Journal of Computer Science 4 (4): 272-281, 2008
ISSN 1549-3636

Operating System Support for Wireless Sensor Networks

Antônio Augusto Fröhlich and Lucas Francisco Wanner
Laboratory for Software and Hardware Integration,
Federal University of Santa Catarina, 88049-900, Florianópolis, SC, Brazil

Abstract: In a wireless sensor network, several sensor nodes obtain local data and communicate among themselves in order to create a global vision of an object of study. The idea of a self-managed network of low-power, autonomous devices, that collects data from an environment and propagates information through a wireless link brought about several new challenges and requirements in application run-time support. Several research projects have aimed at solving the problem of system support for sensor networks. However, most of them have failed in dealing with two requirements: transparent configuration of the data communication channel and efficient and unified sensor hardware abstraction. In this work we designed and implemented a run-time support environment for wireless sensor network applications based on the EPOS operating system. Through this environment, applications were allowed to configure the communication channel according to their needs and to acquire sensor data through a family-based, uniform, sensor data acquisition API. Our tests showed that the introduction this environment did not incur in excessive overhead and presented significant advantages in relation to the solutions found in other operating systems for sensor networks.

Key words: Embedded operating systems, sensor abstraction, configurable communication systems

INTRODUCTION

Recent advances in hardware design and miniaturization have enabled the emergence of a new set of applications to the fundamental concept of computer, in the form of low-power, wireless micro-sensors. These micro-sensors are equipped with analog or digital sensor devices (e.g., temperature, magnetic, acoustic sensor), a digital processor, a wireless communication module (e.g., low-power radio) and a power module (e.g., battery, solar cell). Each individual sensor is able to obtain a local vision of its environment and to coordinate and communicate with other sensors in order to create a global vision of a given object of study.

The idea of a self-managed network of autonomous devices, which collect and forward data through a wireless link, brings about a series of new challenges to hardware design. In order to be unobtrusive and to operate autonomously for long periods of time, the sensor nodes must be small and low-power. To allow a wide range of applications to share a common platform, the nodes must be modular and allow different sensing devices to be used according to the needs of specific applications. Similarly, the communication hardware should allow wide configuration of the data channel, so that different applications may benefit from different modulation and medium access control schemes. As the complexity of wireless sensor network technologies increases, the need for runtime support to mediate hardware capabilities and application needs becomes critical.

System requirements for sensor networks include basic operating system functionality, power management, field reprogramming mechanisms, sensing hardware abstraction and a configurable communication stack. Restricted hardware capabilities require these systems to operate with limited resources and make the use and adaptation of traditional operating systems impossible. Several research projects[1-3,5,7,9] have aimed at solving the problem of system support for sensor networks. However, most of them have failed in dealing with two requirements: transparent configuration of the data communication channel and efficient and unified sensor hardware abstraction.

The EPOS system[4,11] is a component-based framework for the generation of dedicated runtime support environments. The EPOS system framework allows programmers to develop platform-independent applications and analysis tools allow components to be automatically adapted to fulfill the requirements of these particular applications. By definition, one instance

Corresponding Author: Antônio Augusto Fröhlich, Laboratory for Software and Hardware Integration, Federal University of Santa Catarina, 88049-900, Florianópolis, SC, Brazil

of the system aggregates all the necessary support for its dedicated application and nothing else. EPOS provides a wide set of operating system services through platform-independent interfaces and supports a wide range of platforms, such as IA32, PowerPC, Sparc, MIPS, H8 and AVR

This study shows the design and implementation of a run-time support environment for wireless sensor network applications based on the EPOS operating system. This environment includes platform support, power management services, configurable communication through the C-MAC (Configurable MAC) medium access control protocol, which allows applications to configure the communication channel according to their needs and a sensor data acquisition system, which abstracts families of sensing devices in an uniform fashion, without incurring excessive overhead and presenting significant advantages in relation to the solutions found in other operating systems for sensor networks.

OPERATING SYSTEMS FOR SENSOR NETWORKS

In a sensor network, application-specific requirements drive the entire hardware design, from processing capabilities to radio bandwidth and sensor modules, thus requiring the hardware to be modular. However, these requirements have led to a huge variety of hardware components, making wireless sensor networks hardware not only modular, but also heterogeneous. In this scenario, a sensor application developed for a given platform will seldom be portable to a different one, unless the run-time support systems on those platforms deliver mechanisms that abstract and encapsulate the sensor platform in an adequate manner. At the same time, the limited resources typically found in sensor networks hardware require any runtime support for these systems to be efficient and not to use excessive resources.

The need for connectivity, hardware abstraction and management of limited resources makes operating system support imperative for sensor network applications. Considering current research, technology and applications[7], we may list a series of operating system requirements for wireless sensor networks. Such a system should:

Provide basic operating system functionality: In order not to restrict the functionality and portability of sensor networks applications, an operating system for such devices should provide traditional operating system services such as: hardware abstraction, process management (usually following the mono-task, multi-thread prism), timing services and memory management.

Provide efficient power management mechanisms: Efficient power management in the sensor nodes is a determining factor for the network's life time. A runtime support system for sensor networks applications should provide power management mechanisms to the applications, as well as use as little power as possible to provide its services.

Provide field reprogramming mechanisms: Given that the sensor nodes may be located in inhospitable regions and that application requirements and parameters may change with time, field reprogramming through the communication network is an important service in this type of networks. An operating system for sensor networks should ideally provide total or partial field reprogramming mechanisms for deployed applications.

Abstract heterogeneous sensing hardware in a uniform fashion: The application-specific requirements of sensor networks make its hardware not only modular, but also heterogeneous. In this scenario, a sensor application developed for a given platform will seldom be portable to a different one, unless the run-time support systems on those platforms deliver mechanisms that abstract and encapsulate the sensor platform in an adequate manner. Architectural differences aside, sensor modules (e.g., temperature, light and motion sensors) present an even wider range of variability. Sensor modules presenting the same functionality often vary in their access interface, operational characteristics and parameters. A properly designed run-time support system could free application programmers from such architectural dependencies and promote application portability among different sensing platforms.

Provide a configurable communication stack: Given the specific communication requirements of different applications, communication hardware for sensor networks should be widely configurable. The operating system should provide means to configure the communication protocol stack, starting from the medium access control protocols.

Operate with limited resources: As sensor nodes must be low power, their hardware design will tradeoff computation capabilities for lower power consumption. As such, the nodes will have limited processing power

and memory resources. An operating system for sensor networks should deliver the required application services without using a significant amount of the computational resources available to the nodes.

Typical embedded operating systems, such as VxWorks, QNX, OS-9, WinCE and μClinux provide a programming environment similar to those existing in traditional computers, usually through POSIX-compliant services. Many of these operating systems provide and thus require hardware support to, memory protection. Although these systems are adequate for mobile phones, set-top-boxes and other complex embedded applications, their memory and processing requirements makes their use in wireless sensor networks impossible. Several systems have been developed specifically for these networks, including MagnetOS[2], Contiki[3] and AmbientRT[9]. However, the most prominent of these systems are the TinyOS[7], MANTIS OS[1] and SOS[5] systems.

TinyOS is an event-based operating system for sensor networks[7]. The system is organized as a collection of components. Each TinyOS configuration is composed by and application and its required operating system services and consists in a scheduler and a component graph. Each component is composed by commands, event handlers, tasks and an execution frame. Each component declares the commands to which it responds and the events it signals. Commands are non-blocking method calls and are typically used to initiate software and hardware requests and, conditionally, initiate tasks. Event handlers are used to handle hardware interrupts and may call commands or post tasks.

The system provides a simplified concurrence model, based in run-to-completion tasks, which may only be preempted by interrupts. This model brings about negative and positive consequences. In a traditional thread-based model, where each thread has its own stack, each thread must reserve space in the node's limited memory for its execution context. Depending on the architecture, context switching may be a lengthy operation. By restricting this model, TinyOS reduces most of this overhead, but also loses most of the characteristics of a traditional multithread model. This restriction of concurrence may also hinder the system's ability to deal with real-time metrics. TinyOS does not provide dynamic memory allocation mechanisms. Timing services are provided by a Timer interface. The component model of TinyOS, along with its simplified concurrence model, allows the system to run in platforms with less than 1KB of RAM.

Power management in TinyOS is implemented by the task scheduler, which makes used of the StdControl interface to start and stop components. When the scheduler queue is empty, the main processor is put in sleep mode. This way, new tasks will only be posted in the execution of an interrupt handler. This method yields good results for the main microcontroller, but leaves more aggressive methods (including starting and stopping peripheral components) to the application.

TinyOS features a three-tiered hardware abstraction architecture, comprised by a Hardware Presentation Layer, a Hardware Adaptation Layer and a Hardware Interface Layer[6]. The Hardware Presentation Layer is placed directly over the underlying hardware and presents the hardware to the operating system. Components in the Hardware Presentation Layer are unique for each device they present, but may share a common structure. The Hardware Adaptation Layer groups the hardware-specific components into domain-specific models, such as Alarm or ADC Channel. The Hardware Adaptation Layer provides the 'best' possible abstraction in terms of effective resource usage, but also tries not to hinder application portability. The Hardware Interface Layer uses the adapted components to implement platform-independent abstractions. The TinyOS application developer may choose to use any of the available interface levels, trading off application portability and efficient resource usage.

