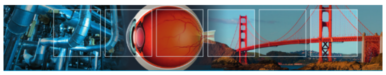
Chapter 12: Sensor Network Programming



Chapter 12: Roadmap

- Challenges
- Node-centric programming
- Macroprogramming
- Dynamic reprogramming
- Sensor network simulators

Challenges in WSN Programming

- Reliability
  - resilience to changes and failures is important in WSNs
  - should be supposed by a programming environment
- Resource constraints
  - resource limitations of WSNs affect maximum code size, performance, memory/storage capacities
  - programming environment should allow programmer to exploit energy-saving techniques
- Scalability
  - sensor networks can be very large, i.e., programming models must scale
  - manual configuration, maintenance, repair may be infeasible
- Data-centric networks
  - focus is on the data, not the devices

Node-Centric Programming

- Programming abstractions, languages, and tools focus on development of SW on a per-node basis
- Overall network-wide sensing task is then a collection of pairwise interactions of sensor nodes

nesC Programming Language

- TinyOS and nesC have become the de facto standard in WSN programming
- nesC is an extension to C programming language
- Provides set of language construct to implement SW for sensors
- nesC applications consist of a collection of components and each component has "provides" and "uses" interfaces
  - interface: describes the use of some kind of service
  - provides: set of method calls that are exposed to higher layers
  - uses: set of method calls that hide details of lower-layer components

nesC Programming Language

```
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 ();
}
```

nesC Programming Language

```
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);
}
```


nesC Programming Language

- Example: timer interface:
  - two types of commands (functions): start, stop
    - commands are implemented by providers of an interface
  - event (function)
    - events are implemented by the users
- Components have an implementation
  - modules are components implemented by application code
  - configurations are components implemented by connecting interfaces of existing components
  - every nesC application has a top-level configuration
    - describes how components are "wired together"
  - functions are described as *f.i* (*f* is a function in an interface *i*)
  - functions are invoked using the call operation (for commands) and the signal operation (for events)

nesC Programming Language

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

nesC Programming Language

```
implementation P
    uint16_t sensorReading;

    command result_t StdControl.init () {
            return call Timer.start (TIMER_REPEAT, 1000);
    }
    event result_ Timer.fired () {
            call ADC.getData ();
            return SUCCESS;
    }
    event result_t ADC.dataReady (uint16_t data) {
            sensorReading = data;
            …
            return SUCCESS;
    }
    …
}
```


nesC Programming Language

- StdControl.init is called at boot time
  - creates a timer that expires every 1000ms
  - upon time expiration, a new sensor sample is obained
  - ADC.getData triggers actual sensor data acquisition (ADC.dataReady)

