Microkernel for Nodes of Wireless Sensor Networks

Vinícius Coelho de Almeida, Luiz Filipe Menezes Vieira, Marcos Augusto Menezes Vieira, Breno Augusto Dias Vitorino,
Antônio Otávio Fernandes, Diógenes Cecílio da Silva, Claudionor Nunes Coelho Jr.
Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Projeto: SensorNet/CNPq

Introduction

Wireless sensor networks (WSN) may be composed from thousands of nodes, which possess storing, processing,communication and sensing capacity, although with strong limitations. They must have self-configuration and adaptation mechanisms to support fault-tolerance. Moreover, they may be equipped with different sensors, given the applications' nature inside them, such as: temperature, pressure, movement, etc.

Figure 1: A typical WSN

A microkernel may be defined as an operating system(OS) core. Its objective is to ease the programmer's work in developing application for a platform and to manage the resources that constitute the sensor node. The nodes' microkernel must provide to the programmer routines that allow the development of his applications.

This work is part of SensorNet project, whose research is focused in architecture, protocols, management and applications in WSNs. Its final goal Is to develop a microkernel for nodes in WSNs, using eligible concepts from existing systems. This microkernel will run in a device still being developed In that project.

Features

Our microkernel for nodes in WSNs, currently in development stage, is being implemented in C language. This decision was taken in order to grant more portability across different microcontrollers, not depending on a specific architecture. We analyzed different OSs for nodes in WSNs and embedded devices, such as: TinyOS, Bertha, SensorWare and Eyes. This allowed us to identify their eligible features and concepts, which constitute the base for the new microkernel in development.

Event-driven data delivery model

WSNs can be divided in 4 delivery models: continuous, event-driven, observer-initiated and hybrid. In continuous type, the sensors will report their readings at determined rate. In event-driven, sensors will inform to the application when certain events occur. In observer-initiated, sensors will reply to a request from applications. At last, in hybrid, the later approaches can be present at the same time.

Due to the reactive behavior of the most of applications in WSNs, the event-driven model was the chosen to integrate the new microkernel. It is the most attractive because the energy expenses of this model are low, when compared to the other cited approaches.

Code mobility support

Transmission in WSNs is the process that consumes the most energy, being a good practice to minimize code mobility. However, this functionality should be supported because it allows to dynamically deploying different algorithms over the network. In addition, the nodes themselves can be programmed automatically, from a "programmer" node.

Application program interfaces separation

Nodes memory in WSNs is a very scarce resource. So it must be used parsimoniously. Some OS functionalities must be available all the time, but there are others that are application specific. Separating these functionalities in distinct application program interfaces (API) and installing them in the nodes in a need-to-use base, the memory will be used most efficiently. For example, if a node will only read temperature, it is unnecessary and wasteful to load the movement sensing API, too. The objective of this approach is clear: load in a node only the needed functionality in order to save memory.

Operation on constrained resources

Nodes in WSNs possess storing, processing, communication and sensing capability, but they are very limited. The microkernel must consider this fact, in order to attend the specific requirements of these nodes and their WSNs. The identified demands to be attended are: low energy consumption, small computing power, fault-tolerance and self-configuration.

Nodes Management

Tasks scheduling and priority designation; memory allocation and deallocation for applications, contiguous addressing space between RAM and Flash memory; access to hardware resources, mainly radio and sensors.

Fault-tolerance

Nodes may be distributed at harsh environments or difficult access regions. So, they must try recover themselves from eventual failures.Otherwise, the node must, at least, try to restart.

Services

The microkernel is situated between hardware and software levels, as depicted in Figure 2. The services are divided in APIs and integrate the microkernel core, being always available in a node. Their signatures, in C-like format, are described below. Other APIs, specific to the sensing type (pressure, temperature, etc), will be created from that core services.

Figure 2: Microkernel: located between software and hardware

Radio API

Int *Send* (Packet p): sends a data packet through the radio.
Packet *Receive*(): receives a data packet from the radio.
int *SetMode*(int mode): sets the operation mode, where each mode provides different functionality levels (and consuming different energy levels).
int *SetRange*(int meter): sets the radio's range, in meters.

Sensor API

int *Read*(int port): reads one byte.
int *SetMode*(int mode): same as Radio API.
int *SetInterrupt*(int port): prepares the sensor to trigger an interruption when it detects an event.
int *DisableInterrupt*(int port): frees the sensor from the obligation to trigger an interruption when it detects an event.

Memory API

int *Load*(int size, char *addr, char *buf): read bytes from the memory.
int *Store*(int size, char *addr, char *buf): store bytes on memory.
Int *SetMode*(int mode): same as Radio API, but to Flash memory.

I/O Ports API

int *Read*(int port): same as Sensor API.
int *Write*(int buf, int port): writes one byte in a port.
int *SetInterrupt*(int port): same as Sensor API.
int *DisableInterrupt*(int port): same as Sensor API.

Battery API

int *EnergyLevel*(): returns the avaliable energy level in the node's battery.

Mobility API

int *SendCode*(char *code): sends code on radio to execution in a neighbor node.
int *ReceiveCode*(char *code): receives code on radio from a neighbor node, for posterior execution.

Microcontroller API

int *SetMode*(int mode): same as Radio API.
int *SetTimer*(int period, int timer): set the timer to trigger an interruption at periodic intervals.
Int *DisableTimer*(int timer): frees the microcontroller from the obligation to trigger an interruption at periodic intervals.
Other functionalities: system clock generator, RAM clearing routine, RAM self-test, ROM checksum, integer multiplication, interrupt management.

Conclusions

The microcontroller to be used to execute the microkernel is a MSP430, from Texas Instruments. However, we believe that the proposed features and services may be ported to other similar microcontrollers smoothly. The implementation language is a C extension specific for the chosen microcontroller. The sensor node is being built in SensorNet project and it will be used in order to validate the functioning of our proposed microkernel.

UFMG - ICEx
DEPARTAMENTO DE CIÊNCIA DA
COMPUTAÇÃO

Laboratório de Engenharia de Computadores
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais