

# Wireless sensor networks dynamic runtime configuration

Stefan Dulman<sup>1</sup>, Tjerk Hofmeijer<sup>2</sup>, Paul Havinga<sup>1</sup>

<sup>1</sup>EEMCS Faculty, Twente University

<sup>2</sup>Ambient Systems, the Netherlands

P.O.Box 217, 7500AE, Enschede, the Netherlands

E-mails: {dulman, havinga}@cs.utwente.nl, hofmeijer@ambient-systems.net

*Abstract*— **Current Wireless Sensor Networks (WSN) use fixed layered architectures, that can be modified only at compile time. Using a non-layered architecture, which allows dynamic loading of modules and automatic reconfiguration to adapt to the surrounding environment was believed to be too resource consuming to be employed.**

**We have created a so-called data centric architecture and developed a new operating system (DCOS), to support it. As we will show in this paper, the new architecture and operating system are good candidates for WSNs, allowing flexibility in the configuration and exploitation of the sensor network.**

**Keywords**— real time, operating system, sensor network

## I. INTRODUCTION

Wireless Sensor Networks have gained a lot of attention lately. Due to technological advances, building small-sized, energy-efficient reliable devices, capable of communicating with each other and organizing themselves in ad-hoc networks has become possible. These devices have brought a new perspective to the world of computers as we know it: they can be embedded into the environment in such a way that the user is unaware of them. There is no need for reconfiguration and maintenance as the network organizes itself to inform the users of the most relevant events detected or to assist them in their activity.

One key issue that brings wireless sensor networks (WSN) one step closer to reality is the system software running inside each sensor node. Ideally, this software component should allow the user to write his/her own application directly, without spending additional time on managing the low level hardware or the basic functionality of the node. The most convenient framework would be an operating system like software

that offers at least real time scheduling mechanisms, memory management and resource management. Additionally, drivers for the existing hardware, as well as design templates for the software components are desirable.

This document describes the design and operation of DCOS. The objective of designing DCOS was to create a Real Time Operating System that fits inside the limited memory of a sensor node while still supporting low-power modes and causing only minimal overhead.

In fact, by supplying higher layer tasks and real time support, the overall efficiency can be improved. Against the intuitive idea that real time scheduling and re-configurability are too expensive in terms of resources, the initial results show that with the current hardware used this is possible and relieves the application programmer to concern about a lot of tiny details.

DCOS is a Real-Time Operating System for embedded devices with very limited memory, processing, and energy resources. Despite these limitations, DCOS has very powerful features like, real-time scheduling, dynamic memory allocation, online re-configurability, and support for a data driven architecture. Where other operating systems for tiny embedded applications offer configuration only during compile time, DCOS is a dynamic system, able to adapt its functionality to create the most efficient configuration for every situation.

Furthermore, with the module support in DCOS, applications can be modular in binary format, as well as in code. Firmware can be upgraded by replacing only certain parts, instead of the complete binary. This simplifies the upgrade process, and it conserves precious energy.

## II. MOTIVATION

Current WSN architectures mostly use the same fixed protocol stack for each node. For simple applications, the choice of the internal architecture might not be a problem, but as soon as we address more complex applications the need of having a more advanced, dynamically reconfigurable architecture appears. Taking a look at a fixed protocol stack, the following issues can be raised:

- Sensor networks are usually deployed in dynamic environments where each node must adapt itself to changes. The idea behind node re-configurability is that the network can adapt its functionality to the current situation, in order to lessen the use of the scarce energy and memory resources, while maintaining the integrity of its operation.
- The layers tend to merge. Functionality originally included in all layers might be deployed in a separate, specific block to allow better usage of the resources or the other way around. For example: error control normally resides in all layers so that for all layers the worst case scenario is covered. For a WSN this redundancy might be too expensive, as the worst case scenario never happens in the same time for all layers. Removing parts of the error control from layers and placing it in a specialized component of the architecture might allow comparable tolerance to errors achieved with a reduced energy consumption. Power control is normally done in the MAC layer only. In a WSN it can also be found at the application, routing, clustering and link layers. A specialized component to monitor the energy level and to coordinate the function of the other blocks based on it makes sense in the case of WSNs.
- The place of a protocol inside the stack might change with time. Protocols like timing and synchronization, localization, calibration, can change place after their transient phase is over or when their results might allow a different, more efficient, protocol stack.
- New protocols or corrected versions of the existing ones may become available after the network deployment. Using the new protocols could easily become a stringent requirement for the sensor network. In this case, it would be desirable to allow the network to automatically spread and download into the sensor nodes only the new protocols instead of compiling the whole

code again and programming each node separately.