The TinyOS communication stack is based on the B-MAC medium access control protocol[12]. The protocol is implemented in layers (low-level hardware control and protocol logic). The low-level control layer allows static and dynamic configuration of basic communication parameters (e.g., frequency, transmission power). The system also allows some level of configuration of protocol logic (duty cycle, free channel detection algorithm, use of acknowledgements).

MANTIS OS (Multimodal networks of *in situ* sensors)[1] is a multithread operating system for sensor networks, with an application programming interface inspired by POSIX adapted to the needs and restrictions of wireless sensor networks. The architecture of MANTIS is based on the classical layered multithreaded design. The system's application programming interface is preserved between different platforms. The system kernel is comprised of a scheduler and device drivers. A communication stack and a command server are provided as user-level services.

The MANTIS scheduler provides a subset of the POSIX thread package, with priority-based round-robin scheduling. The system supports static and dynamic heap allocation for threads. The scheduler is called

periodically according to a timer, or through semaphore operations. An idle thread is used as entry point for the system's power management policies, which put the processor in sleep mode whenever there are no threads waiting for the processor. Timing and synchronization services are provided through POSIX-like interfaces. The complex scheduling mechanism used in MANTIS incurs in greater overhead than that of a simpler, event-based model. Thus, the system has a larger footprint than, for example, TinyOS. However, the system is still adequate for use in current sensor network prototypes.

MANTIS uses a monolithic hardware abstraction layer, with dev read(), dev write(), dev mode() and dev ioctl() functions. Each function takes a device as a parameter and a function table redirects general calls to specific device drivers. Parameters for the dev mode() and dev ioctl() are device-specific and there is no unified abstraction for sensing hardware (each device driver has specific semantics).

The system provides a unified communication interface through user-level threads. There is a unified packet format for different communication interfaces (e.g., serial interfaces, USB, radio). This communication layer manages packet synchronization and buffering. Underneath this communication API, MANTIS uses traditional device drivers. The monolithic nature of the MANTIS system may incur in unnecessary overhead. On the other hand, the apparent advantages of a single communication entry-point are diminished due to the specific semantics and parameters of the communication methods for each interface.

SOS[5] is a dynamically reconfigurable operating system for sensor networks. The system's kernel includes message passing services, dynamic memory allocation and dynamic module loading. SOS is organized as a series of binary modules that implement specific tasks. These components are comparable in functionality to TinyOS components. An application is comprised by a series of interacting modules, which present both a method call interface and a message passing interface. Message passing is asynchronous and coordinated by a scheduler that uses a priority-ordered queue. Direct function calls are used for synchronous operations between modules. Module loading and distribution are implemented by kernel-independent distribution protocols and meta-description structures. The system integrates dynamic memory allocation and garbage collection. As in the TinyOS and MANTIS systems, SOS puts the processor in sleep mode whenever there are no messages to schedule. The dynamic reconfiguration model of SOS incurs in considerably higher overhead than its static counterparts. However, this overhead is still acceptable for most sensor network applications[5].

SOS provides services for dynamically including, updating and removing modules to previously deployed sensing programs. The system divides program memory in pages and keeps state and context structures in RAM for each module.

The system uses the loadable kernel modules mechanisms for sensing hardware abstractions. Through this architecture device drivers can register their services and associate it to a name, allowing applications to access components through these names. For instance, an analog sensor driver can bind itself to an ADC Channel and register a sensor type as PHOTO. When the application requests data from PHOTO, the kernel uses the registered driver to obtain the appropriate ADC reading. This semantic abstraction of sensor readings promotes application portability. However, since the operating system has to keep a table of function pointers indexed by name, the registering of drivers incurs in some memory overhead.

THE EPOS SYSTEM

EPOS (Embedded Parallel Operating System)[4,11] is a component-based framework for the generation of dedicated runtime support environments. The EPOS system framework allows programmers to develop platform-independent applications and analysis tools allow components to be automatically adapted to fulfill the requirements of these particular applications. By definition, one instance of the system aggregates all the necessary support for its dedicated application and nothing else.

