Ασύρματα Δίκτυα Αισθητήρων

Πέππας Κωνσταντίνος

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Application development for sensor networks differs in many ways from programming “traditional” distributed computing systems.
- Examples of such differences include the continuous interaction of sensor nodes with their physical environment, the stringent resource constraints of sensor nodes, the ad hoc deployment of many sensor networks, and the frequent changes in network topology due to failures or mobility.
- From the network developer’s perspective, the goal is to design and program a reliable and efficient wireless sensor network that can cope with the dynamics and uncertainties present in sensing systems.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- From the user's perspective, the network is often viewed as a database and the users interact with sensor nodes via queries, which must be responded to in a reliable and efficient fashion.
- Many simulation tools and techniques are closely tied to the operating system used on sensor nodes.
- Sensor network programming approaches can be classified as either *node-centric* or *application-centric*.
- Node-centric languages and programming tools focus on the development of sensor software on a per-node level.
- In contrast, programming using an application-centric approach considers parts or all of the network as one single entity

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- A sensor network differs from traditional computing environments in various aspects, thereby necessitating programming frameworks and tools that consider a sensor network's unique characteristics.
- Specifically, the following characteristics significantly affect the design of sensor network programming tools:
- 1. *Reliability:* Wireless sensor networks are inherently more unreliable than other distributed systems.
- Therefore, sensor networks are built to adapt to changing dynamics and node and link errors such that the network continues to serve its intended purpose even when parts of the network have failed.
- While many faults in a network will never be noticed by an application (e.g., a routing protocol autonomously reroutes traffic around a failed node), resilience to failures and topology changes should be supported by a programming environment.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- *Resource constraints:* Wireless sensor networks are typically very resource-constrained, which affects the programming approach, maximum code size, and other aspects of application development.
- Most notably, energy efficiency is particularly critical in WSNs and penetrates every aspect of sensor network design, from duty cycles to routing protocols to in-network data processing.
- Therefore, programming tools and models should allow a developer to effectively exploit energy-saving techniques and approaches, while details should be hidden from the programmer

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- *Scalability:* Sensor networks can scale up to hundreds and thousands of sensor nodes, therefore programming models should support developers in designing applications and software for large-scale (and possibly heterogeneous) networks.
- Manual configuration, maintenance, and repair of individual sensor nodes will be infeasible due to the large number of devices, therefore necessitating support for self-management and self-configuration.
- The scale of a network can also be addressed by using programming models that consider the entire network as one whole entity instead of focusing on each individual device

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- *Data-centric networks:* In many wireless sensor networks, not only are the individual sensor nodes of interest, but also the data they generate and disseminate.
- Sensor network applications are therefore concerned about obtaining useful information in a timely fashion, where it is irrelevant which sensor node(s) generated this information.
- Many applications are only concerned with the collection of data at a central point, for example, a server that stores, analyzes, or visualizes the sensor data.
- Other applications require immediate processing and analysis of data within the network, for example, to eliminate redundant information, to aggregate data from multiple sensors, and to quickly identify if sensor data should be propagated further or acted upon.
- Each category will require different programming models, where the latter category will also require support for collaboration, that is, programming a network results in generating distributed algorithms that must work across many or all nodes in a resource-efficient manner.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Under the node-centric model, programming abstractions, languages, and tools focus on the development of sensor software on a *per-node* level.
- The overall network-wide sensing application is then described as a collection of pairwise interactions of individual sensor nodes. This section describes examples of programming models that focus on software development for individual nodes

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The combination of the TinyOS operating system and the nesC (Gay *et al.* 2003) programming language has become the de facto standard for node-centric programming in WSNs.
- The programming language nesC is an extension to the popular C programming language and provides a set of language constructs to implement code for distributed embedded systems such as motes.
- TinyOS is a component-based OS written in nesC.
- Unlike traditional programming languages, nesC must address the unique challenges of WSNs.
- For example, activities in a sensor network (e.g., sensor acquisition, message transmission and arrival) are initiated by *events* such as the detection of a change in the physical environment.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- These events may occur while a node is processing data, that is, sensor nodes must be able to concurrently perform their processing tasks while responding to events.
- In addition, sensor nodes are typically very resource-constrained and prone to hardware failures;
- therefore, programming languages for sensor nodes should take these characteristics into consideration.
- Applications based on nesC consist of a collection of *components*, where each component *provides* and *uses* interfaces.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- A “provides” interface in nesC is a set of method calls that are exposed to higher layers, while a “uses” interface is a set of method calls that hide details of lower-layer components.
- An interface describes the use of some kind of service (e.g., sending a message).
- The following code shows a concrete example from the TinyOS timer service. This example provides the StdControl and Timer interfaces and uses a Clock interface (Gay *et al.* 2003).

Προγραμματισμός ασύρματων δικτύων αισθητήρων

```
module TimerModule {
    provides {
        interface StdControl;
        interface Timer;
    }
    uses interface Clock as Clk;
}

interface StdControl {
    command result_t init ();
}

interface Timer {
    command result_t start (char type, uint32_t interval);
    command result_t stop ();
    event result_t fired ();
}
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

```
interface Clock {
    command result_t setRate (char interval, char scale);
    event result_t fire ();
}

interface Send {
    command result_t send (TOS_Msg *msg, uint16_t length);
    event result_t sendDone (TOS_Msg *msg, result_t success
}

interface ADC {
    command result_t getData ();
    event result_t dataReady (uint16_t data);
}
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- This example also shows the definitions for the Timer, StdControl, Clock, Send (communication), and sensor (ADC) interfaces.
- The Timer interface defines two types of *commands* (which are essentially functions): start and stop.
- The Timer interface further defines an *event*, which is also a function. While commands are implemented by the providers of an interface, events are implemented by the users.
- Similarly, all other interfaces in this example define both commands and events.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Besides the interface specification, components in nesC also have an implementation.
- *Modules* are components implemented by application code, while *configurations* are components that are implemented by connecting interfaces of existing components.
- Every nesC application has a *top-level configuration* that describes how components are wired together.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Functions (i.e., commands and events) in nesC are described as *f.i*, where *f* is a function in an interface *i*.
- Functions are invoked using the *call* operation (for commands) and the *signal* operation (for events).
- The following code shows a brief excerpt of an implementation of an application that periodically obtains sensor readings (Gay *et al.* 2003).

Προγραμματισμός ασύρματων δικτύων αισθητήρων

```
module PeriodicSampling {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface Send;
}

implementation {
    uint16_t sensorReading;

    command result_t StdControl.init () {
        return call Timer.start (TIMER_REPEAT, 1000);
    }
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

```
    event result_t Timer.fired () {
        call ADC.getData ();
        return SUCCESS;
    }

    event result_t ADC.dataReady (uint16_t data) {
        sensorReading = data;
        ...
        return SUCCESS;
    }
    ....
}
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- In this example, StdControl.init is called at boot time, where it creates a repeat timer that expires every 1000 ms.
- Upon timer expiration, a new sensor sample is obtained by calling ADC.getData, which triggers the actual sensor data acquisition (ADC.dataReady).
- Returning to the TinyOS timer example, the following code sequence shows how the timer service in TinyOS (TimerC) is built by wiring two subcomponents, TimerModule and HWClock (which provides access to the on-chip clock).

Προγραμματισμός ασύρματων δικτύων αισθητήρων

```
configuration TimerC {
    provides {
        interface StdControl;

interface Timer;
    }
}

implementation {
    components TimerModule, HWClock;

    StdControl = TimerModule.StdControl;
    Timer = TimerModule.Timer;

    TimerModule.Clk -> HWClock.Clock;
}
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- In TinyOS, code executes either asynchronously (in response to an interrupt) or synchronously (as a scheduled task).
- Race conditions can occur when concurrent updates to shared state are performed.
- In nesC, code that is reachable from at least one interrupt handler is called *asynchronous code* (AC) and code that is only reachable from tasks is called *synchronous code* (SC).
- Synchronous code is always atomic to other synchronous codes, because tasks are always executed sequentially and without preemption.
- However, race conditions are possible when shared state is modified from AC or when shared state is modified from SC that is also modified from AC.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Therefore, nesC provides programmers with two options to ensure atomicity.
- The first option is to convert all of the sharing code to tasks (i.e., SC only).
- The second option is to use *atomic sections* to modify shared state, that is, brief code sequences that nesC will always run atomically.
- Atomic sections are indicated with the *atomic* keyword, which indicates that a block of statements should be executed atomically, that is, without preemption, as shown in the following code excerpt.
- Nonpreemption can be obtained by disabling interrupts for the duration of an atomic section.
- However, to ensure that interrupts are not disabled for too long, no call commands or signal events are allowed within atomic sections

Προγραμματισμός ασύρματων δικτύων αισθητήρων