These reasons led to the creation of the new data centric architecture and development of the DCOS operating system to support it.

## III. DATA CENTRIC ARCHITECTURE

As we have seen in Section II, a large number of architectures currently in use in sensor networks use a layered protocol stack. At the same time, individual components and the architecture as a whole can be tuned mainly at the compilation time and thus have limited capabilities of re-configurability during runtime.

We propose as an alternative, to place all the protocols around a central component that will have the task to enable or disable certain functionalities of each layer and also more generally, to select, enable or disable layers during run-time. In order to achieve a working structure, the protocols must comply to certain design rules and provide some additional information in their descriptors, as will be described bellow.

In the following, any protocol or algorithm that can run inside a sensor node, will be addressed as *Data Centric Entity* (DCE). The DCEs produce data. They can be triggered by other data (when it becomes available) and they might also request to read global data at certain moments of times. The process of producing data by a DCE defines its *functionality*. For the same functionality several DCEs might exist. To discern between them, one should take into consideration their *capabilities* (the cost for that DCE to perform its functionality and achieved performance and quality).

The nodes running inside a sensor network are usually driven by events. These events range from phenomena in the environment the WSN is exploring, to the commands given by the user. We will use the term *data* as an abstraction of regular events, as an extension to them. Data includes the events but also the mechanisms by which internal components exchange information among themselves. Each piece of data is characterized by a name and the content itself. Additional information (such as how long the data is valid, etc.) can be added.

The implementation of any given architecture can not abstract itself from the poor resources each sensor node has (the small amount of RAM memory and a weak processor). So, going down from the abstraction level to where the device is actually working, a compulsory step is implementing the envisioned architecture in a dedicated operating system.

Usually, these two issues are treated separately, both the architecture and the operating system in use try to be as general as possible and to cover all the possible cases. A simple view of a running operating system is a *scheduler* that takes care of a set of tasks. This scheduler is the central component that decides the systems operation. In order to obtain the smallest overhead possible there should be a tight integration between the function of the central nucleus from the architecture and the function of the scheduler from the operating system. In other words, a very close relationship between the DCEs and the operating system running on the nodes has to be developed.

Simple applications (as monitoring the value of one or more non-critical sensors with a period in the order of seconds) do not require real time capabilities from the underlying operating system. The granularity of time can be reasonably large and synchronization problems can be easily ignored for this sort of applications. Nevertheless, there is a broad range of applications where real time capabilities are needed. This includes scenarios where critical sensors are being monitored and guarantees need to be given on the system response time, applications where complex signal processing needing accurate timing is involved, the usage of advanced TDMA based Media Access Control protocols that need strict synchronization, etc.

Another reason for which we decided to include real time capabilities in the operating system we designed, is to see if soft real time scheduling can be achieved on such a resource limited processor and what would be the trade-off between the added overhead and the benefits it provides. In the proposed operating system (DCOS), the task scheduler functionality is extended to dynamically manage the existing DCEs. The central component in the sensor node architecture will be referred to as the *scheduler*. This has the meaning of *set of DCEs scheduler*. It will combine the common functionality of the set of DCEs scheduler from the architecture and the task scheduler from the operating system. The main challenge is keeping the footprint of the operating system small, leaving enough space for the networking protocols and user applications.

The *Local Data Centric Architecture* combines the functionality of the core of an soft real time operating system and a centralized architecture for a WSN. The scheduler is called the *Data Centric Scheduler*. It is the central component that keeps track of all the DCEs inside a sensor node and has two main functions: to decide which DCEs should be activated (and how they should be connected), thus controlling the architecture

and functionality of the node, and to manage the data flow between them. Tuning at run time even goes one step even further, the first described function is allowing also the replacement of the scheduler itself.

#### A. Data Driven Decisions

As mentioned earlier, the general concept of *data* is used rather than the *event* one. In order for the decision based on data to work, there are some additional requirements to be met. First of all, all the DCEs need to declare the name of the data that will trigger their action, the name of the data they will need to read during their action (this can generically incorporate all the shared resources in the system) and the name of the data they will produce. All this information is available for the scheduler to make the decisions.

From the point of view of the operating system, a new component that takes care of all the data exchange needs to be implemented. This would be in fact an extended message passing mechanism, with the added feature of notifying the scheduler when new data types become available. The mapping of this module in the architecture is the constraint imposed to the protocols to send/receive data via a publish/subscribe mechanism to the central scheduler.

An efficient naming system for the DCEs and the data is also needed. Downloading new DCEs to a sensor node involves issues similar to services discovery. Several DCEs with the same functionality but with different requirements and capabilities might coexist. The data centric scheduler has to make the decision which one is the best.