The modular design of EPOS was guided by the Application-Oriented System Design (AOSD) methodology. AOSD elaborates on the well-known domain decomposition strategies behind Family-Based Design (FBD) and Object-Orientation (OO), i.e., commonality and variability analysis, to add the concept of aspect identification and separation yet at the early stages of design[4]. In this way, AOSD guides domain engineering towards families of components, of which execution scenario dependencies are factored out as ‘aspects’ and external relationships are captured in a component framework. This domain engineering strategy consistently addresses some of the most relevant issues in component-based software development:

Reusability: Components tend to be highly reusable, for they are modeled as abstractions of real elements of a given domain and not as parts of a target system.

Moreover, by factoring out execution scenario dependencies as aspects, components can be reused unmodified in a variety of scenarios simply by defining new aspect programs.

Complexity management: The identification and separation of execution scenario dependencies implicitly reduces the number of components in each family, since those components that would have been modeled to express a variation in the domain that originates from a scenario dependency are suppressed whenever the dependency can be modeled as an aspect. Simply stated, a set of 100 components could be modeled as a set of 10 components plus a set of 10 aspects and a mechanism to apply aspects to components. The overall complexity (and functionality) in the new set of 100 generated components is the same, but it is now confined in fewer constructs. This directly improves maintainability.

Composability: By capturing component relationships in a component framework, AOSD enables components to be more easily combined while generating a system instance. It also put some limits to the misbehaviors that can arise from applying aspect programs to pre-validated components. Feature-based models are of great value at this point to capture configuration knowledge and thus make system generation a more predictable procedure.

Figure 1 shows the application-oriented system design domain decomposition process. Abstractions are identified from the problem domain and arranged in families according to their common characteristics. Scenario dependencies are modeled as aspects that may be applied through scenario adapters. Families of abstractions are visible to applications through inflated interfaces, which export their members as a single super-component. System architectures are captured in component frameworks, which are defined in terms of scenario aspects.

Families of abstractions in EPOS represent traditional operating system abstractions and implement services such as memory and process management, process coordination, timing and communication. Abstractions are designed and implemented independently from execution scenarios and architectures. All architecture-dependent hardware units are abstracted as hardware mediators which export, though their platform independent interfaces, the functionality demanded by abstractions. Due to the use of static meta-programming and function inlining, hardware mediators implement their functionality without forming a conventional hardware abstraction

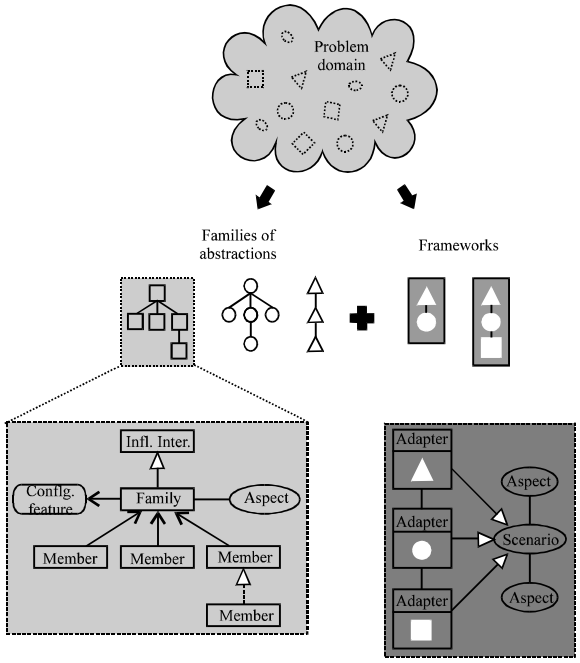

Fig. 1: AOSD domain decomposition process

layer. Through the use of hardware mediators, EPOS 's abstractions have reached a level of reusability that allows, for example, the same family of Thread abstractions to be used in a mono-task or multitask environment, as part of a µkernel or completely embedded in the application, in an 8-bit microcontroller or a 64-bit processor.

Processes in EPOS are managed by the Thread and Task abstractions. Each Thread stores its context in its own stack. The Context abstraction defines all data that must be stored for an execution flow and this way, each architecture defines its own context.

Time is handled by the Timepiece family of abstractions. These abstractions are supported through the Timer, Timestamp Counter (TSC) and Real-Time Clock (RTC) mediators. The Clock abstraction is responsible for keeping track of current time and is available only on systems that feature a real-time clock device. The Alarm abstraction can be used to generate events that can wake-up a thread or call a function. Alarms also have a master event with high priority that is associated with a certain period of time. This master event is used to call the process scheduling algorithm at each quantum of time, when the active scheduler feature is configured on the system. Finally, the Chronometer abstraction is used to perform time measurements.

The Synchronizer family of abstractions provides mechanisms to ensure data consistency in a concurrent process environment. The Mutex member implements a simple mutual exclusion mechanism that supplies two atomic operations: lock and unlock. The Semaphore member realizes a semaphore variable, that is a integer variable whose value can only be manipulated indirectly through the atomic operations p and v. The Condition member realizes a system abstraction inspired on the condition variable language concept, which allows a thread to wait for a predicate to become true.

In EPOS, details pertaining to address space protection and translation, as well as memory allocation, are abstracted through the MMU (Memory Management Unit) family of mediators. The Address Space abstraction is a container for chunks of physical memory called segments. It does not implement any protection, translation or allocation duties, handing then over to the MMU mediator. The Flat Address Space defines a memory model in which logical and physical addresses match, thus eliminating the need for MMU hardware. In platforms that do not feature a MMU, the MMU mediator simply mantains the interface contract with the Flat abstraction, providing empty method implementations whenever necessary. Methods concerning memory allocation operate on bytes in a way that is similar to libc's malloc function.

Input/Output control for peripheral devices in EPOS is provided by the hardware's corresponding mediator. The Machine mediator stores I/O locations and handles dynamic interrupt registering. The IC (Interrupt Controller) mediator handles enabling and disabling individual interrupts. In order to deal with different interrupts available in different platforms and contexts, EPOS assigns platform-independent name and syntax to interrupts pertinent to the system (e.g., timer interrupt).

RESULTS AND DISCUSSION

Wireless sensor network applications present specific requirements in addition to traditional operating system services. These include efficient power management, field reprogramming, uniform abstraction of heterogeneous sensor devices and configurable communication services. In this work, we introduced extensions to the EPOS operating system in order to fulfill these requirements.

EPOS provides application-driven power management services that allow power aware operation of deeply embedded systems, without compromising application portability and without incurring excessive overhead. The goal of our power management system is to allow applications to express when certain software components are not being used, permitting the system to migrate hardware resources associated with these components to lower power levels. Several issues regarding architectural differences between different hardware devices and concurrent access of hardware resources by different software components emerged from this goal. In order to deal with these issues, our system was built upon a generic power management interface, a message propagation system and on the formalization of changes in operating modes[8].

In our power management strategy, the application programmer is expected to specify in his source code, whenever certain components will not be used. Thus, a uniform API to allow power management was defined. This interface allows interaction between the application and the system, between system components and hardware devices and directly between application and hardware. In order to free the application programmer from having to wake up components whenever they are needed, the power managing mechanism abstracted by this interface ensures that components return to their previous operational states whenever they are used.

The application may, for example, access a global component (System) that has knowledge of every other component in the system, triggering a system-wide power mode change. Another way the application may use this interface is through subsystems (e.g., Inter-Process Communication (IPC), Processing and Sensing). In this way, messages are propagated only to the components used in the implementation of each subsystem. The application may also access the hardware directly, using the API available in the device drivers, such as Network Interface Card (NIC), CPU, Thermistor. The same API is also used between the system's components.

In order to attain application portability and to facilitate application development, the power managing interface was defined with a minimal set of methods and universal operating modes with unified semantics throughout the system. Portability comes from the fact that the application does not need to implement specific procedures for each device in order to change its operating mode. These procedures are abstracted by the API. Easiness of use comes from the fact that the application programmer does not need to analyze specific hardware manuals in order to identify available operating modes, the procedures to change those modes and the consequences of these changes.

In order to map coherent connectivity between different abstraction levels in the system, a formal

operating mode migration net was defined. In this study, we describe this formal mechanism, which was defined through Petri nets. These nets feature clear graphical representation and a wide range of mathematical analysis models. These models allow proof of liveness and reachability of desirable states, as well unreachability of incorrect states. Although the procedures to migrate power states are specific to each component (both software and hardware), the control and dispatch of these migrations may be expressed in a generic form. In order to allow that, a network of mode migrations, that specifies the transitions between different operating modes was formalized.

By using the hierarchical architecture by which system components are organized in EPOS, effective power management was achieved for deeply embedded systems without the need for costly techniques or strategies, thus incurring in no unnecessary processing or memory overheads. Case studies[8] have shown significant power savings, with minimal application intervention. This hierarchical power management infra-structure is also used by an active, opportunistic power manager, which is executed either periodically or when there are no tasks to schedule. This power manager checks the utilization timestamps of each registered component against the current timestamp of the system. A configurable power management heuristic then decides if and when to change a component's power mode. In its simplest form, the power manager puts all idle components (components that have not been accessed for a pre-set period of time) into sleep mode.

In order to allow field reprogramming, EPOS makes use of an indirection mechanism similar to Remote Procedure Calls. In this infrastructure, the invocation of a component's method of the client application passes through a Proxy that sends a message to an Agent. After the method execution, a message with the return value is sent back to the application. With this structure, an indirection level is created among the application method calls, making the Agent the only entity aware of component's position in the system memory. The Agent controls the access to the component's method through a synchronizer (Semaphore), not allowing calls to a component that is currently being updated. A system thread is responsible for receiving an update request and the new component code. This request is sent to agent, which overwrites the old code by the new one. The framework infrastructure for system update is transparent to the application and may be 'turned off' without overhead. However, when update support enabled in the system, system footprint increases and the component method calls suffer a small delay.

In order to provide sensing support for applications, EPOS relies on software/hardware interface that is able to abstract families of sensing devices in a uniform fashion[14]. We define classes of sensing devices based on their finality (e.g., sensing acceleration, sensing temperature) and establish a common substrate for each class. Each individual device in a class is able to describe itself and its properties, in a similar fashion to the IEEE 1451 standard sensors transducer electronic data sheet. A thin software layer adapts individual devices (e.g., converts ADC readings into contextualized values, performs calibration) to fit the minimal requirements of its sensor class. Thus, a simple thermistor is exported to an application in the exact same fashion as a complex digital temperature sensor. Software-based self-description allows applications to use individual sensors' extended characteristics. Thus, an application may use a Thermometer abstraction, without having to address a particular temperature sensor.

In the EPOS sensing subsystem, common methods for all sensing devices are defined by the Sensor Common interface. The get() method provides a single sensor, single channel reading (i.e., enables the device, waits for data to be ready, reads the sensor, disables the device and returns readings converted into pre-determined physical units). The enable(), disable(), data ready() and get raw() methods allow the operating system and applications to perform fine-grain control over sensor readings (e.g., performing sequential readings, obtaining raw sensor values). The convert(int v) method may be used to convert raw sensor readings (e.g., ADC or duty-cycle outputs) into scientific or engineering units. The calibrate() method performs a device and platform specific calibration method, which may require user interaction, depending on the sensor.

Each sensor family may extend the Sensor Common interface in order to properly abstract specific family characteristics. The Magnetometer family may add, for example, method for sampling and reading different axes. A Thermistor family, on the other hand, will probably not need to extend the basic common interface. Each family also defines a specific Descriptor structure, which defines specific fields for operation, accuracy, timing, calibration data and physical units. Every sensing device implements one of the defined interfaces and may provide specific methods for calibration, configuration and operation. Furthermore, each sensing device fills a family-specific Descriptor structure with device-specific values. Default configuration parameters (e.g., frequency, gain, etc.) for each device are stored in a configuration traits structure.

Table 1: Sensing components footprint

	Footprint (bytes)					
	TinyOS		Mantis		EPOS	
Sensor	Code	Data	Code	Data	Code	Data
System	10188	455	25500	596	7046	213
AVR ADC	550	4	538	9	64	3
ADXL202	722	4	936	10	266	9
Thermistor	1366	12	1050	11	1064	3
Photocell	1366	12	1050	11	1064	3
HMC1002	748	7	910	10	246	9

Table 2: Maximum sampling rate

	Sampling rate (Hz)		
Sensor	TinyOS	Mantis	EPOS
AVR ADC	8084	3685	24597
ADXL202	7657	3401	21711
Thermistor	5766	3107	10999
Photocell	6009	3117	11121
HMC1002	7494	3408	23024

Whenever the operating system or an application need to refer to a sensing device, they may either refer to the specific device (e.g., MicaSB Temperature) and perform device-specific operations, or refer to the device class (e.g., Temperature Sensor) and restrict to operations defined by that class. The configuration traits structure lists all the devices in a given class which are present in a given system configuration. A statically meta-programmed realization of the device class interface aggregates all the devices listed by the configuration traits. This realization is concrete when all the devices in a class are of the same type and polymorphic when different sensor types are present in a class.

Table 1 shows the memory footprint for sensing components in EPOS and their equivalents in TinyOS and MANTIS. Table 2 shows the maximum sampling rate obtained in tests with the three systems. The lowest overhead and higher sampling rate in EPOS are a direct result of the system's design, which minimizes dependencies between sensing components and the rest of the system. In EPOS, a component which abstracts an analog sensor usually depends only on the platform's analog-to-digital converter and its I/O subsystem, which is in turn abstracted by inline or meta-programmed operators. This minimizes overhead, even considering that EPOS includes conversion and calibration functions which the other systems do not include in equivalent components.

The EPOS communication infra-structure relies on the C-MAC protocol to provide low-level communication support. C-MAC[13] is a Configurable Protocol for medium access control in wireless sensor networks equipped with low power radio transceivers. Its configurable characteristic allows the user to adjust several communication parameters (e.g., synchronization, data detection, acknowledgments, contention, sending and receiving), in order to adjust the protocol to the needs of different applications.

Given the simplicity of communication hardware for sensor networks, Medium Access Control protocols and other data link layer services must be implemented in software. Services such as data packet detection, error detection and treatment, addressing, packet filtering and others traditionally implemented in hardware become one of the main parts of a communication stack implemented by operating systems for wireless sensor networks.

Medium access control protocols for sensor networks compromise performance (latency, throughput) for cost (power consumption). Power consumption is minimized mainly by shortening the period in which the radio listens to the channel when there are no communications (idle listening).

Contention-based protocols, such as B-MAC[12] attain energy efficiency by increasing the message preamble, allowing the radio channel to be verified with lower periodicity. Slot-based protocols, such as S-MAC[15], reduce power consumption by limiting communication to well-defined periods. Comparisons in different application scenarios show that there is no 'optimal' protocol for sensor networks[10]. The choice of an adequate MAC protocol for a wireless sensor network application depends on the level of compromise between power efficiency and communication flexibility. Characteristics such as: complexity, special hardware requirements (e.g., synchronization hardware) and application data communication patterns must be taken into consideration when determining the ideal MAC for a given scenario. In what regards communication support in an operating system for sensor networks applications, configuration flexibility may be considered the most desirable trait.

In the EPOS system, the C-MAC protocol uses a meta-programmed framework to build a configurable communication kernel, over which other protocols may be composed. Protocol configuration is performed at compile-time and run-time configuration of protocol characteristics is not treated in the current C-MAC architecture. The overhead of maintaining several configuration possibilities programmed in the node and the need of a second protocol for synchronization makes the use of a run-time configuration system impracticable for a protocol as widely configurable as C-MAC. The main C-MAC configuration points include:

Basic communication characteristics: These configurations are handled by the communication hardware and include: transmission frequency and power (which may be altered in runtime); modulation type (e.g., Manchester, NRZ); transmission data rate.

Duty cycle and organization: The duty cycle determines the active period in which the radio may operate. In a simple CSMA-based configuration, the radio may transmit at any time it detects the channel is free. On the other hand, in a slot-based protocol, the duty cycle is limited to the active part of the protocol's time slot.

Collision-avoidance mechanism: The collision-avoidance mechanism in a wireless sensor networks MAC protocol may be comprised of a carrier sense algorithm, the exchange of contention packets (Request to Send (RTS) and Clear to Send (CTS), or a combination of both. Furthermore, there must be the possibility to not use any collision-avoidance mechanism, for example, in a sparse network with little communication, in which eventually retransmitting corrupted packets is less costly than the mechanism itself.

Collision-detection mechanism: As hardware for communications in wireless sensor networks is mostly half-duplex, the most widely used mechanism for collision detection is the use of acknowledgment packets, sent from the receiving node to indicate that the data was correctly received. In situations where packet loss is not a problem (e.g., a densely installed network, where many information packets are redundant), the collision detection mechanism may be eliminated from the protocol configuration, thus increasing power efficiency.

Collision handling mechanism: When a collision is detected, the protocol may retransmit the packet, or simply increment a packet loss counter.

C-MAC's configurable characteristics are selected by the programmer through Configurable Traits in EPOS. Configurable Traits are parameterized classes whose static members describe the properties of a certain class. When a certain property is selected, the functionality it describes is included into the protocol. On the other hand, due to the use of function inlining and static meta-programming when a certain characteristic is not selected, no overhead associated with it is added to the final object code of the protocol. Furthermore, C-MAC's modular design allows different radio transceivers to be used with no alterations in the protocol's logic.

Tests with C-MAC have presented slightly superior performance than a protocol configured in an equivalent fashion, with smaller memory footprint[13]. This advantage is magnified by C-MAC's configuration system, which allows the creation of application-specific protocols, with only the necessary overhead.

CONCLUSION

This work presented the design and implementation of a runtime support environment for wireless sensor network applications based on the EPOS system. This environment includes a power management strategy, a field reprogramming strategy, a sensor data acquisition system and a configurable medium access control protocol for sensor network radios.