nesC Programming Language

- Example of wiring subcomponents: timer service in TinyOS (TimerC)

```
configuration TimerC {
        provides {
                interface StdControl;
                interface Timer;
                }
}
implementation {
        components TimerModule, HWClock;

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

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

nesC Programming Language

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

nesC Programming Language

- nesC provides two options to ensure atomicity
    - convert all sharing code to tasks (SC only)
    - use atomic sections to modify shared state

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

- Nonpreemption can be obtained by disabling interrupts during atomic section (no call/signal allowed to ensure that atomic sections are brief)

TinyGALS

- Globally Asynchronous and Locally Synchronous (GALS)
- TinyGALS consists of modules composed of components
- A component C has
    - set of internal variables V_C
    - set of external variables X_C
    - set of methods I_C (that operate on these variables)
- Methods are further divided:
    - calls in the $ACCEPTS_C$ set (can be called by other components)
    - calls in the $USES_C$ set (needed by C and may belong to other components)

TinyGALS

- TinyGALS defines components using an interface definition and implementation (similar to nesC)

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


TinyGALS

- Implementation for the *DownSample* component (*_active* is an internal boolean variable that ensures that for every other *fire()* method, the component will call *fireOut()* with the same integer argument)

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


TinyGALS

- TinyGALS modules consist of components
- Module M is a 6-tuple

$M=(COMPONENTS_M, INIT_M, INPORTS_M, OUTPORTS_M, PARAMETERS_M, LINKS_M)$

$COMPONENTS_M$ … set of components of M

$INIT_M$ … list of methods of M's components

$INPORTS_M$ … inputs of the module

$OUTPORTS_M$ … outputs of the module

$PARAMETERS_M$ … set of variables external to the components

$LINKS_M$ … relationship between the method call interfaces and the inputs and outputs of the module

TinyGALS

- Modules are connected to each other to form complete TinyGALS system, where a system is a 5-tuple

  $S=(\text{MODULES}_S,\text{GLOBALS}_S,\text{VAR\_MAPS}_S,\text{CONNECTIONS}_S,\text{START}_S)$

  - MODULES_S … set of modules
  - GLOBALS_S … global variables
  - VAR_MAPS_S … set of mappings (map global variable to a parameter of a module)
  - CONNECTIONS_S … list of connections between module output and input ports
  - START_S … name of an input port of exactly one module (starting point for execution)

TinyGALS

- Highly structured architecture of TinyGALS can be exploited to automate the generation of scheduling and event handling code
  - frees developers from writing error-prone concurrency control code
- Code generation tools can automatically produce:
  - all necessary code for component links and module connections
  - code system initialization
  - code for start of execution
  - code for intermodule communication
  - code for global variables reads and writes
- Modules use message passing and are therefore decoupled from each other (easier independent development)
  - each message triggers scheduler and activate receiving module
  - TinyGUYS (Guarded Yet Synchronous) variables:
    - modules read global variables without delay (synchronously)
    - modules write global variables using buffer (asynchronously)
      - buffer size is 1 (i.e., last writing module wins)

Sensor Network Application Construction Kit

- SNACK consists of a configuration language, component and service library, and compiler
- Goals are:
  - to provide smart libraries that can be combined to form WSN applications
  - to simplify the development process
  - to be efficient
- It should be possible to write simple pieces of code such as:

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

Sensor Network Application Construction Kit

- Syntax of SNACK code:

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

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

Sensor Network Application Construction Kit

- SNACK library
  - variety of components for sensing, aggregation, transmission, routing, and data processing
  - several core components supported
    - *Network*: receives/sends messages from/to TinyOS radio stack
    - *MsgSink*: ends inbound call chains and destroys received buffers
    - *MsgSrc*: generates periodic empty SNACK messages and passes them to outbound interface
    - *Timing*:
      - *TimeSrc*: generates a timestamp signal sent over signal interface at specified minimum rate
      - *TimeSink*: consumes such signals
    - *Storage*: implemented by components such as Node-Store64M, which implements an associative array of eight-byte values keyed by node ID