#### B. Architecture Extension

The architecture presented could be extended to groups of sensor nodes. Several Data Centric Schedulers together with a small, fixed number of protocols can communicate with each-others and form a virtual backbone of the network. DCEs running inside sensor nodes can be activated using data types that become available at other sensor nodes (for example, imagine one node using his neighbor routing entity because it needs the memory to process some other data). Of course, this approach raises new challenges.

A naming system for the functionalities and data types, reliability issues of the system (for factors such as mobility, communication failures, node failures, security attacks) are just a few examples. Related work on these topics already exist (for example: [1][2]).

## IV. DCOS

The ideas presented in Section III led to a new operating system, DCOS. DCOS is addressed to resource limited embedded devices like the sensor nodes of a WSN, and enables the data-centric architecture. Its current implementation targets the MSP430 Texas Instruments processor. DCOS has very powerful features like, real-time scheduling, dynamic memory allocation, online reconfigurability.

### A. Kernel Overview

DCOS uses real-time preemptive scheduling on hardware considered having too limited resources to do so. In order to simplify scheduling, other systems use methods as cooperative scheduling, where the real-time behavior is mostly the responsibility of the programmer, or event driven operation like TinyOS [3], where system behavior is unpredictable and real-time guarantees can not be given at all.

By using these methods, the other systems rule out the advantages of preemptive scheduling (thought of being too resource demanding to be feasible on resource-poor sensor nodes), like a better responsiveness. Furthermore, with DCOS, priorities do not have to be assigned to tasks but are dynamically defined based on the timing properties and the moment in time. This makes the system have a better processor utilization compared to systems that use the aforementioned methods.

DCOS supports the data centric architecture. With the data centric architecture the components of an embedded applications can be enabled or disabled, or mutually rearranged. Connections between the different components are data streams that are centrally coordinated. Based on the available data types and on the environmental state, the best fitted components can be selected to run. In other words, we provide reconfiguration of functionality online, at run time, instead at application compilation only. Supporting the data centric architecture also implies that no added functionality for *Inter Process Communication* (IPC) is needed.

The kernel uses a scheduling methods called *Earliest Deadline First with Inheritance* (EDFI) [4] that enforces mutual exclusion of shared resources without the need of semaphores, or monitors. By adding to a task a list of used resources, the scheduler can determine if there are tasks that share resources and schedules these tasks so that concurrent access to these resources is excluded. This simplifies application development in contrast to other systems where the programmer must provide these mechanisms. The latter is a complex task, liable to

errors that can cause dead-lock.

### B. The Data Manager

DCOS incorporates the extended message passing mechanism component as described in Section III, which in DCOS is called the *Data Manager*. The functional part of the data centric scheduler will be divided in two: the real-time scheduler and the data manager. The DCEs  $E_1$  to  $E_n$  will be defined as tasks and are scheduled by a real-time scheduler. Data is defined as a set of distinct objects. A single object is called a *Data Type* (DT). The flow of DTs between DCEs (tasks) is regulated by the data manager.

Data is defined as a set of DTs. Each DT has a fixed size and a unique identifier. The data manager maintains a table that holds a descriptor for each DT. In the descriptor the name, size, and memory location of the DT is defined. For tasks it is possible to produce and consume any subset of the available DTs in the table.

The memory area of a DT is defined as a shared resource, so that with the use of the resource sharing mechanisms available in the kernel, mutual exclusion is obtained, preventing data corruption by making it impossible for two tasks to write a DT's memory area simultaneously. For tasks to have access to DTs, the data manager provides functionality that enables tasks to obtain a reference to the memory area of the DT.

Through a publish/subscribe system the data manager enables tasks to respond (subscribe) when another task alters a certain DT (publish). The data manager keeps a list of subscribed tasks for each DT. When a task writes the memory area of a DT, it has to inform the data manager about this. The data manager will then traverse through the list of subscribers for that DT, requesting the task scheduler to release each entry. Special masking provisions in the descriptor of a task makes it possible to temporarily disable certain subscriptions.

The publish/subscribe mechanism is also used for system interrupts. When e.g. a timer goes off, its ISR will publish a system defined DT. By doing so, tasks can respond to system events by subscribing to their special DTs. Memory management plays an important role in any operating system. DCOS does heap allocation using what we call the *First-Free-Fit and Shrink* (3FS) algorithm. The allocation algorithm searches for the first free block of memory that is large enough to fit the requested size and shrinks the free block by that size. The chopped-off top of the block is then marked allocated. When an allocated block is freed it is marked free and if the block just before and/or after it is also a free block then the blocks are merged into a single free