Our power management strategy allows applications to express when certain software components are not being used, permitting the system to migrate hardware resources associated with these components to lower power levels and features an autonomous, opportunistic power manager. Our field reprogramming strategy allows dynamic update of applications and the system through a transparent indirection mechanism. Our uniform abstraction of families of sensing device allows applications to collect data from sensors without having to deal with specific hardware details and without incurring excessive overhead. The C-MAC (Configurable MAC) protocol, allows applications to configure the communication channel according to their needs, including in the final protocol only the services selected by the application developer.

REFERENCES

1. Abrach, H., S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng and R. Han. MANTIS: System support for multimodal networks of *in-situ* sensors. In: 2nd ACM International Workshop on Wireless Sensor Networks and Applications, September, San Diego, USA, pp: 50-59. DOI: 10.1145/941350.941358.
2. Barr, R., J.C. Bicket, D.S. Dantas, B. Du, T.W.D. Kim, B. Zhou and E.G. Sirer, 2002. On the need for system-level support for ad-hoc and sensor networks. SIGOPS Operat. Syst. Rev., 36: 1-5. DOI: 10.1145/509526.509528.
3. Dunkels, A., B. Grönvall and T. Voigt, 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the First IEEE Workshop on Embedded Networked Sensors, November, Tampa, USA, pp. 455-462. DOI: 10.1109/LCN.2004.38.

4. Fröhlich, A.A., 2001. Application-Oriented Operating Systems. GMD-Forschungszentrum Informationstechnik, Sankt Augustin, Germany. First Edition. ISBN: 3-88457-400-0.
5. Han, C.C., R. Kumar, R. Shea, E. Kohler and M. Srivastava, 2005. A dynamic operating system for sensor nodes. In: Proceedings of the 3rd International Conference on Mobile Systems, Applications and Services, June, ACM Press, New York, USA, pp: 163-176. DOI: 10.1145/1067170.1067188.
6. Handziski, V., J. Polastre, J. Hauer and C. Sharp, 2004. Flexible hardware abstraction of the TI MSP430 microcontroller in TinyOS. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems. November, ACM, New York, pp: 277-278. DOI: 10.1145/1031495.1031534.
7. Hill, J., R. Szewczyk, A. Woo, S. Hollar, D. Culler and K. Pister, 2000. System architecture directions for networked sensors. In: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, November, Cambridge, USA, pp: 93-104. DOI: 10.1145/356989.356998.
8. Hoeller Junior, A.S., L.F. Wanner and A.A. Fröhlich, 2006. A hierarchical approach for power management on mobile embedded systems. In: 5th IFIP Working Conference on Distributed and Parallel Embedded Systems, October, Braga, Portugal, pp: 265-274. DOI: 10.1007/978-0-387-39362-9_28.
9. Hofmeijer, T., S. Dulman, P. Jansen and P. Havinga, 2004. AmbientRT-Real time system software support for data centric sensor networks. In: 2nd International Conference on Intelligent Sensors, Sensor Networks and Information Processing, December, IEEE Computer Society Press, Melbourne, Australia, pp: 61-66. DOI: 10.1109/ISSNIP.2004.1417438.
10. Langendoen, K. and G. Halkes, 2005. Embedded Systems Handbook. Chapter Energy-Efficient Medium Access Control. CRC Press. ISBN: 9780849328244
11. Marcondes, H., A.S. Hoeller Junior, L.F. Wanner and A.A. Fröhlich, 2006. Operating systems portability: 8 bits and beyond. In: 11th IEEE International Conference on Emerging Technology and Factory Automation, September, Prague, pp: 124-130. DOI: 10.1109/ETFA.2006.355371.
12. Polastre, J., J. Hill and D. Culler, 2004. Versatile low power media access for wireless sensor networks. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, November, ACM Press, New York, USA, pp: 95-107. DOI: 10.1145/1031495.1031508.
13. Wanner, L.F., A.B. de Oliveira and A.A. Fröhlich, 2007. Configurable medium access control for wireless sensor networks. In: International Embedded System Symposium, May, Irvine, CA, USA, pp: 401-410. DOI: 10.1007/978-0-387-72258-0_34.
14. Wanner, L.F., A.S. Hoeller Junior, A.B. de Oliveira and A.A. Fröhlich, 2006. Operating system support for data acquisition in wireless sensor networks. In: 11th IEEE International Conference on Emerging Technology and Factory Automation, September, Prague, pp: 582-585. DOI: 10.1109/ETFA.2006.355355.
15. Ye, W., J. Heidemann and D. Estrin, 2002. An energy-efficient MAC protocol for wireless sensor networks. In: 21st Conference of the IEEE Computer and Communications Societies, Vol. 3, June, IEEE, New York, USA, pp: 1567-1576. DOI: 10.1109/INFCOM.2002.1019408.