SPINE
Signal Processing In Node Environment

Marco Avvenuti
Pervasive Computing & Networking Lab. (PerLab)
Dept. of Information Engineering
University of Pisa
m.avvenuti@iet.unipi.it
Wireless Body Sensor Networks (BSN)

Enable continuous and real-time medical assistance at low cost.

Health-care applications based on BSN include:

- early detection or prevention of diseases,
- elderly person assistance at home,
- e-fitness, rehabilitation after surgeries,
- motion and gestures detection,
- cognitive and emotional recognition,
- medical assistance in disaster events
Available approaches for BSN application development

<table>
<thead>
<tr>
<th></th>
<th>Application-specific code</th>
<th>Domain-specific framework</th>
<th>General-purpose middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Reusability</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Rapid prototyping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ease of debugging</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Code efficiency</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>System interoperability</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Specific support to flexible sensing operations</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Specific support to in-node signal processing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The prototyping of BSN applications can be significantly facilitated by a domain-specific framework with libraries and protocols that allow to implement signal processing tasks efficiently in star-topology networks.
What is SPINE?

A domain-specific open source framework for the design and fast prototyping of Wireless Body Sensor Network (BSN) applications.
Enables efficient implementations of signal processing algorithms for analysis and classification of sensor data through libraries of processing functionalities.
It also embeds an application-level communication protocol.
Signal Processing In Node Environment (SPINE)

INTERNET Cloud Computing

Coordinator Device (e.g. Smart-Phone)

SPINE Framework

Heart Rate and ECG

Blood Pressure, EMG, GSR, PPG, SpO2

Motion (Accelerometer, Gyroscope)

Pressure, FSR

System Software for Ambient Intelligence
SPINE is organized in two interacting macro-components, which are respectively implemented on commercially available:

- TinyOS-enabled sensor platforms (Shimmer rev. 1.3 and 2R; TelosB/Tmote Sky; MicaZ).
- personal coordinator (such as an Android smart-phone or tablet, or a personal computer).

Communication among these devices is wireless, using the Bluetooth 2.1 or IEEE 802.15.4 standards.
Physical Sensors

Natively supported by SPINE (some only on specific sensor platforms):

- Accelerometers and gyroscopes, force and pressure (for postural, gesture and activity monitoring);
- Electro-cardiogram sensor (ECG) (for cardiac monitoring);
- Electro-impedance-plethysmographic sensor (EIP) (for respiratory rhythm and volume);
- Electromyographic sensor (EMG) (for muscular activity);
- Photo-plethysmographic sensor (PPG) and SpO2 (pulse oximetry and blood oxygen saturation, for indirect measurement of heart-rate and breathing);
- Galvanik Skin Response sensor (GSR) (e.g. for emotion recognition);
- Environmental temperature, humidity sensors;
- Light sensor.
SPINE's API

Dynamic and flexible configuration of sensing and processing functionalities available at the sensor node level (coordinator level Java API).

Many biophysical sensors and signal processing functionalities are natively implemented and available to application developers.

The application developer can perform data collection of the raw signals from one or multiple available sensors (even simultaneously). It is possible to configure dynamically sensor parameters such as the sampling frequency (from less than 1Hz up to about 200Hz).

New, custom-defined sensor drivers and processing functionalities can be easily integrated into the SPINE framework.
## Common tasks supported by BSN applications at node-side

<table>
<thead>
<tr>
<th>TASK</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLING</td>
<td>The sensor sampling process represents the first step for developing a BSN application. Selecting the appropriate sampling time to satisfy the application requirements is important because the amount of data generated and processed, and under certain degree the energy consumed depend on it.</td>
</tr>
<tr>
<td>FEATURE EXTRACTION</td>
<td>Classifier algorithms very rarely use raw data. Instead, attributes (or features) are typically extracted on data windows and used to detect events and classify activities. Extracting features directly on the wireless nodes also allows the reduction of the radio usage, as aggregated results are sent instead of several raw data values.</td>
</tr>
<tr>
<td>QUERIES</td>
<td>Support for selective queries on the available sensors of a node is important because application requirements can change over time and not all the sensors are necessarily involved for algorithms execution at any time.</td>
</tr>
<tr>
<td>NODE SYNCHRONIZATION</td>
<td>In a WBSN, nodes should be kept synchronized when sampling the sensors and processing data, because data gathered from multiple wearable nodes must refer to the same time interval to be aggregated to recognize correctly e.g. physical activities or other events of interest.</td>
</tr>
<tr>
<td>DUTY CYCLING</td>
<td>Duty cycling is a mechanism for handling the radio status (idle, on, off) to reduce power consumption of a sensor node and therefore its battery lifetime. In particular, radio duty cycling must be tuned very carefully, reducing as much as possible the active time (transmitting, receiving, and listening).</td>
</tr>
</tbody>
</table>
Signal Processing

Sensor nodes can be configured to perform simple pre-processing operations to user-selectable time windows of the sensor signals, for instance to reduce the amount of data the sensor nodes has to transmit over-the-air back to the coordinator.