Criterion	DCOS	TinyOS	Salvo
Scheduling method	Preemptive	FIFO buffer	Cooperative
Task switches per second	12500	40000	40000 <sup>1</sup>
Maximum number of tasks	HeapSpace/14	unknown	Unknown
Maximum number of scheduled tasks	HeapSpace/14	unknown	Unknown
Maximum task runtime	inf <sup>2</sup>	unknown	Inf
Available ROM	61440	8192	61440
Available RAM	2048	512	2048
Kernel ROM	3800	432	1550 <sup>3</sup>
Kernel RAM	32 <sup>4</sup>	46	48 <sup>3</sup>
Dynamic task priorities	Yes	No	No
Resource access protocol	Yes	No	No
Dynamic memory allocation	Yes	No	No
Dynamic loadable modules	Yes	No	No

<sup>1</sup> Using a MSP430 running at 8 MHz

<sup>2</sup> If the scheduler quantum guarding option is enabled this figure would be 2s

<sup>3</sup> Using Salvo tu6pro

<sup>4</sup> Excluding the heap space

Table 1: DCOS compared to other sensor network operating systems

block so that fragmentation is reduced.

The first advantage of the algorithm is that it only needs a single value per allocated block and two values for a free one on overhead. These values are stored in the block itself. It is no problem to use a bit more memory overhead on a free block than an allocated block because it is unused memory anyway.

Allocating the block even means freeing up its overhead. The other advantage is that it only keeps track of the free blocks and in that way reduces the length of the linked list and thereby the search time.

## V. RESULTS

The DCOS kernel prototype is implemented on a Texas Instruments MSP430 microcontroller running at 4.6MHz. The controller has 2048 bytes of RAM, and 60KB of program flash memory.

For the performance metric of the scheduler, we have measured its latency using task-sets of different sizes, ranging from 1 to 16 tasks. Latency is the maximum computation time of the scheduler, and is the time between the activation of the scheduler and the moment the CPU continues, or starts executing a task. The measured latency ranges from 80 $\mu$ s for the smallest task-set, to 110 $\mu$ s for the largest, which is approximately less than double the latency incurred with cooperative scheduling. Based on the measured latency, the

maximum number of task switches per second ranges from 9000 to 12500.

For the basic kernel implementation, consisting of the scheduler, data manager, dynamic memory allocator, and the minimum required *Hardware Abstraction Layer* (HAL), the kernel uses 3800 bytes of program flash memory. The absolute minimum RAM usage of the kernel is 32 bytes and 26 bytes of possible stack usage. For each task an additional 10 bytes of heap space is needed.

Table 1 presents a comparison between DCOS, Salvo and TinyOS in terms of memory consumption and latency. As it can be noticed, the figures lie in the same order of magnitude. The latency of the scheduler is, as expected, larger than the other two systems and the difference is basically the price paid for the real time scheduling and the data management that occurs each time a task exists the processor.

## VI. CONCLUSIONS

In this paper we have outlined the main ideas for a new architecture for wireless sensor networks, more suited to the dynamical environment in which these devices need to operate. The new architecture allows nodes to reconfigure themselves online, adapting to the external conditions (such as the density of nodes, availability of certain services in the network, focus on sensing in the presence of the events or just relaying

function, etc.) and also to the internal ones (changing of the energy level, availability of new software modules or availability of new data that allows running of advanced protocols and internal architecture, etc.).

Based on the proposed data centric architecture we designed a new operating system (DCOS) for wireless sensor networks. We have chosen a light weight soft real time scheduler for it, to allow the extension of usage of sensor networks to more complex scenarios where there is a need for real time guarantees.

The scheduler of DCOS provides soft real time guarantees for the chosen set of tasks. Additionally, it offers the user additional benefits as automatic mutual exclusion mechanism, memory management, automatic reconfiguration at run time, etc. DCOS has a small footprint and a small context switch time, in the same order of magnitude as the other existing operating systems for sensor networks.

At a comparable cost it offers the user a lot of added functionality that makes it an attractive choice for this sort of networks.

#### REFERENCES

- [1] P. Verissimo and C. A. Event-driven support of real-time sentient objects. In Proceedings of the Eighth IEEE International Workshop on Object Oriented Real-time Dependable Systems (WORDS 2003), Jan. 2003.
- [2] E. Cheong, J. Liebman, J. Liu, and F. Zhao. Tinygals: A programming model for event-driven embedded systems. In ACM Symposium on Applied Computing, 2003.
- [3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. ASPLOS 2000, Nov 2000.
- [4] P. G. Jansen, S. J. Mullender, P. J. M. Havinga, and J. Scholten. Lightweight EDF scheduling with deadline inheritance. Technical report TR-CTIT-03-23, CTIT, Univ. of Twente, The Netherlands, May 2003.