    - *Service*: variety of services (e.g., *RoutingTree* implements a tree designed to send data up to some root)

Thread-Based Model

- Multiple tasks allowed to make progress in execution without concern that a task may block other tasks (or be blocked) indefinitely
- Task scheduler manages task execution
  - example: time-slicing approach, where tasks execute for certain amount of time
- MANTIS (MultimodAl system for NeTworks of In-situ wireless Sensors)
  - thread-based operating system for WSNs
  - memory-efficient
    - requires less than 500 bytes of RAM
    - 14 kbytes of flash memory
  - energy-efficiency
    - microcontroller switches to low-power sleep state after all active threads have called the *sleep()* function

Thread-Based Model

- TinyThread
  - adds support for multithreaded programming to TinyOS and nesC
  - procedural programming of sensors
  - includes suite of interfaces that provide blocking I/O operations and synchronization primitives
- Protothreads
  - lightweight stackless type of threads; all protothreads share the same stack
  - context switch is done by stack rewinding
  - variables must be saved before calling a blocking wait (variables with function-local scope that are automatically allocated on stack are not saved across wait calls)
- Y-Threads
  - preemptive multithreading (distinguish preemptable from nonpreemptable code)
  - shared stack for nonblocking parts, thread-specific stack for blocking calls
    - blocking portions of code require only small amounts of stack, leading to better memory utilization compared to other preemptive approaches

Macroprogramming

- Focus on programming group of nodes, instead of individual nodes
- Abstract Regions
  - focuses on group-level cooperation
    - group of nodes working together to sample, process, and communicate sensor data
  - region-based collective communication interface
  - defines neighborhood relationship between nodes
    - *"the set of nodes within distance d"*
  - type of definition of abstract region depends on the type of application
  - examples of implementations of abstract regions
    - N-radio hop (nodes within N radio hops)
    - k-nearest neighbor (k nearest nodes within N radio hops)
    - spanning tree (rooted at a single node, used for data aggregation over entire network)

Macroprogramming

- Abstract Region (contd.)
  - example:
    - regions defined using hop distances
    - discovery of region members using periodic broadcasts (advertisements)
    - data can be shared between region members using a "push" (broadcasting) or "pull" (issue a fetch message) approach

Macroprogramming

- EnviroTrack
  - object-based middleware library
  - geared toward target-tracking sensor applications
  - free developer from details of
    - interobject communication
    - object mobility
    - maintenance of tracking objects and their state
  - also uses the concept of groups, which are formed by sensors which detect certain user-defined entities in the physical environment
  - groups are identified by context labels (logical addresses that follow the external tracked entity around in the physical environment)

Macroprogramming

- EnviroTrack (contd.)
  - tracking objects: objects can be attached to context labels to perform context-specific operations; executed on the sensor group of the context label
  - type of context label depends on entity (e.g., context label *car* is created wherever a car is tracked)
  - context label of some type *e*:
    - function $sense_e()$: describes sensory signature identifying tracked environmental target (car: magnetometer and motion sensor readings)
      - also used to track group membership
    - aggregation function $state_e()$: environmental state shared by all objects attached to a context label
      - acts on the readings of all sensors for which $sense_e()$ is true
    - aggregation is performed by sensor node acting as group leader

Macroprogramming

- Database approaches treat entire WSN as a distributed database that can be queried
- TinyDB
  - network is represented logically as a table (called *sensors*)
    - one row per node per instant in time
    - each column corresponds to sensor readings (light, temperature, pressure, …)
    - new record is added when a sensor is queried
    - new information is usually stored for a short period of time only
  - queries are like SQL-based queries (SELECT, FROM, WHERE, etc.)

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

    - initiates data collection at beginning of each epoch (specified in SAMPLE PERIOD clause); results are streamed to the root of the network

Macroprogramming

- TinyDB (contd.)
  - supports group aggregation queries
  - example: monitoring microphone-equipped rooms; look for rooms where the average volume is over a certain threshold:

```
SELECT AVG(volume), room FROM sensors
   WHERE floor = 6
   GROUP BY room
   HAVING AVG(volume) > threshold
   SAMPLE PERIOD 30s
```

- Similar projects
  - Cougar: resource-efficient database appraoch
  - SINA: models WSN as collection of distributed objects; supports SQTL scripts in SQL queries
  - MiLAN: sensor applications can specify QoS needs

Dynamic Reprogramming

- Sometimes necessary to disseminate code to all or some sensor nodes
- Virtual machines
  - Maté
    - small VM on top of TinyOS
    - capsules (sequence of 24 instructions) inside a single TinyOS packet
    - every capsule includes type and version information
      - *message send* capsules
      - *message receive* capsules
      - *timer* capsules
      - *subroutine* capsules
    - programs execute in response to events (e.g., timer firing, packet being sent/received)
    - each event has a capsule and an execution context

Dynamic Reprogramming

- Virtual machines (contd.)
  - Maté (contd.)
    - jumps to first instruction of capsule and executes until **halt** instruction
    - when subroutine called, return address is pushed onto a return address stack; upon return, address is taken from the stack
- Trickle
  - controlled flooding protocol for disseminating small pieces of code
  - uses metadata to describe code (allows node to determine if code update needed)
  - metadata is exchanged among neighbors via broadcast
    - periodic time intervals, each node randomly selects broadcast time during each interval
  - when a node hears outdated metadata, it broadcasts its own code, giving outdated node chance to update
  - when a node overhears newer metadata, it broadcasts its own metadata, triggering the neighbor to broadcast newer code

Dynamic Reprogramming

- Melete
  - similar to Maté and Trickle
  - supports multiple concurrent applications
  - supports selective dissemination by limiting dissemination range
    - code is only forwarded within a forwarding region
- Deluge
  - occasionally advertises the most recent code version using broadcasts
  - if a node receives an update with old code, it responds with new code version (allowing neighbor to request new code)
  - eliminates redundant advertisements and request messages
  - provides robustness
    - uses a three-phase handshake to ensure that only symmetric links are used
    - allowing a node to search for a new neighbor to request code if it has not completely received the code after *k* requests
  - dynamically adjusts rate of advertisements for quick propagation when needed, but consuming few resources in steady state

Dynamic Reprogramming

- Pump Slowly, Fetch Quickly (PSFQ)
  - slowly pace propagation of packets (pump slowly)
  - aggressively fetch lost packets (fetch quickly)
  - nodes do not relay packets out of order
  - prevents loss events from propagating downstream
  - *localized recovery* allows nodes to recover lost packets from immediate neighbors (reduces recovery costs)
- Push Aggressively with Lazy Error Recovery (PALER)
  - based on observation that pushing data downstream and recovering lost packets simultaneously leads to excessive contention
  - eliminates in-order reception requirement
  - pushes all data aggressively
  - nodes keep list of missing packets and request retransmission after the broadcast period
  - retransmission requests are handled by neighbors (if they don't have a copy of missing data, they issue their own request to their neighbors)

Sensor Network Simulators

- Large scale of sensor networks makes implementation and experimentation difficult and expensive
- Instead, simulation is used to evaluate novel WSN tools, mechanisms, protocols, and applications
- Quality of simulations depends on choice of appropriate models for
  - sensor node hardware/software characteristics
  - wireless communication
  - physical environment
  - node mobility
- Simulators typically come with tools for collecting, analyzing, and visualizing sensor data

ns-2

- Discrete event simulator called network simulator (ns-2)
- Written in C++ and Otcl
- Highly extensible, many extensions have been developed, e.g.:
  - extension adding the concept of a phenomenon (physical event)
    - uses broadcast packets over designated channel to represent physical phenomena (fire, moving vehicle, chemical cloud)
    - uses PHENOM routing protocol: emits packets with certain configurable pulse rate and whose arrival triggers a receive event
  - many routing protocols
  - many MAC-layer protocols
  - variations of packet contents
  - models for multi-homed devices
  - mobility models

GloMoSim and QualNet

- GloMoSim
  - based on the PARSEC (PARallel Simulation Environment for Complex systems) simulation environment
    - C-based simulation language
    - represents sets of objects in physical environment as logical processes
    - represents interactions among these objects as time-stamped message exchanges
  - supports variety of models at different protocol layers
  - supports different mobility models
  - intended for academic use
- QualNet
  - commercial version of GloMoSim
  - produced by Scalable Network Technologies, Inc.

JiST/SWANS

- Java in Simulation Time (JiST)
  - discrete event simulator
  - efficient
    - run program in parallel
    - dynamically optimize simulator configuration
  - transparent
    - transform simulation programs automatically to run with simulation time semantics (instrument simulations such that no programmer intervention or calls to specialized libraries are needed to support concurrency, consistency, reconfiguration, etc.)
- Scalable Wireless Ad hoc Network Simulator (SWANS)
  - built on top of JiST
  - collection of independent SW components that can be aggregated to form complete wireless (ad hoc) simulations

OMNeT++

- Objective Modular Network Testbed
  - discrete event simulator for simulating communication networks, multiprocessors, and distributed systems
  - open-source based on C++
  - models consist of modules that communicate using message passing
    - simple modules and compound modules
  - uses topology description language NED to define structure of a module
  - includes graphical editor
  - lacks protocol models

TOSSIM

- Simulator for TinyOS-based networks
- Generates discrete event simulations directly from TinyOS components (i.e., runs the same code that runs on the sensors)
- Replaces low-level components (e.g., interrupts) with events in the simulations
- Simulator event queue delivers these events to components
- Works at bit level, i.e., event is generated for each sent or transmitted bit
  - allows for experimentation with low-level protocols
- TinyViz: visualization tool
- Very scalable and extensible
- Lacks energy profiling and use is limited to TinyOS systems

EmStar

- Targeted at high capability nodes called microservers (e.g., cluster heads)
- Consists of a Linux microkernel extension, libraries, and several tools
- EmSim: operates many virtual nodes in parallel in a simulation that models radio and sensor channels
- EmCee: runs the EmSim core and is an interface to real low-power radios
- EmView: graphic visualizer

Avrora

- Flexible simulator framework in Java
- Each node is its own thread and code is executed instruction-by-instruction
- Event queue:
  - targets nodes operating in long sleep modes
  - event queue takes advantage of that to boost performance
  - when node sleeps, only a time-triggered event that causes an interrupt can wake up the node
  - such an event is inserted into event queue of the node to be woken up at a certain time
  - simulator processes events in order until one of them triggers a hardware interrupt, which re-awakes a node
- Fast and scalable simulator; can simulate down to the level of individual clock cycles