: Exploiting ultra wideband flooding and diversity to provide robust, scalable, high-fidelity indoor localization *(SenSys'16)*.

[13] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently *(SOSP'17)*.

[14] Dietmar Loy, Dietmar Dietrich, and Hans-Joerg Schweinzer. 2012. *Open control networks: LonWorks/EIA 709 technology*.

[15] Frank Mokaya, Roland Lucas, Hae Young Noh, and Pei Zhang. 2016. Burnout: a wearable system for unobtrusive skeletal muscle fatigue estimation *(IPSN'16)*.

[16] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. 1990. Amoeba: A distributed operating system for the 1990s. *Computer* (1990).

[17] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: heterogeneous multiprocessing with satellite kernels *(SOSP'09)*.

[18] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. 1990. Real-Time Mach: Towards a Predictable Real-Time System.. In *USENIX Mach Symposium*.

[19] Steve Vinoski et al. 1997. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications magazine* (1997).

[20] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. 1983. The LOCUS distributed operating system. In *ACM SIGOPS Operating Systems Review*.