



The SPINE Manual Version 1.3

The SPINE Team

February 2010



1	Introduction	3
1.1	How to install SPINE	5
1.1.1	Node Side.....	5
1.1.2	Server Side.....	6
2	How to use SPINE	7
2.1	How to run a simple application using SPINE1.3	8
2.2	SPINE Logging capabilities	12
3	SPINE architecture	13
3.1	Communication.....	14
3.2	Sensing	18
3.2.1	Sensors now supported.....	18
3.2.2	How to introduce a new sensor	19
3.3	Processing.....	21
3.3.1	Feature Engine.....	23
3.3.2	Alarm Engine.....	26
3.3.3	Step Counter	28
3.3.4	Buffered Raw Data	28
3.3.5	HMM.....	29
3.3.6	How to add a new function	30
3.3.7	How to plug-in processing functions into SPINE	32
3.3.8	How to enable a secure communication: SECURE SPINE	33
4	Applications	34
4.1	HOWTO port a SPINE1.2 application to SPINE1.3	34
4.2	SPINE1.3 apps.....	35
4.3	How to contribute to SPINE1.3 apps	36
5	Data Collector and SPINE Node Emulator	37
5.1	Data Collector application.....	37
5.1.1	Dataset file syntax	37
5.1.2	Data Collector GUI and functionality.	43
5.2	SPINE Node Emulator application.....	50
5.2.1	SPINE Node Emulator GUI and functionality.....	51
5.2.2	Virtual sensor node: functionality and sensors.	53
5.2.3	Use case: TestGUI for testing “virtual sensor node”	54



1 Introduction

SPINE (*Signal Processing In Node Environment*) is a framework for the distributed implementation of signal processing algorithms in wireless sensor networks.

It provides a set of on node services that can be tuned and activated by the user depending on application needs.

SPINE is released as Open Source project under LGPL 1.2 license and is available on line at <http://spine.tilab.com>.

If you are willing to contribute to the Open Source project, please email spine-contrib@avalon.tilab.com by specifying a short work plan with a description of the new functionalities, the impact on existing code, and the expected time for the first stable release.

The SPINE framework has two main components

1. Sensor Node side. It is developed in TinyOS2.x environment and provides on node services such as sensor data sampling and storage, data processing and more;
2. Server side. It is developed in Java SE and acts as coordinator of the sensor networks. Therefore, it manages the network, set up and activate on node services depending on the application requirements and more.

The framework has been redesigned and the newest release (1.3) provides many more levels of expansibility than before.

The core framework is now organized into three main parts that take care of different aspects, namely the communication, the sensing and the processing parts.

The new structure reflects into the file organization, especially on the sensor node side (Figure 1.1).

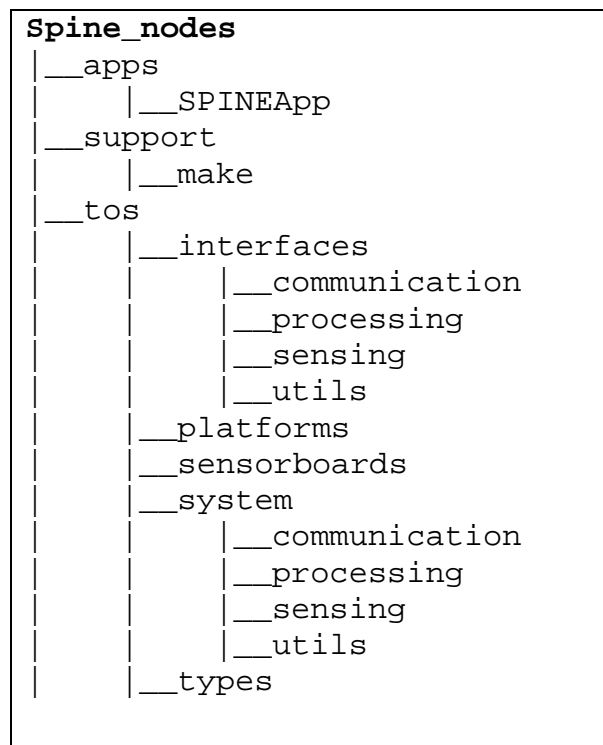


Figure 1.1 : Spine_nodes1_3 source code organization



The Server Side has a slightly different organization (Figure 1.2).

This structure reflects the need of having the framework logic not depending on the kind of network it is communicating with. In other words, the core implementation of SPINE does not use any TinyOS specific APIs and can be run independently on the underlying protocol stack (e.g. ZigBee networks).

Platform-independent code may be found into:

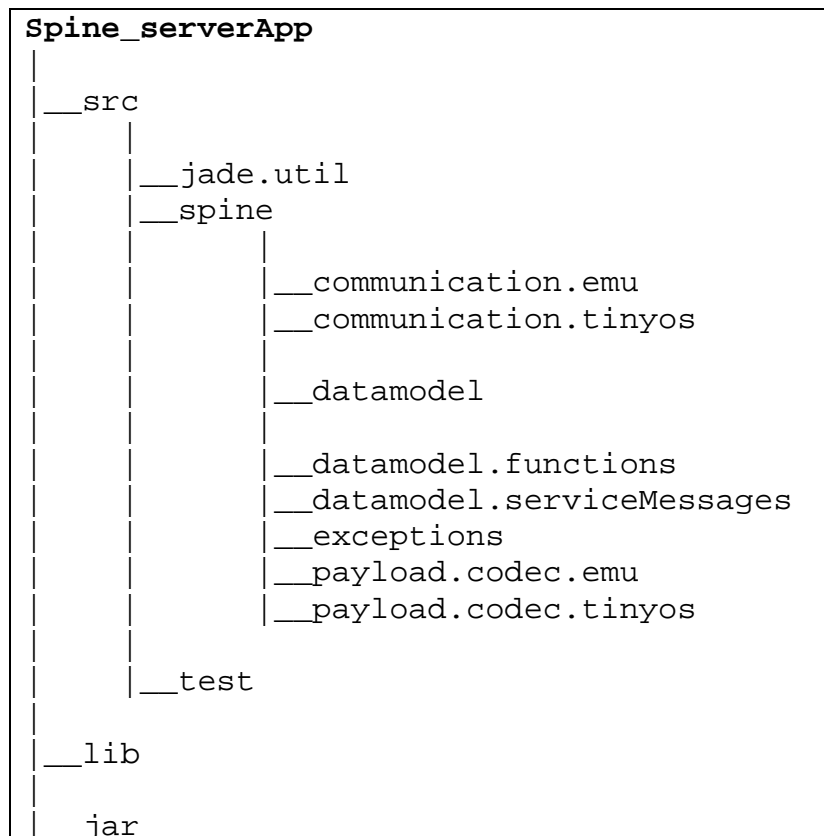
- `spine` package contains SPINE core logic
- `spine.datamodel` package contains data entities used by the framework
- `spine.datamodel.functions` sub-package defines the structure of the function
- `spine.datamodel.serviceMessages` sub-package defines various type of service messages
- `spine.exceptions` sub-package contains exception classes that might be thrown by SPINE

SPINE1.3 server side provides an implementation for TinyOS2.x network and for “virtual sensor node” network; therefore it provides the support for TinyOS low level communication:

- `spine.communication.tinyos` contains TinyOS specific logic and low level communication procedures (calling `tinyos.jar` APIs).
- `spine.payload.codec.tinyos` sub-package contains the low level messages codecs for the TinyOS platform.

and for “SPINE Node Emulator” (each “Node Emulator” instance is a “virtual sensor node”; see 5. Data Collector and SPINE Node Emulator) low level communication:

- `spine.communication.emu` contains logic and low level communication procedures for virtual sensor node.
- `spine.payload.codec.emu` sub-package contains the low level messages codecs for the virtual sensor node message.



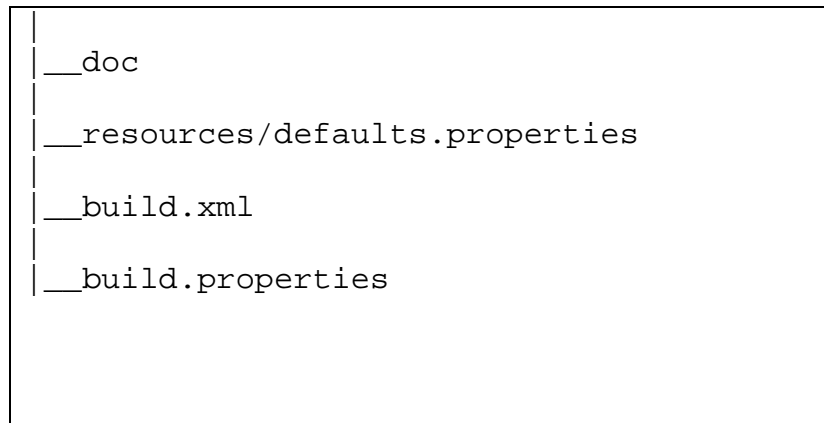


Figure 1.2: Spine_serverApp1_2 code organization

SPINE1.3 release provides also the SPINE.jar that can be imported in any project that uses SPINE APIs and the full javadoc documentation.

1.1 How to install SPINE

1. Download SPINE 1.3 from SPINE website (<http://spine.tilab.com/>).
2. The unzipped spine folder contains:
 - a. Spine_nodes folder with TinyOS2.x code to be run on the motes
 - b. Spine_serverApp folder with Java code to be run on a computer
 - c. COPYING and License text files containing info about the licensing
 - d. SPINE_manual

1.1.1 Node Side

Spine_nodes contains code to be compiled in TinyOS2.x and then flashed on sensor nodes. Spine_nodes 1.3 has been developed and tested with TinyOS version 2.1.0. Older TinyOS2.x versions have also been tested, and Makefile can be configured to support an older version, but the SPINE Team strongly suggests to use to TinyOS2.1.0 release.

1. Copy Spine_nodes folder into your tinyos-2.x-contrib folder
2. From the app/SPINEApp folder compile and install SPINE1.3 framework on your platform.
At this time, platforms supported by SPINE1.3 are
 - a. Telosb motes with spine sensor board
SENSORBOARD=spine make telosb
 - b. Telosb motes with biosensor sensor board
SENSORBOARD=biosensor make telosb
 - c. Telosb motes with the moteiv sensor kit
SENSORBOARD=moteiv make telosb
 - d. Micaz motes with mts300 board
SENSORBOARD=mts300 make micaz
 - e. shimmer motes
SENSORBOARD=shimmer make shimmer

Note that for each supported platform, a default SENSORBOARD has been defined. Therefore, unless differently specified (e.g. by defining the SENSORBOARD param in the make command):

- **telosb** defaults to "spine" sensorboard,
- **tmote** defaults to "moteiv" sensorboard,
- **micaz** defaults to "mts300" sensorboard,
- **shimmer** defaults to "shimmer" sensorboard.



To change these defaultings, the corresponding details can be found into the `tos/types/spine.extra` file.

If you want your specific sensor or mote platform to be supported by SPINE, please refer to 3.2.1 for further details and email spine-dev@avalon.tilab.com if you have specific questions.

1.1.2 Server Side

`Spine_serverApp` contains the Java code for running the server side (e.g. coordinator) of a SPINE network.

1. `src` contains SPINE1.3 source code organized into
 - a. `spine`
 - b. `jade`
 - c. `test`
2. `defaults.properties` contains the framework properties
3. `lib`: contains a jar file that SPINE must include
4. `docs`: contains SPINE1.3 javadoc documentation
5. `jar`: contains the framework jar file
6. `build.properties` and `build.xml` files for ant

You can compile and run SPINE framework and its test application either using textual ant commands or creating a java project using a IDE (such as Eclipse, NetBeans...).

Please note that you have to **add an external jar** (`tinyos.jar`) to your project. This jar is not part of the SPINE distribution and can be found in the `tinyos2.x\support\jdk\java` folder or downloadable at <http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x/support/sdk/java/>.

`tinyos.jar` should be placed into `spine_serverApp/ext-lib` folder.

For any problems and specific questions about the installation, please send e-mail to spine-dev@avalon.tilab.com



2 How to use SPINE

SPINE framework provides, on the Server Side, simple Java APIs to develop applications on the coordinator. Therefore, the main strength of the SPINE framework is to allow users to be ready to develop applications in sensor networks without bothering with node-side programming. Of course, SPINE Node Side source code is distributed as well and next paragraph will give details about it. Developers can easily form, manage and collect data from the sensors in the network writing a simple Java program: no more firmware programming is needed!

On the Java side, the user can develop its own application that will have to implement the SPINELListener interface and can use any of the API provided by the SPINEManager.

Since the application on the server side must implement the SPINELListener interface, it has to implement the following methods:

Method Summary	
void	received (Data data) This method is invoked by the SPINEManager to its registered listeners when it receives new data from the specified node. The Node object that generated this data is embodied into the data object.
void	discoveryCompleted (java.util.Vector activeNodes) This method is invoked by the SPINEManager to its registered listeners when the discovery procedure timer fires.
void	newNodeDiscovered (Node newNode) This method is invoked by the SPINEManager to its registered listeners when it receives a ServiceAdvertisement message from a BSN node
void	received (ServiceMessage msg) This method is invoked by the SPINEManager to its registered listeners when a ServiceMessage is received from a particular node. The Node object that generated this service message is embodied into the msg object.

Then, the application can use any APIs exposed by the SPINEManager:

Method Summary	
void	activate (Node node, SpineFunctionReq functionReq) Activates a function (or even only function sub-routines) on the given sensor.
void	addListener (SPINELListener listener) Registers a SPINELListener to the manager instance
void	bootUpWsn () Currently, it does nothing!
void	deactivate (Node node, SpineFunctionReq functionReq) Deactivates a function (or even only function sub-routines) on the given sensor.
void	discoveryWsn () Commands the SPINEManager to discovery the surrounding WSN nodes
void	discoveryWsn (long timeout) Commands the SPINEManager to discovery the surrounding WSN nodes within the given timeout
java.util.Vector	getActiveNodes () Returns the list of the discovered nodes as a Vector of spine.datamodel.Node objects
spine.datamodel.Node	getBaseStation () Returns the Node object representing the BaseStation
Jade.util.Logger	static getLogger ()



	Returns the static Logger of the SPINE Framework. The Logger can be used to set the logging level and to add custom log handlers (e.g. to log into a file).
<code>spine.datamodel.Node</code>	getNodeByLogicalID (<code>spine.datamodel.Address id</code>) Returns the node with the given logical address
<code>spine.datamodel.Node</code>	getNodeByPhysicalID (<code>spine.datamodel.Address id</code>) Returns the node with the given physical address
<code>void</code>	getOneShotData (<code>Node node</code> , <code>byte sensorCode</code>) Commands the given node to do a 'immediate one-shot' sampling on the given sensor.
<code>boolean</code>	isStarted () Returns true if the manager has been asked to start the processing in the wsn
<code>void</code>	removeListener (<code>SPINELListener listener</code>) removes a SPINELListener from the manager instance
<code>void</code>	resetWsn () Commands a software reset of the whole WSN.
<code>void</code>	setDiscoveryProcedureTimeout (<code>long discoveryTimeout</code>) This method sets the timeout for the discovery procedure.
<code>void</code>	setup (<code>Node node</code> , <code>SpineSetupFunction setupFunction</code>) Setups a specific function of the given node.
<code>void</code>	setup (<code>Node node</code> , <code>SpineSetupSensor setupSensor</code>) Setups a specific sensor of the given node.
<code>void</code>	startWsn (<code>boolean radioAlwaysOn</code>) Starts the WSN sensing and computing the previously requested functions.
<code>void</code>	startWsn (<code>boolean radioAlwaysOn</code> , <code>boolean enableTDMA</code>) Starts the WSN sensing and computing the previously requested functions.
<code>void</code>	syncWsn () Commands a software 'on node local clock' synchronization of the whole WSN.

The SPINEManager instance can be retrieved only via the SPINEFactory:

Method Summary	
<code>SPINEManager</code>	createSPINEManager (<code>String appPropertiesFile</code>) Initializes the SPINE Manager. The SPINEManager instance is connected to the base-station and platform obtained transparently from the app.properties file.

Examples about which function can be set, which data can be received and other details can be found in the SPINETest application which comes with the release and is illustrated in the following chapter. More examples about how to use the Java side are given all through this document.

For further details about the Java side, please refer to the Javadoc documentation that can be found in the release.

2.1 How to run a simple application using SPINE1.3

The SPINE1.3 release comes with a simple test application that can be easily run to experiment the framework basic functionalities.

Take the following steps:

1. compile and flash on your platform the SPINE1.3 node side framework;
2. compile and flash a TinyOS2.x BaseStation. Please check that sensor nodes and base station are both working on the same radio channel, have been compiled with the same max message payload length, and the same TinyOS version has been used for flashing all the nodes.



3. plug the BaseStation to your computer and type "motelist" from your shell: this will tell you your port number;
4. create an application properties file (e.g. under MyApp/resources/app.properties) and set the MOTECOM and the PLATFORM parameter according to one of the following options depending if you are using the serial forwarder on a Linux or Windows machine (eg. **a.**) or directly communicating with the serial port on your PC using a Windows machine (eg. **b.**), or if you intend to emulate a sensor node network (eg. **c.**), whose use is detailed in section 5.
 - a. MOTECOM=**sf@127.0.0.1:9002**
PLATFORM=**sf**
 - b. MOTECOM=**serial@COM41:telosb**
PLATFORM=**tinyos**
 - c. MOTECOM=**4444**
PLATFORM=**emu**

Option b may be used also on a Linux machine, but you need to build libgetenv and libtoscomm from your tinyos before you can install and run any SPINE application. Also, MOTECOM value would look like "**serial@/dev/ttyS0:telosb**".

```
cd $TOSROOT/support/sdk/java && make
sudo tos-install-jni
```

In a virtual sensor network (option c) MOTECOM value is set to NodeCoordinatorPort value (see [5.2.1 SPINE Node Emulator GUI and functionality](#)).

If needed, other application-dependent properties can be stored into this property file without any side effect to the SPINE framework.

5. edit Spine_serverApp/test/SPINETest.java and go through the code if you want to customize the test application. The code documentation helps to understand what functionalities SPINE exposes to the java developer.

As mentioned before, SPINETest.java implements the SPINEListener interface (to get notified of SPINE-related events) and uses the SPINEFactory to retrieve the SPINEManager, which, in turn, has the APIs for managing and communicating with the nodes in the network.

The SPINETest provided within the SPINE 1.3 release performs the following actions:

- a. a discovery message is broadcasted to check how the PAN is composed:

```
manager.discoveryWsn();
```

- b. when the discovery is completed, all the received info about nodes present in the PAN is displayed.

```
curr = (Node)activeNodes.elementAt(j);
// we print for each node its details (nodeID, sensors and functions
provided)
System.out.println(curr);
```

The information displayed at this point is:

- i. node id
 - ii. supported sensors
 - iii. supported functionalities
- c. if a node with an accelerometer is found:
 - i. the accelerometer is set with sampling time SAMPLING_TIME=50 msec

```
SpineSetupSensor sss = new SpineSetupSensor();
```



```
sss.setSensor(sensor);
sss.setTimeScale(SPINESensorConstants.MILLISEC);
sss.setSamplingTime(SAMPLING_TIME);
manager.setup(curr, sss);
```

- ii. the feature engine function is set on that node to work on data coming from the accelerometer sensor with window WINDOW_SIZE=40 and shift SHIFT_SIZE=20

```
FeatureSpineSetupFunction ssf = new FeatureSpineSetupFunction();
ssf.setSensor(sensor);
ssf.setWindowSize(WINDOW_SIZE);
ssf.setShiftSize(SHIFT_SIZE);
manager.setup(curr, ssf);
```

- iii. few features are activated on that node on the accelerometer data (MODE, MEDIAN, MAX and MIN on all the accelerometer's channels)

```
FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();
sfr.setSensor(sensor);
sfr.add(new Feature(SPINEFunctionConstants.MODE,
    ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask()));
sfr.add(new Feature(SPINEFunctionConstants.MEDIAN,
    ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask()));
sfr.add(new Feature(SPINEFunctionConstants.MAX,
    ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask()));
sfr.add(new Feature(SPINEFunctionConstants.MIN,
    ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask()));
manager.activate(curr, sfr);
```

- iv. more features are activated (MEAN,AMPLITUDE)

```
FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();
sfr.setSensor(sensor);
sfr.add(new Feature(SPINEFunctionConstants.MEAN,
    ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask()));
sfr.add(new Feature(SPINEFunctionConstants.AMPLITUDE,
    ((Sensor) curr.getSensorsList().elementAt(i)).getChannelBitmask()));
manager.activate(curr, sfr);
```

- v. the alarm engine function is set on the node to work on data coming from the accelerometer sensor with window WINDOW_SIZE=40 and shift SHIFT_SIZE=20. Please note that Feature and Alarm engines can be set with different settings, since they are two separate components. However, in this test application, they have been set with the same value to better check the results.

```
AlarmSpineSetupFunction ssf2 = new AlarmSpineSetupFunction();
ssf2.setSensor(sensor);
ssf2.setWindowSize(WINDOW_SIZE);
ssf2.setShiftSize(SHIFT_SIZE);
manager.setup(curr, ssf2);
```

- vi. two alarms are set on the accelerometer sensor, so that an alarm message will be sent back when:

1. the MAX value on CH1 is greater than upperThreshold value = 40
- 2.

```
AlarmSpineFunctionReq sfr2 = new AlarmSpineFunctionReq();
sfr2.setDataTypes(SPINEFunctionConstants.MAX);
```



```
sfr2.setSensor(SPINESensorConstants.ACC_SENSOR);
sfr2.setValueType((SPINESensorConstants.CH1_ONLY));
sfr2.setLowerThreshold(lowerThreshold);
sfr2.setUpperThreshold(upperThreshold);
sfr2.setAlarmType(SPINEFunctionConstants.ABOVE_THRESHOLD);
manager.activate(curr, sfr2);
```

3. the AMPLITUDE on CH2 is lower than lowerThreshold value = 2000

```
sfr2.setDataType(SPINEFunctionConstants.AMPLITUDE);
sfr2.setSensor(SPINESensorConstants.ACC_SENSOR);
sfr2.setValueType((SPINESensorConstants.CH2_ONLY));
sfr2.setLowerThreshold(lowerThreshold);
sfr2.setUpperThreshold(upperThreshold);
sfr2.setAlarmType(SPINEFunctionConstants.BELOW_THRESHOLD);
manager.activate(curr, sfr2);
```

- d. if a node with internal CPU temperature sensor is found:
 - i. the temperature sensor is set with sampling time OTHER_SAMPLING_TIME = 100 msec
 - ii. the feature engine function is set on that node to work on data coming from the temperature sensor with window OTHER_WINDOW_SIZE=80 and shift OTHER_SHIFT_SIZE=40
 - iii. few features are activated on that node on the temperature data (MODE, MEDIAN, MAX and MIN)
 - iv. the alarm engine function is set on the node to work on data coming from the accelerometer sensor with window WINDOW_SIZE=40 and shift SHIFT_SIZE=20.
 - v. Then one alarm is set on the internal CPU temperature sensor, so that an alarm message will be sent back when:
 - 1. the MIN value on CH1 is greater than 1000 and lower than 3000
- e. once all the request are set, the network starts

```
manager.startWsn(true, true);
```

- f. on reception of the activated data (`received(Data data)`), data payload is displayed

```
System.out.println(data);
```

- g. during application runtime, functions can be deactivated and activated. Here for instance:
 - i. After receiving 5 feature packets, the first activated feature on that sensor is deactivated

```
if(counter == 5) {
    // it's possible to deactivate functions computation at runtime (even when
the radio on the node works in low-power mode)
    FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();
    sfr.setSensor(features[0].getSensorCode());
    sfr.remove(new Feature(features[0].getFeatureCode(),
        SPINESensorConstants.ALL);
    manager.deactivate(data.getNode(), sfr);
}
```

- ii. After receiving 10 feature packet a new feature (RANGE) is computed on the first channel of that sensor

```
if(counter == 10) {
    // and, of course, we can activate new functions at runtime
```



```
FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();
sfr.setSensor(features[0].getSensorCode());
sfr.add(new Feature(SPINEFunctionConstants.RANGE,
                   SPINESensorConstants.CH1_ONLY));
manager.activate(data.getNode(), sfr);
}
```

iii. After 20 alarm packets the

```
if(counter_alarm == 20) {
    AlarmSpineFunctionReq sfr2 = new AlarmSpineFunctionReq();
    sfr2.setSensor(SPINESensorConstants.ACC_SENSOR);
    sfr2.setAlarmType(SPINEFunctionConstants.ABOVE_THRESHOLD);
    sfr2.setDataTypes(SPINEFunctionConstants.MAX);
    sfr2.setValueType(SPINESensorConstants.CH1_ONLY);
    manager.deactivate(data.getNode(), sfr2);
}
```

2.2 SPINE Logging capabilities

The SPINE Framework uses a Logger to print info or warning messages, to notify of exceptions, and so on.

This enables a convenient way to filter undesired messages, to forward logs to output files, and much more.

From a SPINE user point of view, it can be useful to use the SPINEManager static method `getLogger()` e.g. to modify the default logging level (INFO):

```
SPINEManager.getLogger().setLevel(Level.WARNING);
```

From a SPINE developer point of view, it worth to report the correct way to print using the logger:

```
if (SPINEManager.getLogger().isLoggable(Logger.[SEVERE|WARNING|INFO]))
    SPINEManager.getLogger().log(Logger.[SEVERE|WARNING|INFO], "msg");
```

Logging levels are hierarchical in terms of gravity. For instance, if logging level has been set to WARNING, only the SEVERE and the WARNING messages will be logged, while INFO messages will not.

Please refer to the Jade Framework logging tutorial (<http://jade.tilab.com/doc/tutorials/logging/JADELoggingService.html>) and to the java.util.logging javadocs for further details.

3 SPINE architecture

SPINE is a framework for signal processing in wireless sensor networks; therefore it has to take care of communication among nodes into the network, on-node sensor management and signal processing functionalities.

SPINE1.3 node side architecture reflects all these functionalities and divides on-node functionalities into three main logic blocks:

1. Communication: takes care of radio communication, data packet build and parse, radio duty cycle and channel access schemes;
2. Sensing: manages data sampling from sensors and storage on a shared buffer;
3. Processing: takes care of the data processing

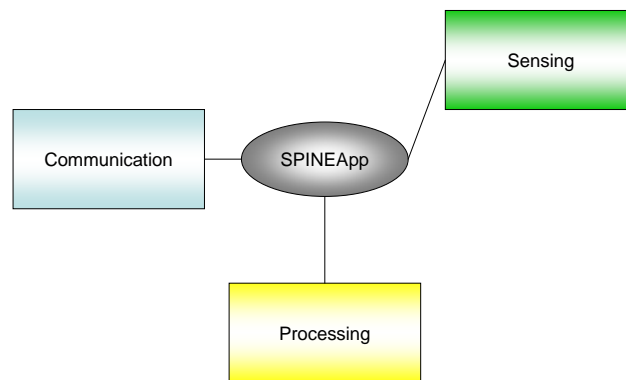


Figure 3.1: SPINE 1.3 node side, high level architecture

The lifecycle of SPINE1.3 framework on the node side is managed by the SPINEApp component. SPINEApp acts as a dispatcher among the aforementioned modules and in particular it provides the following services:

- Upon receipt of the Service Discovery message, asks for information about supported sensors to the SensorBoard Controller and about supported functionalities to the Function Manager and then sends back to the coordinator the Service Advertisement packet.
- 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)

SPINEApp code may be found into the `Spine_nodes/apps` folder.

This is the place from where you'll type the `make` and `make install` commands.



3.1 Communication

On the node side, SPINE 1.3 communication part (Figure 3.2) is composed of two main components: the Radio Controller and the Packet Manager.

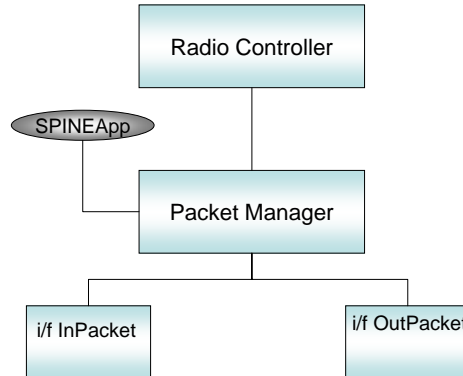


Figure 3.2: SPINE 1.3 node side, Communication Modules

The **Radio Controller** takes care of the access to the radio (send and receive operation) and, when activated, the low power mode and the TDMA access to the radio.

Upon receipt of the start message, the node will start working as configured before (e.g. computing the function as defined during set up phase).

Into the start message payload, server side application can set two different flags, `radioAlwaysOn` and `enableTDMA`.

radioAlwaysOn

When the `radioAlwaysOn` parameter is set to `FALSE`, the radio will be turned off if no message needs to be sent and turn on whenever the node has a message to send.

Since bidirectional communication must be still guarantee, the radio controller component implements the logic for listening to the messages the coordinator might be holding for it (Figure 3.3 and Figure 3.4).

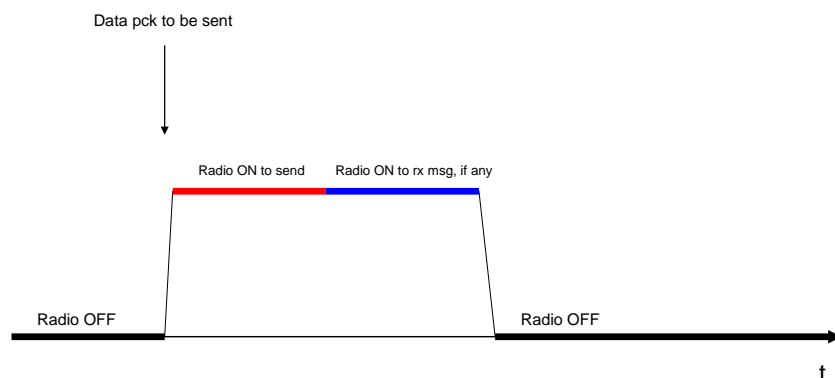


Figure 3.3: radio behaviour when `radioAlwaysOn=FALSE`

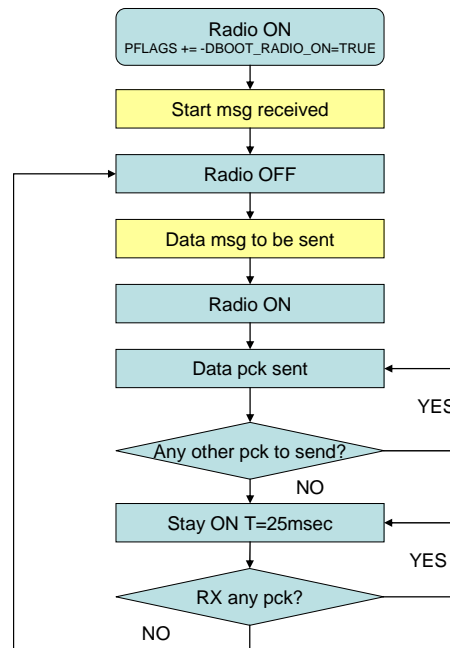


Figure 3.4: flow chart of RadioController component when radioAlwaysOn=FALSE

On the server side, if the network has been started up with `radioAlwaysOn = FALSE` and the SPINEManager has messages to send to a node, it stores the message until the next time it gets a data message form that node. Moreover, the coordinator waits for an acknowledgment from the node before deleting the message from the queue, otherwise it tries to retransmit it. The implementation of this mechanism might be found into the `TOSLocalNodeAdapter.java`.

enableTDMA

When the `enableTDMA` parameter is set to `TRUE`, the Radio Controller will apply a TDMA schema to all its transmissions. Therefore, whenever it has to send a data packet it will wait for the next time slot that has been assigned to it. This may be needed in scenarios where a large amount of data must be sent to the coordinator from several nodes and a pure CSMA-CA schema could not be enough for achieving a reliable communication.

The start message will carry also the total number of nodes present into the network at that time. Upon reception of the start message, if the 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. This is the reason why, users must flash nodes with sequential IDs (1,2,...) and, if needed, they can change the `TDMA_FRAME_PERIOD` into the Makefile, defined by default to be 600msec. Note that since the coordinator is not assigned to a particular time slot, it will try to send packets as soon as possible, relying on the default radio access schema of the base-station (typically CSMA-CA for TinyOS basestations).

The **Packet Manager** component takes care of building and parsing SPINE packet payload and header; moreover, if necessary, it takes care of the fragmentation of the packet before sending it to the radio. In this way, whenever the Function Manager has to send a packet, it does not have to care about the length and simply call `PacketManager.build`. Then the PacketManager will, if needed, parse the packet and send multiple packets. On the server side, the SPINE Manager will properly rebuild the packet. Note that, at this stage, the mechanism is not implemented in the other direction: therefore the server side will never send fragments of the same packet but will build messages according to the maximum length allowed.

The Packet Manager may take care of different types of incoming and outgoing packets as far as they implement the defined interfaces (`InPacket` with the `parse` command and `OutPacket` with the `build` command).



On the node side, communication interfaces might be found in the `tos\interfaces\communication` folder and their implementations in the `tos\system\communication` one.

On the server side, SPINE 1.3 core framework has been designed to be independent on the network it is communicating with (Figure 3.5). Therefore, all the SPINE Server Side core code contained into the `spine` and `spine.datamodel` packages do not use any TinyOS specific APIs.

TinyOS specific messages and low level communication APIs are used and defined into the `spine.communication.tinyos` package.

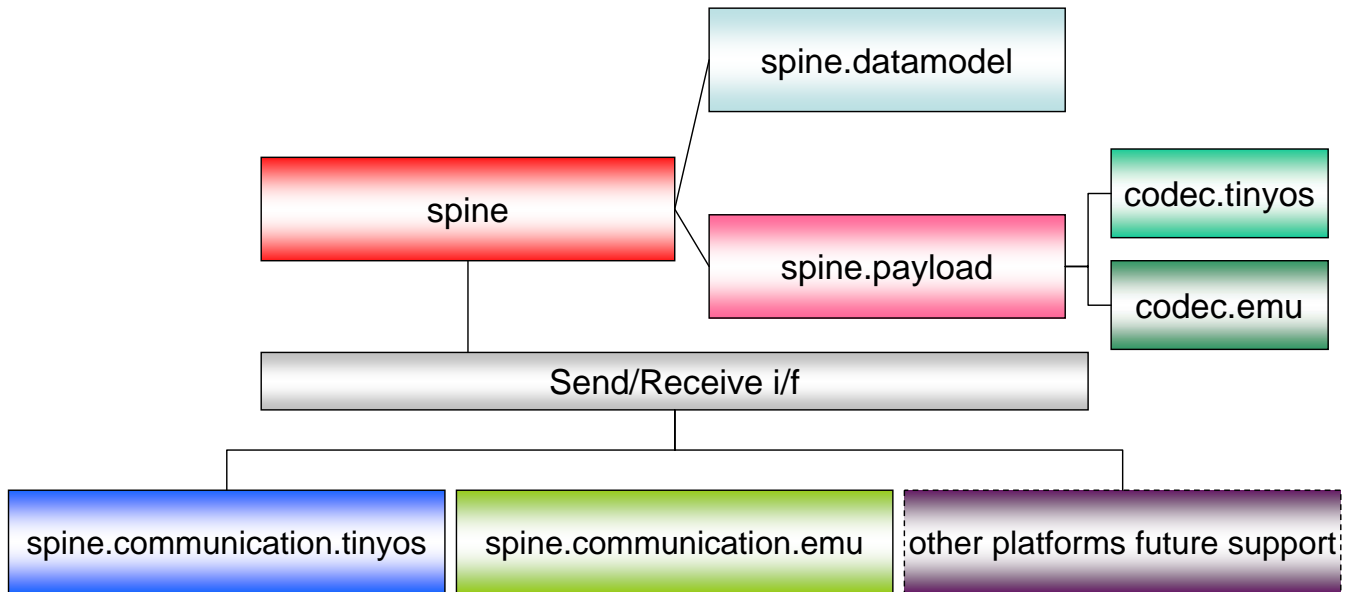


Figure 3.5: SPINE 1.3 Server Side, architecture.

SPINE1.3 defines a bidirectional protocol for communication between the coordinator node and the sensor nodes.

All the messages have the same SPINE protocol header that is build as illustrated below:

Bit	2	1	5	8	16	16	8	8	8
	Version	Extension	Type	GroupID	SourceNodeID	DestNodeID	SequenceNr	Total fragments	Fragment seqNr

In particular, it supports management messages from the coordinator to the nodes such as:

- Network Discovery. It is an empty packet, sent as broadcast so that can reach all the nodes into the network;
- Set Up Sensor. It contains the sampling time settings needed to set the sensor

Bit	4	2	2	16
	Sensor Code	Time Scale	Reserved	Sampling Time

- Set Up Function. It contains the general settings that are needed by a function. Parameters list is basically a list of byte so that can be very generic and used for different functions.

Bit	8	8	Variable
	Function Code	Param Length	Param List

- Activate/Deactivate Function. It contains details on the functionalities to be activated. As above, parameter list is a generic list of array.

Bit	8	8	8	Variable
-----	---	---	---	----------



	Function Code	Activate/Deactivate	Param Len	Param List
--	---------------	---------------------	-----------	------------

- Start Network. It 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).

Bit	16	8	8
	#nodes in the net	Radio always ON	Enable TDMA

On the other hand, sensor nodes can send to the coordinator:

- Service Advertisement reports information about the sensors and the functions supported by the node.

Bit	8	8*#sensors		8	Variable
	#sensors	Sensor Code	Bitmask	#functions	Functions

The function field is filled in by the specific Function and contains codes useful to indentify the function itself and other sub-functions if present.

- Data Packets with the function type and a list of bytes with the specific data. It has been defined to be very generic and may be easily used by all the other functionalities.

Bit	8	8	Variable
	Function Code	Param Len	Param List

- Service Message Packets used by the remote nodes to notify of warning and error events which may occur, and to transmit other system info messages.

Bit	8	8
	Type Code	Detail Code

All SPINE TinyOS packets have been designed to be very general, therefore adding new sensors and new signal processing functionalities should not imply defining new messages.

If new messages are really needed, SPINE1.3 framework can be easily enhanced. Please contact the SPINE Team through the spine-dev@avalon.tilab.com mailing list for more details.

At this stage peer to peer communication among nodes is not supported yet, but its integration into the SPINE framework could be done very easily.

3.2 Sensing

The sensing part is composed by three main components: the Sensor Registry, the Sensor Board Controller and the Buffer Pool (Figure 3.6).

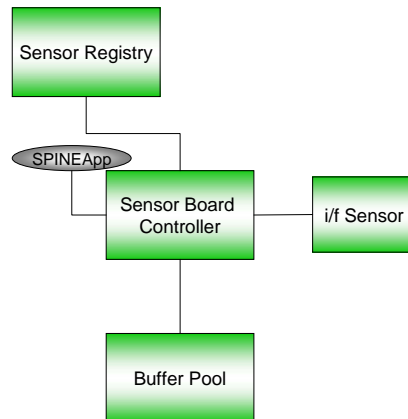


Figure 3.6: SPINE 1.3 node side, Sensing Modules

The **Sensor Board Controller** manages all the sensors that are registered, thanks to the **Sensor Registry**, to SPINE. Its main functionalities are setting sensor parameters (e.g. sampling time) and get data from them.

Data is then stored in a pool of buffers (**Buffer Pool**) that stores the data in a FIFO manner. Buffer Pool is set according to Makefile parameters `BUFFER_POOL_SIZE` and `BUFFER_LENGTH` to allocate a buffer for every channel of every registered sensor. For convenience, the `BUFFER_POOL_SIZE` parameter is automatically set while using predefined sensorboards.

The sensing part may be enhanced by adding new sensors to be supported by SPINE.

3.2.1 Sensors now supported

“SPINE” Motion SensorBoard

Developed in collaboration with the University of California at Berkeley and Tampere University of Technology, the SPINE motion sensorboard mounts a tri-axial accelerometer sensor (STMicroelectronics LIS3LV02DQ) and a dual-axial gyroscope sensor (InvenSense IDG-300).

In addition, to retrieve the actual voltage level of the built-in Lilon battery, a voltage sensor can be queried.

Finally, the MSP430 microcontroller internal temperature sensor is also supported.

The SPINE Motion sensorboard is open-source and all the required documentation can be released upon request.

“Moteiv” Sensor kit

Thanks to the effort of Carlo Caione from Università di Bologna, Italy, the optional sensor kit of the Moteiv Tmote Sky platform has been introduced in SPINE.

In particular, supported sensors are the environmental temperature and humidity sensors (Sensirion AG SHT11) and the PAR/TSR photodiodes sensors (Hamamatsu S1087).

In addition, the MSP430 microcontroller internal temperature sensor as well as the voltage diode are also supported.

“Shimmer” SensorBoard



The Shimmer platform has been introduced into SPINE with the contribution of Pering Trevor from Intel Research Lab, Santa Clara. Currently, it is supported a three axis accelerometer sensor (Freescale MMA7260Q)

“Mts300” SensorBoard

SPINE supports the Micaz platform and its MTS300 sensorboard. However, to date the only supported sensor is the dual axis accelerometer.

“BioSensor” Board

The biosensor board has been developed by Ville Seppa-Pekka from Tampere University of Technology. This very sophisticated board uses four electrodes connected at the board itself and attached to the surface of the patient's ribcage. The same electrodes are used for both heart and breathing measurement. Breathing is measured using electrical impedance pneumography (EIP). The ECG signal can be easily used to detect the heart rate.

The biosensor board features also a three axis accelerometer.

3.2.2 How to introduce a new sensor

SPINE1.3 supports different kinds of sensors and different implementations of the same sensor (e.g accelerometer).

Adding a new sensor to the SPINE framework is quite easy. Sensor's driver must implement a defined Sensor interface to then be part of the SPINE framework and use all its functionalities.

NODE SIDE

The steps to be followed to integrate your “myNewSensor” in SPINE1.3 TinyOS implementation on the node side are:

1. Look into `tos\interfaces\sensing` folder at `Sensor.nc`: this file contains the interface that your sensor driver must implement.
2. `tos\types\SensorConstants.h` must be updated to support a new sensor code `SensorCode (MY_NEW_SENSOR)`

3. Take one of the following approaches depending on how you answer this question: *do you already have a driver implemented to support your myNewSensor?*

Both approaches might be used depending if you can change or write from scratch your driver implementation (approach a.) to make it according to SPINE 1.3 or you want to leave the driver implementation as it is and wrap it into the SPINE framework (approach b.).

- a. *NO, I don't have any driver already implemented* or *YES but I want to change it according to SPINE*. This means you now will write the driver for supporting your sensor according to Sensor SPINE interface. It is very simple: you'll have to implement all the methods defined in `tos\interfaces\sensing\Sensor.nc`.

SPINE1.3 supports, among others, a 3-axis accelerometer for the spine board. This is a good example to start with:

- i. Look into `tos\system\sensing` folder at `AccSensorC.nc`: this is the component that provides the platform independent layer of the AccSensor. You must create here a `MyNewSensorC.nc` file that links with the actual implementation of the driver (`HilMyNewSensorC`).

If your sensor is already present among the ones supported by the current release of SPINE (e.g accelerometer), you do not have to implement this component, since it is already present (you can have a look to the actual



implementation of the AccSensor both for the spine and the mts300 sensor boards).

- ii. HilAccSensorC can be found into `tos\sensorboards\spine` folder. Please note that **if you want to develop your own sensor board you can create a “mySensorBoard” folder that looks like spine folder with your drivers inside (HilMyNewSensorC)**
 - iii. Having a look to `HilAccSensorP.nc`, you can see that it actually implements itself all the APIs defined in the `Sensor.nc` interface. This contains all the driver logic by itself. In your case will be `HilMyNewSensorP.nc` to implement the driver according to the defined APIs.
- b. *YES and I need it as it is*: in this case you will not have to change your actual driver implementation but you will wrap it to match with the SPINE sensor interface. The SPINE gyroscope is a good reference implementation for this approach:
- i. Look into `tos\system\sensing` folder at `GyroSensorC.nc`: it is the same as above and you can see that the implementation is done by the `HilGyroSensorC` component (in your case `MyNewSensorC.nc` must be created).
If your sensor is already present among the ones supported by the current release of SPINE (e.g accelerometer), you do not have to implement this component, since it is already present (you can have a look to the actual implementation of the AccSensor both for the spine and the mts300 sensor boards).
 - ii. `HilGyroSensorC` can be found into `tos\sensorboards\spine` folder, that's where `HilMyNewSensorC` should be put. Please note that **if you want to develop your own sensor board you can create a “mySensorBoard” folder that looks like spine folder with your drivers inside (HilMyNewSensorC)**
 - iii. Look at `HilGyroSensorP.nc`, and notice that it implements all the APIs defined in the `Sensor.nc` interface wrapping already existing drivers (namely `GyroXSensorC` and `GyroYSensorC`) adding the logic to match with the defined interface.
4. In the implementation of the interface, in both cases, you need to take care of the registration to the `SensorBoardRegistry`. That's very simple: in the `Boot.booted()` event implementation of the `HilMySensorP.nc`, you need to write:

```
call SensorsRegistry.registerSensor(MY_NEW_SENSOR);
```

5. If you defined a “mySensorBoard” you have to define it in the `support/make/SPINE.extra`

```
ifeq ($(SENSORBOARD),mySensorBoard)
    PFLAGS += -DMY_SENSORBOARD
    PFLAGS += -DBUFFER_POOL_SIZE=n
endif
```

Where n should be the total number of sensor channels that are supposed to be used at the same time by your application. In the worst case, you'll need to specify the total sum of the channels of each built-in sensor. For instance, if your sensorboard is equipped with a 3D accelerometer and a single-axis pressure sensor, n would be equal to 4.

6. Now, you need to do some wiring. Go into `tos\system\sensing` folder and open `SensorBoardControllerC.nc`. You simply have to add 4 lines for each sensor supported by your sensor board. Those are needed basically to allow the `SensorBoardController` accessing the new sensor and reserving a timer for sampling it.

```
#ifdef MY_SENSOR_BOARD
/* For the myNewSensor Sensor */
```



```

        components myNewSensorC;
        components new TimerMilliC() as myNewSensorTimer;
        SensorBoardControllerP.SensorImpls[MY_NEW_SENSOR] -> myNewSensorC;
        SensorBoardControllerP.SamplingTimers[MY_NEW_SENSOR] ->
        myNewSensorTimer;
    #endif

```

Indeed, if you add a new sensor to an existing sensor platform (e.g. spine), you'll have to add the wiring into the `#ifdef SPINE_SENSOR_BOARD`.

It is worth reminding that the abstract configuration module of a new sensor must be implemented and placed into `tos/system/sensing` ONLY if does not yet exist such a module for the generic sensor type (e.g. accelerometer, pressure, temperature, humidity). In other words, it is also incorrect to create this configuration module with a specific name such as `MANUFACTURER_CHIPMODEL_SENSORTYPE`; if you are adding a new SpO2 sensor, no matter which is the model code or manufacturer, you'll just name the generic configuration module as `SPO2SensorC.nc`, and the actual sensor driver components will be named as `HilSPO2SensorC.nc` and `HilSPO2SensorP.nc`.

That is done with the intent of simplifying the introduction of other e.g SpO2 sensor and to reduce the number of generic configuration components.

Please reference to comments into the `Sensor.nc` file to better understand what the interface methods have to provide.

SERVER SIDE

Since SPINE1.3 server side provides APIs for managing on node functionalities, adding a new sensor support into the Java SPINE1.3 Server side, as far as it does not imply any new functionality, it is very simple.

You only have to add the `MY_NEW_SENSOR` constant into the `spine\SPINESensorConstants.java` as well as the string that defines it.

You'll only have to add 2 lines copying the ones for the accelerometer support here listed and setting the code and the string according your values.

```

    public static final byte ACC_SENSOR = 0x01;

    case ACC_SENSOR: return ACC_SENSOR_LABEL;

```

Please note that `MY_NEW_SENSOR` value MUST be the same you set into the node side and MUST be different from the ones already supported by SPINE1.3, as listed in the `SensorsConstants.h` file.

If your new sensor needs functionalities that are now not present in SPINE1.3, you'll have to add them as described in the following paragraph.

3.3 Processing

The processing part is composed by a function manager that takes care of all the functions supported on the node (Figure 3.7).

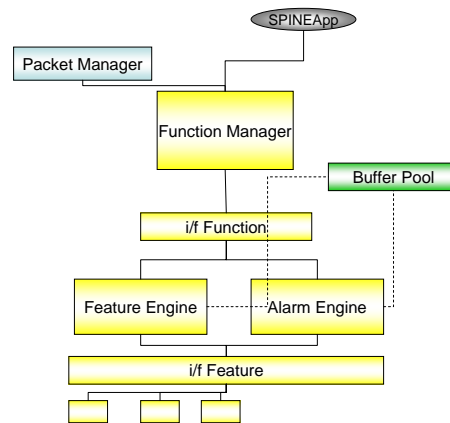


Figure 3.7: SPINE 1.3 node side, Processing Modules

On the node side, a generic function must implement the **Function** interface (tos\interfaces\processing) to be then managed by the **Function Manager**. The interface has been defined to be very general, in this way all the logic may be implemented inside the function.

The server Side sends two different commands to the Function Manager (setUpFunction and activateFunction) addressing a specific functions and sending a generic array of parameters that will then decoded by the implemented functionality. The data packet sent back to the Server Side with the computed values is composed as well by a generic array of bytes. The processing of the functions will then start upon receipt of the “start” message from the server Side.

Functions supported by SPINE1.3 release are Feature Engine and Alarm Engine.



3.3.1 Feature Engine

Data sampled by sensors on the motes may be sent to the coordinator without any processing (raw data) and then be analyzed on the server side. This may be not the most efficient procedure in terms of energy consumption and channel optimization.

Therefore, data may be processed on the node and only the result of the computation been sent to the coordinator.

SPINE 1.3 Feature Engine provides periodic calculation of simple feature on sensed data. The server side application can request a feature using two setup messages: a setup function message with values for the window and shift with a given sensor and an activate function message with the features desired for that sensor.

Features include AMPLITUDE, MAX, MEAN, MIN, MODE, PITCH&ROLL, RANGE, RAW DATA, RMS, STANDARD DEVIATION, TOTAL ENERGY, VARIANCE, VECTOR MAGNITUDE, ENTROPY, and Kcal. Features can be calculated over multiple channels. For instance MEAN calculates the mean value over each active channel while VECTOR_MAGNITUDE is computed over all channels and returns a single value.

Feature engine can be configured by the serverSide application using setUpFunction and activateFunction messages.

SETUP FEATURE ENGINE

From the java application, the Feature Engine can be setup with different window and shift time for different sensors. For Feature Engine, we assume Window and Shift Size to be unique for a certain sensor (e.g. the same settings apply to all the channels of that sensor).

```
FeatureSpineSetupFunction ssf = new FeatureSpineSetupFunction();  
  
ssf.setSensor(sensor);  
  
ssf.setWindowSize(WINDOW_SIZE);  
  
ssf.setShiftSize(SHIFT_SIZE);  
  
manager.setup(curr, ssf);
```

ACTIVATE FEATURES CALCULATION

Once the feature engine is setup, several features can be activated to be computed on a single or multiple channels.

In the example below, the feature engine is set to compute the mode feature over the first channel and the minimum value over the second and the third channel of the sensor setup before.

```
FeatureSpineFunctionReq sfr = new FeatureSpineFunctionReq();  
  
sfr.setSensor(sensor);  
  
sfr.add(new Feature(SPINEFunctionConstants.MODE, CH1_ONLY));  
  
sfr.addFeature(SPINEFunctionConstants.MIN, CH2_CH3_ONLY);  
  
manager.activate(currNode, sfr);
```

FEATURE DATA

Once the feature engine is set up and the needed features are activated, the Feature Engine waits for the "start" message to start computing the features. The Feature Engine will send back to the server application a feature packet every shift time. The Server Side will get, into the



received(Data) event, a FeatureData packet that contains an array of Feature object. Therefore the application can access to the calculated Features values.

```
public void received(Data data) {
    switch (data.getFunctionCode()) {
        case SPINEFunctionConstants.FEATURE: {
            Node source = data.getNode();
            Feature[] feats = ((FeatureData)data).getFeatures();
            Feature firsFeat = feats[0];
            byte sensor = firsFeat.getSensorCode();
            byte featCode = firsFeat.getFeatureCode();
            int chlValue = firsFeat.getChlValue();
            // do something with this feature data...
            break;
        }
        default: break;
    }
}
```

3.3.1.1 How to add a new feature into the feature engine

SPINE1.3 feature engine supports all the features listed above but it can be easily enhanced to compute other features on the sensed data.

NODE SIDE

1. In Spine_nodes1_2\tos\types\Functions.h add MY_NEW_FEATURE code into enum FeatureCodes
2. Into tos/system/processing you have to implement the feature logic. For this reason you'll create 2 new files: MyNewFeatureC.nc and MyNewFeatureP.nc. MyNewFeature module will have to provide the Feature interface (look into tos/interfaces/processing), meaning it will have to implement the .calculate and the .getResultSize commands.
 - a. Feature.calculate command computes your feature based on the available data, which is a window length number of samples per each active channel.
 - b. Feature.getResultSize returns the word size (e.g. 2 for uint16_t) for the array of elements which is written by the Feature.calculate command.

Remind that, since the Boot interface is used, you'll have to implement the event Boot.booted(), where we suggest to register the feature to the FeatureEngine to be then discovered during discovery time.

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

3. Open tos/system/processing/FeatureEngineC.nc and add the following lines:


```
components MyNewFeatureC;
FeatureEngineP.Features[MY_NEW_FEATURE] -> MyNewFeatureC;
```




SERVER SIDE

Open `spine.SPINEFunctionConstants.java` and add the following lines that define the new feature:

```
public static final byte MY_NEW_FEATURE = 0x11;

public static final String MY_NEW_FEATURE_LABEL = "MyNewFeature"

case MY_NEW_FEATURE: return MY_NEW_FEATURE_LABEL;
```

The user must also add the new constants to switch statements of the methods in `SPINEFunctionConstants.java` (`functionCodeByString`, `functionalityCodeToString`).



3.3.2 Alarm Engine

Data sampled by sensors on the nodes may be needed by the coordinator only if they respect certain conditions. Therefore conditions may be verified by the node itself and data sent to the coordinator only when needed: this is the aim of the SPINE Alarm Engine.

SPINE 1.3 Alarm Engine provides node-generated events whenever functions' values do not respect previously set thresholds. Alarms can be set on any of the supported features, including raw data, specifying window and shift settings as well as alarm-specific parameters.

The Alarm Engine provides notification for four different kinds of alarms on all the features available on the node (Figure 3.8).

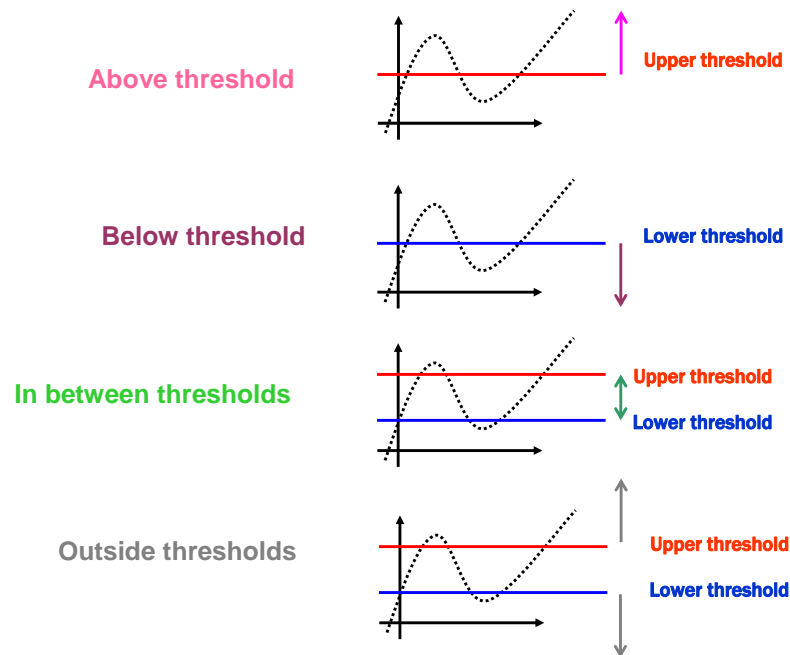


Figure 3.8: Alarm types

Alarm engine can be configured by the serverSide application using `setUpFunction` and `activateFunction` messages.

SETUP ALARM ENGINE

From the java application, the Alarm Engine can be setup with different window and shift time for different sensors. As for the Feature Engine, we assume Window and Shift Size to be unique for a certain sensor (e.g. the same settings apply to all the channels of that sensor).

```
AlarmSpineSetupFunction ssf = new AlarmSpineSetupFunction();  
  
ssf.setSensor(sensor);  
  
ssf.setWindowSize(WINDOW_SIZE);  
  
ssf.setShiftSize(SHIFT_SIZE);  
  
manager.setup(currNode, ssf);
```



ACTIVATE ALARMS

Once the alarm engine is setup, different alarms can be activated. A single alarm is activated defining the kind of data it has to check (DataType), on which sensor (Sensor) and which channel (ValueType), thresholds (upper and lower) and which kind of alarm (AlarmType) to be pick among the four described in Figure 3.8.

In the example below, the Alarm Engine is configured to report the Maximum value of the accelerometer on Channel 1 (X axis) when it is above the upperThreshold.

Once the preset condition is verified, the node sends back to the coordinator all the information about the event that occurred.

```
AlarmSpineSetupFunction sfr = new AlarmSpineFunctionReq();

sfr.setDataType(SPINEFunctionConstants.MAX);

sfr.setSensor(SPINESensorConstants.ACC_SENSOR);

sfr.setValueType((SPINESensorConstants.CH1_ONLY));

sfr.setLowerThreshold(lowerThreshold);

sfr.setUpperThreshold(upperThreshold);

sfr.setAlarmType(SPINEFunctionConstants.ABOVE_Threshold);

manager.activate(currNode, sfr);
```

ALARM DATA

Once the Alarm Engine is set up and the requested alarms are activated, the Alarm Engine waits for the "start" message to start monitoring the data to check if an alarm condition is present.

The Alarm Engine will send back to the server application an alarm packet whenever the alarm condition occurs. The Server Side will get, into the dataReceived event, a AlarmData packet that contains information about the specific alarm.

Therefore the application can access to the values reported by the Alarm Engine.

```
public void received(Data data) {
    switch (data.getFunctionCode()) {
        case SPINEFunctionConstants.ALARM: {
            Node source = data.getNode();
            byte alarmType = ((AlarmData)data).getAlarmType();
            int value = ((AlarmData)data).getCurrentValue();
            byte sensor = ((AlarmData)data).getSensorCode();
            // do something with this alarm data...
            break;
        }
        default: break;
    }
}
```



3.3.3 Step Counter

The Step Counter is another predefined function that uses accelerometer data to detect locally on a sensing mote placed on the waist the number of steps performed while walking. As usual, the function can be enabled upon request. Once enabled, it notifies the coordinator as soon as a new step is detected, transmitting the total number of steps detected from the beginning (this allow message drops tolerance).

The Step Counter engine relies on a naïve algorithm without any purpose of being effective in any possible situation.

The Step Counter engine can be configured by the serverSide application using `setUpFunction` and `activateFunction` messages.

SETUP STEP COUNTER ENGINE

From the java application, the Step Counter can be setup very easily as it does not currently requires parameters.

```
manager.setup(waistNode, new StepCounterSpineSetupFunction());
```

ACTIVATE THE STEP COUNTER

Once the Step Counter engine is setup, it can be activated as follows:

```
manager.activate(waistNode, new StepCounterSpineFunctionReq());
```

STEP COUNTER DATA

Once the Step Counter Engine is set up and activated, it waits for the "start" message to actually start operating.

The Step Counter notifies the coordinator as soon as a new step is detected, transmitting the total number of steps detected from the beginning

```
public void received(Data data) {
    switch (data.getFunctionCode()) {
        case SPINEFunctionConstants. STEP_COUNTER: {
            int steps = ((StepCounterData) data).getStepsCount();
            // do something with this data...
            break;
        }
        default: break;
    }
}
```

3.3.4 Buffered Raw Data

Buffered Raw Data is a useful function in particular when high sampling rate of the sensors is requested. It basically allows packing a buffer of sensor readings in a single message with a size defined at runtime by the user application.

Each BufferedRawData message contains readings from a single sensor, possibly over multiple sensor channels if requested.

If a BufferedRawData service is enabled on different sensors, separate messages will be generated per each sensor.

Sensor readings are stored into the two-dimensional array "values" of `spine.datamodel.BufferedRawData`: rows refer to the sensor channels (if a specific sensor channel is not requested, the corresponding row will be null) and column represent samples over time. The



samples are chronologically ordered into the matrix: the first element (column "0") is the oldest sample, last element (column "bufferSize-1") is the newest one.

It's possible to specify for a given sensor, the sensor channels of interest and the buffer size (in number of samples) to be transmitted at a time.

Moreover, specifying a "buffer shift ahead" in the range [1, bufferSize-1] could be useful for debugging purposes. ShiftSize = 0 is not permitted while ShiftSize = bufferSize means no overlap between adjacent buffers.

The Buffered Raw Data engine can be configured by the serverSide application using setUpFunction and activateFunction messages.

SETUP BUFFERED RAW DATA ENGINE

From the java application, the Buffered Raw Data function can be setup as shown below:

```
BufferedRawDataSpineSetupFunction brdsf = new BufferedRawDataSpineSetupFunction();

brdsf.setSensor(sensor);
brdsf.setChannelsBitmask(SPINESensorConstants.CH1_ONLY);
brdsf.setBufferSize(BUFFER_SIZE);
brdsf.setShiftSize(SHIFT_SIZE);

manager.setup(currNode, brdsf);
```

ACTIVATE THE BUFFERED RAW DATA

Once the Buffered Raw Data engine is setup, it still need the sensor on which actually activate it:

```
BufferedRawDataSpineFunctionReq brdfr = new BufferedRawDataSpineFunctionReq();
brdfr.setSensor(sensor);
manager.activate(curr, brdfr);
```

The Buffered Raw Data function can be enabled on multiple sensors. Different data messages will be sent for each of them.

BUFFERED RAW DATA

Once the Buffered Raw Data Engine is set up and activated, it waits for the "start" message to actually start operating.

The buffered sensor readings can be retrieved as shown below:

```
public void received(Data data) {
    if (data instanceof BufferedRawData &&
        ((BufferedRawData)data).getSensorCode() == sensor) {
        int[][] values = ((BufferedRawData)data).getValues();
        // do something with this data...
    }
}
```

The SPINE 1.3 provides a simple yet useful "BufferedRawDataToFile" application located in spine.test that uses the Buffered Raw Data function to store into a text file readings from the specified sensor over all the discovered remote nodes.

3.3.5 HMM

The Hidden Markov Model (HMM) Classifier has been introduced into SPINE by Prof. Roozbeh Jafari's group from University of Texas at Dallas. The actual implementation is an event-based



variant of the classic HMM and runs on the remote node. It is functionally composed of three entities: a pre-filter, a feature annotator, and the classifier itself.

Details and related papers on this algorithm can be found at <http://www.essp.utdallas.edu/>.

Currently, the pre-defined available trained model (represented by the HMM chain as a emission probabilities matrix) performs gait analysis. The algorithm itself is general-purpose and can be used for detecting events of diverse nature. However, the HMM training software, is not currently available is SPINE, and must be downloaded from the aforementioned website as well. This training algorithm will eventually generates the file representing the new HMM chain to be used by the HMM classifier on the remote mote.

HMM ENGINE

From the java application, it is possible to specify whether the remote mote has to transmit a new classification result as soon as it is available, or it can buffer them to fill up the data message payload before sending the message over the air. The HMM can be setup as shown below:

```
HmmSpineSetupFunction hssf = new HmmSpineSetupFunction()
Hssf.setSendFullMsg(true);
manager.setup(node, hssf);
```

ACTIVATE THE HMM

Once the HMM engine is setup, it can be activated as follows:

```
manager.activate(waistNode, new HmmSpineFunctionReq());
```

HMM DATA

Once the HMM Engine is set up and activated, it waits for the “start” message to actually start operating.

```
public void received(Data data) {
    if (data instanceof HmmData) {
        int[] states = ((HmmData)data).getStates();
        // do something with this data...
    }
}
```

3.3.6 How to add a new function

NODE SIDE

A generic function must implement the Function.nc (Spine_nodes1_2\tos\interfaces\processing\)\ interface and therefore provide the setUpFunction, activateFunction, disableFunction, getFunctionList, startComputing and stopComputing commands.

The inner implementation of these commands depends on the function logic and it's transparent to the SPINE core that communicates with all the functions, through the FunctionManager, with a general array of parameters.

Here few steps to integrate myNewFunction algorithm into SPINE1.3:

1. In Spine_nodes1_2\tos\types\Functions.h add MY_NEW_FUNCTION code into the enum FunctionCodesFunctionCodes
2. Open Spine_nodes1_2\tos\interfaces\processing\Function.nc: this is the interface that myNewFunction must implement.



You have to place your implementation into the Spine_nodes1_2\tos\system\processing folder. In detail, you will write 2 files:

- a. MyNewFunctionEngineP.nc, module component that implements the function logic (e.g. classification algorithm...). This module has to implement the Function.nc interface as well as three events:

- i. Boot.booted
- ii. FunctionManager.sensorWasSampledAndBuffered
- iii. BufferPool.newElem

Remind that into the Boot.booted event implementation you'll have to register the the new function to the function manager to be then announced during the discovery process. A typical implementation is:

```
event void Boot.booted() {  
    if (!registered) {  
        call FunctionManager.registerFunction(MY_NEW_FUNCTION);  
        registered = TRUE;  
    }  
}
```

- b. MyNewFunctionEngineC.nc is the configuration component.

Developing myNewFunction logic, whenever it is needed to send a data a packet to the coordinator, the FunctionManager.send command must be called with the MY_NEW_FUNCTION function code and the data payload array of bytes.

3. Then, open the Spine_nodes1_2\tos\system\processing\FunctionManagerC.nc and add the following lines:
 - a. components MyNewFunctionEngineC.;
 - b. FunctionManagerP.Functions[MY_NEW_FUNCTION] -> MyNewFunctionEngineC;

Now that your new function logic is implemented on the node side, you have to enhance the server side as well to be able to correctly activate the function and receive data from it.

SERVER SIDE

SPINE1.3 provides two messages for setting on node functionalities from the server side. As all the other functions already present, myNewFunction developed on the node side, must be set using those messages:

1. Set up Function contains the function code and a generic list of bytes to set up the function.

Bit	8	8	Param Length * 8
	Function Code	Param Length	Param List

You can define your own parameters according to the settings needed by your function.

On receipt of this message on the node side, parameters will be given to the function calling the setUpFunction method defined into the Function.nc interface.

You'll have to implement the encoding logic into spine.communication.tinyos.MyNewFunctionSpineSetupFunction.java that will extend the already defined SpineSetupFunction.java abstract class (refer to spine.communication.tinyos package for SpineSetupFunction.java)

2. Activate/Deactivate Function

Depending on the functionality, you may need to activate (or deactivate) with specific settings subparts of this functionality. Hence, you'll have to fill in and send a activate/deactivate function packet.

Bit	8	8	8	Variable
	Function Code	Activate/Deactivate	Param Len	Param List



You can define your own parameters according to the settings needed by the activation of all or part of your function.

On receipt of this message on the node side, parameters will be given to the function calling the activateFunction method defined into the Function.nc interface.

You'll have to implement the encoding logic into

`spine.communication.tinyos.MyNewFunctionSpineFunctionReq.java` that will extend the already defined `SpineFunctionReq.java` abstract class (refer to `spine.communication.tinyos` package for `MyNewFunctionSpineFunctionReq.java`).

Once `myNewFunction` has been set and activated, it will send data to the server side, therefore it should be able of decoding it.

3. Data Packets

SPINE1.3 data packet format has been designed to be very general: it is basically an array of bytes which meaning change depending on the functionality.

Bit	8	8	Variable
	Function Code	Param Len	Param List

`MyNewFunction` implementation on the node side will send a data payload with proper fields that must be decoded at the server side.

When data is received as array of bytes at the server side, this array is passed to the decode method of `SpineData` class. This dynamically loads the implementation corresponding to the function type.

SPINE1.3 server side has been designed to be platform independent, which means that its core logic can run using different transport layers and protocols. For this reason, data formats must be defined in a platform independent way (into the `spine.datamodel` package) as well as in a platform specific way (in this release we support `TinyOS2.x` and the SPINE protocol on top of that).

Platform independent classes – into the `spine.datamodel` package:

- `MyNewFunction.java`: this class represents the `MyNewFunction` entity and therefore contains a constructor, a `toString` and getters methods.
- `MyNewFunctionData.java`: this class represents the `MyNewFunctionData` entity and it contains the decode method for converting low level `MyNewFunction` type data into an high level object.

Platform dependent class – `TinyOS2.x` implementation into the `spine.communication.tinyos` package

- `MyNewFunctionSpineData.java`: this class contains the static method to parse (decompress) a `TinyOS SPINE MyNewFunction Data` packet payload into a platform independent one. This class is invoked only by the `SpineData` class, thru the dynamic class loading. Note that this class is only used internally at the framework.

3.3.7 How to plug-in processing functions into SPINE

As the number of processing functions integrated into SPINE is keep growing, we needed a way to customize the framework to fit developers need.

While on the coordinator the burden of supporting new processing functions is very little considering the available hardware resources, on the node side, there are hard physical constraints, such as the limited code and data memory, that makes impossible to plug all those processing functions at the same time.

That being said, SPINE 1.3 introduced a fast and convenient way to plug-in each single pre-defined processing function.

Any function is independent from any others, so they can be plugged-in freely, given the mote-platform available memory.



Note that the following only affects the node side code (tinyos), while the coordinator code (java) does not require any customization.

To plug-in (or unplug), configuration flags have been defined in the SPINEApp/Makefile:

```
CFLAGS += -DENABLE_BUFFERED_RAWDATA
CFLAGS += -DENABLE_FEATURES
#CFLAGS += -DENABLE_ALARMS
#CFLAGS += -DENABLE_STEP_COUNTER
#CFLAGS += -DENABLE_HMM
```

As it can be seen, by default, only the Buffered Raw Data and the Feature engines are plugged-in.

Although is not required, it is highly suggested to follow this approach for custom-defined functions as well.

The previous paragraph discussed how to add a new processing function into SPINE. In particular, step 3 explains how to eventually wire the new implemented function to the core system.

To simplify the plug-in of your new function, you just need to add a new flag in the SPINEApp/Makefile:

```
CFLAGS += -DENABLE_MY_NEW_FUNCTION
```

and to enclose the function wiring in the FunctionManagerC.nc configuration file within a pre-processor directive that checks for the presence of that flag, as show below:

```
#ifdef ENABLE_MY_NEW_FUNCTION
    components MyNewFunctionEngineC;
    FunctionManagerP.Functions[MY_NEW_FUNCTION] -> MyNewFunctionEngineC;
#endif
```

3.3.8 How to enable a secure communication: SECURE SPINE

SPINE 1.3 introduces an optional encryption service that enables the remote motes to communicate securely with the basestation node. This work has been conducted by Philip Kuryloski at Cornell University's School of Electrical & Computer Engineering

This service is currently available for CC2420 based platforms such as the telosb/tmote and the shimmer. That is because it uses the built-in AES-128 hardware encryption.

If you intent to enable the security service, you'll need to install the SecureBaseStation application on your basestation node. SecureBaseStation can be downloaded from the TinyOS-2.x contrib CVS repository under "wsnlab" folder at <http://tinysos.cvs.sourceforge.net/viewvc/tinysos/tinysos-2.x-contrib/wsnlab/> using the command

```
cvs -z3 -d:pserver:anonymous@tinysos.cvs.sourceforge.net:/cvsroot/tinysos co -P tinysos-2.x-contrib/wsnlab
```

The default key is contained into support/make/secure_key.h and must match the key used by the SecureBaseStation secure_key.h file.

Security is disabled by default. To enable the security, define 'SECURE=Y' e.g. inside the Makefile or while typing the make command (e.g. SECURE=Y make telosb).

Please note that in order to enable the security service, both the SecureBaseStation and the SPINEApp require the latest T2 branch with CC2420 security support, which can be downloaded from the GIT repository: <http://hinrg.cs.jhu.edu/git/?p=jgko/tinysos-2.x.git> using the command

```
git clone git://hinrg.cs.jhu.edu/git/jgko/tinysos-2.x.git
```



4 Applications

4.1 HOWTO port a SPINE1.2 application to SPINE1.3

The SPINE 1.3 release introduces a compatibility break over the previous version. Several of the modifications consist of slight changes on method names, others involve wrapping of method parameters, and just a few cause major differences. Below are described the differences between the SPINE 1.2 and 1.3 APIs. However, for any issues you may have while porting your SPINE 1.2 application under SPINE 1.3, please contact the SPINE Team through the SPINE-dev mailinglist.

SPINE Configuration

SPINE 1.2 required to set `COMPORT` and the `"LocalNodeAdapter_ClassName"` properties in the `defaults.properties` file. It was then necessary to set `"LocalNodeAdapter_ClassName"` as a System property:

```
System.setProperty(Properties.LOCALNODEADAPTER_CLASSNAME_KEY,  
                    SPINEManager.getProperties().getProperty(  
                        Properties.LOCALNODEADAPTER_CLASSNAME_KEY));
```

and to get a `SPINEManager` instance by providing the `COMPORT` String:

```
String[] args = {SPINEManager.getProperties().getProperty(Properties.MOTECOM_KEY)};  
SPINEManager manager = SPINEManager.getInstance(args);
```

In SPINE 1.3 that is not necessary anymore. The `defaults.properties` file is not supposed to be edited by SPINE-application developers (in fact, it is also embedded in the `SPINE.jar`). Conversely, SPINE 1.3 applications must provide an `app.properties` text file (the actual name/extension can be different) which has to include the `COMPORT` (as in SPINE 1.2) and the `PLATFORM` properties. `PLATFORM` refers to the desired method to connect to the Basestation node. Three platforms have been defined to date (see also `spine.SPINESupportedPlatforms`):

1. **sf**, if you want to connect through a Serial Forwarder server
2. **tinyos**, if you want to connect using the `tinyos.jar` library
3. **emu**, if you want to connect to virtual SPINE nodes (see also [5. Data Collector and SPINE Node Emulator](#))

Note that, if desired, additional application-specific properties can also be included in the `app.properties` file. That won't cause interferences in the SPINE core.

SPINE Manager

In SPINE 1.2, the `SPINEManager` instance were retrieved as following:

```
SPINEManager manager = SPINEManager.getInstance(args);
```

SPINE 1.3 introduces a `SPINEFactory` class which is solely used to instantiate the `SPINEManager`:

```
manager = SPINEFactory.createSPINEManager("resources/app.properties");
```

Note that SPINE 1.3 only require to specify the `app.properties` file. It will then automatically retrieve the necessary properties.

In SPINE 1.2, applications would have registered a `SPINEListener` instance as following:

```
manager.registerListener(...);
```

In SPINE 1.3 this method changed name in `addListener`. Same thing happens to the `deregisterListener`, which becomes `removeListener`.



Issuing Requests to the sensing nodes

SPINE 1.3 refactoring involves slight modifications for some of the methods of the SPINEManager used to issue commands to the remote nodes. In particular, all the “setup”, “activate” and “deactivate” methods lose their suffix (e.g. “setupSensor” becomes “setup”, and “activateFunction” becomes “activate”). Furthermore, the “node” attribute, which in SPINE 1.2 was always addressed as an “int”, in SPINE 1.3 becomes a Node object. Applications should never instantiate the Node class, and the discovered nodes can be retrieved e.g. using two new methods of the SPINEManager: `getNodeByPhysicalID(Address id)` and `getNodeByLogicalID(Address id)`.

Also, the “start” method is now named “startWsn”, and “readNow” changes is “getOneShotData”.

Finally, the refactoring involves the location of the “SpineSetupSensor”, “[Feature/Alarm]SpineSetupFunction”, “[Feature/Alarm]SpineFunctionRequest”, and “[Feature/Alarm]SpineData” classes themselves.

In SPINE 1.2 they were located in `spine.communication.tinyos`, while in SPINE 1.3 must be imported from `spine.datamodel` and `spine.datamodel.functions` packages. Also note that “[Feature/Alarm]SpineData” are now named just “[Feature/Alarm]Data”.

Receiving Events (SPINE Listener)

A minor refactor also affects two methods of the SPINEListener interface.

`dataReceived(int nodeID, Data data)`, in SPINE 1.3 becomes `received(Data data)`, and `serviceMessageReceived(int nodeID, ServiceMessage msg)` becomes `received(ServiceMessage msg)`. In both cases, the node issuing the data or the service message can be retrieved by calling the `getNode()` method on the received msg.

Data retrieval

Besides the difference described in the previous section, SPINE 1.3 introduce a small change in the `addFeature` method of the `FeatureSpineFunctionReq` class. Instead of passing the `featureCode` and the `channelBitmask` parameters directly, they have been wrapped in a dedicated `Feature` constructor.

Hence,

```
sfr.add(SPINEFunctionConstants.MEAN, SPINESensorConstants.CH1_ONLY),
in SPINE 1.3 becomes
sfr.add(new Feature(SPINEFunctionConstants.MEAN, SPINESensorConstants.CH1_ONLY))
```

4.2 SPINE1.3 apps

A number of open-source applications using SPINE 1.3 can be found at https://avalon.cselt.it/svn/spine/trunk/Spine_apps.

Physical Energy Expenditure (KCAL)

This application estimates the energy consumed while performing daily basic activities. It uses a single sensor node equipped with a three-axis accelerometer, placed on the belt. Because accelerometer data are pre-filtered removing the gravity components (see David Mizell, “Using Gravity to Estimate Accelerometer Orientation”), the mote can be arbitrarily oriented. Furthermore, experiments on walking, running, ascending/descending stairs, and sitting showed good results even when the mote has been put in the front pants pocket.



The application requires a SPINE 1.3 mote with the KCAL feature (integrated by default). Indeed, part of the algorithm runs on the node. The accelerometer is sampled at 33Hz and “activity counts” on the horizontal and vertical axis are computed on 30-sample windows and sent to the java application every second. The java application collects 60 results and computes an estimation of the energy expenditure (expressed in Kcal) every minute, The algorithm can also be tuned with the subject gender and weight.

This work has been contributed by Edmund Seto and Po Yan from UC Berkeley, and by Raffaele Gravina from WSNLab Berkeley and it’s based on the published algorithm by Kong Y. Chen and Ming Sun.

4.3 How to contribute to SPINE1.3 apps

SPINE provides a set of API for managing and configuring the WSN and get back data coming from sensor nodes.

That’s a suggested outline for Java SPINE application that will use SPINE.jar as a library to build applications.

applicationName

```
|
|__README.txt (with indication about what the application does and what it needs to be set)
|__Licence (the licence it comes with)
|__CHANGELOG.txt (listing major changes in different commits)
|__build.xml (ant build file)
|__build.properties (properties file for the ant build)
|__doc (folder for the javadocs)
|__resources (this folder will contain all the resources useful for the application)
|
|   |__app.properties (mandatory to configure how the spine framework will be used)
|   |__img (folder for images, if any)
|   |__dataSets (optionally file for storing useful data, e.g. training sets in a classification app)
|   |__configuration (optionally xml files for configuring the net or the application params)
|__lib empty on the repo, to be filled in by users with (SPINE.jar, external jars, external
resources)
|__dist empty on the repo, to be filled by ‘ant dist’ contains application.jar, javadocs...
|__script (optional folder for scripting files if any)
|__src (source code organized according to the programmer’s and the application’s needs)
```



5 Data Collector and SPINE Node Emulator

5.1 Data Collector application

The main goal of the SPINE “Data Collector” application is collect/share/publish sensors dataset. It implements the following functions:

- configure sensors nodes
- collect sensors data (raw data or feature sets)
- store sensors data

The data file formats used are: **ARFF** (Attribute Relation File Format), **CSV** (Comma Separated Values) and **TXT** (Text).

5.1.1 Dataset file syntax

Example:

“Sensor setting”:

- “cpu temperature” and “accelerometer” with “sampling time=100 ms”, “window=1” and “shift=1”

“Function setting”:

- feature “Raw Data” (on channel 1) on sensor “cpu temperature”
- features “Min” (on channels 1,2,3) and “Raw Data” (on channels 1,2,3) on sensor “accelerometer”



Node: 1

Node Info

Node ID (deprecated): 1
Physical Node ID: 1
Logical Node ID: null
OnBoard Sensors:
cpu temperature, ch1

Sensor Code cpu temperature

Sensor Setting

Sample Time 100

Time Type ms

Function Setting

Function Code Feature

window 1

Shift 1

Feature Setting

☒ Raw Data X axis

☐ Min NULL

☐ Max NULL


☐ Range NULL

☐ Mean NULL

OK Exit

Figure 5.1 : Data Collector – Sensor Setting window

Confirm

 Have you set correct parameters?

Sensor Setup {sensor = cpu temperature, timeScale = ms, samplingTime = 100}

Feature Function Setup {sensor = cpu temperature, window = 1, shift = 1}

Feature Function Activation {sensor = cpu temperature, feature = Raw Data, channels = ch1}

Yes No

Note: X axis = channel 1, Y axis = channel 2 and Z axis = channel 3



Node: 1

Node Info

Node ID (deprecated): 1
Physical Node ID: 1
Logical Node ID: null
OnBoard Sensors:
cpu temperature, ch1

Sensor Code accelerometer

Sensor Setting

Sample Time 100
Time Type ms

Function Setting

Function Code Feature
window 1
Shift 1

Feature Setting

☒ Raw Data Z Y X axis
☒ Min Z Y X axis
☐ Max NULL
☐ Range NULL
☐ Mean NULL

OK Exit

Confirm

? Have you set correct parameters?

Sensor Setup {sensor = accelerometer, timeScale = ms, samplingTime = 100}

Feature Function Setup {sensor = accelerometer, window = 1, shift = 1}

Feature Function Activation {sensor = accelerometer, feature = Min, channels = ch1, ch2, ch3, feature = Raw Data, channels = ch1, ch2, ch3}

Yes No

Sensor data are stored in dataset file.



ARFF dataset file:

```
% DB SPINE - DataCollector Application
%Data acquired from "cpu temperature" and "accelerometer".
%
%!Node phyID:2
%!Sensor_Setup: sensor=cpu temperature(4), timeScale=ms, samplingTime=100
%!Sensor_Setup: sensor=accelerometer(1), timeScale=ms, samplingTime=100
%!Feature_Setup: sensor=cpu temperature(4), window=1, shift =1
%!Feature_Setup: sensor=accelerometer(1), window=1, shift =1
%!Feature_Activation: sensor=cpu temperature(4), feature=Raw Data{1} channels=ch1[0]
%!Feature_Activation: sensor=accelerometer(1), feature=Min{3} channels=ch1 ch2 ch3[0 1 2],
feature=Raw Data{1} channels=ch1 ch2 ch3[0 1 2]
@relation spine
@attribute CLASS_LABEL {Section_one}
@attribute featureDataId numeric
@attribute featureId numeric
@attribute sensorCode_featureCode_chNum string
@attribute featureValue numeric
@data
Section_one,0,0,1_3_0,0
Section_one,0,0,1_3_1,0
Section_one,0,0,1_3_2,0
Section_one,0,1,1_1_0,0
Section_one,0,1,1_1_1,0
Section_one,0,1,1_1_2,0
Section_one,1,0,4_1_0,2986
Section_one,2,0,1_3_0,86
Section_one,2,0,1_3_1,65429
Section_one,2,0,1_3_2,1014
Section_one,2,1,1_1_0,86
Section_one,2,1,1_1_1,65429
Section_one,2,1,1_1_2,1014
Section_one,3,0,4_1_0,2988
Section_one,4,0,1_3_0,86
Section_one,4,0,1_3_1,65428
Section_one,4,0,1_3_2,1013
Section_one,4,1,1_1_0,86
Section_one,4,1,1_1_1,65428
Section_one,4,1,1_1_2,1013
```

CSV and TXT dataset file:

CSV and TXT data format are very simple for data but lacks meta-information; to supply additional information like what the input and output features are, simple time etc ... we store feature values in a CSV (comma separated values) and TXT (blank separated values) file and the meta information in a TXT file.

- comment file

```
% DB SPINE - DataCollector Application
%Data acquired from "cpu temperature" and "accelerometer".
%
%!Node phyID:2
%!Sensor_Setup: sensor=cpu temperature(4), timeScale=ms, samplingTime=100
%!Sensor_Setup: sensor=accelerometer(1), timeScale=ms, samplingTime=100
```




```
%!Feature_Setup: sensor=cpu temperature(4), window=1, shift =1
%!Feature_Setup: sensor=accelerometer(1), window=1, shift =1
%!Feature_Activation: sensor=cpu temperature(4), feature=Raw Data{1} channels=ch1[0]
%!Feature_Activation: sensor=accelerometer(1), feature=Min{3} channels=ch1 ch2 ch3[0 1 2],
feature=Raw Data{1} channels=ch1 ch2 ch3[0 1 2]
@relation spine
@attribute CLASS_LABEL {Section_one}
@attribute featureDataId numeric
@attribute featureId numeric
@attribute sensorCode_featureCode_chNum string
@attribute featureValue numeric
```

- CVS data file

```
CLASS_LABEL;featureDataId;featureId;sensorCode_featureCode_chNum;featureValue
Section_one;0;0;1_3_0;0
Section_one;0;0;1_3_1;0
Section_one;0;0;1_3_2;0
Section_one;0;1;1_1_0;0
Section_one;0;1;1_1_1;0
Section_one;0;1;1_1_2;0
Section_one;1;0;4_1_0;2987
Section_one;2;0;1_3_0;85
Section_one;2;0;1_3_1;65434
Section_one;2;0;1_3_2;1014
Section_one;2;1;1_1_0;85
Section_one;2;1;1_1_1;65434
Section_one;2;1;1_1_2;1014
Section_one;3;0;4_1_0;2986
Section_one;4;0;1_3_0;85
Section_one;4;0;1_3_1;65433
Section_one;4;0;1_3_2;1013
Section_one;4;1;1_1_0;85
Section_one;4;1;1_1_1;65433
Section_one;4;1;1_1_2;1013
```

- TXT data file

```
CLASS_LABEL featureDataId featureId sensorCode_featureCode_chNum featureValue
Section_one 0 0 1_3_0 0
Section_one 0 0 1_3_1 0
Section_one 0 0 1_3_2 0
Section_one 0 1 1_1_0 0
Section_one 0 1 1_1_1 0
Section_one 0 1 1_1_2 0
Section_one 1 0 4_1_0 2991
Section_one 2 0 1_3_0 84
Section_one 2 0 1_3_1 65434
Section_one 2 0 1_3_2 1014
Section_one 2 1 1_1_0 84
Section_one 2 1 1_1_1 65434
Section_one 2 1 1_1_2 1014
Section_one 3 0 4_1_0 2986
Section_one 4 0 1_3_0 84
Section_one 4 0 1_3_1 65434
Section_one 4 0 1_3_2 1014
Section_one 4 1 1_1_0 84
Section_one 4 1 1_1_1 65434
Section_one 4 1 1_1_2 1014
```



Our sensors dataset format focus on a two-dimensional array where rows are data points and columns are:

- column 1: **CLASS_LABEL** – data sensors label can be use to classify (E.g. outdoor_walking)
- column 2: **featureDataId** - packet data sequence number. Each packet data may store one or more features (E.g. each packet data from accelerometer contains the features Min and Raw Data)
- column 3: **featureId** – index of feature in the packet data
- column 4: **sensorCode_featureCode_chNum** – sensorCode and featureCode are stored in SPINESensorConstants and SPINEFunctionConstants.
- column 5: **feature value**



5.1.2 Data Collector GUI and functionality.

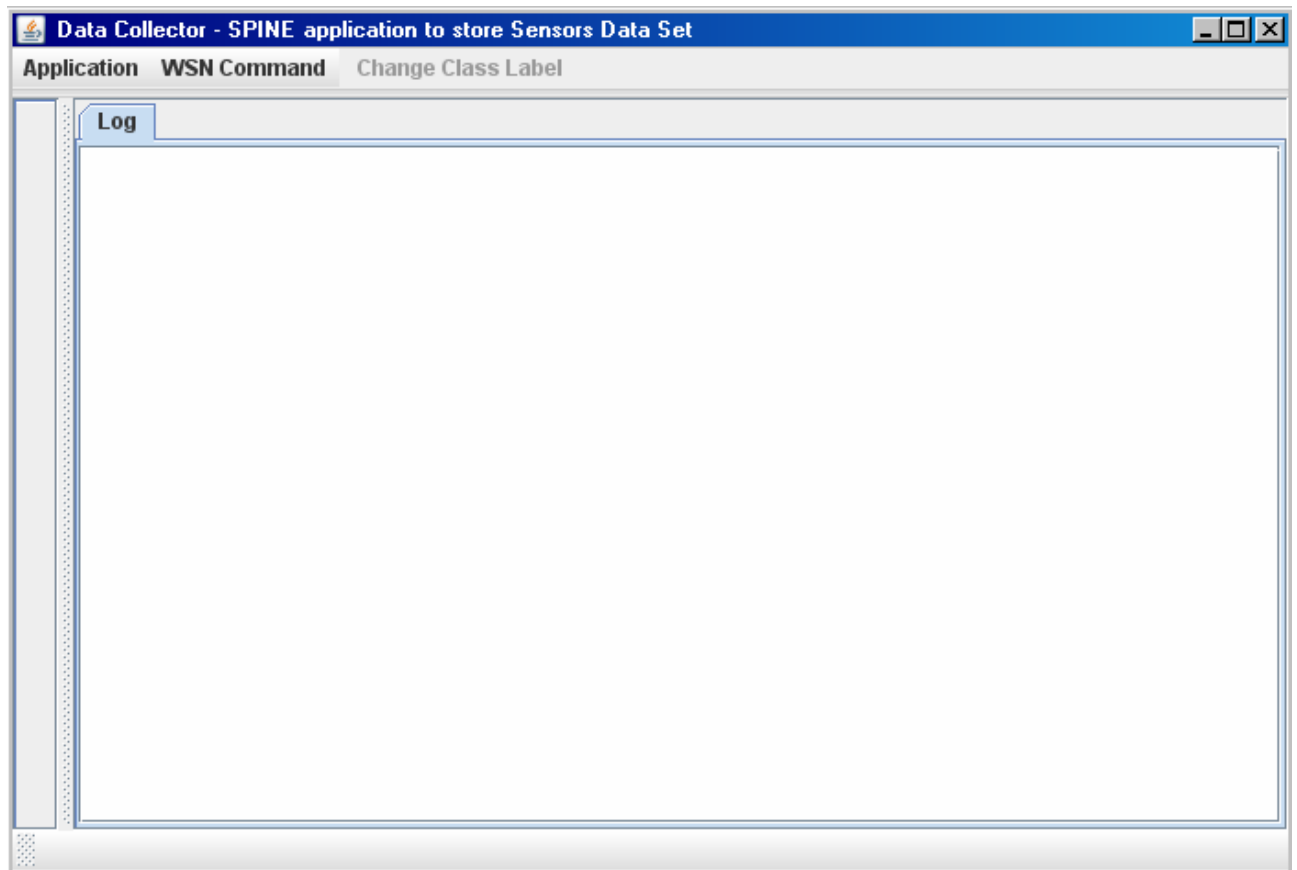
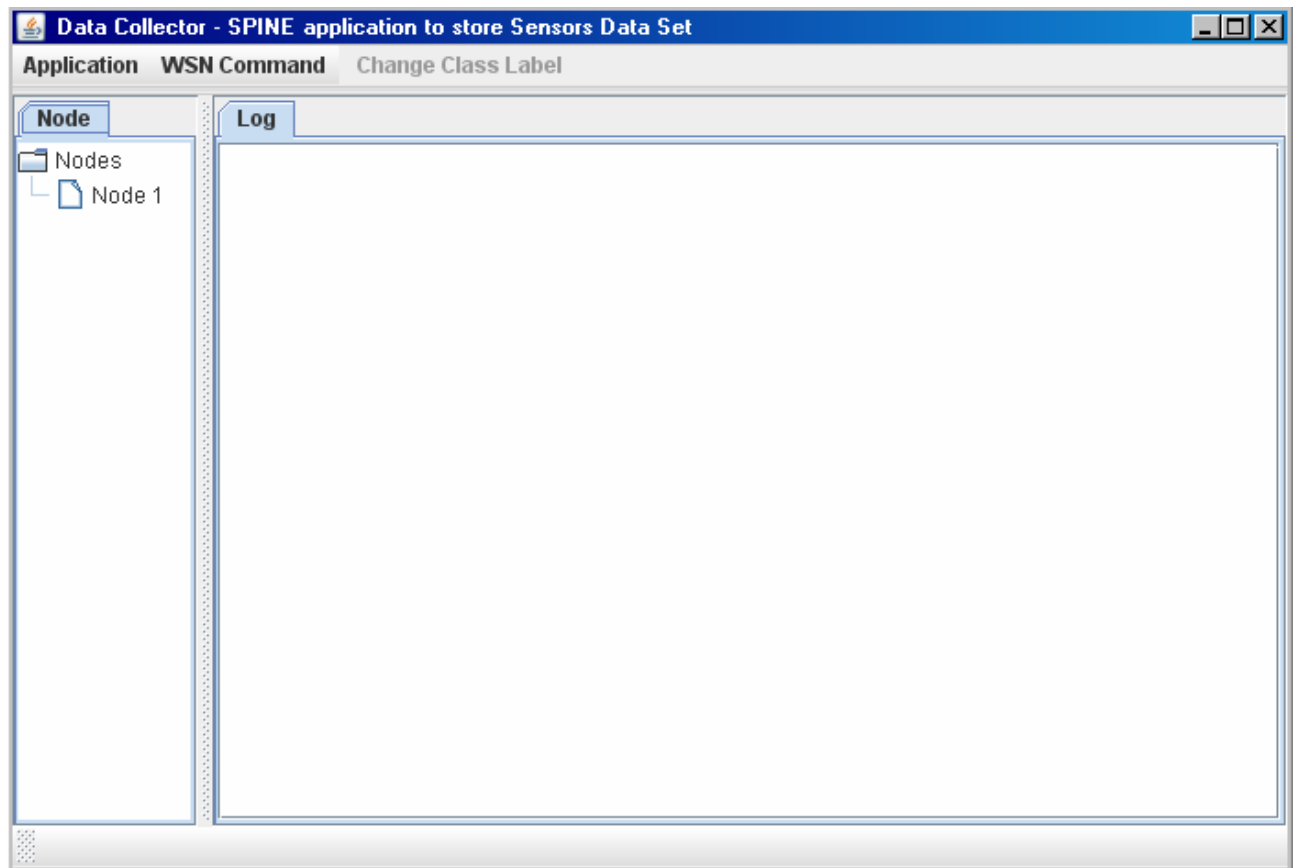
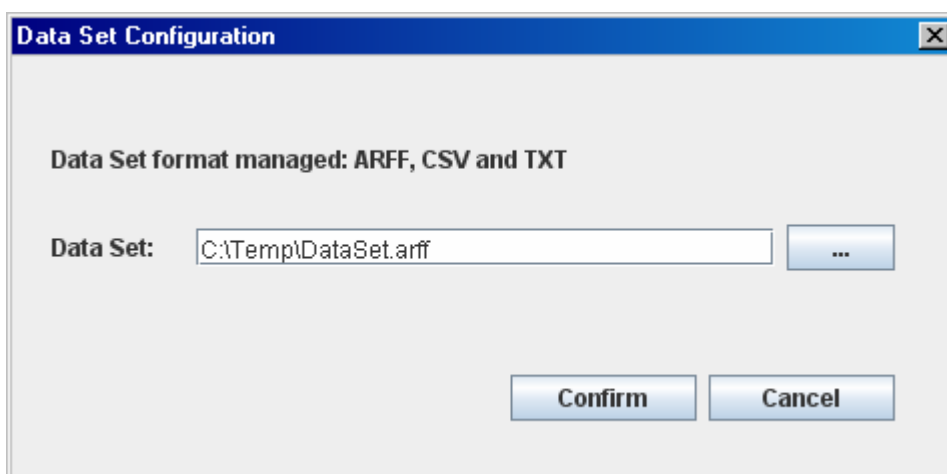
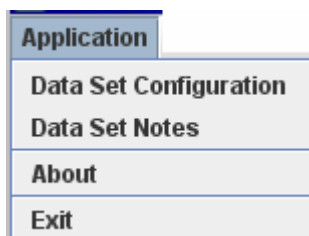


Figure 5.2 : The Data Collector GUI

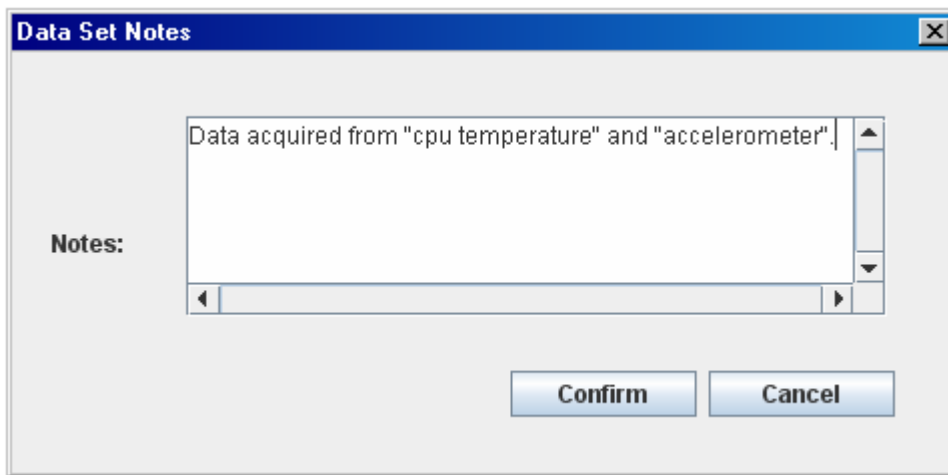
To discover the sensor network: **WSN Command -> Discovery** (E.g. Node 1).



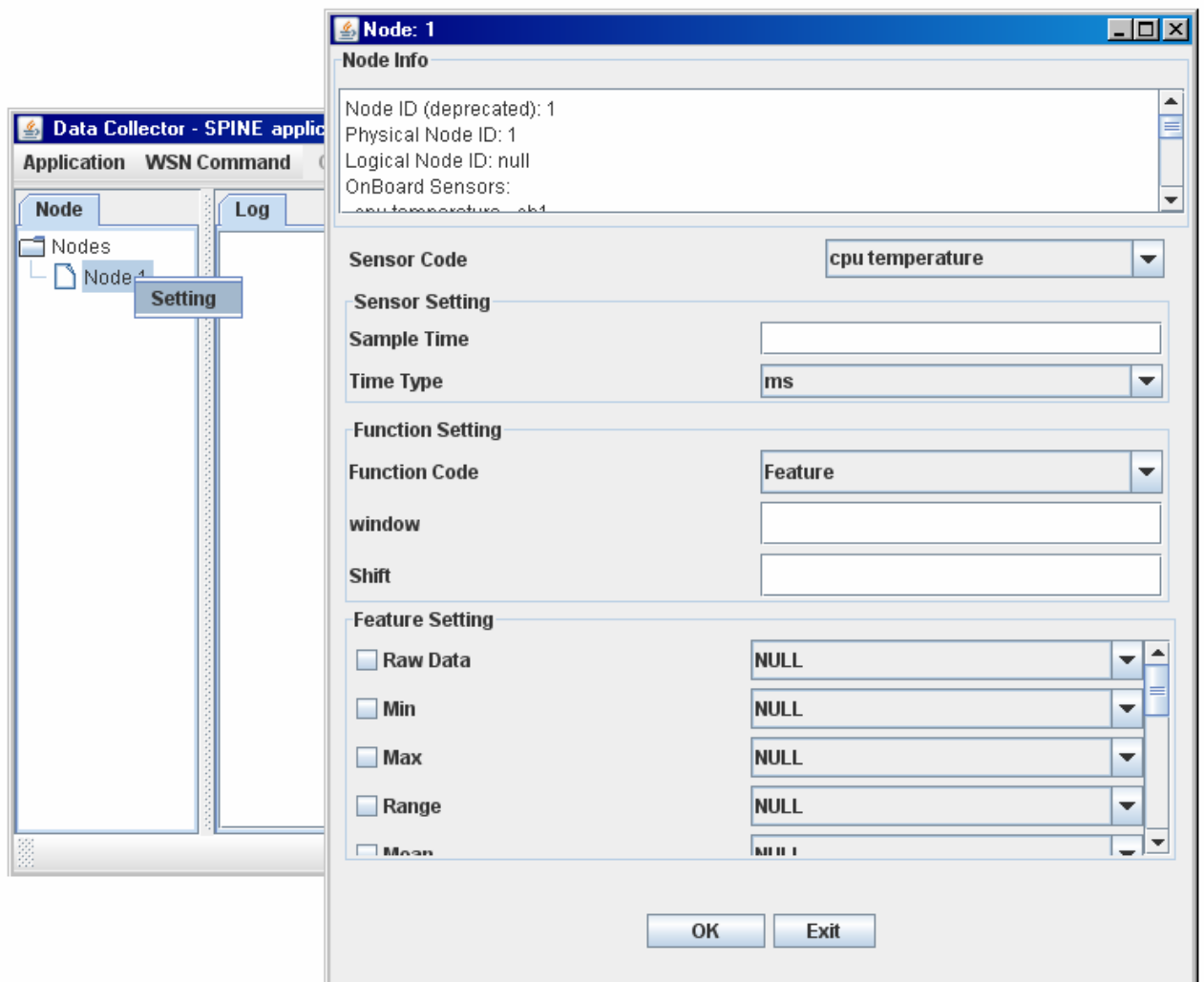
To choose the "dataset file" use the "Application" menu: **Application → Data Set Configuration.**



To set the notes, from the "Application" menu: **Application → Data Set Notes.**



To open the "Node Setting" panel click the right mouse button or double-click the node identifier in the "Nodes tree".





Example:

“Sensor setting”:

- “cpu temperature” with “sampling time=300 ms”, “window=20” and “shift=10”

“Function setting”:

- feature “Raw Data” (on channel 1)

Node: 1

Node Info

Node ID (deprecated): 1
Physical Node ID: 1
Logical Node ID: null
OnBoard Sensors:
cpu temperature_ch1

Sensor Code cpu temperature

Sensor Setting

Sample Time 300
Time Type ms

Function Setting

Function Code Feature
window 20
Shift 10

Feature Setting

☒ Raw Data X axis
☐ Min NULL
☐ Max NULL
☐ Range NULL
☐ Mean NULL

OK Exit

Confirm

? Have you set correct parameters?

Sensor Setup {sensor = cpu temperature, timeScale = ms, samplingTime = 300}
Feature Function Setup {sensor = cpu temperature, window = 20, shift = 10}
Feature Function Activation {sensor = cpu temperature, feature = Raw Data, channels = ch1}

Yes No



"Sensor setting":

- "accelerometer" with "sampling time=500 ms", "window=40" and "shift=20"

"Function setting":

- features "Min" (on channels 1,2,3) and "Raw Data" (on channels 1,2,3)

Node: 1

Node Info

Node ID (deprecated): 1
Physical Node ID: 1
Logical Node ID: null
OnBoard Sensors: ...temperature_ch1

Sensor Code accelerometer

Sensor Setting

Sample Time 500
Time Type ms

Function Setting

Function Code Feature
window 40
Shift 20

Feature Setting

☒ Raw Data Z Y X axis
☒ Min Z Y X axis
☐ Max NULL
☐ Range NULL
☐ Mean NULL

OK Exit

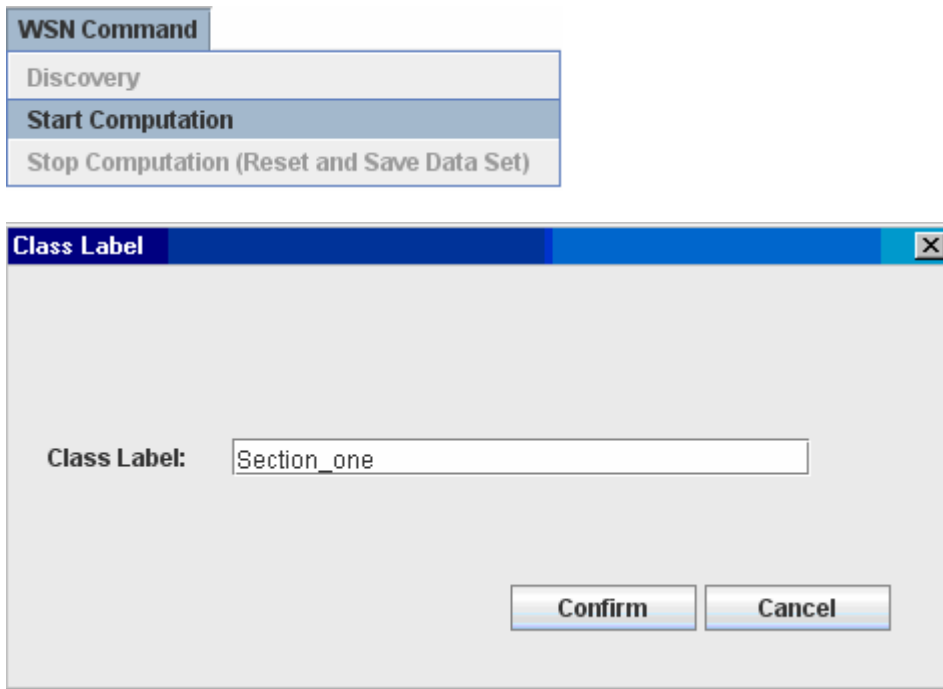
Confirm

? Have you set correct parameters?

Sensor Setup {sensor = accelerometer, timeScale = ms, samplingTime = 500}
Feature Function Setup {sensor = accelerometer, window = 40, shift = 20}
Feature Function Activation {sensor = accelerometer, feature = Min, channels = ch1, ch2, ch3, feature = Raw Data, channels = ch1, ch2, ch3}

Yes No

To start the computation and set the CLASS LABEL: **WSN Command -> Start Computation.**



The data received from the sensors are shown in the “Log” panel.

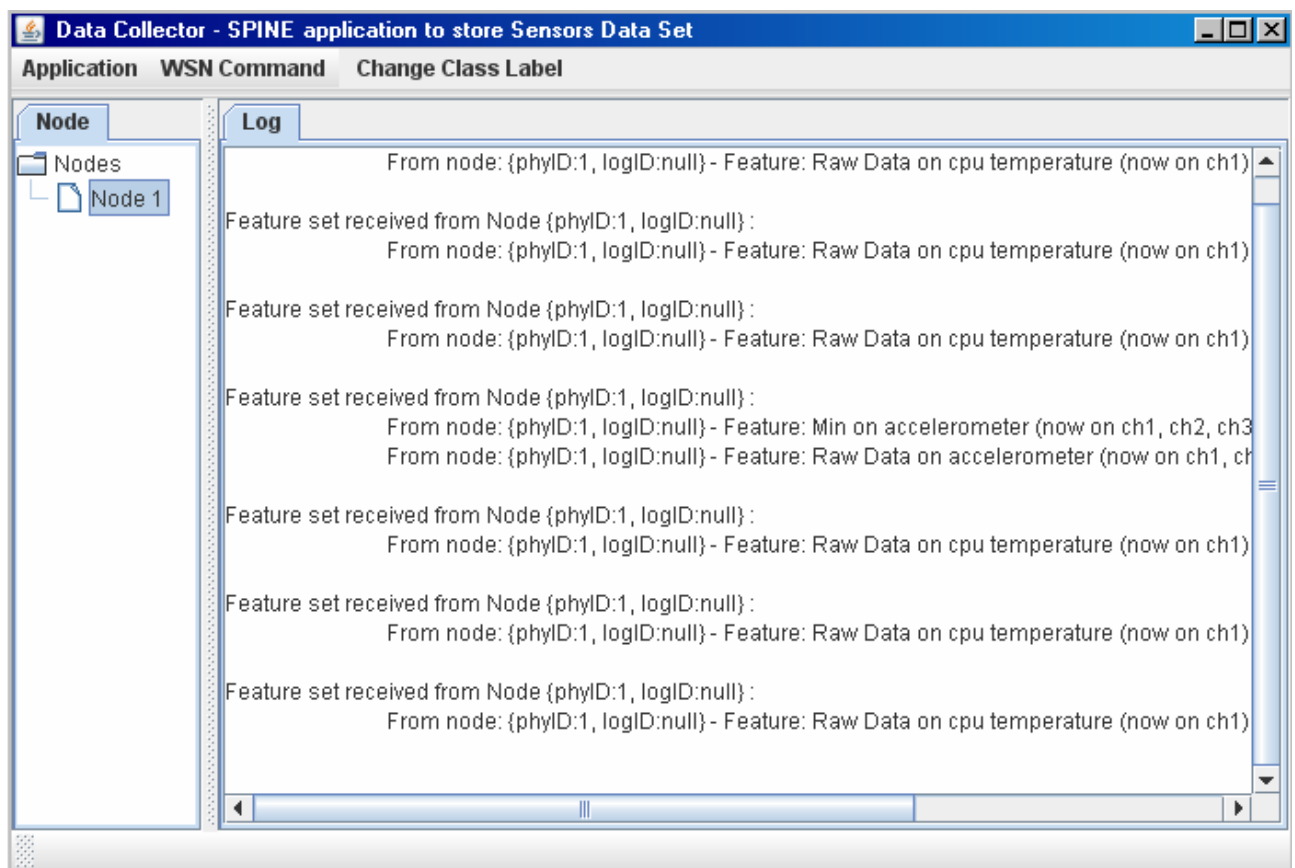
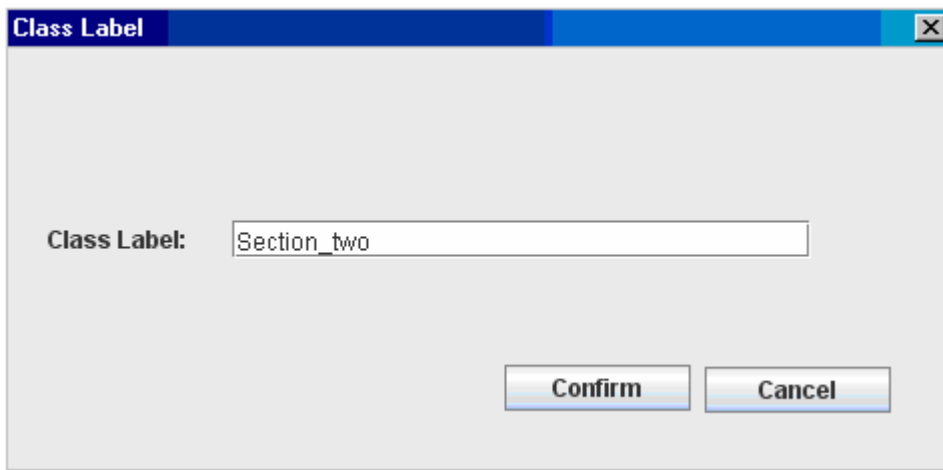


Figure 5.3 : Data Collector ‘Log’ panel

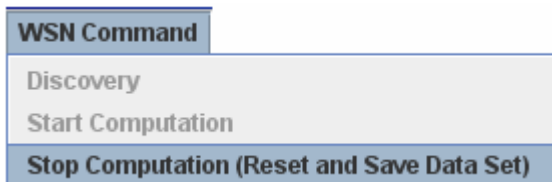
To change the CLASS LABEL select the “Change Class Label” Menu.



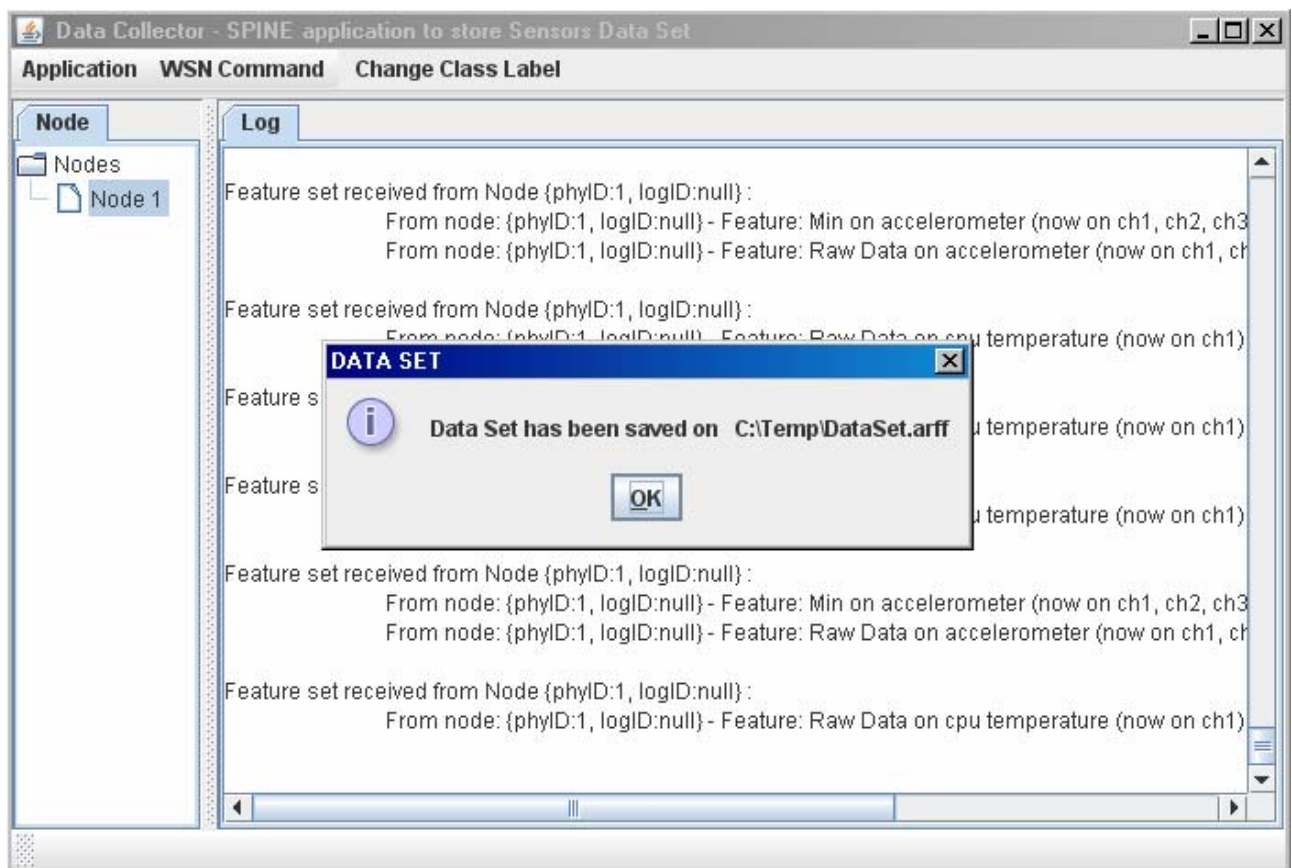
A dialog box titled "Class Label" with a close button (X) in the top right corner. It contains a label "Class Label:" followed by a text input field containing the text "Section_two". At the bottom, there are two buttons: "Confirm" and "Cancel".

Sensor data received during the "Change Class Label" action are discarded.

To "reset" the node and save its data set: **WSN Command -> Stop Computation (Reset and Save Data Set)**.



A menu titled "WSN Command" with three items: "Discovery", "Start Computation", and "Stop Computation (Reset and Save Data Set)". The "Stop Computation (Reset and Save Data Set)" item is highlighted with a blue background.



A screenshot of the "Data Collector - SPINE application to store Sensors Data Set" window. The window has a menu bar with "Application", "WSN Command", and "Change Class Label". On the left, there is a "Node" tree with "Nodes" and "Node 1". The main area is a "Log" window showing sensor data. A "DATA SET" dialog box is overlaid on the log, displaying an information icon and the message "Data Set has been saved on C:\Temp\DataSet.arff" with an "OK" button.



Future development

- Store data set in an SQL Database or use a temporary file to store feature data during collection (Current Data Collector version runs only with a small data set).

5.2 SPINE Node Emulator application

SPINE Node Emulator uses collected data (Data Collector output) to emulate SPINE nodes (each "Node Emulator" instance is a "virtual sensor node").

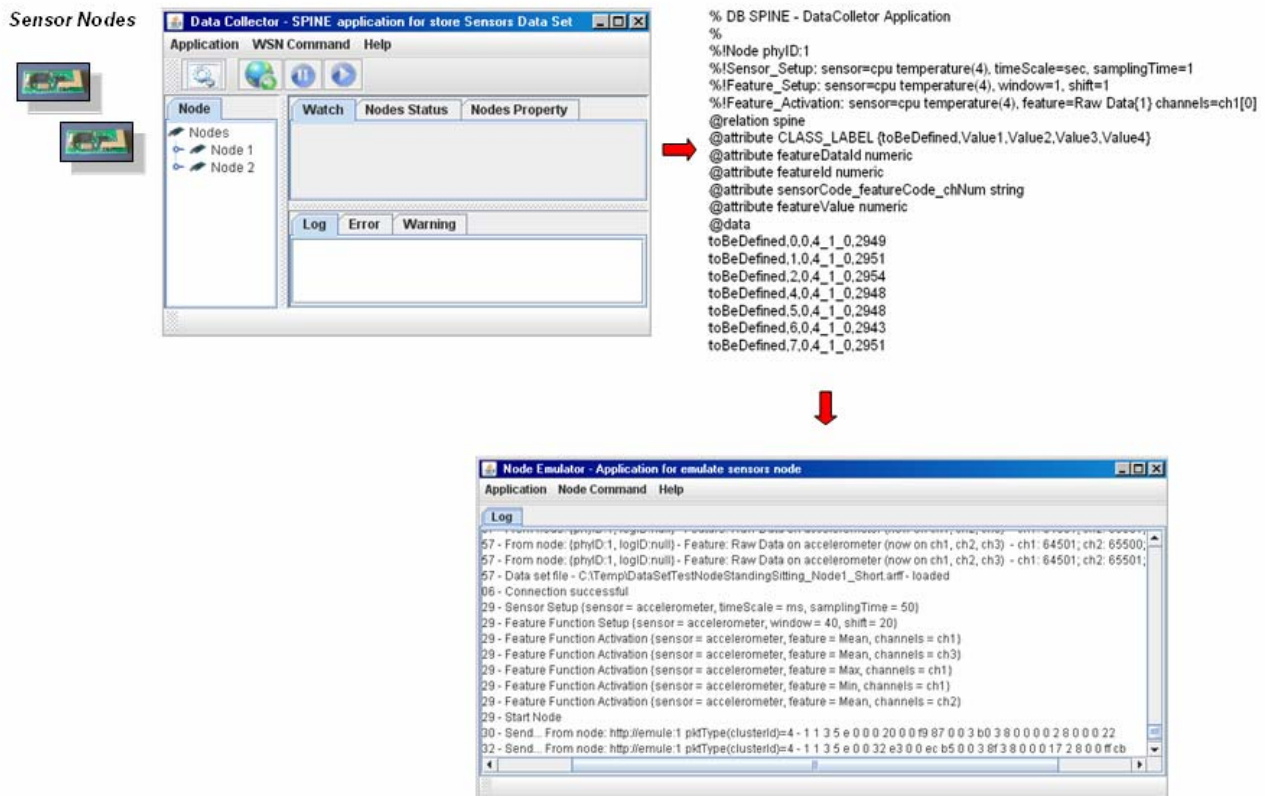


Figure 5.4 : Relation between Data Collector and SPINE Node Emulator



5.2.1 SPINE Node Emulator GUI and functionality.

The configuration properties must be configured (\resources\configuration.properties) before executing "SPINE Node Emulator".

configuration.properties:

MOTECOM=4444	← NodeCoordinatorPort
url_prefix=http://emu:	← Nome Emulator message prefix
compute_feature=ON	← ON if Node Emulator is able to compute feature
#compute_feature=OFF	
#label_algorithm=ALLWITHFREQ	
#label_algorithm=MOREFREQ	
label_algorithm=LAST	← Algorithm to calculate the label for "calculate feature data"

Each sensor data has a label (CLASS_LABEL attribute in dataset); if Node Emulator is able to calculate feature, labelAlgorithm is the algorithm to calculate the label for "calculate feature data":

- o "ALLWITHFREQ" - return all labels in the "window" with occurrence number
- o "MOREFREQ" return the label with highest occurrence in the "window"
- o "LAST"- return the last label (more recent in the "window").

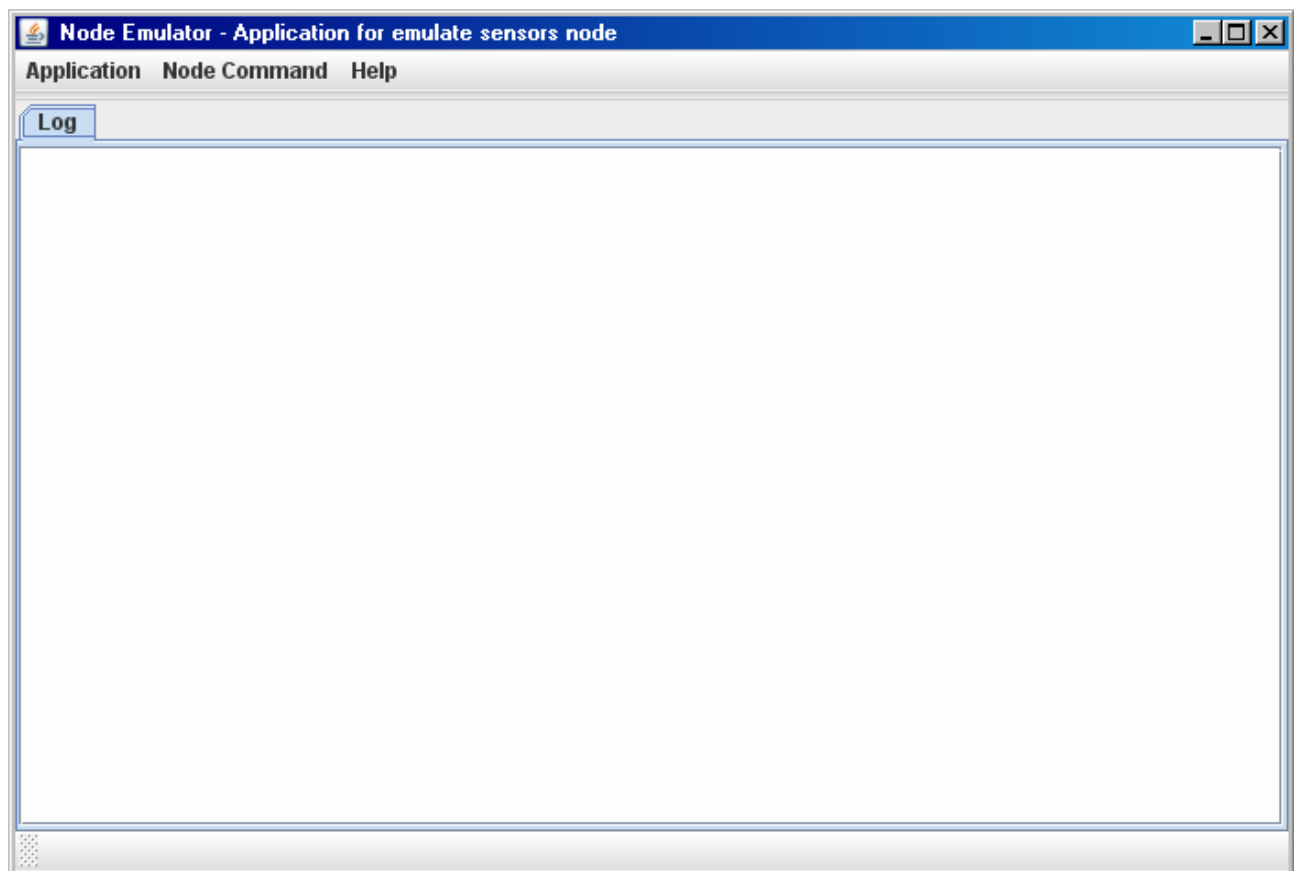
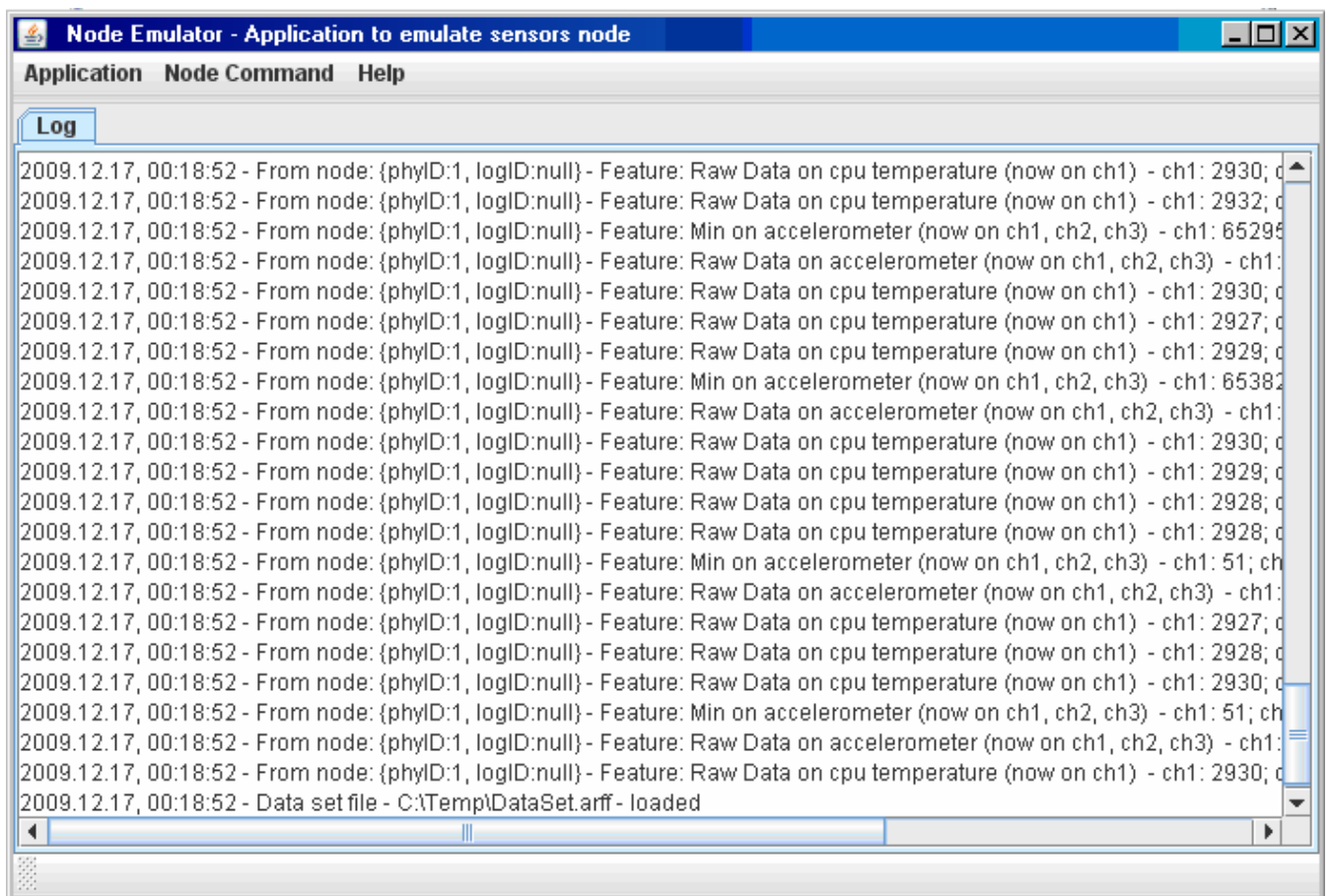
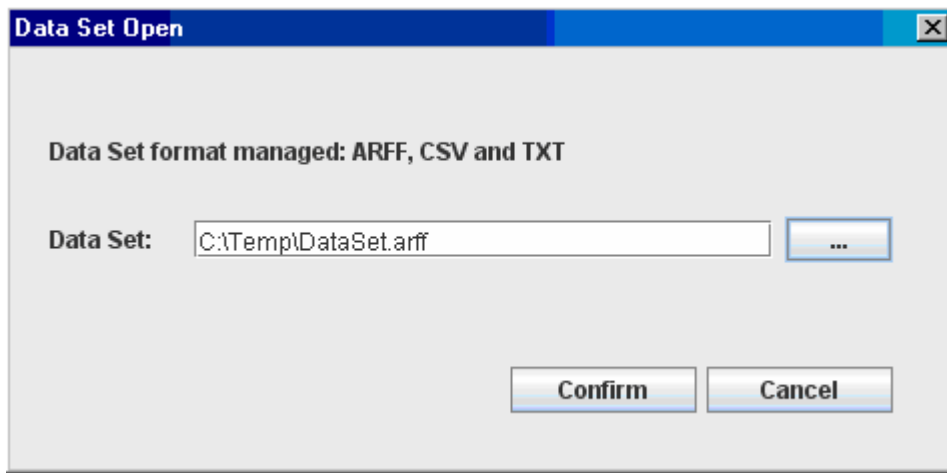


Figure 5.5 : Node Emulator GUI

To load dataset: **Application -> Data Set Open**





5.2.2 Virtual sensor node: functionality and sensors.

SPINE Node Emulator is able to calculate feature (parameter `compute_feature=ON` in `configuration.properties` file)

If `compute_feature=ON`:

- Virtual sensors: set of sensors with Feature_Activation "Raw Data"
- Virtual sensor's channels: feature "Raw Data" channels
- Supported functions: RAW_DATA, MAX, MIN, RANGE, MEAN, AMPLITUDE, RMS, VARIANCE, STDEV, MODE, MEDIAN, TOTENERGY
- Supported SPINE command: *SensorSetup, FunctionSetup e FunctionActivation*

If `compute_feature=OFF`:

- Virtual sensor: set of sensors with Feature_Activation
- Virtual sensor's channels: set of all features channels
- Supported functions: set of functions in Feature_Activation
- Discarded SPINE command: *SensorSetup, FunctionSetup e FunctionActivation*

Example:

```
%!Feature_Activation: sensor=cpu temperature(4), feature= Min{3} channels=ch1[0]
%!Feature_Activation: sensor=accelerometer(1), feature=Min{3} channels=ch1 ch2 ch3[0 1 2], feature=Raw Data{1} channels=ch1 ch2 [0 1]
```

Case `compute_feature=ON`:

- Virtual sensor: accelerometer(1) with channels= ch1 ch2 [0 1]
- Supported functions: RAW_DATA, MAX, MIN, RANGE, MEAN, AMPLITUDE, RMS, VARIANCE, STDEV, MODE, MEDIAN, TOTENERGY
- Supported SPINE command: *SensorSetup, FunctionSetup e FunctionActivation*

Case `compute_feature=OFF`:

- Virtual sensor: cpu temperature(4) with channel=ch1[0], accelerometer(1) with channels= ch1 ch2 ch3[0 1 2]
- Supported functions: RAW_DATA, MIN
- Discarded SPINE command: *SensorSetup, FunctionSetup e FunctionActivation*

5.2.3 Use case: TestGUI for testing “virtual sensor node”

TestGUI is a SPINE application for testing SPINE sensor node.

Step 1. Configure TestGUI application to use “virtual sensor node”

app.properties:

MOTECOM=4444

PLATFORM=emu

MOTECOM parameter in TestGUI app.properties must be equal to MOTECOM parameter in Node Emulator configuration.properties

Step 2. Run TestGUI application

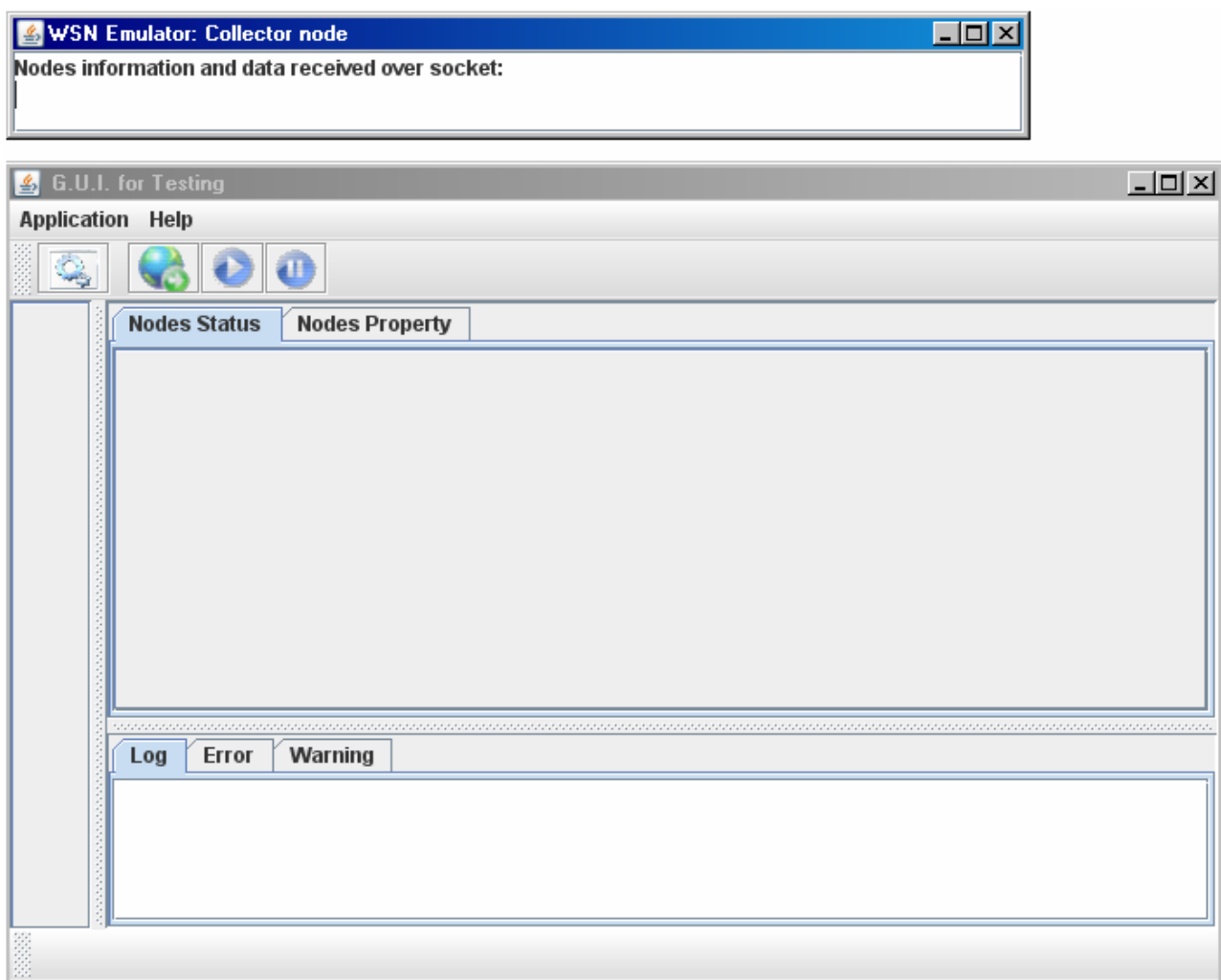


Figure 5.6 : TestGUI application and ‘WSN Emulator : Collector node’ window

if TestGUI uses “virtual sensor nodes” the window “WSN Emulator: Collector node” is created.

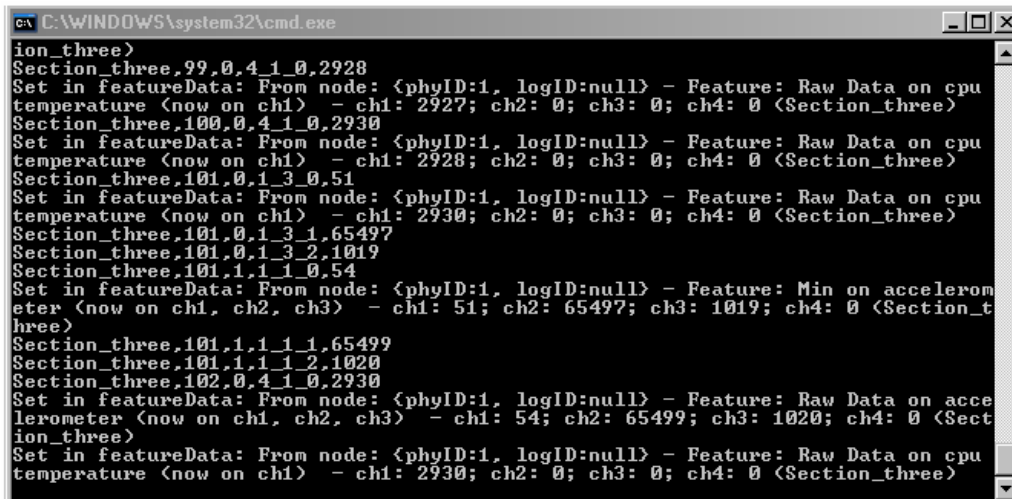
If TestGUI fails to load the images of the buttons then we have to add the “resource” folder of the project (in which the GIF files are included) as source folder in the Java Build Path section of eclipse project configuration.



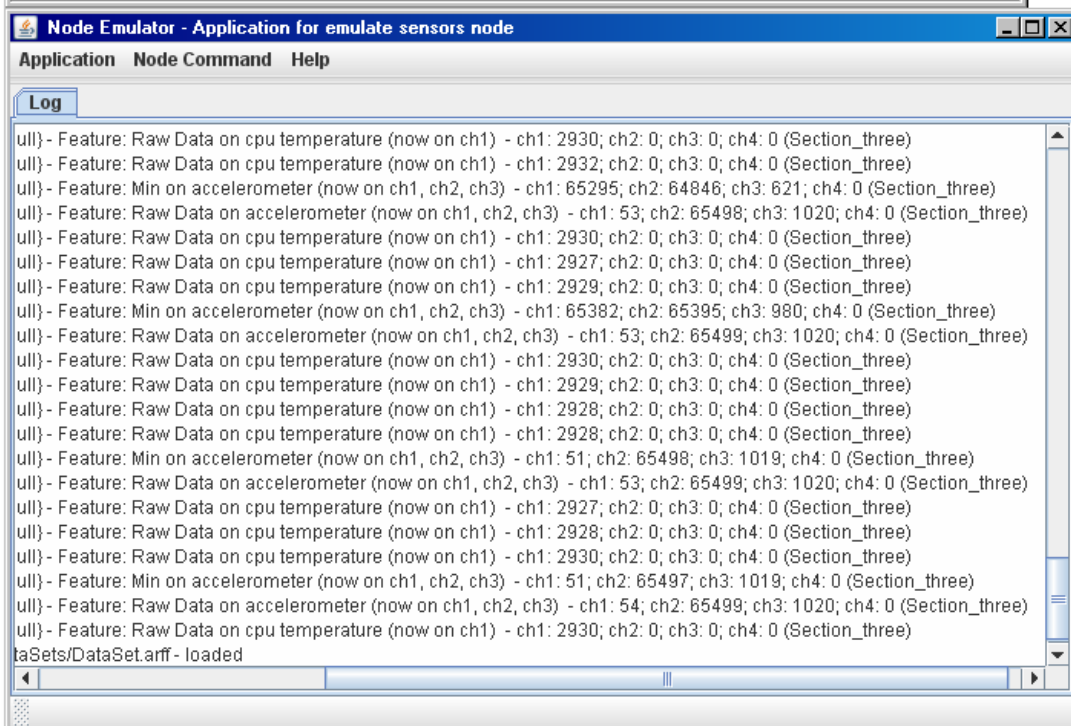
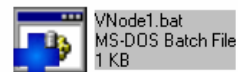
Step 3. Create “virtual sensor nodes”: run SPINE Node Emulator and load resources\dataSets\DataSet.arff or run script\VNode1.bat

script\VNode1.bat:

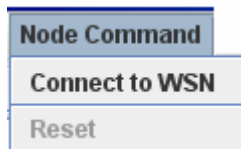
```
cd ..
java -cp ./bin;./lib/LocalNodeAdapter.jar;./lib/spine.jar; main.NodeEmulatorConsole
dataSetFile=./resources/dataSets/DataSet.arff connectToWSN=false computeFeature=ON
labelAlgorithm=ALLWITHFREQ
```



```
C:\WINDOWS\system32\cmd.exe
ion_three>
Section_three,99,0,4,1,0,2928
Set in featureData: From node: {phyID:1, logID:null} - Feature: Raw Data on cpu
temperature (now on ch1) - ch1: 2927; ch2: 0; ch3: 0; ch4: 0 (Section_three)
Section_three,100,0,4,1,0,2930
Set in featureData: From node: {phyID:1, logID:null} - Feature: Raw Data on cpu
temperature (now on ch1) - ch1: 2928; ch2: 0; ch3: 0; ch4: 0 (Section_three)
Section_three,101,0,1,3,0,51
Set in featureData: From node: {phyID:1, logID:null} - Feature: Raw Data on cpu
temperature (now on ch1) - ch1: 2930; ch2: 0; ch3: 0; ch4: 0 (Section_three)
Section_three,101,0,1,3,1,65497
Section_three,101,0,1,3,2,1019
Section_three,101,1,1,1,0,54
Set in featureData: From node: {phyID:1, logID:null} - Feature: Min on accelerom
eter (now on ch1, ch2, ch3) - ch1: 51; ch2: 65497; ch3: 1019; ch4: 0 (Section_t
hree)
Section_three,101,1,1,1,1,65499
Section_three,101,1,1,1,2,1020
Section_three,102,0,4,1,0,2930
Set in featureData: From node: {phyID:1, logID:null} - Feature: Raw Data on acce
lerometer (now on ch1, ch2, ch3) - ch1: 54; ch2: 65499; ch3: 1020; ch4: 0 (Sect
ion_three)
Set in featureData: From node: {phyID:1, logID:null} - Feature: Raw Data on cpu
temperature (now on ch1) - ch1: 2930; ch2: 0; ch3: 0; ch4: 0 (Section_three)
```



Step 4. Connect “virtual sensor node” to “Virtual WSN” (In a “Virtual WSN” there are one or more “virtual sensor node”): **Node Command** → **Connect to WSN (SPINE Node Emulator Menu)**

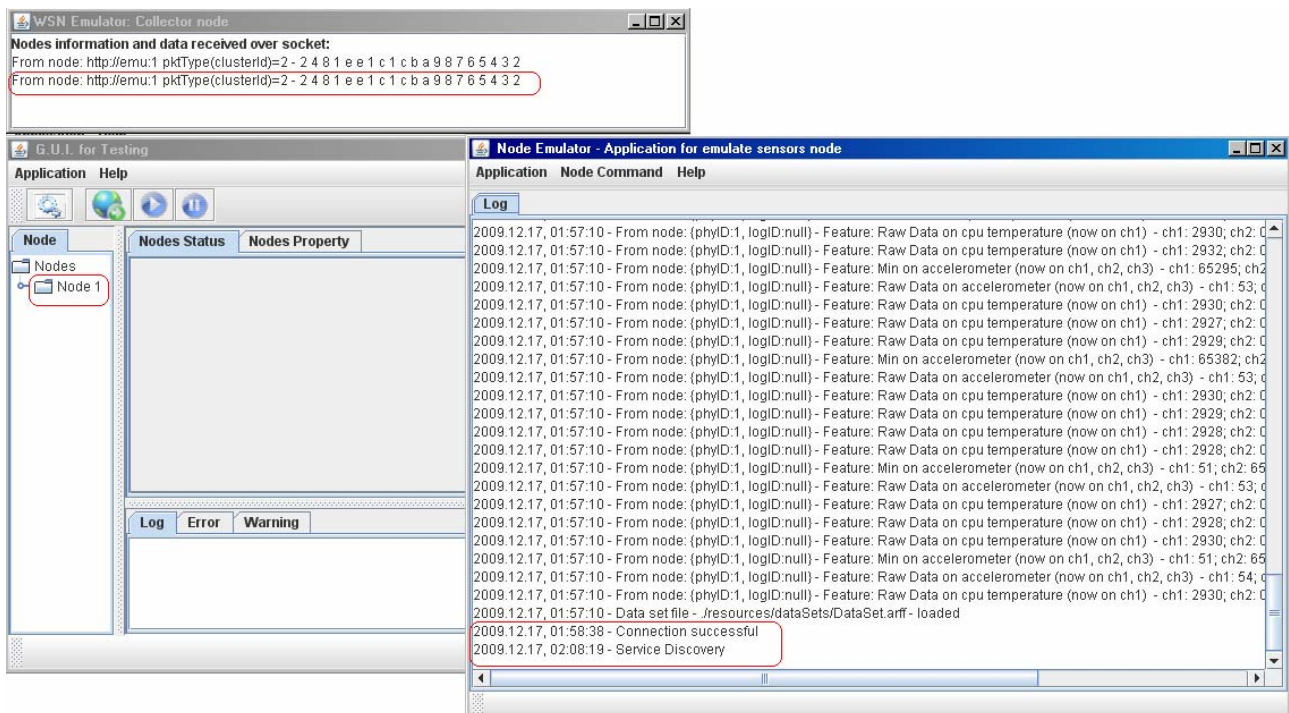


When connection is done “WSN Emulator: Connector Node” receive a advertisement message.



Figure 5.7 ‘WSN Emulator : Collector node’ : advertisement message.

Step 5. Discovery sensor node network: **Application -> Discovery (TestGUI Menu)**
(E.g. Node 1)



To open the “Nodes Property” panel in TestGUI, right click or double-click the node identifier in the “Nodes tree” panel or click “Nodes Property” tab.

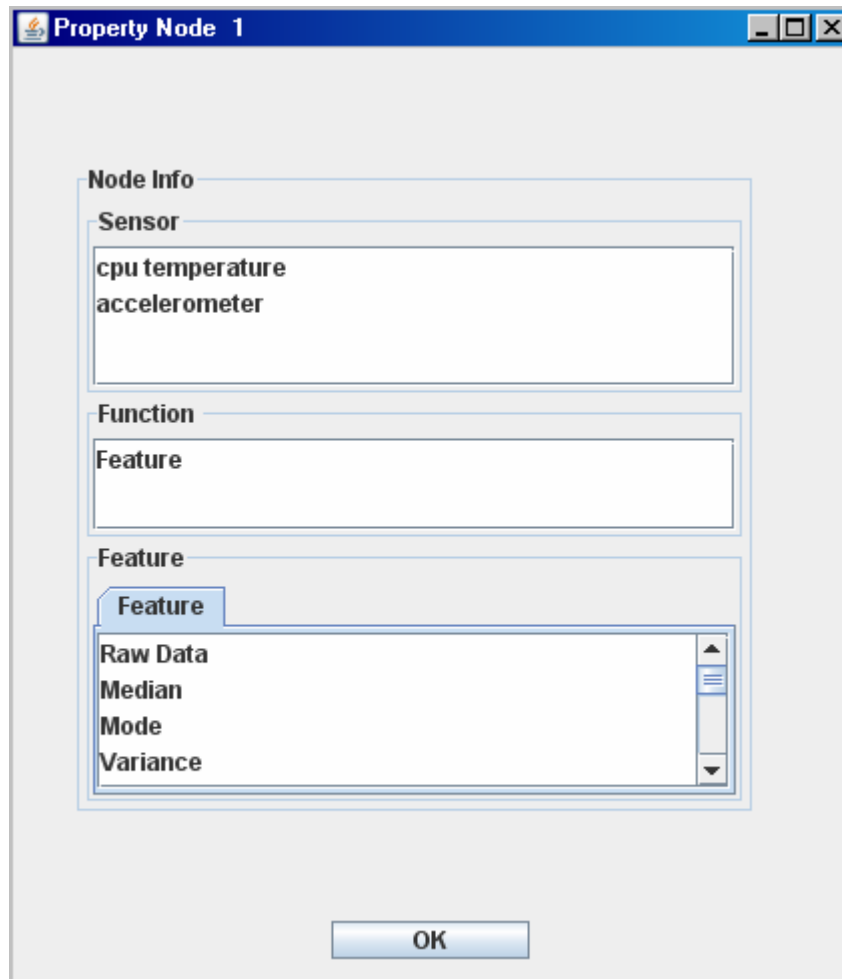


Figure 5.8 TestGUI : 'Node Property' panel .

Step 6. "Setup Sensor", "Setup Function" and "Activate Function" in "virtual sensor node"

To "Setup Sensor", "Setup Function" or "Activate Function" double-click on the action in the "Nodes-tree" panel (Virtual Node does not support "Disable Function")

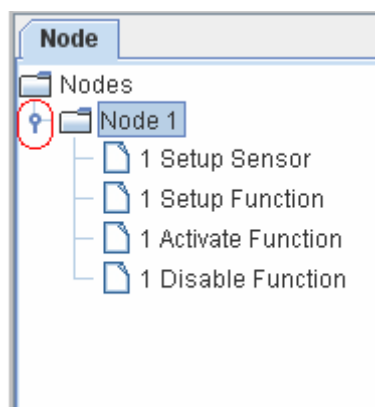


Figure 5.9 TestGUI : 'Nodes-tree' panel .

Example:

"cpu temperature" setting:

- "sampling time=3000 ms", "window=1" and "shift=1"
- feature "Raw Data" (on channel 1)

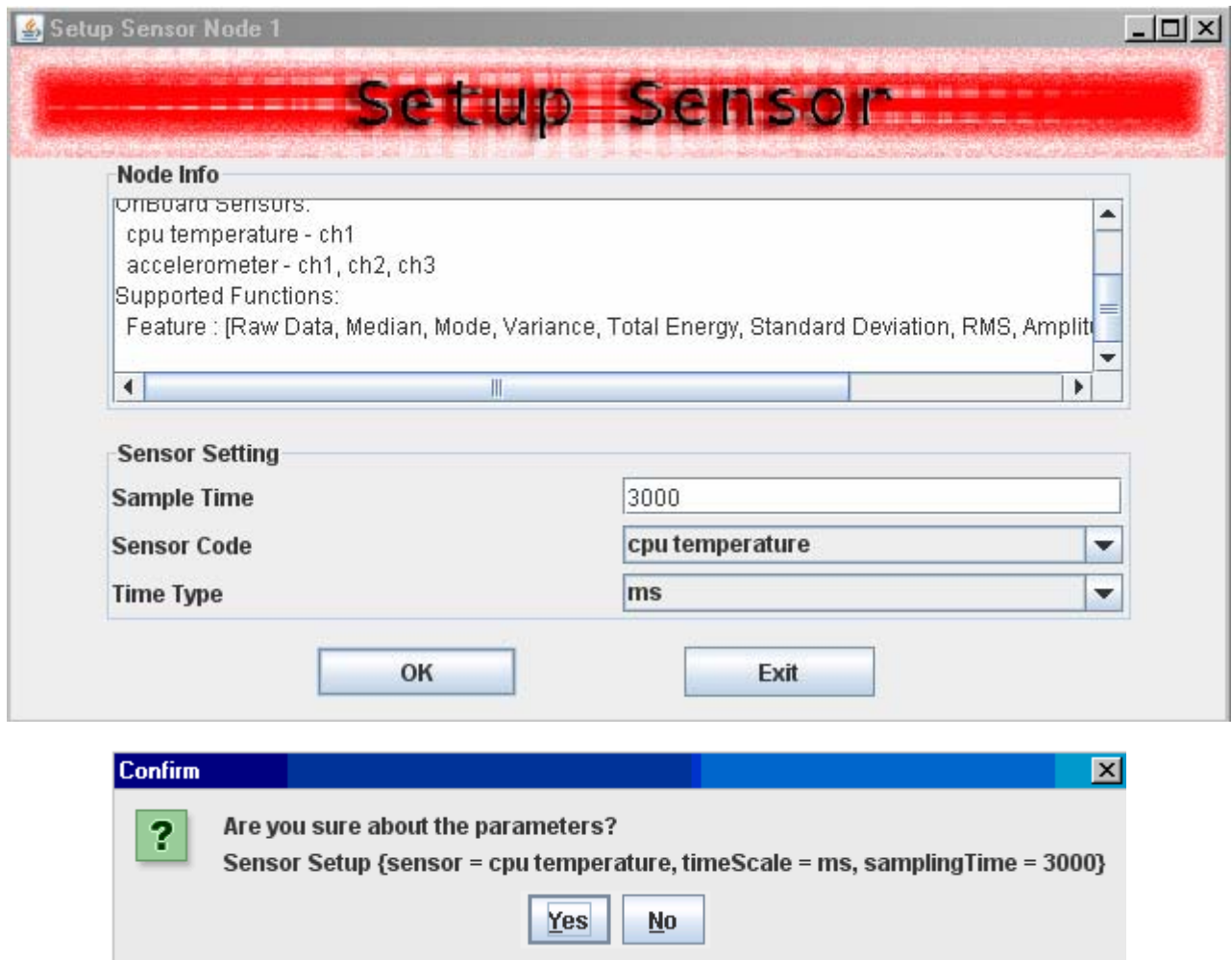


Figure 5.10 TestGUI : ‘Setup Sensor’ .

Each action is shown in SPINE Node Emulator “Log” panel.

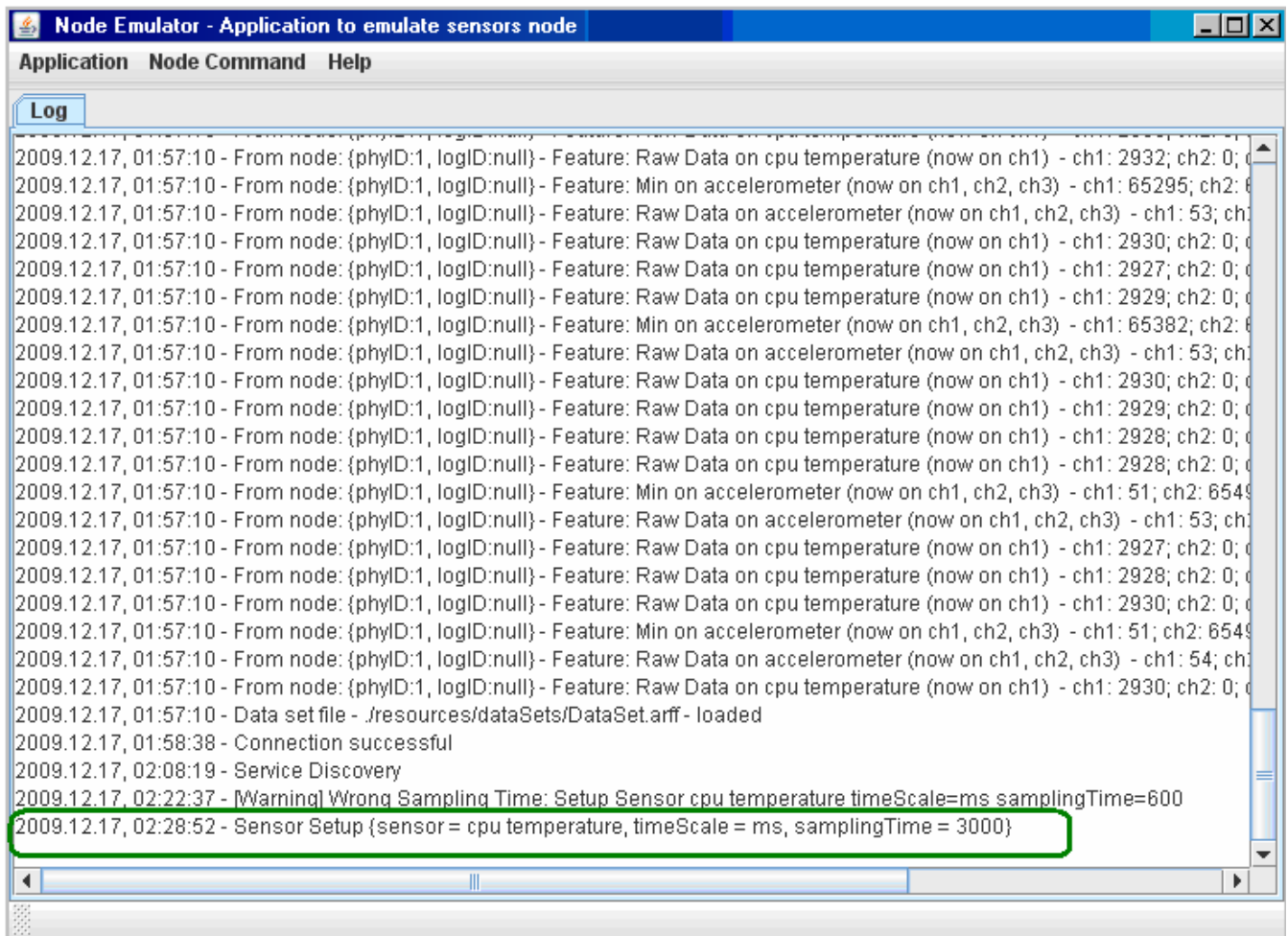


Figure 5.11 SPINE Node Emulator : ‘Setup Sensor’ feedback OK .

In setup sensor function, samplingTime must be a multiple of samplingTimeDataSet*shiftDataSet (E.g. “cpu temperature” samplingTimeDataSet=300ms and shiftDataSet=10: the samplingTime must be a multiple of 3000)

Setup Sensor Node 1

Setup Sensor

Node Info

OnBoard Sensors:
cpu temperature - ch1
accelerometer - ch1, ch2, ch3

Supported Functions:
Feature : [Raw Data, Median, Mode, Variance, Total Energy, Standard Deviation, RMS, Amplitude]

Sensor Setting

Sample Time: 600

Sensor Code: cpu temperature

Time Type: ms

OK Exit

Confirm

Are you sure about the parameters?
Sensor Setup {sensor = cpu temperature, timeScale = ms, samplingTime = 600}

Yes No

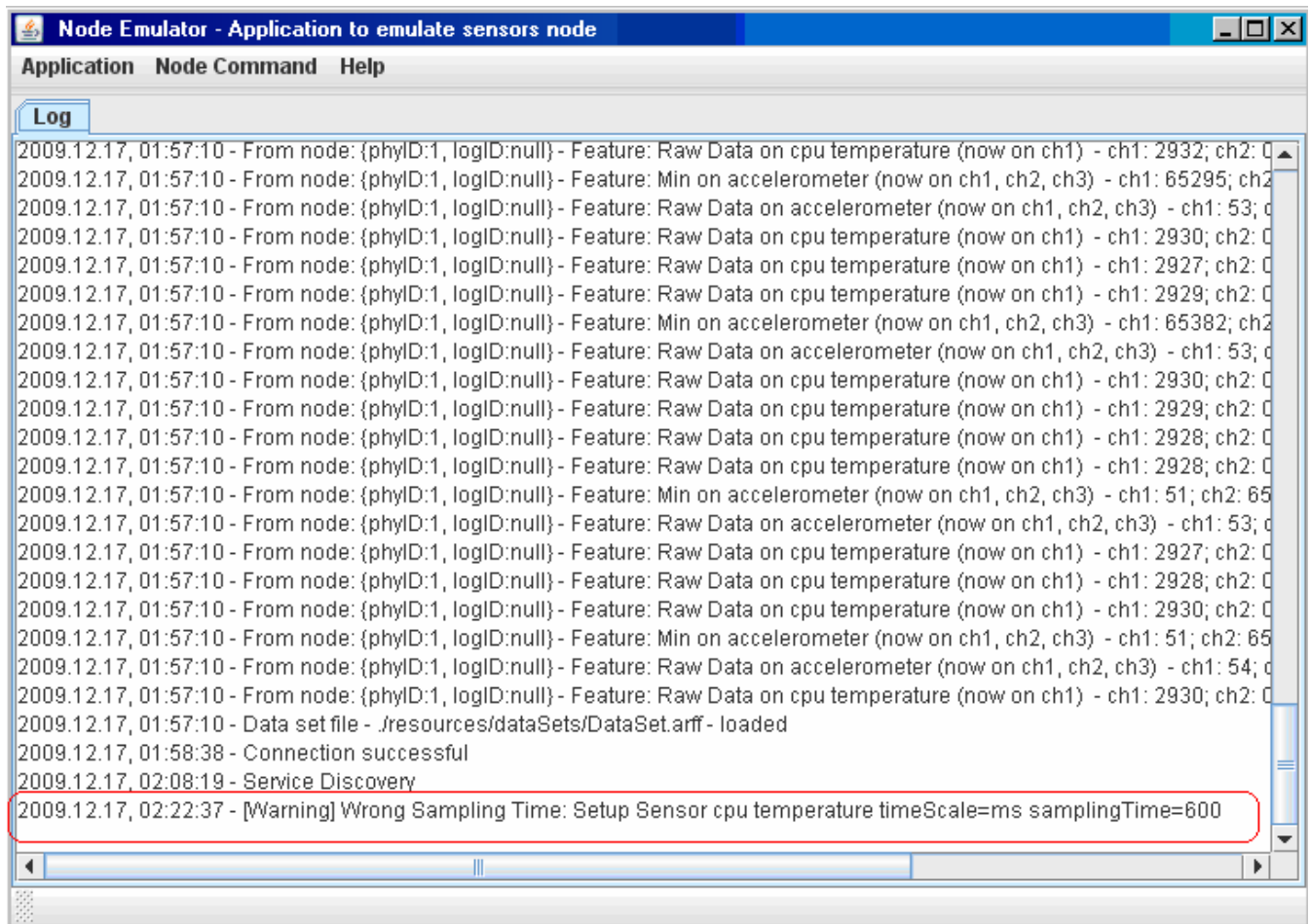


Figure 5.12 SPINE Node Emulator : ‘Setup Sensor’ feedback NOT OK .

SPINE Node Emulator supports only Function Code = Feature

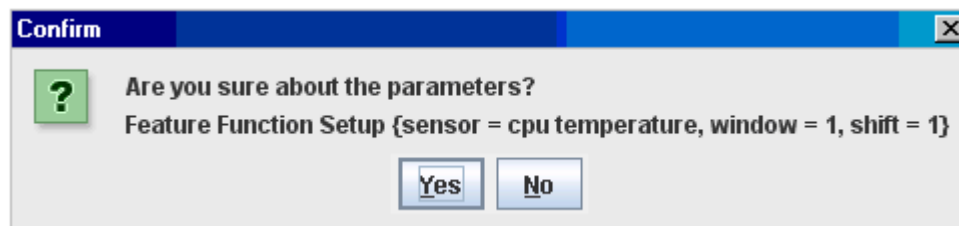
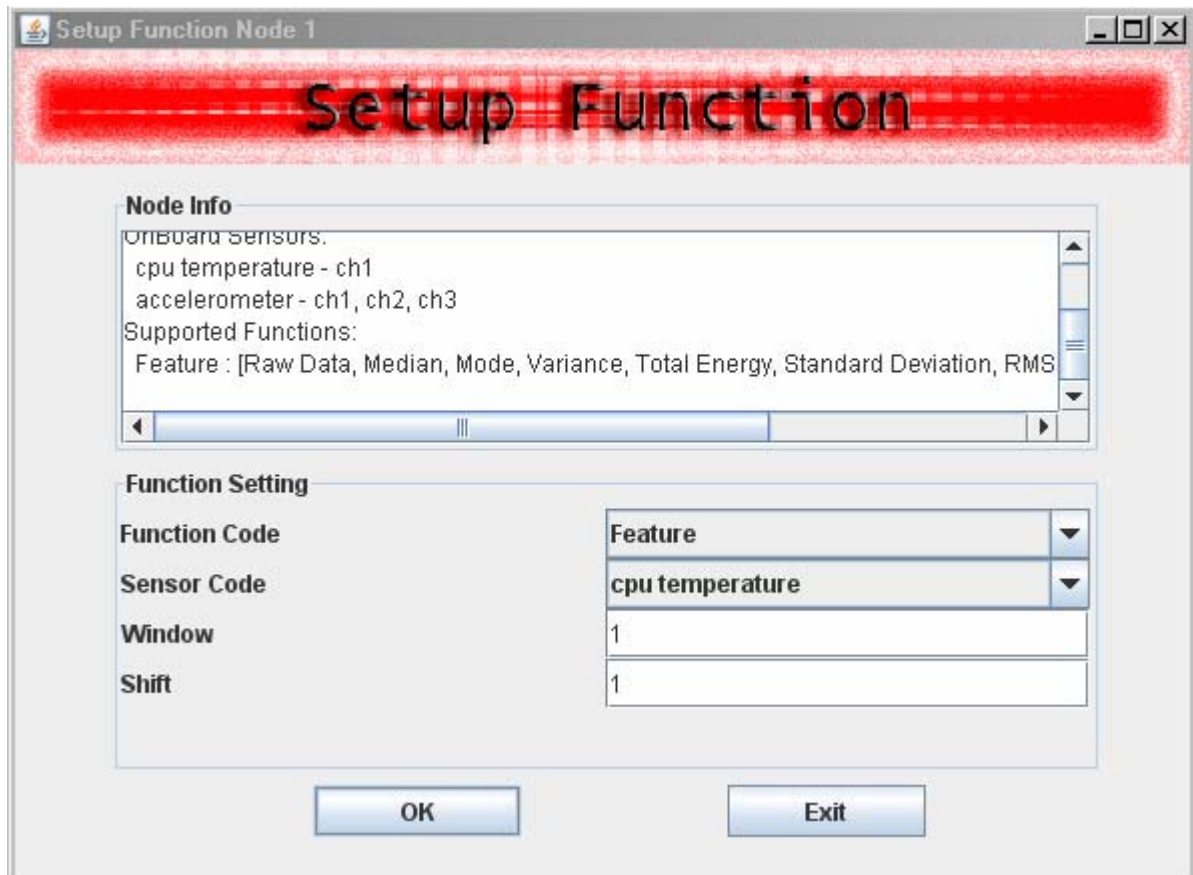


Figure 5.13 TestGUI : 'Setup Function' .

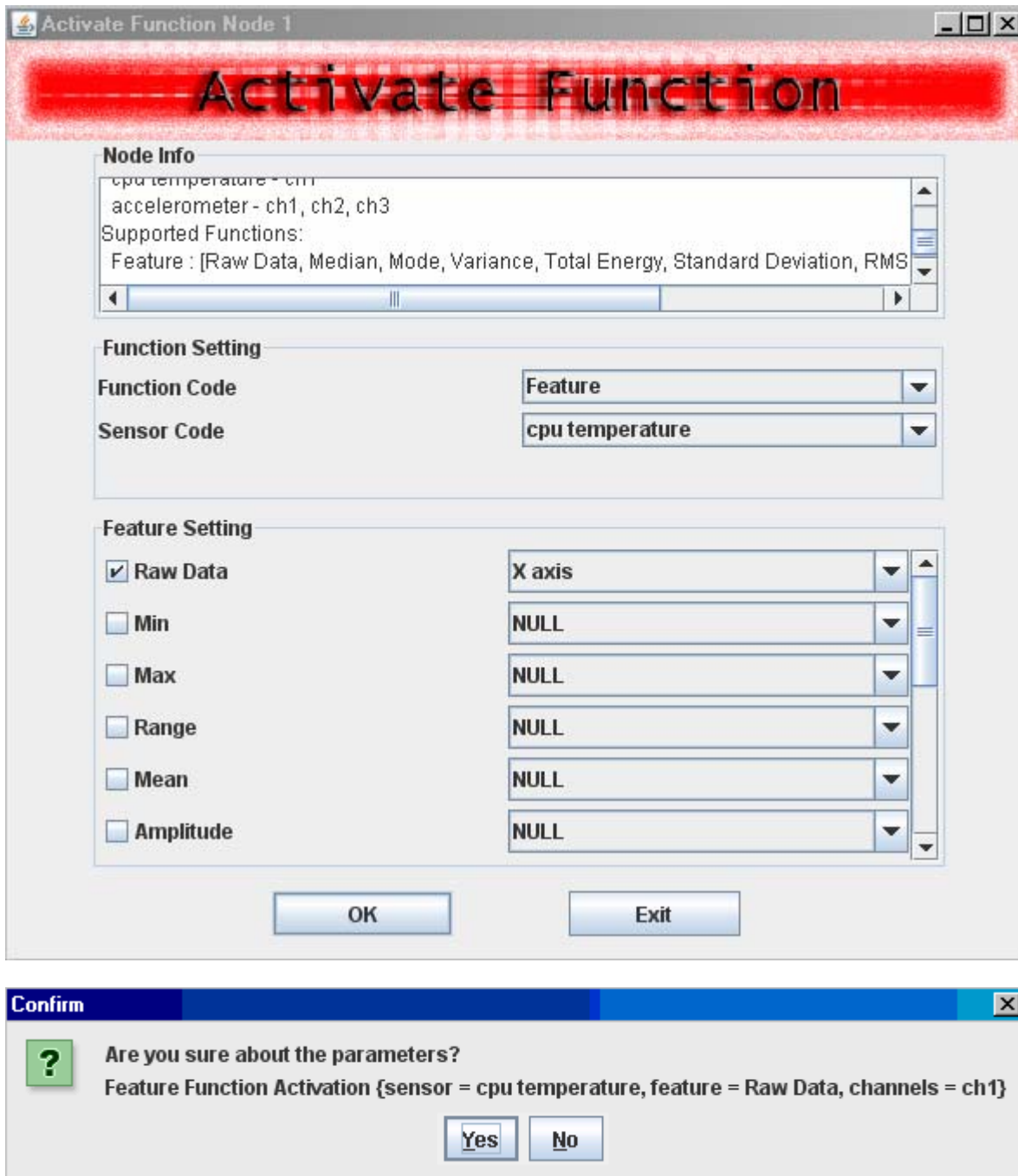


Figure 5.14 TestGUI : 'Activate Function' .

Example:

"accelerometer" setting:

- "sampling time=10000 ms", "window=2" and "shift=1"
- feature "Max" (on channel 1) and "Min" (on channel 1)

Setup Sensor Node 1

Setup Sensor

Node Info

Onboard sensors:
cpu temperature - ch1
accelerometer - ch1, ch2, ch3

Supported Functions:
Feature : [Raw Data, Median, Mode, Variance, Total Energy, Standard Deviation, RMS, Amplitude]

Sensor Setting

Sample Time: 10000

Sensor Code: accelerometer

Time Type: ms

OK Exit

Confirm

? Are you sure about the parameters?
Sensor Setup {sensor = accelerometer, timeScale = ms, samplingTime = 10000}

Yes No

Setup Function Node 1

Setup Function

Node Info

OnBoard Sensors:

cpu temperature - ch1

accelerometer - ch1, ch2, ch3

Supported Functions:

Feature : [Raw Data, Median, Mode, Variance, Total Energy, Standard Deviation, RMS]

Function Setting

Function Code

Sensor Code

Window

Shift

Feature

accelerometer

2

1

OK

Exit

Confirm

?

Are you sure about the parameters?

Feature Function Setup {sensor = accelerometer, window = 2, shift = 1}

Yes

No

Activate Function Node 1

Activate Function

Node Info

accelerometer - ch1, ch2, ch3

Supported Functions:

Feature : [Raw Data, Median, Mode, Variance, Total Energy, Standard Deviation, RMS]

Function Setting

Function Code: Feature

Sensor Code: accelerometer

Feature Setting

<input type="checkbox"/> Raw Data	NULL
<input checked="" type="checkbox"/> Min	X axis
<input checked="" type="checkbox"/> Max	X axis
<input type="checkbox"/> Range	NULL
<input type="checkbox"/> Mean	NULL
<input type="checkbox"/> Amplitude	NULL

Confirm

Are you sure about the parameters?

Feature Function Activation {sensor = accelerometer, feature = Max, channels = ch1, feature = Min, channels = ch1}

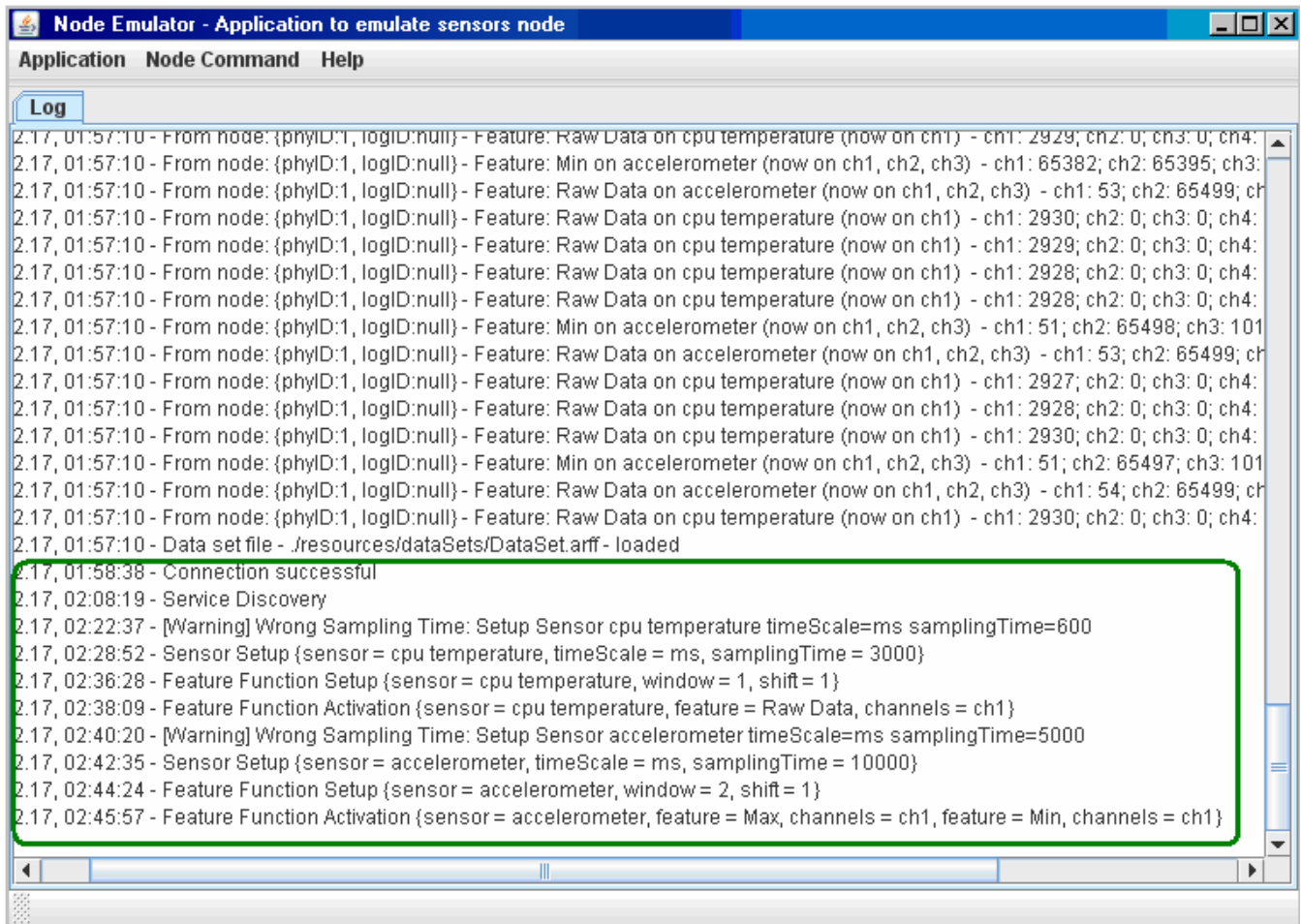
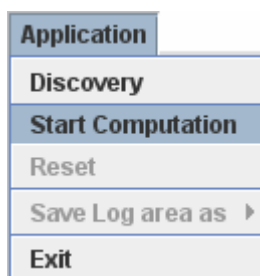
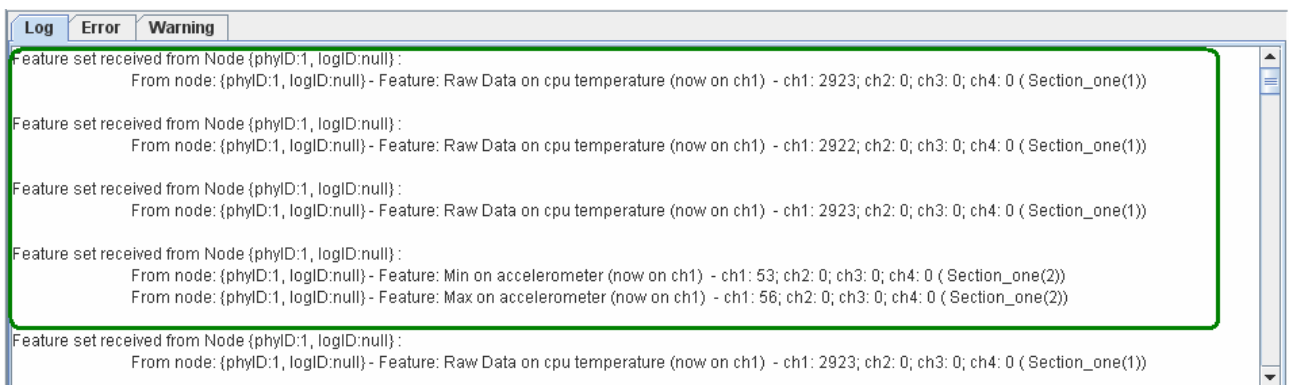
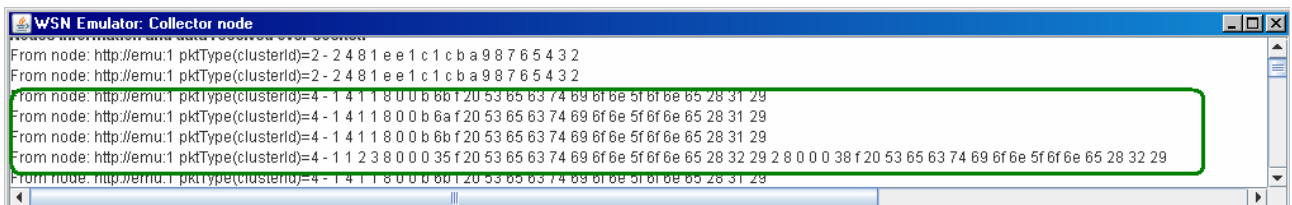
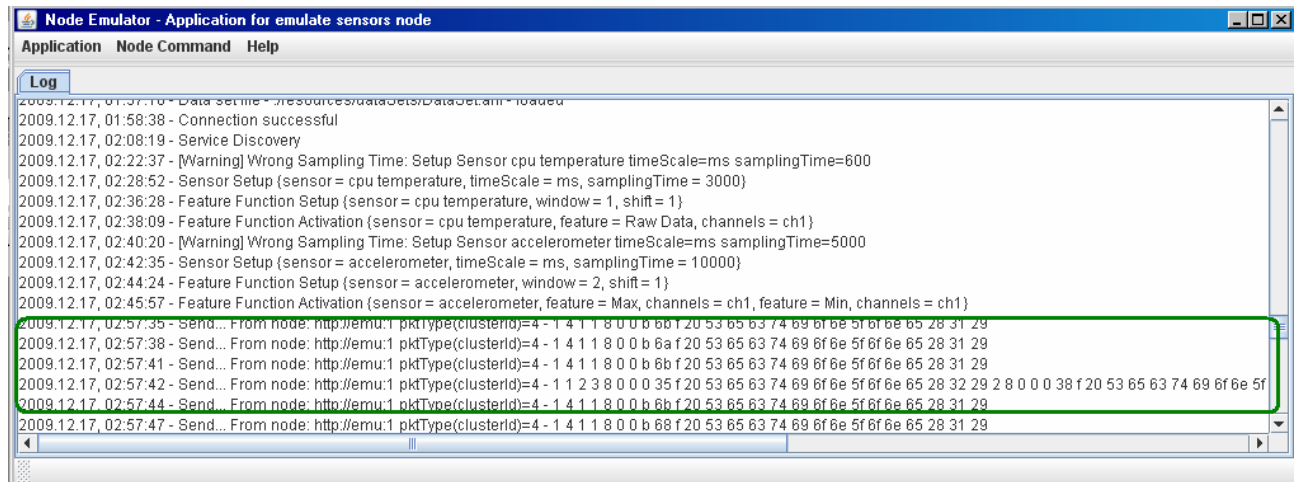


Figure 5.15 SPINE Node Emulator : ‘Setup Sensor’, ‘Setup Function ‘ and ‘Activate function’ feedback .

Step 7. Start computation: **Application -> Start Computation (TestGUI Menu)**



SPINE Node Emulator “Log” panel, “WSN Emulator: Collector node” window and TestGUI “Log” panel show emu message between “virtual sensor node” and “Collector node”.



Step 8. Stop computation: Application -> Reset (TestGUI Menu)

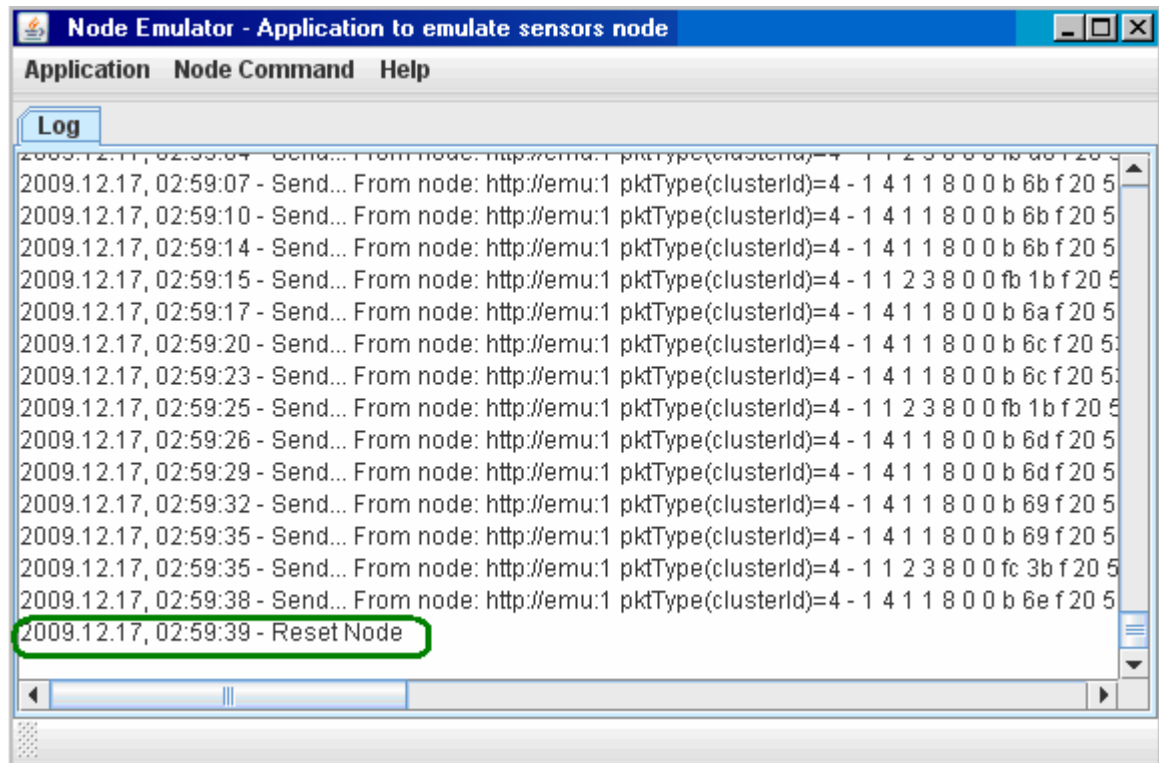


Figure 5.16 SPINE Node Emulator : 'Reset' feedback .

Future development

- Load data set from SQL Database or load feature data on demand during emulation
- Implementation of the Alarm and BufferRawData function.