Specifically, SPINE provides native support to several general-purpose processing functions, including:

- Average, Median
- RMS (Root Mean Square):
  \[ \sqrt{ \frac{1}{n} \left( x_1^2 + x_2^2 + \cdots + x_n^2 \right) } \]
- Max and Min;
- Amplitude;
- Variance and standard deviation;
- Zero-crossing;
- Entropy \( H(X) = - \sum_{i=1}^{n} \{p(x_i) \log p(x_i)\} \)
BSN research prototypes

As a middleware for the development of BSN applications, SPINE has been adopted to prototype several BSN research systems, including:

- Physical rehabilitation assistant;
- Fall detector;
- Physical activity monitor;
- Smart step-counter;
- Mental stress detector;
- Handshake detection system.
SPINE 1.2 Tutorial

Thanks to: SPINE Team http://spine.tilab.com
SPINE Overview

- SPINE is a software Framework for the design of Body Sensor Network (BSN) applications
  - Open Source
    - available to the research community under LGPL license
  - High-level abstractions
    - Includes libraries of features, utilities and protocols for dynamic configuration of capabilities of BSN nodes via an OTA protocol
  - Interoperability through APIs
    - lightweight Java APIs that can be used by local and remote applications to manage the sensor nodes or issue service requests
  - Distributed implementations of classification algorithms
    - Enables efficient implementations of signal processing algorithms and allows to decrease development time
- Available for Download at http://spine.tilab.com
SPINE Framework Architecture

- **BSN Network**
  - Sensor nodes
  - Coordinator node that manages the BSN, collects/analyzes the data from the sensor nodes, performs gateway functions to connect the BSN with wide area networks for remote data access

- **SPINE Software Architecture**
  - Utilities for signal processing such as data storage buffers
  - Mathematical function libraries and common feature extractors used in signal processing
  - Over-the-air communication protocol to transfer data with the gateway
  - Java-based interface that can be used by an application running on the gateway or on a remote server to manage the sensor nodes or make service requests.
Sensor Node Architecture

- Each node can include multiple sensors
- SPINE libraries and service management currently implemented in TinyOS environment
- Supported TinyOS hardware platforms: Telos, MicaZ, Shimmer
SPINE Functional Architecture

- Application Layer
  - Application logic at the coordinator or at a remote server
- Service Layer
  - Data Measurement Service (manages sensors and data buffering)
  - Classification Service (computes features, classifies data)
- Transport Layer
  - End-to-end protocol between coordinator and sensor nodes
  - Radio Controller: defines duty cycle based on data availability
Network Architecture

- Star topology including a Coordinator and multiple body or environmental Sensor Nodes
- Easy to extend with multi-hop and point-to-point sensor node communication
- Parameters: number of nodes, transmission rate, protocol
SPINE network

- Communication nodes ↔ coordinator
- On node sensor management
- Framework announce its capabilities
- Runtime function/sensor activation

BSN

GW

SPINE node side

SPINE server side

TinyOS2.x code

Java code
Introduction to SPINE1.2: what’s new – node side

- **Communication**
  - takes care of radio communication, data packet build and parse, radio duty cycle and channel access schemes;

- **Sensing**
  - manages data sampling from sensors and storage on a shared buffer;

- **Processing**
  - takes care of the on node processing functionalities
Radio Controller
- access to the radio
- Optional low power mode and TDMA

Packet Manager
- Build and parse SPINE packets
- Fragments if necessary

InPacket
- Generic i/f to be implemented by pkts incoming into the nodes (parse method)

OutPacket
- Generic i/f to be implemented by pkts outgoing from the nodes (build method)
- Sensor Board Controller
  - Manages all on-board sensor

- Sensor Registry
  - Register sensors present on the board

- Buffer Pool
  - For storing data sampled form the sensor to be then elaborated by the functions.
Function Manager: dispatcher among functions

Function i/f to be implemented by all supported functions

commands sent to the FM

- setUpFunction
- Act/deactFunction
- Start

FM to coordinator
- data packet
The lifecycle of **SPINE1.2** framework

Acts as a dispatcher among the aforementioned modules and in particular it provides the following services:

- Service Discovery management
- Dispatch incoming packet depending on their types:
  - `SetUpSensor` messages to the Sensing modules (Sensor Board Controller);
  - `SetUpFunction` and `ActivateFunction` messages to the Processing modules (Function Manager)
- Handle start and reset message (sync message still need to be supported)
The core implementation does not use any TinyOS specific APIs and can be run on any kind of network (e.g., ZigBee networks).

Platform-independent code may be found into:

- `spine`: contains SPINE core logic
- `spine.datamodel`: contains data structures used by the framework

At this stage, SPINE1.2 server side provides an implementation for TinyOS2.x network; therefore it provides the support for TinyOS low level communication:

- `spine.communication.tinyos` contains TinyOS specific logic (packet format and parse/build operations) and low level communication procedures (calling tinyos.jar APIs).
Test Application...what SPINE1.2 can do!!!

- SPINE Manager APIs to be used by the application
  - discoveryWsn()
  - setupSensor(int nodeID, SpineSetupSensor setupSensor)
  - setupFunction(int nodeID, SpineSetupFunction setupFunction)
  - activateFunction(int nodeID, SpineFunctionReq functionReq)
  - deactivateFunction(int nodeID, SpineFunctionReq functionReq)
  - readNow(int nodeID, byte sensorCode)
  - start(boolean radioAlwaysOn, boolean enableTDMA)
  - resetWsn()

- SPINE Listener methods
  - newNodeDiscovered(Node newNode)
  - serviceMessageReceived(int nodeID, ServiceMessage msg)
  - dataReceived(int nodeID, Data data)
  - discoveryCompleted(Vector activeNodes)
Communication: Radio Controller and Packet Manager

- **Radio Controller**
  - Access to the radio
  - Optional low power mode and TDMA
  - Radio access independent on the SPINE framework

- **Packet Manager**
  - Build and parse SPINE packets (payload and header)
  - Fragment if necessary
Radio Controller - radioAlwaysOn

- Configurable both at compile and run time
- **Compile-Time** (node side, Makefile)
  - PFLAGS += -DBOOT_RADIO_ON=TRUE/FALSE
- **Run-Time** (coordinator side, Java)
  - SPINEManager.start(boolean radioAlwaysOn, boolean enableTDMA)

Data pck to be sent

Radio ON to send  Radio ON to rx msg, if any

Radio OFF

Radio ON

Start msg received

Radio OFF

Data msg to be sent

Radio ON

Data pck sent

Any other pck to send?

Stay ON T=25msec

RX any pck?

YES

NO

YES

NO
Radio Controller - enableTDMA

- Configurable both at compile and run time
  - Compile-Time (node side, Makefile)
    - PFLAGS += -DENABLE_TDMA=TRUE/FALSE
    - PFLAGS += -DDTDMA_FRAME_PERIOD=...
  - Run-Time (Coordinator side, Java)
    - SPINEManager.start(boolean radioAlwaysOn, boolean enableTDMA)

TDMA is enabled
- The node will allocate to its transmission a time slot that depends on its own ID and the total number of nodes into the network at that time.
- If the radio is also duty-cycling, it’s kept off if for a given slot there aren’t msgs to send

TDMA is disabled
- The node will send packets as soon as they are requested to be sent using the default radio access scheme (ex. CSMA-CA)

If users wish to use TDMA communication they must flash nodes with sequential IDs (1,2,...)
- By adding joining procedure and nodes re-addressing it’s possible to remove this constraint
Packet Manager

- Get/Send packets from/to the Radio Controller
- Parse/build SPINE messages payload and header
- Implements Fragmentation
  - messages too big are transparently split into multiple fragments and eventually reassembled inside the coordinator
  - Fragmentation is currently “one-way” from nodes to the coordinator
- Parameterized interfaces for incoming and outgoing SPINE packets for simplifying addition of new packets
- Filters not-SPINE packets or bad/corrupted ones
HOW TO... use SPINE1.2 communication protocol

- management messages FROM the coordinator TO the nodes
  - **Network Discovery**: is an empty packet, sent as broadcast so that can reach all the nodes into the network;
  - **Set Up Sensor**: contains the sampling time settings needed to set the sensor
  - **Set Up Function**: contains the general settings that are needed by a function. Parameters list is basically a list of bytes so that can be very generic and used for different functions.
  - **Activate/Deactivate Function**: contains details on the functionalities to be activated. As above, parameter list is a generic array of bytes.
  - **Start Network**: is a broadcast packet that works as trigger to start the computation and the sampling on the nodes. It contains information about the total number of nodes present into the network (used if TDMA is enables) as well as flags for radio behaviors (duty cycling and TDMA).
HOW TO... use SPINE1.2 communication protocol

- Management messages FROM the coordinator TO the nodes
  - **ReadNow**: is a unicast packet for easy-to-use an immediate one-shot sensor reading
  - **Reset Network**: is a broadcast packet that forces all the nodes of the network to a complete software reset
  - **Synchronize Network**: is a broadcast packet for synchronizing nodes local clock. Logic is NOT implemented yet
HOW TO... use SPINE1.2 communication protocol

- Management messages FROM the nodes TO the coordinator
- The high level API generates user-friendly events from lower level SPINE packets:
  - `newNodeDiscovered` is issued whenever a new SPINE node is discovered
  - `discoveryCompleted` is issued as the nodes discovery procedure ends
  - `serviceMessageReceived` forwards a service message (errors, warnings, ... ) generated by a certain node
  - `dataReceived` reports objects representing particular (one-shot raw data, feature values, threshold-based alarms, ... ) function-specific data generated by a certain node
Sensing: Sensor Board Manager, Buffer Pool and Sensor interface

- **Sensor Drivers** must implement Sensor i/f
- **Sensor Board Controller** takes care of setting up sensors (Sampling Time), sample sensors and store readings into buffer pool
- **Sensor Registry** for mounted sensors retrieval
- **Buffer Pool** is a general purpose storage manager, built as single array logically divided into multiple indexed circular buffers
HOW TO... add a new sensor

- Platform independent MyNewSensorC.nc implementation into 
  tos\system\sensing that links to the driver implementation into 
  tos\sensorboards\myboard (or into tos\platforms\myplatform)

- Driver implementation HilMySensorC/P.nc has to implement the 
  tos\interfaces\sensing\Sensor.nc interface and register its 
  code to the tos\system\sensing\SensorRegistry.nc

- Define the new sensor platform into the spine.extra makefile

- Add new sensor wiring in to the 
  tos\system\sensing\SensorBoardController.nc

- Update constants both node and server side

- The new sensor is now fully integrated into and all the SPINE services 
  can be applied to data coming form myNewSensor.
Processing: Function Manager

- New function must implement the Function i/f
- Function take data from the buffer pool, analyze it and send result back to the coordinator
- Function are set with:
  - setUpFunction
  - activateFunction
- Function can be deactivated
  - Deactivate Function
- Data coming from the function = array of bytes
Feature Engine

- **What it is:** The Feature Engine provides periodic calculation of simple features on sensed data
  - A server app requests a feature using two setup messages:
    - Set a sampling time for a given sensor if necessary
    - Associate a window size and shift size with a given sensor
    - Request the features desired for that sensor
  - Features include MIN, MAX, MEAN, AMPLITUDE, ...
  - Can be calculated over multiple channels
    - MEAN calculates over each active channel
    - VECTOR_MAGNITUDE calculates over all channels and returns a single value
  - A feature is calculated every shift size samples, over a buffer of length window size
Feature Engine

- How it works: The feature engine keeps a table of requested features, and listens for sampling events to trigger feature calculation and sending.
  - A buffer for each active sensor channel is managed by the SensorBoardController.
  - The FeatureEngine organizes data in these buffers so that a feature may be calculated, then generates and sends a message to send calculated features to the base station.
  - Features are implemented as tinynos components conforming to the Feature interface, allowing new Features to be easily introduced.
    - A single feature can calculate across channels but not across multiple sensors.
HOW TO... introduce New Feature

- Node Side:

1. In $\text{Spine_nodes1}_2/\text{tos/}\text{types/}\text{Functions.h}$ add $\text{MY_NEW_FEATURE}$ code into enum $\text{FeatureCodes}$
2. Into $\text{tos/system/processing}$ you have to implement the feature logic. For this reason you’ll create 2 new files: $\text{MyNewFeatureC.nc}$ and $\text{MyNewFeatureP.nc}$.
MyNewFeature module will have to provide the Feature interface (look into $\text{tos/interfaces/processing/Feature.nc}$), meaning it will have to implement the $\text{.calculate}$ and the $\text{.getResultSize}$ commands.
   a. $\text{Feature.calculate}$ command does the actual feature implementation over a certain
   b. $\text{Feature.getResultSize}$ command simply returns the number of byte the resulting feature is composed of.
Remember that, since the Boot interface is used, you’ll have to implement the event $\text{Boot.booted()}$. We suggest that you register the feature to the FeatureEngine at this time.

```java
event void Boot.booted() {
    if (!registered) {
        // the feature self-registers to the FeatureEngine at boot time
        call FeatureEngine.registerFeature(MY_NEW_FEATURE);
        registered = TRUE;
    }
}
```

3. Finally, wire your new Feature into the FeatureEngine. Open $\text{tos/system/processing/FeatureEngineC.nc}$ and add the following lines:
   ```java
   components MyNewFeatureC;
   FeatureEngineP.Features[MY_NEW_FEATURE] -> MyNewFeatureC;
   ```
HOW TO... introduce New Feature

Server Side:

- The server need only
- Open `spine.SPINEFunctionConstants.java` and add the following lines that define the new feature:

```java
public static final byte MY_NEW_FEATURE =

public static final String MY_NEW_FEATURE_LABEL = "MyNewFeature"

case MY_NEW_FEATURE: return MY_NEW_FEATURE_LABEL;
```
Alarm Engine – what it is

- The Alarm Engine provides an event notification whenever function values are outside input thresholds
- Alarms may be set on raw data as well as functions
- Available alarm types:
  - Above threshold
  - Below threshold
  - In between thresholds
  - Outside thresholds
Alarm Engine – how to set it up

- Alarm may set up by coordinator (window and shift)
  
  ```java
  AlarmSpineSetupFunction ssf = new AlarmSpineSetupFunction();
  ssf.setSensor(sensor);
  ssf.setWindowSize(WINDOW_SIZE);
  ssf.setShiftSize(SHIFT_SIZE);
  manager.setupFunction(curr.getNodeID(), ssf);
  ```

- Alarm may be activate by coordinator (Sensor type and channel Data type Thresholds Alarm type)
  
  ```java
  AlarmSpineFunctionReq sfr = new AlarmSpineFunctionReq();
  sfr.setDataType(SPINEFunctionConstants.RAW_DATA);
  sfr.setSensor(SPINESensorConstants.ACC_SENSOR);
  sfr.setValueType((SPINESensorConstants.CH1_ONLY));
  sfr.setLowerThreshold(lowerThreshold);
  sfr.setUpperThreshold(upperThreshold);
  sfr.setAlarmType(SPINEFunctionConstants.ABOVE_Threshold);
  manager.activateFunction(curr.getNodeID(), sfr);
  ```

- Node will send back alarm data only when the event occurs, reporting
  
  - Sensor type and channel
  - Data type
  - Alarm Type
  - Value that caused the alarm
HOW TO... add a new function - NODE SIDE

- MyNewFunction logic must be implemented into tos/system/processing
- MyNewFunctionEngineP.nc must implement the tos/interface/processig/Function.nc i/f and register itself to the function manager
- Function.nc provides setUpFunction, activateFunction, disableFunction, getFunctionList, startComputing and stopComputing commands
- The module has to also implement 3 events: boot.booted, functionmanager.sensorWasSampledAndBuffered and BufferPool.newElem.
- Parameters into SetUpFunction, activateFunction and data packets must be set according to the function logic
- Wiring must be added to the FunctionManager and constants must be defined
HOW TO... add a new function – SERVER SIDE

- **SetUp Function**
  - Function code
  - param length
  - param list

  `spine.communication.tinyos` (TinyOS SPINE communication)
  - NewFunctionSpineSetupFunction.java implement the tinyos encoding logic and extends the SpineSetupFunction.java abstract class

- **Activate/Deactivate Function**
  - Function code
  - Act/deact
  - param length
  - param list

  `spine.communication.tinyos` (TinyOS SPINE communication)
  - NewFunctionSpineFunctionReq.java implement the tinyos encoding logic and extends the SpineFunctionReq.java abstract class
HOW TO... add a new function – SERVER SIDE

DATA

<table>
<thead>
<tr>
<th>bit</th>
<th>8</th>
<th>8</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Function code</td>
<td>data length</td>
<td>data</td>
</tr>
</tbody>
</table>

**spine.datamodel** (platform independent data structures)

- **MyFunctionData.java**: decode byte array ➔ **MyFunction** object
  - **MyFunction.java**: define MyFunction entity (constructor, toString and getters)

**spine.communication.tinyos** (TinyOS SPINE communication)

- **MyFunctionSpineData.java**: method for parse (decompress TinyOS SPINE MyFunction data pck). Called by SpineData class through dynamic class loading