```
...
event result_t Timer.fired () {
    bool localBusy;
    atomic {
        localBusy = busy;
        busy = TRUE;
    }
    ...
}
...
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- TinyGALS (Cheong *et al.* 2003) is a globally asynchronous and locally synchronous (GALS) approach for programming event-driven embedded systems.
- A TinyGALS program consists of modules, which are composed of components (the most basic elements).
- A component C has a set of internal variables *V*C, a set of external variables *X*C,. and a set of methods *I*C that operate on these variables.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Methods are further divided into calls in the ACCEPTSC set (which can be called by other components) and calls in the USESC set (which are those needed by C and may belong to other components).
- Similar to nesC and TinyOS, TinyGALS defines components using an interface definition and an implementation.
- For example, a possible interface description of a component DownSample is shown below, where the interface has two methods in the ACCEPTS set and one method in the USES set.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

```
COMPONENT DownSample
ACCEPTS {
    void init (void);
    void fire (int in);
};
USES {
    void fireOut (int out);
};
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The following code sequence shows the corresponding implementation for the Down-Sample component
- where _active is an internal boolean variable that ensures that for every other fire() method called, the component will call the fireOut() method with the same integer argument.

```
void init () {
    _active = true;
}
void fire (int in) {
    if (_active) {
        CALL_COMMAND (fireOut) (in);
        _active = false;
    } else {
        _active = true;
    }
}
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- TinyGALS modules consist of one or more components.
- A module M is a 6-tuple M =(COMPONENTSM, INITM, INPORTSM, OUTPORTSM, PARAMETERSM, LINKSM),
- COMPONENTSM is the set of components of M,
- INITM is a list of methods of M’s components,
- INPORTSM and OUTPORTSM specify the inputs and outputs of the module,
- PARAMETERSM is a set of variables external to the components,
- LINKSM specifies the relationships between the method call interfaces and the inputs and outputs of the module.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Modules are further connected to each other to form a complete TinyGALS system,
- A system is a 5-tuple S = (MODULESS, GLOBALSS, VAR_MAPSS, CONNECTIONSS, STARTS).
- The set of modules is described in MODULESS, global variables are described in GLOBALS,
- a set of mappings (each of which maps a global variable to a parameter of a module in MODULESS) is contained in VAR_MAPSS,
- CONNECTIONSS is a list of the connections between module output ports and input ports,
- and STARTS is the name of an input port of exactly one module, which is used as a starting point for the execution of the system.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The highly structured architecture of TinyGALS can be exploited to automate the generation of scheduling and event handling code, freeing software developers from writing error-prone concurrency control code
- Code generation tools can automatically produce all of the necessary code for component links and module connections, system initialization, start of execution, intermodule communication, and global variables reads and writes.
- Further, through the use of message passing, modules in TinyGALS become decoupled from each other, therefore facilitating their independent development.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Each message passed will trigger the scheduler and activate a receiving module.
- However, this may become quickly inefficient if there is global state that must be updated frequently.
- Therefore, TinyGALS provides another mechanism, called TinyGUYS (Guarded Yet Synchronous) variables, where modules may read global variables synchronously (without delay), but writes to the variables are asynchronous in the sense that all writes are buffered.
- The buffer is of size 1, that is, the last module that writes to a variable wins.
- TinyGUYS variables are updated by the schedule only when it is safe to do so, for example, after one module finishes and before the scheduler triggers the next module.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The Sensor Network Application Construction Kit (SNACK) is a configuration language, component and service library, and compiler for the development of sensor network applications (Greenstein *et al.* 2004).
- SNACK's goal is to provide *smart libraries* that can be combined to form sensor network applications, while, on one hand, simplifying the development process and, on the other, not losing control over efficiency.
- For example, to program a sensor node to periodically take temperature and light measurements and forward the sensor data to some sink, it should be possible to write a simple code sequence such as:

```
SenseTemp -> [collect] RoutingTree;
SenseLight -> [collect] RoutingTree;
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The following examples shows the syntax of SNACK code.

```
service Service {
    src :: MsgSrc;
    src [send:MsgRcv] -> filter :: MsgFilter -> [send] Network;
    in [send:MsgRcv] -> filter;
}
```

- Here, *n* :: *T* declares an instance named *n* of a component type *T* , that is, an instance is effectively an object of the given type.
- Further, *n*[*i* : τ] indicates an output interface on component *n* with name *i* and interface type τ (similarly, [*i* : τ]*n* refers to an input interface).
- A component *provides* its input interfaces and *uses* its output interfaces

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The SNACK library of components and services contains a variety of components for sensing, aggregation, transmission, routing, and data processing.
- For example, the messaging architecture of SNACK supports several core components, including
  - Network (which receives messages from and sends messages to the TinyOS radio stack),
  - MsgSink (which ends inbound call chains and destroys buffers it receives),
  - and MsgSrc which periodically generates empty SNACK messages and passes them on via an outbound interface).

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The SNACK Timing system has two core components:
  - TimeSrc, which generates a timestamp signal, emitted over its signal interface at a specified minimum rate,
  - and TimeSink, which consumes that signal.
- Storage in SNACK is implemented by components such as Node-Store64M, which implements an associative array of eight-byte values keyed by node ID.
- Finally, the SNACK Service library contains a variety of services, that is, combinations of primitive components.
- For example, the RoutingTree service implements a tree designed to send data up to some root.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The thread-based paradigm is popular in many computing systems and it has recently also found its way into sensor networks.
- In traditional event-based systems, event handlers are executed in response to events, and these handlers (tasks) run to completion without interruption from other tasks.
- The main advantage of the thread-based approach is that multiple tasks can make progress in their execution without the concern that a task may block other tasks (or be blocked by other tasks) indefinitely.
- For example, a task scheduler can execute a task for a certain amount of time, then preempt this task in order to execute another task.
- This *time-slicing* approach simplifies the programming of sensor systems, but also comes at the cost of increased operating system complexity

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- An example of a thread-based operating system for sensor networks is the MANTIS (MultimodAl system for NeTworks of In-situ wireless Sensors) OS,
- which occupies less than 500 bytes of RAM and about 14 kbytes of flash memory (Bhatti *et al.* 2005).
- For example, the ATMega128 sensor nodes have 4 kbytes of RAM and 128 kbytes of flash storage, that is, MANTIS OS leaves sufficient space for multiple sensor application threads.
- Besides memory efficiency, MANTIS OS also aims for energy efficiency by switching the microcontroller to a low-power sleep state after all active threads have called the operating system's *sleep()* function.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The goal of the TinyThread (McCartney and Sridhar 2006) library is to add support for multithreaded programming to sensor networks based on TinyOS and nesC.
- TinyThread enables procedural programming of sensor nodes and includes a suite of interfaces that provide several blocking I/O operations and synchronization primitives that make multithreaded programming safe and easy.
- Protothreads (Dunkels *et al.* 2005) are a very lightweight stackless type of threads. Instead of using a stack for each protothread, all protothreads run on the same stack and context switching is done by stack rewinding.
- A limitation of protothreads is that contents of variables must be explicitly saved before calling a blocking wait, since variables with function-local scope that are automatically allocated on the stack are not saved across such wait calls.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Finally, Y-Threads (Nitta *et al.* 2006) is another lightweight threading model that provides preemptive multithreading.
- Application developers identify the preemptable and nonpreemptable parts of a program.
- All threads share a common stack for their nonblocking computations, while each thread has its own stack for blocking calls.
- The key concept behind this approach is that the blocking portions of a program require only small amounts of stack, therefore achieving better memory utilization compared to other preemptive multithreading approaches.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Macroprogramming refers to a development approach where the focus is not on individual sensor nodes, but on the programming of groups of sensor nodes, including approaches that treat an entire network as a single entity.
- In-network processing is often performed to address the bandwidth and energy limitations of WSNs.
- However, decomposing data collection tasks into parallel programs with local communication among sensor nodes can be a challenging problem.
- Therefore, the goal of *abstract regions* (Welsh and Mainland 2004) is to provide higher-level programming interfaces that hide complex details from the developer, while still being flexible enough to support the implementation of efficient algorithms.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Many sensor applications are often characterized by group-level cooperation,
- that is, a group of nodes work together to sample, process, and communicate sensor data.
- Therefore, abstract regions are a communication abstraction intended to simplify the development process by providing a region-based collective communication interface.
- An abstract region defines the neighborhood relationship between a node and other nodes in the network, for example, as expressed by *“the set of nodes within distance d”*.
- Specifically, the type of definition of an abstract region will depend on the type of application

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Examples of implementations of abstract regions include N-radio hop (nodes within *N* radio hops), k-nearest neighbor (*k* nearest nodes within *N* radio hops), and spanning tree (a spanning tree rooted at a single node, used for aggregating data over the entire network).
- For example, for regions defined using hop distances, discovery of region members can be achieved using periodic broadcasts (advertisements).
- Data among region members can be shared using either a “push” (broadcasting updates to neighboring nodes) or “pull” (issue a fetch message to the corresponding node) approach.
- Reduction is another programming abstraction, which takes a shared variable key and an associative operator (e.g., sum, max, or min) and reduces the shared variable across nodes in the region.
- In abstract regions based on hop distances, reduction involves collecting shared variable values locally, combining them with the reduction operator, and storing the result in a new shared variable

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The EnviroTrack (Abdelzaher *et al.* 2004) object-based middleware library is a programming abstraction geared toward target-tracking sensor applications.
- Its goal is to free the developer from the details of interobject communication, object mobility, and the maintenance of tracking objects and their state.
- Similar to abstract regions, EnviroTrack uses the concept of groups.
- However, instead of concrete descriptions of the shape or size of a group, groups in EnviroTrack are formed by sensors which detect certain user-defined entities in the physical environment, with one group formed around each entity.
- Groups are identified by *context labels*, which can be thought of as logical addresses that follow the external tracked entity around in the physical environment.
- Further, objects can be attached to context labels to perform context-specific operations. These *tracking objects* are executed on the sensor group of the context label.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- The type of context label depends on the entity being tracked (e.g., a context label of *car* is created wherever a car is tracked).
- A programmer must provide several pieces of information to declare a context label of some type *e*.
- First, a function sensee() describes the sensory signature identifying the tracked environmental target, for example, for a car-tracking application, sensee() might be a function of magnetometer and motion sensor readings.
- Whenever the EnviroTrack middleware detects a target, it creates a sensor group around the target.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- This function is also used to maintain group membership, that is, all nodes that sense the given target (i.e., sensee() is true) are group members.
- Next, a programmer declares an environmental state shared by all objects attached to a context label by defining an aggregation function statee() that acts on the readings of all sensors for which sensee() is true.
- Aggregation is performed locally by a sensor node that acts as group leader.
- The EnviroTrack library contains a variety of distributed aggregation functions such as addition, averaging, and median computation.
- Finally, the programmer specifies which objects are to be attached to a context label.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- Another commonly used abstraction for sensor network programming is to treat a WSN as a distributed database that can be queried (e.g., using SQL-like queries) to obtain sensor data.
- A representative example of a distributed query processor for sensor nodes is TinyDB (Madden *et al.* 2005).
- Here, the network is represented logically as a table (called *sensors*) that has one row per node per instant in time.
- Each column in this table corresponds to a type of sensor reading such as light, temperature, pressure, etc.

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- A new record in this virtual table (i.e., a new row) is added only when a sensor is queried and this new information is usually stored for a short period of time only.
- Queries in TinyDB are very much like any other SQL-based database, that is, they use clauses such as SELECT, FROM, WHERE, and GROUP BY to build queries.
- For example, the following query specifies that each device should report its own identifier (nodeid), light reading, and temperature reading once per second for 10 seconds:

```
SELECT nodeid, light, temp
  FROM sensors
  SAMPLE PERIOD 1s FOR 10s
```

Προγραμματισμός ασύρματων δικτύων αισθητήρων

- As a result of this query, nodes initiate data collection at the beginning of each epoch (as specified in the SAMPLE PERIOD clause) and the results of such a query are streamed to the root of the network.
- TinyDB also supports grouped aggregation queries, that is, as data from an aggregation query flows up the tree, it is aggregated in-network according to an aggregation function and value-based partitioning specified in the query.
- For example, imagine a user who wishes to use microphone-equipped sensor nodes to monitor the occupancy of a room on a particular floor of a building.
- Assuming that rooms have multiple sensors, the goal is to look for rooms where the average volume is over a certain threshold. A query for this sensing request could be expressed as: